

FOURTH EDITION



THE Quick Python Book

Naomi Ceder

Foreword by Luciano Ramalho

contents

<i>foreword</i>	<i>xviii</i>
<i>preface</i>	<i>xx</i>
<i>acknowledgments</i>	<i>xxii</i>
<i>about this book</i>	<i>xxiv</i>
<i>about the author</i>	<i>xxix</i>
<i>about the cover illustration</i>	<i>xxx</i>

PART 1 STARTING OUT.....1

1

About Python 3

1.1 Why should I use Python? 4

1.2 What Python does well 4

Python is easy to use 4 ▪ *Python is expressive* 5 ▪ *Python is readable* 5 ▪ *Python is complete: “Batteries included”* 6

Python has a rich ecosystem of third-party libraries 7

Python is cross-platform 7 ▪ *Python is free* 7

1.3 What Python is improving 8

Python is getting faster 8 ▪ *Python doesn’t enforce variable types at compile time* 8 ▪ *Python is improving mobile support* 9
Python is improving support for multiple processors 9

2

Getting started 11

- 2.1 Paradox of choice: Which Python? 12
 - Python versions 12 ▪ Sources of Python 12 ▪ Devices, platforms, and operating systems 13 ▪ Environments and tools 13*
- 2.2 Colaboratory: Jupyter notebooks in the cloud 13
 - Getting the source notebooks 14 ▪ Getting started with Colaboratory 14 ▪ Colaboratory's Python version 16*
- 2.3 Writing and running code in Colaboratory 17
 - Hello, World 18 ▪ Dealing with errors in Jupyter 18*
- 2.4 Using help and dir to explore Python 19
- 2.5 Using AI tools to write Python code 22
 - Benefits of AI tools 22 ▪ Negatives of using AI tools 22 ▪ AI options 23*

3

The quick Python overview 24

- 3.1 Python synopsis 25
- 3.2 Built-in data types 25
 - Numbers 25 ▪ Lists 27 ▪ Tuples 30 ▪ Strings 30 ▪ Dictionaries 31 ▪ Sets, frozensets 32 ▪ File objects 32*
- 3.3 Type hints in Python 34
- 3.4 Control flow structures 34
 - Boolean values and expressions 34 ▪ The if-elif-else statement 34 ▪ Structural pattern matching with match 35 ▪ The while loop 35 ▪ The for loop 36 ▪ Function definition 36 ▪ Exceptions 37 ▪ Context handling using the with keyword 38*
- 3.5 Module creation 38
- 3.6 Object-oriented programming 40

PART 2 THE ESSENTIALS.....43

4

The absolute basics 45

- 4.1 Indentation and block structuring 45
- 4.2 Differentiating comments 47
- 4.3 Variables and assignments 47

4.4	Optional type hints in Python	50
	<i>Why use type hints?</i>	51
	<i>Why not use type hints?</i>	51
	<i>Progressive typing</i>	51
4.5	Expressions	52
4.6	Strings	52
4.7	Numbers	53
	<i>Built-in numeric functions</i>	55
	<i>Advanced numeric functions</i>	55
	<i>Numeric computation</i>	55
	<i>Complex numbers</i>	55
	<i>Advanced complex-number functions</i>	56
4.8	The None value	57
4.9	Getting input from the user	58
4.10	Built-in operators	58
4.11	Basic Python style	58

5 *Lists, tuples, and sets* 61

5.1	Lists are like arrays	62
5.2	List indices	63
5.3	Modifying lists	65
5.4	Sorting lists	68
	<i>Custom sorting</i>	69
	<i>The sorted () function</i>	70
5.5	Other common list operations	71
	<i>List membership with the in operator</i>	71
	<i>List concatenation with the + operator</i>	72
	<i>List initialization with the * operator</i>	72
	<i>List minimum or maximum with min and max</i>	72
	<i>List search with index</i>	73
	<i>List matches with count</i>	73
	<i>Summary of list operations</i>	74
5.6	Nested lists and deep copies	75
5.7	Tuples	77
	<i>Tuple basics</i>	77
	<i>One-element tuples need a comma</i>	79
	<i>Packing and unpacking tuples</i>	80
	<i>Converting between lists and tuples</i>	82
5.8	Sets	82
	<i>Set operations</i>	82
	<i>Frozen sets</i>	83
5.9	Lab: Examining a list	84
	<i>Why solve it the old-fashioned way?</i>	84
	<i>Solving the problem with AI code generation</i>	85
	<i>Solutions and discussion</i>	86

6

Strings 89

- 6.1 Strings as sequences of characters 89
- 6.2 Basic string operations 90
- 6.3 Special characters and escape sequences 91
 - Basic escape sequences 91 ▪ Numeric (octal and hexadecimal) and Unicode escape sequences 92 ▪ Printing vs. evaluating strings with special characters 93*
- 6.4 String methods 94
 - The split and join string methods 94 ▪ Converting strings to numbers 96 ▪ Getting rid of extra whitespace 97 ▪ String searching 99 ▪ Modifying strings 101 ▪ Modifying strings with list manipulations 102 ▪ Useful methods and constants 103*
- 6.5 Converting objects to strings 105
- 6.6 Using the format method 106
 - The format method and positional parameters 106 ▪ The format method and named parameters 107 ▪ Format specifiers 107*
- 6.7 String interpolation with f-strings 108
- 6.8 Formatting strings with % 109
 - Using formatting sequences 110 ▪ Named parameters and formatting sequences 111*
- 6.9 Bytes 112
- 6.10 Preprocessing text 113
 - Solving the problem with AI-generated code 114 ▪ Solutions and discussion 114*

7

Dictionaries 118

- 7.1 What is a dictionary? 119
- 7.2 Other dictionary operations 121
- 7.3 Word counting 124
- 7.4 What can be used as a key? 125
- 7.5 Sparse matrices 126
- 7.6 Dictionaries as caches 127
- 7.7 Efficiency of dictionaries 128
- 7.8 Word counting 128
 - Solving the problem with AI-generated code 129 ▪ Solutions and discussion 129*

8 Control flow 133

- 8.1 The if-elif-else statement 133
- 8.2 Structural pattern matching with match 135
- 8.3 The while loop 135
- 8.4 The for loop 137
 - The range function 137*
- 8.5 Controlling range with starting and stepping values 138
- 8.6 The for loop and tuple unpacking 139
- 8.7 The enumerate function 139
- 8.8 The zip function 140
- 8.9 List, set, and dictionary comprehensions 140
 - Generator expressions 142*
- 8.10 Statements, blocks, and indentation 142
- 8.11 Boolean values and expressions 145
 - Most Python objects can be used as Booleans 146 ▪ Comparison and Boolean operators 146*
- 8.12 Writing a simple program to analyze a text file 148
- 8.13 Refactoring word_count 149
 - Solving the problem with AI-generated code 149 ▪ Solutions and discussion 149*

9 Functions 153

- 9.1 Basic function definitions 153
- 9.2 Function parameter options 155
 - Positional parameters 155 ▪ Passing arguments by parameter name 156 ▪ Variable numbers of arguments 157*
 - Mixing argument-passing techniques 158*
- 9.3 Mutable objects as arguments 159
 - Mutable objects as default values 160*
- 9.4 Local, nonlocal, and global variables 161
- 9.5 Assigning functions to variables 164
- 9.6 lambda expressions 164
- 9.7 Generator functions 165

- 9.8 Decorators 167
- 9.9 Useful functions 168
 - Solving the problem with AI-generated code 168 ▪ Solutions and discussion 168*

10 *Modules and scoping rules 176*

- 10.1 What is a module? 177
- 10.2 A first module 177
- 10.3 The import statement 181
- 10.4 The module search path 182
 - Where to place your own modules 183*
- 10.5 Private names in modules 184
- 10.6 Library and third-party modules 185
- 10.7 Python scoping rules and namespaces 186
 - The built-in namespace 190*
- 10.8 Creating a module 193
 - Solving the problem with AI-generated code 193 ▪ Solutions and discussion 194*

11 *Python programs 203*

- 11.1 Creating a very basic program 204
 - Starting a script from a command line 204 ▪ Command-line arguments 205 ▪ Executing code only as main script 206*
 - Redirecting the input and output of a script 206 ▪ The argparse module 208 ▪ Using the fileinput module 209*
- 11.2 Running scripts in different operating systems 211
 - Making a script directly executable on UNIX 211 ▪ Scripts on macOS 212 ▪ Script execution options in Windows 212*
- 11.3 Programs and modules 213
- 11.4 Distributing Python applications 219
 - Wheels packages 219 ▪ zipapp and pex 219 ▪ py2exe and py2app 220 ▪ Creating executable programs with freeze 220*
- 11.5 Creating a program 220
 - Solving the problem with AI-generated code 220 ▪ Solutions and discussion 221*

12 Using the filesystem 228

- 12.1 os and os.path vs. pathlib 229
- 12.2 Paths and pathnames 229
 - Absolute and relative paths 230 ▪ The current working directory 231 ▪ Accessing directories with pathlib 232*
 - Manipulating pathnames 232 ▪ Manipulating pathnames with pathlib 235 ▪ Useful constants and functions 236*
- 12.3 Getting information about files 237
 - Getting information about files with scandir 239*
- 12.4 More filesystem operations 239
 - More filesystem operations with pathlib 241*
- 12.5 Processing all files in a directory subtree 244
- 12.6 More file operations 245
 - Solving the problem with AI-generated code 245 ▪ Solutions and discussion 246*

13 Reading and writing files 256

- 13.1 Opening files and file objects 257
- 13.2 Closing files 257
- 13.3 Opening files in write or other modes 258
- 13.4 Functions to read and write text or binary data 258
 - Using binary mode 260*
- 13.5 Reading and writing with pathlib 261
- 13.6 Terminal input/output and redirection 262
- 13.7 Handling structured binary data with the struct module 265
- 13.8 Pickling objects to files 267
 - Reasons not to pickle 271*
- 13.9 Shelving objects 271
- 13.10 Final fixes to wc 273
 - Solving the problem with AI-generated code 274 ▪ Solutions and discussion 274*

14**Exceptions 281**

- 14.1 Introduction to exceptions 282
General philosophy of errors and exception handling 282
A more formal definition of exceptions 284 ▪ Handling different types of exceptions 284
- 14.2 Exceptions in Python 285
Types of Python exceptions 286 ▪ Raising exceptions 288
Catching and handling exceptions 289 ▪ Defining new exceptions 290 ▪ Exception groups 292 ▪ Debugging programs with the assert statement 292 ▪ The exception inheritance hierarchy 293 ▪ Example: A disk-writing program in Python 294 ▪ Example: Exceptions in normal evaluation 295
Where to use exceptions 296
- 14.3 Context managers using the with keyword 296
- 14.4 Adding exceptions 298
Solving the problem with AI-generated code 298 ▪ Solutions and discussion 298

PART 3 ADVANCED LANGUAGE FEATURES..... 307**15****Classes and object-oriented programming 309**

- 15.1 Defining classes 310
Using a class instance as a structure or record 310
- 15.2 Instance variables 311
- 15.3 Methods 312
- 15.4 Class variables 314
An oddity with class variables 315
- 15.5 Static methods and class methods 316
Static methods 316 ▪ Class methods 318
- 15.6 Inheritance 319
- 15.7 Inheritance with class and instance variables 321
- 15.8 Recap: Basics of Python classes 322
- 15.9 Private variables and private methods 325
- 15.10 Using @property for more flexible instance variables 326
- 15.11 Scoping rules and namespaces for class instances 328

- 15.12 Destructors and memory management 332
- 15.13 Multiple inheritance 332
- 15.14 HTML classes 334
 - Solving the problem with AI-generated code 334 □ Solutions and discussion 334*

16 Regular expressions 339

- 16.1 What is a regular expression? 340
- 16.2 Regular expressions with special characters 340
- 16.3 Regular expressions and raw strings 341
 - Raw strings to the rescue 342*
- 16.4 Extracting matched text from strings 343
- 16.5 Substituting text with regular expressions 347
 - Using a function with sub 347*
- 16.6 Phone number normalizer 348
 - Solving the problem with AI-generated code 349 □ Solutions and discussion 349*

17 Data types as objects 355

- 17.1 Types are objects too 356
- 17.2 Using types 356
- 17.3 Types and user-defined classes 357
- 17.4 Duck typing 359
- 17.5 What is a special method attribute? 360
- 17.6 Making an object behave like a list 361
- 17.7 The `__getitem__` special method attribute 362
 - How it works 363 □ Implementing full list functionality 364*
- 17.8 Giving an object full list capability 364
- 17.9 Subclassing from built-in types 367
 - Subclassing list 367 □ Subclassing UserList 368*
- 17.10 When to use special method attributes 369
- 17.11 Creating a string-only key-value dictionary 370
 - Solving the problem with AI-generated code 370 □ Solutions and discussion 371*

18 Packages 375

- 18.1 What is a package? 376
- 18.2 A first example: mathproj 376
- 18.3 Implementing the mathproj package 377
 - __init__.py files in packages 379 ▪ Basic use of the mathproj package 380 ▪ Loading subpackages and submodules 380*
 - import statements within packages 381*
- 18.4 The __all__ attribute 382
- 18.5 Proper use of packages 383
- 18.6 Creating a package 384
 - Solving the problem with AI-generated code 384 ▪ Solutions and discussion 384*

19 Using Python libraries 397

- 19.1 “Batteries included”: The standard library 398
 - Managing various data types 398 ▪ Manipulating files and storage 400 ▪ Accessing operating system services 400*
 - Using internet protocols and formats 401 ▪ Development and debugging tools and runtime services 402*
- 19.2 Moving beyond the standard library 403
- 19.3 Adding more Python libraries 403
- 19.4 The Python Package Index 403
- 19.5 Installing Python libraries using pip and venv 404
 - Installing with the –user flag 405 ▪ Virtual environments 405*
 - Other options 406*

PART 4 WORKING WITH DATA.....409

20 Basic file wrangling 411

- 20.1 The problem: The never-ending flow of data files 411
- 20.2 Scenario: The product feed from hell 412
- 20.3 More organization 414
- 20.4 Saving storage space: Compression and grooming 416
 - Compressing files 416 ▪ Grooming files 418*

21 Processing data files 420

- 21.1 Welcome to ETL 420
- 21.2 Reading text files 421
 - Text encoding: ASCII, Unicode, and others* 421 ▪ *Unstructured text* 423 ▪ *Delimited flat files* 425 ▪ *The csv module* 427
 - Reading a csv file as a list of dictionaries* 430
- 21.3 Excel files 430
- 21.4 Data cleaning 432
 - Cleaning* 432 ▪ *Sorting* 433 ▪ *Data cleaning problems and pitfalls* 435
- 21.5 Writing data files 435
 - CSV and other delimited files* 435 ▪ *Writing Excel files* 437
 - Packaging data files* 437
- 21.6 Weather observations 437
 - Solving the problem with AI-generated code* 438 ▪ *Solutions and discussion* 438

22 Data over the network 443

- 22.1 Fetching files 443
 - Using Python to fetch files from an FTP server* 444 ▪ *Fetching files with SFTP* 445 ▪ *Retrieving files over HTTP/HTTPS* 446
- 22.2 Fetching data via an API 447
- 22.3 Structured data formats 450
 - JSON data* 450 ▪ *XML data* 454
- 22.4 Scraping web data 457
- 22.5 Tracking the weather 461
 - Solving the problem with AI-generated code* 461 ▪ *Solutions and discussion* 461

23 Saving data 466

- 23.1 Relational databases 467
 - The Python database API* 467
- 23.2 SQLite: Using the sqlite3 database 467
- 23.3 Using MySQL, PostgreSQL, and other relational databases 470

- 23.4 Making database handling easier with an object relational mapper 470
 - SQLAlchemy* 471 ▪ *Using Alembic for database schema changes* 475
- 23.5 NoSQL databases 478
- 23.6 Key-value stores with Redis 478
- 23.7 Documents in MongoDB 482
- 23.8 Creating a database 486

24

Exploring data 488

- 24.1 Python tools for data exploration 488
 - Python’s advantages for exploring data* 489 ▪ *Python can be better than a spreadsheet* 489
- 24.2 Python and pandas 489
 - Why you might want to use pandas* 489 ▪ *Installing pandas* 490 ▪ *Data frames* 490
- 24.3 Data cleaning 492
 - Loading and saving data with pandas* 492 ▪ *Data cleaning with a data frame* 495
- 24.4 Data aggregation and manipulation 497
 - Merging data frames* 497 ▪ *Selecting data* 498
 - Grouping and aggregation* 500
- 24.5 Plotting data 501
- 24.6 Why you might not want to use pandas 502
 - Case study* 504
- appendix A guide to Python’s documentation* 521
- index* 539

foreword

I read the first edition of *The Quick Python Book* by Daryl Harms and Kenneth McDonald in late 1999. At the time there were only a handful of books about Python, and I believe I had them all. *The Quick Python Book* was the best book-length Python tutorial in the market. It offered a gentle learning curve and lots of practical advice, and it went beyond small snippets to discuss how to organize complete Python modules and programs.

All the qualities I mentioned remain, enhanced by new examples and insights by Naomi Ceder, who took over as the main author in the second edition.

The most important quality of *The Quick Python Book* has always been the practical context around the technical details. The labs at the end of the chapters and the chapter-length case study are about the creation and evolution of complete, useful programs.

In this fourth edition, Naomi complemented coverage of new Python features with two major changes. The first is using interactive notebooks on Google's Colaboratory. That allows readers to jump right into coding and experimenting without installing anything on their computers. It also helps them get comfortable with the most popular coding environment for data science and AI: Jupyter notebooks.

The second major change is in the labs. Naomi now compares her solutions with AI-generated code by the Colaboratory and GitHub Copilot. She shows the pros and cons of using those AI assistants, discusses their effects on the coding workflow, and advises on how to write prompts to get the best results.

Updating and improving this excellent book over the last 15 years is only one of Naomi's many contributions to Python. She has devoted countless hours of volunteer time to the Python Software Foundation and the wider Python international community.

Thousands of Python books later, this is still the best book-length Python tutorial that I know about. Thank you, Naomi, for keeping it fresh and ever more relevant in the 21st century!

—LUCIANO RAMALHO
PYTHONISTA SINCE 1998 AND AUTHOR OF *FLUENT PYTHON*

****preface****

Since I wrote the third edition of this book, more than five years have passed, and in that time we've been through some changes—including a pandemic and various crises but also the continued growth of data science and machine learning, the rise of AI-assisted code (AI everything, it seems), and of course the continued growth of Python, which is now one of the most used coding languages on the planet.

I've been coding in Python for almost 25 years now, far longer than in any other language I've ever used. I've used Python for system administration, web applications, database management, and data analysis over those years, but most important, I've come to use Python just to help myself think about a problem more clearly.

Based on my earlier experience, I would have expected that by now I would have been lured away by some other language that was faster, cooler, sexier, whatever. I think there are two reasons that didn't happen. First, while other languages have come along, none has helped me do what I needed to do quite as effectively as Python. Even after all these years, the more I use Python and the more I understand it, the more I feel the quality of my programming improves and matures.

The second reason I'm still around is the Python community. It's one of the most welcoming, inclusive, active, and friendly communities I've seen, embracing scientists, quants, web developers, systems people, and data scientists on every continent. It's been a joy and honor to work with members of this community, and I encourage everyone to join in.

Writing this book has shown me again how time passes and things change. While we're still on Python 3, today's Python 3 has evolved considerably even from the Python 3.6 of the last edition of this book. Things that were never dreamed of are now features of the language, and even the old nemesis of multicore processing, the GIL, is on the verge of workable solutions.

As the language has changed, so too have the ways people use Python. Although my goal has always been to keep the best bits of the previous edition, there have been a fair number of additions, deletions, and reorganizations that I hope make this edition both useful and timely. I've tried to keep the style clear, low-key, and also accessible to the many Python coders around the world whose first language is not English.

For me, the aim of this book is to share the positive experiences I've gotten from coding in Python by introducing people to Python 3, the latest and, in my opinion, greatest version of Python to date. May your journey be as satisfying as mine has been.

acknowledgments

I want to thank David Fugate of LaunchBooks for getting me into this book in the first place and for all of the support and advice he has provided over the years. I can't imagine having a better agent and friend. I also need to thank Jonathan Gennick of Manning for pushing the idea of doing a fourth edition of this book and supporting me in my efforts to make it as good as the first three. I also thank every person at Manning who worked on this project, with special thanks to Marjan Bace for his support, Doug Rudder for guidance in the development phases, Kathy Rossland for getting the book (and me) through the production process, Kari Lucke for her patience in copy editing, and Katie Tennant for proofreading. Likewise, hearty thanks go to the many reviewers whose insights and feedback were of immense help, especially Ken Alger, the technical editor for this edition of the book. Ken has taught, written about, and presented on various Python topics. In 2017, he served on the board of directors of the Django Software Foundation. Ken has worked with Python off and on since the mid-1990s, helping several industries automate tasks using Python.

Thanks as well go to the many others who provided valuable feedback, including all the reviewers: Aaron S. Bush, Advait Patel, Alfonso Harding, Anders Persson, Anuj Tyagi, Arian Ahmdinejad, Asaad Saad, Cay Horstmann, Chalamayya Batchu, Christian Sutton, Daivid Morgan, Dane Hillard, Darrin Bishop, Felipe Provezano Coutinho, Gajendra Babu Thokala, Ganesh Harke, Glen Mules, Greg Wagner, Harshita Asnani, Heather Ward, Ian Brierley, James Bishop, James Brett, James McCorrie, Jim Mason, Jose Apablaza, Juan Gomez, Justin Reiser, Ken Youens-Clark, Kiran Kumar Gunturu, Laurence Baldwin, Lav Kumar, Lev Veyde, Monisha Athi Kesavan Premalatha, Nicolas Chartier, Peter Van Caeneghem, Raghunath Mysore, Raz Pavel, Rich Hilliar, Robert F. Scheyder, Steve Grey-Wilson, Tanvir Kaur, Vatche Jabagchourian, Venkatesh Rajagopal, and William Jamir. Your suggestions helped make this a better book.

I have to thank the authors of the first edition, Daryl Harms and Kenneth MacDonald, for writing a book so sound that it has remained in print well beyond the average lifespan of most tech books, giving me a chance to do the second, third, and now fourth edition, as well as everyone who bought and gave positive reviews to the previous editions. I hope this version carries on the successful and long-lived tradition of the first three editions.

Thanks also go to Luciano Ramalho for the kindness with which he wrote the foreword to this edition, as well as for our years of friendship and the many years he has spent sharing his vast Python knowledge with communities in Brazil and throughout the world. Muito obrigada, meu amigo. I also owe thanks to the global Python community, an unfailing source of support, wisdom, friendship, and joy over the years. To borrow the quote one more time from Brett Cannon, “I came for the language, but I stayed for the community.” Thank you all, my friends.

Most important, as always, I thank my wife, Becky, who both encouraged me to take on this project and supported me through the entire process. I really couldn’t have done it without her.

about this book

The Quick Python Book, Fourth Edition, is intended for people who already have experience in one or more programming languages and want to learn the basics of Python 3 as quickly and directly as possible. Although some basic concepts are covered, there's no attempt to teach fundamental programming skills in this book, and the basic concepts of flow control, object-oriented programming (OOP), file access, exception handling, and the like are assumed. This book may also be of service to users of earlier versions of Python who want a concise reference for Python 3.

Who should read this book

This book is intended for readers who know how to write some code and understand basic programming concepts. They might be coming from another programming language, they might still be learning Python and wanting to level up their knowledge, or they might be looking to update and refresh their knowledge. This book provides a compact and accessible view of the Python landscape at the level of the features that are used to get 90% of the work done.

How this book is organized: A road map

Part 1 introduces Python and explains how to use Python via Google's Colaboratory and how to obtain the source code Jupyter notebooks from the book's GitHub repository. It also includes a very general survey of the language, which will be most useful for experienced programmers looking for a high-level view of Python:

- Chapter 1 discusses the strengths and weaknesses of Python and shows why Python is a good choice of programming language for many situations.

- Chapter 2 covers how to use Python via Google’s Colaboratory and how to obtain the source code as Jupyter notebooks from the book’s GitHub repository.
- Chapter 3 is a short overview of the Python language. It provides a basic idea of the philosophy, syntax, semantics, and capabilities of the language.

Part 2 is the heart of the book. It covers the ingredients necessary for obtaining a working knowledge of Python as a general-purpose programming language. The chapters are designed to allow readers who are beginning to learn Python to work their way through sequentially, picking up knowledge of the key points of the language. These chapters also contain some more advanced sections, allowing you to return to find in one place all the necessary information about a construct or topic:

- Chapter 4 starts with the basics of Python. It introduces Python variables, expressions, strings, and numbers. It also introduces Python’s block-structured syntax.
- Chapters 5, 6, and 7 describe the five powerful built-in Python data types: lists, tuples, sets, strings, and dictionaries.
- Chapter 8 introduces Python’s control flow syntax and use (loops, `if-else` statements, and the new `match-case` statement).
- Chapter 9 describes function definition in Python along with its flexible parameter-passing capabilities.
- Chapter 10 describes Python modules, which provide an easy mechanism for segmenting the program namespace.
- Chapter 11 covers creating standalone Python programs, or scripts, and running them on Windows, macOS, and Linux platforms. The chapter also covers the support available for command-line options, arguments, and I/O redirection.
- Chapter 12 describes how to work with and navigate through the files and directories of the filesystem. It shows how to write code that’s as independent as possible of the actual operating system you’re working on.
- Chapter 13 introduces the mechanisms for reading and writing files in Python, including the basic capability to read and write strings (or byte streams), the mechanism available for reading binary records, and the ability to read and write arbitrary Python objects.
- Chapter 14 discusses the use of exceptions, the error-handling mechanism used by Python. It doesn’t assume that you have any previous knowledge of exceptions, although if you’ve previously used them in C++ or Java, you’ll find them familiar.

Part 3 introduces advanced language features of Python—elements of the language that aren’t essential to its use but that can certainly be a great help to a serious Python programmer:

- Chapter 15 introduces Python’s support for writing object-oriented programs.
- Chapter 16 discusses the regular expression capabilities available for Python.

- Chapter 17 introduces more advanced OOP techniques, including the use of Python’s special method attributes mechanism, metaclasses, and abstract base classes.
- Chapter 18 introduces the package concept in Python for structuring the code of large projects.
- Chapter 19 is a brief survey of the standard library. It also includes a discussion of where to find other modules and how to install them.

Part 4 describes more advanced or specialized topics that are beyond the strict syntax of the language. You may read these chapters or not, depending on your needs.

- Chapter 20 dives deeper into manipulating files in Python.
- Chapter 21 covers strategies for reading, cleaning, and writing various types of data files.
- Chapter 22 surveys the process, issues, and tools involved in fetching data over the network.
- Chapter 23 discusses how Python accesses relational and NoSQL databases.
- Chapter 24 is a brief introduction to using Python, Jupyter notebooks, and pandas to explore datasets.
- The case study walks you through using Python to fetch data, clean it, and then graph it. The project combines several features of the language discussed in the chapters, and it gives you a chance to see a project worked through from beginning to end.
- The appendix contains a guide to obtaining and accessing Python’s full documentation, the Pythonic style guide, PEP 8, and “The Zen of Python,” a slightly wry summary of the philosophy behind Python.

A suggested plan if you’re new to Python programming and want to skip straight to the language is to start by reading chapter 3 to obtain an overall perspective and then work through the chapters in part 2 that are applicable. Enter in the interactive examples as they are introduced to immediately reinforce the concepts. You can also easily go beyond the examples in the text to answer questions about anything that may be unclear. This has the potential to amplify the speed of your learning and the level of your comprehension. If you aren’t familiar with OOP or don’t need it for your application, skip most of chapter 15.

Those who are familiar with Python should also start with chapter 3. It’s a good review and introduces differences between Python and what may be more familiar. It’s also a reasonable test of whether you’re ready to move on to the advanced chapters in parts 3 and 4 of this book.

It’s possible that some readers, although new to Python, will have enough experience with other programming languages to be able to pick up the bulk of what they need to get going by reading chapter 3 and by browsing the Python standard library modules listed in chapter 19 and the documentation for the Python standard library.

About the code

The code samples in this book are deliberately kept as simple as possible, because they aren't intended to be reusable parts that can be plugged into your code. Instead, the code samples are stripped down so that you can focus on the principle being illustrated.

This book is based on Python 3.13, including some features not yet supported on Colaboratory (which is on Python 3.11 at the time of writing), and all examples should work on any subsequent version of Python 3. Other than the newest features, the examples also work on Python 3.9 and later, but I strongly recommend using the most current version you can; there are no advantages to using the earlier versions, and each version has several subtle improvements over the earlier ones.

In keeping with the idea of ease of use, the code examples are presented as Jupyter notebook cells that you can run in Google Colaboratory or with any other Jupyter server. Where possible, you should enter and experiment with these samples as much as you can.

The Jupyter notebooks with the code for the examples and the answers to most of the Quick Check and Try This exercises in this book are available from the publisher's website at www.manning.com/books/the-quick-python-book-fourth-edition. You can also find all of the code in a GitHub repository at <https://github.com/nceder/qpb4e>.

You can get executable snippets of code from the liveBook (online) version of this book at <https://livebook.manning.com/book/the-quick-python-book-fourth-edition>.

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a `fixed-width font` like this to separate it from ordinary text. The output of the code when run is also `in bold`.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In some cases, even this was not enough, and listings include line-continuation markers (`→`). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

In some cases, a longer code sample is needed, and these cases are identified in the text as file listings. You can download these files from the source code repository or save the listings as files with names matching those used in the text and run them as stand-alone scripts.

liveBook discussion forum

Purchase of *The Quick Python Book, Fourth Edition*, includes free access to liveBook, Manning's online reading platform. Using liveBook's exclusive discussion features, you can attach comments to the book globally or to specific sections or paragraphs. It's a snap to make notes for yourself, ask and answer technical questions, and receive help from the author and other users. To access the forum, go to <https://livebook.manning.com/book/the-quick-python-book-fourth-edition/discussion>. You can also learn

more about Manning's forums and the rules of conduct at <https://livebook.manning.com/book/discussion>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest their interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the author



NAOMI CEDER has been programming in various languages for over 30 years and has been a Linux system administrator, programming teacher, developer, and system architect. She started using Python in 2001 and since then has taught Python to users at all levels, from 12-year-olds to professionals. She gives talks on Python and the benefits of an inclusive community to whom-ever will listen. A past chair of the Python Software Foundation and recipient of the PSF Distinguished Service Award, Naomi is now retired, which leaves her more free time to play the classical guitar.

about the cover illustration

The figure on the cover of *The Quick Python Book, Fourth Edition*, is “Bavaroise” or “Woman from Bavaria,” taken from a 19th-century edition of Sylvain Maréchal’s four-volume compendium, *Costumes Civils Actuels de Tous les Peuples Connus*. Each illustration is finely drawn and colored by hand.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.

Part 1

Starting out

T

hese first three chapters tell you a little bit about Python, its strengths and weaknesses, and why you should consider learning Python. In chapter 2 you see how to run Python using Google Colaboratory, how to access the book's example code repository on GitHub, and how to write a simple program. Chapter 3 is a quick, high-level survey of Python's syntax and features. If you're looking for the quickest possible introduction to Python, start with chapter 3.

About Python



This chapter covers

- Why use Python?
- What Python does well
- What Python is improving

This book is intended to help people get a solid general understanding of Python as quickly as possible, avoiding getting bogged down in advanced topics but covering the essentials to write and read Python code. In particular, this book is intended for people who are coming to Python from other languages and for those who know a bit of Python but are looking to level up their skills as Python continues to gain popularity, particularly in areas like data science, machine learning, and the sciences. While no prior knowledge of Python is needed, some knowledge and experience with programming is necessary to get the most out of this book.

After introducing Python and offering some advice on getting started with a Python environment, the book provides a quick summary of Python's syntax, followed by chapters that build from the built-in data types up through creating functions, classes, and packages, as well as some more advanced features and a case study in handling data.

With the rise of AI tools based on large language models (LLMs), it is now possible to generate increasing amounts of usable code if (and that's a big "if") one has enough knowledge of coding to guide the process intelligently. While this book is *not* a tutorial on AI and its use in code generation, the coding problems posed at the end of each chapter, starting with chapter 5, provide examples of AI responses to the same questions along with a brief discussion of what the AI got right (and wrong). This will help build an understanding of how to use AI tools to generate code that actually works.

1.1 **Why should I use Python?**

Hundreds of programming languages are available today, from mature languages like C and C++, to newer entries like Rust, Go, and C#, to enterprise juggernauts like Java, to the more web-oriented JavaScript and Typescript. This abundance of choice makes deciding on a programming language difficult. Although no one language is the right choice for every possible situation, I think that Python is a good choice for a large number of programming problems, and it's also a good choice if you're learning to program. Millions of programmers around the world use Python, and the number grows every year.

Python continues to attract new users for a variety of reasons. It's a true cross-platform language, running equally well on Windows, Linux/UNIX, and iOS platforms, as well as others, ranging from supercomputers to mobile devices. It can be used to develop small applications and rapid prototypes, but it scales well to permit the development of large programs. It comes with a powerful and easy-to-use graphical user interface (GUI) toolkit, web programming libraries, and more. Python has also become a vital tool for scientific computing and for data science, machine learning, and work with AI. *And it's free.*

1.2 **What Python does well**

Python is a modern programming language developed by Guido van Rossum in the 1990s (and named after a famous comedic troupe). Although Python isn't perfect for every application, its strengths make it a good choice for many situations.

1.2.1 **Python is easy to use**

Programmers familiar with traditional languages will find it easy to learn Python. All of the familiar constructs—loops, conditional statements, arrays, and so forth—are included, but many are easier to use in Python. Here are a few of the reasons why:

- *Types are associated with objects, not variables.* A variable can be assigned a value of any type, and a list can contain objects of many types. This also means that type casting usually isn't necessary and that your code isn't locked into the straitjacket of predeclared types. But while types are not *required* for variables, Python does allow *type hints* that allow developers to check that their code is consistent in the type of object used for parameters, return values, etc.; we'll talk a little bit about type hints later in the book.

- *Python typically operates at a much higher level of abstraction.* This is partly the result of the way the language is built and partly the result of an extensive standard code library that comes with the Python distribution. A program to download a web page can be written in two or three lines!
- *Syntax rules are very simple.* Although becoming an expert Pythonista takes time and effort, even beginners can absorb enough Python syntax to write useful code quickly.

Python is well suited for rapid application development. It isn't unusual for coding an application in Python to take one-fifth the time it would in C or Java and to take as little as one-fifth the number of lines of the equivalent C program. This depends on the particular application, of course; for a numerical algorithm performing mostly integer arithmetic in `for` loops, there would be much less of a productivity gain. For the average application, the productivity gain can be significant.

1.2.2 Python is expressive

Python is a very expressive language. *Expressive* in this context means that a single line of Python code can do more than a single line of code in most other languages. The advantages of a more expressive language are obvious: the fewer lines of code you have to write, the faster you can complete the project. The fewer lines of code there are, the easier the program will be to maintain and debug.

To get an idea of how Python's expressiveness can simplify code, consider swapping the values of two variables, `var1` and `var2`. In a language like Java, this requires three lines of code and an extra variable:

```
int temp = var1;
var1 = var2;
var2 = temp;
```

The variable `temp` is needed to save the value of `var1` when `var2` is put into it, and then that saved value is put into `var2`. The process isn't terribly complex, but reading those three lines and understanding that a swap has taken place takes a certain amount of overhead, even for experienced coders.

By contrast, Python lets you make the same swap in one line and in a way that makes it obvious that a swap of values has occurred:

```
var2, var1 = var1, var2
```

Of course, this is a very simple example, but you can find the same advantages throughout the language.

1.2.3 Python is readable

Another advantage of Python is that it's easy to read. You might think that a programming language needs to be read only by a computer, but humans have to read your

code as well: whoever debugs your code (quite possibly you), whoever maintains your code (could be you again), and whoever might want to modify your code in the future. In all of those situations, the easier the code is to read and understand, the better it is.

The easier code is to understand, the easier it is to debug, maintain, and modify. Python's main advantage in this department is its use of indentation. Unlike most languages, Python *insists* that blocks of code be indented. Although this strikes some people as odd, it has the benefit that your code is always formatted in a very easy-to-read style.

The following are two short programs, one written in Perl and one in Python. Both take two equal-size lists of numbers and return the pairwise sum of those lists. I think the Python code is more readable than the Perl code; it's visually cleaner and contains fewer inscrutable symbols:

```
# Perl version.
sub pairwise_sum {
    my($arg1, $arg2) = @_;
    my @result;
    for(0 .. $#$arg1) {
        push(@result, $arg1->[$_] + $arg2->[$_]);
    }
    return(\@result);
}

# Python version.
def pairwise_sum(list1, list2):
    result = []
    for i in range(len(list1)):
        result.append(list1[i] + list2[i])
    return result
```

Both pieces of code do the same thing, but the Python code wins in terms of readability. (There are other ways to do this in Perl, of course, some of which are much more concise—but, in my opinion, are harder to read than the one shown.)

1.2.4 Python is complete: “Batteries included”

Another advantage of Python is its “batteries included” philosophy when it comes to libraries. The idea is that when you install Python, you should have everything you need to do real work without the need to install additional libraries. This is why the Python standard library comes with modules for handling email, web pages, databases, operating-system calls, GUI development, and more.

For example, with Python, you can write a web server to share the files in a directory with just two lines of code:

```
import http.server
http.server.test(HandlerClass=http.server.SimpleHTTPRequestHandler)
```

There's no need to install libraries to handle network connections and HTTP; it's already in Python, right out of the box.

1.2.5 Python has a rich ecosystem of third-party libraries

While Python is “batteries included,” there are still many situations where one needs to go beyond even a well-stocked standard library—a specialized task, a new data format, more complex applications, and so on. Here Python has really come to the forefront over the past decade. In many areas, from web applications and APIs to data handling and visualization, to machine learning and data science and more, Python has one of the richest ecosystems of packages, libraries, and frameworks of any current language. It’s very unlikely that you’ll find yourself in a situation where there are no Python packages to meet your needs.

1.2.6 Python is cross-platform

Python is also an excellent cross-platform language. Python runs on many platforms, including Windows, Mac, Linux, UNIX, and others. Because it’s interpreted, the same code can run on any platform that has a Python interpreter, and almost all current platforms have one. There are even versions of Python that run on Java (Jython), .NET (IronPython), and microcontrollers (MicroPython and CircuitPython), giving you even more possible platforms that run Python.

1.2.7 Python is free

Python is also free. Python was originally, and continues to be, developed under the open source model, and it’s freely available. You can download and install practically any version of Python and use it to develop software for commercial or personal applications, and you don’t need to pay a dime.

Although attitudes are changing, some people are still leery of free software because of concerns about a lack of support, fearing such software lacks the clout of paying customers. But Python is used by many established companies as a key part of their business; Google, Bloomberg, NVIDIA, and Capital One are just a few examples. These companies and many others know Python for what it is: a very stable, reliable, and well-supported product with an active and knowledgeable user community. You’ll get an answer to even the most difficult Python question more quickly in various Python internet forums than you will on most tech-support phone lines, and the Python answer will be free and correct.

Python and open source software

Not only is Python free but its source code is also freely available, and you’re free to modify, improve, and extend it if you want. Because the source code is freely available, you have the ability to go in yourself and change it (or to hire someone to go in and do so for you). You rarely have this option at any reasonable cost with proprietary software.

If this is your first foray into the world of open source software, you should understand that you’re not only free to use and modify Python but also able (and encouraged) to contribute to it and improve it. Depending on your circumstances, interests, and

(continued)

skills, those contributions might be financial, as in a donation to the Python Software Foundation, or they may involve participating in one of the special interest groups, testing and giving feedback on releases of the Python core or one of the auxiliary modules, or contributing some of what you or your company develops back to the community. The level of contribution (if any) is, of course, up to you; but if you're able to give back, definitely consider doing so. Something of significant value is being created here, and you have an opportunity to add to it.

Python has a lot going for it: expressiveness, readability, rich included libraries, and cross-platform capabilities. Also, it's open source. What's the catch?

1.3 **What Python is improving**

Although Python has many advantages, no language can do everything, so Python isn't the perfect solution for all your needs. While Python is improving in all of the following areas, to decide whether it is the right language for your situation, you also need to consider these areas where Python doesn't do as well.

1.3.1 **Python is getting faster**

A possible drawback with Python is its speed of execution. It isn't a fully compiled language. Instead, it's first compiled to an internal bytecode form, which is then executed by a Python interpreter. There are some tasks, such as string parsing using regular expressions, for which Python has efficient implementations and is as fast as, or faster than, any C program you're likely to write. Nevertheless, most of the time, using Python results in slower programs than in a language like C. But you should keep this in perspective. Modern computers have so much computing power that, for the vast majority of applications, the speed of the program isn't as important as the speed of development, and Python programs can typically be written much more quickly than others. In addition, it's easy to extend Python with modules written in C or C++, which can be used to run the CPU-intensive portions of a program.

Python's core developers are also hard at work creating new versions of Python that are more efficient, load and run faster, and take better advantage of multiple processor cores. This work has already yielded significant improvements in performance, and the work will continue in the future, so if you have a performance-critical application, you may want to consider carefully if Python will do the job—but don't write it off immediately.

1.3.2 **Python doesn't enforce variable types at compile time**

Unlike in some languages, Python's variables don't work like containers; instead, they're more like labels that refer to various objects: integers, strings, class instances, whatever. That means that although those objects themselves have types, the variables referring to them aren't bound to those particular types. It's possible (if not necessarily

desirable) to use the variable `x` to refer to a string in one line and an integer in another (note: output of the code is in **bold**):

```
x = "2"  
x  
  
'2'           ← Output of x is the string '2'.  
  
x = int(x)  
x  
  
2             ← Output of x is now the integer 2.
```

The fact that Python associates types with objects and not with variables means that the interpreter doesn't help you catch variable type mismatches. If you intend a variable `count` to hold an integer, Python won't complain if you assign the string "`two`" to it. Traditional coders count this as a disadvantage, because you lose an additional free check on your code.

In response to this concern, Python has added syntax and tools to allow coders to specify the desired type of the object a variable refers to, as well as function parameters, return values, and the like. With these *type hints*, as they are called, various tools can flag any inconsistencies in the types of objects before runtime. In smaller programs, type errors usually aren't hard to find and fix even without type hints, and in any case, Python's testing features makes avoiding type errors manageable. Many Python programmers feel that the flexibility of dynamic typing more than outweighs any advantage mandatory variable typing might offer.

1.3.3 Python is improving mobile support

In the past decade, the numbers and types of mobile devices have exploded, and smartphones, tablets, phablets, Chromebooks, and more are everywhere, running on a variety of operating systems. Python isn't a strong player in this space, but various projects are working on the problem, developing toolkits and frameworks that allow writing apps for both iOS and Android platforms. This situation is improving, but as of this writing, using Python to write and distribute commercial mobile apps is a bit of a pain.

1.3.4 Python is improving support for multiple processors

Multiple-core processors are everywhere now, producing significant increases in performance in many situations. However, the standard implementation of Python isn't designed to use multiple cores, due to a feature called the *global interpreter lock*. As mentioned in the context of speed, Python's development team is currently working on ways to make Python work more seamlessly and efficiently with multiple processor cores, and the development of multicore Python will continue over the next several years.

Summary

- Python is a modern, high-level language with dynamic typing and simple, consistent syntax and semantics.
- Python is multiplatform, highly modular, and suited for both rapid development and large-scale programming.
- It's reasonably fast and can be easily extended with C or C++ modules for higher speeds.
- Python has built-in advanced features, such as persistent object storage, advanced hash tables, expandable class syntax, and universal comparison functions.
- Python includes a wide range of libraries, such as numeric processing, image manipulation, user interfaces, and web scripting.
- It's supported by a dynamic community.

Getting started

This chapter covers

- Available Python options
- Getting started with Colaboratory
- Accessing the GitHub repository for this book
- Writing a simple program and handling errors
- Using the `help()` and `dir()` functions
- AI tools for generating Python code

This chapter gives you a quick survey of the many options for installing and using Python and guides you through getting up and running with Google Colaboratory, a web-hosted Python environment. Combined with the code for each chapter, available in a GitHub repository at <https://github.com/nceder/qpb4e>, Colaboratory is one of the quickest and easiest ways to get up and running with Python so that you can test out the code examples in the text and work through the lab exercises throughout the book.

2.1 Paradox of choice: Which Python?

As Python has matured, particularly over the past 10 years, the number of options for running Python has exploded. After a brief survey of the versions, sources, devices, environments, and tools in the Python ecosystem, we'll look at our recommended solution, Google Colaboratory, and how to use it to run the sample code snippets in this book and write the code for the lab exercises in most of the chapters.

2.1.1 Python versions

For many years, Python had a somewhat variable release cycle, with new versions being released when the core developers felt there were enough new features to be worth it. From Python 3.1 in 2009 to 3.8 in 2018, new versions were released roughly every 18 months. As Python's popularity continued to grow, a quicker and more regular cadence for new releases became the goal. So, beginning with version 3.9, the decision was made (recorded in PEP 602: Annual Release Cycle for Python) to move to an annual release cycle, with a release every October.

The current release schedule for Python means that after the October 2024 release of version 3.13, version 3.14 will come out in October 2025, and so on. As of this writing, Python 3.13 is the most current version, having been released in October 2024. Since the features of 3.13 have already been frozen, I have included the new features of 3.13 and I have tested the code on both 3.12 and 3.13.

Choosing a Python version

Having a new version of Python come out every year is a good thing for releasing new features, but it's not always the most convenient thing for anyone who needs to test and deploy a new version of the language. Because of this, versions of Python are officially supported for a total of five years: the first two years with full releases and then three years with source code only with releases of security fixes.

While it's usually preferable to have the latest version, any supported version (3.9 or later as of this writing) should be fine to use with this book. If you have an earlier version, there will be a few new features that you won't have, but you can still do a lot of work without upgrading.

2.1.2 Sources of Python

The classic way to get Python is by downloading and installing a version from the Python Software Foundation's website at <https://www.python.org/downloads/>. There you can find installers for several operating systems and platforms and versions going back to the very beginnings of Python, over 30 years ago. It's possible, however, that the exact version you need or want isn't on python.org. You can also get a data science-oriented distribution of Python from Anaconda at <https://www.anaconda.com/download/>, or you may be able to use the app store or package manager of your operating system

to install Python. If you are into the technical details, you can also fork the source code repository on GitHub and download and compile the source yourself. The choice depends on your preferences and your situation, and since the solution I recommend doesn't require a local install, you can wait to make that decision until you know Python a bit better.

2.1.3 **Devices, platforms, and operating systems**

You can run Python on a wide variety of devices and on many operating systems. While in the early days, a server or desktop computer was the norm, today Python runs on many processors, from embedded devices and single board computers to phones and tablets, to Chromebooks, laptops, and desktops, to virtual machines and containers, to server clusters, and so on. While the most common implementation of Python is based on the C language, there are versions that are written in and run on Java, in the browser based on WASM, or even in applications like Excel.

2.1.4 **Environments and tools**

There are also a number of ways that you can run and develop code in Python. It's possible to run the Python interpreter from a command line and interact using either the built-in shell or an enhanced shell like IPython and to write code in the text editor of your choice. You can also use education-oriented IDEs like IDLE (provided with Python), Mu, or Thonny, or you can opt for more advanced Python IDEs like VSCode, Wingware, or PyCharm. Of course, in all of those categories there are too many other fine options to list.

Another option for using and writing Python combines features of the IPython shell with a web interface and is called Jupyter. Jupyter runs on a server and is accessed via browser, and its web interface allows you to combine text (formatted using Markdown) with cells for executing code in a "notebook." The combination has proven to be so useful that Jupyter notebooks are increasingly popular for teaching, data exploration and data science, AI experimentation, and many other uses. While you can run your own Jupyter server locally, it's even more convenient to use a server hosted in the cloud, which brings us to my recommendation for this book.

2.2 **Colaboratory: Jupyter notebooks in the cloud**

As mentioned at the beginning of this chapter, for this book I recommend using Jupyter notebooks, specifically, the hosted version of Jupyter from Google, Colaboratory. Using Colaboratory means that you don't need to worry about installing (or updating) Python, and you can take advantage of Jupyter's friendly interface, which is becoming increasingly popular, particularly for data science.

Another advantage of Colaboratory is that you can launch a session with one of the code notebooks from this book's source repository on GitHub with a single click, and you can access it on virtually any device that has a modern web browser.

For that reason, in this book, I'll present sample code and problem solutions in Jupyter notebooks. Of course, the code will still work in other environments, and I'll provide

text files of the code as well, but unless stated otherwise, you can assume that the code presented is in notebook form.

2.2.1 Getting the source notebooks

The simplest way to get started with Colaboratory and this book is by accessing a notebook in the GitHub repository. If you are not familiar with GitHub, it's a very popular online version control service based on the version control tool Git. To access the notebooks in the repository, you don't need to know how to use Git, nor do you need to have an account on GitHub—just following the links given here will get you started.

You can find the GitHub repository at <https://github.com/nceder/qp4e/tree/main>; the notebooks for each chapter are in separate directories inside the `code` directory. To get started, let's find and open the notebook for chapter 2 (this chapter). If you look in the `Chapter 02` directory, you will find a notebook file called `Chapter_02.ipynb` (direct link at <https://mng.bz/gaDG>). You should see a page similar to the one shown in figure 2.1.

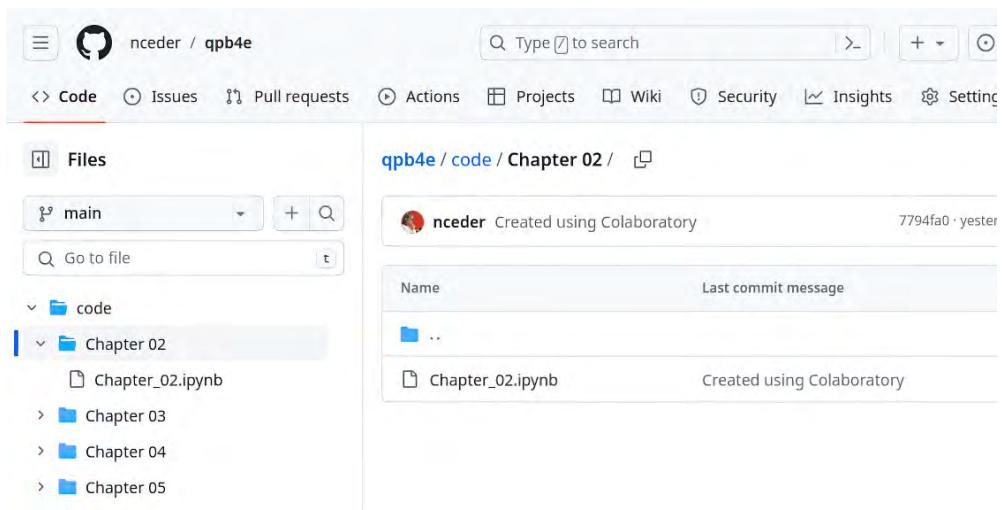


Figure 2.1 GitHub repository, showing Chapter_02 notebook

2.2.2 Getting started with Colaboratory

If you click the `Chapter_02.ipynb` file, you can view its contents using GitHub's viewer (as shown in figure 2.2), and at the top of the code window (right above 2.2 Getting Started with Python and Colaboratory), you should also see a blue button link Open in Colab.

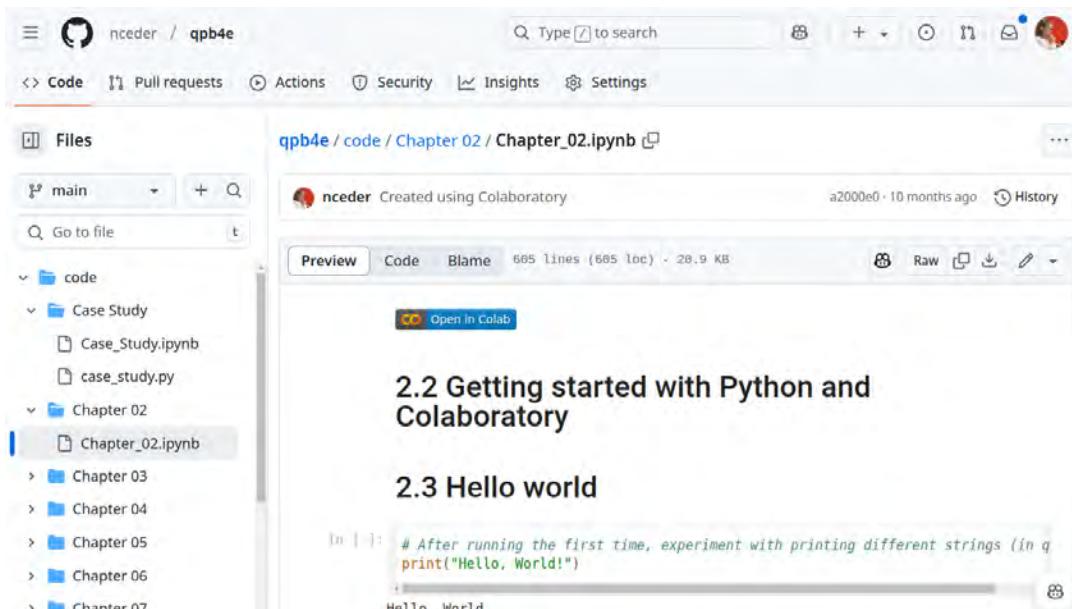


Figure 2.2 Chapter_02.ipynb notebook viewed in GitHub

If you click that link, it will open that notebook in a Colaboratory session where you can edit, run, and experiment with the code, as shown in figure 2.3.



Figure 2.3 Chapter_02.ipynb notebook open in Colaboratory

As with GitHub, you don't need a Google account to access Colaboratory, but to run code and use all of the features, including the ability to save sessions and files, logging in with a Google account is required.

LOGGING INTO COLABORATORY FIRST

You can also access Colaboratory directly by using your browser to open <https://colab.research.google.com/>. This will open Colaboratory with a “Welcome to Colab” notebook (as shown in figure 2.4) that explains some of the platform’s features. If you are unfamiliar with Jupyter notebooks, there are some links at the bottom of this notebook to other notebooks that explain how Colaboratory’s version of Jupyter notebooks works. If you find yourself a bit confused at first, those resources can help.

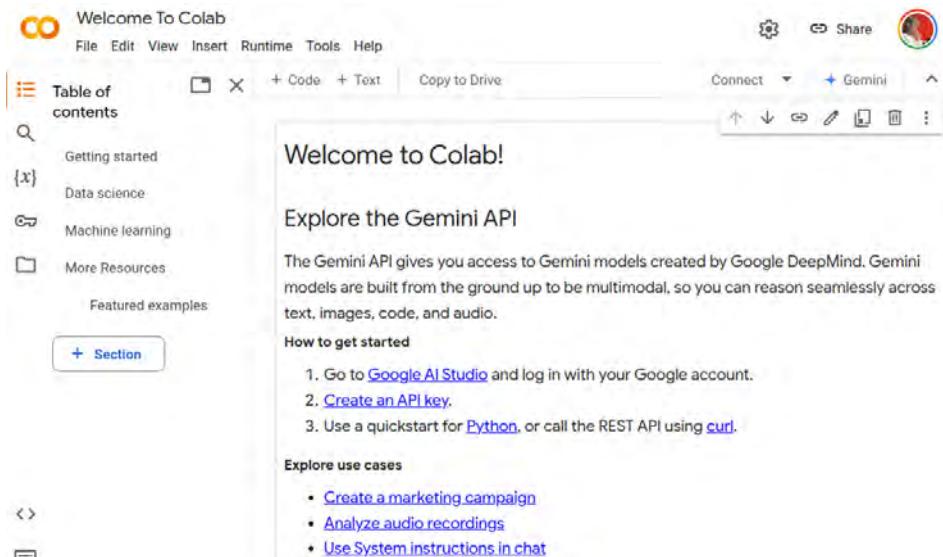


Figure 2.4 Opening Colaboratory

To open a file for this book, you would select the File menu under Welcome to Colab at the upper left of the window and select the Open notebook option. To illustrate how it works, again let’s look at the notebook for this chapter.

Enter <https://github.com/nceder/qpb4e> where it asks for a GitHub URL, and click the search icon at the end of the line, as shown in figure 2.5. The paths to all of the notebooks will appear below, and you should then click code/Chapter 02/Chapter_02.ipynb to open the notebook for chapter 2.

2.2.3 *Colaboratory’s Python version*

The good news about using Colaboratory is that you don’t have to worry about installing and maintaining a Python environment, and the Jupyter notebooks are accessible from almost any device with a standard web browser. One minor downside of using Colaboratory is that the version of Python that it runs may be a few months behind the latest version, which means that a few of the very newest features of Python may not

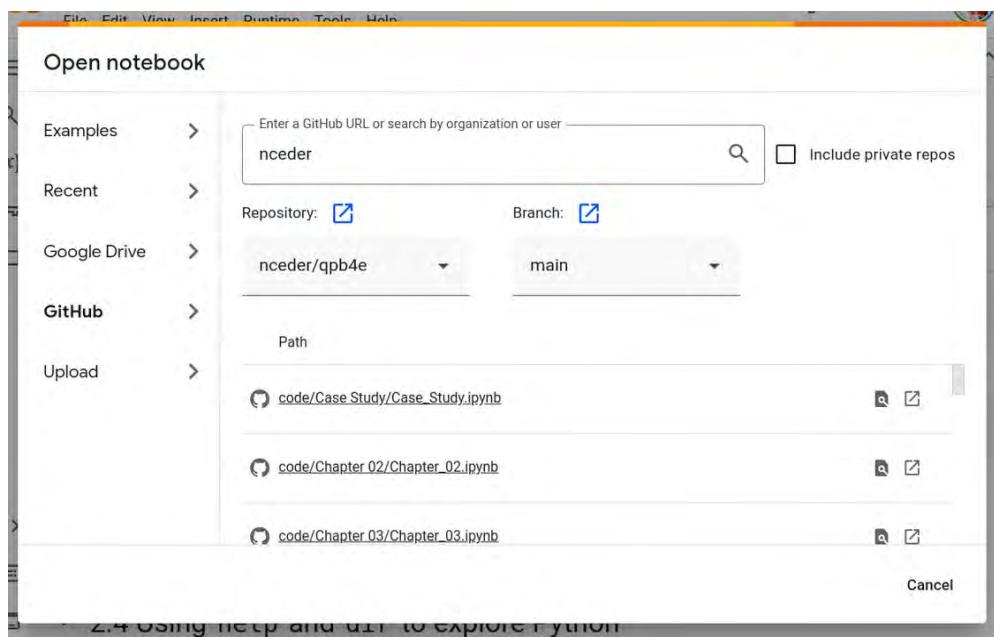


Figure 2.5 Opening a notebook from the GitHub repository

be available. However, Colaboratory's ease of access and use far outweighs this slight drawback.

2.3 Writing and running code in Colaboratory

A Jupyter notebook consists of two types of cells: text cells and code cells. The text cells are meant to contain text, which can be formatted using the Markdown text formatting language (there's a guide to Colaboratory's flavor of Markdown at <https://mng.bz/eyjq>). If you double-click a text cell or right-click and select Edit, the cell will be opened in edit mode, and the text and formatting can be changed. To save the changes and go back to text mode, you can use either Ctrl-Enter or you can right-click the cell and choose Select. Any changes you make to the source notebooks will not be changed in the GitHub repository, nor will they be saved in Colaboratory unless you are logged in with a Google account.

The code cells are for code that can be executed and are always in edit mode. To execute the code, you can either use Ctrl-Enter or you can right-click the cell and click Run the Focused Cell. You can also click the little triangle in a circle that appears to the left of the cell when you hover your mouse over the cell. The first time you run code from a notebook loaded from GitHub, you may see a warning that the code is being loaded from GitHub. This is normal, and it's fine to click the Run Anyway choice to execute the code. If there is any output from the execution of the code, it will appear

in the area immediately below the cell. We'll see how all of this works as we run our first bit of code next.

2.3.1 Hello, World

Let's start with the traditional Hello, World program, which is a one-liner in Python:

```
print("Hello, World")      ← Code entered in code cell
Hello, World             ← Output from running code, below code cell
```

Here, the code for the `print()` function, along with the string "Hello, World" as its parameter, has already been entered into the first code cell of the chapter 2 notebook. To run it, as mentioned earlier, you can use Ctrl-Enter and the output, "Hello, World" will be printed below the cell. If you are new to Jupyter notebooks (or to Python), feel free to experiment a bit with the `print` function or other commands in this code cell.

Note that throughout this book, I will show the code that is entered into a code cell in Jupyter (or a Python shell prompt in some other environment) using a normal-weight monospace font and the output as a **monospace bold** font. For comparison, the previous code and output should look something like figure 2.6 in Colaboratory.

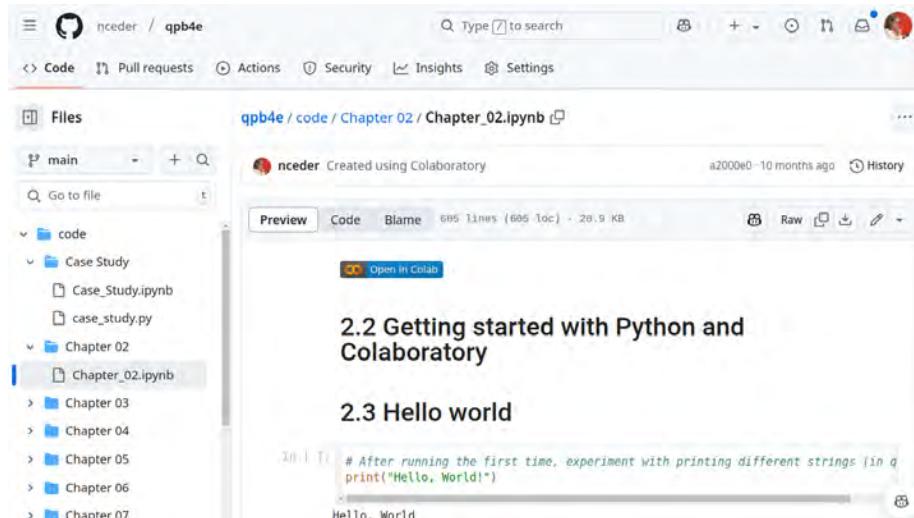


Figure 2.6 Text cells, code cell, and output

2.3.2 Dealing with errors in Jupyter

Errors are something all coders have to deal with in virtually every piece of code we write. If there is an error while running code in a code cell, Jupyter prints Python's

error message below the cell where the output would go. Code error messages have been famously difficult to understand, but the latest versions of Python have improved the clarity and usefulness of error messages:

```
# this cell will raise a syntax error
print("Hello, World!"
```

The diagram shows a Python syntax error message with several annotations:

- Code missing closing parenthesis**: A callout pointing to the opening parenthesis in the code.
- Message indicating the line where the error occurs**: A callout pointing to the line number in the stack trace.
- Caret indicating location in line**: A callout pointing to the caret (^) character in the code line.
- Name or explanation of error**: A callout pointing to the error type and message.

```
File "<ipython-input-3-0d1705def0fc>", line 2
    print("Hello, World!"
          ^
SyntaxError: incomplete input
```

In this case, the code deliberately has a syntax error—the closing parenthesis has been omitted. Python reports the message and the line where it occurs, puts a caret (^) under the error's location in the line, and finally names the problem “incomplete input.”

A nice feature of Colaboratory is that below the Python error message it may offer an option to attempt to fix the error automatically, with a Next Steps: section and a Fix Error button, as shown in figure 2.7. In this case, clicking the Fix Error button will enter the fix in the previous code cell and allow the user to either accept or reject it.

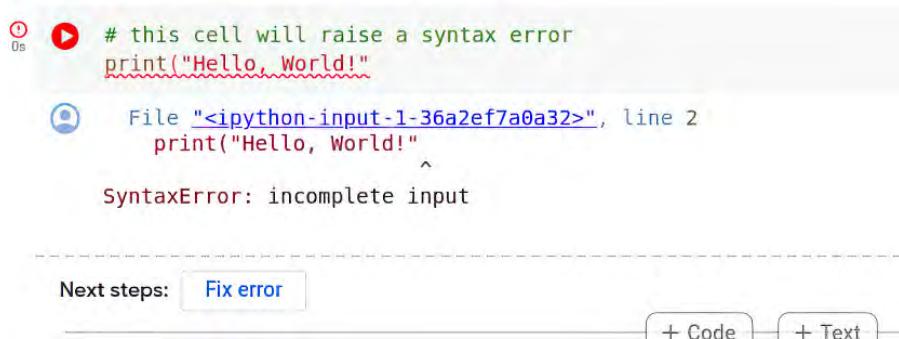


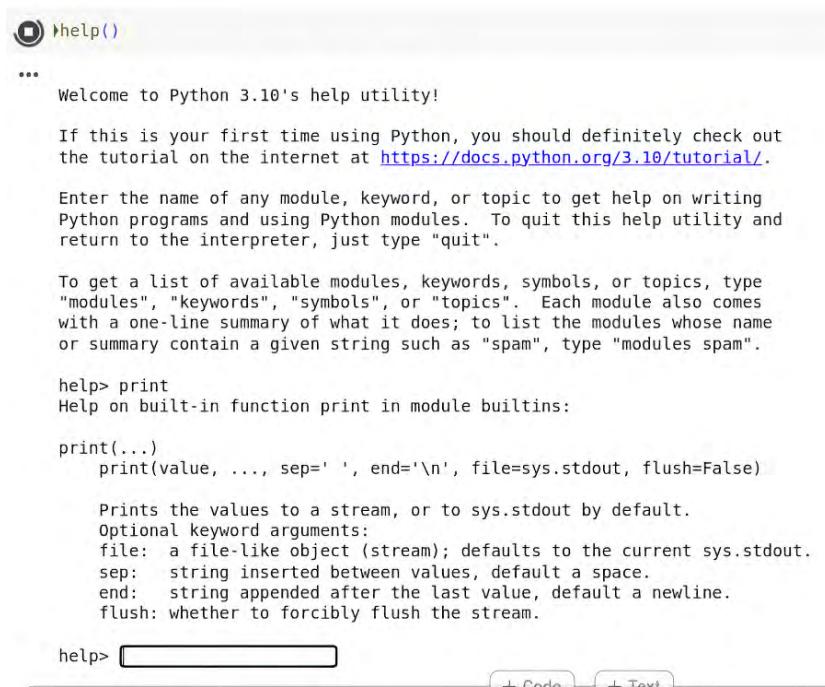
Figure 2.7 Error with Fix error option

It’s a good idea to examine the suggested fix carefully. While for a simple error like this the suggestion is sound, the more complex the code, the more chance there is that the fix won’t be reliable.

2.4 Using help and dir to explore Python

There are a couple of handy tools that can help you explore Python. The first is the `help()` function, which has two modes. You can execute `help()` in a code cell to enter

the help system, where you can get help on modules, keywords, or topics. When you're in the help system, you see a `help>` prompt, and you can enter a module name, such as `print` or some other topic, to read Python's documentation on that topic. When you want to leave the help system, just press Enter, with no other input. Figure 2.8 shows the result of entering `help()` and then entering `print` into the input box. The help for the `print` function is displayed, and a new `help>` prompt and input box appear below it.



The screenshot shows a terminal window with the Python help utility running. The title bar says "help()". The content area displays the following text:

```

...
*** Welcome to Python 3.10's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the internet at https://docs.python.org/3.10/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> print
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.

help> 
```

At the bottom of the window, there are two buttons: "+ Ende" and "+ Text".

Figure 2.8 Using `help()` for the `print` function

Usually, it's more convenient to use `help()` in a more targeted way. Entering a type or variable name as a parameter for `help()` gives you an immediate display of that object's documentation and exits:

```

x = 2
help(x)

Help on int object:

class int(object)
|   int(x=0) -> integer
|   int(x, base=10) -> integer
|
|   Convert a number or string to an integer, or return 0 if no arguments

```

```
| are given. If x is a number, return x.__int__(). For floating point
| numbers, this truncates towards zero.
|
| If x is not a number or if base is given, then x must be a string,
| bytes, or bytearray instance representing an integer literal in the...
(continues with the documentation for an int...)
```

Using `help()` in this way is handy for checking the exact syntax of a method or the behavior of an object.

The `help()` function is part of the `pydoc` library, which has several options for accessing the documentation built into Python libraries. Because every Python installation comes with complete documentation, you can have all the official documentation at your fingertips, even if you aren't online. See the appendix for more information on accessing Python's documentation.

The other useful function is `dir()`, which lists the objects in a particular namespace. Used with no parameters, it lists the current global namespace, but it can also list objects for a module or even a type:

```
dir()

['In',
 'Out',
 '_',
 '_',
 '_',
 '_',
 '__builtins__',
 '__builtin__',
 '__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_dh',
 '_i',
 '_i1',
 '_i2',
 '_ih',
 '_ii',
 '_iii',
 '_oh',
 'exit',
 'get_ipython',
 'quit',
 'x']

dir(int)

['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 '__ceil__',
 '__class__',
 '__contains__',
 '__del__',
 '__delattr__',
 '__delitem__',
 '__dir__',
 '__div__',
 '__divmod__',
 '__eq__',
 '__float__',
 '__floordiv__',
 '__ge__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__int__',
 '__le__',
 '__lt__',
 '__ne__',
 '__neg__',
 '__pos__',
 '__radd__',
 '__rmul__',
 '__str__',
 '__sub__',
 '__truediv__']
```

```
'__class__',
'__delattr__',
'__dir__',
'__divmod__',
'__doc__',
'__eq__',
'__float__',
'__floor__',
'__floordiv__',
'__format__',
'__ge__',
'__getattribute__',
... (continues with more items... ]
```

`dir()` is useful for finding out what methods and data are defined, for reminding yourself at a glance of all the members that belong to an object or module, and for debugging, because you can see what is defined where.

2.5 **Using AI tools to write Python code**

The rise of large language model-based tools, such as ChatGPT, CoPilot, and many others, is beginning to revolutionize how we write software. It is now possible to have these tools generate working code in response to a text prompt. Note, however, that while it is *possible*, it is not yet always *reliable*. I am one of many who believe that human understanding is the vital ingredient for good code but that experience using AI tools to generate code will be increasingly helpful in the future.

With that in mind, we will use some of these tools to generate code for the solutions for the lab exercises in this book, and we will discuss and evaluate the AI-generated solutions. This will give you a feel for what is possible, how to create prompts, and what the potential drawbacks are.

2.5.1 **Benefits of AI tools**

It's undeniable that AI code generation has developed to the point that it has several advantages. First, it's fast. Even fairly large chunks of code can be generated in less than a minute, so even if you spend some time crafting and entering a good prompt for the chatbot to work from, the overall amount of time you need to spend to get a fair bit of code is definitely less than what even a lightning-fast human can achieve.

Second, the code tends to be free of all of the little glitches we humans have in typing in code—a chatbot will probably spell things correctly, use punctuation correctly, etc.

But—and this is a significant “but”—there are some drawbacks to consider.

2.5.2 **Negatives of using AI tools**

One thing to consider is that AI bots use a lot of resources. In this age of concern for the climate and environment, many will be worried about the resource (particularly energy and water) consumption of machines behind these coding tools. One can only hope that future developments will reduce that consumption to sustainable levels.

Second, using an AI bot for coding means that you are by definition sharing your code and using other's code, and the most common mechanisms for that sharing are weak in terms of protecting privacy and intellectual property.

Third, not all of the AI code generators are free, and as companies decide to recover the costs of the resources consumed by their machines, it's likely that fewer will remain free of charge for basic users.

Finally, while the code produced is surprisingly good, it's far from perfect. The bugs and inefficiencies in AI-generated code are not always obvious and can take a trained eye to spot. Putting AI-generated code into production without careful review and testing is at least as dangerous as uncritically using a junior developer's code.

While none of these drawbacks are dealbreakers, they are all things that responsible organizations and developers need to consider before deciding to use generated code.

2.5.3 *AI options*

The most convenient AI code generator is the one offered in Colaboratory, which is currently free of charge for most users. For comparison, we will also discuss solutions generated by GitHub Copilot, which works best with Microsoft's VSCode IDE and a local version of Python. Copilot also requires a \$10 monthly subscription fee as of this writing. Given that you need to install Python and VSCode on your machine and pay the subscription, it may not be worth the trouble for the code examples and problems in this book. There are also other options, such as Codeium, that offer a free tier and work better with VSCode than with Colaboratory. Personally, if I were writing production code, I would prefer Copilot, but any of them can work for experimentation, and using an AI tool is purely optional—the results I obtained are all shared in the text and in the source notebooks.

Summary

- Python is available across a wide variety of operating systems, hardware platforms, and user interfaces.
- For ease of use and maintenance, Google Colaboratory is recommended for use with the Jupyter notebooks of example code for this book.
- You can access both the GitHub repository and Colaboratory without an account, but more features may be available with an account.
- Jupyter notebooks consist of two types of cells: formattable text cells and cells for executable code.
- Errors are usually reported in the same area as output below code cells.
- The `help()` and `dir()` functions can be useful in learning more about Python.
- AI tools can be used to generate Python code, and we will use and critique a couple of those tools when we discuss the solutions to the lab exercises later.



The quick Python overview

This chapter covers

- Surveying Python
- Using built-in data types
- Controlling program flow
- Creating modules
- Using object-oriented programming

The purpose of this chapter is to give you a basic feeling for the syntax, semantics, capabilities, and philosophy of the Python language. It has been designed to provide an initial perspective or conceptual framework to which you'll be able to add details as you encounter them in the rest of the book.

On an initial read, you needn't be concerned about working through and understanding the details of the code segments. You'll be doing fine if you pick up a bit of an idea about what's being done. The subsequent chapters walk you through the specifics of these features and don't assume previous knowledge. You can always return to this chapter and work through the examples in the appropriate sections as a review after you've read the later chapters.

3.1 Python synopsis

Python has several built-in data types, such as integers, floats, complex numbers, strings, lists, tuples, dictionaries, and file objects. These data types can be manipulated using language operators, built-in functions, library functions, or a data type's own methods.

Programmers can also define their own classes and instantiate their own class instances. These class instances can be manipulated by programmer-defined methods, as well as by the language operators and built-in functions for which the programmer has defined the appropriate special method attributes.

NOTE The Python documentation and this book use the term “object” to refer to instances of any Python data type, not just what many other languages would call “class instances.” This is because all Python objects are instances of one class or another.

Python provides conditional and iterative control flow through an `if-elif-else` construct along with `while` and `for` loops and a new structural pattern-matching `match-case` feature. It allows function definition with flexible argument-passing options. Exceptions (errors) can be raised by using the `raise` statement, and they can be caught and handled by using the `try-except-else-finally` construct.

Variables (or identifiers) don't have to be declared and can refer to any built-in data type, user-defined object, function, or module.

3.2 Built-in data types

Python has several built-in data types, from scalars, such as numbers and Booleans, to more complex structures, such as lists, dictionaries, and files.

NOTE In the examples, code is in normal monospace and `output` is in **bold monospace type**.

3.2.1 Numbers

Python's four number types are integers, floats, complex numbers, and Booleans:

- *Integers*—`-1`, `-3`, `42`, `355`, `8888888888888888`, `-7777777777` (integers aren't limited in size except by available memory)
- *Floats*—`3.0`, `31e12`, `-6e-4`
- *Complex numbers*—`3 + 2j`, `-4 - 2j`, `4.2 + 6.3j`
- *Booleans*—`True`, `False`

You can manipulate them by using the arithmetic operators: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `//` (division with truncation to integer), `**` (exponentiation), and `%` (modulus).

The following examples use integers:

```

x = 5 + 2 - 3 * 2
x

1

5 / 2

2.5          ← Float result using /

5 // 2
2           ← Result forced to
             integer using //

5 % 2

1

2 ** 8

256

1000000001 ** 3
1000000003000000003000000001           ← Integer size limited only
                                         by available memory

```

Division of integers with `/` results in a float, and division of integers with `//` results in truncation to an integer. Note that integers are of unlimited size; they grow as large as you need them to, limited only by the memory available. The following examples work with floats, which are based on the doubles in C:

```

x = 4.3 ** 2.4
x

33.13784737771648

3.5e30 * 2.77e45

9.695e+75

1000000001.0 ** 3

1.00000003e+27

```

These examples use complex numbers:

```

(3+2j) ** (2+3j)

(0.6817665190890336-2.1207457766159625j)

x = (3+2j) * (4+9j)
x           ← x assigned to a
             complex number

(-6+35j)

```

```
x.real
-6.0
```

Complex numbers consist of both a real element and an imaginary element, suffixed with `j`. In the preceding code, variable `x` is assigned to a complex number. You can obtain its “real” part by using the attribute notation `x.real` and obtain the “imaginary” part with `x.imag`.

Several built-in functions can operate on numbers. In addition, the library module `cmath` contains functions for complex numbers, and the library module `math` contains functions for the other three types:

```
round(3.49)           ← round called with 3.49
3
import math
math.ceil(3.49)       ← The math library module is imported,
                      and its ceil function is called.
4
```

Built-in functions are always available and are called by using a standard function-calling syntax. In the preceding code, `round` is called with a float as its input argument.

The functions in library modules are made available via the `import` statement. In the previous code, the `math` library module is imported, and its `ceil` function is called using attribute notation: `module.function(arguments)`.

The following examples use Booleans:

```
x = False
x
False
not x
True
y = True * 2           ← Boolean treated as integer 1
y
2
```

Other than their representation as `True` and `False`, Booleans behave like the numbers 1 (True) and 0 (False).

3.2.2 Lists

Python has a powerful built-in list type:

```
[]  
[1]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 12]
[1, "two", 3, 4.0, ["a", "b"], (5,6)]
```

A list **list** contains **ints**, **float**, **string**, **list**, and **tuple**.

A list can contain a mixture of other types as its elements, including strings, tuples, lists, dictionaries, functions, file objects, and any type of number. A list can be indexed from its beginning or end. You can also refer to a subsegment, or *slice*, of a list by using slice notation:

```
x = ["first", "second", "third", "fourth"]
x[0]
```

**Index from the front
using positive index
starting from 0**

```
x[2]
```

```
'third'
```

```
x[-1]
```

**Index from the back
using negative index
(-1 is the last element)**

```
'fourth'
```

```
x[-2]
```

```
'third'
```

```
x[1:-1]
```

```
['second', 'third']
```

```
x[0:3]
```

```
['first', 'second', 'third']
```

```
x[-2:-1]
```

```
['third']
```

```
x[:3]
```

```
['first', 'second', 'third']
```

```
x[-2:]
```

```
['third', 'fourth']
```

**A slice[m:n], where m is
the starting point and n is
the exclusive ending point**

**The slice starts at its
beginning (m) and goes
to list's end.**

Lists can be accessed by index from the front using positive indices (starting with 0 as the first element). Index from the back using negative indices (starting with -1 as the last element). Obtain a slice using [m:n], where m is the inclusive starting point and n is the exclusive ending point (see table 3.1). An [:n] slice starts at a list's beginning, and an [m:] slice goes to its end.

Table 3.1 List indices

x=	["first" ,	"second" ,	"third" ,	"fourth"]
Positive indices		0	1	2	3	
Negative indices		-4	-3	-2	-1	

You can use this notation to add, remove, and replace elements in a list or to obtain an element or a new list that's a slice from it:

```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
x[1] = "two"
x[8:9] = []
x
```

```
[1, 'two', 3, 4, 5, 6, 7, 8]
```

```
x[5:7] = [6.0, 6.5, 7.0]
x
```

← The size of x increases by 1.

```
[1, 'two', 3, 4, 5, 6.0, 6.5, 7.0, 8]
```

```
x[5:]
```

```
[6.0, 6.5, 7.0, 8]
```

The size of the list increases or decreases if the new slice is bigger or smaller than the slice it's replacing.

Some built-in functions (`len`, `max`, and `min`), some operators (`in`, `+`, and `*`), the `del` statement, and the list methods (`append`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse`, and `sort`) operate on lists:

```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
len(x)
```

```
9
```

```
[-1, 0] + x
```

← + and * each create a new list.

```
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
x.reverse()
x
```

← reverse() method called
by using x.reverse()

```
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

The operators `+` and `*` each create a new list, leaving the original unchanged. A list's methods are called by using attribute notation on the list itself: `x.method(arguments)`.

Some of these operations repeat functionality that can be performed with slice notation, but they improve code readability.

3.2.3 Tuples

Tuples are similar to lists but are *immutable*—that is, they can't be modified after they've been created. The operators (`in`, `+`, and `*`) and built-in functions (`len`, `max`, and `min`) operate on them the same way as they do on lists because none of them modify the original. Index and slice notation work the same way for obtaining elements or slices but can't be used to add, remove, or replace elements. Also, there are only two tuple methods: `count` and `index`. An important purpose of tuples is for use as keys for dictionaries. They're also more efficient to use when you don't need modifiability:

```
()                                     ← One-element tuple with comma
(1,)                                     ← Different types in one list
(1, 2, 3, 4, 5, 6, 7, 8, 12)
(1, "two", 3, 4.0, ["a", "b"], (5, 6))
```

A one-element tuple needs a comma. A tuple, like a list, can contain a mixture of other types as its elements, including strings, tuples, lists, dictionaries, functions, file objects, and any type of number.

A list can be converted to a tuple by using the built-in function `tuple`:

```
x = [1, 2, 3, 4]
tuple(x)

(1, 2, 3, 4)
```

Conversely, a tuple can be converted to a list by using the built-in function `list`:

```
x = (1, 2, 3, 4)
list(x)

[1, 2, 3, 4]
```

3.2.4 Strings

String processing is one of Python's strengths. There are many options for delimiting strings:

```
"A string in double quotes can contain 'single quote' characters."
'A string in single quotes can contain "double quote" characters.'
'''\\tA string which starts with a tab; ends with a newline character.\n'''
"""This is a triple double quoted string - triple quoted strings (single
or double quoted) are only kind that can contain real newlines."""
```

Strings can be delimited by single (`' '`), double (`" "`), triple single (`''' '''`), or triple double (`"""" """"`) quotations and can contain tab (`\t`) and newline (`\n`) characters.

Strings are also immutable. The operators and functions that work with them return new strings derived from the original. The operators (`in`, `+`, and `*`) and built-in functions (`len`, `max`, and `min`) operate on strings as they do on lists and tuples. Index and

slice notation works the same way for obtaining elements or slices but can't be used to add, remove, or replace elements.

Strings have several methods to work with their contents, and the `re` library module also contains functions for working with strings:

```
x = "live and      let \t    \tlive"
x.split()

['live', 'and', 'let', 'live']

x.replace("      let \t    \tlive", "enjoy life")

'live and enjoy life'

import re
regexpr = re.compile(r»[\t ]+»)
regexpr.sub(" ", x)

'live and let live'
```

← re module used to replace all spaces

The `re` module provides regular expression functionality. It provides more sophisticated pattern extraction and replacement capabilities than the `string` module.

The `print` function outputs strings. Other Python data types can be easily converted to strings and formatted:

```
e = 2.718
x = [1, "two", 3, 4.0, ["a", "b"], (5, 6)]
print("The constant e is:", e,
      "and the list x is:", x)
```

← list is automatically converted to string by `print()`.


```
The constant e is: 2.718 and the list x is:
[1, 'two', 3, 4.0,['a', 'b'], (5, 6)]
```

← f in front of a string creates an “f-string” with values in { }.


```
print(f"the value of e is: {e}")
```

← the value of e is: 2.718

Objects are automatically converted to string representations for printing. The `%` operator provides formatting capability similar to that of C's `sprintf`.

3.2.5 Dictionaries

Python's built-in dictionary data type provides associative array functionality implemented by using hash tables. The built-in `len` function returns the number of key-value pairs in a dictionary. The `del` statement can be used to delete a key-value pair. As is the case for lists, several dictionary methods (`clear`, `copy`, `get`, `items`, `keys`, `update`, and `values`) are available:

```
x = {1: "one", 2: "two"}
x["first"] = "one"
```

← Sets the value of a new key, "first", to "one"

```
x[("Delorme", "Ryan", 1995)] = (1, 2, 3)
list(x.keys())
[1, 2, 'first', ('Delorme', 'Ryan', 1995)]
x[1]
'one'
x.get(1, "not available")
'one'
x.get(4, "not available")
```

← Keys can be numbers, strings, tuples, or other immutable types.

'not available'

← get returns "not available" when a key isn't in a dictionary.

Keys must be of an immutable type, including numbers, strings, and tuples. Values can be any kind of object, including mutable types, such as lists and dictionaries. If you try to access the value of a key that isn't in the dictionary, a `KeyError` exception is raised. To avoid this error, the dictionary method `get` optionally returns a user-definable value when a key isn't in a dictionary.

3.2.6 Sets, frozensets

A `set` in Python is an unordered collection of objects, used in situations where membership and uniqueness in the set are the main things you need to know about that object. Sets behave as collections of dictionary keys without any associated values:

```
x = set([1, 2, 3, 1, 3, 5])
{x, 1 in x, True, 4 in x}
```

← Set created by using `set()` on a list

← Duplicates are removed.

← Used to check for membership in a set

False

You can create a set by using `set` on a sequence, like a list. When a sequence is made into a set, duplicates are removed. The `in` keyword is used to check for membership of an object in a set.

A `frozenset` is a set that is immutable. That means that after the set has been created with code like `x = frozenset([1, 2, 3, 1, 3, 5])`, it cannot be changed, so no elements can be added removed.

3.2.7 File objects

A file is accessed through a Python file object:

```

f = open("myfile", "w")
f.write("First line with necessary newline character\n")
44
f.write("Second line to write to the file\n")

33

f.close()
f = open("myfile", "r")
line1 = f.readline()
line2 = f.readline()
f.close()
print(line1, line2)

First line with necessary newline character
Second line to write to the file

import os
directory = os.getcwd()
print(directory)

/content

filename = os.path.join(directory, "myfile")
print(filename)

/content/myfile

f = open(filename, "r")
print(f.readline())

First line with necessary newline character

f.close()

```

The open statement creates a file object, in write ("w") mode.

Opens the same file again, this time in read ("r") mode

The os module provides functions for working with files and paths.

The directory shown will depend on your environment.

The join() method combines the directory and filenames into an absolute pathname.

The `open` statement creates a file object. Here, the file `myfile` in the current working directory is being opened in write (`"w"`) mode. After writing two lines to it and closing it, you open the same file again—this time in read (`"r"`) mode. The `os` module provides several functions for moving around the filesystem and working with the pathnames of files and directories. Here, you move to another directory. But by referring to the file by an absolute pathname, you’re still able to access it.

Several other input/output capabilities are available. As we’ll see later, you can use the built-in `input` function to prompt and obtain a string from the user. The `sys` library module allows access to `stdin`, `stdout`, and `stderr`. The `struct` library module provides support for reading and writing files that were generated by, or are to be used by, C programs. The Pickle library module delivers data persistence through the ability to easily read and write the Python data types to and from files.

3.3 Type hints in Python

Unlike many programming languages, Python by design does not use typed variables and return values. While this makes the language more flexible and readable, it means that in many cases the type of an object referred to by a variable, or needed as a parameter, or returned by a function or method is not always immediately obvious. While inadvertently mixing incompatible types of objects will cause a runtime exception in Python, it will not raise an error at compile time. Particularly for large projects, there are many times when having the types of objects more explicitly available would be useful. For this, Python has added *type hints*.

The type hinting notation can be read by type-checking tools like mypy, pyright, pyre, or pttype, as well as several common IDEs, to flag the use of an incompatible or unexpected type. While these tools can report the error, Python itself does *not* raise a runtime error if the type hints are not followed.

For examples of type hints and a fuller discussion of their use, refer to section 4.4 in the next chapter.

3.4 Control flow structures

Python has a full range of structures to control code execution and program flow, including common branching and looping structures.

3.4.1 Boolean values and expressions

Python has several ways of expressing Boolean values. The Boolean constant `False`, 0, the Python nil value `None`, and empty values (for example, the empty list `[]` or empty string `" "`) are all taken as `False`. The Boolean constant `True` and everything else is considered `True`.

You can create comparison expressions by using the comparison operators (`<`, `<=`, `==`, `>`, `>=`, `!=`, `is`, `is not`, `in`, `not in`) and the logical operators (`and`, `not`, `or`), which all return `True` or `False`.

3.4.2 The `if-elif-else` statement

The block of code after the first `True` condition (of an `if` or an `elif`) is executed. If none of the conditions is `True`, the block of code after the `else` is executed:

```
x = 5
if x < 5:
    y = -1
    z = 5
elif x > 5:
    y = 1
    z = 11
else:
    y = 0
    z = 10
print(x, y, z)
```

Optional elif and else clauses

Python uses indentation to delimit blocks.

The `elif` and `else` clauses are optional, and there can be any number of `elif` clauses. Python uses indentation to delimit blocks. No explicit delimiters, such as brackets or braces, are necessary. Each block consists of one or more statements separated by new-lines. All these statements must be at the same level of indentation. The output in the example would be `5 0 10`.

3.4.3 Structural pattern matching with `match`

Versions of Python from 3.10 have the `match` statement, which allows for flexible and powerful pattern matching, superficially similar to the `switch` statements familiar to C++ and Java programmers but with more complex and subtle options:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

We will discuss the many possible options for `match` in the chapters to come.

3.4.4 The `while` loop

The `while` loop is executed as long as the condition (which here is `x > y`) is True:

```
u, v, x, y = 0, 0, 100, 30
while x > y:
    u = u + y
    x = x - y
    if x < y + 2:
        v = v + x
        x = 0
    else:
        v = v + y + 2
        x = x - y - 2
    print(f"\{u=} {v}=")
print(u, v)
```

u and v are both assigned a value of 0, x is set to 100, and y to 30.

This is the loop block.

Prints values of u and v each time through the loop

The first line is a shorthand notation. Here, `u` and `v` are assigned a value of 0, `x` is set to 100, and `y` obtains a value of 30. In this example, the output would be `60 40`.

It's also possible for a loop to contain a `break` statement, which ends the loop and jumps to an `else` block if it exists. There can also be a `continue` statement, which aborts the current iteration of the loop and returns to the start of the loop block.

3.4.5 The for loop

The `for` loop is simple but powerful because it's possible to iterate over any iterable type, such as a list or tuple. Unlike in many languages, Python's `for` loop iterates over each of the items in a sequence, making it more of a `foreach` loop. The following loop finds the first occurrence of an integer that's divisible by 7:

```
item_list = [3, "string1", 23, 14.0, "string2", 49, 64, 70]
for x in item_list:
    if not isinstance(x, int):
        continue
    if not x % 7:
        print(f"found an integer divisible by seven: {x}")
        break
```

`x` is sequentially assigned each value in the list. If `x` isn't an integer, the rest of this iteration is aborted by the `continue` statement. Flow control continues with `x` set to the next item from the list. After the first appropriate integer is found, the loop is ended by the `break` statement. The output would be

```
found an integer divisible by seven: 49
```

Just as in `while` loops, a `break` will cause the loop to end, and the execution will jump to an `else` block if it exists or to the next line after the loop.

3.4.6 Function definition

Python provides flexible mechanisms for passing arguments to functions:

```
def funct1(x, y, z):
    value = x + 2*y + z**2
    if value > 0:
        return x + 2*y + z**2
    else:
        return 0

u, v = 3, 4
funct1(u, v, 2)

15

funct1(u, z=v, y=2)           ← Here, z and y are entered by name.

23

def funct2(x, y=1, z=1):
    return x + 2 * y + z ** 2   ← Function parameter with defaults

funct2(3, z=4)
```

21

```

def funct3(x, y=1, z=1, *tup):
    print((x, y, z) + tup)

funct3(2)
(2, 1, 1)

funct3(1, 2, 3, 4, 5, 6, 7, 8, 9)
(1, 2, 3, 4, 5, 6, 7, 8, 9)

def funct4(x, y=1, z=1, **kwargs):
    print(x, y, z, kwargs)
funct4(1, 2, m=5, n=9, z=3)
1 2 3 {'m': 5, 'n': 9}

```

With a *, this collects all extra positional arguments into a tuple.

With **, this collects all extra keyword arguments into a dictionary.

Functions are defined by using the `def` statement, with the `return` statement being used to return a value. This value can be of any type, and if no `return` statement is encountered, Python's `None` value is returned.

Function arguments can be entered either by position or by name (keyword). A special parameter with `*` can be defined that collects all extra positional arguments in a function call into a tuple. Likewise, a special parameter can be defined with `**` that collects all extra keyword arguments in a function call into a dictionary.

3.4.7 Exceptions

Exceptions (errors) can be caught and handled by using the `try-except-else-finally` compound statement. This statement can also catch and handle exceptions you define and raise yourself. Any exception that isn't caught causes the program to exit. The following code snippet shows basic exception handling; we will discuss more complex examples in our chapter dealing with exceptions:

```

class EmptyFileError(Exception):
    pass
file_names = ["myfile1.txt", "non_existent", "empty_file", "myfile2.txt"]
for file_name in file_names:
    try:
        file = open(file_name, 'r')
        line = file.readline()
        if line == "":
            file.close()
            raise EmptyFileError(f'{file_name}: is empty')
    except IOError as error:
        print(f'{file_name}: could not be opened: {error.strerror}')
    except EmptyFileError as error:
        print(error)
    else:

```

Exception type defined from the base `Exception` type

Block checked for `IOError` or `EmptyFileError`

An `IOError` might occur.

`EmptyFileError` raised

An optional `else` clause is executed if no exception occurs.

```

        print(f"{file_name}: {line}")
    finally:
        print("Done processing", file_name)

```

An optional finally clause is always executed at the end of the block.

Here, you define your own exception type inheriting from the base `Exception` type. If an `IOError` or `EmptyFileError` occurs during the execution of the statements in the `try` block, the associated `except` block is executed. This is where an `IOError` might be raised. Here, you raise the `EmptyFileError`. The `else` clause is optional; it's executed if no exception occurs in the `try` block. (Note that in this example, `continue` statements in the `except` blocks could have been used instead.) The `finally` clause is optional; it's executed at the end of the block whether an exception was raised or not.

3.4.8 Context handling using the `with` keyword

A more streamlined way of encapsulating the `try-except-finally` pattern is to use the `with` keyword and a context manager. Python defines context managers for things like file access, and it's possible for the developer to define custom context managers. One benefit of context managers is that they may (and usually do) have default cleanup actions defined, which always execute whether or not an exception occurs.

The following example shows opening and reading a file by using `with` and a context manager:

```

filename = "myfile1.txt"
with open(filename, "r") as file:
    for line in file:
        print(line)

```

Here, `with` establishes a context manager that wraps the `open` function and the block that follows. In this case, the context manager's predefined cleanup action closes the file, even if an exception occurs; so, as long as the expression in the first line executes without raising an exception, the file is always closed. That code is equivalent to the following code:

```

filename = "myfile1.txt"
try:
    file = open(filename, "r")
    for line in file:
        print(line)
except Exception as e:
    raise e
finally:
    file.close()

```

3.5 Module creation

It's easy to create your own modules, which can be imported and used in the same way as Python's built-in library modules. The example in the following listing is a

deliberately oversimplified module with one function that prompts the user to enter a filename and determines the number of times that words occur in this file.

Listing 3.1 File wo.py

```
"""wo module. Contains function: words_occur()"""
# interface functions
def words_occur():
    """words_occur() - count the occurrences of words in a file."""
    # Prompt user for the name of the file to use.
    file_name = input("Enter the name of the file: ")
    # Open the file, read it and store its words in a list.
    file = open(file_name, 'r')
    word_list = file.read().split()
    file.close()
    # Count the number of occurrences of each word in the file.
    occurs_dict = {}
    for word in word_list:
        # increment the occurrences count for this word
        occurs_dict[word] = occurs_dict.get(word, 0) + 1
    # Print out the results.
    print(f"File {file_name} has {len(word_list)} words")
    print(f"{'CA'}{len(occurs_dict)} are unique")
    print(occurs_dict)
if __name__ == '__main__':
    words_occur()
```

The code is annotated with several callout boxes:

- A box labeled "Docstring to document module" points to the first line of the code.
- A box labeled "Comment to explain code" points to the multi-line docstring.
- A box labeled "read returns all the characters in a file; split splits the string on whitespace." points to the line `word_list = file.read().split()`.
- A box labeled "Common form for running module as command with python wo.py at a command line" points to the final line `if __name__ == '__main__': words_occur()`.
- A box labeled "Function called by using the module.function" points to the line `wo.words_occur()` at the bottom of the page.

Documentation strings, or *docstrings*, are standard ways of documenting modules, functions, methods, and classes. Comments are anything beginning with a `#` character and should be used to explain something specific about the code. `read` returns a string containing all the characters in a file, and `split` returns a list of the words of a string “split out” based on whitespace. The final `if` statement allows the program to be run as a script by typing `python wo.py` at a command line.

If a file is in the current directory or in one of the directories on the module search path, which can be found in `sys.path`, it can be imported like any of the built-in library modules by using the `import` statement:

```
import wo
wo.words_occur()
```

Function called by using the module.function

This function is called by using the same attribute syntax used for library module functions.

Note that if you change the file `wo.py` on disk, `import` won’t bring your changes into the same interactive session. You use the `reload` function from the `imp` library in this situation:

```
import imp
imp.reload(wo)

<module 'wo'>
```

For larger projects, there is a generalization of the module concept called *packages*, which allows you to easily group modules in a directory or directory subtree and then import and hierarchically refer to them by using a `package.subpackage.module` syntax. This entails little more than creating a possibly empty initialization file for each package or subpackage.

3.6 Object-oriented programming

Python provides full support for object-oriented programming (OOP). Listing 3.2 is an example that might be the start of a simple shapes module for a drawing program. It's intended mainly to serve as a reference if you're already familiar with OOP. The callout notes relate Python's syntax and semantics to the standard features found in other languages.

Listing 3.2 File shape.py

```
"""shape module. Contains classes Shape, Square and Circle"""
class Shape:
    """Shape class: has method move"""
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move(self, deltaX, deltaY):
        self.x = self.x + deltaX
        self.y = self.y + deltaY
class Square(Shape):
    """Square Class: inherits from Shape"""
    def __init__(self, side=1, x=0, y=0):
        Shape.__init__(self, x, y)
        self.side = side
class Circle(Shape):
    """Circle Class: inherits from Shape and has method area"""
    pi = 3.14159
    def __init__(self, r=1, x=0, y=0):
        Shape.__init__(self, x, y)
        self.radius = r
    def area(self):
        """Circle area method: returns the area of the circle."""
        return self.radius * self.radius * self.pi
    def __str__(self):
        return f"Circle of radius {self.radius} at ({self.x}, {self.y})"
```

Class defined by class keyword

The instance initializer `__init__`

x and y are created and initialized here.

Methods defined by using the `def` keyword

Class Circle inherits from class Shape.

pi is a class variable, readable by all instances of the class.

Subclass calls the initializer of its base class.

The `__str__` method (used by the `print` function)

Classes are defined by using the `class` keyword. The instance initializer method (constructor) for a class is always called `__init__`, and instance variables `x` and `y` are created and initialized here. Methods, like functions, are defined by using the `def` keyword. The first argument of any method is by convention called `self`. When the method is invoked, `self` is set to the instance that invoked the method. Class `Circle` inherits from class `Shape` and is similar to, but not exactly like, a standard class variable. A subclass must, in its initializer, explicitly call the initializer of its base class. The `__str__` method returns a string version of the class and is used by the `print` function. Other special method attributes permit operator overloading or are employed by built-in methods such as the length (`len`) function.

Importing this file makes these classes available:

```
import shape
c1 = shape.Circle()
c2 = shape.Circle(5, 15, 20)
print(c1)

Circle of radius 1 at coordinates (0, 0)

print(c2)
Circle of radius 5 at coordinates (15, 20)

c2.area()

78.53974999999998
c2.move(5,6)
print(c2)

Circle of radius 5 at coordinates (20, 26)
```

The initializer is called when an instance is created.

The print function implicitly uses `__str__`.

Circle uses the move method of parent class Shape.

The initializer is implicitly called, and a circle instance is created. The `print` function implicitly uses the special `__str__` method. Here, you see that the `move` method of `Circle`'s parent class `Shape` is available. A method is called by using attribute syntax on the object instance: `object.method()`. The first (`self`) parameter is set implicitly.

Summary

- Python has a number of built-in data types, including integers, floating points, and complex numbers; lists and tuples; strings; dictionaries; sets and frozensets; and file objects.
- Python also allows the programmer to make optional non-enforced notations of the type expected for variables and function/method parameters and return values.
- Python has a rich set of control flow structures including `for` and `while` loops, `if-elif-else` structures, structural pattern matching, functions, exceptions, and context handlers.

- Modules are just files of Python code that can be used by loading them with the `import` command.
- Python is an object-oriented language—not only are the built-in data types objects, it's also easy to create custom classes and inherit from other classes.

Part 2

The essentials

I

n the chapters that follow, I show you the essentials of Python. I start from the absolute basics of what makes a Python program and move through Python's built-in data types and control structures, as well as defining functions and using modules. The last chapter of this part moves on to show you how to write stand-alone Python programs, manipulate files, handle errors, and use classes.

The absolute basics



This chapter covers

- Indenting and block structuring
- Differentiating comments
- Assigning variables
- Optional type hints
- Evaluating expressions
- Using common data types
- Getting user input
- Using correct Pythonic style

This chapter describes the absolute basics in Python: how to use assignments and expressions, how to use numbers and strings, how to indicate comments in code, and so forth. It starts with a discussion of how Python block structures its code, which differs from every other major language.

4.1 *Indentation and block structuring*

Python differs from most other programming languages because it uses whitespace and indentation to determine block structure (that is, to determine what constitutes

the body of a loop, the `else` clause of a conditional, and so on). Most languages use braces of some sort to do this. The following is C code that calculates the factorial of 9, leaving the result in the variable `r`:

```
/* This is C code */
int n, r;
n = 9;
r = 1;
while (n > 0) {
    r *= n;
    n--;
}
```

The braces delimit the body of the `while` loop, the code that is executed with each repetition of the loop. The code is usually indented more or less as shown, to make clear what's going on, but it could also be written as follows:

```
/* And this is C code with arbitrary indentation */
int n, r;
n = 9;
r = 1;
while (n > 0) {
r *= n;
n--;
}
```

The code still would execute correctly, even though it's rather difficult to read.

The Python equivalent is

```
# This is Python code.
n = 9
r = 1
while n > 0:
    r = r * n
    n = n - 1
```

Python also supports
C-style `r *= n`.

Python also supports `n -= 1`.

Python doesn't use braces to indicate code structure; instead, the indentation itself is used. The last two lines of the previous code are the body of the `while` loop because they come immediately after the `while` statement and are indented one level further than the `while` statement. If those lines weren't indented, they wouldn't be part of the body of the `while`.

Note that while it's more explicit and clearer to say `r = r * n`, it's also possible and good Pythonic style to use the shorter `r *= n` form—both are fine in Python.

Using indentation to structure code rather than braces may take some getting used to, but there are significant benefits:

- It's impossible to have missing or extra braces. You never need to hunt through your code for the brace near the bottom that matches the one a few lines from the top.

- The visual structure of the code reflects its real structure, which makes it easy to grasp the skeleton of the code just by looking at it.
- Python coding styles are mostly uniform. In other words, you’re unlikely to go crazy from dealing with someone’s idea of aesthetically pleasing code. Everyone’s code will look pretty much like yours.

You probably use consistent indentation in your code already, so this won’t be a big step. If you’re using IDLE or one of the most common coding editors or IDEs—Emacs, VIM, VS Code, or PyCharm, to name a few—it will automatically indent lines. You just need to backspace out of levels of indentation when that block is finished. One thing that may trip you up once or twice until you get used to it is the fact that the Python interpreter returns an error message if you have a space (or spaces) preceding the commands you enter in a Jupyter notebook cell or at a Python shell’s >>> prompt.

4.2 Differentiating comments

For the most part, anything following a # symbol in a Python file is a comment and is disregarded by the interpreter. The obvious exception is a # in a string, which is just a character of that string:

```
# Assign 5 to x
x = 5
x = 3           # Now x is 3
x = "# This is not a comment"
```

You’ll put comments in Python code frequently.

4.3 Variables and assignments

The most commonly used command in Python is assignment, which looks pretty close to what you might’ve used in other languages. Python code to create a variable called `x` and assign that variable to the value 5 is

```
x = 5
```

In Python, unlike in many other computer languages, neither a variable type declaration nor an end-of-line delimiter (like a ;) is necessary. The line is ended by the end of the line. Variables are created automatically when they’re first assigned.

Variable names are case sensitive and can include any alphanumeric character as well as underscores but must start with a letter or underscore. See section 4.11 for more guidance on the Pythonic style for creating variable names.

Variables in Python: Buckets or labels?

The name `variable` is somewhat misleading in Python; `name` or `label` would be more accurate. However, it seems that pretty much everyone calls variables `variables` at

(continued)

some time or another. Whatever you call them, you should know how they really work in Python.

A common, but inaccurate, explanation is that a variable is a container that stores a value, somewhat like a bucket. This would be reasonable for many programming languages (C, for example).

However, in Python variables aren't buckets. Instead, they're labels or tags that refer to objects in the Python interpreter's namespace. Any number of labels (or variables) can refer to the same object, and when that object changes, the value referred to by *all* of those variables also changes.

To see what this means, look at the following simple code:

```
a = [1, 2, 3]
b = a
c = b
b[1] = 5
print(a, b, c)
[1, 5, 3] [1, 5, 3] [1, 5, 3]
```

If you're thinking of variables as containers, this result makes no sense. How could changing the contents of one container simultaneously change the other two? However, if variables are just labels referring to objects, it makes sense that changing the object that all three labels refer to would be reflected everywhere.

If the variables are referring to constants or immutable values, this distinction isn't quite as clear:

```
a = 1
b = a
c = b
b = 5
print(a, b, c)
1 5 1
```

Because the objects they refer to can't change, the behavior of the variables in this case is consistent with either explanation. In fact, in this case, after the third line, `a`, `b`, and `c` all refer to the same unchangeable integer object with the value 1. The next line, `b = 5`, makes `b` refer to the integer object 5 but doesn't change the references of `a` or `c`.

Python variables can be set to any object, whereas in C and many other languages, variables can store only the type of value they're declared as. The following is perfectly legal Python code:

```
x = "Hello"
print(x)
```

```
Hello
```

```
x = 5
print(x)
```

```
5
```

`x` starts out referring to the string object "Hello" and then refers to the integer object 5. Of course, this feature can be abused, because arbitrarily assigning the same variable name to refer successively to different data types can make code confusing.

A new assignment overrides any previous assignments. The `del` statement deletes the variable (but not necessarily the object it was attached to). Trying to print the variable's object after deleting it results in an error, as though the variable had never been created in the first place:

```
x = 5
print(x)
```

```
5
```

```
del x
print(x)
```

```
-----
NameError                                 Traceback (most recent call last)
<ipython-input-2-260d38800877> in <cell line: 5>()
      3
      4 del x
----> 5 print(x)           ← Line where error occurs
```

```
NameError: name 'x' is not defined    ← Name of exception that occurred
```

Here, you have your first look at a *traceback*, which is printed when an error, called an *exception*, has been detected. Notice that when we try to access a variable *after* we've deleted it, there is a line of dashes followed by the name of the exception, a `NameError`. The traceback has a little arrow ----> pointing to the line with the error. The last line explains the exception that was detected, which in this case is a `NameError` exception because `x` is no longer defined or valid after its deletion.

In general, the full dynamic call structure of the existing function at the time of the error's occurrence is returned. If you're using another Python environment, you might obtain the same information with some small differences. The traceback could look something like the following:

```
Traceback (most recent call last):
  File "/home/naomi/Dropbox/QPB4/code/vscode/ch04.py", line 5, in <module>
    print(x)
    ^
Error location indicated with
a ^ under the problem
NameError: name 'x' is not defined
```

This format is a bit simpler, with just a ^ under the error on the next line, but the same information is there.

Chapter 14 describes this mechanism in more detail. A full list of the possible exceptions and what causes them is in the Python standard library documentation. Use the index to find any specific exception (such as `NameError`) you receive.

4.4 Optional type hints in Python

While Python does not *require* that you specify types for variables (and function parameters and return values, etc.), current versions of Python do *allow* you to do so. As mentioned in the previous chapter, in Python many times the type of an object referred to by a variable, or needed as a parameter, or returned by a function or method is not always immediately obvious.

Mixing incompatible types of objects will cause a runtime exception in Python, but it will not raise an error at compile time. Particularly for large projects, there are many times when having the types of objects more explicitly available would be useful. For this, Python has added *type hints*.

The type hinting notation can be read by type-checking tools like `mypy`, `pyright`, `pyre`, or `pytype`, as well as several common IDEs, to flag the use of an incompatible or unexpected type. While these tools can report the error, Python itself will *not* raise a runtime error if the type hints are not followed.

For example, we can create a simple function to add two numbers and use type hinting to indicate that it should only receive and return objects of type `int`:

```
def add_ints(x: int, y: int) -> int:
    return x + y
```

Parameters and return value noted as ints

Variable p marked as int but set to float

Function called with floats

The function definition specifies that both of the parameters and the return value will be of type `int`. In the first line of code after the function, we specify that the variable `p` should be of type `int` but then try to set it to a `float`. Then the function `add_ints` is called first with two integers and then with two `floats`.

If we run this code, Python will execute it without any warning or complaint. However, if we save this code in a file `test_types.py` and run the `mypy` checker on that file, three errors will be flagged:

Flags setting int variable p to a float

```
naomi@naomi-NUC:~$ mypy test_types.py
test_types.py:4: error: Incompatible types in assignment (expression has
  type "float", variable has type "int") [assignment]
```

```
test_types.py:8: error: Argument 1 to "add_ints" has incompatible type
  ↪ "float"; expected "int" [arg-type]
test_types.py:8: error: Argument 2 to "add_ints" has incompatible type
  ↪ "float"; expected "int" [arg-type]
Found 3 errors in 1 file (checked 1 source file)
```

Flags calling function with
two floats instead of ints

When you run the `mypy` type checker on this code, it reports an error with setting the variable `p` to `2.3`, not because `2.3` is a bad value for `p` but because the annotation indicated that it should be an `int`. It also flags two errors in the second call to `add_ints`, since both of the parameters are `floats`, but the function is marked as taking two `ints`. In both cases, Python will run the code without complaint, but the things marked by `mypy` indicate some confusion between intention and actual use, and that confusion is likely to lead to bugs later on.

4.4.1 Why use type hints?

One reason that type hints are increasingly used is that they help reduce confusion as to what type is expected, in many situations. In a large codebase, it can be particularly frustrating to hunt a bug where an unexpected type of object is returned. Type hints allow many such situations to be caught and fixed before runtime. Type information also allows many editors and IDEs to warn about such errors and suggest appropriate alternatives during the coding process.

4.4.2 Why not use type hints?

While type hints can be useful in many situations, there are situations where they may not be worth the trouble:

- In smaller, more informal scripts, it may not be worth the extra time and effort to add type hints.
- Trying to fully and explicitly type hint *everything* in a program may make function or method definitions harder to read. In that case, a decision needs to be made on the tradeoff between human readability and completeness of typing.
- Beware of going too far down the type-hinting rabbit hole. Spending too much time obsessed with detailed type hints may not be productive.
- Going back and changing working code solely to add type hints also may not be the best use of your time. It's usually wiser to leave working code alone unless you have a good reason to change it. Adding type hints can be done in the course of other maintenance or refactoring.

4.4.3 Progressive typing

Fortunately, since type hints are optional, you don't have to add type hints for everything all at once. If you have decided it's a good idea to use type hints, you can add them progressively, starting with the places where having the type information handy will do the most good.

In this text, we will not routinely use type hints in the short code examples, but we will use them in some of the longer examples and answers to the lab exercises.

4.5 Expressions

Python supports arithmetic and similar expressions; these expressions will be familiar to most readers. The following code calculates the average of 3 and 5, leaving the result in the variable `z`:

```
x = 3
y = 5
z = (x + y) / 2
```

Note that arithmetic operators involving only integers do *not* always return an integer. Even though all the values are integers, division (starting with Python 3) returns a floating-point number, so the fractional part isn't truncated. If you want traditional integer division returning an integer, you can use `//` instead.

Standard rules of arithmetic precedence apply. If you'd left out the parentheses in the last line, the code would've been calculated as `x + (y / 2)`.

Expressions don't have to involve just numerical values; strings, Boolean values, and many other types of objects can be used in expressions in various ways. I discuss these objects in more detail as they're used.

Try this: Variables and expressions

In a Jupyter notebook, create some variables. What happens when you try to put spaces, dashes, or other nonalphanumeric characters in the variable name? Play around with a few complex expressions, such as `x = 2 + 4 * 5 - 6 / 3`. Use parentheses to group the numbers in different ways and see how the result changes compared with the original ungrouped expression.

4.6 Strings

You've already seen that Python, like most other programming languages, indicates strings through the use of double quotes. This line leaves the string "Hello, World" in the variable `x`:

```
x = "Hello, World"
```

Backslashes can be used to escape characters—to give them special meanings. `\n` means the newline character, `\t` means the tab character, `\\"` means a single normal backslash character, and `\ "` is a plain double-quote character. It doesn't end the string:

```
x = "\tThis string starts with a \"tab\"."
x = "This string contains a single backslash(\\"\")."
```

You can use single quotes instead of double quotes. The following two lines do the same thing:

```
x = "Hello, World"  
x = 'Hello, World'
```

The only difference is that you don't need to backslash " characters in single-quoted strings or ' characters in double-quoted strings:

```
x = "Don't need a backslash"  
x = 'Can\'t get by without a backslash'  
x = "Backslash your \" character!"  
x = 'You can leave the " alone'
```

You can't split a normal string across lines. This code won't work:

```
# This Python code will cause an ERROR -- you can't  
# split the string across two lines.  
x = "This is a misguided attempt to  
put a newline into a string without using backslash-n"  
File "<ipython-input-3-dc07520d5086>", line 2  
    x = "This is a misguided attempt to  
        ^  
SyntaxError: unterminated string literal (detected at line 2)
```

But Python offers triple-quoted strings, which let you have multiline strings and include single and double quotes without backslashes:

```
x = """Starting and ending a string with triple " or ' characters  
permits embedded newlines, and the use of " and ' without  
backslashes"""
```

Now x is the entire sentence between the """ delimiters. [You can use triple single quotes ('') instead of triple double quotes to do the same thing.]

Python offers enough string-related functionality that chapter 6 is devoted to the topic.

4.7 Numbers

Because you're probably familiar with standard numeric operations from other languages, this book doesn't contain a separate chapter describing Python's numeric abilities. This section describes the unique features of Python numbers, and the Python documentation lists the available functions.

Python offers four kinds of numbers: *integers*, *floats*, *complex numbers*, and *Booleans*. An integer constant is written as an integer—0, -11, +33, 123456—and has unlimited range, restricted only by the resources of your machine. A float can be written with a decimal point or in scientific notation: 3.14, -2E-8, 2.718281828. The precision of these values is governed by the underlying machine but is typically equal to double (64-bit)

types in C. Complex numbers are probably of limited interest and are discussed separately later in the section. Booleans are either `True` or `False` and behave identically to 1 and 0 except for their string representations.

Arithmetic is much like it is in C. Operations involving two integers produce an integer, except for division (`/`), which results in a float. If the `//` division symbol is used, the result is an integer, rounding down. Operations involving a float always produce a float. Here are a few examples:

```
5 + 2 - 3 * 2
```

```
1
5 / 2
```

**Floating-point result
with normal division**

```
2.5
5 / 2.0
```

**Also a floating-point
result**

```
2.5
5 // 2
```

**Integer result when
divided using //**

```
2
30000000000
```

**This would be too large to be
an int in many languages.**

```
30000000000
```

```
30000000000 * 3
```

```
90000000000
```

```
30000000000 * 3.0
```

```
90000000000.0
```

```
2.0e-8
```

**Scientific notation
gives back a float.**

```
2e-08
```

```
3000000 * 3000000
```

```
9000000000000
```

```
int(200.2)
```

Converts a float to an int

```
200
```

```
int(2e2)
```

Converts an exponent to an int

```
200
```

```
float(200)
```

Converts an int to a float

```
200.0
```

The last three commands are explicit conversions between types. Note that converting from a float to an int will truncate the value and discard the decimal portion. On the other hand, converting an int to a float will always add a `.0`.

Numbers in Python have two advantages over C or Java: integers can be arbitrarily large, and the division of two integers results in a float.

4.7.1 Built-in numeric functions

Python provides the following number-related functions as part of its core:

```
abs, divmod, float, hex, int, max, min, oct,  
pow, round
```

See the documentation for the details of how each works.

4.7.2 Advanced numeric functions

More advanced numeric functions, such as the trig and hyperbolic trig functions, as well as a few useful constants, aren't built into Python but are provided in a standard module called `math`. I explain modules in detail later. For now, it's sufficient to know that you must make the `math` functions in this section available by starting your Python program or interactive session with the statement

```
from math import *
```

The `math` module provides the following functions and constants:

```
acos, asin, atan, atan2, ceil, cos, cosh, e, exp, fabs, floor, fmod,  
frexp, hypot, ldexp, log, log10, mod, pi, pow, sin, sinh, sqrt, tan,  
tanh
```

See the documentation for details.

4.7.3 Numeric computation

The core Python installation isn't well suited to intensive numeric computation because of speed constraints. But the powerful Python extension `NumPy` provides highly efficient implementations of many advanced numeric operations. The emphasis is on array operations, including multidimensional matrices and more advanced functions such as the fast Fourier transform. You should be able to find `NumPy` (or links to it) at www.scipy.org.

4.7.4 Complex numbers

Complex numbers are created automatically whenever an expression of the form `nj` is encountered, with `n` having the same form as a Python integer or float. `j` is, of course, standard notation for the imaginary number equal to the square root of `-1`, for example:

```
(3+2j)
```

```
(3+2j)
```

Note that Python expresses the resulting complex number in parentheses as a way of indicating that what's printed to the screen represents the value of a single object:

```
3 + 2j - (4+4j)
```

```
(-1-2j)
```

```
(1+2j) * (3+4j)
```

```
(-5+10j)
```

```
1j * 1j
```

```
(-1+0j)
```

Calculating `j * j` gives the expected answer of `-1`, but the result remains a Python complex-number object. Complex numbers are never converted automatically to equivalent real or integer objects. But you can easily access their real and imaginary parts with `real` and `imag`:

```
z = (3+5j)
z.real
```

```
3.0
```

```
z.imag
```

```
5.0
```

Note that real and imaginary parts of a complex number are always returned as floating-point numbers.

4.7.5 Advanced complex-number functions

The functions in the `math` module don't apply to complex numbers; the rationale is that most users want the square root of `-1` to generate an error, not an answer! Instead, similar functions, which can operate on complex numbers, are provided in the `cmath` module:

```
acos, acosh, asin, asinh, atan, atanh, cos, cosh, e, exp, log, log10,
pi, sin, sinh, sqrt, tan, tanh.
```

To make clear in the code that these functions are special-purpose complex-number functions and to avoid name conflicts with the more normal equivalents, it's best to import the `cmath` module with

```
import cmath
```

and then to explicitly refer to the `cmath` package when using the function:

```
import cmath  
cmath.sqrt(-1)
```

```
1j
```

Minimizing from <module> import *

This is a good example of why it's best to minimize the use of the `from <module> import *` form of the `import` statement. If you used it to import first the `math` module and then the `cmath` module, the commonly named functions in `cmath` would override those of `math`. It's also more work for someone reading your code to figure out the source of the specific functions you use. Some modules are explicitly designed to use this form of import.

See chapter 10 for more details on how to use modules and module names.

The important thing to keep in mind is that by importing the `cmath` module, you can do almost anything with complex numbers that you can do with other numbers.

Try this: Manipulating strings and numbers

In a Jupyter notebook, create some string and number variables (integers, floats, and complex numbers). Experiment a bit with what happens when you do operations with them, including across types. Can you multiply a string by an integer, for example, or can you multiply it by a float or complex number? Also load the `math` module and try a few of the functions; then load the `cmath` module and do the same. What happens if you try to use one of those functions on an integer or float after loading the `cmath` module? How might you get the `math` module functions back?

4.8 The None value

In addition to standard types such as strings and numbers, Python has a special basic data type that defines a single special data object called `None`. As the name suggests, `None` is used to represent an empty value. It appears in various guises throughout Python. For example, a procedure in Python is just a function that doesn't explicitly return a value, which means that, by default, it returns `None`.

`None` is often useful in day-to-day Python programming as a placeholder to indicate a point in a data structure where meaningful data will eventually be found, even though that data hasn't yet been calculated. You can easily test for the presence of `None` because there's only one instance of `None` in the entire Python system (all references to `None` point to the same object), and `None` is equivalent only to itself.

4.9 Getting input from the user

You can use the `input()` function to get input from the user. Use the prompt string you want to display to the user as `input`'s parameter:

```
name = input("Name? ")  
  
Name? Jane  
  
print(name)  
  
Jane  
age = int(input("Age? "))  
  
Age? 28  
  
print(age)  
  
28
```

 **Converts input from
a string to an int**

This is a fairly simple way to get user input. The one catch is that the input comes in as a string, so if you want to use it as a number, you have to use the `int()` or `float()` function to convert it to an int or float.

Try this: Getting input

Experiment with the `input()` function to get string and integer input. Using code similar to the previous code, what is the effect of not using `int()` around the call to `input()` for integer input? Can you modify that code to accept a float—say, 28.5? What happens if you deliberately enter the wrong type of value? Examples include a float in which an integer is expected and a string in which a number is expected—and vice versa.

4.10 Built-in operators

Python provides various built-in operators, from the standard (`+`, `*`, and so on) to the more esoteric, such as operators for performing bit shifting, bitwise logical functions, and so forth. Most of these operators are no more unique to Python than to any other language; hence, I won't explain them in the main text. You can find a complete list of the Python built-in operators in the documentation.

4.11 Basic Python style

Python has relatively few limitations on coding style with the obvious exception of the requirement to use indentation to organize code into blocks. Even in that case, the amount of indentation and type of indentation (tabs versus spaces) isn't mandated. However, there are preferred stylistic conventions for Python, contained in Python

Enhancement Proposal (PEP) 8, which is summarized in appendix A and available online at www.python.org/dev/peps/pep-0008/. A selection of Pythonic conventions is provided in table 4.1, but to fully absorb Pythonic style, periodically reread PEP 8.

Table 4.1 Pythonic coding conventions

Situation	Suggestion	Example
Module/package names	Short, all lowercase, underscores only if needed	imp, sys
Function names	All lowercase, underscores_for_readability	foo(), my_func()
Variable names	All lowercase, underscores_for_readability	my_var
Class names	CapitalizeEachWord	MyClass
Constant names	ALL_CAPS_WITH_UNDERSCORES	PI, TAX_RATE
Indentation	Four spaces per level, no tabs	
Comparisons	Don't compare explicitly to True or False.	if my_var: if not my_var:

I strongly urge you to follow the conventions of PEP 8. They're wisely chosen and time tested, and they'll make your code easier for you and other Python programmers to understand.

Quick check: Pythonic style

Which of the following variable and function names do you think are not good Pythonic style? Why?

```
bar(), varName, VERYLONGVARNAME, foobar, longvarname,
foo_bar(), really_very_long_var_name
```

Summary

- Python code is organized by using levels of indentation.
- Python comments can be marked with an initial # character.
- Strings in triple quotes at the beginning of a module, function, or method are “docstrings,” which explain what the object does and/or how it behaves.
- The intended types of variables, parameters, and function return values can optionally be marked using Python’s type hints.
- Python has the usual data types for strings and numbers, as well as data structures like lists, tuples, sets, and dictionaries.

- The `input()` function can be used to get input from the user as a string.
- Python has several built-in operators, including `+`, `-`, `/`, `//`, `%`, `*`, etc.
- The preferred Python style can be found in PEP 8 in the online Python documentation.

5

Lists, tuples, and sets

This chapter covers

- Manipulating lists and list indices
- Modifying lists
- Sorting
- Using common list operations
- Handling nested lists and deep copies
- Using tuples
- Creating and using sets

In this chapter, I discuss the two major Python sequence types: lists and tuples. At first, lists may remind you of arrays in many other languages, but don't be fooled: lists are a good deal more flexible and powerful than plain arrays.

Tuples are like lists that can't be modified; you can think of them as a restricted type of list or as a basic record type. I discuss the need for such a restricted data type later in the chapter. This chapter also discusses another Python collection type: sets. Sets are useful when an object's membership in the collection, as opposed to its position, is important.

Most of the chapter is devoted to lists, because if you understand lists, you pretty much understand tuples. The last part of the chapter discusses the differences between lists and tuples in both functional and design terms.

5.1 *Lists are like arrays*

A list in Python is similar to an array in Java or C or any other language; it's an ordered collection of objects. You create a list by enclosing a comma-separated list of elements in square brackets, as follows:

```
# This assigns a three-element list to x
x = [1, 2, 3]
```

Note that you don't have to worry about declaring the list or fixing its size ahead of time. This line creates the list as well as assigns it, and a list automatically grows or shrinks as needed.

Arrays in Python

A typed array module available in Python provides arrays based on C data types. Information on its use can be found in the documentation for the Python standard library. I suggest that you look into it only if you really need performance improvement. If a situation calls for numerical computations, you should consider using NumPy, mentioned in chapter 4 and available at www.scipy.org/.

Unlike lists in many other languages, Python lists can contain different types of elements; a list element can be any Python object. The following is a list that contains a variety of elements:

```
# First element is a number, second is a string, third is another list.
x = [2, "two", [1, 2, 3]]
```

Probably the most basic built-in list function is the `len` function, which returns the number of elements in a list:

```
x = [2, "two", [1, 2, 3]]
len(x)
```

3

Note that the `len` function doesn't count the items in the inner, nested list.

Quick check: `len()`

What would `len()` return for each of the following? `[0]; []; [[1, 3, [4, 5], 6], 7]`

5.2 List indices

Understanding how list indices work will make Python much more useful to you. Please read the whole section!

Elements can be extracted from a Python list by using a notation like C's array indexing. Like C and many other languages, Python starts counting from 0; asking for element 0 returns the first element of the list, asking for element 1 returns the second element, and so forth. The following are a few examples:

```
x = ["first", "second", "third", "fourth"]
x[0]

'first'

x[2]

'third'
```

But Python indexing is more flexible than C indexing. If indices are negative numbers, they indicate positions counting from the end of the list, with -1 being the last position in the list, -2 being the second-to-last position, and so forth. Continuing with the same list `x`, you can do the following:

```
a = x[-1]
a

'fourth'

x[-2]

'third'
```

For operations involving a single list index, it's generally satisfactory to think of the index as pointing at a particular element in the list. For more advanced operations, it's more correct to think of list indices as indicating positions *between* elements. In the list `["first", "second", "third", "fourth"]`, you can think of the indices as pointing as follows.

<code>x = [</code>		<code>"first",</code>		<code>"second",</code>		<code>"third",</code>		<code>"fourth"</code>		<code>]</code>
Positive indices	0		1		2		3			
Negative indices	-4		-3		-2		-1			

This is irrelevant when you're extracting a single element, but Python can extract or assign to an entire sublist at once—an operation known as *slicing*. Instead of entering `list[index]` to extract the item just after `index`, enter `list[index1:index2]` to

extract all items, including `index1` and up to (but not including) `index2`, into a new list. The following are some examples:

```
x = ["first", "second", "third", "fourth"]
x[1:-1]

['second', 'third']

x[0:3]

['first', 'second', 'third']

x[-2:-1]

['third']
```

It may seem reasonable that if the second index indicates a position in the list *before* the first index, this code would return the elements between those indices in reverse order, but this isn't what happens. Instead, this code returns an empty list:

```
x[-1:2]
[]
```

When slicing a list, it's also possible to leave out `index1` or `index2`. Leaving out `index1` means “Go from the beginning of the list,” and leaving out `index2` means “Go to the end of the list”:

```
x[:3]

['first', 'second', 'third']

x[2:]

['third', 'fourth']
```

Omitting both indices makes a new list that goes from the beginning to the end of the original list—that is, copies the list. This technique is useful when you want to make a copy that you can modify without affecting the original list:

```
y = x[:]
y[0] = '1 st'
y

['1 st', 'second', 'third', 'fourth']

x

['first', 'second', 'third', 'fourth']
```

Try this: List slices and indexes

Using what you know about the `len()` function and list slices, how would you combine the two to get the second half of a list when you don't know what size it is? Experiment in the Python shell to confirm that your solution works.

5.3 Modifying lists

You can use list index notation to modify a list as well as to extract an element from it. Put the index on the left side of the assignment operator:

```
x = [1, 2, 3, 4]
x[1] = "two"
x

[1, 'two', 3, 4]
```

Slice notation can be used here too. Saying something like `lista[index1:index2] = listb` causes all elements of `lista` between `index1` and `index2` to be replaced by the elements in `listb`. `listb` can have more or fewer elements than are removed from `lista`, in which case the length of `lista` is altered. You can use slice assignment to do several things, as shown in the following:

```
x = [1, 2, 3, 4]
x[len(x):] = [5, 6, 7]           ← Append a list to end of a list
x

[1, 2, 3, 4, 5, 6, 7]

x[:0] = [-1, 0]                  ← Append a list to front of a list
x

[-1, 0, 1, 2, 3, 4, 5, 6, 7]

x[1:-1] = []                     ← Removes elements from a list
x

[-1, 7]
```

Appending a single element to a list is such a common operation that there's a special `append` method for it:

```
x = [1, 2, 3]
x.append("four")
x

[1, 2, 3, 'four']
```

One problem can occur if you try to append one list to another. The list gets appended as a single element of the main list:

```
x = [1, 2, 3, 4]
y = [5, 6, 7]
x.append(y)
x

[1, 2, 3, 4, [5, 6, 7]]
```

The `extend` method is like the `append` method except that it allows you to add one list to another:

```
x = [1, 2, 3, 4]
y = [5, 6, 7]
x.extend(y)
x

[1, 2, 3, 4, 5, 6, 7]
```

There's also a special `insert` method to insert new list elements between two existing elements or at the front of the list. `insert` is used as a method of a list object and takes two additional arguments. The first additional argument is the index position in the list where the new element should be inserted, and the second is the new element itself:

```
x = [1, 2, 3]
x.insert(2, "hello")
x

[1, 2, 'hello', 3]

x.insert(0, "start")
x

['start', 1, 2, 'hello', 3]
```

`insert` understands list indices as discussed in section 5.2, but for most uses, it's easiest to think of `list.insert(n, elem)` as meaning `insert elem just before the nth element of the list`. `insert` is just a convenience method. Anything that can be done with `insert` can also be done with slice assignment. That is, `list.insert(n, elem)` is the same thing as `list[n:n] = [elem]` when `n` is nonnegative. Using `insert` makes for somewhat more readable code, and `insert` even handles negative indices:

```
x = [1, 2, 3]
x.insert(-1, "hello")
print(x)

[1, 2, 'hello', 3]
```

The `del` statement is the preferred method of deleting list items or slices. It doesn't do anything that can't be done with slice assignment, but it's usually easier to remember and easier to read:

```
x = ['a', 2, 'c', 7, 9, 11]
del x[1]
x

['a', 'c', 7, 9, 11]

del x[:2]
x

[7, 9, 11]
```

In general, `del list[n]` does the same thing as `list[n:n+1] = []`, whereas `del list[m:n]` does the same thing as `list[m:n] = []`.

The `remove` method isn't the inverse of `insert`. Whereas `insert` inserts an element at a specified location, `remove` looks for the first instance of a given value in a list and removes that value from the list:

```
x = [1, 2, 3, 4, 3, 5]
x.remove(3)
x

[1, 2, 4, 3, 5]

x.remove(3)
x

[1, 2, 4, 5]

x.remove(3)

-----
ValueError                                     Traceback (most recent call last)
<ipython-input-9-be7b9eddb459> in <cell line: 1>()
----> 1 x.remove(3)

ValueError: list.remove(x): x not in list
```

If `remove` can't find anything to remove, it raises an error. You can catch this error by using the exception-handling abilities of Python, or you can avoid the problem by using `in` to check for the presence of something in a list before attempting to remove it (see section 5.5.1 for examples of `in`).

The `reverse` method is a more specialized list modification method. It efficiently reverses a list in place:

```
x = [1, 3, 5, 6, 7]
x.reverse()
```

```
x
[7, 6, 5, 3, 1]
```

Try this: Modifying lists

Suppose that you have a list of 10 items. How might you move the last three items from the end of the list to the beginning, keeping them in the same order?

5.4 Sorting lists

Lists can be sorted by using the built-in Python `sort` method:

```
x = [3, 8, 4, 0, 2, 1]
x.sort()
x

[0, 1, 2, 3, 4, 8]
```

This method does an in-place sort—that is, changes the list being sorted. To sort a list without changing the original list, you have two options. You can use the `sorted()` built-in function, discussed in section 5.4.2, or you can make a copy of the list and sort the copy:

```
x = [2, 4, 1, 3]
y = x[:]
y.sort()
y

[1, 2, 3, 4]
```

A full list slice makes a copy of the list.

Sorts method on copy, not original

Note that here we used the `[:]` notation to make a slice of the entire list. Since slicing creates a new list, this in effect creates a copy of the entire list, which we can then sort using the `sort()` method.

Sorting works with strings too:

```
x = ["Life", "Is", "Enchanting"]
x.sort()
x

['Enchanting', 'Is', 'Life']
```

The `sort` method can sort just about anything because Python can compare just about anything. But there's one caveat in sorting: the default key method used by `sort`

requires all items in the list to be of comparable types. That means that using the `sort` method on a list containing both numbers and strings raises an exception:

```
x = [1, 2, 'hello', 3]
x.sort()

-----
TypeError                                     Traceback (most recent call last)
<ipython-input-8-9c6228d80c69> in <cell line: 2>()
      1 x = [1, 2, 'hello', 3]
----> 2 x.sort()
      3

TypeError: '<' not supported between instances of 'str' and 'int'
```

Conversely, you can sort a list of lists:

```
x = [[3, 5], [2, 9], [2, 3], [4, 1], [3, 2]]
x.sort()
x

[[2, 3], [2, 9], [3, 2], [3, 5], [4, 1]]
```

According to the built-in Python rules for comparing complex objects, the sublists are sorted first by the ascending first element and then by the ascending second element.

`sort` is even more flexible; it has an optional `reverse` parameter that causes the sort to be in reverse order when `reverse=True`, and it's possible to use your own key function to determine how elements of a list are sorted.

5.4.1 Custom sorting

To use custom sorting, you need to be able to define functions—something I haven't talked about yet. In this section, I also discuss the fact that `len(string)` returns the number of characters in a string. String operations are discussed more fully in chapter 6.

By default, `sort` uses built-in Python comparison functions to determine ordering, which is satisfactory for most purposes. At times, though, you want to sort a list in a way that doesn't correspond to this default ordering. Suppose you want to sort a list of words by the number of characters in each word, as opposed to the lexicographic sort that Python normally carries out.

To do this, write a function that returns the value, or key, that you want to sort on and use it with the `sort` method. That function in the context of `sort` is a function that takes one argument and returns the key or value that the `sort` function is to use.

For number-of-characters ordering, a suitable key function could be

```
def num_of_chars(string1):
    return len(string1)
```

This key function is trivial. It passes the length of each string back to the `sort` method, rather than the strings themselves.

After you define the key function, using it is a matter of passing it to the `sort` method by using the `key` keyword. Because functions are Python objects, they can be passed around like any other Python objects. Here's a small program that illustrates the difference between a default sort and your custom sort:

```
def comp_num_of_chars(string1):
    return len(string1)
word_list = ['Python', 'is', 'better', 'than', 'C']
word_list.sort()
print(word_list)

['C', 'Python', 'better', 'is', 'than']

word_list = ['Python', 'is', 'better', 'than', 'C']
word_list.sort(key=comp_num_of_chars)
print(word_list)

['C', 'is', 'than', 'Python', 'better']
```

The first list is in lexicographical order (with uppercase coming before lowercase), and the second list is ordered by ascending number of characters.

It's also possible to use an anonymous `lambda` function in the `sort()` call itself. This can be handy if the sort function is very short (which it usually is) and only used for sorting (again, as it should be). The code using the function `comp_num_of_chars` could also be written with a `lambda` function as

```
word_list.sort(key=lambda x: len(x))
```

The `lambda` is defined with three elements: a variable for the parameter, in this case `x`; a colon; and the return value, in this example `len(x)`. For simple functions with only a simple return value, a `lambda` function saves having to create more function names, and you can see how the sort key works without having to look at the function elsewhere.

Custom sorting is very useful, but if performance is critical, it may be slower than the default. Usually, this effect is minimal, but if the key function is particularly complex, the effect may be more than desired, especially for sorts involving hundreds of thousands or millions of elements.

One particular place to avoid custom sorts is where you want to sort a list in descending, rather than ascending, order. In this case, use the `sort` method's `reverse` parameter set to `True`. If for some reason you don't want to do that, it's still better to sort the list normally and then use the `reverse` method to invert the order of the resulting list. These two operations together—the standard sort and the reverse—will still be much faster than a custom sort.

5.4.2 *The sorted () function*

Lists have a built-in method to sort themselves, but other iterables in Python, such as the keys of a dictionary, don't have a `sort` method. Python also has the built-in function

`sorted()`, which returns a sorted list from any iterable. `sorted()` uses the same `key` and `reverse` parameters as the `sort` method:

```
x = (4, 3, 1, 2)
y = sorted(x)
y

[1, 2, 3, 4]

z = sorted(x, reverse=True)
z

[4, 3, 2, 1]
```

Try this: Sorting lists

Suppose that you have a list in which each element is in turn a list: `[[1, 2, 3], [2, 1, 3], [4, 0, 1]]`. If you wanted to sort this list by the second element in each list so that the result would be `[[4, 0, 1], [2, 1, 3], [1, 2, 3]]`, what function would you write to pass as the `key` value to the `sort()` method?

5.5 Other common list operations

Several other list methods are frequently useful, but they don't fall into any specific category.

5.5.1 List membership with the `in` operator

It's easy to test whether a value is in a list by using the `in` operator, which returns a Boolean value. You can also use the inverse: the `not in` operator:

```
3 in [1, 3, 4, 5]
True

3 not in [1, 3, 4, 5]
False

3 in ["one", "two", "three"]
False

3 not in ["one", "two", "three"]
True
```

5.5.2 List concatenation with the + operator

To create a list by concatenating two existing lists, use the `+` (list concatenation) operator, which leaves the argument lists unchanged:

```
z = [1, 2, 3] + [4, 5]
z

[1, 2, 3, 4, 5]
```

5.5.3 List initialization with the * operator

Use the `*` operator to produce a list of a given size, which is initialized to a given value. This operation is a common one for working with large lists whose size is known ahead of time. Although you can use `append` to add elements and automatically expand the list as needed, you obtain greater efficiency by using `*` to correctly size the list at the start of the program. A list that doesn't change in size doesn't incur any memory reallocation overhead:

```
z = [None] * 4
z

[None, None, None, None]
```

When used with lists in this manner, `*` (which in this context is called the *list multiplication operator*) replicates the given list the indicated number of times and joins all the copies to form a new list. This is the standard Python method for defining a list of a given size ahead of time. A list containing a single instance of `None` is commonly used in list multiplication, but the list can be anything:

```
z = [3, 1] * 2
z

[3, 1, 3, 1]
```

5.5.4 List minimum or maximum with min and max

You can use `min` and `max` to find the smallest and largest elements in a list. You'll probably use `min` and `max` mostly with numerical lists, but you can use them with lists containing any type of element. Trying to find the maximum or minimum object in a set of objects of different types causes an error if comparing those types doesn't make sense:

```
min([3, 7, 0, -2, 11])
-2

max([4, "Hello", [1, 2]])
```

```
-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-7-15ab1869d5d5> in <cell line: 1>()  
----> 1 max([4, "Hello", [1, 2]])  
  
TypeError: '>' not supported between instances of 'str' and 'int'
```

5.5.5 List search with index

If you want to find where in a list a value can be found (rather than wanting to know only whether the value is in the list), use the `index` method. This method searches through a list looking for a list element equivalent to a given value and returns the position of that list element:

```
x = [1, 3, "five", 7, -2]  
x.index(7)  
  
3  
  
x.index(5)  
  
-----  
ValueError                                     Traceback (most recent call last)  
<ipython-input-6-96ad5df81983> in <cell line: 1>()  
----> 1 x.index(5)  
  
ValueError: 5 is not in list
```

Attempting to find the position of an element that doesn't exist in the list raises an error, as shown here. This error can be handled in the same manner as the analogous error that can occur with the `remove` method (that is, by testing the list with `in` before using `index`).

5.5.6 List matches with count

`count` also searches through a list, looking for a given value, but it returns the number of times that the value is found in the list rather than positional information:

```
x = [1, 2, 2, 3, 5, 2, 5]  
x.count(2)  
  
3  
  
x.count(5)  
  
2  
  
x.count(4)  
  
0
```

5.5.7 Summary of list operations

You can see that lists are very powerful data structures, with possibilities that go far beyond those of plain old arrays. List operations are so important in Python programming that it's worth laying them out for easy reference, as shown in table 5.1.

Table 5.1 List operations

List operation	Explanation	Example
[]	Creates an empty list	x = []
len	Returns the length of a list	len(x)
append	Adds a single element to the end of a list	x.append('y')
extend	Adds another list to the end of the list	x.extend(['a', 'b'])
insert	Inserts a new element at a given position in the list	x.insert(0, 'y')
del	Removes a list element or slice	del(x[0])
remove	Searches for and removes a given value from a list	x.remove('y')
reverse	Reverses a list in place	x.reverse()
sort	Sorts a list in place	x.sort()
+	Adds two lists together	x1 + x2
*	Replicates a list	x = ['y'] * 3
min	Returns the smallest element in a list	min(x)
max	Returns the largest element in a list	max(x)
index	Returns the position of a value in a list	x.index('y')
count	Counts the number of times a value occurs in a list	x.count('y')
Sum	Sums the items (if they can be summed)	sum(x)
In	Returns whether an item is in a list	'y' in x

Being familiar with these list operations will make your life as a Python coder much easier.

Quick check: List operations

What would be the result of `len([[1,2]] * 3)`? What are two differences between using the `in` operator and a list's `index()` method? Which of the following will raise an exception? `min(["a", "b", "c"]); max([1, 2, "three"]); [1, 2, 3].count("one")`

Try this: List operations

If you have a list `x`, write the code to safely remove an item if—and only if—that value is in the list. Modify that code to remove the element only if the item occurs in the list more than once.

5.6 Nested lists and deep copies

This section covers another advanced topic that you may want to skip if you’re just learning the language.

Lists can be nested. One application of nesting is to represent two-dimensional matrices. The members of these matrices can be referred to by using two-dimensional indices. Indices for these matrices work as follows:

```
m = [[0, 1, 2], [10, 11, 12], [20, 21, 22]]
m[0]

[0, 1, 2]

m[0][1]

1

m[2]

[20, 21, 22]

m[2][2]

22
```

This mechanism scales to higher dimensions in the manner you’d expect.

Most of the time, this is all you need to concern yourself with. But you may run into a problem with nested lists—specifically, the way that variables refer to objects and how some objects (such as lists) can be modified (are mutable). An example is the best way to illustrate:

```
nested = [0]
original = [nested, 1]
original

[[0], 1]
```

Figure 5.1 shows what this example looks like.

Now the value in the nested list can be changed by using either the `nested` or the `original` variables:

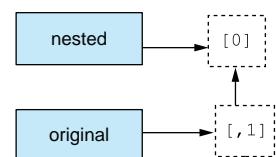


Figure 5.1 A list with its first item referring to a nested list

```

nested[0] = 'zero'
original

[['zero'], 1]

original[0][0] = 0
nested

[0]

original

[[0], 1]

```

But if `nested` is set to another list, the connection between them is broken:

```

nested = [2]
original

[[0], 1]

```

Figure 5.2 illustrates this condition.

You've seen that you can obtain a copy of a list by taking a full slice (that is, `x[:]`). You can also obtain a copy of a list by using the `+` or `*` operators (for example, `x + []` or `x * 1`). These techniques are slightly less efficient than the slice method. All three create what is called a *shallow* copy of the list, which is probably what you want most of the time. But if your list has other lists nested in it, you may want to make a *deep* copy. You can do this with the `deepcopy` function of the `copy` module:

```

original = [[0], 1]
shallow = original[:]
import copy
deep = copy.deepcopy(original)

```

See figure 5.3 for an illustration.

The lists pointed at by the `original` or `shallow` variables are connected. Changing the value in the nested list through either one of them affects the other:

```

shallow[1] = 2
shallow

[[0], 2]

original

```

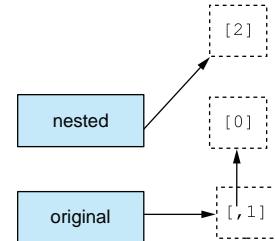


Figure 5.2 The first item of the original list is still a nested list, but the `nested` variable refers to a different list.

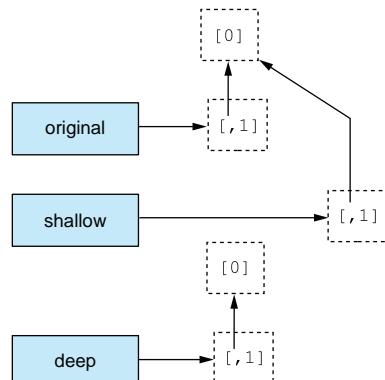


Figure 5.3 A deep copy copies nested lists.

```
[[0], 1]

shallow[0][0] = 'zero'
original

[['zero'], 1]
```

The deep copy is independent of the original, and no change to it has any effect on the original list:

```
deep[0][0] = 5
deep

[[5], 1]

original

[['zero'], 1]
```

This behavior is the same for any other nested objects in a list that are modifiable (such as dictionaries).

Now that you've seen what lists can do, it's time to look at tuples.

Try this: List copies

Suppose that you have the following list: `x = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]` What code could you use to get a copy `y` of that list in which you could change the elements *without* the side effect of changing the contents of `x`?

5.7 Tuples

Tuples are data structures that are very similar to lists, but they can't be modified; they can only be created. Tuples are so much like lists that you may wonder why Python bothers to include them. The reason is that tuples have important roles that can't be efficiently filled by lists, such as keys for dictionaries.

5.7.1 Tuple basics

Creating a tuple is similar to creating a list: assign a sequence of values to a variable. A list is a sequence that's enclosed by `[` and `]`; a tuple is a sequence that's enclosed by `(` and `)`:

```
x = ('a', 'b', 'c')
```

This line creates a three-element tuple.

After a tuple is created, using it is so much like using a list that it's easy to forget that tuples and lists are different data types:

```
x[2]
'c'
x[1:]
('b', 'c')
len(x)
3
max(x)
'c'
min(x)
'a'
5 in x
False
5 not in x
True
```

The main difference between tuples and lists is that tuples are immutable. An attempt to modify a tuple results in a confusing error message, which is Python's way of saying that it doesn't know how to set an item in a tuple:

```
x[2] = 'd'
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-2-dcc4b983047e> in <cell line: 1>()
----> 1 x[2] = 'd'

TypeError: 'tuple' object does not support item assignment
```

You can create tuples from existing ones by using the + and * operators:

```
x + x
('a', 'b', 'c', 'a', 'b', 'c')
2 * x
('a', 'b', 'c', 'a', 'b', 'c')
```

A “copy” (actually an alias pointing to the original object) of a tuple can be made in any of the same ways as for lists:

```
x[ : ]  
( 'a' , 'b' , 'c' )  
x * 1  
( 'a' , 'b' , 'c' )  
x + ()  
( 'a' , 'b' , 'c' )
```

Tuples themselves can't be modified. But if a tuple contains any mutable objects (for example, lists or dictionaries), these objects may be changed if they're still assigned to their own variables. Tuples that contain mutable objects aren't allowed as keys for dictionaries.

Copying tuples

Because tuples can't be modified, the shallow copy methods shown here don't return new objects but aliases pointing to the same object. This is true not only for tuples but also for string and byte objects, which are covered in the next chapter. In practice you can still think of these aliases as "copies" since there is nothing you could do with a true copy that you can't also do with an alias (and aliases are much faster and more memory efficient).

5.7.2 One-element tuples need a comma

A small syntactical point is associated with using tuples. Because the square brackets used to enclose a list aren't used elsewhere in Python, it's clear that [] means an empty list and that [1] means a list with one element. The same thing isn't true of the parentheses used to enclose tuples. Parentheses can also be used to group items in expressions to force a certain evaluation order. If you say (x + y) in a Python program, do you mean that x and y should be added and then put into a one-element tuple, or do you mean that the parentheses should be used to force x and y to be added before any expressions to either side come into play?

This situation is a problem only for tuples with one element because tuples with more than one element always include commas to separate the elements, and the commas tell Python that the parentheses indicate a tuple, not a grouping. In the case of one-element tuples, Python requires that the element in the tuple be followed by a comma to disambiguate the situation. In the case of zero-element (empty) tuples, there's no problem. An empty set of parentheses must be a tuple because it's meaningless otherwise:

```
x = 3  
y = 4  
(x + y) # This line adds x and y.
```

7

```
(x + y,) # Including a comma indicates that the parentheses denote a tuple.  
(7,)  
() # To create an empty tuple, use an empty pair of parentheses.  
()
```

5.7.3 **Packing and unpacking tuples**

As a convenience, Python permits tuples of variables to appear on the left side of an assignment operator, in which case variables in the tuple receive the corresponding values from the tuple on the right side of the assignment operator. The following is a simple example:

```
(one, two, three, four) = (1, 2, 3, 4)  
one  
  
1  
  
two  
  
2
```

This example can be written even more simply because Python recognizes tuples in an assignment context even without the enclosing parentheses. The values on the right side are packed into a tuple and then unpacked into the variables on the left side:

```
one, two, three, four = 1, 2, 3, 4
```

One line of code has replaced the following four lines of code:

```
one = 1  
two = 2  
three = 3  
four = 4
```

This technique is a convenient way to swap values between variables. Instead of saying

```
temp = var1  
var1 = var2  
var2 = temp
```

simply say

```
var1, var2 = var2, var1
```

To make things even more convenient, Python 3 has an extended unpacking feature, allowing an element marked with * to absorb any number of elements not matching the other elements. Again, some examples make this feature clearer:

```
x = (1, 2, 3, 4)
a, b, *c = x
a, b, c

(1, 2, [3, 4])

a, *b, c = x
a, b, c

(1, [2, 3], 4)

*a, b, c = x
a, b, c

([1, 2], 3, 4)

a, b, c, d, *e = x
a, b, c, d, e

(1, 2, 3, 4, [])
```

Note that the starred element receives all the surplus items as a list and that if there are no surplus elements, the starred element receives an empty list.

Packing and unpacking can also be performed with lists:

```
[a, b] = [1, 2]
[c, d] = 3, 4
[e, f] = (5, 6)
(g, h) = 7, 8
i, j = [9, 10]
k, l = (11, 12)
a

1

[b, c, d]

[2, 3, 4]

(e, f, g)

(5, 6, 7)

h, i, j, k, l

(8, 9, 10, 11, 12)
```

5.7.4 Converting between lists and tuples

Tuples can be easily converted to lists with the `list` function, which takes any sequence as an argument and produces a new list with the same elements as the original sequence. Similarly, lists can be converted to tuples with the `tuple` function, which does the same thing but produces a new tuple instead of a new list:

```
list((1, 2, 3, 4))
[1, 2, 3, 4]

tuple([1, 2, 3, 4])
(1, 2, 3, 4)
```

As an interesting side note, `list` is a convenient way to break a string into characters:

```
list("Hello")
['H', 'e', 'l', 'l', 'o']
```

This technique works because `list` (and `tuple`) apply to any Python sequence, and a string is just a sequence of characters. (Strings are discussed fully in chapter 6.)

Quick check: Tuples

Explain why the following operations aren't legal for the tuple `x = (1, 2, 3, 4)`:

```
x.append(1)
x[1] = "hello"
del x[2]
```

If you had a tuple `x = (3, 1, 4, 2)`, how might you end up with `x` sorted?

5.8 Sets

A *set* in Python is an unordered collection of objects used when membership and uniqueness in the set are the main things you need to know about that object. Like dictionary keys (discussed in chapter 7), the items in a set must be immutable and hashable. This means that ints, floats, strings, and tuples can be members of a set, but lists, dictionaries, and sets themselves can't.

5.8.1 Set operations

In addition to the operations that apply to collections in general, such as `in`, `len`, and iteration in `for` loops, sets have several set-specific operations:

```
x = set([1, 2, 3, 1, 3, 5])           ← Creates a set by using set on a list
x
```

```

{1, 2, 3, 5}           ← Set created, duplicates removed
x.add(6)
x

{1, 2, 3, 5, 6}        ← Uses add to add elements
x.remove(5)
x

{1, 2, 3, 6}           ← Uses remove to
                        remove elements
1 in x
True
4 in x
False

y = set([1, 7, 8, 9])  ← in checks to see if
                        an object is in a set.
x | y
{x, 7, 8, 9}           ← Uses | to get the union, or
                        combination, of two sets

{x, 7, 8, 9}           ← x & y
                        ← Uses & to get their intersection

{1}                     ← x ^ y
                        ← Uses ^ to find elements that are in
                        one set or the other but not both
{x, 7, 8, 9}

```

You can create a set by using `set` on a sequence, such as a list. When a sequence is made into a set, duplicates are removed. After creating a set by using the `set` function, you can use `add` and `remove` to change the elements in the set. The `in` keyword is used to check for membership of an object in a set. You can also use `|` to get the union, or combination, of two sets, `&` to get their intersection, and `^` to find their symmetric difference—that is, elements that are in one set or the other but not both.

These examples aren't a complete listing of set operations, but they are enough to give you a good idea of how sets work. For more information, refer to the official Python documentation.

5.8.2 Frozen sets

Because sets aren't immutable and hashable, they can't belong to other sets. To remedy that situation, Python has another set type, `frozenset`, which is just like a set but can't be changed after creation. Because frozen sets are immutable and hashable, they can be members of other sets:

```

x = set([1, 2, 3, 1, 3, 5])
z = frozenset(x)

```

```

z

frozenset({1, 2, 3, 5})

z.add(6)

-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-4-60917eecf5a3> in <cell line: 1>()
----> 1 z.add(6)

AttributeError: 'frozenset' object has no attribute 'add'

x.add(z)
x

{1, 2, 3, 5, frozenset({1, 2, 3, 5})}

```

Quick check: Sets

If you were to construct a set from the following list, how many elements would the set have? [1, 2, 5, 1, 0, 2, 3, 1, 1, (1, 2, 3)]

5.9 Lab: Examining a list

In this lab, the task is to read a set of temperature data (the monthly high temperatures at Heathrow Airport for 1948 through 2016) from a file and then find some basic information: the highest and lowest temperatures, the mean (average) temperature, and the median temperature (the temperature in the middle if all the temperatures are sorted).

The temperature data is in the file `lab_05.txt` in the source code directory for this chapter. Because I haven't yet discussed reading files, here's the code to read the files into a list:

```

temperatures = []
with open('lab_05.txt') as infile:
    for row in infile:
        temperatures.append(float(row.strip()))

```

You should find the highest and lowest temperature, the average, and the median. You'll probably want to use the `min()`, `max()`, `sum()`, `len()`, and `sort()` functions/methods. As a bonus, determine how many unique temperatures are in the list.

5.9.1 Why solve it the old-fashioned way?

You should have a try at creating a solution to this problem using your knowledge and the material presented in this chapter. The preceding code will open a file and read

its contents in as a list of floats. You can then use the functions mentioned to get the answers required. And for a bonus, the key is to think of how to convert a list so that only unique values remain.

You may be thinking, “Why should I write this code? Can’t I use AI to generate a solution?” And the answer is yes, you can, but trying to create a solution on your own first helps you both understand the problem better and learn how Python works. Both of those are vital for using an AI code generator—you need a solid understanding of the problem to create an effective prompt for the AI, and a solid understanding of how Python works is essential for evaluating the generated code.

If you are using the Jupyter notebook for this chapter, there is a cell with the code to load the file where you can put in your code to find the mean, median, etc. temperatures. Next, we’ll discuss a sample solution created by a human (me) and compare it to the AI solution.

5.9.2 Solving the problem with AI code generation

As mentioned in chapter 2, there are several AI code generation tools, and the landscape is rapidly evolving, so it’s quite possible you will be using something different from the options available as I write this. For the sake of illustration, I will use the code generator available in Google Colaboratory, which is currently available for free, and GitHub Copilot, which is available by subscription with a free trial and runs in Microsoft’s VS Code IDE. I’ll discuss one of the solutions and note if the other system produces something dramatically different.

Prompt creation

To generate code in Colaboratory, we need to click the `generate` link shown in an empty cell, “Start coding or generate with AI.” Once we do that, we get a code generation dialog waiting for a prompt telling the code generator what we want to do. Once we enter the prompt in the field after the `Using...` button (as shown in the following figure), we can click the `Generate` button to generate our code.



The code generation prompt in Colaboratory

Creating a prompt is an evolving art, but since the prompt field is limited, we can’t simply copy and paste the whole problem statement. Since we already have the code

(continued)

to read the file into a list as a series of floats, our prompt should focus only on what we need the code to do.

With this guidance, you can go ahead and try creating a prompt and generating some code. Try to evaluate the generated code and, if necessary, refine your prompt.

5.9.3 Solutions and discussion

To create our prompt, we focused on what we needed the code to do. The following is the list of things we needed done:

- 1 Use the list temperatures.
- 2 Find the high and low temperature.
- 3 Find the mean and median.
- 4 State how many unique temperatures there are (yes, let's do the bonus).

We might combine this into a prompt as

```
Using the list temperatures, find the high, low, mean, and median
temperatures, and show how many unique temperatures are in the list.
```

This prompt is complete and concise and covers exactly what we need. Remember that we already had the code to load the file, so we don't need to ask for that, but of course we do need to add that code to the generated code:

```
temperatures = []
with open('lab_05.txt') as infile:
    for row in infile:
        temperatures.append(float(row.strip()))
```

THE HUMAN CODED SOLUTION

The solution I came up with follows.

```
max_temp = max(temperatures)
min_temp = min(temperatures)
mean_temp = sum(temperatures)/len(temperatures)
# we'll need to sort to get the median temp
temperatures.sort()
median_temp = temperatures[len(temperatures)//2]
print(f"max = {max_temp}")
print(f"min = {min_temp}")
print(f"mean = {mean_temp}")
print(f"median = {median_temp}")
# Bonus
unique_temps = len(set(temperatures))

print(f"number of unique temps - {unique_temps}")
```

This solution does the job and reflects my bias to keep things simple and to keep comments to a minimum.

THE AI-GENERATED SOLUTION

The code generated by AI is remarkably similar—probably because the problem is not complex. Using Copilot (the Colaboratory code was nearly identical) we get

```
# High temperature
high_temp = max(temperatures)

# Low temperature
low_temp = min(temperatures)

# Mean temperature
mean_temp = sum(temperatures) / len(temperatures)

# Median temperature
sorted_temps = sorted(temperatures)
mid = len(sorted_temps) // 2
if len(sorted_temps) % 2 == 0:
    median_temp = (sorted_temps[mid - 1]
+ sorted_temps[mid]) / 2
else:
    median_temp = sorted_temps[mid]

# Unique temperatures
unique_temps = len(set(temperatures))

print("High temperature:", high_temp)
print("Low temperature:", low_temp)
print("Mean temperature:", mean_temp)
print("Median temperature:", median_temp)
print("Unique temperatures:", unique_temps)
```

This code also does the job, with what is arguably nicer formatting and more comments.

The AI version uses more whitespace, which makes the code easier to read. It also uses more comments—to my mind maybe more than needed. It's good style in Python to use comments only where needed to explain why something is as it is, and in my opinion a comment `# High temperature` right before the `high_temp` is not needed. On the other hand, the comment in the human version, `# we'll need to sort to get the median temp`, explains why we're sorting, which is lacking in the AI version. But you could argue that it's a question of preferences.

Besides some difference in printing the results, the main difference between the two is in finding the median. Again, the median is the value exactly in the middle of a series of sorted values, and the human version sorts the list in place and just picks the value in slot `len(temperatures) // 2`. The AI version makes a sorted copy of the list using the `sorted` function, and then, if the list has an even number of elements, meaning that there is no exact middle of the list, it extrapolates a value.

You might think that the AI version is superior: it has more code, and it preserves the original list and takes a more sophisticated view of the median. In real-world programming, however, things are not so clear. If the list of temperatures were several million lines long, then creating a sorted copy may not be an efficient use of memory, and it may well be that there is no need to preserve a version in the original order. It may also be the case that we don't want an extrapolated median but an actual value in the list, or that we don't care if the median value is actually from a slot one higher than the midpoint of the list. In that case, the simpler code would be preferable.

The main takeaway from this example is that while AI tools can generate working code that looks nice, it is important to keep in mind when analyzing the code both the problem to be solved and how the code functions to solve it.

Summary

- Lists and tuples are structures that embody the idea of a sequence of elements, as are strings.
- Lists are like arrays in other languages but with automatic resizing, slice notation, and many convenience functions.
- Tuples are like lists but can't be modified, so they use less memory and can be dictionary keys (see chapter 7).
- Sets are iterable collections, but they're unordered and can't have duplicate elements. Frozen sets are sets that can't be modified.
- AI tools can generate useful code, but it's important to evaluate that code in light of both the problem and how Python works.



Strings

This chapter covers

- Understanding strings as sequences of characters
- Using basic string operations
- Inserting special characters and escape sequences
- Converting from objects to strings
- Formatting strings
- Using the `bytes` type

Handling text—from user input to filenames to chunks of text to be processed—is a common chore in programming. Python comes with powerful tools to handle and format text. This chapter discusses the standard string and string-related operations in Python.

6.1 **Strings as sequences of characters**

For the purposes of extracting characters and substrings, strings can be considered to contain sequences of characters, which means that you can use index or slice notation:

```
x = "Hello"
x[0]

'H'

x[-1]

'o'

x[1:]

'ello'
```

One use for slice notation with strings is to chop the newline off the end of a string (usually, a line that's just been read from a file):

```
x = "Goodbye\n"
x = x[:-1]
x

'Goodbye'
```

This code is just an example. You should know that Python strings have other, better methods to strip unwanted characters, but this example illustrates the usefulness of slicing.

It's also worth noting that there is no separate character type in Python. Whether you use an index, slicing, or some other method, when you extract a single “character” from a Python string, it's still a one-character string, with the same methods and behavior as the original string. The same is true for the empty string "".

You can also determine how many characters are in the string by using the `len` function, which is used to find the number of elements in a list:

```
len("Goodbye")
7
```

But strings aren't lists of characters. The most noticeable difference between strings and lists is that, unlike lists, *strings can't be modified*. Attempting to say something like `string.append('c')` or `string[0] = 'H'` results in an error. You'll notice in the previous example that I stripped off the newline from the string by creating a string that was a slice of the previous one, not by modifying the previous string directly. This is a basic Python restriction, imposed for efficiency reasons.

6.2 Basic string operations

The simplest (and probably most common) way to combine Python strings is to use the string concatenation operator `+`:

```
x = "Hello " + "World"  
x  
  
'Hello World'
```

Python also has an analogous string multiplication operator that I've found to be useful sometimes but not often:

```
8 * "x"  
  
'xxxxxxxx'
```

6.3 Special characters and escape sequences

You've already seen a few of the character sequences that Python regards as special when used within strings: `\n` represents the newline character, and `\t` represents the tab character. Sequences of characters that start with a backslash and that are used to represent other characters are called *escape sequences*. Escape sequences are generally used to represent *special characters*—that is, characters (such as tab and newline) that don't have a standard one-character printable representation. This section covers escape sequences, special characters, and related topics in more detail.

6.3.1 Basic escape sequences

Python provides a brief list of two-character escape sequences to use in strings (see table 6.1). The same sequences also apply to `bytes` objects, which will be introduced at the end of this chapter.

Table 6.1 Escape sequences for string and byte literals

Escape sequence	Character represented
<code>\'</code>	Single-quote character
<code>\"</code>	Double-quote character
<code>\\"</code>	Backslash character
<code>\a</code>	Bell character
<code>\b</code>	Backspace character
<code>\f</code>	Form-feed character
<code>\n</code>	Newline character
<code>\r</code>	Carriage-return character (not the same as <code>\n</code>)
<code>\t</code>	Tab character
<code>\v</code>	Vertical tab character

The ASCII character set defines quite a few more special characters. These characters are accessed by the numeric escape sequences, described in the next section.

6.3.2 **Numeric (octal and hexadecimal) and Unicode escape sequences**

You can include any ASCII character in a string by using an octal (base 8) or hexadecimal (base 16) escape sequence corresponding to that character. An octal escape sequence is a backslash followed by three digits defining an octal number; the ASCII character corresponding to this octal number is substituted for the octal escape sequence. A hexadecimal escape sequence begins with `\x` rather than just `\` and can consist of any number of hexadecimal digits. The escape sequence is terminated when a character is found that's not a hexadecimal digit. For example, in the ASCII character table, the character *m* happens to have decimal value 109. The value of decimal 109 is octal value 155 and hexadecimal value 6D, so

```
'm'  
'm'  
'\155'  
'm'  
'\x6D'  
'm'
```

All three expressions represent a string containing the single character *m*. But these forms can also be used to represent characters that have no printable representation. The newline character `\n`, for example, has octal value 012 and hexadecimal value 0A:

```
'\n'  
'\n'  
'\012'  
'\n'  
'\x0A'  
'\n'
```

Because all strings in Python 3 are Unicode strings, they can also contain almost every character from every language available. Although a discussion of the Unicode system is far beyond the scope of this book, the following examples illustrate that you can also escape any Unicode character, either by number (as shown earlier) or by Unicode name:

```

unicode_a = '\N{LATIN SMALL LETTER A}'           ← Escapes by Unicode name
unicode_a

'a'          ← ASCII characters are
              Unicode characters.

unicode_a_with_acute = '\N{LATIN SMALL LETTER A WITH ACUTE}' 
unicode_a_with_acute

'á'          ← Escapes by number, using \u

"\u00E1"      ← Escapes by number, using \u
'á'

```

This code shows how you can access characters, including common ASCII characters, by using their Unicode names with `\N{Unicode name}` or by using their number with `\u`.

6.3.3 Printing vs. evaluating strings with special characters

I talked earlier about the difference between evaluating a Python expression interactively and printing the result of the same expression by using the `print` function. Although the same string is involved, the two operations can produce screen outputs that look different. A string that's evaluated at the top level of an interactive Python session is shown with all of its special characters as octal escape sequences, which makes clear what's in the string. Meanwhile, the `print` function passes the string directly to the terminal program, which may interpret special characters in special ways. The following is what happens with a string consisting of an `a` followed by a newline, a tab, and a `b`:

```

'a\n\tb'

'a\n\tb'

print('a\n\tb')

a
  b

```

In the first case, the newline and tab are shown explicitly in the string; in the second, they're used as newline and tab characters.

A normal `print` function also adds a newline to the end of the string. Sometimes (that is, when you have lines from files that already end with newlines), you may not want this behavior. Giving the `print` function an `end` parameter of `" "` causes the `print` function to suppress the final newline:

```

print("abc\n")

abc

```

```
print("abc\n", end=" ")
abc
```

6.4 *String methods*

Most of the Python string methods are built into the standard Python string class, so all string objects have them automatically. The standard `string` module also contains some useful constants. Modules are discussed in detail in chapter 10.

For the purposes of this section, you need only remember that most string methods are attached to the string object they operate on by a dot (`.`), as in `x.upper()`. That is, they're prepended with the string object followed by a dot. Because strings are immutable, the string methods are used only to obtain their return value and don't modify the string object they're attached to in any way.

I begin with those string operations that are the most useful and most commonly used; then I discuss some less commonly used but still useful operations. At the end of this section, I discuss a few miscellaneous points related to strings. Not all the string methods are documented here. See the documentation for a complete list of string methods.

6.4.1 *The split and join string methods*

Anyone who works with strings is almost certain to find the `split` and `join` methods invaluable. They're the inverse of one another: `split` returns a list of substrings in the string, and `join` takes a list of strings and puts them together to form a single string with the original string between each element. Typically, `split` uses whitespace as the delimiter of the strings it's splitting, but you can change that behavior via an optional argument.

String concatenation using `+` is useful but not efficient for joining large numbers of strings into a single string, because each time `+` is applied, a new string object is created. The previous Hello, World example produces three string objects, two of which are immediately discarded. A better option is to use the `join` function, which creates only one new string object:

```
" ".join(["join", "puts", "spaces", "between", "elements"])
'join puts spaces between elements'
```

By changing the string used to `join`, you can put anything you want between the joined strings:

"::".join(["Separated", "with", "colons"])
'Separated::with::colons'

You can even use an empty string, `" "`, to join elements in a list:

```
"".join(["Separated", "by", "nothing"])
'Separatedbynothing'
```

The most common use of `split` is probably as a simple parsing mechanism for string-delimited records stored in text files. By default, `split` splits on any whitespace, not just a single space character, but you can also tell it to split on a particular sequence by passing it an optional argument:

```
x = "You\t\t can have tabs\t\n\t and newlines \n\n " \
      "mixed in"
x.split()

['You', 'can', 'have', 'tabs', 'and', 'newlines', 'mixed', 'in']

x = "Mississippi"
x.split("ss")

['Mi', 'i', 'ippi']
```

Sometimes it's useful to permit the last field in a joined string to contain arbitrary text, perhaps including substrings that may match what `split` splits on when reading in that data. You can do this by specifying how many splits `split` should perform when it's generating its result, via an optional second argument. If you specify n splits, `split` goes along the input string until it has performed n splits (generating a list with $n + 1$ substrings as elements) or until it runs out of string. The following are some examples:

```
x = 'a b c d'
x.split(' ', 1)

['a', 'b c d']

x.split(' ', 2)

['a', 'b', 'c d']

x.split(' ', 9)

['a', 'b', 'c', 'd']
```

When using `split` with its optional second argument, you must supply a first argument. To get it to split on runs of whitespace while using the second argument, use `None` as the first argument.

I use `split` and `join` extensively, usually when working with text files generated by other programs. If you want to create more standard output files from your programs, good choices are the `csv` and `json` modules in the Python standard library.

Quick check: split and join

How could you use `split` and `join` to change all the whitespace in string `x` to dashes, such as changing "this is a test" to "this-is-a-test"?

6.4.2 Converting strings to numbers

You can use the functions `int` and `float` to convert strings to integer or floating-point numbers, respectively. If they're passed a string that can't be interpreted as a number of the given type, these functions raise a `ValueError` exception. Exceptions are explained in chapter 14.

In addition, you may pass `int` an optional second argument, specifying the numeric base to use when interpreting the input string:

```
float('123.456')
123.456

float('xxyy')

-----
ValueError                                Traceback (most recent call last)
<ipython-input-30-14b25c5b2052> in <cell line: 1>()
----> 1 float('xxyy')

ValueError: could not convert string to float: 'xxyy'

int('3333')
3333
int('123.456')                         Can't have decimal
                                         point in integer

-----
ValueError                                Traceback (most recent call last)
<ipython-input-32-ed4c46a302ea> in <cell line: 1>()
----> 1 int('123.456')

ValueError: invalid literal for int() with base 10: '123.456'

int('10000', 8)                         Interprets 10,000 as
                                         octal number
4096

int('101', 2)                           Binary number
5

int('ff', 16)                           Hexadecimal number
255
```

```
int('123456', 6)           ← Can't interpret 123,456 as base 6 number  
-----  
ValueError                  Traceback (most recent call last)  
<ipython-input-36-bdc1281d81c5> in <cell line: 1>()  
----> 1 int('123456', 6)  
  
ValueError: invalid literal for int() with base 6: '123456'
```

Did you catch the reason for that last error? I requested that the string be interpreted as a base 6 number, but the digit 6 can never appear in a base 6 number. Sneaky!

Quick check: Strings to numbers

Which of the following will not be converted to numbers, and why?

```
int('a1')  
int('12G', 16)  
float("12345678901234567890")  
int("12*2")
```

6.4.3 Getting rid of extra whitespace

A trio of surprisingly useful simple methods are the `strip`, `lstrip`, and `rstrip` functions. `strip` returns a new string that's the same as the original string, except that any whitespace at the *beginning or end* of the string has been removed. `lstrip` and `rstrip` work similarly, except that they remove whitespace only at the left or right end of the original string, respectively:

```
x = "Hello,      World\t\t"  
x.strip()  
  
'Hello,      World'  
  
x.lstrip()  
  
'Hello,      World\t\t'  
  
x.rstrip()  
  
'  Hello,      World'
```

In this example, tab characters are considered to be whitespace. The exact meaning may differ across operating systems, but you can always find out what Python considers to be whitespace by accessing the `string.whitespace` constant. On my Windows system, Python returns the following:

```
import string  
string.whitespace
```

```
' \t\n\r\x0b\x0c'
" \t\n\r\v\f"
' \t\n\r\x0b\x0c'
```

The characters given in backslashed hex (`\xnn`) format represent the vertical tab and form-feed characters. The space character is in there as itself. It may be tempting to change the value of this variable, to attempt to affect how `strip` and so forth work, but don't do it. Such an action isn't guaranteed to give you the results you're looking for.

You can, however, change which characters `strip`, `rstrip`, and `lstrip` remove by passing a string containing the characters to be removed as an extra parameter:

```
x = "www.python.org"
x.strip("w")           ────────── Strips off all w's
'.python.org'

x.rstrip("gor")        ────────── Strips off all g's, o's, and r's
'www.python.'          from right side of string

x.strip(".gorw")       ────────── Strips off all dots, g's, o's,
'python'               r's, and w's
```

Note that `strip` removes any and all of the characters in the extra parameter string, no matter in which order they occur.

The most common use for these functions is as a quick way to clean up strings that have just been read in. This technique is particularly helpful when you're reading lines from files because Python always reads in an entire line, including the trailing newline, if one exists. When you get around to processing the line read in, you typically don't want the trailing newline. `rstrip` is a convenient way to get rid of it.

There are also two new string methods to strip prefixes and suffixes:

```
x = "www.python.org"
x.removeprefix("www.")   ────────── Strips off only the
'python.org'            prefix "www"

x.removesuffix(".org")  ────────── Strips off only the
'www.python'           suffix ".org"
```

Unlike `strip`, `removesuffix` and `removeprefix` will only remove suffixes and prefixes that are exact matches for their parameters, which makes their behavior more predictable.

Quick check: strip

If the string `x` equals "(name, date), \n", which of the following would return a string containing "name, date"?

```
x.rstrip(") ,")  
x.strip(") ,\n")  
x.strip("\n) ,")
```

6.4.4 String searching

The string objects provide several methods to perform simple string searches. Before I describe them, though, I'll talk about another module in Python: `re`. (This module is discussed in depth in chapter 16.)

Another method for searching strings: The re module

The `re` module also does string searching but in a far more flexible manner, using *regular expressions*. Rather than search for a single specified substring, a `re` search can look for a string pattern. You could look for substrings that consist entirely of digits, for example.

Why am I mentioning this when `re` is discussed fully later? In my experience, many uses of basic string searches are inappropriate. You'd benefit from a more powerful searching mechanism but aren't aware that one exists, so you don't even look for something better. Perhaps you have an urgent project involving strings and don't have time to read this entire book. If basic string searching does the job for you, that's great. But be aware that you have a more powerful alternative.

The four basic string-searching methods are similar: `find`, `rfind`, `index`, and `rindex`. A related method, `count`, counts how many times a substring can be found in another string. I describe `find` in detail and then examine how the other methods differ from it.

`find` takes one required argument: the substring being searched for. `find` returns the position of the first character of the first instance of `substring` in the `string` object, or `-1` if `substring` doesn't occur in the string:

```
x = "Mississippi"  
x.find("ss")  
  
2  
  
x.find("zz")  
  
-1
```

`find` can also take one or two additional, optional arguments. The first of these arguments, if present, is an integer `start`; it causes `find` to ignore all characters before

position `start` in `string` when searching for `substring`. The second optional argument, if present, is an integer `end`; it causes `find` to ignore characters at or after position `end` in `string`:

```
x = "Mississippi"  
x.find("ss", 3)  
  
5  
  
x.find("ss", 0, 3)  
  
-1
```

`rfind` is almost the same as `find`, except that it starts its search at the end of `string` and so returns the position of the first character of the last occurrence of `substring` in `string`:

```
x = "Mississippi"  
x.rfind("ss")  
  
5
```

`rfind` can also take one or two optional arguments, with the same meanings as those for `find`.

`index` and `rindex` are identical to `find` and `rfind`, respectively, except for one difference: if `index` or `rindex` fails to find an occurrence of `substring` in `string`, it doesn't return `-1` but raises a `ValueError` exception. Exactly what this means will be clear after you read chapter 14.

`count` is used identically to any of the previous four functions but returns the number of nonoverlapping times the given substring occurs in the given string:

```
x = "Mississippi"  
x.count("ss")  
  
2
```

You can use two other string methods to search strings: `startswith` and `endswith`. These methods return a `True` or `False` result, depending on whether the string they're used on starts or ends with one of the strings given as parameters:

```
x = "Mississippi"  
x.startswith("Miss")  
  
True  
  
x.startswith("Mist")  
  
False
```

```
x.endswith("pi")
```

True

```
x.endswith("p")
```

False

Both `startswith` and `endswith` can look for more than one string at a time. If the parameter is a tuple of strings, both methods check for all the strings in the tuple and return `True` if any one of them is found:

```
x.endswith(("i", "u"))
```

True

`startswith` and `endswith` are useful for simple searches where you're sure that what you're checking for is at the beginning or end of a line.

Quick check: String searching

If you wanted to check whether a line ends with the string "rejected", what string method would you use? Would there be any other ways to get the same result?

6.4.5 Modifying strings

Strings are immutable, but string objects have several methods that can operate on a string and return a new string that's a modified version of the original string. This provides much the same effect as direct modification, for most purposes. You can find a more complete description of these methods in the documentation.

You can use the `replace` method to replace occurrences of `substring` (its first argument) in the string with `newstring` (its second argument). This method also takes an optional third argument (see the documentation for details):

```
x = "Mississippi"
x.replace("ss", "++")
'Mi++i++ippi'
```

Like the string search functions, the `re` module is a much more powerful method of substring replacement.

The functions `string.maketrans` and `string.translate` may be used together to translate characters in strings into different characters. Although rarely used, these functions can simplify your life when they're needed.

Suppose that you're working on a program that translates string expressions from one computer language into another. The first language uses `~` to mean logical not,

whereas the second language uses !; the first language uses ^ to mean logical, and the second language uses &; the first language uses (and), whereas the second language uses [and]. In a given string expression, you need to change all instances of ~ to !, all instances of ^ to &, and all instances of (to [, and all instances of) to]. You could do this by using multiple invocations of `replace`, but an easier and more efficient way is

```
x = "~x ^ (y % z)"
table = x.maketrans(~^(), "!&[ ]")
x.translate(table)

'!x & [y % z]'
```

The second line uses `maketrans` to make up a translation table from its two string arguments. The two arguments must each contain the same number of characters, and a table is made such that looking up the *n*th character of the first argument in that table gives back the *n*th character of the second argument.

Next, the table produced by `maketrans` is passed to `translate`. Then `translate` goes over each of the characters in its `string` object and checks to see whether they can be found in the table given as the second argument. If a character can be found in the translation table, `translate` replaces that character with the corresponding character looked up in the table to produce the translated string.

You can also use `translate` with an optional argument to specify characters that should be removed from the string. To remove characters, you would use empty strings for the first and second arguments and include a third string of characters to remove:

```
x = "~x ^ (y % z)"
table = x.maketrans("", "", "()~^%")
x.translate(table)

'x y z'
```

Here all of the punctuation marks in the original string are in the third parameter, a string containing the characters to remove.

Other functions in the `string` module perform more specialized tasks. `string.lower` converts all alphabetic characters in a string to lowercase, and `upper` does the opposite. `capitalize` capitalizes the first character of a string, and `title` capitalizes all words in a string. `swapcase` converts lowercase characters to uppercase and uppercase to lowercase in the same string. `expandtabs` gets rid of tab characters in a string by replacing each tab with a specified number of spaces. `ljust`, `rjust`, and `center` pad a string with spaces to justify it in a certain field width. `zfill` left-pads a numeric string with zeros. Refer to the documentation for details on these methods.

6.4.6 *Modifying strings with list manipulations*

Because strings are immutable objects, you have no way to manipulate them directly in the same way that you can manipulate lists. Although the operations that produce new

strings (leaving the original strings unchanged) can be useful, sometimes you want to be able to manipulate a string as though it were a list of characters. In that case, turn the string into a list of characters, do whatever you want, and then turn the resulting list back into a string:

```
text = "Hello, World"
wordList = list(text)
wordList[6:] = []
wordList.reverse()
text = "".join(wordList)
print(text)

,olleH
```

← **Removes everything after comma**

← **Joins with no space between**

You can also turn a string into a tuple of characters by using the built-in `tuple` function. To turn the list back into a string, use `" ".join()`.

You shouldn't go overboard with this method because it causes the creation and destruction of new `string` objects, which is relatively expensive. Processing hundreds or thousands of strings in this manner probably won't have much of an impact on your program; processing millions of strings probably will.

Quick check: Modifying strings

What would be a quick way to change all punctuation in a string to spaces?

6.4.7 Useful methods and constants

`string` objects also have several useful methods to report various characteristics of the string, such as whether it consists of digits or alphabetic characters or is all uppercase or lowercase:

```
x = "123"
x.isdigit()
```

True

```
x.isalpha()
```

False

```
x = "M"
x.islower()
```

False

```
x.isupper()
```

True

For a list of all the possible string methods, refer to the string section of the official Python documentation.

Finally, the `string` module defines some useful constants. You've already seen `string.whitespace`, which is a string made up of the characters Python thinks of as whitespace on your system. `string.digits` is the string '`'0123456789'`'. `string.hexdigits` includes all the characters in `string.digits`, as well as '`'abcdefABCDEF'`', the extra characters used in hexadecimal numbers. `string.octdigits` contains '`'01234567'`—only those digits used in octal numbers. `string.ascii_lowercase` contains all lowercase ASCII alphabetic characters; `string.ascii_uppercase` contains all ASCII uppercase alphabetic characters; `string.ascii_letters` contains all the characters in `string.ascii_lowercase` and `string.ascii_uppercase`. You might be tempted to try assigning to these constants to change the behavior of the language. Python would let you get away with this action, but it probably would be a bad idea.

Remember that strings are sequences of characters, so you can use the convenient Python `in` operator to test for a character's membership in any of these strings, although usually the existing string methods are simpler and easier. The most common string operations are shown in table 6.2.

Table 6.2 Common string operations

String operation	Explanation	Example
<code>+</code>	Adds two strings together	<code>x = "hello " + "world"</code>
<code>*</code>	Replicates a string	<code>x = " " * 20</code>
<code>upper</code>	Converts a string to uppercase	<code>x.upper()</code>
<code>lower</code>	Converts a string to lowercase	<code>x.lower()</code>
<code>title</code>	Capitalizes the first letter of each word in a string	<code>x.title()</code>
<code>find, index</code>	Searches for the target in a string	<code>x.find(y)</code> <code>x.index(y)</code>
<code>rfind, rindex</code>	Searches for the target in a string from the end of the string	<code>x.rfind(y)</code> <code>x.rindex(y)</code>
<code>startswith, endswith</code>	Checks the beginning or end of a string for a match	<code>x.startswith(y)</code> <code>x.endswith(y)</code>
<code>replace</code>	Replaces the target with a new string	<code>x.replace(y, z)</code>
<code>strip, rstrip, lstrip</code>	Removes whitespace or other characters from the ends of a string	<code>x.strip()</code>
<code>encode</code>	Converts a Unicode string to a <code>bytes</code> object	<code>x.encode("utf_8")</code>

Note that these methods don't change the string itself; they return either a location in the string or a string.

Try this: String operations

Suppose that you have a list of strings in which some (but not necessarily all) of the strings begin and end with the double-quote character:

```
x = ['"abc"', 'def', '"ghi"', '"klm"', 'nop']
```

What code would you use on each element to remove just the double quotes?

What code could you use to find the position of the last p in Mississippi? When you've found that position, what code would you use to remove just that letter?

6.5 Converting objects to strings

In Python, almost anything can be converted to some sort of a string representation by using the built-in `repr` function. Lists are the only complex Python data types you're familiar with so far, so here, I turn some lists into their representations:

```
repr([1, 2, 3])  
'[1, 2, 3]'  
  
x = [1]  
x.append(2)  
x.append([3, 4])  
'the list x is ' + repr(x)  
  
'the list x is [1, 2, [3, 4]]'
```

The example uses `repr` to convert the list `x` to a string representation, which is then concatenated with the other string to form the final string. Without the use of `repr`, this code wouldn't work. In an expression like `"string" + [1, 2] + 3`, are you trying to add strings, add lists, or just add numbers? Python doesn't know what you want in such a circumstance, so it does the safe thing (raises an error) rather than make any assumptions. In the previous example, all the elements had to be converted to string representations before the string concatenation would work.

Lists are the only complex Python objects that I've described to this point, but `repr` can be used to obtain some sort of string representation for almost any Python object. To see this, try `repr` around a built-in complex object, which is an actual Python function:

```
repr(len)  
'<built-in function len>'
```

Python hasn't produced a string containing the code that implements the `len` function, but it has at least returned a string—`<built-in function len>`—that describes what that function is. If you keep the `repr` function in mind and try it on each Python

data type (dictionaries, tuples, classes, and the like) in the book, you'll see that no matter what type of Python object you have, you can get a string that describes something about that object.

This is great for debugging programs. If you're in doubt about what's held in a variable at a certain point in your program, use `repr` and print out the contents of that variable.

I've covered how Python can convert any object to a string that describes that object. The truth is, Python can do this in either of two ways. The `repr` function always returns what might be loosely called the *formal string representation* of a Python object. More specifically, for simpler objects `repr` returns a string representation of a Python object from which the original object can be rebuilt. For large, complex objects, this may not be the sort of thing you want to see, so `repr` returns some descriptive text.

Python also provides the built-in `str` function. In contrast to `repr`, `str` is intended to produce *printable* string representations, and it can be applied to any Python object. `str` returns what might be called the *informal string representation* of the object. A string returned by `str` need not define an object fully and is intended to be read by humans, not by Python code.

You won't notice any difference between `repr` and `str` when you start using them, because until you begin using the object-oriented features of Python, there's no difference. `str` applied to any built-in Python object always calls `repr` to calculate its result. Only when you start defining your own classes does the difference between `str` and `repr` become important, as discussed in chapter 15.

So why talk about this now? I want you to be aware that there's more going on behind the scenes with `repr` than just being able to easily write `print` functions for debugging. As a matter of good style, you may want to get into the habit of using `str` rather than `repr` when creating strings for displaying information.

6.6 **Using the `format` method**

You can format strings in Python 3 in three ways. One way is to use the `string` class's `format` method. The `format` method combines a format string containing replacement fields marked with `{ }`, with replacement values taken from the parameters given to the `format` command. If you need to include a literal `{` or `}` in the string, you double it to `{}{ or }{}`. The `format` command is a powerful string-formatting mini-language that offers almost endless possibilities for manipulating string formatting. Conversely, it's fairly simple to use for the most common use cases, so I look at a few basic patterns in this section. Then, if you need to use the more advanced options, you can refer to the string-formatting section of the standard library documentation.

6.6.1 **The `format` method and positional parameters**

A simple way to use the `string` `format` method is with numbered replacement fields that correspond to the parameters passed to the `format` function:

```
"{0} is the {1} of {2}".format("Ambrosia", "food", "the gods")
```

'Ambrosia is the food of the gods'

```
"{{Ambrosia}} is the {0} of {1}".format("food", "the gods")
```

'{{Ambrosia}} is the food of the gods'

The format method is applied to the format string, which can also be a string variable.

Doubling the {} characters escapes them so that they don't mark a replacement field.

Note that the `format` method is applied to the format string, which can also be a string variable. Doubling the `{ }` characters escapes them so that they don't mark a replacement field.

This example has three replacement fields, `{0}`, `{1}`, and `{2}`, which are in turn filled by the first, second, and third parameters. No matter where in the format string you place `{0}`, it's always replaced by the first parameter, and so on.

You can also use named parameters.

6.6.2 The `format` method and named parameters

The `format` method also recognizes named parameters and replacement fields:

```
"{food} is the food of {user}".format(food="Ambrosia",
                                       user="the gods")
```

'Ambrosia is the food of the gods'

In this case, the replacement parameter is chosen by matching the name of the replacement field with the name of the parameter given to the `format` command.

You can also use both positional and named parameters, and you can even access attributes and elements within those parameters:

```
"{0} is the food of {user[1]}".format("Ambrosia",
                                         user=["men", "the gods", "others"])
```

'Ambrosia is the food of the gods'

In this case, the first parameter is positional, and the second, `user[1]`, refers to the second element of the named parameter `user`.

6.6.3 Format specifiers

Format specifiers let you specify the result of the formatting with even more power and control than the formatting sequences of the older style of string formatting. The format specifier lets you control the fill character, alignment, sign, width, precision, and type of the data when it's substituted for the replacement field. As noted earlier, the

syntax of format specifiers is a mini-language in its own right and too complex to cover completely here, but the following examples give you an idea of its usefulness:

```
"{0:10} is the food of gods".format("Ambrosia")      ← :10 makes the field 10 spaces wide, padded with spaces.
'Ambrosia    is the food of gods'

"{0:{1}} is the food of gods".format("Ambrosia", 10) ← :{1} width is from the second parameter.
'Ambrosia    is the food of gods'

"food:{width}" is the food of gods".format(food="Ambrosia", width=10)

'Ambrosia    is the food of gods'

"{0:>10} is the food of gods".format("Ambrosia")      ← :>10 forces right justification.
' Ambrosia is the food of gods'

'{0:&>10} is the food of gods".format("Ambrosia")      ← :&>10 forces right justification padded with & character.
'&&Ambrosia is the food of gods'
```

:10 is a format specifier that makes the field 10 spaces wide and pads with spaces. :{1} takes the width from the second parameter. :>10 forces right justification of the field and pads with spaces. :&>10 forces right justification and pads with & instead of spaces.

Quick check: The `format()` method

What will be in `x` when the following snippets of code are executed?

```
x = "{1:{0}}".format(3, 4)
x = "{0:$>5}".format(3)
x = "{a:{b}}".format(a=1, b=5)
x = "{a:{b}}:{0:$>5}".format(3, 4, a=1, b=5, c=10)
```

6.7 *String interpolation with f-strings*

Starting in Python 3.6, the newest way to create strings is called *f-strings*. F-strings, as they're commonly called because they are prefixed with `f`, are a way to include the values of arbitrary Python expressions inside literal strings. They use curly braces ("{}") to include and evaluate Python expressions. The syntax of f-strings is almost identical to that of the `format` method, but without an explicit call to `format()` they are more compact and easier to read. Thanks to their simplicity, they have become very common.

To use an f-string, you simply put an "f" right before the first quote and then include the Python expressions you want interpolated into the string in curly braces. The following examples give you a basic idea of how f-strings work:

```

value = 42
message = f"The answer is {value}"
print(message)

The answer is 42

primes = [1, 2, 3, 5]
f"sum of first 4 primes is {sum(primes)}"

'sum of first 4 primes is 11'

```

Just as with the format method, format specifiers may be added:

```

PI = 3.1415
print(f"PI is {PI:{10}.{2}}")

PI is      3.1

```

Another useful feature of f-strings is that adding an “=” after an expression in curly braces will give you “debugging” output by showing both the expression and its value:

```

print(f"PI={:10}.{2}")
PI=      3.1

primes = [1, 2, 3, 5]
f"sum of first 4 primes is {sum(primes)=}"
'sum of first 4 primes is sum(primes)=11'

```

All of the format specifiers that work for the format method described here should also work for f-strings. While f-strings were a bit limited in their early versions, they have proved so useful as a concise and readable way of creating strings with the values of variables and expressions that they have been extended and are currently used extensively.

6.8 Formatting strings with %

This section covers formatting strings with the *string modulus* (%) operator. This operator is used to combine Python values into formatted strings for printing or other use. C users will notice a strange similarity to the printf family of functions. The use of % for string formatting is the old style of string formatting, and I cover it here because it was the standard in earlier versions of Python, and you’re likely to see it in code that’s been ported from earlier versions of Python or was written by coders who are familiar with those versions. This style of formatting shouldn’t be used in new code, however, because it’s slated to be deprecated and then removed from the language in the future.

The following is an example:

```
"%s is the %s of %s" % ("Ambrosia", "food", "the gods")
'Ambrosia is the food of the gods'
```

The string modulus operator (the last % that occurs in the line, not the three instances of %s that come before it in the example) takes two parts: the left side, which is a string, and the right side, which is a tuple. The string modulus operator scans the left string for special *formatting sequences* and produces a new string by substituting the values on the right side for those formatting sequences, in order. In this example, the only formatting sequences on the left side are the three instances of %s, which stands for “Stick a string in here.”

Passing in different values on the right side produces different strings:

```
"%s is the %s of %s" % ("Nectar", "drink", "gods")
'Nectar is the drink of gods'

"%s is the %s of the %s" % ("Brussels Sprouts", "food",
    "foolish")
'Brussels Sprouts is the food of the foolish'
```

The members of the tuple on the right have `str` applied to them automatically by %s, so they don’t have to already be strings:

```
x = [1, 2, "three"]
"The %s contains: %s" % ("list", x)

"The list contains: [1, 2, 'three']"
```

6.8.1 Using formatting sequences

All formatting sequences are substrings contained in the string on the left side of the central %. Each formatting sequence begins with a percent sign and is followed by one or more characters that specify what is to be substituted for the formatting sequence and how the substitution is to be accomplished. The %s formatting sequence used previously is the simplest formatting sequence; it indicates that the corresponding string from the tuple on the right side of the central % should be substituted in place of the %s.

Other formatting sequences can be more complex. The following sequence specifies the field width (total number of characters) of a printed number to be 6, specifies the number of characters after the decimal point to be 2, and left-justifies the number in its field. I’ve put this formatting sequence in angle brackets so you can see where extra spaces are inserted into the formatted string:

```
"Pi is <%-6.2f>" % 3.14159 # use of the formatting sequence: %-6.2f
'Pi is <3.14  >'
```

All the options for characters that are allowable in formatting sequences are given in the documentation. There are quite a few options, but none is particularly difficult to use. Remember that you can always try a formatting sequence interactively in Python to see whether it does what you expect it to do.

6.8.2 Named parameters and formatting sequences

Finally, one additional feature available with the % operator can be useful in certain circumstances. Unfortunately, to describe it, I have to employ a Python feature that I haven't yet discussed in detail: *dictionaries*, commonly called *hash tables* or *associative arrays* in other languages. You can skip ahead to chapter 7 to learn about dictionaries; skip this section for now and come back to it later; or read straight through, trusting the examples to make things clear.

Formatting sequences can specify what should be substituted for them by name rather than by position. When you do this, each formatting sequence has a name in parentheses immediately following the initial % of the formatting sequence, like so:

"%(pi).2f" ← Note the name in parentheses.

In addition, the argument to the right of the % operator is no longer given as a single value or tuple of values to be printed but as a dictionary of values to be printed, with each named formatting sequence having a correspondingly named key in the dictionary. Using the previous formatting sequence with the string modulus operator, you might produce code like the following:

```
num_dict = {'e': 2.718, 'PI': 3.14159}
print("%(PI).2f - %(PI).4f - %(e).2f" % num_dict)
```

3.14 - 3.1416 - 2.72

This code is particularly useful when you're using format strings that perform a large number of substitutions, because you no longer have to keep track of the positional correspondences of the right-side tuple of elements with the formatting sequences in the format string. The order in which elements are defined in the dict argument is irrelevant, and the template string may use values from dict more than once (as it does with the pi entry).

Controlling output with the print function

Python's built-in `print` function also has some options that can make handling simple string output easier. When used with one parameter, `print` prints the value and a new-line character, so that a series of calls to `print` prints each value on a separate line:

```
print("a")
print("b")
```

(continued)

```
a
b
```

But `print` can do more. You can also give the `print` function several arguments, and those arguments are printed on the same line, separated by spaces and ending with a newline:

```
print("a", "b", "c")
a b c
```

If that's not quite what you need, you can give the `print` function additional parameters to control what separates each item and what ends the line:

```
print("a", "b", "c", sep="|")
a|b|c
print("a", "b", "c", end="\n\n")
a b c
```

Finally, the `print` function can be used to print to files as well as console output:

```
print("a", "b", "c", file=open("testfile.txt", "w"))
```

Using the `print` function's options gives you enough control for simple text output, but more complex situations are best served by using the `format` method.

Quick check: Formatting strings with %

What would be in the variable `x` after the following snippets of code have executed?

```
x = "%.2f" % 1.1111
x = "%(a).2f" % {'a':1.1111}
x = "%(a).08f" % {'a':1.1111}
```

6.9 Bytes

A `bytes` object is similar to a `string` object but with an important difference: A `string` is an immutable sequence of Unicode characters, whereas a `bytes` object is a sequence of integers with values from 0 to 256. Bytes can be necessary when you're dealing with binary data, such as reading from a binary data file.

The key thing to remember is that `bytes` objects may look like strings, but they can't be used exactly like strings or combined with strings:

```
unicode_a_with_acute = '\N{LATIN SMALL LETTER A WITH ACUTE}'
unicode_a_with_acute
```

```
'á'
```

```

xb = unicode_a_with_acute.encode()
xb
b'\xc3\xa1'          ← The encoded bytes object is 2 bytes;
                     it doesn't print as a character.

xb += 'A'            ← bytes string objects are different types.

-----
TypeError           Traceback (most recent call last)
<ipython-input-93-2c4ecee004ef> in <cell line: 1>()
----> 1 xb += 'A'

TypeError: can't concat str to bytes
xb.decode()          ← The object's decode method will
                     convert it back to a string.

'á'

```

The first thing you can see is that to convert from a regular (Unicode) string to bytes, you need to call the string's `encode` method. After it's encoded to a `bytes` object, the character is 2 bytes and no longer prints the same way that the string did. Further, if you attempt to add a `bytes` object and a string object together, you get a type error because the two types are incompatible. Finally, to convert a `bytes` object back to a string, you need to call that object's `decode` method.

Most of the time, you shouldn't need to think about Unicode or bytes at all. But when you need to deal with international character sets (an increasingly common problem), you must understand the difference between regular strings and `bytes`.

Quick check: Bytes

For which of the following kinds of data would you want to use a string? For which could you use bytes?

Data file storing binary data

Text in a language with accented characters

Text with only uppercase and lowercase roman characters

A series of integers no larger than 255

6.10 Preprocessing text

In processing raw text, it's quite often necessary to clean and normalize the text before doing anything else. If you want to find the frequency of words in text, for example, you can make the job easier if, before you start counting, you make sure that everything is lowercase (or uppercase, if you prefer) and that all punctuation has been removed. You can also make things easier by breaking the text into a series of words. In

this lab, the task is to read the first part of the first chapter of *Moby Dick*, make sure that everything is one case, remove all punctuation, and write the words one per line to a second file. As mentioned in the previous chapter, you should attempt to create a solution on your own before using AI to generate one.

To solve the problem, your code will need to open the source file, `moby_01.txt`. If you are using the Jupyter notebook, you can just execute the cell right before the lab. You can also download the file from the GitHub repository's folder for this chapter and save it in the directory you are working in.

You will also need to perform the operations to clean the text: making everything lowercase, removing punctuation, splitting the text into words, and writing the cleaned output to a file. My suggestion is that you process the file line by line since that will save memory, and it's easy to read and write line by line in Python.

Because I haven't yet covered reading and writing files, I provide the code for those operations—just put your code in place of the comments in the `for` loop:

```
with open("moby_01.txt") as infile:  
    with open("moby_01_clean.txt", "w") as outfile:  
        for line in infile:  
            # make all one (lower) case  
            # remove punctuation  
            # split into words  
            # write all words for line  
            outfile.write(cleaned_words)
```

Hint: You can work on one cleaning step at a time. First, create the code to make everything lowercase, and check to see that the text in the output file is lowercase. Then, work on removing the punctuation, and make sure that works, and so on.

6.10.1 Solving the problem with AI-generated code

Using AI to generate code for this problem is fairly simple. In this case, it's probably easier to let the AI tool take care of opening, reading, and writing the files. To create a prompt, we need to be sure to include requirements:

- 1 Open the file `moby_01.txt`.
- 2 Read the lines of text.
- 3 Make all the text lowercase.
- 4 Remove any punctuation.
- 5 Split the line into words.
- 6 Write the words of the line to a file `moby_01_clean.txt`.

If you want to try generating a solution using AI, go ahead and create a prompt and see what you get, and test to see if that code works correctly.

6.10.2 Solutions and discussion

Unsurprisingly, my solution and the AI solution were very similar. As we will see, however, the AI version misses a couple of tricks that would slightly improve efficiency.

THE HUMAN SOLUTION

Solving this problem using the framework provided here is fairly straightforward—just take it one step at a time:

```

import string
punct = str.maketrans("", "", string.punctuation)

with open("moby_01.txt") as infile:
    with open("moby_01_clean.txt", "w") as outfile:
        for line in infile:
            # make all one case
            cleaned_line = line.lower()

            # remove punctuation
            cleaned_line = cleaned_line.translate(punct)

            # split into words
            words = cleaned_line.split()
            cleaned_words = "\n".join(words) ←
            # write all words for line
            outfile.write(cleaned_words)

```

This solution is concise and reasonably efficient. In it we import the string library so that we can create a translation table to remove punctuation using Python’s list of punctuation rather than creating our own. In processing a line of text, we first use the `lower()` method on the entire line, and then use the `translate` method on the entire line. We then split the line into words, which will return a list of the words without any whitespace, and finally rejoin the words with a newline, which means that when written to the file, there will be one word per line.

THE AI VERSION

Creating the AI version is a bit simpler in that we don’t need to keep the scaffolding for opening, reading, and writing files, since the code generator should know how to do that. A reasonably concise prompt for what we need would be

For every line the file `moby_01.txt`, make everything lower case, remove punctuation, and split into words. Write the words one per line to the file `moby_01_clean.txt`.

For this problem, the solutions generated in Colaboratory and in Copilot were very similar. A good representative is the following solution generated by Copilot:

```

import string

# Open the input file
with open('moby_01.txt', 'r') as input_file:
    # Open the output file
    with open('moby_01_clean.txt', 'w') as output_file:
        # Iterate over each line in the input file

```

```

for line in input_file:
    # Convert the line to lowercase
    line = line.lower()

    # Remove punctuation from the line
    line = line.translate(str.maketrans('', '', string.punctuation))

    # Split the line into words
    words = line.split()

    # Write each word to the output file
    for word in words:
        output_file.write(word + '\n')

```



Creates a new translation table for every line

The solution is nicely formatted and well commented, although I might say that a comment like `# Iterate over each line in the input file` is obvious and redundant coming right before the line `for line in input_file:`, but that's not important.

A more serious problem is the way that the code uses the `translate` method. Note that instead of creating a translation table for punctuation once at the beginning of the program (as the human-created version does), it creates a new translation table on the fly for every line. Experienced coders will cringe—repeating an operation inside a loop unnecessarily can make your code run much, much slower and is the sort of rookie mistake most coders want to avoid.

One solution offered by Colaboratory makes this problem even worse. Instead of using the `translate` method, it suggests looping over every character in `string.punctuation` using the `replace` method to remove it from the string:

```

# Remove punctuation
for char in string.punctuation:
    line = line.replace(char, '')

```

Here again we have a loop where `translate` is a single operation, and in this case, we are looping over all of the punctuation characters in every line and creating a new string object every time a punctuation character is found and removed. While this isn't a problem with a relatively small file, it can quickly become a big problem if we have to process a dataset with millions of lines.

A similar problem occurs in the last section, which writes the words to the output file. While not as much of a potential performance problem as the first one, it's still a problem, for two reasons. First, it loops over the list of words in the line, and second, it uses string concatenation (+) to add the newlines. While Python `for` loops are heavily optimized, using a loop when one isn't needed will still make things slower than they have to be, and using the + to add two strings together forces the creation of a new string, which also takes a bit of time.

The human version gets around this by using the `join` method with a newline character. Instead of a loop, this is a single operation, and instead of creating a new string object for every word, it creates just one for the entire line.

In this lab, the code generated by AI tools does the job, but if you tried to put that code into production under any sort of load, the slow performance might well become a problem.

Summary

- Python strings have powerful text-processing features, including searching and replacing, trimming characters, and changing case.
- Strings are immutable; they can't be changed in place.
- Operations that appear to change strings actually return a copy with the changes.
- There are three ways to combine strings and the values of Python expressions: the format method, f-strings, and the legacy method using %.
- The format method has an extensive formatting language to control its output.
- F-strings are a very convenient way to create strings containing the values of variables and expressions and are now used widely.
- Python has a `bytes` data type that contains ints in the range 0 to 255; they look similar to strings but need to be decoded to be converted to strings.



Dictionaries

This chapter covers

- Defining a dictionary
- Using dictionary operations
- Determining what can be used as a key
- Creating sparse matrices
- Using dictionaries as caches
- Trusting the efficiency of dictionaries

This chapter discusses dictionaries, Python’s name for associative arrays or maps, which it implements by using hash tables. Dictionaries are amazingly useful, even in simple programs.

Because dictionaries are less familiar to many programmers than other basic data structures such as lists and strings, some of the examples illustrating dictionary use are slightly more complex than the corresponding examples for other built-in data structures. It may be necessary to read parts of chapter 8 to fully understand some of the examples in this chapter.

7.1 What is a dictionary?

If you've never used associative arrays or hash tables in other languages, a good way to start understanding the use of dictionaries is to compare them with lists:

- Values in lists are accessed by means of integers called *indices*, which indicate where in the list a given value is found.
- Dictionaries access values by means of integers, strings, or other Python objects called *keys*, which indicate where in the dictionary a given value is found. In other words, both lists and dictionaries provide indexed access to arbitrary values, but the set of items that can be used as dictionary indices is much larger than, and contains, the set of items that can be used as list indices. Also, the mechanism that dictionaries use to provide indexed access is quite different from that used by lists.
- Both lists and dictionaries can store objects of any type.
- Values stored in a list are implicitly *ordered* by their positions in the list, because the indices that access these values are consecutive integers. You may or may not care about this ordering, but you can use it if desired. Values stored in a dictionary are *not* implicitly ordered relative to one another because dictionary keys aren't just numbers.
- If you're using a dictionary but also care about the order of the items (the order in which they were added, that is), since Python 3.6 dictionaries store their items in the order they were added. There is also an *ordered dictionary*, which is a dictionary subclass that can be imported from the `collections` module and which offers a few extra features, such as a `move_to_end` method and the ability to access items in reverse order. You can also define an order on the items in a dictionary by using another data structure (often a list) to store such an ordering explicitly.

In spite of the differences between them, the use of dictionaries and lists often appears to be the same. As a start, an empty dictionary is created much like an empty list but with curly braces instead of square brackets:

```
x_list = []      ← A list
y_dic = {}       ← A dictionary
```

Here, the first line creates a new, empty list and assigns the variable `x` to it. The second line creates a new, empty dictionary and assigns the variable `y` to it.

After you create a dictionary, you may store values in it as though it were a list:

```
y_dic[0] = 'Hello'
y_dic[1] = 'Goodbye'
```

Even in these assignments, there's already a significant operational difference between the dictionary and list usage. Trying to do the same thing with a list would result in an

error, because in Python, it's illegal to assign to a position in a list that doesn't exist. For example, if you try to assign to the *0*th element of the list *x*, you receive an error:

```
x_list[0] = 'Hello'
-----
IndexError                                     Traceback (most recent call last)
<ipython-input-1-ba4ef4c8f6ab> in <cell line: 3>()
      1 x_list = []
      2 y_dic = {}
----> 3 x_list[0] = 'Hello'

IndexError: list assignment index out of range
```

This isn't a problem with dictionaries; new positions in dictionaries are created as necessary.

Having stored some values in the dictionary, now you can access and use them:

```
print(y_dic[0])
print(y_dic[1] + ", Friend.")

Hello
Goodbye, Friend.
```

Overall, this makes a dictionary look pretty much like a list. Now for the big difference: you can store (and use) values under keys that aren't integers:

```
y_dic['greeting'] = "Bon jour"
y_dic['farewell'] = "Adios"
print(y_dic['greeting'])
print(y_dic['farewell'] + ", Friend.")

Bon jour
Adios, Friend.
```

This is definitely something that can't be done with lists! Whereas list indices must be integers, dictionary keys are much less restricted; they may be numbers, strings, or one of a wide range of other Python objects. This makes dictionaries a natural for jobs that lists can't do. For example, it makes more sense to implement a telephone directory application with dictionaries than with lists because the phone number for a person can be stored and indexed by that person's last name.

A dictionary is a way of mapping from one set of arbitrary objects to an associated but equally arbitrary set of objects. Actual dictionaries, thesauri, or translation books are good analogies in the real world. To see how natural this correspondence is, here's the start of an English-to-French color translator:

```
english_to_french = {}
english_to_french['red'] = 'rouge'           ← Creates empty dictionary
english_to_french['blue'] = 'bleu'           ← Stores three words in it
english_to_french['green'] = 'vert'
```

```
print("red is", english_to_french['red'])           ← Obtains value for "red"  
red is rouge
```

Try this: Create a dictionary

Write the code to ask a user for three names and three ages. After the names and ages are entered, ask the user for one of the names, and print the correct age.

7.2 Other dictionary operations

Besides basic element assignment and access, dictionaries support several operations. You can define a dictionary explicitly as a series of key-value pairs separated by commas:

```
english_to_french = {'red': 'rouge', 'blue': 'bleu', 'green': 'vert'}  
len returns the number of entries in a dictionary:  
len(english_to_french)  
  
3
```

You can obtain all the keys in the dictionary with the `keys` method. This method is often used to iterate over the contents of a dictionary using Python's `for` loop, described in chapter 8:

```
list(english_to_french.keys())  
  
['green', 'blue', 'red']
```

In Python 3.5 and earlier, the order of the keys in a list returned by `keys` has no meaning; the keys aren't necessarily sorted, and they don't necessarily occur in the order in which they were created. Your Python code may print out the keys in a different order than my Python code did. If you need keys sorted, you can store them in a list variable and then sort that list. However, as mentioned earlier, starting with Python 3.6, dictionaries preserve the order that the keys were created and return them in that order.

It's also possible to obtain all the values stored in a dictionary by using `values`:

```
list(english_to_french.values())  
  
['vert', 'bleu', 'rouge']
```

This method isn't used nearly as often as `keys`.

You can use the `items` method to return all keys and their associated values as a sequence of tuples:

```
list(english_to_french.items())  
  
[('green', 'vert'), ('blue', 'bleu'), ('red', 'rouge')]
```

Like `keys`, this method is often used in conjunction with a `for` loop to iterate over the contents of a dictionary.

The `del` statement can be used to remove an entry (key-value pair) from a dictionary:

```
list(english_to_french.items())
[('green', 'vert'), ('blue', 'bleu'), ('red', 'rouge')]
del english_to_french['green']
list(english_to_french.items())
[('blue', 'bleu'), ('red', 'rouge')]
```

Dictionary view objects

The `keys`, `values`, and `items` methods return not lists but *views* that behave like sequences but are dynamically updated whenever the dictionary changes. That's why you need to use the `list` function to make them appear as a list in these examples. Otherwise, they behave like sequences, allowing code to iterate over them in a `for` loop, using `in` to check membership in them, and so on.

The view returned by `keys` (and in some cases the view returned by `items`) also behaves like a set, with `union`, `difference`, and `intersection` operations.

Attempting to access a key that isn't in a dictionary is an error in Python. To handle this error, you can test the dictionary for the presence of a key with the `in` keyword, which returns `True` if a dictionary has a value stored under the given key and `False` otherwise:

```
'red' in english_to_french
True
'orange' in english_to_french
False
```

Alternatively, you can use the `get` function. This function returns the value associated with a key if the dictionary contains that key but returns its second argument if the dictionary doesn't contain the key:

```
print(english_to_french.get('blue', 'No translation'))
```

bleu

Chartreuse not found, “No translation” is returned.

```
print(english_to_french.get('chartreuse', 'No translation'))
```

No translation

The second argument is optional. If that argument isn't included, `get` returns `None` if the dictionary doesn't contain the key.

Similarly, if you want to safely get a key's value *and* make sure that it's set to a default in the dictionary, you can use the `setdefault` method:

```
print(english_to_french.setdefault('chartreuse', 'No translation'))
```

No translation

Chartreuse not found and is added as key, and “No translation” is returned and added as its value.

The difference between `get` and `setdefault` is that after the `setdefault` call, there's a key in the dictionary 'chartreuse' with the value 'No translation'.

There is also a `defaultdict` subclass of `dict` that you can import from the `collections` module, and instances of `defaultdict` can be given a default value and will automatically work the same was as `setdefault`.

You can obtain a copy of a dictionary by using the `copy` method:

```
x = {0: 'zero', 1: 'one'}
y = x.copy()
y

{0: 'zero', 1: 'one'}
```

This method makes a shallow copy of the dictionary, which is likely to be all you need in most situations. For dictionaries that contain any modifiable objects as values (for example, lists or other dictionaries), you may want to make a deep copy by using the `copy.deepcopy` function. See chapter 5 for an introduction to the concept of shallow and deep copies.

The `update` method updates the dictionary it is a member of with all the key-value pairs of a second dictionary. For keys that are common to both dictionaries, the values from the second dictionary override those of the first:

```
z = {1: 'ONE', 2: 'Two'}
x = {0: 'zero', 1: 'one'}
x.update(z)
x

{0: 'zero', 1: 'ONE', 2: 'Two'}
```

Dictionary methods give you a full set of tools to manipulate and use dictionaries. For quick reference, table 7.1 lists some of the main dictionary functions.

Table 7.1 Dictionary operations

Dictionary operation	Explanation	Example
<code>{}</code>	Creates an empty dictionary	<code>x = {}</code>
<code>len</code>	Returns the number of entries in a dictionary	<code>len(x)</code>
<code>keys</code>	Returns a view of all keys in a dictionary	<code>x.keys()</code>

Table 7.1 Dictionary operations (continued)

Dictionary operation	Explanation	Example
values	Returns a view of all values in a dictionary	x.values()
items	Returns a view of all items in a dictionary	x.items()
del	Removes an entry from a dictionary	del(x[key])
in	Tests whether a key exists in a dictionary	'y' in x
get	Returns the value of a key or a configurable default	x.get('y', None)
setdefault	Returns the value if the key is in the dictionary; otherwise, sets the value for the key to the default and returns the value	x.setdefault('y', None)
copy	Makes a shallow copy of a dictionary	y = x.copy()
update	Combines the entries of two dictionaries	x.update(z)

This table isn't a complete list of all dictionary operations. For a complete list, refer to the Python standard library documentation.

Quick check: Dictionary operations

Assume that you have a dictionary `x = {'a':1, 'b':2, 'c':3, 'd':4}` and a dictionary `y = {'a':6, 'e':5, 'f':6}`. What would be the contents of `x` after the following snippets of code have executed?

```
del x['d']
z = x.setdefault('g', 7)
x.update(y)
```

7.3

Word counting

Assume that you have a file that contains a list of words, one word per line. You want to know how many times each word occurs in the file. You can use dictionaries to perform this task easily:

```
sample_string = "To be or not to be"
occurrences = {}
for word in sample_string.split():
    occurrences[word] = occurrences.get(word, 0) + 1
for word in occurrences:
    print("The word", word, "occurs", occurrences[word],
          "times in the string")
```

Increments the
occurrences count
for the word

The word To occurs 1 times in the string
The word be occurs 2 times in the string
The word or occurs 1 times in the string

```
The word not occurs 1 times in the string
The word to occurs 1 times in the string
```

This is a good example of the power of dictionaries, using each word as a dictionary key and the `get()` method to increment the count for that word if found. The code is simple, but because dictionary operations are highly optimized in Python, it's also quite fast. This pattern is so handy, in fact, that it's been standardized as the `Counter` class in the `collections` module of the standard library.

7.4 What can be used as a key?

The previous examples use strings as keys, but Python permits more than just strings to be used in this manner. Any Python object that is immutable and hashable can be used as a key to a dictionary.

In Python, as discussed earlier, any object that can be modified is called *mutable*. Lists are mutable because list elements can be added, changed, or removed. Dictionaries are also mutable for the same reason. Numbers are immutable. If a variable `x` is referring to the number 3 and you assign 4 to `x`, you've made `x` refer to a different number (4) but you haven't changed the number 3 itself; 3 still has to be 3. Strings are also immutable. `list[n]` returns the *n*th element of `list`, `string[n]` returns the *n*th character of `string`, and `list[n] = value` changes the *n*th element of `list`, but `string[n] = character` is illegal in Python and causes an error, because strings in Python are immutable.

Unfortunately, the requirement that keys be immutable and hashable means that lists can't be used as dictionary keys, but in many instances, it would be convenient to have a listlike key. For example, it's convenient to store information about a person under a key consisting of the person's first and last names, which you could easily do if you could use a two-element list as a key.

Python solves this difficulty by providing tuples, which are basically immutable lists; they're created and used similarly to lists, except that, once created, they can't be modified. There's one further restriction: keys must also be hashable, which takes things a step further than just immutable. To be hashable, an object must have a hash value (provided by a `__hash__` method) that never changes throughout the life of the value. That means that tuples containing mutable objects are *not* hashable, although the tuples themselves are technically immutable. Only tuples that don't contain any mutable objects nested within them are hashable and valid to use as keys for dictionaries. Table 7.2 illustrates which of Python's built-in types are immutable, hashable, and eligible to be dictionary keys.

Table 7.2 Python values eligible to be used as dictionary keys

Python type	Immutable?	Hashable?	Dictionary key?
<code>int</code>	Yes	Yes	Yes
<code>float</code>	Yes	Yes	Yes

Table 7.2 Python values eligible to be used as dictionary keys (continued)

Python type	Immutable?	Hashable?	Dictionary key?
boolean	Yes	Yes	Yes
complex	Yes	Yes	Yes
str	Yes	Yes	Yes
bytes	Yes	Yes	Yes
bytearray	No	No	No
list	No	No	No
tuple	Yes	Sometimes	Sometimes
set	No	No	No
frozenset	Yes	Yes	Yes
dictionary	No	No	No

The next sections give examples illustrating how tuples and dictionaries can work together.

Quick check: What can be a key?

Decide which of the following expressions can be a dictionary key: 1; 'bob'; ('tom', [1, 2, 3]); ["filename"]; "filename"; ("filename", "extension").

7.5 Sparse matrices

In mathematical terms, a *matrix* is a two-dimensional grid of numbers, usually written in textbooks as a grid with square brackets on each side, as shown here:

$$\begin{bmatrix} 3 & 0 & -2 & 11 \\ 0 & 9 & 0 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & -5 \end{bmatrix}$$

A fairly standard way to represent such a matrix is by means of a list of lists. In Python, a matrix is presented like this:

```
matrix = [[3, 0, -2, 11], [0, 9, 0, 0], [0, 7, 0, 0], [0, 0, 0, -5]]
```

Elements in the matrix can be accessed by row and column number:

```
element = matrix[rownum][colnum]
```

But in some applications, such as weather forecasting, it's common for matrices to be very large—thousands of elements to a side, meaning millions of elements in total. It's also common for such matrices to contain many zero elements. In some applications, all but a small percentage of the matrix elements may be set to zero. To conserve memory, it's common for such matrices to be stored in a form in which only the nonzero elements are actually stored. Such representations are called *sparse matrices*.

It's simple to implement sparse matrices by using dictionaries with tuple indices. For example, the previous sparse matrix can be represented as follows, with each key being the row and column of a value:

```
matrix = {(0, 0): 3, (0, 2): -2, (0, 3): 11,
          (1, 1): 9, (2, 1): 7, (3, 3): -5}
```

Now you can access an individual matrix element at a given row and column number by this bit of code:

```
if (rownum, colnum) in matrix:
    element = matrix[(rownum, colnum)]
else:
    element = 0
```

A slightly less clear (but more efficient) way of doing this is to use the dictionary `get` method, which you can tell to return `0` if it can't find a key in the dictionary and, otherwise, return the value associated with that key, preventing one of the dictionary lookups:

```
element = matrix.get((rownum, colnum), 0)
```

If you're considering doing extensive work with matrices, you may want to look into NumPy, the numeric computation package.

7.6 Dictionaries as caches

This section shows how dictionaries can be used as *caches*, data structures that store results to avoid recalculating those results over and over. Suppose that you need a function called `sole`, which takes three integers as arguments and returns a result. The function might look something like the following:

```
def sole(m, n, t):
    # . . . do some time-consuming calculations . . .
    return(result)
```

But if this function is very time consuming, and if it's called tens of thousands of times, the program might run too slowly.

Now suppose that `sole` might be called with about 200 different combinations of arguments during any program run. That is, you might call `sole(12, 20, 6)` 50 or more times during the execution of your program and similarly for many other combinations

of arguments. By eliminating the recalculation of `sole` on identical arguments, you'd save a huge amount of time. You could use a dictionary with tuples as keys, like so:

```
sole_cache = {}
def sole(m, n, t):
    if (m, n, t) in sole_cache:
        return sole_cache[(m, n, t)]
    else:
        # . . . do some time-consuming calculations . . .
        sole_cache[(m, n, t)] = result
    return result
```

The rewritten `sole` function uses a global variable to store previous results. The global variable is a dictionary, and the keys of the dictionary are tuples corresponding to argument combinations that have been given to `sole` in the past. Then, anytime `sole` passes an argument combination for which a result has already been calculated, it returns that stored result rather than recalculating it.

Try this: Using dictionaries

Suppose that you're writing a program that works like a spreadsheet. How might you use a dictionary to store the contents of a sheet? Write some sample code to both store a value in and retrieve a value from a particular cell. What might be some drawbacks to this approach?

7.7 Efficiency of dictionaries

If you come from a traditional compiled-language background, you may hesitate to use dictionaries, worrying that they're less efficient than lists (arrays). The truth is that the Python dictionary implementation is quite fast. Many of the internal language features rely on dictionaries, and a lot of work has gone into making them efficient. Because all of Python's data structures are heavily optimized, you shouldn't spend much time worrying about which is faster or more efficient. If the problem can be solved more easily and cleanly by using a dictionary than by using a list, do it that way, and consider alternatives only if it's clear that dictionaries are causing an unacceptable slowdown.

7.8 Word counting

In the previous lab, you took the text of the first chapter of *Moby Dick*, normalized the case, removed punctuation, and wrote the separated words to a file. In this lab, we will use the file created in chapter 6 to count the number of times each word occurs.

To do this, we will read the file and use a dictionary as described previously to count the occurrences, using the words as the keys and the number of occurrences as the values. Once we have processed the file, we'll print out the five most common and five least common words and the number of times they occur.

To open and read the file, we can use code that's similar to what we used the last time, but we won't need an output file:

```
with open("moby_01_clean.txt", "r") as infile:
    for word in infile:
        # use strip() method to remove any extra whitespace
        # increase the count for current word
```

In addition to using a dictionary to count occurrences, you'll need to use the `items()` method to get the keys and values as a series of tuples that you can sort. For this you may need to refer to chapter 5 on lists.

Hint: You will also need to create an empty dictionary to hold the word counts *before* you start processing the words.

7.8.1 Solving the problem with AI-generated code

Again, the main factor in using AI to generate the code will be in creating the prompt. As in the previous chapter, we don't need to worry about telling the AI chatbot how to read a file, but in this case if we want to be sure it uses a dictionary and not the `Counter` class, we will need to specify that. So our prompt should mention the following:

- 1 Open the file `moby_01_clean.txt`.
- 2 Use a dictionary to count the number of occurrences of each word.
- 3 Print the five most common words and their number of occurrences.
- 4 Print the five least common words and their number of occurrences.

If you are using an AI chatbot, create a prompt and see what solution you get.

7.8.2 Solutions and discussion

While this problem is a little more complex than the last lab, it is still straightforward enough that my solution and the AI solutions are similar. The AI solutions, however, still suggest doing things in slightly less efficient ways.

THE HUMAN SOLUTION

The solution to this problem requires a bit more thought than the one in the previous chapter, since more processing has to occur after the file is read, and you need to remember to create a dictionary to hold the word counts before processing the words:

```
word_count = {}
with open("moby_01_clean.txt", "r") as infile:
    for word in infile:
        # use strip() method to remove any extra whitespace
        word = word.strip()
        # increase the count for current word
        word_count.setdefault(word, 0)
        word_count[word] += 1
word_list = list(word_count.items())
```

If word not present, adds and sets count to zero
Adds 1 to count
Gets items as list of tuples

```

word_list.sort(key=lambda x: x[1]) ← Sorts the items by frequency
print("Most common words:")
for word in reversed(word_list[-5:]): ← Reverses order so higher
    print(word)
print("\nLeast common words:")
for word in word_list[:5]:
    print(word)

```

The key to this solution is using the dictionary's `setdefault()` method to set the word's entry in the dictionary to zero if it doesn't already exist in the dictionary. Then we add 1 to the word's value in the dictionary.

Once all of the words have been counted, we use the dictionary's `items()` method to get a list of word-frequency tuples we can sort by frequency. We can then reverse that list to get the most frequent and use the original order to get the least frequent.

THE AI VERSION

Just as with the previous problem, the AI generators provide code that works and is nicely commented.

As mentioned earlier, the prompt should be explicit that the `Counter` class should not be used, so the prompt that I used was

Open the file `moby_01_clean.txt` and use a dictionary to count the occurrences of each word. Do not use a `Counter`. Print the five most common words and their number of occurrences. Print the five least common words and their number of occurrences.

Both Colaboratory and Copilot came up with very similar approaches. In this case, let's look at Colaboratory's solution:

```

# Open the file and read the text into a variable.
with open('moby_01_clean.txt', 'r') as f:
    text = f.read() ← Reads entire file

# Split the text into words.
words = text.split() ← Splits entire file into list of words

# Create a dictionary to count the occurrences of each word.
word_counts = {}
for word in words:
    if word not in word_counts: ← Uses if to check if
        word_counts[word] = 0 ← word is in dictionary
    word_counts[word] += 1

# Find the five most common words and their number of occurrences.
most_common = sorted(word_counts.items(), key=lambda item: item[1],
                     reverse=True)[:5] ← Creates reversed sorted list
# Find the five least common words and ← to get most frequent
    # CA}their number of occurrences. ← Creates normal sorted list

```

to get least frequent

```

least_common = sorted(word_counts.items(), key=lambda item: item[1])[:5]

# Print the results.
print("Most common words:")
for word, count in most_common:
    print(f"{word}: {count}")

print("\nLeast common words:")
for word, count in least_common:
    print(f"{word}: {count}")

```

The Copilot version is similar, and both solutions read the entire file into memory and then split it into one big list of words. This might be a problem if the size of the file is quite large. The human version takes advantage of there being one word per line to process the file line by line, which would avoid this problem. Interestingly, even if the AI chatbots were told in the prompt that there was only one word per line, they wouldn't take advantage of that fact.

There are a couple of interesting differences between the two. The Copilot version uses a slightly more verbose way of checking the list:

```

for word in words:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1

```

Note that, in this case, if the word isn't already in the dictionary, it adds it with a value of 1, since it wouldn't have checked if it hadn't hit an occurrence of the word. Otherwise, it increments the value for the word in place. Theoretically this might be a tiny bit faster, but neither solution will be quite as good as using `setdefault` (or `get`), which the human version does.

The second difference is in getting the most- and least-common words. To do this, the list of items needs to be sorted at least once. The problem with the Colaboratory solution is that it does *two* sorts: one in reverse order and one in normal order. Obviously, doing twice the work could be a problem with larger datasets. The Copilot version only does one sort, in reverse order, and then gets the first five elements as the most frequent and the last five as the least frequent:

```

# Sort the word counts in descending order
sorted_word_counts = sorted(word_counts.items(), key=lambda x: x[1],
                            reverse=True)

```

The human version also does only one sort, in normal order, and then, to get the most frequent, it uses the `reversed` function on a slice of the last five items, so that they are printed from most frequent to least frequent. `reversed` returns an iterator so in fact no new list is created, even with those five items. In fact, a reverse slice would also have worked just as efficiently.

Just as in the previous labs, in this lab both AI chatbots produced code that works but with variations that could be problematic in production environments with large datasets.

Summary

- Dictionaries are powerful data structures, used for many purposes, even within Python itself.
- Items in a dictionary are accessed by keys, not by their order in the dictionary.
- Dictionary keys must be immutable, but any immutable object can be a dictionary key.
- All dictionaries maintain items in the order in which they were added, but ordered dictionaries (found in the collections library) offer some extra features.
- By using the keys as part of the data, dictionaries can be used for tasks like counting elements, caching values, or creating sparse matrices.
- Using keys means accessing collections of data more efficiently and with less code than many other solutions.

8 Control flow

This chapter covers

- Making decisions: the `if-elif-else` statement
- Structural pattern matching
- Repeating code with a `while` loop
- Iterating over a list with a `for` loop
- Using list and dictionary comprehensions
- Delimiting statements and blocks with indentation
- Evaluating Boolean values and expressions

Python provides a complete set of control flow elements, with loops and conditionals. This chapter examines each element in detail.

8.1 **The `if-elif-else` statement**

One of the most important control flow features in most programming languages is the one for branching—that is, for making decisions on whether or not to execute or skip a piece of code based on some conditional. In Python, as in many languages,

there are three parts to this: an `if` section, optionally one or more `elif` sections that are only checked if the `if` condition (and any previous `elifs`) is false, and a final optional `else` section that is only executed if neither the `if` or `elif` sections were executed.

The general form of the `if-elif-else` construct in Python is

```
if condition1:
    body1
elif condition2:
    body2
elif condition3:
    body3
.
.
.
elif condition(n-1):
    body(n-1)
else:
    body(n)
```

It says: if `condition1` is `True`, execute `body1`; otherwise, if `condition2` is `True`, execute `body2`; otherwise—and so on until it either finds a condition that evaluates to `True` or hits the `else` clause, in which case it executes `body(n)`. The `body` sections are again sequences of one or more Python statements that are separated by newlines and are at the same level of indentation.

For example, if you wanted to see if a guess in a guessing game was low, correct, or high, you might do this:

```
guess = 4
target = 7

if guess < target:
    print("Low")
elif guess == target:
    print("Correct!")
else:           ← If the guess is not low and
                not equal, it must be high.
    print("High")
```

LOW

You don't need all that baggage for every conditional, of course. Both the `elif` and the `else` parts are optional. If a conditional can't find any body to execute (no conditions evaluate to `True`, and there's no `else` part), it does nothing.

The body after the `if` statement is required. But you can use the `pass` statement here (as you can anywhere in Python where a statement is required). The `pass` statement serves as a placeholder where a statement is needed, but it performs no action:

```
if x < 5:
    pass
else:
    x = 5
```

8.2 Structural pattern matching with `match`

Starting with Python 3.11, Python has the match-case statement, which can select between multiple conditions. It is similar but more powerful than the `case` and `switch` statements of other languages. The official name for what the `match` statement does is *structural pattern matching* and the “structural” part is important—unlike some other languages, the match-case statement can make matches not only on the basis of equal values but also in terms of matching *types*.

The structure of a match-case statement is

```
match expression:
    case test_expression:
        code block for match
    case _:
        code block if no match (optional)
```

Usually, the `expression` will be a variable or object, and the `test_expression` can either be a specific value or object or it can be a type function or a class constructor. It's also possible to match more than one option by using the “or” operator `|`. Optionally, you may include a case for `_`, which will be matched if nothing else is. If none of the cases is matched, no code is executed.

For example, the following code will check both for matches to the string "A" or the integer 0, one of the values 1, 2, or 3, or any other string or integer:

```
x = 5
match x:
    case "A":           ← Matches exactly "A"
        print("A")
    case str():          ← Matches any other string
        print("some other string")
    case 0:              ← Matches exactly 0
        print("Zero")
    case 1 | 2 | 3:      ← Matches any of 1, 2, or 3
        print(f"value {x} in range")
    case int():          ← Matches any other int
        print(f"The integer {x}")
    case _:               ← Executed if no match (optional)
        print("Neither string nor int")
```

Pattern matching is quite powerful, but it is also deceptively complex. Given all of the ways that expressions can be matched, it's quite possible to get code that behaves unexpectedly. Before using `match` in your code, it would be a good idea to carefully read the documentation (the Python tutorial online at <https://docs.python.org> is a good place to start) and to think carefully about possible edge cases and ways to test for them.

8.3 The `while` loop

You've come across the basic `while` loop several times already. The full `while` loop looks like the following:

```
while condition:
    body
else:
    post-code
```

`condition` is a Boolean expression—that is, one that evaluates to a `True` or `False` value. As long as it's `True`, the `body` is executed repeatedly. When the `condition` evaluates to `False`, the `while` loop executes the `post-code` section and then terminates. If the condition starts out by being `False`, the `body` won't be executed at all—just the `post-code` section. The `body` and `post-code` are each sequences of one or more Python statements that are separated by newlines and are at the same level of indentation. The Python interpreter uses this level to delimit them. No other delimiters, such as braces or brackets, are necessary.

Note that the `else` part of the `while` loop is optional and not often used. That's because as long as there's no `break` in the `body`, this loop

```
while condition:
    body
else:
    post-code
```

and this loop

```
while condition:
    body
post-code
```

do the same things—and the second is simpler to understand. On the other hand, if a `break` is used in the `body` of the loop, then the `else` block is *not* executed. This can be useful since it eliminates the need to set a flag before the `break` and check it after the loop completes.

For example, a `while` loop that repeats until the user enters a "Q" could be

```
response = ""
while response != "Q":
    response = input("Q to quit, B to break")
    if response == "B":
        break
    else:
        print("no break")
```

The break will force the while loop to end and skip the else block.

The else block will only be executed if there is no break.

The two special statements `break` and `continue` can be used in the `body` of a `while` loop. If `break` is executed, it immediately terminates the `while` loop, and as just mentioned, not even the `post-code` (if there is an `else` clause) is executed. If `continue` is executed, it causes the remainder of the `body` to be skipped over; the `condition` is evaluated again, and the loop proceeds as normal.

8.4 The for loop

A `for` loop in Python is different from `for` loops in some other languages. The traditional pattern is to increment and test a variable on each iteration, which is what C `for` loops usually do. In Python, a `for` loop iterates over the values returned by any iterable object—that is, any object that can yield a sequence of values. For example, a `for` loop can iterate over every element in a list, a tuple, or a string. But an iterable object can also be a special function called `range` or a special type of function called a *generator* or a *generator expression*, which can be quite powerful. The general form is

```
for item in sequence:
    body
else:
    post-code
```

`body` is executed once for each element of `sequence`. `item` is set to be the first element of `sequence`, and `body` is executed; then `item` is set to be the second element of `sequence`, and `body` is executed; and so on for each remaining element of the `sequence`.

The `else` part is optional. Like the `else` part of a `while` loop, it's rarely used, but just as in a `while` loop, the `post-code` block following the `else` is not executed if a `break` occurs. The `break` and `continue` statements do the same thing in a `for` loop as they do in a `while` loop. If `break` is executed, it immediately terminates the `for` loop, and not even the `post-code` (if there is an `else` clause) is executed. If `continue` is executed in a `for` loop, it causes the code to skip the remainder of the `body`, and the loop proceeds from the top as normal with the next item.

This small loop prints out the reciprocal of each number in `x`, skipping zeroes and stopping if it hits a negative number:

```
x = [1.0, 2.0, 3.0, 0.0, -1.0]
for n in x:
    if n == 0.0:
        print("skipping 0.0")
        continue
    elif n < 0.0:
        print("no negative numbers!")
        break
    print(1 / n)
else:
    print("Loop completed normally")
```

The loop skips back to the top of the loop with the next item `n`.

The loop exits immediately; the `else` block is skipped.

Only executed if no `break` is executed

```
1.0
0.5
0.3333333333333333
skipping 0.0
no negative numbers!
```

8.4.1 The range function

Sometimes you need to loop with explicit indices (such as the positions at which values occur in a list). You can use the `range` command together with the `len` command on a

list to generate a sequence of indices for use by the `for` loop. This code prints out all the positions in a list where it finds negative numbers:

```
x = [1, 3, -7, 4, 9, -5, 4]
for i in range(len(x)):
    if x[i] < 0:
        print("Found a negative number at index ", i)
```

Given a number n , `range(n)` returns a sequence $0, 1, 2, \dots, n-2, n-1$. So passing it the length of a list (found using `len`) produces a sequence of the indices for that list's elements. The `range` function doesn't build a Python list of integers; it just appears to. Instead, it creates a `range` object that produces integers on demand. This is useful when you're using explicit loops to iterate over really large lists. Instead of building a list with 10 million elements in it, for example, which would take up quite a bit of memory, you can use `range(10000000)`, which takes up only a small amount of memory and generates a sequence of integers from 0 up to (but *not* including) 10,000,000 as needed by the `for` loop.

8.5 Controlling range with starting and stepping values

You can use two variants on the `range` function to gain more control over the sequence it produces. If you use `range` with two numeric arguments, the first argument is the starting number for the resulting sequence, and the second number is the number the resulting sequence goes up to (but doesn't include). Here are a few examples:

<code>list(range(3, 7))</code>	←	list() used to return range() values as a list
<code>[3, 4, 5, 6]</code>	←	
<code>list(range(2, 10))</code>	←	
<code>[2, 3, 4, 5, 6, 7, 8, 9]</code>	←	
<code>list(range(5, 3))</code>	←	
<code>[]</code>	←	

In the preceding code, `list()` is used only to force the items `range` to appear as a list and may not be used often in production code.

Using a starting point after the end point doesn't allow you to count backward, which is why the value of `list(range(5, 3))` is an empty list. To count backward, or to count by any amount other than 1, you need to use the optional third argument to `range`, which gives a step value by which counting proceeds:

```
list(range(0, 10, 2))
[0, 2, 4, 6, 8]
```

```
list(range(5, 0, -1))

[5, 4, 3, 2, 1]
```

Sequences returned by `range` always include the starting value given as an argument to `range` and never include the ending value given as an argument.

8.6 The for loop and tuple unpacking

You can use tuple unpacking to make some `for` loops cleaner. The following code takes a list of two-element tuples and calculates the value of the sum of the products of the two numbers in each tuple (a moderately common mathematical operation in some fields):

```
somelist = [(1, 2), (3, 7), (9, 5)]
result = 0
for t in somelist:
    result = result + (t[0] * t[1])
```

Here's the same thing but cleaner:

```
somelist = [(1, 2), (3, 7), (9, 5)]
result = 0

for x, y in somelist:
    result = result + (x * y)
```

This code uses a tuple `x, y` immediately after the `for` keyword instead of the usual single variable. On each iteration of the `for` loop, `x` contains element 0 of the current tuple from `list`, and `y` contains element 1 of the current tuple from `list`. Using a tuple in this manner is a convenience of Python, and doing this indicates to Python that each element of the list is expected to be a tuple of appropriate size to unpack into the variable names mentioned in the tuple after the `for`.

8.7 The enumerate function

You can combine tuple unpacking with the `enumerate` function to loop over both the items and their index. This is similar to using `range` but has the advantage that the code is clearer and easier to understand. Like the previous example, the following code prints out all the positions in a list where it finds negative numbers:

```
x = [1, 3, -7, 4, 9, -5, 4]
for i, n in enumerate(x):
    if n < 0:
        print("Found a negative number at index ", i)
```

enumeration() returns tuples of (index, item).

Accesses item without the index

The index is also available.

The `enumerate` function returns tuples of (`index`, `item`) for each item in the list. You can access either value without looking at the other.

8.8 The `zip` function

Sometimes it's useful to combine two or more iterables before looping over them. The `zip` function takes the corresponding elements from one or more iterables and combines them into tuples until it reaches the end of the shortest iterable:

```
x = [1, 2, 3, 4]
y = ['a', 'b', 'c']           ← y is three elements;
z = zip(x, y)                ← x is four elements.
list(z)

[(1, 'a'), (2, 'b'), (3, 'c')] ← z has only three elements.
```

Try this: Looping and if statements

Suppose that you have a list `x = [1, 3, 5, 0, -1, 3, -2]`, and you need to remove all negative numbers from that list. Write the code to do this.

How would you count the total number of negative numbers in the following list?

```
y = [[1, -1, 0], [2, 5, -9], [-2, -3, 0]]?
```

What code would you use to print `very low` if the value of `x` is below `-5`, `low` if it's from `-5` up to `0`, `neutral` if it's equal to `0`, `high` if it's greater than `0` up to `5`, and `very high` if it's greater than `5`?

8.9 List, set, and dictionary comprehensions

The pattern of using a `for` loop to iterate through a list, modify or select individual elements, and create a new list or dictionary is very common. Such loops often look a lot like the following:

```
x = [1, 2, 3, 4]
x_squared = []
for item in x:
    x_squared.append(item * item)

x_squared
[1, 4, 9, 16]
```

This sort of situation is so common that Python has a special shortcut for such operations, called a *comprehension*. You can think of a list, set, or dictionary comprehension as a one-line `for` loop that creates a new list, set, or dictionary from a sequence. The pattern of a list comprehension is as follows:

```
new_list = [expression1 for variable in old_list if expression2]
```

Note that any sequence can be used in place of `old_list`, and what makes the value of the comprehension a list is the use of square brackets `[]` around the comprehension. A set comprehension would be similar but would use `{}` around the comprehension, and the value returned would be a set:

```
new_set = {expression1 for variable in old_list if expression2}
```

Finally, a dictionary comprehension looks like a set comprehension but needs both a key and value expression (e.g., list might contain two element tuples):

```
new_dict = {expression1:expression2 for variable in old_list if expression3}
```

In both cases, the heart of the expression is similar to the beginning of a `for` loop—`for variable in list`—with some expression using that variable to create a new key or value and an optional conditional expression using the value of the variable to select whether it's included in the new list or dictionary. The following code does exactly the same thing as the previous code but is a list comprehension:

```
x = [1, 2, 3, 4]
x_squared = [item * item for item in x]
x_squared

[1, 4, 9, 16]
```

You can even use `if` statements to select items from the list:

```
x = [1, 2, 3, 4]
x_squared = [item * item for item in x if item > 2]
x_squared

[9, 16]
```

Dictionary comprehensions are similar, but you need to supply both a key and a value. If you want to do something similar to the previous example but have the number be the key and the number's square be the value in a dictionary, you can use a dictionary comprehension, like so:

```
x = [1, 2, 3, 4]
x_squared_dict = {item: item * item for item in x}
x_squared_dict

{1: 1, 2: 4, 3: 9, 4: 16}
```

List, set, and dictionary comprehensions are very flexible and powerful, and when you get used to them, they make list-processing operations much simpler. I recommend

that you experiment with them and try them anytime you find yourself writing a `for` loop to process a list of items.

8.9.1 Generator expressions

Generator expressions don't use square brackets or curly braces (as list or dictionary comprehensions do); they use parentheses instead. The following example is the generator-expression version of the list comprehension already discussed:

```
x = [1, 2, 3, 4]
x_squared = (item * item for item in x)
x_squared

<generator object <genexpr> at 0x102176708>

for square in x_squared:
    print(square,)

1 4 9 16
```

As well as the change from square brackets, notice that this expression doesn't return a list. Instead, it returns a generator object that could be used as the iterator in a `for` loop, as shown, which is very similar to what the `range()` function does. The advantage of using a generator expression is that the entire list isn't generated in memory, so arbitrarily large sequences can be generated with little memory overhead.

Try this: Comprehensions

What list comprehension would you use to process the list `x` so that all negative values are removed?

Create a generator that returns only odd numbers from 1 to 100. (Hint: A number is odd if there is a remainder if divided by 2; use `% 2` to get the remainder of division by 2.)

Write the code to create a dictionary of the numbers and their cubes from 11 through 15.

8.10 Statements, blocks, and indentation

Because the control flow constructs you've encountered in this chapter are the first to make use of blocks and indentation, this is a good time to revisit the subject.

Python uses the indentation of the statements to determine the delimitation of the different blocks (or bodies) of the control flow constructs. A block consists of one or more statements, which are usually separated by newlines. Examples of Python statements are the assignment statement, function calls, the `print` function, the placeholder `pass` statement, and the `del` statement. The control flow constructs (`if-elif-else`, `while`, and `for` loops) are compound statements:

```
compound statement clause:  
    block  
compound statement clause:  
    block
```

A compound statement contains one or more clauses that are each followed by indented blocks. Compound statements can appear in blocks just like any other statements. When they do, they create nested blocks.

You may also encounter a couple of special cases. While it's usually not good Python style, multiple statements may be placed on the same line if they're separated by semicolons. A block containing a single line may be placed on the same line after the colon of a clause of a compound statement:

```
x = 1; y = 0; z = 0  
if x > 0: y = 1; z = 10  
else: y = -1  
  
print(x, y, z)  
  
1 1 10
```

Improperly indented code results in an exception being raised. You may encounter two forms of this exception. The first is

```
x = 1  
x = 2  
  
File "<ipython-input-21-c75cba843d9b>", line 2  
    x = 2  
    ^  
IndentationError: unexpected indent
```

This code indented a line that shouldn't have been indented. In the basic mode, the caret (^) indicates the spot where the problem occurred.

One situation where this can occur can be confusing. If you're using an editor that displays tabs in four-space increments (or Windows interactive mode, which indents the first tab only four spaces from the prompt) and indent one line with four spaces and then the next line with a tab, the two lines may appear to be at the same level of indentation. But you receive this exception because Python maps the tab to eight spaces. The best way to avoid this problem is to use only spaces in Python code. If you must use tabs for indentation, or if you're dealing with code that uses tabs, be sure never to mix them with spaces.

If you are using the basic interactive shell mode or the IDLE Python shell, you've likely noticed that you need an extra line after the outermost level of indentation:

```
>>> x = 1  
... if x == 1:  
...     y = 2
```

```

...     if y > 0:
...         z = 2
...         v = 0
...
>>> x = 2

```

No line is necessary after the line `z = 2`, but in a basic shell one is needed after the line `v = 0`. This line is unnecessary if you're using Jupyter notebooks or placing your code in a module in a file.

The second form of exception occurs if you indent a statement in a block less than the legal amount:

```

x = 1
if x == 1:
    y = 2
    z = 2
File "<tokenize>", line 4
    z = 2
    ^
IndentationError: unindent does not match any outer indentation level

```

In this example, the line containing `z = 2` isn't lined up properly below the line containing `y = 2`. This form is rare, but I mention it again because in a similar situation, it may be confusing.

Python allows you to indent any amount and won't complain regardless of how much you vary indentation as long as you're consistent within a single block. Please don't take improper advantage of this flexibility. The recommended standard is to use four spaces for each level of indentation.

Before leaving indentation, I'll cover breaking up statements across multiple lines, which of course is necessary more often as the level of indentation increases. You can explicitly break up a line by using the backslash character. You can also implicitly break any statement between tokens when within a set of `()`, `{ }`, or `[]` delimiters (that is, when typing a set of values in a list, a tuple, or a dictionary; a set of arguments in a function call; or any expression within a set of brackets). You can indent the continuation line of a statement to any level you desire:

```

print('string1', 'string2', 'string3' \
    , 'string4', 'string5')

string1 string2 string3 string4 string5

x = 100 + 200 + 300 \
    + 400 + 500
x

1500

v = [100, 300, 500, 700, 900,
    1100, 1300]

```

```
v  
[100, 300, 500, 700, 900, 1100, 1300]  
  
max(1000, 300, 500,  
     800, 1200)  
  
1200  
  
x = (100 + 200 + 300  
      + 400 + 500)  
x  
  
1500
```

You can break a string with a \ as well. But any indentation tabs or spaces become part of the string, and the line *must* end with the \. To avoid this situation, be sure to use quotes around the segments you are breaking up. Remember that any string literals separated by whitespace are automatically concatenated by the Python interpreter:

```
"strings separated by whitespace "    \  
    '''are automatically'''    ' concatenated'  
  
'strings separated by whitespace are automatically concatenated'  
  
x = 1  
if x > 0:  
    string1 = "this string broken by a backslash will end up \  
               with the indentation tabs in it"  
  
string1  
  
'this string broken by a backslash will end up                         with the  
indentation tabs in it'  
if x > 0:  
    string1 = "this can be easily avoided by splitting the " \  
              "string in this way"  
  
string1  
  
'this can be easily avoided by splitting the string in this way'
```

8.11 Boolean values and expressions

The previous examples of control flow use conditional tests in a fairly obvious manner but never really explain what constitutes true or false in Python or what expressions can be used where a conditional test is needed. This section describes these aspects of Python.

Python has a Boolean object type that can be set to either `True` or `False`. Any expression with a Boolean operation returns `True` or `False`.

8.11.1 Most Python objects can be used as Booleans

In addition, Python is similar to C with respect to Boolean values, in that C uses the integer 0 to mean false and any other integer to mean true. Python generalizes this idea: 0 or empty values are `False`, and any other values are `True`. In practical terms, this means the following:

- The numbers 0, 0.0, and 0+0j are all `False`; any other number is `True`.
- The empty string "" is `False`; any other string is `True`.
- The empty list [] is `False`; any other list is `True`.
- The empty dictionary {} is `False`; any other dictionary is `True`.
- The empty set `set()` is `False`; any other set is `True`.
- The special Python value `None` is always `False`.

We haven't looked at some Python data structures yet, but generally, the same rule applies. If the data structure is empty or 0, it's taken to mean false in a Boolean context; otherwise, it's taken to mean true. Some objects, such as file objects and code objects, don't have a sensible definition of a 0 or empty element, and these objects shouldn't be used in a Boolean context.

8.11.2 Comparison and Boolean operators

You can compare objects by using normal operators: `<`, `<=`, `>`, `>=`, and so forth. `==` is the equality test operator, and `!=` is the "not equal to" test. There are also `in` and `not in` operators to test membership in sequences (lists, tuples, strings, and dictionaries), as well as `is` and `is not` operators to test whether two objects are the same.

Expressions that return a Boolean value may be combined into more complex expressions using the `and`, `or`, and `not` operators. The following code snippet checks to see whether a variable is within a certain range:

```
if 0 < x and x < 10:  
    ...
```

Python offers a nice shorthand for this particular type of compound statement. You can write it as you would in a math paper:

```
if 0 < x < 10:  
    ...
```

Various rules of precedence apply; when in doubt, use parentheses to make sure that Python interprets an expression the way you want it to. Using parentheses is probably a good idea for complex expressions, regardless of whether it's necessary, because it makes clear to future maintainers of the code exactly what's happening. See the Python documentation for more details on precedence.

The rest of this section provides more advanced information. If you're reading this book as you're learning the language, you may want to skip that material for now.

The `and` and `or` operators return objects. The `and` operator returns either the first false object (that an expression evaluates to) or the last object. Similarly, the `or` operator returns either the first true object or the last object. This may seem a little confusing, but it works correctly; if an expression with `and` has even one false element, that element makes the entire expression evaluate as `False`, and that `False` value is returned. If all of the elements are `True`, the expression is `True`, and the last value, which must also be `True`, is returned. The converse is true for `or`; only one `True` element makes the statement logically `True`, and the first `True` value found is returned. If no `True` values are found, the last (`False`) value is returned. In other words, as with many other languages, evaluation stops as soon as a true expression is found for the `or` operator or as soon as a false expression is found for the `and` operator:

```
[2] and [3, 4]
[3, 4]           ← The second element
                  must be evaluated.

[] and 5
[]             ← Only the first element
                  needs to be evaluated.

[2] or [3, 4]
[2]           ← Only the first element
                  needs to be evaluated.

[] or 5
5             ← The second element
                  must be evaluated.
```

The `==` and `!=` operators test to see whether their operands contain the same values. `==` and `!=` are used in most situations, as opposed to `is` and `is not` operators, which test to see whether their operands are the same object:

```
x = [0]
y = [x, 1]
x is y[0]
True           ← They reference
                  the same object.

x = [0]
x is y[0]
False          ← x has been assigned
                  to a different object.

x == y[0]
True
```

Revisit section 5.6 if this example isn't clear.

Quick check: Booleans and truthiness

Decide whether the following statements are true or false: 1, 0, -1, [0], 1 and 0, 1 > 0 or [].

8.12 Writing a simple program to analyze a text file

To give you a better sense of how a Python program works, this section looks at a small example that roughly replicates the UNIX `wc` utility and reports the number of lines, words, and characters in a file. The sample in this listing is deliberately written to be clear to programmers who are new to Python and to be as simple as possible.

Listing 8.1 word_count.py

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

""" Reads a file and returns the number of lines, words,
    and characters - similar to the UNIX wc utility
"""

infile = open('word_count.tst')           ← Opens file
lines = infile.read().split("\n")          ← Reads file; splits into lines
line_count = len(lines)                  ← Gets number of lines with len()

word_count = 0                           | Initializes other counts
char_count = 0

for line in lines:                      ← Iterates through lines
    words = line.split()                ← Splits into words
    word_count += len(words)

    char_count += len(line)            ← Returns number of characters

print("File has {0} lines, {1} words, {2} characters".format(
    line_count, word_count, char_count))   ← Prints answers
```

To test, you can run this sample against a sample file containing the first paragraph of this chapter's summary, as shown in the following listing.

Listing 8.2 word_count.tst

Python provides a complete set of control flow elements, including while and for loops, and conditionals. Python uses the level of indentation to group blocks of code with control elements.

If you are running the preceding code in a notebook, be sure to also execute the cell that creates the file `word_count.tst`. If you have saved the code in a file named `word_count.py`, you could also run the code from a command line:

```
naomi@mac:~/quickpythonbook/code $ python3 word_count.py
```

Either way, as long as the file `word_count.tst` was in the same directory, you would get the following output:

```
File has 4 lines, 30 words, 186 characters
```

This code can give you an idea of a Python program. There isn't much code, and most of the work gets done in three lines of code in the `for` loop. In fact, this program could be made even shorter and more idiomatic, as we'll see in this chapter's lab. Most Pythonistas see this conciseness as one of Python's great strengths.

8.13 Refactoring word_count

Rewrite the word-count program from section 8.1.2 to make it shorter. You may want to look at the string and list operations already discussed, as well as think about different ways to organize the code. You may also want to make the program smarter so that only alphabetic strings (not symbols or punctuation) count as words.

8.13.1 Solving the problem with AI-generated code

In this lab we're *refactoring* some existing code—that is, rewriting it to improve it. The goal is to make the code shorter and optionally to count only words that don't contain symbols or numbers.

To refactor the code, we need to give the AI chatbot the existing code and some guidance as to how we want it refactored:

- 1 Indicate what code we want refactored.
- 2 Give our goal in refactoring: shorter code.
- 3 (Optional) Ask for a change so that only alphabetic strings are counted.

8.13.2 Solutions and discussion

Since both my solution and the AI solutions start from the same program, they will tend to be quite similar. Again, however, the AI versions are less cautious with resources, choosing to read the entire file into memory.

THE HUMAN SOLUTION

The human solution is a bit longer, since again I chose to read the file line by line:

```
""" Reads a file and returns the number of lines, words,
and characters - similar to the UNIX wc utility
"""
```

```

# initialize counts
line_count = 0
word_count = 0
char_count = 0

# open the file
with open('word_count.tst') as infile:
    for line in infile:           ← Processes file line by line
        line_count += 1
        char_count += len(line)
        word_count += len(line.split())

# print the answers using the format() method
print("File has {0} lines, {1} words, {2} characters".format(line_count,
                                                            word_count, char_count))

```

Initializes counts to zero**Processes file line by line**

That means that the count variables all need to be initialized to zero. The other consequence of going line by line is that all of the counts need to be incremented with the values for each line. This approach would be able to handle files of almost any size, but, as we'll see, it does result in code that's a tiny bit longer than AI versions. Put it down to the instincts of an old data wrangler, but in most cases this caution would probably be unnecessary.

THE AI VERSION

In Colaboratory, we can just refactor the code of the preceding word counter program, so the prompt is

Refactor the word-count program above to make it shorter. You may want to look at the string and list, as well as think about different ways to organize the code.

For Copilot, we give it the existing code in the chat window and add the instructions, including the optional task of counting only all-letter words:

Refactor this code to make it shorter. You may want to look at the string and list, as well as think about different ways to organize the code. You may also want to make the program smarter so that only alphabetic strings (not symbols or punctuation) count as words.

The two versions are similar, so let's start from the Copilot version:

```

""" Reads a file and returns the number of lines, words,
    and characters - similar to the UNIX wc utility
"""

with open('word_count.tst') as infile:
    lines = infile.readlines()           ← The readlines method returns
                                         a list of lines (with newlines).

    line_count = len(lines)
    word_count = sum(len(line.split()) for line in lines) ← Generator expression
                                                               to get number of
                                                               words in each line

```

The readlines method returns a list of lines (with newlines).**Generator expression to get number of words in each line**

```
char_count = sum(len(line) for line in lines)
print(f"{line_count} lines, {word_count} words, {char_count} characters")
```

**Generator expression to
get length of each line**

This version reads all of the file in at once but uses the `readlines()` method, which returns a list of lines. This means that it doesn't need to break up the lines, but it might be a problem with a very large file. Once the lines have been read, the number of lines is the line count, and the word and character counts are determined using the `sum` function over generator expressions iterating over the list of lines.

If we want to count only strictly alphabetic strings as words, the `word_count` line is a little more complex: Copilot suggests a list comprehension inside the generator expression to filter on the `isalpha` string method:

```
word_count = sum(len([word for word in line.split() if word.isalpha()]))
for line in lines)
```

Either way, the code Copilot suggests is concise and works fine, as long as the data size is not too much for memory.

The Colaboratory code has an important difference in the body of the code:

```
with open('word_count.tst') as infile:
    lines = infile.read().split("\n")
```

← Splits entire file on newline

This version reads the entire file into memory and then splits it into lines by using the `split` method on the newline character ("`\n`"). This means that it has to perform a split operation on the entire file, which neither the human nor the Copilot version has to do. In fact, this is what the original program does as well, and both programs share a subtle bug. When you split a string, you get a list of substrings broken by the specified string (or whitespace if no string is specified). What is *not* included is the separator string ("`\n`"). Since there are four lines in our sample, separated by three newlines, both the original version and the Colaboratory refactor give character counts that are 3 less than they should be: 186 instead of the correct 189 returned by both the human and Copilot refactors.

In the end, the AI versions performed quite well on this refactoring problem. While neither version was quite as cautious about saving memory as I was, the Copilot version managed to improve performance over the original by removing a `split` function, and it also fixed the bug in the original's character count.

Summary

- Python uses indentation to group blocks of code.
- Python has loops using `while` and `for`, conditionals using `if-elif-else`, and a `match` statement that selects from multiple options.

- Some common `for` loop patterns can be replaced by list, dictionary, or set comprehensions.
- Generator expressions are similar to comprehensions but usually use less memory.
- Python has the Boolean values `True` and `False`, which can be referenced by variables.
- Most Python objects can also be evaluated as Booleans, with zero or empty values being `False` and all other values being `True`.



Functions

This chapter covers

- Defining functions
- Using function parameters
- Passing mutable objects as parameters
- Understanding local and global variables
- Creating and using generator functions
- Creating and using lambda expressions
- Using decorators

This chapter assumes that you’re familiar with function definitions in at least one other computer language and with the concepts that correspond to function definitions, arguments, parameters, and so forth.

9.1 Basic function definitions

The basic syntax for a Python function definition is

```
def name(parameter1, parameter2, . . .):  
    body
```

As it does with control structures, Python uses indentation to delimit the body of the function definition. The following simple example puts the code to calculate a factorial into a function body, so you can call a `fact` function to obtain the factorial of a number:

```
def fact(n):
    '''Return the factorial of the given number.'''
    r = 1
    while n > 0:
        r = r * n
        n = n - 1
    return r
```

Optional documentation string, or docstring

Value sent back to the code calling the function

If the function has a `return` statement, like the preceding one, that value will be returned to the code calling the function and can be assigned to a variable. If no `return` is used, the function will return a `None` value.

You can obtain the value of the docstring (the second line of the preceding function) by printing `fact.__doc__`. The intention of docstrings is to describe the external behavior of a function and the parameters it takes, whereas comments should document internal information about how the code works. Docstrings are strings that immediately follow the first line of a function definition and are usually triple-quoted to allow for multiline descriptions. Browsing tools are available that extract the first line of document strings. It's standard practice for multiline documentation strings to give a synopsis of the function in the first line, follow this synopsis with a blank second line, and end with the rest of the information.

Procedure or function?

In some languages, a function that doesn't return a value is called a *procedure*. Although you can (and will) write functions that don't have a `return` statement, they aren't really procedures. All Python procedures are functions; if no explicit `return` is executed in the procedure body, the special Python value `None` is returned, and if `return result` is executed, the value `result` is immediately returned. Nothing else in the function body is executed after a `return` has been executed. Because Python doesn't have true procedures, I'll refer to both types as *functions*.

Although all Python functions return values, it's up to you whether a function's return value is used:

```
fact(4)
24
x = fact(4)
x
24
```

No variable assigned to the return value.

The value is printed, not saved.

The variable x is assigned to the return value.

Even if a function returns a value, it's not required to capture and use that value. If no variable is assigned to the return value of a function, it will be discarded, although if the function call is the last statement in a cell, its value will be displayed. If a variable is assigned, that variable and its value can be used like any other variable.

9.2 Function parameter options

Most functions need parameters, and each language has its own specifications for how function parameters are defined. Python is flexible and provides three options for defining function parameters. These options are outlined in this section.

9.2.1 Positional parameters

The simplest way to pass parameters to a function in Python is by position. In the first line of the function, you specify variable names for each parameter; when the function is called, the parameters used in the calling code are matched to the function's parameter variables based on their order. The following function computes x to the power of y :

```
def power(x, y):
    r = 1
    while y > 0:
        r = r * x
        y = y - 1
    return r

power(3, 3)
```

27

This method requires that the number of parameters used by the calling code exactly matches the number of parameters in the function definition; otherwise, a `TypeError` exception is raised:

```
power(3)

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-5-1bbb88d7a603> in <cell line: 1>()
      1 power(3)

TypeError: power() missing 1 required positional argument: 'y'
```

Default values

Function parameters can have default values, which you declare by assigning a default value in the first line of the function definition, like so:

```
def fun(arg1, arg2=default2, arg3=default3, . . .)
```

(continued)

Any number of parameters can be given default values. Parameters with default values must be defined as the last ones in the parameter list because Python, like most languages, pairs arguments with parameters on a positional basis. There must be enough arguments to a function that the last parameter in that function's parameter list without a default value gets an argument. See section 9.2.2 for a more flexible mechanism.

The following function also computes `x` to the power of `y`. But if `y` isn't given in a call to the function, the default value of 2 is used, and the function is just the square function:

```
def power(x, y=2):
    r = 1
    while y > 0:
        r = r * x
        y = y - 1
    return r
```

You can see the effect of the default argument in the following interactive session:

```
power(3, 3)
```

```
27
```

```
power(3)
```

```
9
```

9.2.2 Passing arguments by parameter name

You can also pass arguments into a function by using the name of the corresponding function parameter rather than its position. Continuing with the previous interactive example, you can type

```
power(y=2, x=3)
```

```
9
```

Because the arguments to `power` in the final invocation are named, their order is irrelevant; the arguments are associated with the parameters of the same name in the definition of `power`, and you get back 3^2 . This type of argument passing is called *keyword passing*.

Keyword passing, in combination with the default argument capability of Python functions, can be highly useful when you're defining functions with large numbers of possible arguments, most of which have common defaults. Consider a function that's intended to produce a list with information about files in the current directory and that

uses Boolean arguments to indicate whether that list should include information such as file size, last modified date, and so forth for each file. You can define such a function along these lines:

```
def list_file_info(size=False, create_date=False, mod_date=False, ...):
    ...get file names...
    if size:
        # code to get file sizes goes here
    if create_date:
        # code to get create dates goes here
    # do any other stuff desired

    return fileinfostructure
```

and then call it from other code using keyword argument passing to indicate that you want only certain information (in this example, the file size and modification date but *not* the creation date):

```
fileinfo = list_file_info(size=True, mod_date=True)
```

This type of argument handling is particularly suited for functions with very complex behavior, and one place where such functions occur is in a GUI. If you ever use the `tkinter` package to build GUIs in Python, you'll find that the use of optional, keyword-named arguments like this is invaluable.

9.2.3 Variable numbers of arguments

Python functions can also be defined to handle variable numbers of arguments, which you can do in two ways. One way handles the relatively familiar case in which you want to collect an unknown number of arguments at the end of the argument list into a list, which is commonly called `*args` in the argument list. The other method can collect an arbitrary number of keyword-passed arguments, which have no correspondingly named parameter in the function parameter list, into a dictionary, usually named `**kwargs` in the parameter list. These two mechanisms are discussed next.

DEALING WITH AN INDEFINITE NUMBER OF POSITIONAL ARGUMENTS

Prefixing the final parameter name of the function with a `*` causes all excess non-keyword arguments in a call of a function (that is, those positional arguments not assigned to another parameter) to be collected together and assigned as a tuple to the given parameter. The following is a simple way to implement a function to find the maximum in a list of numbers.

First, implement the function:

```
def maximum(*numbers):
    if len(numbers) == 0:
        return None
    else:
        maxnum = numbers[0]           ← numbers is a list of all parameters.

* means capture any
number of parameters.
```

```
for n in numbers[1:]:
    if n > maxnum:
        maxnum = n
return maxnum
```

Putting the * in front of `numbers` makes it capture all positional parameters as a list. Now test the behavior of the function:

```
maximum(3, 2, 8)
8
maximum(1, 5, 9, -2, 2)
9
```

DEALING WITH AN INDEFINITE NUMBER OF ARGUMENTS PASSED BY KEYWORD

An arbitrary number of keyword arguments can also be handled. If the final parameter in the parameter list is prefixed with **, it collects all excess keyword-passed arguments into a dictionary. The key for each entry in the dictionary is the keyword (parameter name) for the excess argument. The value of that entry is the value of the argument. An argument passed by keyword is excess in this context if the keyword by which it was passed doesn't match one of the parameter names in the function definition—for example:

```
def example_fun(x, y, **other):
    print(f"x: {x}, y: {y}, keys in 'other': {list(other.keys())}'")
    other_total = 0
    for k in other.keys():
        other_total = other_total + other[k]
    print(f"The total of values in 'other' is {other_total}")

** indicates capture all
named parameters.
```

←

Named parameters collected
in dictionary other

Using ** will result in all unmatched keyword parameters being collected into a dictionary, with the keys being the names and values being the parameter values.

Trying out this function in an interactive session reveals that it can handle arguments passed in under the keywords `foo` and `bar`, even though `foo` and `bar` aren't parameter names in the function definition:

```
example_fun(2, y="1", foo=3, bar=4)

x: 2, y: 1, keys in 'other': ['foo', 'bar']
The total of values in 'other' is 7
```

9.2.4 Mixing argument-passing techniques

It's possible to use all of the argument-passing features of Python functions at the same time, although it can be confusing if not done with care. The general rule for

using mixed-argument passing is that positional arguments come first, then named arguments, followed by the indefinite positional argument with a single *, and last of all the indefinite keyword argument with **. See the documentation for full details.

Quick check: Functions and parameters

How would you write a function that could take any number of unnamed arguments and print their values out in reverse order?

What do you need to do to create a procedure or void function—that is, a function with no return value?

What happens if you capture the return value of a function with a variable?

9.3 *Mutable objects as arguments*

Arguments are passed in by object reference. The parameter becomes a new reference to the object. For immutable objects (such as tuples, strings, and numbers), what is done with a parameter has no effect outside the function. But if you pass in a mutable object (such as a list, dictionary, or class instance), any change made to the object changes what the argument is referencing outside the function. Reassigning the parameter doesn't affect the argument, as shown in figures 9.1 and 9.2:

```
def f(n, list1, list2):
    list1.append(3)           ← One item added to list1
    list2 = [4, 5, 6]         ←
    n = n + 1
    x = 5
    y = [1, 2]
    z = [4, 5]
    f(x, y, z)
    x, y, z

(5, [1, 2, 3], [4, 5])
```

list2 is replaced locally
with a new list.

Figure 9.1 shows the state of the variables before the code is executed. Note that both `n` and `x` are referring to the constant 5, both `y` and `list1` refer to the list `[1, 2]`, and both `z` and `list2` refer to the list `[4, 5]`. When we execute the function, we get the result illustrated in figure 9.2.

Figures 9.2 illustrates what happens when function `f` is called. The variable `x` isn't changed because it's immutable. Instead, the function parameter `n` is set to refer to the new value of 6. Likewise, variable `z` is unchanged because inside function `f`, its corresponding parameter `list2` was set to refer to a new object, `[4, 5, 6]`. Only `y` sees a change because the actual list it points to was changed.

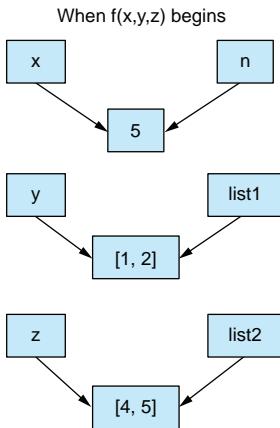


Figure 9.1 At the beginning of function `f()`, both the initial variables and the function parameters refer to the same objects.

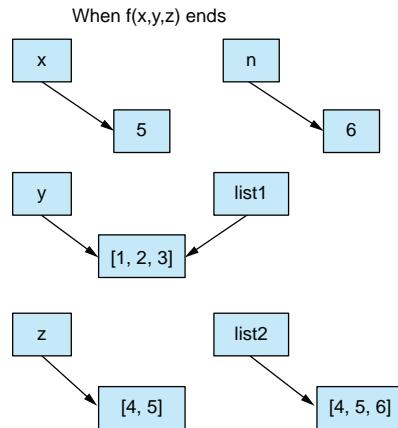


Figure 9.2 At the end of function `f()`, `y` (`list1` inside the function) has been changed internally, whereas `n` and `list2` refer to different objects.

9.3.1 Mutable objects as default values

Passing mutable objects as parameter values can cause bugs but is quite often the most convenient and efficient way to do things. It does mean that you need to be aware that changing that object might have a side effect outside the function.

It's much worse to use a mutable object as a default parameter value and then mutate that object. For example, suppose I'm going to check a list for odd numbers and add them to a list of odd numbers I pass in as a parameter:

```
def odd_numbers(test_list, odds):
    for number in test_list:
        if number % 2:
            odds.append(number)
    return odds

odds = []
odds = odd_numbers([1, 5, 7, 9, 10], odds)
odds

[1, 5, 7, 9]
```

This works, but say you decide you don't want to have to bother with creating an empty list for the odd numbers, so you make an empty list to be the default value of the odds parameter. Now you don't need to explicitly create or pass that list, right?

```
def odd_numbers(test_list, odds=[]):
    for number in test_list:
        if number % 2:
```

Empty list as default parameter value

```
    odds.append(number)
    return odds

odds = odd_numbers([1, 5, 7, 9, 10])
odds

[1, 5, 7, 9]
```

It looks like this works just fine, but if you run the same function a second time, the result changes:

```
odds = odd_numbers([1, 5, 7, 9, 10])
odds

[1, 5, 7, 9, 1, 5, 7, 9]
```

When a default parameter value is used, Python assigns the object to be used as the default when the function is first compiled, and it does not change for the life of the program. So, if you have a mutable object as the default and mutate, as we did by appending items to our list of odd numbers, every time that default value is used it will be the same object, and that object will reflect all of the times the function has been called with it.

Quick check: Mutable function parameters

What would be the result of changing a list or dictionary that was passed into a function as a parameter value? Which operations would be likely to create changes that would be visible outside the function? What steps might you take to minimize that risk?

9.4 Local, nonlocal, and global variables

Let's return to the definition of `fact` from the beginning of this chapter:

```
def fact(n):
    """Return the factorial of the given number."""
    r = 1
    while n > 0:
        r = r * n
        n = n - 1
    return r
```

Both the variables `r` and `n` are *local* to any particular call of the factorial function; changes to them made when the function is executing have no effect on any variables outside the function. Any variables in the parameter list of a function, and any variables created within a function by an assignment (like `r = 1` in `fact`) are local to the function.

You can explicitly make a variable global by declaring it so before the variable is used, using the `global` statement. Global variables can be accessed and changed by the function. They exist outside the function and can also be accessed and changed by other functions that declare them global or by code that's not within a function. The following is an example that shows the difference between local and global variables:

```
def fun():
    global a
    a = 1
    b = 2
```

This example defines a function that treats `a` as a global variable and `b` as a local variable and attempts to modify both `a` and `b`.

Now test this function:

```
a = "one"
b = "two"

fun()
a

1

b

'two'
```

The assignment to `a` within `fun` is an assignment to the global variable `a` also existing outside `fun`. Because `a` is designated `global` in `fun`, the assignment modifies that global variable to hold the value `1` instead of the value `"one"`. The same isn't true for `b`; the local variable called `b` inside `fun` starts out referring to the same value as the variable `b` outside `fun`, but the assignment causes `b` to point to a new value that's local to the function `fun`.

Similar to the `global` statement is the `nonlocal` statement, which causes an identifier to refer to a previously bound variable in the closest enclosing scope. I discuss scopes and namespaces in more detail in chapter 10, but the point is that `global` is used for a top-level variable, whereas `nonlocal` can refer to any variable in an enclosing scope, as the example in the following listing illustrates.

Listing 9.1 File nonlocal.py

```
g_var = 0
nl_var = 0
print("top level-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
def test():
    nl_var = 2
    print("in test-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
```

The diagram illustrates the scope resolution for variables in the `nonlocal.py` script. It shows three occurrences of `g_var` and two occurrences of `nl_var`. The first `g_var` and the first `nl_var` are labeled `g_var in global scope` and `nl_var in global scope` respectively. The second `nl_var` is labeled `nl_var in test (not global)`, indicating it refers to the local variable defined within the `test()` function, not the global one.

```

def inner_test():
    global g_var
    nonlocal nl_var
    g_var = 1
    nl_var = 4
    print("in inner_test-> g_var: {} nl_var: {}".format(g_var,
                                                          nl_var))

inner_test()
print("in test-> g_var: {} nl_var: {}".format(g_var, nl_var))

test()
print("top level-> g_var: {} nl_var: {}".format(g_var, nl_var))

```

The preceding code shows how using the `global` and `nonlocal` keywords can make variables in one scope refer either to a variable at the top level of the module or to the closest variable of the same name in an enclosing scope.

When run, this code prints the following:

```

top level-> g_var: 0 nl_var: 0
in test-> g_var: 0 nl_var: 2
in inner_test-> g_var: 1 nl_var: 4
in test-> g_var: 1 nl_var: 4
top level-> g_var: 1 nl_var: 0

```

Note that the value of the top-level `nl_var` hasn't been affected, which would happen if `inner_test` contained the line `global nl_var`.

The bottom line is that if you want to *assign* to a variable existing outside a function, you must explicitly declare that variable to be `nonlocal` or `global`. But if you're just *reading* a variable that exists outside the function, you don't need to declare it `nonlocal` or `global`. If Python can't find a variable name in the local function scope, it attempts to look up the name in the enclosing scopes and the global scope. Hence, accesses to global variables are automatically sent through to the correct global variable. Personally, I don't recommend using this shortcut. It's much clearer to a reader if all global variables are explicitly declared as `global`. Further, you probably want to limit the use of global variables within functions to rare occasions.

Try this: Global vs. local variables

Assuming that `x = 5`, what will be the value of `x` after `funct_1()` below executes? After `funct_2()` executes?

```

def funct_1():
    x = 3
def funct_2():
    global x
    x = 2

```

9.5 Assigning functions to variables

Functions can be assigned, like other Python objects, to variables, as shown in the following example:

```
def f_to_kelvin(degrees_f):
    return 273.15 + (degrees_f - 32) * 5 / 9
 $\swarrow$  Defines the f_to_kelvin kelvin function

def c_to_kelvin(degrees_c):
    return 273.15 + degrees_c
 $\swarrow$  Defines the c_to_kelvin function

abs_temperature = f_to_kelvin
abs_temperature(32)
 $\swarrow$  Assigns function to variable

273.15
 $\swarrow$ 

abs_temperature = c_to_kelvin
abs_temperature(0)
 $\swarrow$ 

273.15
```

In the preceding code, we define two different functions: one for converting Fahrenheit temperatures to Kelvin and the other for converting Celsius. We can then assign either one of the functions to the variable `abs_temperature` and use that as function, which will call whichever function it is currently assigned to.

You can place functions in lists, tuples, or dictionaries:

```
t = {'FtoK': f_to_kelvin, 'CtoK': c_to_kelvin}
t['FtoK'](32)
 $\swarrow$  Creates dictionary with both functions as values

273.15
 $\swarrow$ 

t['CtoK'](0)
 $\swarrow$  Accesses the f_to_kelvin function as value in dictionary

273.15
```

A variable that refers to a function can be used in exactly the same way as the function itself. This last example shows how you can use a dictionary to call different functions by the value of the strings used as keys. This pattern is common in situations in which different functions need to be selected based on a string value, and in many cases, it takes the place of the `switch` structure found in languages such as C and Java.

9.6 lambda expressions

Short functions like those you just saw can also be defined by using `lambda` expressions of the form

```
lambda parameter1, parameter2, . . . : expression
```

`lambda` expressions are anonymous little functions that you can quickly define inline. Often, a small function needs to be passed to another function, like the `key` function

used by a list's `sort` method. In such cases, a large function is usually unnecessary, and it would be awkward to have to define the function in a separate place from where it's used. The dictionary in the previous subsection can be defined all in one place with

```
t2 = {'FtoK': lambda deg_f: 273.15 + (deg_f - 32) * 5 / 9,
      'CtoK': lambda deg_c: 273.15 + deg_c}
t2['FtoK'](32)
273.15
```

A diagram illustrating the execution of the code. A callout points to the line `t2['FtoK'](32)` with the text "lambda functions as dictionary values". An arrow points from this callout to the line `t2['FtoK']`. Another callout points to the output `273.15`.

This example defines `lambda` expressions as values of the dictionary, rather than using the formally defined functions.

Note that `lambda` expressions don't have a `return` statement because the value of the expression is automatically returned.

9.7 Generator functions

A *generator* function is a special kind of function that you can use to define your own iterators. When you define a generator function, you return each iteration's value using the `yield` keyword. When the generator function is called, it returns a *generator object*, which can be used as an iterator. Each time the generator is called, it runs the code up to `yield`, uses the `yield` to return a value, and then will resume right after the `yield` the next time it's called. The generator will stop returning values when there are no more iterations or it encounters either an empty `return` statement or the end of the function. Local variables in a generator function are saved from one call to the next, unlike in normal functions:

```
def four():
    x = 0
    while x < 4:
        print("in generator, x =", x)
        yield x
        x += 1
    for i in four():
        print(f"Value from generator {i}")

in generator, x = 0
Value from generator 0
in generator, x = 1
Value from generator 1
in generator, x = 2
Value from generator 2
in generator, x = 3
Value from generator 3
```

A diagram illustrating the execution flow of the generator function. Callouts point to specific parts of the code with descriptive text: "Sets initial value of x to 0" points to `x = 0`; "Returns current value of x" points to the `yield x` line; "Increments value of x" points to the `x += 1` line; and "A call to the generator function creates a generator (iterator) for use in a for loop." points to the `for i in four():` line.

When the function `four` is called in the `for` loop, a generator object is returned, and that generator is the iterator that controls the `for` loop.

Note that this generator function has a `while` loop that limits the number of times the generator executes. Depending on how it's used, a generator that doesn't have some condition to halt it could cause an endless loop when called.

yield vs. yield from

Starting with Python 3.3, a new key word for generators, `yield from`, joins `yield`. Basically, `yield from` makes it possible to string generators together. `yield from` behaves the same way as `yield`, except that it delegates the generator machinery to a subgenerator. So in a simple case, you could do this:

```
def subgen(x):
    for i in range(x):
        yield i

def gen(y):
    yield from subgen(y)

for q in gen(6):
    print(q)

0
1
2
3
4
5
```

This example allows the `yield` expression to be moved out of the main generator, making refactoring easier.

You can also use generator functions with `in` to see whether a value is in the series that the generator produces:

```
2 in four()

in generator, x = 0
in generator, x = 1
in generator, x = 2
True

5 in four()

in generator, x = 0
in generator, x = 1
in generator, x = 2
in generator, x = 3
False
```

Quick check: Generator functions

What would you need to modify in the previous code for the function `four()` to make it work for any number? What would you need to add to allow the starting point to also be set?

9.8 Decorators

Because functions are first-class objects in Python, they can be assigned to variables, as you've seen. Functions can also be passed as arguments to other functions and passed back as return values from other functions.

It's possible, for example, to write a Python function that takes another function as its parameter, wraps it in another function that does something related, and then returns the new function. This new combination can be used instead of the original function:

```
def decorate(func):
    print("in decorate function, decorating", func.__name__)
    def wrapper_func(*args):
        print("Executing", func.__name__)
        return func(*args)
    return wrapper_func

def myfunction(parameter):
    print(parameter)

myfunction = decorate(myfunction)

in decorate function, decorating myfunction

myfunction("hello")

Executing myfunction
hello
```

A decorator is syntactic sugar for this process and lets you wrap one function inside another with a one-line addition. It still gives you exactly the same effect as the previous code, but the resulting code is much cleaner and easier to read.

Very simply, using a decorator involves two parts: defining the function that will be wrapping or "decorating" other functions and then using an @ followed by the decorator immediately before the wrapped function is defined. The decorator function should take a function as a parameter and return a function, as follows:

```
def decorate(func):
    print("in decorate function, decorating",
          func.__name__)
    def wrapper_func(*args):
        print("Executing", func.__name__)
        return func(*args)
    return wrapper_func

@decorate
def myfunction(parameter):
    print(parameter)

in decorate function, decorating myfunction
```

Prints the name of function being wrapped (points to the line `func.__name__`)

Returns the wrapped function (points to the line `return wrapper_func`)

myfunction is decorated using @decorate. (points to the line `@decorate`)

The wrapped function is called. (points to the line `print(parameter)`)

```
myfunction("hello")  
  
Executing myfunction  
hello
```

The `decorate` function prints the name of the function it's wrapping when the function is defined, and when it's finished, the decorator returns the wrapped function. This is exactly the same process as in our first example, but instead of the explicit code, `myfunction` is decorated using `@decorate`. Then the wrapped function is called using the function's original name.

Using a decorator to wrap one function in another can be handy for several purposes. In web frameworks such as Django, decorators are used to make sure that a user is logged in before executing a function, and in graphics libraries, decorators can be used to register a function with the graphics framework.

Try this: Decorators

How would you modify the code for the decorator function to remove unneeded messages and enclose the return value of the wrapped function in "`<html>`" and "`</html>`" so that `myfunction ("hello")` would return "`<html>hello</html>`"?

9.9 *Useful functions*

Looking back at the labs in chapters 6 and 7, refactor the code to clean and count the words in a text file into functions for cleaning and processing the data. The goal should be that most of the logic is moved into functions. Use your own judgment as to the types of functions and parameters, but keep in mind that functions should do just one thing, and they shouldn't have any side effects that carry over outside the function.

9.9.1 *Solving the problem with AI-generated code*

This problem is similar to the one in the previous chapter, except that instead of asking the AI generators to refine our code on a line level, we will want to improve the structure of the code. One factor that makes things simpler, both in terms of creating a prompt and in actually producing the code, is that we are starting from existing, working code, which of course we need to give to the current sessions of the AI bots.

In deciding what we want to ask for in the prompt, we want to be clear that we want the code to be rewritten using functions and that we want the bulk of the data processing to be done in those functions.

9.9.2 *Solutions and discussion*

The solutions to this problem involve understanding and modifying existing code. As we'll see, this is not necessarily easier for AI models.

THE HUMAN SOLUTION

The human version works and breaks things down acceptably, if not brilliantly. This was a simple exercise in taking the human solution from the last chapter and moving the individual parts into functions. The trick with making the functions was to create reasonable function names and to make sure that the functions returned values and that the main code used those returned values.

One danger in doing this sort of refactoring is that instead of the function returning a value, the old variables will end up being used as global variables. For example, since the variable `cleaned_line` is used in both the function and the calling code, if we didn't make sure that `cleaned_line` in the calling code was set to the return of the function, it might have ended up being used as a global variable, which could make debugging deceptive:

```
import string
punct = str.maketrans('', '', string.punctuation)

def clean_line(line):
    """changes case and removes punctuation"""
    # make all one case
    cleaned_line = line.lower()

    # remove punctuation
    cleaned_line = cleaned_line.translate(punct)
    return cleaned_line

def get_words(line):
    """splits line into words, and rejoins with newlines"""
    words = line.split()
    return "\n".join(words) + "\n"

def count_words(words):
    """takes list of cleaned words, returns count dictionary"""
    word_count = {}
    for word in words:
        count = word_count.setdefault(word, 0)
        word_count[word] += 1
    return word_count

def word_stats(word_count):
    """Takes word count dictionary and returns top
       and bottom five entries"""
    word_list = list(word_count.items())
    word_list.sort(key=lambda x: x[1])
    least_common = word_list[:5]
    most_common = word_list[-1:-6:-1]
    return most_common, least_common

with open("moby_01.txt") as infile, open("moby_01_clean.txt", "w") as
    outfile:
```

```

for line in infile:
    cleaned_line = clean_line(line)

    cleaned_words = get_words(cleaned_line)

    # write all words for line
    outfile.write(cleaned_words)

moby_words = []
with open('moby_01_clean.txt') as infile:
    for word in infile:
        if word.strip():
            moby_words.append(word.strip())

word_count = count_words(moby_words)

most, least = word_stats(word_count)
print("Most common words:")
for word in most:
    print(word)
print("\nLeast common words:")
for word in least:
    print(word)

```

THE AI SOLUTIONS

For Colaboratory, I put the old code into two cells right above the active cell, and the prompt instructed the chatbot to refactor that code:

Refactor the code in the previous 2 cells into functions for cleaning and processing the data. The goal should be that most of the logic is moved into functions.

The following was generated in Colaboratory:

```

# Function to clean a line of text by converting it to lowercase and
# removing punctuation
def clean_line(line):
    line = line.lower()
    line = line.translate(str.maketrans('', '', string.punctuation))
    return line

# Function to split a line of text into words and write each word to a new
# line in a file
def write_words_to_file(line, output_file):
    words = line.split()
    for word in words:
        output_file.write(word + '\n')

# Function to count the occurrences of each word in a list of words
def count_word_occurrences(words):
    word_counts = {}
    for word in words:

```

```

if word not in word_counts:
    word_counts[word] = 0
word_counts[word] += 1
return word_counts

# Function to find the most/least common words in a dictionary
def find_most_and_least_common_words(word_counts):
    most_common = sorted(word_counts.items(), key=lambda item: item[1],
reverse=True)[:5]
    least_common = sorted(word_counts.items(), key=lambda item: item[1])[-5:]
    return most_common, least_common

# Open the input file for reading
with open('moby_01.txt', 'r') as input_file:
    # Open the output file for writing
    with open('moby_01_clean.txt', 'w') as output_file:
        # Iterate over each line in the input file
        for line in input_file:
            # Clean the line of text
            clean_line = clean_line(line)

            # Write each word in the clean line to the output file
            write_words_to_file(clean_line, output_file)

# Open the file and read the text into a variable.
with open('moby_01_clean.txt', 'r') as f:
    text = f.read()

# Split the text into words.
words = text.split()

# Count the occurrences of each word.
word_counts = count_word_occurrences(words)

# Find the most and least common words.
most_common, least_common = find_most_and_least_common_words(word_counts)

# Print the results.
print("Most common words:")
for word, count in most_common:
    print(f"{word}: {count}")

print("\nLeast common words:")
for word, count in least_common:
    print(f"{word}: {count}")

```

When we run the Colaboratory version, we get a pair of surprises:

```

NameError                                     Traceback (most recent call last)
<ipython-input-4-dd9d43b3f3c2> in <cell line: 34>()
      38     for line in input_file:
      39         # Clean the line of text
---> 40         clean_line = clean_line(line)
      41

```

```

42      # Write each word in the clean line to the output file

<ipython-input-4-dd9d43b3f3c2> in clean_line(line)
    7 def clean_line(line):
    8     line = line.lower()
----> 9     line = line.translate(str.maketrans('', '', string.punctuation))
    10    return line
    11

NameError: name 'string' is not defined

```

The error here is that the string library hasn't been imported. Adding the line

```
import string
```

at the top of the cell or file will fix that one. But after that, we still get an error:

```

-----  

TypeError                                     Traceback (most recent call last)  

<ipython-input-5-dd9d43b3f3c2> in <cell line: 34>()  

    38     for line in input_file:  

    39         # Clean the line of text  
          ← Loops over all lines of input  

---> 40         clean_line = clean_line(line)           ←  

    41  

    42     # Write each word in the clean line to the output file  

TypeError: 'str' object is not callable          clean_line name assigned to  

                                                result of clean_line function

```

That's right, the version that Colaboratory created from its own original code has an error. The traceback shows that in line 40 the return value of the `clean_line` function, which will be a string cleaned of punctuation characters, is assigned the name `clean_line`, the same name as the function. That is legal in Python, but since we are in a loop, the next iteration of that loop will try to use the name `clean_line` as a function; since it now refers to a string, it can't be executed. The code will work if we use a different name to refer to the return value of the function, say, `cleaned_line`:

```

for line in input_file:  

    # Clean the line of text  

    cleaned_line = clean_line(line)  

    # Write each word in the clean line to the output file  

    write_words_to_file(cleaned_line, output_file)

```

But note that we must change the name both where the function is called and where we use its return value three lines later. Otherwise, the Colaboratory refactor is fine, even iterating over the file line by line, where the original read the entire file at once.

For Copilot, the process was similar but easier. I pasted the two programs into the same file; then selected all of the code, right-clicked, and opened Copilot; and finally started an inline chat with a similar prompt:

Refactor the selected code into functions for cleaning and processing the data. The goal should be that most of the logic is moved into functions.

Copilot then produced a split screen diff-style listing of the old code and its proposed refactor, which was accepted without changes.

The following solution was offered by Copilot:

```
# Generated by Copilot
import string

def clean_line(line):
    # Convert the line to lowercase
    line = line.lower()

    # Remove punctuation from the line
    line = line.translate(str.maketrans('', '', string.punctuation))

    return line

def write_words_to_file(words, output_file):
    # Write each word to the output file
    for word in words:
        output_file.write(word + '\n')

def count_word_occurrences(words):
    # Count the occurrences of each word using a dictionary
    word_counts = {}
    for word in words:
        if word in word_counts:
            word_counts[word] += 1
        else:
            word_counts[word] = 1

    return word_counts

def print_common_words(word_counts, n):
    # Sort the word counts in descending order
    sorted_word_counts = sorted(word_counts.items(),
                                key=lambda x: x[1], reverse=True)

    # Print the n most common words and their occurrences
    print(f"{n} most common words:")
    for word, count in sorted_word_counts[:n]:
        print(f"{word}: {count}")

def print_least_common_words(word_counts, n):
    # Sort the word counts in descending order
    sorted_word_counts = sorted(word_counts.items(),
                                key=lambda x: x[1], reverse=True)

    # Print the n least common words and their occurrences
    print(f"{n} least common words:")
    for word, count in sorted_word_counts[-n:]:
        print(f"{word}: {count}")
```

```

def process_file(input_file_path, output_file_path):
    # Open the input file for reading
    with open(input_file_path, 'r') as input_file:
        # Open the output file for writing
        with open(output_file_path, 'w') as output_file:
            # Iterate over each line in the input file
            for line in input_file:
                # Clean the line
                cleaned_line = clean_line(line)

                # Split the line into words
                words = cleaned_line.split()

                # Write the words to the output file
                write_words_to_file(words, output_file)

def main():
    input_file_path = 'moby_01.txt'
    output_file_path = 'moby_01_clean.txt'

    # Process the file
    process_file(input_file_path, output_file_path)

    # Open the file
    with open(output_file_path, 'r') as file:
        # Read the file content
        content = file.read()

        # Split the content into words
        words = content.split()

        # Count word occurrences
        word_counts = count_word_occurrences(words)

        # Print the five most common words and their occurrences
        print_common_words(word_counts, 5)

        # Print the five least common words and their occurrences
        print_least_common_words(word_counts, 5)

if __name__ == "__main__":
    main()

```

The Copilot refactored version doesn't have any errors and in general is nicely done, but it still reads in the entire file at once rather than line by line, just as it did in the original code.

One problem with both of the refactors is that the functions created are documented with comments beginning with `#`. For the Colaboratory version, those comments are before the function definition, while for the Copilot version they are after the definition:

```

# Function to count the occurrences of each word in a list of words
def count_word_occurrences(words):

```

```
def clean_line(line):
    # Convert the line to lowercase
```

Of course, having documentation is better than not having it, but for functions, the preferred Python style would be to use a docstring, a triple-quoted string, right after the definition, as is done in the human version:

```
def clean_line(line):
    """changes case and removes punctuation"""\n\n← | Docstring
```

This isn't purely an esthetic thing—the comments will be totally ignored when the code is loaded, while a docstring will become the value of the function object's `__doc__` property available for use by documentation tools and Python's `help()` function.

Overall, both AI tools were able to handle changing the structure of our previous code so that the main functionality was in functions. Even with the minor error made by Colaboratory, the ability to refactor structure, not just lines of code, is quite helpful.

Summary

- The basic function is defined using the `def` keyword, a name, parenthesis, and colon followed by a code block with an optional `return` statement at the end.
- Arguments may be passed by position or by parameter name.
- Default values may be provided for function parameters.
- Functions can collect arguments into tuples, giving you the ability to define functions that take an indefinite number of arguments.
- Functions can collect arguments into dictionaries, giving you the ability to define functions that take an indefinite number of arguments passed by parameter name.
- External variables can easily be accessed within a function by using the `global` or `nonlocal` keywords.
- The `lambda` keyword is used to create anonymous inline functions.
- Generator objects are created by functions that use `yield` or `yield from` instead of `return`.
- Functions are first-class objects in Python, which means that they can be assigned to variables, accessed by way of variables, and decorated.

10

Modules and scoping rules

This chapter covers

- Defining a module
- Writing a first module
- Using the `import` statement
- Modifying the module search path
- Making names private in modules
- Importing standard library and third-party modules
- Understanding Python scoping rules and namespaces

Modules are used to organize larger Python projects. The Python standard library is split into modules to make it more manageable. You don't need to organize your own code into modules, but if you're writing any programs that are more than a few pages long or any code that you want to reuse, you should probably do so.

10.1 What is a module?

A *module* is a file containing code. It defines a group of Python functions or other objects, and the name of the module is derived from the name of the file.

Modules most often contain Python source code, but they can also be compiled C or C++ object files. Compiled modules and Python source modules are used the same way.

As well as grouping related Python objects, modules help avert name-clash problems. You might write a module for your program called `mymodule`, which defines a function called `reverse`. In the same program, you might also want to use somebody else's module called `othermodule`, which also defines a function called `reverse` that does something different from your `reverse` function. In a language without modules, it would be impossible to use two different functions named `reverse`. In Python, the process is trivial; you refer to the functions in your main program as `mymodule.reverse` and `othermodule.reverse`.

Using the module names keeps the two `reverse` functions straight because Python uses namespaces. A *namespace* is essentially a dictionary of the identifiers available to a block, function, class, module, and so on. I will discuss namespaces a bit more at the end of this chapter, but be aware that each module has its own namespace, which helps prevent naming conflicts.

Modules are also used to make Python itself more manageable. Most standard Python functions aren't built into the core of the language but are provided via specific modules, which you can load as needed.

Using modules with Colaboratory

Since we are using Colaboratory as our preferred Python environment, we need to say a bit about using modules with it. To use a module with Colaboratory, that module needs to be uploaded to your current notebook's session, which you can do by clicking the Files icon at the left, and then the upload icon at the top.

In the notebooks that go with this book, there is code to either create a module or fetch it from the code repository so that it's available when you need it. In the notebook for this chapter, for example, the first code cell contains code to write the first module, and you should execute that code before trying to access the module in the following cells.

If you are using a Python interpreter installed on your local machine instead, then you will need to manually copy any modules to the current directory where you are using the Python interpreter.

10.2 A first module

The best way to learn about modules is probably to make one, which we start in this section.

Colaboratory doesn't easily support creating or editing a simple Python module, but there are some workarounds. For the examples we discuss in the text, there will be code

to write the contents of the module to disk so that you can import the resulting module. Be sure to execute the cell that contains such code.

The following is what the cell in the notebook uses to create that module file on disk. Note that it puts the entire file's contents in triple single quotes and then uses the `open` function to open a file for writing and the `write` method to write to the file:

```
open("mymath.py", "w").write(  
    """  
    mymath - our example math module  
    pi = 3.14159  
    def area(r):  
        """area(r): return the area of a circle with radius r."""  
        return(pi * r * r)  
    """)
```

Listing 10.1 is the plain listing of `mymath.py`. Going forward, we'll use this style for file listings, even if in the associated notebooks the code will either write the file or fetch it from the repository.

Listing 10.1 File `mymath.py`

```
"""  
    mymath - our example math module  
    pi = 3.14159  
    def area(r):  
        """area(r): return the area of a circle with radius r."""  
        return(pi * r * r)
```

The code for this module merely assigns `pi` a value and defines a function `area`. The `.py` filename suffix is *strongly* suggested for all Python code files; it identifies that file to the Python interpreter as consisting of Python source code. As with functions, you have the option of putting in a document string as the first line of your module.

Now try out the following:

```
pi  
  
-----  
NameError Traceback (most recent call last)  
<ipython-input-1-f84ab820532c> in <cell line: 1>()  
----> 1 pi  
  
NameError: name 'pi' is not defined  
  
area(2)  
  
-----  
NameError Traceback (most recent call last)  
<ipython-input-2-8be925061d22> in <cell line: 1>()  
----> 1 area(2)  
  
NameError: name 'area' is not defined
```

Those errors mean that Python doesn't have the constant `pi` or the function `area` built in.

Now we can try importing the module:

```
import mymath           ← Loads module from mymath.py
pi

-----
NameError                                Traceback (most recent call last)
<ipython-input-2-85b32257cf04> in <cell line: 2>()
      1 import mymath
----> 2 pi

NameError: name 'pi' is not defined
```

You've brought in the definitions for `pi` and `area` from the `mymath.py` file, using the `import` statement (which automatically adds the `.py` suffix when it searches for the file defining the module named `mymath`). But the new definitions aren't directly accessible; typing `pi` by itself gave an error, and typing `area(2)` by itself would also give an error.

Instead, you access `pi` and `area` by *prepending* them with the name of the module that contains them:

```
mymath.pi
3.14159
mymath.area(2)
12.56636

mymath.__doc__           ← Contains module's docstring
'mymath - our example math module'

mymath.area.__doc__      ← area function's docstring
'area(r): return the area of a circle with radius r.'
```

Accessing a module's attributes by *prepending* them with the name of the module that contains them guarantees name safety. Another module out there may also define `pi` (maybe the author of that module thinks that `pi` is 3.14 or 3.14159265), but that module is of no concern. Even if that other module is imported, its version of `pi` will be accessed by `othermodulename.pi`, which is different from `mymath.pi`. This form of access is often referred to as *qualification* (that is, the variable `pi` is being qualified by the module `mymath`). You may also refer to `pi` as an *attribute* of `mymath`.

Definitions within a module can access other definitions within that module without prepending the module name. The `mymath.area` function accesses the `mymath.pi` constant as just `pi`.

If you want to, you can also specifically ask for names from a module to be imported in such a manner that you don't have to prepend them with the module name by using the format `from modulename import attribute`:

```
from mymath import pi
pi
3.14159
area(2)

-----
NameError                                 Traceback (most recent call last)
<ipython-input-10-8be925061d22> in <cell line: 1>()
----> 1 area(2)

NameError: name 'area' is not defined
```

The name `pi` is now directly accessible because you specifically requested it by using `from mymath import pi`. The function `area` still needs to be called as `mymath.area`, though, because it wasn't explicitly imported.

Reloading a module

While developing and testing a module interactively, you make changes to a module and then reimport it into the current session to test it. But if you change your module on disk or upload a new version to Colaboratory, retying the `import` command won't cause it to load again, because Python tracks the loaded modules and won't load the same module twice.

To load a fresh version of the module from disk you have two options: you can go to the Runtime menu in Colaboratory and select Restart Session, which gives you a clean, new session of the Python interpreter; or you can use the `reload` function from the `importlib` module for this purpose. The `importlib` module provides an interface to the mechanisms behind importing modules:

```
import mymath, importlib
importlib.reload(mymath)

<module 'mymath' from '/content/mymath.py'>
```

When a module is reloaded (or imported for the first time), all of its code is parsed. A syntax exception is raised if an error is found. On the other hand, if everything is okay, a `.pyc` file (for example, `mymath.pyc`) containing Python byte code is created.

Reloading a module doesn't put you back into exactly the same situation as when you start a new session and import it for the first time. But the differences won't normally cause you any problems. If you're interested, you can look up `reload` in the section on the `importlib` module in the Python language reference, found at <https://docs.python.org/3/reference/import.html> in this page's `importlib` section, to find the details.

Modules don't need to be used only from the interactive Python shell, of course. You can also import them into scripts (or other modules, for that matter); enter suitable `import` statements at the beginning of your program file. Internally to Python, the interactive session and a script are considered to be modules as well.

To summarize:

- A module is a file defining one or more Python objects.
- If the name of the module file is `modulename.py`, the Python name of the module is `modulename`.
- You can bring a module named `modulename` into use with the `import modulename` statement. After this statement is executed, objects defined in the module can be accessed as `modulename.objectname`.
- Specific names from a module can be brought directly into your program by using the `from modulename import objectname` statement. This statement makes `objectname` accessible to your program without your needing to prepend it with `modulename`, and it's useful for bringing in names that are often used.

10.3 The `import` statement

The `import` statement takes three different forms. The most basic is

```
import modulename
```

which searches for a Python module of the given name, parses its contents, and makes it available. The importing code can use the contents of the module, but any references by that code to names within the module must still be prepended with the module name. If the named module isn't found, an error is generated. I will discuss exactly where Python looks for modules in section 10.4.

The second form permits specific names from a module to be explicitly imported into the code:

```
from modulename import name1, name2, name3, . . .
```

Each of `name1`, `name2`, and so forth from within `modulename` is made available to the importing code; code after the `import` statement can use any of `name1`, `name2`, `name3`, and so on without your prepending the module name.

Finally, there's a general form of the `from ... import ...` statement:

```
from modulename import *
```

The `*` stands for all the exported names in `modulename`. `from modulename import *` imports all public names from `modulename`—that is, those that don't begin with an underscore—and makes them available to the importing code without the necessity of prepending the module name. But if a list of names called `__all__` exists in the

module (or the package’s `__init__.py`), the names are the ones imported, whether or not they begin with an underscore.

For example, if we used this style with `mymath.py`:

```
from mymath import *
```

We would be able to directly access both `pi` and `area()` without using the module name.

You should take care when using this particular form of importing. If two modules define the same name, and you import both modules using this form of importing, you’ll end up with a name clash, and the name from the second module will replace the name from the first. This technique also makes it more difficult for readers of your code to determine where the names you’re using originate. When you use either of the two previous forms of the import statement, you give your reader explicit information about where they’re from.

But some modules (such as `tkinter`) name their functions to make it obvious where they originate and to make it unlikely that name clashes will occur. It’s also common to use the general import to save keystrokes when using an interactive shell.

10.4 *The module search path*

Exactly where Python looks for modules is defined in a variable called `path`, which you can access through a module called `sys`. Enter the following:

```
import sys
sys.path
```

```
import sys
sys.path
['/content',
 '/env/python',
 '/usr/lib/python310.zip',
 '/usr/lib/python3.10',
 '/usr/lib/python3.10/lib-dynload',
 '',
 '/usr/local/lib/python3.10/dist-packages',
 '/usr/lib/python3/dist-packages',
 '/usr/local/lib/python3.10/dist-packages/IPython/extensions',
 '/root/.ipython']
```

This listing will depend
on your Python setup.

The value shown depends on the configuration of your system. Regardless of the details, the `sys.path` indicates a list of directories that Python searches (in order) when attempting to execute an `import` statement. The first module found that satisfies the `import` request is used. If there’s no satisfactory module in the module search path, an `ImportError` exception is raised.

The `sys.path` variable is initialized from the value of the environment (operating system) variable `PYTHONPATH`, if it exists, or from a default value that’s dependent on

your installation. In addition, whenever you run a Python script, the `sys.path` variable for that script has the directory containing the script inserted as its first element, which provides a convenient way of determining where the executing Python program is located. In an interactive session such as the previous one, the first element of `sys.path` is set to the empty string, which Python takes as meaning that it should first look for modules in the current directory.

10.4.1 Where to place your own modules

In the example that starts this chapter, the `mymath` module is accessible to Python because (1) when you execute Python interactively, the first element of `sys.path` is "", telling Python to look for modules in the current directory, and (2) you uploaded the `mymath.py` file to Colaboratory's current directory. In a production environment, neither of these conditions would necessarily be true. You won't be running Python interactively, and Python code files won't be located in your current directory. To ensure that your programs can use the modules you coded, you need to

- Place your modules in one of the directories that Python normally searches for modules.
- Place all the modules used by a Python program in the same directory as the program.
- Create a directory (or directories) to hold your modules and modify the `sys.path` variable so that it includes this new directory (or directories).

Of these three options, the first is apparently the easiest and is also an option that you should *never* choose unless your version of Python includes local code directories in its default module search path. Such directories are specifically intended for site-specific code (that is, code specific to your machine) and aren't in danger of being overwritten by a new Python install because they're not part of the Python installation. If your `sys.path` refers to such directories, you can put your modules there.

The second option is a good choice for modules that are associated with a particular program. Just keep them with the program.

The third option is the right choice for site-specific modules that will be used in more than one program at that site. You can modify `sys.path` in various ways. You can assign to it in your code, which is easy, but doing so hardcodes directory locations into your program code. You can set the `PYTHONPATH` environment variable, which is relatively easy, but it may not apply to all users at your site; or you can add it to the default search path by using a `.pth` file.

Examples of how to set `PYTHONPATH` are in the Python documentation in the Python Setup and Usage section (under Command Line and Environment). The directory or directories you set it to are prepended to the `sys.path` variable. If you use `PYTHONPATH`, be careful that you don't define a module with the same name as one of the existing library modules that you're using. If you do that, your module will be found before the library module. In some cases, this may be what you want, but probably not often.

You can avoid this problem by using a .pth file. In this case, the directory or directories you added will be appended to `sys.path`. The last of these mechanisms is best illustrated by an example. On Windows, you can place a .pth file in the directory pointed to by `sys.prefix`. Assume your `sys.prefix` is `c:\program files\python` and place the file in the following listing in that directory.

Listing 10.2 File mymodules.pth

```
mymodules
c:\Users\naomi\My Documents\python\modules
```

The next time a Python interpreter is started, `sys.path` will have `c:\program files\python\mymodules` and `c:\Users\naomi\My Documents\python\modules` added to it, if they exist. Now you can place your modules in these directories. Note that the `mymodules` directory still runs the danger of being overwritten with a new installation. The `modules` directory is safer. You also may have to move or create a `mymodules.pth` file when you upgrade Python. See the description of the `site` module in the documentation for the Python standard library if you want more details on using .pth files.

10.5 Private names in modules

I mentioned earlier in the chapter that you can enter `from module import *` to import *almost* all names from a module. The exception is that identifiers in the module beginning with an underscore can't be imported with `from module import *`. People can write modules that are intended for importation with `from module import *` but still keep certain functions or variables from being imported. By starting all internal names (that is, names that shouldn't be accessed outside the module) with an underscore, you can ensure that `from module import *` brings in only those names that the user will want to access.

To see this technique in action, assume that you have a file called `modtest.py` containing the code in the following listing.

Listing 10.3 File modtest.py

```
"""modtest: our test module"""
def f(x):
    return x
def _g(x):      ←————| Private function name
    return x
a = 4
_b = 2          ←————| Private variable name
```

Now start up an interactive session and enter the following:

```
from modtest import *
f(3)
```

```
_g(3)

-----
NameError                                Traceback (most recent call last)
<ipython-input-15-787abdf7161a> in <cell line: 1>()
----> 1 _g(3)

NameError: name '_g' is not defined

a

4

_b

-----
NameError                                Traceback (most recent call last)
<ipython-input-17-8352e74fe22a> in <cell line: 1>()
----> 1 _b

NameError: name '_b' is not defined
```

As you can see, the names `f` and `a` are imported, but the names `_g` and `_b` remain hidden outside `modtest`. Note that this behavior occurs only with `from ... import *`. You can do the following to access `_g` or `_b`:

```
import modtest
modtest._b

2

from modtest import _g
_g(5)

5
```

The convention of leading underscores to indicate private names is used throughout Python, not just in modules.

10.6 Library and third-party modules

At the beginning of this chapter, I mentioned that the standard Python distribution is split into modules to make it more manageable. After you've installed Python, all the functionality in these library modules is available to you. All that's needed is to import the appropriate modules, functions, classes, and so forth explicitly, before you use them.

Many of the most common and useful standard modules are discussed throughout this book. But the standard Python distribution includes far more than what this book describes. At the very least, you should browse the table of contents of the documentation for the Python standard library.

Available third-party modules and links to them are identified in the Python Package Index, which I discuss in chapter 19. You need to download these modules and install them in a directory in your module search path to make them available for import into your programs.

Quick check: Modules

Suppose that you have a module called `new_math` that contains a function called `new_divide`. What are the ways that you might import and then use that function? What are the pros and cons of each method?

Suppose that the `new_math` module contains a function called `_helper_math()`. How will the underscore character affect the way that `_helper_math()` is imported?

10.7 Python scoping rules and namespaces

Python’s scoping rules and namespaces will become more interesting as your experience as a Python programmer grows. If you’re new to Python, you probably don’t need to do anything more than quickly read through this section to get the basic ideas. For more details, look up *namespaces* in the documentation for the Python standard library.

The core concept here is that of a namespace. A *namespace* in Python is a mapping from identifiers to objects—that is, how Python keeps track of what variables and identifiers are active and what they point to. So a statement like `x = 1` adds `x` to a namespace (assuming that it isn’t already there) and associates it with the value 1. When a block of code is executed in Python, it has three namespaces: *local*, *global*, and *built-in* (see figure 10.1).

When an identifier is encountered during execution, Python first looks in the *local namespace* for it. If the identifier isn’t found, the *global namespace* is checked. If the identifier still hasn’t been found, the *built-in namespace* is checked. If it doesn’t exist there, this situation is considered to be an error, and a `NameError` exception occurs.

For a module, a command executed in an interactive session, or a script running from a file, the global and local namespaces are the same. Creating any variable or function or importing anything from another module results in a new entry, or *binding*, being made in this namespace.

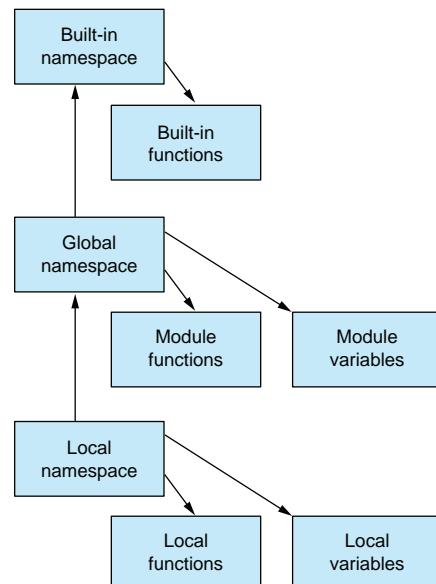


Figure 10.1 The order in which namespaces are checked to locate identifiers

But when a function call is made, a local namespace is created, and a binding is entered in it for each parameter of the call. Then a new binding is entered into this local namespace whenever a variable is created within the function. The global namespace of a function is the global namespace of the containing block of the function (that of the module, script file, or interactive session). It's independent of the dynamic context from which it's called.

In all of these situations, the built-in namespace is that of the `__builtins__` module. This module contains, among other things, all the built-in functions you've encountered (such as `len`, `min`, `max`, `int`, `float`, `list`, `tuple`, `range`, `str`, and `repr`) and the other built-in classes in Python, such as the exceptions (like `NameError`).

One thing that sometimes trips up new Python programmers is the fact that you can override items in the built-in module. If, for example, you create a list in your program and put it in a variable called `list`, you can't subsequently use the built-in `list` function. The entry for your list is found first. There's no differentiation between names for functions and modules and other objects. The most recent occurrence of a binding for a given identifier is used.

Enough talk—it's time to explore some examples. The examples use two built-in functions: `locals` and `globals`. These functions return dictionaries containing the bindings in the local and global namespaces, respectively.

Let's see what those two namespaces are in our current session:

```
locals()

{'__name__': '__main__',
 '__doc__': 'Automatically created module for IPython interactive
environment',
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
 '__builtins__': <module 'builtins' (built-in)>,
 '_ih': [],
 ... (Many other entries}

globals()

{'__name__': '__main__',
 '__doc__': 'Automatically created module for IPython interactive
environment',
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
 '__builtins__': <module 'builtins' (built-in)>,
 '_ih': [],
 ... (Many other entries}
```

The local and global namespaces for this notebook session are the same. They have three initial key-value pairs that are for internal use: (1) a documentation string

`__doc__`, (2) the main module name `__name__` (which, for interactive sessions and scripts run from files is always `__main__`), and (3) the module used for the built-in namespace `__builtins__` (the module `__builtins__`).

If you continue by creating a variable and importing from modules, you see several bindings created (all the way at the bottom of the long list):

```

z = 2
import math
from cmath import cos
globals()

'z': 2,
'math': <module 'math' (built-in)>,
'cos': <function cmath.cos(z, /)>

locals()

'z': 2,
'math': <module 'math' (built-in)>,
'cos': <function cmath.cos(z, /)>

math.ceil(3.4)

4

```

As expected, the local and global namespaces continue to be equivalent. Entries have been added for `z` as a number, `math` as a module, and `cos` from the `cmath` module as a function.

You can use the `del` statement to remove these new bindings from the namespace (including the module bindings created with the `import` statements):

```

del z, math, cos
locals()

{'__builtins__': <module 'builtins' (built-in)>, '__package__': None,
 '__name__': '__main__', '__doc__': None}

math.ceil(3.4)

-----
NameError Traceback (most recent call last)
<ipython-input-26-307799385d47> in <cell line: 1>()
----> 1 math.ceil(3.4)

NameError: name 'math' is not defined

import math
math.ceil(3.4)

4

```

The result isn't drastic, because you're able to import the `math` module and use it again. Using `del` in this manner can be handy when you're in the interactive mode.

NOTE Using `del` and then `import` again won't pick up changes made to a module on disk. It isn't removed from memory and then loaded from disk again. The binding is taken out of and then put back into your namespace. You still need to use `importlib.reload` if you want to pick up changes made to a file.

For the trigger-happy, yes, it's also possible to use `del` to remove the `__doc__`, `__main__`, and `__builtins__` entries. But resist doing this, because it wouldn't be good for the health of your session!

Now look at a function created in an interactive session:

```
def f(x):\n    print("Entry local: ", locals())\n    y = x\n    print("Exit local: ", locals())\n\nz = 2\n\nf(z)\n\nEntry local: {'x': 2}\nExit local: {'x': 2, 'y': 2}
```

When you look at the locals from inside the function, you see that, as expected, upon entry the parameter `x` is the original entry in `f`'s local namespace, but by the time the function ends, `y` has been added.

In a production environment, you normally call functions that are defined in modules. Their global namespace is that of the module in which the functions are defined. Assume that you've created the file in the following listing.

Listing 10.4 File scopetest.py

```
"""scopetest: our scope test module"""\nv = 6\n\ndef f(x):\n    """f: scope test function"""\n    print("global: ", list(globals().keys()))\n    print("entry local:", locals())\n    y = x\n    w = v\n    print("exit local:", locals().keys())
```

Note that you'll be printing only the keys (identifiers) of the dictionary returned by `globals` to reduce clutter in the results. You print only the keys because modules are optimized to store the whole `__builtins__` dictionary as the value field for the `__builtins__` key:

```

import scopetest
z = 2
scopetest.f(z)

global:  ['__name__', '__doc__', '__package__', '__loader__', '__spec__',
          '__file__', '__cached__', '__builtins__', 'v', 'f']
entry local: {'x': 2}
exit local: dict_keys(['x', 'w', 'y'])

```

Now the global namespace is that of the `scopetest` module and includes the function `f` and integer `v` (but not `z` from your interactive session). Thus, when creating a module, you have complete control over the namespaces of its functions.

10.7.1 The built-in namespace

I've covered local and global namespaces. Next, I move on to the built-in namespace. This example introduces another built-in function, `dir`, which, given a module, returns a list of the names defined in it:

```

dir(__builtins__)

['ArithmetricError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EncodingWarning', 'EnvironmentError', 'Exception',
 'False', 'FileExistsError', 'FileNotFoundException', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
 'OverflowError', 'PendingDeprecationWarning', 'PermissionError',
 'ProcessLookupError', 'RecursionError', 'ReferenceError',
 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '__IPYTHON__',
 '__build_class__', '__debug__', '__doc__', '__import__', '__loader__',
 '__name__', '__package__', '__spec__', 'abs', 'aiter', 'all', 'anext',
 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes',
 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright',
 'credits', 'delattr', 'dict', 'dir', 'display', 'divmod', 'enumerate',
 'eval', 'exec', 'execfile', 'filter', 'float', 'format', 'frozenset',
 'get_ipython', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex',
 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license',
 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object',
 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr',
 'reversed', 'round', 'runfile', 'set', 'setattr', 'slice', 'sorted',
 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

```

There are a lot of entries here. Those entries ending in `Error` and `Exit` are the names of the exceptions built into Python, which I discuss in chapter 14.

The last group (from `abs` to `zip`) is built-in functions of Python. You've already seen many of these functions in this book and will see more, but I don't cover all of them here. If you're interested, you can find details on the rest in the documentation for the Python standard library. You can also easily obtain the documentation string for any of them by using the `help()` function or by printing the docstring directly:

```
print(max.__doc__)

max(iterable, *, default=obj, key=func) -> value
max(arg1, arg2, *args, *, key=func) -> value

With a single iterable argument, return its biggest item. The
default keyword-only argument specifies an object to return if
the provided iterable is empty.
With two or more arguments, return the largest argument.
```

As I mentioned earlier, it's not unheard of for a new Python programmer to inadvertently override a built-in function:

```
list("Peyto Lake")

['P', 'e', 'y', 't', 'o', ' ', 'L', 'a', 'k', 'e']

list = [1, 3, 5, 7]           ← Makes list refer to [1, 3, 5, 7]
list("Peyto Lake")

-----
TypeError                                         Traceback (most recent call last)
<ipython-input-39-7845f2807d9f> in <cell line: 2>()
      1 list = [1, 3, 5, 7]
----> 2 list("Peyto Lake")

TypeError: 'list' object is not callable
```

The Python interpreter won't look beyond the new binding for `list` as a `list`, even though you're using the built-in `list` function syntax.

The same thing happens, of course, if you try to use the same identifier twice in a single namespace. The previous value is overwritten, regardless of its type:

```
import mymath
mymath = mymath.area
mymath.pi

AttributeError                                         Traceback (most recent call last)
<ipython-input-40-4e7325de62ae> in <cell line: 3>()
      1 import mymath
      2 mymath = mymath.area
----> 3 mymath.pi

AttributeError: 'function' object has no attribute 'pi'
```

When you’re aware of this situation, it isn’t a significant problem. Reusing identifiers, even for different types of objects, wouldn’t make for the most readable code anyway. If you do inadvertently make one of these mistakes when in interactive mode, it’s easy to recover. You can use `del` to remove your binding, to regain access to an overridden built-in, or to import your module again to regain access:

```
del list
list("Peyto Lake")

['P', 'e', 'y', 't', 'o', ' ', 'L', 'a', 'k', 'e']

import mymath
mymath.pi
```

3.14159

The `locals` and `globals` functions can be useful as simple debugging tools. The `dir` function doesn’t give the current settings, but if you call it without parameters, it returns a sorted list of the identifiers in the local namespace. This practice helps you catch the mistyped variable error that compilers usually catch for you in languages that require declarations:

```
x1 = 6
x1 = x1 - 2      ← The lowercase letter "L" is
x1                   not the same as number "1."
6
dir()

...               ← Jupyter notebook variables
'mymath',
'quit',
'x1',
'x1',
'z']
```

The variables tool in Colaboratory, accessed via the `{x}` icon at the left, also allows you to view the variable values currently active in your notebook session.

Quick check: Namespaces and scope

Consider a variable `width` that’s in the module `make_window.py`. In which of the following contexts is `width` in scope?

- (A) Within the module itself
- (B) Inside the `resize()` function in the module
- (C) Within the script that imported the `make_window.py` module

10.8 Creating a module

Package the functions created at the end of chapter 9 as a standalone module. Although you can include code to run the module as the main program, the goal should be for the functions to be completely usable from another script. To test, create a new Jupyter notebook and write the code to load and use the module to get the same results as the code in chapter 9.

Uploading a module to Colaboratory

If you create your own module using a code editor or IDE, you will need to upload it to Colaboratory. To do that, you should select the file folder icon at the right and then upload the document icon (as shown in figure 10.2), which will open a file upload dialog.

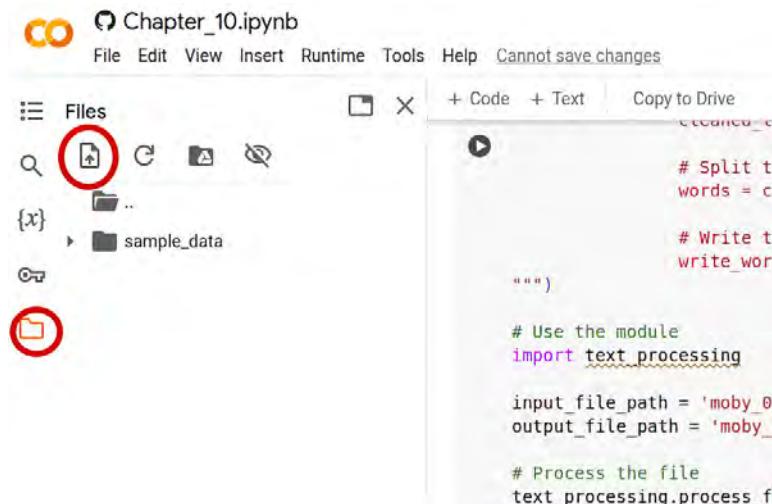


Figure 10.2 The file and upload icons in Colaboratory

10.8.1 Solving the problem with AI-generated code

This lab isn't about writing new code but again involves repackaging existing code using a new structure. The functions we created in the previous chapter now need to be packaged into a separate file or module, and the remaining code that uses the functions needs to be modified to import the module and correctly call the functions from that module.

To be sure we're separating things properly, our prompt will need to be explicit about the separation of the function code into one file and the client code into another.

Finally, we'll need to be sure that the bots have access to the code we refactored in the previous chapter, either in specific cells in our current notebook or in the current file if we are using VS Code and Copilot.

10.8.2 Solutions and discussion

This problem doesn't require much actual coding, just the reorganization and tweaking of existing code. The functions can be simply copied and pasted into a new module file, while the new main file needs to be changed so that the module functions are imported and then called correctly, depending on how they were imported.

For example, it would be legal code (but questionable Python style) to import everything into the active namespace:

```
from clean_string_module import *
```

In this case, the bare function names could still be used in the main program. If, on the other hand, the `import clean_string_module` style of import is used, the module name would need to be attached to every function call.

THE HUMAN SOLUTION

The human solution is pretty straightforward; it just moves the functions to module file `text_processing_author.py` and then imports the module and adds the module name in front of all calls to functions from the module:

```
with open("text_processing_author.py", "w") as f:
    f.write(r'''
# Author's version
import string
punct = str.maketrans('', '', string.punctuation)

def clean_line(line):
    """changes case and removes punctuation"""
    # make all one case
    cleaned_line = line.lower()

    # remove punctuation
    cleaned_line = cleaned_line.translate(punct)
    return cleaned_line

def get_words(line):
    """splits line into words, and rejoins with newlines"""
    words = line.split()
    return "\n".join(words) + "\n"

def count_words(words):
    """takes list of cleaned words, returns count dictionary"""
    word_count = {}
    for word in words:
        count = word_count.setdefault(word, 0)
```

```
word_count[word] += 1
return word_count

def word_stats(word_count):
    """Takes word count dictionary and returns top
       and bottom five entries"""
    word_list = list(word_count.items())
    word_list.sort(key=lambda x: x[1])
    least_common = word_list[:5]
    most_common = word_list[-6:-1]
    return most_common, least_common
    ...
)

import text_processing_author

with open("moby_01.txt") as infile, open("moby_01_clean.txt",
                                         "w") as outfile:
    for line in infile:
        cleaned_line = text_processing_author.clean_line(line)

        cleaned_words = text_processing_author.get_words(cleaned_line)

        # write all words for line
        outfile.write(cleaned_words)

moby_words = []
with open('moby_01_clean.txt') as infile:
    for word in infile:
        if word.strip():
            moby_words.append(word.strip())

word_count = text_processing_author.count_words(moby_words)

most, least = text_processing_author.word_stats(word_count)
print("Most common words:")
for word in most:
    print(word)
print("\nLeast common words:")
for word in least:
    print(word)
```

THE AI SOLUTIONS

For Colaboratory, the process is straightforward. Since the code we developed in the lab for the previous chapter is our starting point, we need to add that code (the code that created functions for the text cleaning/processing functionality) to a cell in our current notebook.

Once we have the code in a cell, we can tell the bot to reference that cell to create our module and client code, which we do using this prompt:

Package the functions in the cell above as a standalone module so that the module can be used by other code. Then write the code to use the module to achieve the same result as the cell above.

The results from Colaboratory look promising, and we can see that the bot's suggestion is even clever enough to write the module to a file, enclosing the actual module code in a long triple-quoted string, just as we did in the human version. As we'll see in a moment, however, the bot wasn't *quite* clever enough. In any case, Colaboratory suggests

```
# Generated by colaboratory
# prompt: Package the functions in the cell above as a standalone module
#           so that the module can be used by other code. Then write the
#           code to use the module to achieve the same result as the cell
#           above.

# Save the functions as a module named "text_processing.py"
with open("text_processing.py", "w") as f:
    f.write("""
def clean_line(line):
    # Convert the line to lowercase
    line = line.lower()

    # Remove punctuation from the line
    line = line.translate(str.maketrans('', '', string.punctuation))

    return line

def write_words_to_file(words, output_file):
    # Write each word to the output file
    for word in words:
        output_file.write(word + '\n')

def count_word_occurrences(words):
    # Count the occurrences of each word using a dictionary
    word_counts = {}
    for word in words:
        if word in word_counts:
            word_counts[word] += 1
        else:
            word_counts[word] = 1

    return word_counts

def print_common_words(word_counts, n):
    # Sort the word counts in descending order
    sorted_word_counts = sorted(word_counts.items(),
                                key=lambda x: x[1], reverse=True)

    # Print the n most common words and their occurrences
    print(f"{n} most common words:")
    for word, count in sorted_word_counts[:n]:
        print(f"{word}: {count}")
```

```
def print_least_common_words(word_counts, n):
    # Sort the word counts in descending order
    sorted_word_counts = sorted(word_counts.items(),
                                 key=lambda x: x[1], reverse=True)

    # Print the n least common words and their occurrences
    print(f"{n} least common words:")
    for word, count in sorted_word_counts[-n:]:
        print(f"{word}: {count}")

def process_file(input_file_path, output_file_path):
    # Open the input file for reading
    with open(input_file_path, 'r') as input_file:
        # Open the output file for writing
        with open(output_file_path, 'w') as output_file:
            # Iterate over each line in the input file
            for line in input_file:
                # Clean the line
                cleaned_line = clean_line(line)

                # Split the line into words
                words = cleaned_line.split()

                # Write the words to the output file
                write_words_to_file(words, output_file)
    """

# Use the module
import text_processing

input_file_path = 'moby_01.txt'
output_file_path = 'moby_01_clean.txt'

# Process the file
text_processing.process_file(input_file_path, output_file_path)

# Open the file
with open(output_file_path, 'r') as file:
    # Read the file content
    content = file.read()

# Split the content into words
words = content.split()

# Count word occurrences
word_counts = text_processing.count_word_occurrences(words)

# Print the five most common words and their occurrences
text_processing.print_common_words(word_counts, 5)

# Print the five least common words and their occurrences
text_processing.print_least_common_words(word_counts, 5)
```

The problem comes with the way that Python writes the big string containing the module code to a file. When we try to execute this cell, we get a puzzling error:

```
File "/usr/local/lib/python3.10/dist-packages/IPython/core/  
    interactiveshell.py", line 3553, in run_code  
        exec(code_obj, self.user_global_ns, self.user_ns)  
  
File "<ipython-input-1-b6ebc2b78dfe>", line 70, in <cell line: 70>  
    import text_processing  
  
File "/content/text_processing.py", line 14  
    output_file.write(word + '  
                        ^  
  
SyntaxError: unterminated string literal (detected at line 14)
```

If we download the generated `text_processing.py` module, we see that at line 14 we have

```
    output_file.write(word + '\n')
```

instead of the

```
output_file.write(word + '\n')
```

that the AI bot intended. In other words, in writing the module file to disk, the '`\n`' that we want the module to add to lines of text it processes is instead written as a line break *in the code*, which causes the error. The fix for this is easy, but only if you know it—to keep that '`\n`' from being translated into an actual newline too early, we need to tell Python to treat that entire string as “raw” text that shouldn’t be interpreted normally. We can do that by adding the “raw” operator `r` immediately before the triple quotes, as in the previous human solution:

```
with open("text_processing.py", "w") as f:  
    f.write(r"""
```

When we do that and retry the code, we get yet another exception:

```
8
9     return line

NameError: name 'string' is not defined
```

If we look at the module code, we can see that it does not import the string library needed to provide the punctuation characters in `string.punctuation`.

To fix that, we can add `import string` at the top of our module code:

```
with open("text_processing.py", "w") as f:
    f.write(r"""
import string
def clean_line(line):
```

If we do that and execute the cell to regenerate our module, the error seems to persist. That's because the original flawed module is still loaded in our current Python session, as mentioned back in section 10.2's sidebar on reloading a module. Again, we can either restart the session (under the Runtime menu in Colaboratory) or we can import the `reload` function from `importlib`:

```
import importlib
importlib.reload(text_processing)
```

Once that fix is made, the newly created module works just fine.

For Copilot, we can simply open the file we created in chapter 9 in VS Code, select the functions, and give the chatbot the following prompt:

```
Package the selected functions and write them to a standalone module file so that the module can be used by other code. Then write the code to use the module to achieve the same result as the cell above.
```

For this, it's important to use a full Copilot chat session (accessed via the speech balloon icon at the left). Copilot then first outlines the steps it intends to follow:

Here's the plan:

1. Create a new Python file named `text_processing.py`.
2. Copy all the functions from the active selection into this new file.
3. At the end of the file, add a conditional if `__name__ == "__main__"`: to allow or prevent parts of code from being run when the modules are imported.
4. Inside this conditional, write the code to use the functions to achieve the same result as the original code. This will involve calling `process_file` with an input file path and an output file path.
5. In the user's current Python file, import the new module and use its functions.

Copilot then generates both a module file and a main file that uses the module, but both of the files must be manually saved under the correct names, which are specified in comment lines at the top of each file:

```
# FILENAME: word_processing.py

import string

def clean_line(line):
    line = line.lower()
    line = line.translate(str.maketrans('', '', string.punctuation))
    return line

def write_words_to_file(words, output_file):
    for word in words:
        output_file.write(word + '\n')

def count_word_occurrences(words):
    word_counts = {}
    for word in words:
        if word in word_counts:
            word_counts[word] += 1
        else:
            word_counts[word] = 1
    return word_counts

def print_common_words(word_counts, n):
    sorted_word_counts = sorted(word_counts.items(),
                                key=lambda x: x[1], reverse=True)
    print(f"{n} most common words:")
    for word, count in sorted_word_counts[:n]:
        print(f"{word}: {count}")

def print_least_common_words(word_counts, n):
    sorted_word_counts = sorted(word_counts.items(),
                                key=lambda x: x[1], reverse=True)
    print(f"{n} least common words:")
    for word, count in sorted_word_counts[-n:]:
        print(f"{word}: {count}")

def process_file(input_file_path, output_file_path):
    with open(input_file_path, 'r') as input_file:
        with open(output_file_path, 'w') as output_file:
            for line in input_file:
                cleaned_line = clean_line(line)
                words = cleaned_line.split()
                write_words_to_file(words, output_file)
```

Copilot handles the task of creating the module file nicely, producing a working file with the correct import of the `string` library. The only complaint that I have about this module is that it removed the comments that documented each function. I noted in the last chapter that using comments to document functions wasn't ideal Python style, but in this case, removing even that documentation is a negative:

```
# FILENAME: main.py

import word_processing
```

```
input_file_path = "path_to_input_file"
output_file_path = "path_to_output_file"

# Process the file
word_processing.process_file(input_file_path, output_file_path)

# Count word occurrences
with open(output_file_path, 'r') as output_file:
    words = output_file.read().split()
    word_counts = word_processing.count_word_occurrences(words)

# Print common and least common words
word_processing.print_common_words(word_counts, 10)
word_processing.print_least_common_words(word_counts, 10)

10 most common words:
the: 14
and: 9
i: 9
of: 8
is: 7
a: 6
it: 6
me: 5
to: 5
in: 4
10 least common words:
hours: 1
previous: 1
were: 1
out: 1
sight: 1
land: 1
look: 1
at: 1
crowds: 1
watergazers: 1
```

The file main.py only needed to load and use the module, which it did correctly. There was only one aspect that needed a change—it used `path_to_input_file` and `path_to_output_file` as the filenames of the data files. The bot did display a reminder under the generated code to “Please replace “`path_to_your_input_file`” and “`path_to_your_output_file`” with the actual paths to your input and output files,” and once the filenames were adjusted to `moby_01.txt` and `moby_01_clean.txt`, the code ran correctly.

This exercise was a step up in terms of abstraction—we have now moved from creating code to organizing that code into functions to organizing those functions into a module and calling it correctly. As we’ve seen at each step along the way, versions generated by AI are both impressive in their ability to generate a possible solution and susceptible to subtle bugs and simple errors.

Summary

- Python modules allow you to put related code and objects into a file.
- Using modules also helps prevent conflicting variable names, because imported objects are normally named in association with their module.
- `import` can be used to import an entire module or to access specific elements in a module.
- Elements whose names begin with an underscore are not imported by `from module import *`.
- A namespace is the dictionary that connects objects and their names for a particular scope.
- Something in global scope is accessible everywhere in the module.
- Something in local scope is only accessible in the immediately enclosing namespace.
- Objects in the built-in namespace are always accessible.

11

Python programs

This chapter covers

- Creating a very basic program
- Making a program directly executable on Linux/
UNIX
- Writing programs on macOS
- Selecting execution options in Windows
- Combining programs and modules
- Distributing Python applications

Up until now, you've been using the Python interpreter mainly in interactive mode in Colaboratory. For production use, you may need to create Python programs or scripts. Several of the sections in this chapter focus on command-line programs. If you come from a Linux/UNIX background, you may be familiar with scripts that can be started from a command line and given arguments and options that can be used to pass in information and possibly redirect their input and output. If you're from a Windows or Mac background, these things may be new to you, and you may be more inclined to question their value. While you certainly can use Python from

the command line in any of those environments, it becomes a bit trickier if you are using an Android or iOS device. Since we have been using Colaboratory as our reference environment, we'll discuss how to run command-line scripts from within a notebook.

It's true that command-line scripts are sometimes less convenient to use in a GUI environment, but the Mac has the option of a UNIX command-line shell, and Windows also offers enhanced command-line options. It will be well worth your time to read the bulk of this chapter at some point. You may find occasions when these techniques are useful, or you may run across code you need to understand that uses some of them. In particular, command-line techniques are very useful when you need to process large numbers of files.

11.1 Creating a very basic program

Any group of Python statements placed sequentially in a file can be used as a program, or *script*. But it's more standard and useful to introduce additional structure. In its most basic form, this task is a simple matter of creating a controlling function in a file and calling that function.

Listing 11.1 File script1.py

```
def main():
    print("this is our first test script file")
main()           ←→ Controlling function main
                ←→ Calls main
```

In this script, `main` is the controlling—and only—function. First, it's defined, and then it's called. Although it doesn't make much difference in a small program, this structure can give you more options and control when you create larger applications, so it's a good idea to make using it a habit from the beginning.

11.1.1 Starting a script from a command line

If we enter the contents of `script1.py` into a cell in Colaboratory, it works just fine. But we can also use a special syntax in a code cell to run commands in Colaboratory's host environment, which is a version of Ubuntu Linux. To run a command from the "command line" in Colaboratory, we can use the same commands as would be used in Linux but with the `!` character as a prefix. The one catch is that you need to have the file saved in that Colaboratory session. The code notebooks for this book include cells that have code that will either fetch the script files from the GitHub repository or write the code to a file, and to be sure that the same code behaves as expected, you need to be sure to either execute those cells or upload the files manually.

In Colaboratory, using the `!` prefix, you would use the following to start the script:

```
! python script1.py
this is our first test script file
```

If you're using Linux/UNIX, make sure that Python is on your path and you're in the same directory as your script. Then you would use the following:

```
python script1.py
```

If you're using a Mac running macOS, the procedure is the same as for other UNIX systems. You need to open a terminal program, which is in the Utilities folder of the Applications folder. You have several other options for running scripts on macOS, which I will discuss shortly.

If you're using Windows, open the Terminal (Windows 11—you can find it by right-clicking the Start Menu button) or Command Prompt (this can be found in different menu locations depending on the version of Windows; in Windows 10, it's in the Windows System menu) or PowerShell. Either of these opens in your home folder, and if necessary, you can use the `cd` command to change to a subdirectory. Running `script1.py` if it was saved on your desktop would look like the following:

```
C:\Users\naomi> cd Desktop  
C:\Users\naomi\Desktop> python script1.py  
this is our first test script file  
C:\Users\naomi\Desktop>
```

The diagram shows a command-line session on Windows. The user starts at the C:\Users\naomi prompt, changes to the Desktop folder, and then runs the python script1.py command. The output of the script, "this is our first test script file", is shown below the command line. Annotations with arrows point from the text to specific parts of the command line: "Changes to Desktop folder" points to the cd command, "Runs script1.py" points to the python command, and "Output of script1.py" points to the output text.

We'll look at other options for calling scripts later in this chapter, but we'll stick with using the `!` prefix in Colaboratory for our examples.

11.1.2 Command-line arguments

With command-line scripts, it's often useful to get parameters or arguments from the command line that runs the script. A simple mechanism is available for passing in command-line arguments. All of the arguments on the command line are in a list that can be accessed via `sys.argv`. Note that to access this list, you first need to import the `sys` module.

Listing 11.2 File script2.py

```
import sys  
def main():  
    print("this is our second test script file")  
    print(sys.argv)  
main()
```

The diagram shows the code for script2.py. It imports the sys module and defines a main function. The main function prints a message and then prints the sys.argv list. Annotations with arrows point from the text to specific parts of the code: "Imports sys module" points to the import statement, and "Accesses list in sys.argv" points to the print(sys.argv) statement.

If you call this with the line (leave out the `!` if you are not in a notebook)

```
! python script2.py arg1 arg2 3
```

you get

```
this is our second test script file
['script2.py', 'arg1', 'arg2', '3']
```

You can see that the command-line arguments have been stored in `sys.argv` as a list of strings. You can also see that the first item in that list is the command itself. You should also note that any numbers entered on the command line will come in as string representations of those numbers and may need to be converted to the right type before being used.

11.1.3 Executing code only as main script

The most commonly used Python script structure has one more element, which is an `if` statement that surrounds the main code that we want executed when the file is run as the `main` script. This is useful because it protects code from being executed when a file is imported as a module, as we'll see later in this chapter. This safeguard is essential when we want a script to be usable both as a standalone script and as a module.

Adding this feature to a script is a matter of putting the following conditional test around the code that we want executed if the file is run as a script:

```
if __name__ == '__main__':
    main()
else:                                     ← The else section is optional.
    # module-specific initialization code if any
```

If a file with this structure is called as a script, the variable `__name__` is set to `__main__`, which means that the controlling function, `main`, will be called. If the script has been imported into a module by some other script, its name will be its filename, and the code won't be executed. On the other hand, if we include the totally optional and less often used `else` block, that will only be executed if the file has been imported as a module.

When creating a script, I try to use this structure right from the start. This practice allows me to import the file into a session and interactively test and debug my functions as I create them or run the module as a script. This doesn't add much functionality to a tiny script, but if the script grows (as they often do), it can give more flexibility in how it's used.

11.1.4 Redirecting the input and output of a script

Particularly on Unix/Linux systems, it is common to redirect the contents of a file or the output of another process into a script instead of using keyboard output or to redirect the output of your script to the input of yet another process. Redirecting input and output this way allows you to chain small, specialized programs together to achieve an infinite variety of more complex tasks.

To redirect a file into input instead of having the user enter input, you need to specify that you want to read from standard input, which in the Python world is accessed by using the `stdin` object in the `sys` library. You can also specify writing to the standard

output by using `sys.stdout`, although in fact the `print` function also writes to `stdout` by default. To redirect from a file to input, you would use `<` on the command line, and for sending output to a file you would use `>`.

It's easier to understand what's going on with an example like the short script in the following listing.

Listing 11.3 File replace.py

```
import sys
def main():
    contents = sys.stdin.read()
    sys.stdout.write(contents.replace(sys.argv[1], sys.argv[2]))
if __name__ == "__main__":
    main()
```

This script gets its input by reading standard input (`sys.stdin`) as a file. Unlike the `input` function, it will keep reading (or waiting to read) data until it encounters an end-of-file indicator. If this script is run without redirecting standard input, it will read from the terminal and will keep reading and waiting until the user enters a Ctrl-D character to signal the end of the “file.” Once it’s done reading, it writes to its standard output whatever it has read, with all occurrences of its first argument replaced with its second argument.

Called as follows, with redirection of both input and output, the script places in `outfile` a copy of `infile`, with all occurrences of `zero` replaced by `0`:

```
python replace.py zero 0 < infile > outfile
```

Note that this sort of redirection usually only makes sense from a command prompt.

In general, the line

```
python script.py arg1 arg2 arg3 arg4 < infile > outfile
```

has the effect of having any `input` or `sys.stdin` operations directed out of `infile` and any `print` or `sys.stdout` operations directed into `outfile`. The effect is as though you set `sys.stdin` to `infile` with '`r`' (read) mode and `sys.stdout` to `outfile` with '`w`' (write). It's also possible to use `>>` to indicate appending to a file rather than overwriting:

```
python replace.py a A < infile >> outfile
```

This line causes the output to be appended to `outfile` rather than to overwrite it, as happened in the previous example.

You can also use the `|` to *pipe* in the output of one command as the input of another command:

```
python replace.py 0 zero < infile | python replace.py 1 one > outfile
```

This code first runs `replace.py` with the parameters `0 zero` reading the contents of `infile` and replacing all `0`s with `zero`. The output of that process is directed to the input of a second run of `replace.py`, which replaces any `1`s in that stream with `one` and writes the results to `outfile`, which will finally have the contents of `infile`, with all occurrences of `0` changed to `zero` and all occurrences of `1` changed to `one`.

This technique may seem strange if you are used to thinking in terms of larger programs, particularly on Windows or iOS, but combining small, specialized programs in this way is very powerful and at the heart of the Unix coding tradition.

11.1.5 The argparse module

You can configure a script to accept command-line options as well as arguments. The `argparse` module provides support for parsing different types of arguments and can even generate usage messages.

To use the `argparse` module, you create an instance of `ArgumentParser`, populate it with arguments, and then read both the optional and positional arguments. The following listing illustrates the module's use.

Listing 11.4 File opts.py

```
from argparse import ArgumentParser

def main():
    parser = ArgumentParser()
    parser.add_argument("indent", type=int, help="indent for report")
    parser.add_argument("input_file",
                       help="read data from this file")
    parser.add_argument("-f", "--file",
                       dest="filename",
                       help="write report to FILE", metavar="FILE")
    parser.add_argument("-x", "--xray",
                       help="specify xray strength factor")
    parser.add_argument("-q", "--quiet",
                       action="store_false", dest="verbose",
                       default=True,
                       help="don't print status messages to stdout")

    args = parser.parse_args()

    print("arguments:", args)
if __name__ == "__main__":
    main()
```

The code is annotated with three callout boxes:

- A box labeled "Adds indent and input file arguments" points to the first two `add_argument` calls.
- A box labeled "Adds optional -f or --file followed by filename" points to the third `add_argument` call.
- A box labeled "Adds -q or --quiet argument; defaults to True" points to the fourth `add_argument` call.

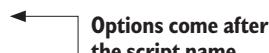
This code creates an instance of `ArgumentParser` and then adds two positional arguments, `indent` and `input_file`, which are the arguments entered after all of the optional arguments have been parsed. *Positional arguments* are those without a prefix character (usually `("-")`) and are required, and in this case, the `indent` argument must also be parsable as an `int`.

The next line adds an optional filename argument with either "-f" or "--file". The final option added, the "quiet" option, also adds the ability to turn off the verbose option, which is True by default (`action="store_false"`). The fact that these options begin with the prefix character "-" tells the parser that they're optional.

The final argument, "-q", also has a default value (True, in this case) that will be set if the option isn't specified. The `action="store_false"` parameter specifies that if the argument is specified, a value of `False` will be stored in the destination.

The `argparse` module returns a namespace object containing the arguments as attributes. You can get the values of the arguments by using dot notation. If there's no argument for an option, its value is `None`. Thus, if you call the previous script with the line

```
! python opts.py -x100 -q -f outfile 2 arg2
```



Options come after the script name.

the following output results:

```
arguments: Namespace(indent=2, input_file='arg2', filename='outfile',
                     xray='100', verbose=False)
```

If an invalid argument is found, or if a required argument isn't given, `parse_args` raises an error:

```
! python opts.py -x100 -r
```

This line results in the following response:

```
usage: opts.py [-h] [-f FILE] [-x XRAY] [-q] indent input_file
opts.py: error: the following arguments are required: indent, input_file
```

11.1.6 Using the `fileinput` module

The `fileinput` module is sometimes useful for scripts. It provides support for processing lines of input from one or more files. It automatically reads the command-line arguments (out of `sys.argv`) and takes them as its list of input files. Then it allows you to sequentially iterate through these lines. The simple example script in the following listing (which strips out any lines starting with `##`) illustrates the module's basic use.

Listing 11.5 File script4.py

```
import fileinput
def main():
    for line in fileinput.input():
        if not line.startswith('##'):
            print(line, end="")
if __name__ == "__main__":
    main()
```

Now assume that you have the data files shown in the next two listings.

Listing 11.6 File sole1.tst

```
## sole1.tst: test data for the sole function
0 0 0
0 100 0
##
##
```

Listing 11.7 File sole2.tst

```
## sole2.tst: more test data for the sole function
12 15 0
##
100 100 0
```

Also assume that you make this call:

```
! python script4.py sole1.tst sole2.tst
```

You obtain the following result with the comment lines stripped out and the data from the two files combined:

```
0 0 0
0 100 0
12 15 0
100 100 0
```

If no command-line arguments are present, the standard input is all that is read. If one of the arguments is a hyphen (-), the standard input is read at that point.

The `fileinput` module provides several other functions. These functions allow you at any point to determine the total number of lines that have been read (`lineno`), the number of lines that have been read out of the current file (`filelineno`), the name of the current file (`filename`), whether this is the first line of a file (`isfirstline`), and/or whether standard input is currently being read (`isstdin`). You can at any point skip to the next file (`nextfile`) or close the whole stream (`close`). The short script in the following listing (which combines the lines in its input files and adds file-start delimiters) illustrates how you can use these functions.

Listing 11.8 File script5.py

```
import fileinput
def main():
    for line in fileinput.input():
        if fileinput.isfirstline():
            print("<start of file {0}>".format(fileinput.filename()))
            print(line, end="")
    if __name__ == "__main__":
        main()
```

Using the call

```
python script5.py file1 file2
```

results in the following (where the dotted lines indicate the lines in the original files):

```
<start of file file1>
.....
.....
<start of file file2>
.....
.....
```

Finally, if you call `fileinput.input` with an argument of a single filename or a list of filenames, they're used as its input files rather than the arguments in `sys.argv`. `fileinput.input` also has an `inplace` option that leaves its output in the same file as its input while optionally leaving the original around as a backup file. See the documentation for a description of this option.

Quick check: Scripts and arguments

Match the following ways of interacting with the command line and the correct use case for each.

Multiple arguments and options	<code>sys.argv</code>
No arguments or just one argument	Use <code>fileinput</code> module
Processing multiple files	Redirect standard input and output
Using the script as a filter	Use <code>argparse</code> module

11.2 Running scripts in different operating systems

Running Python scripts from a command line or shell varies depending on one's operating system. In this section, we'll look briefly at running scripts in Windows, macOS, and Linux.

11.2.1 Making a script directly executable on UNIX

If you're on Colaboratory, as mentioned earlier, or a UNIX-type system, you can easily make a script directly executable, with two steps. First, add the following line at the top of the file:

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

Next, you need to use the `chmod` command to change its mode so that it can be executed directly, with the following command:

```
! chmod +x replace.py
```

Then if you place your script somewhere on your path (for example, in your bin directory), you can execute it regardless of the directory you’re in by typing its name and the desired arguments. In Colaboratory, we’re fine putting our files in the default /content directory. For example, we could execute `replace.py` with

```
! replace.py zero 0 < infile > outfile
```

The `<` and `>` characters cause the input and output to be redirected to and from the file, which works fine on Linux and Mac systems (and in Colaboratory) but may not work on Windows.

If you’re writing administrative scripts on UNIX, several library modules are available that you may find useful. These modules include `grp` for accessing the group database, `pwd` for accessing the password database, `resource` for accessing resource usage information, `syslog` for working with the syslog facility, and `stat` for working with information about a file or directory obtained from an `os.stat` call. You can find information on these modules in the documentation for the Python standard library.

11.2.2 Scripts on macOS

In many ways, Python scripts on macOS behave the same way as they do on Linux/UNIX. You can run Python scripts from a terminal window exactly the same way as on any UNIX box. But on the Mac, you can also run Python programs from the Finder, either by dragging the script file to the Python Launcher app or by configuring Python Launcher as the default application for opening your script (or, optionally, all files with a `.py` extension.)

You have several options for using Python on a Mac. The specifics of all the options are beyond the scope of this book, but you can get a full explanation by going to the www.python.org website and checking out the “Using Python on a Mac” section of the “Python Setup and Usage” section of the documentation for your version of Python (<https://docs.python.org/3.10/using/mac.html>).

If you’re interested in writing administrative scripts for macOS, you should look at packages that bridge the gap between Apple’s Open Scripting Architecture and Python.

11.2.3 Script execution options in Windows

If you’re on Windows, you have several options for starting a script that vary in their capability and ease of use. Unfortunately, exactly what those options might be and how they are configured can vary considerably across the various versions of Windows currently in use.

One option is installing the Linux subsystem for Windows, which gives you a Linux environment you can run as an application in Windows. This environment is based on Ubuntu Linux and gives you all of the options for Linux systems discussed earlier. For information on the other options for running Python on your system, consult the

online Python documentation for your version of Python and look for “Using Python on Windows.”

STARTING A SCRIPT FROM A COMMAND WINDOW OR POWERSHELL

To run a script from a command window or PowerShell window, open a command prompt or PowerShell window. When you’re at the command prompt and have navigated to the folder where your scripts are located, you can use Python to run your scripts in much the same way as on UNIX/Linux/macOS systems:

```
> python replace.py zero 0 < infile > outfile
```

Python doesn't run?

If Python doesn't run when you enter `python3` at the Windows command prompt, it probably means that the location of the Python executable isn't on your command path. You either need to add the Python executable to your system's PATH environment variable manually or rerun the installer to have it do the job. To get more help setting up Python on Windows, refer to the Python Setup and Usage section of the online Python documentation. There you'll find a section on using Python on Windows, with instructions for installing Python. In general, installing Python from the Microsoft Store should get you up and running without problems.

This is the most flexible of the ways to run a script on Windows because it allows you to use input and output redirection.

On Windows, you can edit the environment variables (see the previous section) to add `.py` as a magic extension, making your scripts automatically executable:

```
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.JS;.PY
```

Try this: Making a script executable

Experiment with executing scripts on your platform. Also, try to redirect input and output into and out of your scripts.

11.3 Programs and modules

For small scripts that contain only a few lines of code, a single function works well. But if the script grows beyond this size, separating your controlling function from the rest of the code is a good option to take. The rest of this section illustrates this technique and some of its benefits. I start with an example using a simple controlling function. The script in the following listing returns the English-language name for a given number between 0 and 99.

Listing 11.9 File script6.py

```

#! /usr/bin/env python3
import sys
# conversion mappings
_1to9dict = {'0': '', '1': 'one', '2': 'two', '3': 'three', '4': 'four',
              '5': 'five', '6': 'six', '7': 'seven', '8': 'eight',
              '9': 'nine'}
_10to19dict = {'0': 'ten', '1': 'eleven', '2': 'twelve',
                  '3': 'thirteen', '4': 'fourteen', '5': 'fifteen',
                  '6': 'sixteen', '7': 'seventeen', '8': 'eighteen',
                  '9': 'nineteen'}
_20to90dict = {'2': 'twenty', '3': 'thirty', '4': 'forty', '5': 'fifty',
                  '6': 'sixty', '7': 'seventy', '8': 'eighty', '9': 'ninety'}
def num2words(num_string):
    if num_string == '0':
        return 'zero'
    if len(num_string) > 2:
        return "Sorry can only handle 1 or 2 digit numbers"
    num_string = '0' + num_string
    tens, ones = num_string[-2], num_string[-1]
    if tens == '0':
        return _1to9dict[ones]
    elif tens == '1':
        return _10to19dict[ones]
    else:
        return _20to90dict[tens] + ' ' + _1to9dict[ones]
def main():
    print(num2words(sys.argv[1]))
if __name__ == "__main__":
    main()

```

Pads on left in case it's a single-digit number

Calls num2words with first argument

If you call it with

```
! python script6.py 59
```

you get the following result:

```
fifty nine
```

The controlling function here calls the function `num2words` with the appropriate argument and prints the result. It's standard to have the call at the bottom, but sometimes you'll see the controlling function's definition at the top of the file. I prefer this function at the bottom, just above the call, so that I don't have to scroll back up to find it after going to the bottom to find out its name. This practice also cleanly separates the scripting plumbing from the rest of the file, which is useful when combining scripts and modules.

People combine scripts with modules when they want to make functions they've created in a script available to other modules or scripts. Also, a module may be instrumented so it can run as a script either to provide a quick interface to it for users or to provide hooks for automated module testing.

As mentioned earlier, combining a script and a module is a simple matter of putting the following conditional test around the call to the controlling function:

```
if __name__ == '__main__':
    main()
else:
    # module-specific initialization code if any
```

Again, I recommend using this structure from the start when creating a script. This practice allows me to import it into a session and interactively test and debug my functions as I create them, and only the controlling function needs to be debugged externally. If the script grows and I find myself writing functions I might be able to use elsewhere, I can go ahead and use those functions by importing the script as a module, with the option of moving those functions into their own module later.

The script in listing 11.10 is an extension of the previous script but modified to be safe to be used as a module. The functionality has also been expanded to allow the entry of a number from 0 to 999,999,999,999,999 rather than just from 0 to 99. The controlling function (`main`) does the checking of the validity of its argument and also strips out any commas in it, allowing more user-readable input, like 1,234,567.

Listing 11.10 File n2w.py

```
#!/usr/bin/env python3
"""n2w: number to words conversion module: contains function
    num2words. Can also be run as a script
usage as a script: n2w num
    (Convert a number to its English word description)
    num: whole integer from 0 and 999,999,999,999,999 (commas are
        optional)
example: n2w 10,003,103
    for 10,003,103 say: ten million three thousand
    one hundred three
"""
import sys, string, argparse
_lto9dict = {'0': '', '1': 'one', '2': 'two', '3': 'three',
             '4': 'four',
             '5': 'five', '6': 'six', '7': 'seven', '8': 'eight',
             '9': 'nine'}
_10to19dict = {'0': 'ten', '1': 'eleven', '2': 'twelve',
               '3': 'thirteen', '4': 'fourteen', '5': 'fifteen',
               '6': 'sixteen', '7': 'seventeen', '8': 'eighteen',
               '9': 'nineteen'}
_20to90dict = {'2': 'twenty', '3': 'thirty', '4': 'forty', '5': 'fifty',
               '6': 'sixty', '7': 'seventy', '8': 'eighty', '9': 'ninety'}
_magnitude_list = [(0, ''), (3, ' thousand '), (6, ' million '),
                    (9, ' billion '), (12, ' trillion '),(15, '')]
def num2words(num_string):
    """num2words(num_string): convert number to English words"""
    if num_string == '0':
        return 'zero'
    num_string = num_string.replace(", ", "")
```

Usage message; includes example

Conversion mappings, for internal module use only

Handles special conditions (number is zero or too large)

Removes commas from number

```

num_length = len(num_string)
max_digits = _magnitude_list[-1][0]
if num_length > max_digits:

    return "Sorry, can't handle numbers with more than " \
           "{0} digits".format(max_digits)

num_string = '00' + num_string
word_string = ''
for mag, name in _magnitude_list:
    if mag >= num_length:
        return word_string
    else:
        hundreds, tens, ones = (num_string[-mag-3],
                                  num_string[-mag-2], num_string[-mag-1])
        if not (hundreds == tens == ones == '0'):
            word_string = _handle1to999(hundreds, tens, ones) + \
                           name + word_string
def _handle1to999(hundreds, tens, ones):
    if hundreds == '0':
        return _handle1to99(tens, ones)
    else:
        return _lto9dict[hundreds] + ' hundred ' + _handle1to99(tens, ones)
def _handle1to99(tens, ones):
    if tens == '0':
        return _lto9dict[ones]
    elif tens == '1':
        return _10to19dict[ones]
    else:
        return _20to90dict[tens] + ' ' + _lto9dict[ones]
def test():
    values = sys.stdin.read().split()
    for val in values:
        print("{0} = {1}".format(val, num2words(val)))
def main():
    parser = argparse.ArgumentParser(usage=__doc__)
    parser.add_argument("num", nargs='*')
    parser.add_argument("-t", "--test", dest="test",
                        action='store_true', default=False,
                        help="Test mode: reads from stdin")
    args = parser.parse_args()
    if args.test:
        test()
    else:
        try:
            result = num2words(args.num[0])
        except KeyError:
            parser.error('argument contains non-digits')
        else:
            print("For {0}, say: {1}".format(args.num[0], result))
if __name__ == '__main__':
    main()
else:
    print("n2w loaded as a module")

```

Pads number on left

Initiates string for number

Creates string containing number

For internal module use only

Function for module test mode

Gathers all values for that argument into a list

Runs in test mode if test variable is set

Catches KeyErrors due to argument containing nondigits

Calls main function if run as script

If this file is called as a script, the `__name__` will be `__main__`. If it's imported as a module, it will be named `n2w`. Remember that the variables and functions with names starting with an underscore (`"_"`) are intended to be only used internally by the module itself and should not be used by client code.

This `main` function illustrates the purpose of a controlling function for a command-line script, which in effect is to create a simple UI for the user. It may handle the following tasks:

- Ensure that there's the right number of command-line arguments and that they're of the right types. Inform the user, giving usage information if not. Here, the function ensures that there is a single argument, but it doesn't explicitly test to ensure that the argument contains only digits.
- Possibly handle a special mode. Here, a `--test` argument puts you in a test mode.
- Map the command-line arguments to those required by the functions and call them in the appropriate manner. Here, commas are stripped out, and the single function `num2words` is called.
- Possibly catch and print a more user-friendly message for exceptions that may be expected. Here, `KeyErrors` are caught, which occurs if the argument contains nondigits. (A better way to do this would be to explicitly check for nondigits in the argument using the regular expression module that will be introduced later. This would ensure that we don't hide `KeyErrors` that occur due to other reasons.)
- Map the output if necessary to a more user-friendly form, which is done here in the `print` statement. If this were a script to run on Windows, you'd probably want to let the user open it with the double-click method—that is, to use the `input` to query for the parameter, rather than have it as a command-line option and keep the screen up to display the output by ending the script with the line

```
input("Press the Enter key to exit")
```

But you may still want to leave the test mode as a command-line option.

The test mode in the following listing provides a regression test capability for the module and its `num2words` function. In this case, you use it by placing a set of numbers in a file.

Listing 11.11 File n2w.tst

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 98 99 100
101 102 900 901 999
999,999,999,999,999
1,000,000,000,000,000
```

Then type

```
! python n2w.py --test < n2w.tst

0 = zero
1 = one
2 = two
3 = three
4 = four
5 = five
6 = six
7 = seven
8 = eight
9 = nine
10 = ten
11 = eleven
12 = twelve
13 = thirteen
14 = fourteen
15 = fifteen
16 = sixteen
17 = seventeen
18 = eighteen
19 = nineteen
20 = twenty
21 = twenty one
98 = ninety eight
99 = ninety nine
100 = one hundred
101 = one hundred one
102 = one hundred two
900 = nine hundred
901 = nine hundred one
999 = nine hundred ninety nine
999,999,999,999,999 = nine hundred ninety nine trillion nine hundred ninety
nine billion nine hundred ninety nine million nine hundred ninety nine
thousand nine hundred ninety nine
1,000,000,000,000,000 = Sorry, can't handle numbers with more than 15 digits
```

You could also save the output to a file using redirection—for example, `> test_output.txt`. The output can then easily be checked for correctness. This example was run several times during its creation and can be rerun any time `num2words` or any of the functions it calls are modified. And yes, I’m aware that full exhaustive testing certainly didn’t occur and that well over 999 trillion valid inputs for this program haven’t been checked!

Often, the provision of a test mode for the module is the only reason for having a `main` function in that module. I know of at least one company in which part of the development policy is to always create at least one such test for every Python module developed. Python’s built-in data object types and methods usually make this process easy, and those who practice this technique seem to be unanimously convinced that it’s well worth the effort. See chapter 19, section 19.1.5, for more about testing your Python code.

In this example, it would be easy to create a separate file with only the portion of the `main` function that handles the argument and import `n2w` into this file. Then only the test mode would be left in the `main` function of `n2w.py`.

Quick check: Programs and modules

What problem is the use of `if __name__ == "__main__":` meant to prevent, and how does it do that? Can you think of any other way to prevent this problem?

11.4 Distributing Python applications

You can distribute your Python scripts and applications in several ways. You can share the source files, of course, probably bundled in a zip or tar file, which leaves a lot to be desired, particularly if the eventual user isn't comfortable in Python. Assuming that the applications were written portably *and* that the user had the correct version of Python installed, you could theoretically ship only the bytecode as .pyc files, but that is not recommended.

In fact, how best to distribute Python applications and manage their dependencies has been discussed a lot and various solutions have been (and continue to be) proposed. It's beyond our scope to even scratch the surface of the available options, so we'll stick with the basics.

11.4.1 Wheels packages

The current standard way of packaging and distributing Python modules and applications is to use packages called *wheels*. Wheels are designed to make installing Python code more reliable and to help manage dependencies. The details of how to create wheels are beyond the scope of this chapter, but full details about the requirements and the process for creating wheels are in the Python Packaging User Guide at <https://packaging.python.org>.

11.4.2 zipapp and pex

If you have an application that's in multiple modules, you can also distribute it as an executable zip file. This format relies on two facts about Python.

First, if a zip file contains a file named `__main__.py`, Python can use that file as the entry point to the archive and execute the `__main__.py` file directly. In addition, the zip file's contents are added to `sys.path`, so they're available to be imported and executed by `__main__.py`.

Second, zip files allow arbitrary contents to be added to the beginning of the archive. If you add a shebang line pointing to a Python interpreter, such as `#!/usr/bin/env python3`, and give the file the needed permissions, the file can become a self-contained executable.

In fact, it's not that difficult to manually create an executable `zipapp`. Create a zip file containing a `__main__.py`, add the shebang line to the beginning, and set the permissions.

Starting with Python 3.5, the `zipapp` module is included in the standard library; it can create `zipapps` either from the command line or via the library's API.

A more powerful tool, pex, isn't in the standard library but is available from the package index via pip. Pex does the same basic job but offers many more features and options, and it's available for Python 2.7, if needed. Either way, zip file apps are convenient ways to package and distribute multifile Python apps ready to run.

11.4.3 **py2exe and py2app**

Although it's not the purpose of this book to dwell on platform-specific tools, it's worth mentioning that `py2exe` creates standalone Windows programs and that `py2app` does the same on the macOS platform. By *standalone*, I mean that they're single executables that can run on machines that don't have Python installed. In many ways, standalone executables aren't ideal, because they tend to be larger and less flexible than native Python applications. But in some situations, they're the best—and sometimes the only—solution.

11.4.4 **Creating executable programs with freeze**

It's also possible to create an executable Python program that runs on machines that don't have Python installed by using the `freeze` tool. You'll find the instructions for this in the Readme file inside the `freeze` directory in the Tools subdirectory of the Python source directory. If you're planning to use `freeze`, you'll probably need to download the Python source distribution.

In the process of “freezing” a Python program, you create C files, which are then compiled and linked using a C compiler, which you need to have installed on your system. The frozen application will run only on the platform for which the C compiler you use provides its executables.

Several other tools try in one way or another to convert and package a Python interpreter/environment with an application in a standalone application. In general, however, this path is still difficult and complex, and you probably want to avoid it unless you have a strong need and the time and resources to make the process work.

11.5 **Creating a program**

In chapter 8, you created a version of the UNIX `wc` utility to count the lines, words, and characters in a file. Now that you have more tools at your disposal, refactor that program to make it work more like the original. In particular, the program should have options to show only lines (`-l`), only words (`-w`), and only characters (`-c`). If none of those options are given, all three stats are displayed. But if any of these options are present, only the specified stats are shown.

For an extra challenge, add the `-L` option to show the length of the longest line. You can test your version against the system's `wc` command on Colaboratory or a Linux/UNIX system. For help with `wc` and its options, use `! wc --help` in Colaboratory,

11.5.1 **Solving the problem with AI-generated code**

We are refactoring our previous code in this lab, so we need to be sure that the chatbot session has access to that code. For Colaboratory, we can copy the code from chapter 8's

lab into a cell in our current notebook. The Colaboratory and Copilot versions then were very similar, but there was a minor bug in the Colaboratory code, so we'll use the Copilot version as our starting point for both in this lab.

In addition to providing the code, we'll also need to make sure that the chatbot is told to mimic the Unix `wc` utility with the options `-l`, `-w`, `-c`, and `-L`. We can assume that the chatbot has access to an accurate description of the options for `wc`.

The prompt used for Colaboratory is

```
Refactor the code in the previous cell into a commandline program that works like the Unix wc utility, and supports the -l, -w, and -c options, and the -L option. Output should be similar to output of wc.
```

The Copilot prompt is put forth after selecting the same code, and then once the code is generated, it is saved to a new file:

```
efactor the selected code into a commandline program that works like the Unix wc utility, and supports the -l, -w, and -c options, and the -L option. Output should be similar to output of wc.
```

As we want the bot to create a new file, this prompt has to be entered into the full chat interface.

11.5.2 Solutions and discussion

The actual code for counting lines, words, and characters is the same as what we did before in chapter 8, but handling the options can get a bit tricky. In particular, for `wc`, if none of `-l`, `-w`, or `-c` are present, that means that all of the options should be selected, while `-L` must be explicitly selected whether the other options are there or not. Making this work will take some experimentation.

The other bit of new code that must be written is a way to find the length of the longest line. It's simplest to do this by checking and updating a variable as each line is read.

THE HUMAN SOLUTION

The biggest chunk of code in the human solution is spent in setting up the `argparse` parser object and then processing the options, while the actual processing of the file is fairly brief. In the course of making the code behave like `wc`, there are some Python idioms used in the following code that are worth noting:

```
#!/usr/bin/env python3
# File: word_count_program.py
"""
    Reads a file and returns the number of lines, words,
    and characters - similar to the UNIX wc utility
"""

import sys
import argparse

def main():
    # initialize counts
```

```

line_count = 0
word_count = 0
char_count = 0
longest_line = 0

parser = argparse.ArgumentParser(usage=__doc__)
parser.add_argument("-c", "--characters",
                    action="store_true", dest="chars", default=False,
                    help="display number of characters")
parser.add_argument("-w", "--words",
                    action="store_true", dest="words", default=False,
                    help="display number of words")
parser.add_argument("-l", "--lines",
                    action="store_true", dest="lines", default=False,
                    help="display number of lines")
parser.add_argument("-L", "--longest",
                    action="store_true", dest="longest", default=False,
                    help="display longest line length")
parser.add_argument("filename", help="read data from this file")
args = parser.parse_args()

filename = args.filename # open the file
with open(filename) as infile:
    for line in infile:
        line_count += 1
        char_count += len(line)
        words = line.split()
        word_count += len(words)
        if len(line) > longest_line:           ← longest_line updated if
                                                current line is longer
            longest_line = len(line)

default_args = any([getattr(args, _) for _ in ('chars', 'words',
                                               'lines', 'longest')]) ← List comprehension to
                                                               get option values; any() sees if any are set.

if not default_args:
    args.chars = args.lines = args.words = True ← Setting values
                                                can be chained

if args.lines:
    print(f'{line_count:3}', end=" ")
if args.words:
    print(f'{word_count:4}', end=" ")
if args.chars:
    print(f'{char_count:4}', end=" ")
if args.longest:
    print(f'{longest_line}', end=" ")
print(f'{filename}')
if __name__ == '__main__':
    main()

```

The first thing to touch upon is the setup of the `ArgumentParser`, `parser`. Note that all of the option arguments default to `False`, meaning that if nothing is specified for an option, nothing will be stored in the parser. This is important as we try to decide what gets printed. If there is an option specified, its value is printed, so that if one, two,

three, or all of the options are specified on the command line, the corresponding values will be displayed.

The trick is in detecting the case where no options are given, meaning that the standard set of line count, word count, and character count should be displayed. This solution handles it in a Pythonic way, but it may seem a bit tricky to people (and AI bots) new to Python—we use a list comprehension that includes calling the `getattr` function on the parser with each of the possible options: `'chars'`, `'words'`, `'lines'`, and `'longest'`. If nothing has been set on the command line, this will be a list of `False` values, so calling the `any` function on that list will also be `False`. On the other hand, if even one of those values is `True`, then `any` will also return `True`. The same result could also be achieved with a long `if` statement check using `or` several times but would be more cumbersome to read.

The second part of this feature is what the code does if the value in `default_args` is `False`. Note that all three of the variables are set on the same line, chaining three `=` operators. We can do that (and in fact it's reasonable Python style) because using the `=` operator gives a return value: it returns the value the variable is being set to. So each `=` operation returns `True` to set the variable to the left of the expression until they all are set.

Finally, as in the solution in chapter 8, this solution iterates through the file it's processing line by line, updating the values for all the stats each time. This is an approach I take almost automatically, since it's rarely noticeably slower and it handles files of any size. I've had to go back and switch to this approach enough times in the past that it's now second nature to me.

THE AI SOLUTIONS

This Colaboratory solution was one of four very similar options offered. This one was chosen because its output was closest to that of `wc`. It is worth noting, however, that “closest to” doesn't mean “correct,” as we'll see:

```
# prompt: Refactor the code in the previous cell into a commandline program
#   that works like the Unix wc utility, and supports the -l, -w, and
#   -c options, and the -L option. Output should be similar to
#   output of wc.

#!/usr/bin/env python3

""" Reads a file and returns the number of lines, words,
    and characters - similar to the UNIX wc utility
"""

import argparse

def main():
    parser = argparse.ArgumentParser(usage=__doc__)
    parser.add_argument("filename", nargs='*')
    parser.add_argument("-l", "--lines", dest="lines",
                        action='store_true', default=False,
                        help="Show only line count")
```

```

parser.add_argument("-w", "--words", dest="words",
                    action='store_true', default=False,
                    help="Show only word count")
parser.add_argument("-c", "--chars", dest="chars",
                    action='store_true', default=False,
                    help="Show only character count")
parser.add_argument("-L", "--longest", dest="longest",
                    action='store_true', default=False,
                    help="Show length of longest line")
args = parser.parse_args()

for filename in args.filename:
    with open(filename) as infile:
        lines = infile.readlines()

    line_count = len(lines)
    word_count = sum(len(line.split()) for line in lines)
    char_count = sum(len(line) for line in lines)
    longest_line_length = max(len(line) for line in lines)

    if args.lines:
        print(f"{line_count} {filename}")
    elif args.words:
        print(f"{word_count} {filename}")
    elif args.chars:
        print(f"{char_count} {filename}")
    elif args.longest:
        print(f"{longest_line_length} {filename}")
    else:
        print(f"{line_count} {word_count} {char_count} {filename}")

if __name__ == '__main__':
    main()

```

This solution correctly creates an `ArgumentParser` and adds the needed options. It also maintains the choice in the original code (as discussed in the lab for chapter 8) to read the entire file into memory and then split the lines on the `\n` newline character. It can then use the length of the resulting list as the line count and use generator expressions to iterate through the list of lines to find the number of words and the longest line.

The drawback, as mentioned in chapter 8, is that the entire file must be read into memory, and during the life of the generator expression, a separate list of all the words also must exist in memory. This could be a problem if processing a very large file. The `wc` utility, on the other hand, doesn't have this limitation, nor does my solution.

The other flaw with this solution is that its output doesn't quite work the way the `wc` utility's does. The solution does add the filename after the numeric output, which is correct, and it also can handle any single option being selected, as well as no options being selected. But look at the `if-elif-else` setup. What happens if, say, both the `-l` and `-w` options are selected? That's right: only the line count is displayed, while for the `wc` utility, it would mean that both the line and word count would be shown.

The Copilot solution initially was less like the `wc` utility than Colaboratory version, but after giving it an extra prompt it ended up being the most faithful to the original.

To create a standalone file, rather than just rewrite some code in an existing file, we needed to use full chat interface in VS Code, as we did in the last chapter.

Before it started generating code, Copilot created the following plan and then generated the following code:

- 1 Import the `argparse` module to handle command-line arguments.
- 2 Define a function `count_stats` that takes a file path as input and returns the number of lines, words, and characters.
- 3 Create an `argparse.ArgumentParser` instance to handle the command-line arguments.
- 4 Add arguments for `-l`, `-w`, `-c`, and `-L` options. These arguments should be optional and default to `False`.
- 5 Add an argument for the file path.
- 6 Parse the arguments using `argparse.ArgumentParser.parse_args()`.
- 7 Call `count_stats` with the file path from the parsed arguments.
- 8 Depending on the options provided, print the number of lines, words, and characters.

```
# wc.py

import argparse

def count_stats(file_path):
    with open(file_path) as infile:
        lines = infile.readlines()

    line_count = len(lines)
    word_count = sum(len(line.split()) for line in lines)
    char_count = sum(len(line) for line in lines)
    max_line_length = max(len(line) for line in lines)

    return line_count, word_count, char_count, max_line_length

def main():
    parser = argparse.ArgumentParser(description=
        'Count lines, words, and characters in a file.')
    parser.add_argument('file', help='File to count')
    parser.add_argument('-l', '--lines', action='store_true',
        help='Count lines')
    parser.add_argument('-w', '--words', action='store_true',
        help='Count words')
    parser.add_argument('-c', '--chars', action='store_true',
        help='Count characters')
    parser.add_argument('-L', '--max-line-length', action='store_true',
        help='Print length of the longest line')
    args = parser.parse_args()

    line_count, word_count, char_count, max_line_length = count_stats(args.
        file)
```

```

if args.lines:
    print(f'Lines: {line_count}')
if args.words:
    print(f'Words: {word_count}')
if args.chars:
    print(f'Characters: {char_count}')
if args.max_line_length:
    print(f'Max line length: {max_line_length}')

if __name__ == '__main__':
    main()

```

The plan seems fine, although creating a separate function to do the processing isn't really necessary. However, there are some problems with this solution. First, the values are output on separate lines, unlike `wc`, and with labels describing what the values are. Also, a problem is that if no option is specified on the command line, nothing is output. This behavior isn't wrong as far as processing the text goes, but it isn't like `wc`, which is what we specified in the prompt.

Since it's easy to continue interaction in the chat interface, I asked for a correction with the following prompt:

The options and their output should be the same as the `wc` utility

This was specified in the original prompt, but maybe “similar to” gives it some room for variation, so this time “the same” output is specified. In response, Copilot came up with the following comment and a new plan.

The `wc` utility in Unix outputs the line, word, and byte counts by default, and the output is in a single line. The `-l` option outputs only the line count, `-w` outputs only the word count, `-c` outputs the byte count, and `-L` outputs the length of the longest line. If no option is provided, all counts are output.

The plan was

- 1** Modify the `count_stats` function to return the byte count instead of the character count.
- 2** Modify the `main` function to print the counts in a single line in the order: line count, word count, byte count, filename.
- 3** If the `-l`, `-w`, `-c`, or `-L` options are provided, print only the corresponding count.
- 4** If no option is provided, print all counts.

The commentary and the plan are much more correct this time and cover the requested options accurately. Copilot is correct that technically the `-c` option indicates that bytes, not characters, should be counted (depending on the encoding, some characters might take more than 1 byte), but leaving that aside, the more significant change is in how it handles what to output. In this it gets much closer to the actual output of the `wc` utility, and as it happens, it comes up with something similar to the human-based solution. Most of the code is unchanged, but here is what it came up with for the actual printing of the results:

```
if args.lines:
    print(line_count, end=' ')
if args.words:
    print(word_count, end=' ')
if args.bytes:
    print(byte_count, end=' ')
if args.max_line_length:
    print(max_line_length, end=' ')
if not any([args.lines, args.words, args.bytes, args.max_line_length]):
    print(f'{line_count} {word_count} {byte_count} {args.file}')
else:
    print(args.file)
```

As we can see, this code checks all of the parser options and prints the default if no option is specified. It also handles any combination of options correctly and prints only the values, with no labels. It also prints the filename correctly for all variations, although that may be a bit confusing to pick out the first time someone reads the code—printing the filename for the specified options is in the `else` statement for no options specified.

The good news is that both AI chatbots did know how the `wc` utility works, so there was no need go into great detail in the prompts. On the other hand, it did require some attention to go detail and specific prompts to get exactly the desired result.

Summary

- Python scripts and modules in their most basic form are just sequences of Python statements placed in a file.
- Modules can be instrumented to run as scripts, and scripts can be set up so that they can be imported as modules.
- Scripts can be made executable on the UNIX, macOS, or Windows command lines. They can be set up to support command-line redirection of their input and output, and with the `argparse` module, it's easy to parse out complex combinations of command-line arguments.
- On macOS, you can use the Python Launcher to run Python programs, either individually or as the default application for opening Python files.
- On Windows, you can call scripts in several ways: by opening them with a double-click, using the Run window, or using a command-prompt window.
- Python scripts can be distributed as scripts, as bytecode, or in special packages called wheels.
- `py2exe`, `py2app`, and the `freeze` tool provide an executable Python program that runs on machines that don't contain a Python interpreter.

12

Using the filesystem

This chapter covers

- Managing paths and pathnames
- Getting information about files
- Performing filesystem operations
- Processing all files in a directory subtree

Working with files involves one of two things: basic I/O (described in chapter 13) and working with the filesystem (for example, naming, creating, moving, or referring to files), which is a bit tricky, because different operating systems have different filesystem conventions.

Since we are assuming Google Colaboratory, which is hosted on Linux, as our default environment, most of the examples in this chapter will be based on Linux. It's quite possible, however, that you will want to write scripts that access files that run on other platforms, so we will include mention of how the same operations work on those platforms, particularly Windows, as needed.

It would be easy enough to learn how to perform basic file I/O without learning all the features Python has provided to simplify cross-platform filesystem

interaction—but I wouldn’t recommend it. Instead, read at least the first part of this chapter, which gives you the tools you need to refer to files in a manner that doesn’t depend on your particular operating system. Then, when you use the basic I/O operations, you can open the relevant files in this manner.

12.1 **os and os.path vs. pathlib**

The traditional way that file paths and filesystem operations have been handled in Python is by using functions included in the `os` and `os.path` modules. These functions have worked well enough but often resulted in more verbose code than necessary. Since Python 3.5, a new library, `pathlib`, has been added; it offers a more object-oriented and unified way of doing the same operations. Because a lot of code out there still uses the older style, I’ve retained those examples and their explanations. On the other hand, `pathlib` has a lot going for it and is likely to become the new standard, so after each example of the old method, I include an example (and brief explanation, where necessary) of how the same thing would be done with `pathlib`.

12.2 **Paths and *pathnames***

All operating systems refer to files and directories with strings naming a given file or directory. Strings used in this manner are usually called *pathnames* (or sometimes just *paths*), which is the word I’ll use for them. The fact that pathnames are strings introduces possible complications to working with them. Python does a good job of providing functions that help avert these complications; but to use these Python functions effectively, you need to understand the underlying problems. This section discusses these details.

Pathname semantics across operating systems are very similar because the filesystem on almost all operating systems is modeled as a tree structure, with a disk being the root, and folders, subfolders, and so on being branches, subbranches, and so on. This means that most operating systems refer to a specific file in fundamentally the same manner: with a pathname that specifies the path to follow from the root of the filesystem tree (the disk) to the file in question. (This characterization of the root corresponding to a hard disk is an oversimplification, but it’s close enough to the truth to serve for this chapter.) This pathname consists of a series of folders to descend into to get to the desired file.

Different operating systems have different conventions regarding the precise syntax of pathnames. The character used to separate sequential file or directory names in a Linux/UNIX pathname is `/`, whereas the character historically used to separate file or directory names in a Windows pathname is `\`. In addition, the UNIX filesystem has a single root (which is referred to by having a `/` character as the first character in a pathname), whereas the Windows filesystem has a separate root for each drive, labeled `A:\`, `B:\`, `C:\`, and so forth (with `C:` usually being the main drive). Because of these differences, files have different pathname representations on different operating systems. A file called `C:\data\myfile` in MS Windows might be called `/data/myfile` on UNIX and on the macOS. Python provides functions and constants that allow you to perform common

pathname manipulations without worrying about such syntactic details. There are also differences in the way that the operating system treats upper-versus lowercase strings. In Linux and in some iOS file systems, case matters, so that file_a is not the same filename as File_A. Windows has traditionally ignored case differences. It's a good idea to keep this in mind and follow a convention of, for example, only using lowercase filenames to minimize confusion if your code needs to interact with files on different systems. With a little care, you can write your Python programs in such a manner that they'll run correctly no matter what the underlying filesystem happens to be.

12.2.1 Absolute and relative paths

Most operating systems allow two types of pathnames:

- *Absolute* pathnames specify the exact location of a file in a filesystem without any ambiguity; they do this by listing the entire path to that file, starting from the root of the filesystem.
- *Relative* pathnames specify the position of a file relative to some other point in the filesystem, and that other point isn't specified in the relative pathname itself; instead, the absolute starting point for relative pathnames is provided by the context in which they're used.

As examples, here are two Windows absolute pathnames:

```
C:\Program Files\Doom  
D:\backup\June
```

Here are two Linux absolute pathnames and a Mac absolute pathname:

```
/bin/Doom  
/floppy/backup/June  
/Applications/Utilities
```

Here are two Windows relative pathnames:

```
mydata\project1\readme.txt  
games\tetris
```

These are Linux/UNIX/Mac relative pathnames:

```
mydata/project1/readme.txt  
games/tetris  
Utilities/Java
```

Relative paths need context to anchor them. This context is typically provided in one of two ways.

The simpler way is to append the relative path to an existing absolute path, producing a new absolute path. You might have a relative Windows path, Start Menu\Programs\Startup, and an absolute path, C:\Users\Administrator. By appending the two, you have

a new absolute path: C:\Users\Administrator\Start Menu\Programs\Startup, which refers to a specific location in the filesystem. By appending the same relative path to a different absolute path (say, C:\Users\myuser), you produce a path that refers to the Startup folder in a different user's (myuser's) Profiles directory.

The second way in which relative paths may obtain a context is via an implicit reference to the *current working directory*, which is the particular directory where a Python program considers itself to be at any point during its execution. Python commands may implicitly make use of the current working directory when they're given a relative path as an argument. If you use the `os.listdir(path)` command with a relative path argument, for example, the anchor for that relative path is the current working directory, and the result of the command is a list of the filenames in the directory whose path is formed by appending the current working directory with the relative path argument.

12.2.2 The current working directory

Whenever you edit a document on a computer, you have a concept of where you are in that computer's file structure because you're in the same directory (folder) as the file you're working on. Similarly, whenever Python is running, it has a concept of where in the directory structure it is at any moment. This fact is important because the program may ask for a list of files stored in the current directory. The directory that a Python program is in is called the *current working directory* for that program. This directory may be different from the directory the program resides in.

To see this in action, start Python and use the `os.getcwd()` (get current working directory) command to find Python's initial current working directory:

```
import os  
os.getcwd()  
  
'/content'
```

Note that `os.getcwd` is used as a zero-argument function call, to emphasize the fact that the value it returns isn't a constant but will change as you put forth commands that alter the value of the current working directory. (That directory probably will be '/content' in Colaboratory, or the directory the Python program itself resides in, or the directory you were in when you started Python. On Windows machines, you'll see extra backslashes inserted into the path because Windows uses \ as its path separator, and in Python strings (as discussed in section 6.3.1), \ has a special meaning unless it is itself backslashed.)

Now type

```
os.listdir(os.curdir)  
  
['.config', 'sample_data']
```

The constant `os.curdir` returns a string reflecting whatever the current directory is. On both UNIX and Windows, the current directory is represented as a single dot, but

to keep your programs portable, you should always use `os.curdir` instead of typing just the dot. This string is a relative path, meaning that `os.listdir` appends it to the path for the current working directory, giving the same path. This command returns a list of all the files or folders inside the current working directory (on Colaboratory, it should be the list shown earlier). To look at the folder '`sample_data`', type

```
os.chdir('sample_data')    ← "Change directory" function  
os.getcwd()
```

As you can see, Python moves into the folder specified as the argument of the `os.chdir` function. Another call to `os.listdir(os.curdir)` would return a list of files in the folder '`sample_data`', because `os.curdir` would then be taken relative to the new current working directory. Many Python filesystem operations use the current working directory in this manner.

12.2.3 Accessing directories with `pathlib`

To get the current directory with `pathlib`, you could do the following:

```
import pathlib  
cur_path = pathlib.Path()  
cur_path.cwd()  
  
PosixPath('/content/sample_data')
```

There's no way for `pathlib` to change the current directory in the way that `os.chdir()` does (see the preceding section), since a path is by definition a specific directory location. However, you can work with a new folder by creating a new path object, as discussed in section 12.2.5.

12.2.4 Manipulating pathnames

Now that you have the background to understand file and directory pathnames, it's time to look at the facilities Python provides for manipulating these pathnames. These facilities consist of several functions and constants in the `os.path` submodule, which you can use to manipulate paths without explicitly using any operating-system-specific syntax. Paths are still represented as strings, but you need never think of them or manipulate them as such.

To start, construct a few pathnames on different operating systems using the `os.path.join` function. Note that importing `os` is sufficient to bring in the `os.path` submodule also; there's no need for an explicit `import os.path` statement.

In our notebook, we can do something like this:

```
import os  
print(os.path.join('bin', 'utils', 'disktools'))  
  
bin/utils/disktools
```

The `os.path.join` function interprets its arguments as a series of directory names or filenames, which are to be joined to form a single string understandable as a relative path by the underlying operating system. In a Windows system, that means path component names would be joined with backslashes. In other words, `os.path.join` lets you form file paths from a sequence of directory or filenames without any worry about the conventions of the underlying operating system. `os.path.join` is the fundamental way by which file paths may be built in a manner that doesn't constrain the operating systems on which your program will run.

The arguments to `os.path.join` need not be a single directory or filename; they may also be subpaths that are then joined to make a longer pathname. The following example illustrates this in the Windows environment and is also a case in which you'd find it necessary to use double backslashes in your strings. Note that you could enter the pathname with forward slashes (/) as well, because Python converts them before accessing the Windows operating system:

```
import os
print(os.path.join('mydir\\bin', 'utils\\disktools\\chkdisk'))  
  
mydir\bin\utils\disktools\chkdisk
```

If you always use `os.path.join` to build up your paths, of course, you'll rarely need to worry about this situation. To write this example in a portable manner, you could enter

```
path1 = os.path.join('mydir', 'bin')
path2 = os.path.join('utils', 'disktools', 'chkdisk')
print(os.path.join(path1, path2))  
  
mydir\bin\utils\disktools\chkdisk
```

← On a Windows system

This will work also work correctly on a Linux/Unix-based system:

```
mydir/bin/utils/disktools/chkdisk
```

← On a Linux system

The `os.path.join` command also has some understanding of absolute versus relative pathnames. In Linux/UNIX, an *absolute* path always begins with a / (because a single slash denotes the topmost directory of the entire system, which contains everything else, including the various other drives that might be available). A *relative* path in UNIX is any legal path that does *not* begin with a slash. Under any of the Windows operating systems, the situation is more complicated because the way in which Windows handles relative and absolute paths is messier. Rather than go into all of the details, I'll just say that the best way to handle this situation is to work with the following simplified rules for Windows paths:

- A pathname beginning with a drive letter followed by a colon and a backslash and then a path is an absolute path: C:\Program Files\DOOM. (Note that C: by itself, without a trailing backslash, can't reliably be used to refer to the top-level

directory on the C: drive. You must use C:\ to refer to the top-level directory on C:. This requirement is a result of DOS conventions, not Python design.)

- A pathname beginning with neither a drive letter nor a backslash is a relative path: mydirectory\letters\business.
- A pathname beginning with \\ followed by the name of a server is the path to a network resource.
- Anything else can be considered to be an invalid pathname.

Regardless of the operating system used, the `os.path.join` command doesn't perform sanity checks on the names it's constructing. It's possible to construct pathnames containing characters that, according to your OS, are forbidden in pathnames. If such checks are a requirement, probably the best solution is to write a small path-validity-checker function yourself.

The `os.path.split` command returns a two-element tuple splitting the basename of a path (the single file or directory name at the end of the path) from the rest of the path. You might use this example on a Windows system:

```
import os
print(os.path.split(os.path.join('some', 'directory', 'path')))

('some/directory', 'path')
```

The `os.path.basename` function returns only the basename of the path, and the `os.path.dirname` function returns the path up to but not including the last name, as in this example:

```
import os
os.path.basename(os.path.join('some', 'directory', 'path.jpg'))

'path.jpg'

os.path.dirname(os.path.join('some', 'directory', 'path.jpg'))

'some/directory'
```

To handle the dotted extension notation used by most filesystems to indicate file type (the Mac is a notable exception), Python provides `os.path.splitext`:

```
os.path.splitext(os.path.join('some', 'directory', 'path.jpg'))

('some/directory/path', '.jpg')
```

The last element of the returned tuple contains the dotted extension of the indicated file (if there was a dotted extension). The first element of the returned tuple contains everything from the original argument except the dotted extension.

You can also use more specialized functions to manipulate pathnames. `os.path.commonprefix(path1, path2, ...)` finds the common prefix (if any) for a set of paths.

This technique is useful if you want to find the lowest-level directory that contains every file in a set of files. `os.path.expanduser` expands username shortcuts in paths, such as for UNIX. Similarly, `os.path.expandvars` does the same for environment variables. Here's an example on a Windows system:

```
import os
os.path.expandvars('$HOME\\temp')

'C:\\Users\\administrator\\personal\\temp'
```

12.2.5 Manipulating pathnames with `pathlib`

Just as you did in the preceding section, let's start by constructing a few pathnames on different operating systems, using the `path` object's methods.

First, start Python under Windows:

```
from pathlib import Path
cur_path = Path()
print(cur_path.joinpath('bin', 'utils', 'disktools'))

bin/utils/disktools
```

The same result can be achieved by using the slash operator:

```
cur_path / 'bin' / 'utils' / 'disktools'

PosixPath('bin/utils/disktools')
```

Note that in the representation of the `path` object, forward slashes are always used, but Windows path objects have the forward slashes converted to backslashes as required by the OS. So if you try the same thing in UNIX

```
cur_path = Path()
print(cur_path.joinpath('bin', 'utils', 'disktools'))

bin/utils/disktools
```

the `parts` property returns a tuple of all the components of a path. For example:

```
a_path = Path('bin/utils/disktools')
print(a_path.parts)

('bin', 'utils', 'disktools')
```

The `name` property returns only the basename of the path, the `parent` property returns the path up to but not including the last name, and the `suffix` property handles the dotted extension notation used by most filesystems to indicate file type (but the Mac is a notable exception). Here's an example:

```
a_path = Path('some', 'directory', 'path.jpg')
a_path.name
'path.jpg'

print(a_path.parent)

some/directory

a_path.suffix

'.jpg'
```

Several other methods associated with `Path` objects allow flexible manipulation of both pathnames and files themselves, so you should review the documentation of the `pathlib` module. It's likely that the `pathlib` module will make your life easier and your file-handling code more concise.

12.2.6 Useful constants and functions

You can access several useful path-related constants and functions to make your Python code more system independent than it otherwise would be. The most basic of these constants are `os.curdir` and `os.pardir`, which respectively define the symbol used by the operating system for the directory and parent directory path indicators. In Windows as well as Linux/UNIX and macOS, these indicators are `.` and `..`, respectively, and they can be used as normal path elements. This example

```
os.path.isabs(os.path.join(os.pardir, path))
```

asks whether the parent of the parent of `path` is a directory. `os.curdir` is particularly useful for requesting commands on the current working directory. This example

```
os.listdir(os.curdir)
```

returns a list of filenames in the current working directory (because `os.curdir` is a relative path, and `os.listdir` always takes relative paths as being relative to the current working directory).

The `os.name` constant returns the name of the Python module imported to handle the operating system-specific details. Here's an example on Colaboratory:

```
import os
os.name
'posix'
```

Colaboratory, being hosted on Linux, returns '`posix`'. Note that on Windows, `os.name` returns '`nt`' even though the actual version of Windows could be Windows 10 or 11. Most versions of Windows are identified as '`nt`'.

On a Mac running macOS and on Linux/UNIX, the response is `posix`. You can use this response to perform special operations, depending on the platform you're working on:

```
import os
if os.name == 'posix':
    root_dir = "/"
elif os.name == 'nt':
    root_dir = "C:\\\\"
else:
    print("Don't understand this operating system! ")
```

The only other possible value for `os.name` as of this writing is '`java`', which is returned by Jython (Python running on a Java virtual machine).

You may also see programs use `sys.platform`, which gives more exact information. Even on Windows 11, `sys.platform` is set to `win32`—even though the machine is running the 64-bit version of the operating system. On Linux, you will see `linux`, whereas on macOS it will be `darwin`. There have been changes to the list of possible values as platforms come and go, but you can always find the complete list of possible values in the documentation for `sys.platform`.

All your environment variables and the values associated with them are available in a dictionary called `os.environ`. On most operating systems, this dictionary includes variables related to paths—typically, search paths for binaries and so forth.

At this point, you've received an introduction to the major aspects of working with pathnames in Python. While you now know enough to open files for reading or writing, the rest of the chapter will give you further information about pathnames, testing what they point to, useful constants, and so forth.

Quick check: Manipulating paths

How would you use the `os` module's functions to take a path to a file called `test.log` and create a new file path in the same directory for a file called `test.log.old`? How would you do the same thing using the `pathlib` module?

What path would you get if you created a `pathlib Path` object from `os.pardir`? Try it and find out.

12.3 Getting information about files

File paths are supposed to indicate actual files and directories on your hard drive. You're probably passing a path around, of course, because you want to know something about the file or directory it points to. Various Python functions are available for this purpose.

The most commonly used Python path-information functions are `os.path.exists`, `os.path.isfile`, and `os.path.isdir`, all of which take a single path as an argument.

`os.path.exists` returns `True` if its argument is a path corresponding to something that exists in the filesystem. `os.path.isfile` returns `True` if and only if the path it's given indicates a normal data file of some sort (executables fall under this heading), and it returns `False` otherwise, including the possibility that the path argument doesn't indicate anything in the filesystem. `os.path.isdir` returns `True` if and only if its path argument indicates a directory; it returns `False` otherwise. These examples are valid on my system. You may need to use different paths on yours to investigate the behavior of these functions:

```
import os
os.path.exists('/content/sample_data/')

True

os.path.exists('/content/sample_data/README.md')

True

os.path.exists('/content/sample_data/ljsljkflkjs')

False

os.path.isdir('/content/sample_data/')

True

os.path.isdir('/content/sample_data/README.md')

False

os.path.isfile('/content/sample_data/README.md')

True
```

Several similar functions provide more specialized queries. `os.path.islink` and `os.path.ismount` are useful in the context of Linux and other UNIX operating systems that provide file links and mount points; they return `True` if, respectively, a path indicates a file that's a link or a mount point. `os.path.islink` does *not* return `True` on Windows shortcuts files (files ending with `.lnk`), for the simple reason that such files aren't true links. However, `os.path.islink` returns `True` on Windows systems for true symbolic links created with the `mklink()` command. The OS doesn't assign them a special status, and programs can't transparently use them as though they were the actual file. `os.path.samefile(path1, path2)` returns `True` if and only if the two path arguments point to the same file. `os.path.isabs(path)` returns `True` if its argument is an absolute path; it returns `False` otherwise. `os.path.getsize(path)`, `os.path.getmtime(path)`, and `os.path.getatime(path)` return the size, last modify time, and last access time of a pathname, respectively.

If you want to use `Path` objects, there are methods that have names and functionality similar to the `os` functions mentioned earlier.

12.3.1 Getting information about files with scandir

In addition to the `os.path` functions listed, you can get more complete information about the files in a directory by using `os.scandir`, which returns an iterator of `os.DirEntry` objects. `os.DirEntry` objects expose the file attributes of a directory entry, so using `os.scandir` can be faster and more efficient than combining `os.listdir` (discussed in the next section) with the `os.path` operations. If, for example, you need to know whether the entry refers to a file or directory, `os.scandir`'s ability to access more directory information than just the name will be a plus. `os.DirEntry` objects have methods that correspond to the `os.path` functions mentioned in the previous section, including `exists`, `is_dir`, `is_file`, `is_socket`, and `is_symlink`.

`os.scandir` also supports a context manager using `with`, and using one is recommended to ensure resources are properly disposed of. This example code iterates over all of the entries in a directory and prints both the name of the entry and whether it's a file:

```
with os.scandir('..') as my_dir:  
    for entry in my_dir:  
        print(entry.name, entry.is_file())  
  
.config False  
sample_data False
```

12.4 More filesystem operations

In addition to obtaining information about files, Python lets you perform certain filesystem operations directly through a set of basic but highly useful commands in the `os` module.

I describe only those true cross-platform operations in this section. These commands (and the versions of them that `pathlib` supports, as discussed later) not only work on Windows, iOS, and Linux but also work just fine on the filesystem in Colaboratory (which is Linux). Many operating systems have access to more advanced filesystem functions, and you need to check the main Python library documentation for the details.

You've already seen that, to obtain a list of files in a directory, you use `os.listdir`:

```
os.chdir('/content/sample_data')  
os.listdir(os.curdir)  
  
['anscombe.json',  
'README.md',  
'california_housing_train.csv',  
'california_housing_test.csv',  
'mnist_test.csv',  
'mnist_train_small.csv']
```

Note that unlike the list-directory command in many other languages or shells, Python does *not* include the `os.curdir` and `os.pardir` indicators in the list returned by `os.listdir`.

The `glob` function from the `glob` module (named after an old UNIX function that did pattern matching) expands Linux/UNIX shell-style wildcard characters and character sequences in a pathname, returning the files in the current working directory that match. A `*` matches any sequence of characters. A `?` matches any single character. A character sequence (`[h,H]` or `[0-9]`) matches any single character in that sequence:

```
import glob
glob.glob("*")

['anscombe.json',
'README.md',
'california_housing_train.csv',
'california_housing_test.csv',
'mnist_test.csv',
'mnist_train_small.csv']

glob.glob("*json")

['anscombe.json']

glob.glob("?.tmp")

['1.tmp', '2.tmp', 'a.tmp']

glob.glob("[0-9].tmp")

['1.tmp', '2.tmp']
```

To rename (move) a file or directory, use `os.rename`:

```
os.rename('README.md', 'README.md.old')
os.listdir(os.curdir)

['anscombe.json',
'README.md.old',
'1.tmp',
'2.tmp',
'a.tmp',
'california_housing_train.csv',
'california_housing_test.csv',
'mnist_test.csv',
'mnist_train_small.csv']
```

You can use this command to move files across directories as well as within directories.

Remove or delete a data file with `os.remove`:

```
os.remove('a.tmp')
os.listdir(os.curdir)

['anscombe.json',
'README.md.old',
'1.tmp',
'2.tmp',
```

```
'california_housing_train.csv',
'california_housing_test.csv',
'mnist_test.csv',
'mnist_train_small.csv']
```

Note that you can't use `os.remove` to delete directories. This restriction is a safety feature, to ensure that you don't accidentally delete an entire directory substructure.

Files can be created by writing to them, as discussed in chapter 11. To create a directory, use `os.makedirs` or `os.mkdir`. The difference between them is that `os.mkdir` doesn't create any necessary intermediate directories, but `os.makedirs` does:

```
os.makedirs('mydir')
os.listdir(os.curdir)

['anscombe.json',
'README.md.old',
'mydir',
'1.tmp',
'2.tmp',
'california_housing_train.csv',
'california_housing_test.csv',
'mnist_test.csv',
'mnist_train_small.csv']

os.path.isdir('mydir')

True
```

To remove a directory, use `os.rmdir`. This function removes only empty directories. Attempting to use it on a nonempty directory raises an exception:

```
os.rmdir('mydir')
os.listdir(os.curdir)

['anscombe.json',
'README.md.old',
'1.tmp',
'2.tmp',
'california_housing_train.csv',
'california_housing_test.csv',
'mnist_test.csv',
'mnist_train_small.csv']
```

To remove nonempty directories, use the `shutil.rmtree` function. It recursively removes all files in a directory tree. See the Python standard library documentation for details on its use.

12.4.1 More filesystem operations with `pathlib`

Path objects have most of the same methods mentioned earlier. Some differences exist, however. The `iterdir` method is similar to the `os.path.listdir` function except that it returns an iterator of paths rather than a list of strings:

```

new_path = cur_path.joinpath('/content', 'sample_data')
list(new_path.iterdir())

[PosixPath('/content/sample_data/anscombe.json'),
 PosixPath('/content/sample_data/.ipynb_checkpoints'),
 PosixPath('/content/sample_data/README.md.old'),
 PosixPath('/content/sample_data/1.tmp'),
 PosixPath('/content/sample_data/2.tmp'),
 PosixPath('/content/sample_data/california_housing_train.csv'),
 PosixPath('/content/sample_data/california_housing_test.csv'),
 PosixPath('/content/sample_data/mnist_test.csv'),
 PosixPath('/content/sample_data/mnist_train_small.csv')]

```

Note that in a Windows environment, the paths returned are `WindowsPath` objects, whereas on macOS or Linux, they're `PosixPath` objects.

`pathlib` path objects also have a `glob` method built in, which again returns not a list of strings but an iterator of path objects. Otherwise, this function behaves very much like the `glob.glob` function demonstrated previously:

```

list(cur_path.glob("*"))

[PosixPath('anscombe.json'),
 PosixPath('.ipynb_checkpoints'),
 PosixPath('README.md.old'),
 PosixPath('1.tmp'),
 PosixPath('2.tmp'),
 PosixPath('california_housing_train.csv'),
 PosixPath('california_housing_test.csv'),
 PosixPath('mnist_test.csv'),
 PosixPath('mnist_train_small.csv')]

list(cur_path.glob("*json"))

[PosixPath('anscombe.json')]

list(cur_path.glob("?.tmp"))

[PosixPath('1.tmp'), PosixPath('2.tmp')]

list(cur_path.glob("[0-9].tmp"))

[PosixPath('1.tmp'), PosixPath('2.tmp')]

```

To rename (move) a file or directory, use the path object's `rename` method:

```

old_path = Path('README.md.old')
new_path = Path('README.md')
old_path.rename(new_path)
list(cur_path.iterdir())

[PosixPath('anscombe.json'),
 PosixPath('README.md'),
 PosixPath('.ipynb_checkpoints'),

```

```
PosixPath('1.tmp'),
PosixPath('2.tmp'),
PosixPath('california_housing_train.csv'),
PosixPath('california_housing_test.csv'),
PosixPath('mnist_test.csv'),
PosixPath('mnist_train_small.csv')]
```

You can use this command to move files across directories as well as within directories.

To remove or delete a data file, you can use `unlink`:

```
new_path = Path('1.tmp')
new_path.unlink()
list(cur_path.iterdir())

[PosixPath('anscombe.json'),
PosixPath('README.md'),
PosixPath('.ipynb_checkpoints'),
PosixPath('2.tmp'),
PosixPath('california_housing_train.csv'),
PosixPath('california_housing_test.csv'),
PosixPath('mnist_test.csv'),
PosixPath('mnist_train_small.csv')]
```

Note that, as with `os.remove`, you can't use the `unlink` method to delete directories. This restriction is a safety feature, to ensure that you don't accidentally delete an entire directory substructure.

To create a directory by using a path object, use the path object's `mkdir` method. If you give the `mkdir` method a `parents=True` parameter, it creates any necessary intermediate directories; otherwise, it raises a `FileNotFoundException` if an intermediate directory isn't there:

```
new_path = Path('mydir')
new_path.mkdir(parents=True)
list(cur_path.iterdir())

[PosixPath('anscombe.json'),
PosixPath('README.md'),
PosixPath('.ipynb_checkpoints'),
PosixPath('mydir'),
PosixPath('2.tmp'),
PosixPath('california_housing_train.csv'),
PosixPath('california_housing_test.csv'),
PosixPath('mnist_test.csv'),
PosixPath('mnist_train_small.csv')]

new_path.is_dir()

True
```

To remove a directory, use the `rmdir` method. This method removes only empty directories. Attempting to use it on a nonempty directory raises an exception:

```

new_path = Path('mydir')
new_path.rmdir()
list(cur_path.iterdir())

[PosixPath('anscombe.json'),
 PosixPath('README.md'),
 PosixPath('.ipynb_checkpoints'),
 PosixPath('2.tmp'),
 PosixPath('california_housing_train.csv'),
 PosixPath('california_housing_test.csv'),
 PosixPath('mnist_test.csv'),
 PosixPath('mnist_train_small.csv')]

```

As you can see from the previous examples, `Path` objects are a bit different from the functions offered by the `os` library. It's important to remember that `Path` objects are objects, each of which represents a particular path or location on your system, while the `os` functions are designed to operate on any path or directory location.

12.5 Processing all files in a directory subtree

Finally, a highly useful function for traversing recursive directory structures is the `os.walk` function. You can use it to walk through an entire directory tree, returning three things for each directory it traverses: the root, or path, of that directory; a list of its subdirectories; and a list of its files.

`os.walk` is called with the path of the starting, or top, directory and can have three optional arguments: `os.walk(directory, topdown=True, onerror=None, followlinks=False)`. `directory` is a starting directory path; if `topdown` is `True` or not present, the files in each directory are processed *before* its subdirectories, resulting in a listing that starts at the top and goes down; whereas if `topdown` is `False`, the subdirectories of each directory are processed *first*, giving a bottom-up traversal of the tree. The `onerror` parameter can be set to a function to handle any errors that result from calls to `os.listdir`, which are ignored by default. `os.walk` by default doesn't walk down into folders that are symbolic links unless you give it the `followlinks=True` parameter.

When called, `os.walk` creates an iterator that recursively applies itself to all the directories contained in the `top` parameter. In other words, for each subdirectory `subdir` in `names`, `os.walk` recursively invokes a call to itself, of the form `os.walk(subdir, ...)`. Note that if `topdown` is `True` or not given, the list of subdirectories may be modified (using any of the list-modification operators or methods) before its items are used for the next level of recursion; you can use this to control into which—if any—subdirectories `os.walk` will descend.

To get a feel for `os.walk`, I recommend iterating over the tree and printing out the values returned for each directory. As an example of the power of `os.walk`, list the current working directory and all of its subdirectories along with a count of the number of entries in each of them but without listing the contents of any `.config` directories:

```

import os
for root, dirs, files in os.walk(os.curdir):

```

```
print("{0} has {1} files".format(root, len(files)))
if ".config" in dirs:
    dirs.remove(".config")
```

Checks for directory named .config
Removes .config (only the .config directory) from directory list

This example is complex, and if you want to use `os.walk` to its fullest extent, you should probably play around with it quite a bit to understand the details of what's going on.

The `copytree` function of the `shutil` module recursively makes copies of all the files in a directory and all of its subdirectories, preserving permission mode and stat (that is, access/modify times) information. `shutil` also has the already mentioned `rmtree` function for removing a directory and all of its subdirectories, as well as several functions for making copies of individual files. See the standard library documentation for details.

12.6 More file operations

How might you calculate the total size of all files ending with `.test` that aren't symlinks in a directory? If your first answer was using `os` and `os.path`, also try it with `pathlib`, and vice versa.

Write some code that builds off your solution and moves the `.test` files detected previously to a new subdirectory in the same directory called "backup."

Note that you will need to be sure you have at least some files ending in `.test` in your starting directory and its subdirectories. In Colaboratory, just be sure to execute the cell right before the answers.

12.6.1 Solving the problem with AI-generated code

This exercise calls for a bit more original thinking—while the first part of the problem is not that complex, it requires combining several elements of either the `os` or the `pathlib` libraries to achieve a working solution, which is then built upon for the full solution. Since there are two libraries that can be used to accomplish the task, it's good experience to modify your solution to use the other library.

The prompts for Copilot are

Write a Python script to calculate the total size of all files with a `.test` extension that are not symlinks in the current directory and subdirectories.

To add moving the files to a backup directory:

Modify that script to also move the files with a `.test` extension to a subdirectory of the current directory called `backup`

And to refactor to use `pathlib`:

Refactor the previous script using the `pathlib` library instead of `os` and `shutil`

The prompts for Colaboratory are the same, with the addition “in the previous cell” for the two refactorings.

12.6.2 Solutions and discussion

While the scripts created in this lab are fairly short, they are also fairly dense in that they require several different elements of the libraries, and those elements must be combined in specific ways.

THE HUMAN SOLUTION

My first solution uses `pathlib`, since treating paths as objects makes for clearer, more coherent code in my opinion. The first part of the task, getting the total size of all `.test` files, is fairly straightforward, thanks to the various methods of the `Path` class:

```
import pathlib
cur_path = pathlib.Path("sample_data")

size = 0
for text_path in cur_path.rglob("*.test"):
    if not text_path.is_symlink():
        size += text_path.stat().st_size

print(size)
```

There are three things that this code needs to do—find the target files, which the `glob` method does; check to see if they are symlinks, which is handled by a specific method; and get the size via the `stat()` method and add it to the total size. Having all of those methods be part of the `Path` object for the current directory makes the code concise and clear.

The next task is to refactor that code so that, in addition to finding the target files and totaling up their size, we move those files to a subdirectory called “backup”:

```
import pathlib
cur_path = pathlib.Path("sample_data")
new_path = pathlib.Path("backup")
new_path.mkdir(exist_ok=True)

size = 0
for text_path in cur_path.rglob("*.test"):
    if not text_path.is_symlink():
        size += text_path.stat().st_size
        text_path.rename(new_path / text_path.name)

print(size)
```

The first modification is that we need to be sure that the “backup” directory exists. That means that we need a `Path` object for the “backup” directory. We can then use that object to create the actual directory on disk only if it doesn’t already exist. The

`exist_ok=True` parameter makes sure that a directory is created if it doesn't exist but that no exception is raised if the directory already exists.

The other necessary operation is to actually move the file. In the Unix tradition, we can move a file by changing the name of its location. So, in this case, we can take the current file's path and add our backup directory path to its beginning and tell the file's path to rename itself to the new value. The use of "rename" to move something may seem odd, but keep in mind that the actual location of the bytes on the disk does not move, just the way we refer to its location, which we are in effect giving a new name.

The last part of the exercise is to refactor our solution to use the other library's approach. Since I started with `pathlib`, it's now time to use the `os` library. Normally I would probably just refactor the final answer, but to keep things clear and consistent, I've done it in the same two steps—first just getting the total size and then creating a version to move the files:

```
import os
cur_path = "."

size = 0
for root, dirs, files in os.walk(os.curdir):
    for file in files:
        test_path = os.path.join(root, file)
        if (not os.path.islink(test_path) and
            os.path.splitext(test_path)[-1] == '.test'):
            print(os.path.join(root, file))
            size += os.path.getsize(test_path)
print(f"{size}")
```

Annotations for the `os.walk` loop:

- A bracket on the right side of the loop body is labeled **Visits all directories**.
- Two brackets on the right side of the inner loop are labeled **Checks for files ending in .test and makes sure they are not symlinks**.
- A bracket on the right side of the `if` block is labeled **Gets size and adds total**.

This solution uses `os.walk` to get all of the files in the directory and its subdirectories, and it's clearly a bit more cumbersome and harder to understand. The main loop uses `os.walk` to get each directory and uses that directory recursively as the root directory until we have visited every directory under the current directory. For each directory, we need to check to make sure that the file is not a symlink and that it ends with `.test`. If so, we can get its size and add that to our total.

The refactor to move the target files requires similar changes to the `pathlib` version, except that we don't have `pathlib`'s `/` operator to join parts of pathnames:

```
import os

cur_path = "."
new_path = os.path.join(cur_path, "backup")

# create backup directory
os.makedirs(new_path, exist_ok=True)
size = 0
for root, dirs, files in os.walk(os.curdir):
    for file in files:
        test_path = os.path.join(root, file)
        print(test_path)
```

An annotation for the `os.makedirs` call:

- A bracket on the right side of the call is labeled **Creates backup directory if needed**.

```

if (not os.path.islink(test_path)
    and os.path.splitext(test_path)[-1] == ".test"):
    print(os.path.join(root, file))
    size += os.path.getsize(test_path)
    os.rename(test_path, os.path.join(new_path,
                                     os.path.basename(test_path)))
    ← Moves to new
    directory by renaming
print(f"{size}")

```

The changes to move the files are similar to those used with `Path` objects, but the code is a bit longer and to my mind definitely harder to read. Still, if you have a definite preference, it's just fine to use this approach. While the Zen of Python says, "there should be one—and preferably only one—obvious way to do it," in this case, as the language has developed, there are two perfectly fine ways to do file operations of this type.

THE AI SOLUTIONS

Solving the problem with AI proved a little bit more involved. As mentioned earlier, the prompt for the first part of the problem is

Write a Python script to calculate the total size of all files with an extension that are not symlinks in the current directory and subdirectories.

In response, Copilot offered the following plan for the first part:

- 1 Import the `os` module.
- 2 Initialize a variable `size` to 0.
- 3 Use `os.walk` to iterate over all files in the current directory and its subdirectories.
- 4 For each file, check if it is not a symlink using `os.path.islink`.
- 5 If it is not a symlink, check if its extension is `.test` using `os.path.splitext`.
- 6 If its extension is `.test`, add its size to `size` using `os.path.getsize`.
- 7 After the loop, print `size`.

This is a nicely detailed plan of the steps involved in solving the problem, and it's quite handy to see what the bot is going to do:

```

# txt_file_size.py

import os

size = 0
for root, dirs, files in os.walk(os.curdir):
    for file in files:
        file_path = os.path.join(root, file)
        if (not os.path.islink(file_path)
            and os.path.splitext(file)[1] == '.test'):
            size += os.path.getsize(file_path)

print(f"Total size of all files that are not symlinks: {size} bytes")

```

For the second part, this is the prompt:

```
Modify that script to also move the files with a .test extension to an  
subdirectory of the current directory called backup
```

Copilot answered with the following plan:

- 1 Import the `os` and `shutil` modules.
- 2 Initialize a variable `size` to 0.
- 3 Create a directory named `backup` in the current directory using `os.makedirs`, if it doesn't already exist.
- 4 Use `os.walk` to iterate over all files in the current directory and its subdirectories.
- 5 For each file, check if it is not a symlink using `os.path.islink`.
- 6 If it is not a symlink, check if its extension is `.test` using `os.path.splitext`.
- 7 If its extension is `.test`, add its size to `size` using `os.path.getsize`.
- 8 Move the file to the backup directory using `shutil.move`.
- 9 After the loop, print `size`.

The code that resulted was straightforward and not that different from my solution:

```
# txt_file_size_and_move.py

import os
import shutil

size = 0
os.makedirs('backup', exist_ok=True)
for root, dirs, files in os.walk(os.curdir):
    for file in files:
        file_path = os.path.join(root, file)
        if (not os.path.islink(file_path)
            and os.path.splitext(file)[1] == '.test'
            and "backup" not in file_path):
            size += os.path.getsize(file_path)
            shutil.move(file_path, 'backup')

print(f"Total size of all .test files that are not symlinks: {size} bytes")
```

The only thing that is really different from the human solution is the use of the `shutil` library to move the target files. This is certainly an acceptable way to do it, but it ignores the fact that `os.rename` can do the same job without importing another library. My personal preference would be to import as few libraries as possible. In this example, it's not hard to keep track of what's going on, but as projects grow, they often end up with more and more imports, which can make things harder to parse. In my opinion, the best way to keep that from happening is to be slow to add imports.

The next and final step is to refactor the code to use the other option—in this case, to use `pathlib` instead of `os` and `shutil`. The prompt used for this is

Refactor the previous script using the `pathlib` library instead of `os` and `shutil`

The plan for converting this to use `pathlib` offered by Copilot was

- 1 Import the `pathlib` module.
- 2 Initialize a variable `size` to 0.
- 3 Create a `pathlib.Path` object for the current directory and a subdirectory named `backup`.
- 4 Create the backup directory if it doesn't already exist using `Path.mkdir`.
- 5 Use `Path.rglob` to iterate over all files in the current directory and its subdirectories.
- 6 For each file, check if it is not a symlink using `Path.is_symlink`.
- 7 If it is not a symlink, check if its extension is `.test` using `Path.suffix`.
- 8 If its extension is `.test`, add its size to `size` using `Path.stat`.
- 9 Move the file to the backup directory using `Path.rename`.
- 10 After the loop, print `size`.

The resulting code was quite similar to the human solution:

```
# txt_file_size_and_move.py

from pathlib import Path

size = 0
backup_dir = Path('backup')
backup_dir.mkdir(exist_ok=True)

for file in Path().rglob('*'):
    if not file.is_symlink() and file.suffix == '.test':
        size += file.stat().st_size
        file.rename(backup_dir / file.name)

print(f"Total size of all .test files that are not symlinks: {size} bytes")
```

In fact, there isn't much to say about this code that we haven't already said in discussing the human solution earlier.

The AI bot for Colaboratory was a little less ideal. While three of the four solutions offered broke putting the main code into a function, we chose the one that didn't, to keep the code more or less similar. In any case, all of Colaboratory's solutions took the same approach to the core functionality.

The prompt used is

Write a Python script to calculate the total size of all files with a `.test` extension that are not symlinks in the current directory and subdirectories.

And Colaboratory's solution was

```
# prompt: Write a Python script to calculate the total size of all files
#           with a .test extension that are not symlinks in the current
#           directory and subdirectories.

import os

total_size = 0

for root, _, files in os.walk('.'):
    for file in files:
        if (file.endswith('.test') and
            not os.path.islink(os.path.join(root, file))):
            total_size += os.path.getsize(os.path.join(root, file))

print(f"Total size of .test files: {total_size} bytes")
```

This code is perfectly fine, with only a couple of things worth noting. First, the use of “_” in the first `for` loop is a Python idiom. “_” is a legal character for a variable name and is often used by itself to indicate a temporary loop variable whose contents are not important or not used. Second, the way that the `if` statement is constructed, while legal, is rather long and hard to read.

The second part of the assignment, to move the target files to a backup directory, uses the prompt

Modify that script in the previous cell to also move the files with a `.test` extension to a subdirectory of the current directory called `backup`

The resulting code was reasonable:

```
# prompt: Modify that script in the previous cell to also move the files
#           with a .test extension to a subdirectory of the current directory
#           called backup

import os

total_size = 0
backup_dir = os.path.join(os.getcwd(), 'backup')

# Create the backup directory if it doesn't exist
if not os.path.exists(backup_dir):
    os.makedirs(backup_dir)

for root, _, files in os.walk('.'):
    for file in files:
        if (file.endswith('.test') and
            not os.path.islink(os.path.join(root, file))):
            total_size += os.path.getsize(os.path.join(root, file))
            # Move the file to the backup directory
            os.rename(os.path.join(root, file),
```

```
        os.path.join(backup_dir, file))

print(f"Total size of .test files: {total_size} bytes")
```

In addition to the previous comments, it's worth noting that, in creating the backup directory, this version explicitly checks for the existence of the directory instead of relying on the `exist_ok=True` parameter to simplify the code.

The final step, the refactor to use `pathlib`, is where Colaboratory disappoints. The prompt used is

Rewrite the script in the previous cell to use `pathlib` instead of `os` and `os.path`. Do not use `os` or `os.path` modules.

In spite of the explicit instructions in the prompt, the solution returned used `os` and `os.path`:

```
# prompt: Rewrite the script in the previous cell to use pathlib instead of
#         os and os.path. Do not use os or os.path modules.

from pathlib import Path

total_size = 0
backup_dir = Path.cwd().joinpath('backup')

# Create the backup directory if it doesn't exist
if not backup_dir.exists():
    backup_dir.mkdir(parents=True)

for root, _, files in Path('.').walk():
    for file in files:
        if file.endswith('.test') and not file.is_symlink():
            total_size += file.stat().st_size
            # Move the file to the backup directory
            file.rename(backup_dir.joinpath(file))

print(f"Total size of .test files: {total_size} bytes")
```

This solution has several things worth commenting on. First, this solution uses `Path.walk()`, which was only introduced in Python 3.12. That's not a problem in itself, but it might be a problem for any environment with an older version of Python. In fact, at the time of writing, Colaboratory itself was only at Python 3.10, so it was unable to run its own suggested solution.

In addition, using `walk()` when we don't absolutely need to means that there has to be one `for` loop to get directories and a nested `for` loop to process the files for each directory. Nesting `for` loops can quite often cause the code to be much slower, particularly as the number of elements increases, so one advantage of using `pathlib` is that the `rglob` method handles recursion into all the directories. The final concern is that if the code uses `walk()` instead of the recursive `rglob`, it will need to check explicitly to see if the filename matches the target.

The final conclusion is that, while both Copilot and Colaboratory came up with acceptable solutions, I would prefer Copilot, whose solution was on a par with the human solution.

Summary

- Python provides a group of functions and constants that handle filesystem references (pathnames) and filesystem operations in a manner independent of the underlying operating system.
- For more advanced and specialized filesystem operations that typically are tied to a certain operating system or systems, the two main Python libraries are the `os` and `pathlib` modules.
- Python can handle both relative and absolute paths and pathnames in an operating system agnostic way using both the `os` and `pathlib` libraries.
- Using `os` functions or `pathlib` methods, you can fully access information about files and directories.
- Both file manipulation libraries have functions or methods to move, rename, copy, and delete files and directories.
- All the files in a directory can be processed using either `os.walk` or `Path.rglob` and methods in either the `os` or `pathlib` libraries.
- For convenience, a summary of the functions discussed in this chapter is given in table 12.1 and 12.2.

Table 12.1 Summary of filesystem values and functions

Function	Filesystem value or operation
<code>os.getcwd()</code> , <code>Path.cwd()</code>	Gets the current directory
<code>os.name</code>	Provides generic platform identification
<code>sys.platform</code>	Provides specific platform information
<code>os.environ</code>	Maps the environment
<code>os.listdir(path)</code>	Gets files in a directory
<code>os.scandir(path)</code>	Gets an iterator of <code>os.DirEntry</code> objects for a directory
<code>os.chdir(path)</code>	Changes directory
<code>os.path.join(elements)</code> , <code>Path.joinpath(elements)</code>	Combines elements into a path
<code>os.path.split(path)</code>	Splits the path into a base and tail (the last element of the path)
<code>Path.parts</code>	A tuple of the path's elements
<code>os.path.splitext(path)</code>	Splits the path into a base and a file extension

Table 12.1 Summary of filesystem values and functions (continued)

Function	Filesystem value or operation
<code>Path.suffix</code>	The path's file extension
<code>os.path.basename(path)</code>	Gets the basename of the path
<code>Path.name</code>	The basename of the path
<code>os.path.commonprefix(list_of_paths)</code>	Gets the common prefix for all paths on a list
<code>os.path.expanduser(path)</code>	Expands ~ or ~user to a full pathname
<code>os.path.expandvars(path)</code>	Expands environment variables
<code>os.path.exists(path)</code>	Tests to see if a path exists
<code>os.path.isdir(path), Path.is_dir()</code>	Tests to see if a path is a directory
<code>os.path.isfile(path), Path.is_file()</code>	Tests to see if a path is a file
<code>os.path.islink(path), Path.is_link()</code>	Tests to see if a path is a symbolic link (not a Windows shortcut)
<code>os.path.ismount(path)</code>	Tests to see if a path is a mount point
<code>os.path.isabs(path), Path.is_absolute()</code>	Tests to see if a path is an absolute path
<code>os.path.samefile(path_1, path_2)</code>	Tests to see if two paths refer to the same file
<code>os.path.getsize(path)</code>	Gets the size of a file
<code>os.path.getmtime(path)</code>	Gets the modification time
<code>os.path.getatime(path)</code>	Gets the access time
<code>os.rename(old_path, new_path)</code>	Renames a file
<code>os.mkdir(path)</code>	Creates a directory
<code>os.makedirs(path)</code>	Creates a directory and any needed parent directories
<code>os.rmdir(path)</code>	Removes a directory
<code>glob.glob(pattern)</code>	Gets matches to a wildcard pattern
<code>os.walk(path)</code>	Gets all filenames in a directory tree

Table 12.2 Partial list of `pathlib` properties and functions

Method or property	Value or operation
<code>Path.cwd()</code>	Gets the current directory
<code>Path.joinpath(elements) or Path / element / element</code>	Combines elements into a new path
<code>Path.parts</code>	A tuple of the path's elements
<code>Path.suffix</code>	The path's file extension

Table 12.2 Partial list of `pathlib` properties and functions (*continued*)

Method or property	Value or operation
<code>Path.name</code>	The basename of the path
<code>Path.exists()</code>	Tests to see if a path exists
<code>Path.is_dir()</code>	Tests to see if a path is a directory
<code>Path.is_file()</code>	Tests to see if a path is a file
<code>Path.is_symlink()</code>	Tests to see if a path is a symbolic link (not a Windows shortcut)
<code>Path.is_absolute()</code>	Tests to see if a path is an absolute path
<code>Path.samefile(Path2)</code>	Tests to see if two paths refer to the same file
<code>Path1.rename(Path2)</code>	Renames a file
<code>Path.mkdir([parents=True])</code>	Creates a directory; if <code>parents</code> is <code>True</code> , also creates needed parent directories
<code>Path.rmdir()</code>	Removes a directory
<code>Path.glob(pattern)</code>	Gets matches to a wildcard pattern

13

Reading and writing files

This chapter covers

- Opening files and `file` objects
- Closing files
- Opening files in different modes
- Reading and writing text or binary data
- Redirecting screen input/output
- Using the `struct` module
- Pickling objects into files
- Shelving objects

File operations are a common feature in the daily routines of many developers. The most common file type is the simple text file, which is used in a wide variety of situations, from raw data files to log files to source code and more. If text files can't do the job, binary files can be used to store almost anything, often by using the `struct` module for specific binary formats or even the `pickle` and `shelve` modules to store and retrieve Python objects. Let's walk through reading and writing data to files with Python.

13.1 Opening files and file objects

One truth about computing is that files are everywhere. The data we use is in files; the code we use to process it is files; and much of the time when we process data, we write the results to files. We download them, copy them, search them, attach them, encrypt them, and archive them. But most of all, the single most common thing you'll want to do with files is open and read them.

In Python, you open and read a file by using the built-in `open` function and various built-in reading operations. Instead of worrying about filenames, Python uses that `file` object to keep track of a file and how much of the file has been read or written. All Python file I/O is done using `file` objects. The following short Python program reads in one line from a text file named `myfile`:

```
with open('myfile', 'r') as file_object:  
    line = file_object.readline()
```

The first argument to the `open` function is a pathname. In the previous example, you're opening what you expect to be an existing file called "myfile" in the current working directory. `open` doesn't read anything from the file; instead, it returns an object called a `file` object that you can use to access the opened file.

Note also that this example uses the `with` keyword, indicating that the file will be opened with a context manager, which I'll explain in more detail in chapter 14. For now, it's enough to note that this style of opening files better manages potential I/O errors and is generally preferred.

File objects have various methods to read and write data, change position in the file, and so on. In the previous example, the first call to `readline` reads and returns the first line in the `file` object, everything up to and including the first newline character (or the entire file if there's no newline character in the file); the next call to `readline` returns the second line, if it exists, and so on.

The following opens a file at an absolute location—`c:\My Documents\test\myfile`:

```
import os  
file_name = os.path.join("c:", "My Documents", "test", "myfile")  
file_object = open(file_name, 'r')
```

This example opens the file directly (but doesn't read anything) and also shows a safe way of creating a path string, the `os.path.join()` function, which will join the elements according to the current operating system's conventions.

13.2 Closing files

After all data has been read from or written to a `file` object, it should be closed. Closing a `file` object frees up system resources, allows the underlying file to be read or written to by other code, and in general makes the program more reliable. For small scripts, not closing a `file` object generally doesn't have much of an effect; `file` objects are automatically closed when the script or program terminates. For larger programs,

too many open `file` objects may exhaust system resources, causing the program to abort.

You close a `file` object by using the `close` method when the `file` object is no longer needed. The earlier short program then becomes

```
file_object = open("myfile", 'r')
line = file_object.readline()
# . . . any further reading on the file_object . .
file_object.close()
```

Using the keyword `with` (which automatically invokes a context manager) is also a good way to automatically close files when you're done:

```
with open("myfile", 'r') as file_object:
    line = file_object.readline()
    # . . . any further reading on the file_object . . .
```

In this example, even though there is no explicit call to the `close()` method, the file is closed as part of the context manager, initiated by the `with` keyword.

13.3 **Opening files in write or other modes**

The second argument of the `open` command is a string denoting how the file should be opened. '`r`' means "Open the file for reading," '`w`' means "Open the file for writing" (any data already in the file will be erased), and '`a`' means "Open the file for appending" (new data will be appended to the end of any data already in the file). If you want to open the file for reading, you can leave out the second argument; '`r`' is the default. The following short program writes "Hello, World" to a file:

```
file_object = open("myfile", 'w')
file_object.write("Hello, World\n")
file_object.close()
```

Depending on the operating system, `open` may also have access to additional file modes. These modes aren't necessary for most purposes. As you write more advanced Python programs, you may want to consult the Python reference manuals for details.

`open` can take an optional third argument, which defines how reads or writes for that file are buffered. *Buffering* is the process of holding data in memory until enough data has been requested or written to justify the time cost of doing a disk access. Other parameters to `open` control the encoding for text files and the handling of newline characters in text files. Again, these features aren't things you often need to worry about, but as you become more advanced in your use of Python, you will want to read up on them.

13.4 **Functions to read and write text or binary data**

I've already presented the most common text file-reading function, `readline`. This function reads and returns a single line from a `file` object, including any newline character on the end of the line. If there's nothing more to be read from the file,

`readline` returns an empty string, which makes it easy to (for example) count the number of lines in a file:

```
file_object = open("myfile", 'r')
count = 0
while file_object.readline() != "":
    count = count + 1
print(count)
file_object.close()
```

For this particular problem, an even shorter way to count all the lines is to use the built-in `readlines` method, which reads *all* the lines in a file and returns them as a list of strings, one string per line (with trailing newlines still included):

```
file_object = open("myfile", 'r')
print(len(file_object.readlines()))
file_object.close()
```

If you happen to be counting all the lines in a huge file, of course, this method may cause your computer to run out of memory because it reads the entire file into memory at once. It's also possible to overflow memory with `readline` if you have the misfortune to try to read a line from a huge file that contains no newline characters, although this situation is highly unlikely. To handle such circumstances, both `readline` and `readlines` can take an optional argument affecting the amount of data they read at any one time. See the Python reference documentation for details.

Another way to iterate over all of the lines in a file is to treat the `file` object as an iterator in a `for` loop:

```
file_object = open("myfile", 'r')
count = 0
for line in file_object:
    count = count + 1
print(count)
file_object.close()
```

This method has the advantage that the lines are read into memory and processed one line at a time, so even with large files, running out of memory isn't a concern. The other advantage of this method is that it's simpler and easier to read.

A possible problem with reading text files is that the lines may be terminated by different characters, depending on which OS they were created in. In text mode, on a Mac the default line ending is `\r`, whereas on Windows "`\r\n`" pairs are used. By default, a Python file object will read text files in "universal" mode, which translates both endings to "`\n`". If those strings are written to another text file, they will no longer be using the OS's default line endings, which may be a problem.

You can specify the treatment of newline characters by using the `newline` parameter when you open the file, specifying `newline="\n"`, `"\r"`, or `"\r\n"`, which forces only that string to be used as a newline:

```
input_file = open("myfile", newline="\n")
```

This example forces only "`\n`" to be considered to be a newline. If the file has been opened in binary mode, the `newline` parameter isn't needed, because all bytes are returned exactly as they are in the file.

You can also use `newline=""` with `open`, which will accept all of the various options as line endings but will return whatever was used in the file with no translation.

The write methods that correspond to the `readline` and `readlines` methods are the `write` and `writelines` methods. Note that there's no `writeline` function. `write` writes a single string, which can span multiple lines if newline characters are embedded within the string, as in this example:

```
myfile.write("Hello\nWorld\nHow are you?")
```

`write` doesn't write out a newline after it writes its argument; if you want a newline in the output, you must put it there yourself. If you open a file in text mode (using `w`), any `\n` characters are mapped back to the platform-specific line endings (that is, '`\r\n`' on Windows or '`\r`' on macOS platforms). Again, opening the file with a specified newline prevents this situation.

`writelines` is something of a misnomer because it doesn't necessarily write lines; it takes a list of strings as an argument and writes them, one after the other, to the given `file` object without writing newlines. If the strings in the list end with newlines, they're written as lines; otherwise, they're effectively concatenated in the file. But `writelines` is a precise inverse of `readlines` in that it can be used on the list returned by `readlines` to write a file identical to the file that `readlines` read from. Assuming that `myfile.txt` exists and is a text file, this bit of code creates an exact copy of `myfile.txt` called `myfile2.txt`:

```
input_file = open("myfile.txt", 'r', newline="")
lines = input_file.readlines()
input_file.close()
output = open("myfile2.txt", 'w')
output.writelines(lines)
output.close()
```

While these basic operations to read and write files are simple, in my experience, they are all you need for a wide variety of cases in everyday coding.

13.4.1 Using binary mode

On some occasions, you may want to read all the data in a file into a single `bytes` object, especially if the data isn't a string, and you want to get it all into memory so you can treat it as a byte sequence. Or you may want to read data from a file as `bytes` objects of a fixed size. You may be reading data without explicit newlines, for example, where each line is assumed to be a sequence of characters of a fixed size. To do so, open the file for reading in binary mode (`'rb'`) and use the `read` method. Without

any argument, this method reads all of a file from the current position and returns that data as a bytes object. With a single-integer argument, it reads that number of bytes (or less, if there isn't enough data in the file to satisfy the request) and returns a bytes object of the given size:

```
input_file = open("myfile", 'rb')           ← Reads in binary mode
header = input_file.read(4)                  ← Reads four bytes into header
data = input_file.read()                   ← Reads remaining data as bytes
input_file.close()
```

The first line opens a file for reading in binary mode, the second line reads the first four bytes as a header in bytes, and the third line reads the rest of the file in bytes as a single piece of data.

Keep in mind that files open in binary mode deal only in bytes, not strings. To use the data as strings, you must decode any bytes objects to string objects. This point is often important in dealing with network protocols, where data streams often behave as text files but need to be interpreted as bytes, not strings.

Quick check: Binary mode

What is the significance of adding a "b" to the file open mode string, as in `open("file", "wb")`?

Suppose that you want to open a file named `myfile.txt` and write additional data on the end of it. What command would you use to open `myfile.txt`? What command would you use to reopen the file to read from the beginning?

13.5 Reading and writing with *pathlib*

In addition to its path-manipulation powers discussed in chapter 12, a `Path` object can be used to read and write text and binary files. This capability can be convenient because no `open` or `close` is required, and separate methods are used for text and binary operations. One limitation, however, is that you have no way to append when using `Path` methods, because writing replaces any existing content:

```
from pathlib import Path
p_text = Path('my_text_file')
p_text.write_text('Text file contents')

18                                     ← Number of bytes written

p_text.read_text()

'Text file contents'

p_binary = Path('my_binary_file')
```

```
p_binary.write_bytes(b'Binary file contents')

20          ← Number of bytes written

p_binary.read_bytes()

b'Binary file contents' ← Data as bytes
```

In these examples, the `Path` object takes care of reading and writing in binary mode by using its `read_bytes` and `write_bytes` methods.

13.6 Terminal input/output and redirection

Sometimes you don't want or need to read and write files, but rather you want to interact with a command-line user. This can be handy for simple utilities and for testing, for example. You can use the built-in `input` method to prompt for and read an input string:

```
x = input("enter file name to use: ")

enter file name to use: myfile

x

'myfile'
```

The prompt to the user is optional and can be any string (including f-strings). The user's input is terminated by pressing the Enter key, but the newline at the end of the input line is stripped off. `input` handles everything as a string, so to read in numbers using `input`, you need to explicitly convert the string that it returns to the correct number type. The following example uses `int`:

```
x = int(input("enter your number: "))

enter your number: 39

x

39
```

In this example, the code will try to convert anything entered into an `int`, which will raise an exception if that isn't possible.

`input` writes its prompt to the *standard output* and reads from the *standard input*. Lower-level access to standard input and output and to *standard error* can be obtained by using the `sys` module, which has `sys.stdin`, `sys.stdout`, and `sys.stderr` attributes. These attributes can be treated as specialized `file` objects.

For `sys.stdin`, you have the `read`, `readline`, and `readlines` methods. For `sys.stdout` and `sys.stderr`, you can use the standard `print` function as well as the `write` and `writelines` methods, which operate as they do for other `file` objects:

```

import sys
print("Write to the standard output.")

Write to the standard output.

sys.stdout.write("Write to the standard output.\n")
Write to the standard output.
30

```

sys.stdout.write returns the number of characters written in a command window but not in Colaboratory.

The following example shows how you can use `sys.stdin.readline()` to get input. *This will not work in Colaboratory*, again, since it takes fuller control of standard input and output. The workaround is to save the code as a Python file and run it directly using the ! prefix, as mentioned in chapter 11 in section 11.1.1:

```
s = sys.stdin.readline()
```

An input line

```
s
```

```
'An input line\n'
```

Usually there is little need to use standard input and output, but in some specific cases it can be handy, particularly if you want to redirect I/O to or from files.

You can redirect standard input to read from a file. Similarly, standard output or standard error can be set to write to files and then programmatically restored to their original values by using `sys.__stdin__`, `sys.__stdout__`, and `sys.__stderr__`:

```

import sys
f = open("outfile.txt", 'w')
sys.stdout = f
sys.stdout.writelines(["A first line.\n",
                      "A second line.\n"])
print("A line from the print function") | Writes to outfile.txt (redirected sys.stdout)

sys.stdout = sys.__stdout__
f.close() | Resets sys.stdout to default

```

Opens file

Sets outfile.txt to be new sys.stdout

Since we changed `sys.stdout` to be the file `outfile.txt`, after the call to `sys.stdout.writelines`, the file `outfile.txt` will contain two lines: “A first line” and “A second line.” Likewise, the `print` function will also write to that file, so at the end of this example, `outfile.txt` will also contain a third line, “A line from the `print` function” :

```

A first line.
A second line.
A line from the print function

```

Warning about Colaboratory and Jupyter

Since Colaboratory (and Jupyter in general) takes more control of standard input and output, the use of `sys.__stdout__` (and `sys.__stdin__` and `sys.__stderr__`) shown earlier won't work as indicated in a Colaboratory cell; instead, you need to save the old value of `sys.stdout` before changing it and then restore the old value after using the redirected value:

```
import sys
f = open("outfile.txt", 'w')
old_sys_stdout = sys.stdout
sys.stdout = f
sys.stdout.writelines(["A first line.\n", "A second line.\n"])
print("A line from the print function")
sys.stdout = old_sys_stdout
f.close()
! cat outfile.txt
```

The `print` function also can be redirected to any file without changing standard output, by using its `file` parameter:

```
import sys
f = open("outfile.txt", 'w')
print("A first line.\n", "A second line.\n", file=f)
f.close()
```

This will write two lines to the file:

```
A first line.
A second line.
```

Redirecting the output of `print` can be useful, since `print` has a simpler and more familiar syntax. While the standard output is redirected, you receive prompts and tracebacks from errors but no other output.

You'd normally use this technique when you're running a script or program. But it sometimes happens that, during an interactive session, you will want to temporarily redirect standard output to a file to capture what might otherwise scroll off the screen. The short module shown in the following listing implements a set of functions that provides this capability.

Listing 13.1 File mio.py

```
"""mio: module, (contains functions capture_output, restore_output,
    print_file, and clear_file )"""
import sys
_file_object = None
def capture_output(file="capture_file.txt"):
```

```
"""capture_output(file='capture_file.txt'): redirect the standard
output to 'file'."""
global _file_object
print("output will be sent to file: {}".format(file))
print("restore to normal by calling 'mio.restore_output()'")
_file_object = open(file, 'w')
sys.stdout = _file_object

def restore_output():
    """restore_output(): restore the standard output back to the
    default (also closes the capture file)"""
    global _file_object
    sys.stdout = sys.__stdout__
    _file_object.close()
    print("standard output has been restored back to normal")

def print_file(file="capture_file.txt"):
    """print_file(file="capture_file.txt"): print the given file to the
    standard output"""
    f = open(file, 'r')
    print(f.read())
    f.close()

def clear_file(file="capture_file.txt"):
    """clear_file(file="capture_file.txt"): clears the contents of the
    given file"""
    f = open(file, 'w')
    f.close()
```

Here, `capture_output()` redirects standard output to a file that defaults to “capture_file.txt.” The function `restore_output()` restores standard output to the default. Assuming `capture_output` hasn’t been executed, `print_file()` prints this file to the standard output, and `clear_file()` clears its current contents.

Try this: Redirecting output

Write some code to use the `mio.py` module in listing 13.1 to capture all the print output of a script to a file named `myfile.txt`, reset the standard output to the screen, and print that file to screen.

Note: This will *not* work with Colaboratory or Jupyter notebooks unless you write the code for `mio.py` and your test code to Python files and then execute your file using the `!` prefix. The code notebook has a cell that will write `mio.py` to a Python file.

13.7 Handling structured binary data with the struct module

Generally speaking, when working with your own files, you probably don’t want to read or write binary data in Python. For very simple storage needs, it’s usually best to use text or bytes input and output. For more sophisticated applications, Python provides the ability to easily read or write arbitrary Python objects (*pickling*, described in section

13.8). This ability is much less error prone than directly writing and reading your own binary data and is highly recommended.

But there's at least one situation in which you'll likely need to know how to read or write binary data: when you're dealing with files that are generated or used by other programs. This section describes how to do this by using the `struct` module. Refer to the Python reference documentation for more details.

As you've seen, Python supports explicit binary input or output by using `bytes` instead of strings if you open the file in binary mode. But because most binary files rely on a particular structure to help parse the values, writing your own code to read and split them into variables correctly is often more work than it's worth. Instead, you can use the standard `struct` module to permit you to treat those strings as formatted byte sequences with some specific meaning.

Assume that you want to write and read in a binary file called `data`, containing a series of records generated by a C program. Each record consists of a C short integer, a C double float, and a sequence of four characters that should be taken as a four-character string. You store this data in a Python list of tuples, with each tuple containing an integer, a floating-point number, and a string.

The first thing to do is define a *format string* understandable to the `struct` module, which tells the module how the data in one of your records is packed. The format string uses characters meaningful to `struct` to indicate what type of data is expected where in a record. The character '`h`', for example, indicates the presence of a single C short integer, and the character '`d`' indicates the presence of a single C double-precision floating-point number. Not surprisingly, '`s`' indicates the presence of a string. Any of these may be preceded by an integer to indicate the number of values; in this case, '`7s`' indicates a string consisting of seven characters. For your records, the appropriate format string is therefore '`hd7s`'. `struct` understands a wide range of numeric, character, and string formats. See the documentation for the Python standard library for details.

As you may already have guessed, `struct` provides the ability to take Python values and convert them to packed byte sequences. This conversion is accomplished through the `struct.pack` function, which takes a format string as its first argument and then enough additional arguments to satisfy the format string. To produce a binary record and store it to a file "data," you might do something like this:

```
import struct
record_format = 'hd7s'
data_record = struct.pack(record_format, 42, 3.14, b'goodbye')
data_file = open("data", "wb")
data_file.write(data_record)
data_record

b'*\x00\x00\x00\x00\x00\x00\x00\x00\x1f\x85\xebQ\xb8\x1e\t@goodbye'
```

This will convert and pack the values into the correct binary format.

Before you start reading records from your file, you need to know how many bytes to read at a time. Fortunately, `struct` includes a `calcsize` function, which takes your

format string as an argument and returns the number of bytes used to contain data in such a format.

To read each record, you use the `read` method described earlier in this chapter. Then the `struct.unpack` function conveniently returns a tuple of values by parsing a read record according to your format string. `struct.unpack` is almost the inverse of `struct.pack`. The *almost* is due to the fact that, while `struct.unpack` returns a tuple of Python values, `struct.pack` takes enough separate arguments to satisfy its format string. The program to read your binary data file is remarkably simple:

```
import struct

record_format = 'hd7s'
record_size = struct.calcsize(record_format)
result_list = []
with open("data", 'rb') as input:
    while True:
        record = input.read(record_size)
        if not record:
            break
        result_list.append(struct.unpack(record_format, record))

print(result_list)
[(42, 3.14, b'goodbye')]
```

If the record is empty, you're at the end of the file, so you quit the loop. Note that there's no checking for file consistency; if the last record is an odd size, the `struct.unpack` function raises an error.

`struct` gets even better; you can insert other special characters into the format string to indicate that data should be read/written in big-endian, little-endian, or machine-native-endian format (default is machine-native) and to indicate that things like a C short integer should be sized either as native to the machine (the default) or as standard C sizes. If you need these features, it's nice to know that they exist. See the documentation for the Python standard library for details.

Quick check: `struct`

What use cases can you think of in which the `struct` module would be useful for either reading or writing binary data?

13.8 Pickling objects to files

Python can write any data structure into a file, read that data structure back out of a file, and re-create it with just a few commands. This capability is unusual but can be useful, because it can save you many pages of code that do nothing but dump the state of a program into a file (and can save a similar amount of code that does nothing but read that state back in).

Python provides this capability via the `pickle` module. Pickling is powerful but simple to use. In the words of the Python docs, “‘Pickling’ is the process whereby a Python object hierarchy is converted into a byte stream, and ‘unpickling’ is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy.” So pickling is a way to convert Python objects in a running Python session into a series of bytes that can be stored and/or moved around.

Assume that the entire state of a program is held in three variables: `a`, `b`, and `c`. You can save this state to a file called “state” as follows:

```
import pickle
a = 42
b = 3.14
c = "test"
file = open("state", 'wb')
pickle.dump(a, file)
pickle.dump(b, file)
pickle.dump(c, file)
file.close()
```

It doesn’t matter what was stored in `a`, `b`, and `c`. The content might be as simple as numbers or as complex as a list of dictionaries containing instances of user-defined classes. `pickle.dump` saves everything.

Now, to read that data back in on a later run of the program, just write

```
import pickle
file = open("state", 'rb')
a = pickle.load(file)
b = pickle.load(file)
c = pickle.load(file)
file.close()
print(f"{a=} {b=} {c=}")

a=42 b=3.14 c='test'
```

Any data that was previously in the variables `a`, `b`, or `c` is restored to them by `pickle.load`.

The `pickle` module can store almost anything in this manner. It can handle lists, tuples, numbers, strings, dictionaries, and just about anything made up of these types of objects, which includes all class instances. It also handles shared objects, cyclic references, and other complex memory structures correctly, storing shared objects only once and restoring them as shared objects, not as identical copies. But code objects (what Python uses to store byte-compiled code) and system resources (like files or sockets) can’t be pickled.

More often than not, you won’t want to save your entire program state with `pickle`. Most applications can have multiple documents open at one time, for example. If you saved the entire state of the program, you would effectively save all open documents in one file. An easy and effective way of saving and restoring only data of interest is to write a `save` function that stores all data you want to save into a dictionary and then uses