Effective SOFTWARE DEVELOPMENT SERIES Scott Meyers, Consulting Editor

Effective Python

125 Specific Ways to Write Better Python

HIRD EDITION

Brett Slatkin

FREE SAMPLE CHAPTER | 🕧 💟 🗓

P

Effective Python

Third Edition

This page intentionally left blank

Effective Python

125 SPECIFIC WAYS TO WRITE BETTER PYTHON

Third Edition

Brett Slatkin

♣Addison-Wesley

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned. com or (800) 382-3419. Please contact us with concerns about any potential bias at pearson.com/report-bias.html.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2024945552

Copyright © 2025 Pearson Education, Inc.

Hoboken, NJ

Cover image: Victoria Moloman/Shutterstock

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

ISBN-13: 978-0-13-817218-3 ISBN-10: 0-13-817218-8 \$PrintCode To our family

This page intentionally left blank

Contents at a Glance

Preface	xvii
Acknowledgments	xxiii
About the Author	XXV
Chapter 1: Pythonic Thinking	1
Chapter 2: Strings and Slicing	41
Chapter 3: Loops and Iterators	77
Chapter 4: Dictionaries	109
Chapter 5: Functions	135
Chapter 6: Comprehensions and Generators	173
Chapter 7: Classes and Interfaces	201
Chapter 8: Metaclasses and Attributes	265
Chapter 9: Concurrency and Parallelism	319
Chapter 10: Robustness	399
Chapter 11: Performance	447

Chapter 12: Data Structures and Algorithms	493
Chapter 13: Testing and Debugging	533
Chapter 14: Collaboration	575
Index	627

Contents

Preface		xvii
Acknowl	edgments	xxiii
About th	About the Author	
Chapter	1 Pythonic Thinking	1
Item 1:	Know Which Version of Python You're Using	1
Item 2:	Follow the PEP 8 Style Guide	3
Item 3:	Never Expect Python to Detect Errors at Compile Time	6
Item 4:	Write Helper Functions Instead of Complex Expressions	8
Item 5:	Prefer Multiple-Assignment Unpacking over Indexing	g 11
Item 6:	Always Surround Single-Element Tuples with Parentheses	16
Item 7:	Consider Conditional Expressions for Simple Inline Logic	19
Item 8:	Prevent Repetition with Assignment Expressions	24
Item 9:	Consider match for Destructuring in Flow Control; Avoid When if Statements Are Sufficient	30
Chapter :	2 Strings and Slicing	41
Item 10:	Know the Differences Between bytes and str	41
Item 11:	Prefer Interpolated F-Strings over C-Style Format Strings and str.format	47
Item 12:	Understand the Difference Between repr and str when Printing Objects	58

c	Contents
-	0011001100

	Item	13:	Prefer Explicit String Concatenation over Implicit, Especially in Lists	62
	Item	14:	Know How to Slice Sequences	67
	Item	15:	Avoid Striding and Slicing in a Single Expression	70
	Item	16:	Prefer Catch-All Unpacking over Slicing	72
C	hapt	er	3 Loops and Iterators	77
	Item	17:	Prefer enumerate over range	77
	Item	18:	Use zip to Process Iterators in Parallel	79
	Item	19:	Avoid else Blocks After for and while Loops	82
	Item	20:	Never Use for Loop Variables After the Loop Ends	85
	Item	21:	Be Defensive when Iterating over Arguments	87
	Item	22:	Never Modify Containers While Iterating over Them; Use Copies or Caches Instead	92
	Item	23:	Pass Iterators to any and all for Efficient Short-Circuiting Logic	98
	Item	24:	Consider itertools for Working with Iterators and Generators	102
C	hapt	er	4 Dictionaries	109
	Item	25:	Be Cautious when Relying on Dictionary Insertion Ordering	109
	Item	26:	Prefer get over in and KeyError to	
			Handle Missing Dictionary Keys	117
	Item	27:	Handle Missing Dictionary Keys Prefer defaultdict over setdefault to Handle Missing Items in Internal State	117 122
	Item Item	27: 28:	Handle Missing Dictionary Keys Prefer defaultdict over setdefault to Handle Missing Items in Internal State Know How to Construct Key-Dependent Default Values withmissing	117 122 124
	Item Item Item	27: 28: 29:	Handle Missing Dictionary Keys Prefer defaultdict over setdefault to Handle Missing Items in Internal State Know How to Construct Key-Dependent Default Values withmissing Compose Classes Instead of Deeply Nesting Dictionaries, Lists, and Tuples	117 122 124 127
C	Item Item Item hapt	27: 28: 29:	Handle Missing Dictionary Keys Prefer defaultdict over setdefault to Handle Missing Items in Internal State Know How to Construct Key-Dependent Default Values withmissing Compose Classes Instead of Deeply Nesting Dictionaries, Lists, and Tuples 5 Functions	117 122 124 127 135
C	Item Item Item hapt Item	27: 28: 29: er 30:	 Handle Missing Dictionary Keys Prefer defaultdict over setdefault to Handle Missing Items in Internal State Know How to Construct Key-Dependent Default Values withmissing Compose Classes Instead of Deeply Nesting Dictionaries, Lists, and Tuples 5 Functions Know That Function Arguments Can Be Mutated 	117 122 124 127 135 135
С	Item Item Item hapt Item Item	27: 28: 29: er 30: 31:	 Handle Missing Dictionary Keys Prefer defaultdict over setdefault to Handle Missing Items in Internal State Know How to Construct Key-Dependent Default Values withmissing Compose Classes Instead of Deeply Nesting Dictionaries, Lists, and Tuples 5 Functions Know That Function Arguments Can Be Mutated Return Dedicated Result Objects Instead of Requiring Function Callers to Unpack 	 117 122 124 127 135 135
С	Item Item Item hapt Item Item	27: 28: 29: er 30: 31:	 Handle Missing Dictionary Keys Prefer defaultdict over setdefault to Handle Missing Items in Internal State Know How to Construct Key-Dependent Default Values withmissing Compose Classes Instead of Deeply Nesting Dictionaries, Lists, and Tuples 5 Functions Know That Function Arguments Can Be Mutated Return Dedicated Result Objects Instead of Requiring Function Callers to Unpack More Than Three Variables 	117 122 124 127 135 135
С	Item Item Item Item Item	27: 28: 29: 30: 31: 32:	 Handle Missing Dictionary Keys Prefer defaultdict over setdefault to Handle Missing Items in Internal State Know How to Construct Key-Dependent Default Values withmissing Compose Classes Instead of Deeply Nesting Dictionaries, Lists, and Tuples 5 Functions Know That Function Arguments Can Be Mutated Return Dedicated Result Objects Instead of Requiring Function Callers to Unpack More Than Three Variables Prefer Raising Exceptions to Returning None 	 117 122 124 127 135 138 142

x

Item 34:	Reduce Visual Noise with Variable Positional Arguments	150
Item 35:	Provide Optional Behavior with Keyword Arguments	153
Item 36:	Use None and Docstrings to Specify Dynamic Default Arguments	157
Item 37:	Enforce Clarity with Keyword-Only and Positional-Only Arguments	161
Item 38:	Define Function Decorators with functools.wraps	166
Item 39:	Prefer functools.partial over lambda Expressions for Glue Functions	169
Chapter (6 Comprehensions and Generators	173
Item 40:	Use Comprehensions Instead of map and filter	173
Item 41:	Avoid More Than Two Control Subexpressions in Comprehensions	176
Item 42:	Reduce Repetition in Comprehensions with Assignment Expressions	178
Item 43:	Consider Generators Instead of Returning Lists	182
Item 44:	Consider Generator Expressions for Large List Comprehensions	184
Item 45:	Compose Multiple Generators with yield from	186
Item 46:	Pass Iterators into Generators as Arguments Instead of Calling the send Method	188
Item 47:	Manage Iterative State Transitions with a Class Instead of the Generator throw Method	195
Chapter 7	7 Classes and Interfaces	201
Item 48:	Accept Functions Instead of Classes for Simple Interfaces	201
Item 49:	Prefer Object-Oriented Polymorphism over Functions with isinstance Checks	205
Item 50:	Consider functools.singledispatch for Functional-Style Programming Instead of Object-Oriented Polymorphism	210
Item 51:	Prefer dataclasses for Defining Lightweight Classes	217
Item 52:	Use @classmethod Polymorphism to Construct Objects Generically	230
Item 53:	Initialize Parent Classes with super	235

	Item 54	Consider Composing Functionality with Mix-in Classes	240
	Item 55	Prefer Public Attributes over Private Ones	245
	Item 56	Prefer dataclasses for Creating Immutable Objects	250
	Item 57	Inherit from collections.abc Classes for Custom Container Types	260
С	hapter	8 Metaclasses and Attributes	265
	Item 58	Use Plain Attributes Instead of Setter and Getter Methods	265
	Item 59	Consider @property Instead of Refactoring Attributes	270
	Item 60	Use Descriptors for Reusable @property Methods	274
	Item 61	Usegetattr,getattribute, andsetattr for Lazy Attributes	279
	Item 62	Validate Subclasses withinit_subclass	285
	Item 63	Register Class Existence withinit_subclass	293
	Item 64	Annotate Class Attributes withset_name	299
	Item 65	Consider Class Body Definition Order to Establish Relationships Between Attributes	303
	Item 66	Prefer Class Decorators over Metaclasses for Composable Class Extensions	310
С	hapter	9 Concurrency and Parallelism	319
	Item 67	Use subprocess to Manage Child Processes	320
	Item 68	Use Threads for Blocking I/O; Avoid for Parallelism	324
	Item 69	Use Lock to Prevent Data Races in Threads	330
	Item 70	Use Queue to Coordinate Work Between Threads	333
	Item 71:	Know How to Recognize When Concurrency Is Necessary	344
	Item 72:	Avoid Creating New Thread Instances for On-Demand Fan-out	349
	Item 73	Understand How Using Queue for Concurrency Requires Refactoring	353
	Item 74	Consider ThreadPoolExecutor When Threads	
		Are Necessary for Concurrency	361
	Item 75	Achieve Highly Concurrent I/O with Coroutines	364
	Item 76	Know How to Port Threaded I/O to asyncio	368

Item 77:	Mix Threads and Coroutines to Ease the Transition to asyncio	381
Item 78:	Maximize Responsiveness of asyncio Event Loops with async-Friendly Worker Threads	389
Item 79:	Consider concurrent.futures for True Parallelism	393
Chapter	10 Robustness	399
Item 80:	Take Advantage of Each Block in try/except/else/finally	399
Item 81:	assert Internal Assumptions and raise Missed Expectations	404
Item 82:	Consider contextlib and with Statements for Reusable try/finally Behavior	408
Item 83:	Always Make try Blocks as Short as Possible	412
Item 84:	Beware of Exception Variables Disappearing	414
Item 85:	Beware of Catching the Exception Class	416
Item 86:	Understand the Difference Between Exception and BaseException	419
Item 87:	Use traceback for Enhanced Exception Reporting	424
Item 88:	Consider Explicitly Chaining Exceptions to Clarify Tracebacks	428
Item 89:	Always Pass Resources into Generators and Have Callers Clean Them Up Outside	436
Item 90:	Never Setdebug to False	442
Item 91:	Avoid exec and eval Unless You're Building a Developer Tool	445
Chapter	11 Performance	447
Item 92:	Profile Before Optimizing	448
Item 93:	Optimize Performance-Critical Code Using timeit Microbenchmarks	453
Item 94:	Know When and How to Replace Python with Another Programming Language	458
Item 95:	Consider ctypes to Rapidly Integrate with Native Libraries	462
Item 96:	Consider Extension Modules to Maximize Performance and Ergonomics	467
Item 97:	Rely on Precompiled Bytecode and File System Caching to Improve Startup Time	475

Item 98:	Lazy-Load Modules with Dynamic Imports to Reduce Startup Time	478
Item 99:	Consider memoryview and bytearray for Zero-Copy Interactions with bytes	485
Chapter	12 Data Structures and Algorithms	493
Item 100	: Sort by Complex Criteria Using the key Parameter	493
Item 101:	Know the Difference Between sort and sorted	499
Item 102:	Consider Searching Sorted Sequences with bisect	501
Item 103:	Prefer deque for Producer–Consumer Queues	504
Item 104	Know How to Use heapq for Priority Queues	509
Item 105	Use datetime Instead of time for Local Clocks	519
Item 106	Use decimal when Precision Is Paramount	523
Item 107:	Make pickle Serialization Maintainable with copyreg	526
Chapter	13 Testing and Debugging	533
Item 108	Verify Related Behaviors in TestCase Subclasses	533
Item 109	Prefer Integration Tests over Unit Tests	541
Item 110:	Isolate Tests from Each Other with setUp, tearDown, setUpModule, and tearDownModule	547
Item 111:	Use Mocks to Test Code with Complex Dependencies	550
Item 112:	Encapsulate Dependencies to Facilitate Mocking and Testing	559
Item 113:	Use assertAlmostEqual to Control Precision in Floating Point Tests	563
Item 114:	Consider Interactive Debugging with pdb	565
Item 115:	Use tracemalloc to Understand Memory Usage and Leaks	570
Chapter	14 Collaboration	575
Item 116:	Know Where to Find Community-Built Modules	575
Item 117:	Use Virtual Environments for Isolated and Reproducible Dependencies	576
Item 118:	Write Docstrings for Every Function, Class, and Module	582

Iı	ndex		627
	Item 125:	Prefer Open Source Projects for Bundling Python Programs over zipimport and zipapp	621
	Item 124:	Consider Static Analysis via typing to Obviate Bugs	613
	Item 123:	Consider warnings to Refactor and Migrate Usage	605
	Item 122:	Know How to Break Circular Dependencies	600
	Item 121:	Define a Root Exception to Insulate Callers from APIs	595
	Item 120:	Consider Module-Scoped Code to Configure Deployment Environments	593
	Item 119:	Use Packages to Organize Modules and Provide Stable APIs	588

This page intentionally left blank

Preface

The Python programming language has unique strengths and charms that can be hard to grasp. Many programmers familiar with other languages approach Python from a limited mindset instead of embracing its full capabilities. Some programmers go too far in the other direction, overusing Python features that can cause big problems later.

This book provides insight into the *Pythonic* way of writing programs: the best way to use Python. It builds on a fundamental understanding of the language that I assume you already have. Novice programmers will learn the best practices of Python's critical features. Experienced programmers will learn how to embrace a new tool with confidence.

With this book I hope to help you use Python to accomplish your goals, whatever they may be, or at least to help you have more fun on your journey with programming.

What This Book Covers

Each chapter in this book contains a broad but related set of items. Feel free to jump between items and follow your interest. Each item contains concise and specific guidance explaining how you can write Python programs more effectively. Items include advice on what to do, what to avoid, how to strike the right balance, and why this is the best choice. Items reference each other to make it easier to fill in the gaps as you read.

This third edition covers the language up through Python version 3.13 (see Item 1: "Know Which Version of Python You're Using"). This book includes 35 completely new items compared to the second edition. Most of the items from the second edition have been revised and included, but many have undergone substantial updates. For some items, my advice has completely changed due to best practices evolving as Python has matured over the past five years.

Python takes a "batteries included" approach to its standard library. Many of these built-in packages are so closely intertwined with idiomatic Python that they may as well be part of the language specification. The full set of standard modules is too large to cover in this book, but I've included the ones that I feel are critical to be aware of and use.

Python also has a vibrant ecosystem of community-built modules that extend the language in valuable ways. Although I mention important packages to know about in various items, this book is not intended to be a thorough reference. Similarly, despite the importance of Python package management, I avoid going into the details about it because it's rapidly changing and evolving.

Chapter 1: Pythonic Thinking

The Python community has come to use the adjective *Pythonic* to describe code that follows a particular style. The idioms of Python have emerged over time through experience using the language and collaborating with other programmers. This chapter covers the best ways to do the most common things in Python.

Chapter 2: Strings and Slicing

Python has built-in syntax, methods, and modules for string and sequence processing. These capabilities are so essential that you'll see them in nearly every program. They make Python an excellent language for parsing text, inspecting data formats, and interfacing with the low-level binary representations used by computers.

Chapter 3: Loops and Iterators

Processing through sequential data is a critical need in programs. Loops in Python feel natural and capable for the most common tasks involving built-in data types, container types, and user-defined classes. Python also supports iterators, which enable a more functional-style approach to processing arbitrary streams of data with significant benefits.

Chapter 4: Dictionaries

Python's built-in dictionary type is a versatile data structure for bookkeeping in programs. Compared to simple lists, dictionaries provide much better performance for adding and removing items. Python also has special syntax and related built-in modules that enhance dictionaries beyond what you might expect from hash tables in other languages.

Chapter 5: Functions

Functions in Python have a variety of extra features that can make a programmer's life easier. Some are similar to capabilities in other programming languages, but many are unique to Python. This chapter covers how to use functions to clarify intention, promote reuse, and reduce bugs.

Chapter 6: Comprehensions and Generators

Python has special syntax for quickly iterating through lists, dictionaries, and sets to generate derivative data structures. It also allows for a stream of iterable values to be incrementally returned by a function. This chapter covers how these features can provide better performance, reduced memory usage, and improved readability.

Chapter 7: Classes and Interfaces

Python is an object-oriented language. Getting things done in Python often requires writing new classes and defining how they interact through their interfaces and hierarchies. This chapter covers how to use classes to express intended behaviors with objects.

Chapter 8: Metaclasses and Attributes

Metaclasses and dynamic attributes are powerful Python features. However, they also enable you to implement extremely bizarre and unexpected behaviors. This chapter covers the common idioms for using these mechanisms to ensure that you follow the *rule of least surprise*.

Chapter 9: Concurrency and Parallelism

With features such as threads and asynchronous coroutines, Python makes it easy to write concurrent programs that do many different things seemingly at the same time. Python can also be used to do parallel work through system calls, subprocesses, and special modules. This chapter covers how to best utilize Python in these subtly different situations.

Chapter 10: Robustness

Making programs dependable when they encounter unexpected circumstances is just as important as making programs with correct functionality. Python has built-in features and modules that aid in hardening your programs so they are robust in a wide variety of situations.

Chapter 11: Performance

Python includes a variety of capabilities that enable programs to achieve surprisingly impressive performance with relatively low amounts of effort. Using these features, it's possible to extract maximum performance from a host system while retaining the productivity gains afforded by Python's high-level nature.

Chapter 12: Data Structures and Algorithms

Python includes optimized implementations of many standard data structures and algorithms that can help you achieve high performance with minimal effort. The language also provides battle-tested data types and helper functions for common tasks (e.g., working with currency and time) that allow you focus on your program's core requirements.

Chapter 13: Testing and Debugging

You should always test your code, regardless of what language it's written in, but with Python, testing is especially important. Python's dynamic features can increase the risk of runtime errors in unique ways. Luckily, they also make it easier to write tests and diagnose malfunctioning programs. This chapter covers Python's built-in tools for testing and debugging.

Chapter 14: Collaboration

Collaborating on Python programs requires you to be deliberate about how you write your code. Even if you're working alone, you'll want to understand how to use modules written by others. This chapter covers the standard tools and best practices that enable people to work together on Python programs.

Conventions Used in This Book

Python code snippets in this book are in monospace font and have syntax highlighting. When lines are long, I use \blacktriangleright characters to show when they wrap. I truncate some snippets with ellipses (...) to indicate regions where code exists that isn't essential for expressing the point. You'll need to download the full example code (see below on where to get it) in order to get these truncated snippets to run correctly on your computer.

I take some artistic license with the Python style guide in order to make the code examples better fit the format of a book or to highlight the most important parts. I've also left out embedded documentation to reduce the size of code examples. I strongly suggest that you don't emulate this in your projects; instead, you should follow the style guide (see Item 2: "Follow the PEP 8 Style Guide") and write documentation (see Item 118: "Write Docstrings for Every Function, Class, and Module").

Most code snippets in this book are accompanied by the corresponding output from running the code. When I say "output," I mean console or terminal output: what you see when running the Python program in an interactive interpreter. Output sections are in monospace font, and each is preceded by a >>> line (the Python interactive prompt). The idea is that you could type the code snippets into a Python shell and reproduce the expected output.

Finally, there are some other sections in monospace font that are not preceded by a >>> line. These represent the output of running programs besides the Python interpreter. These examples often begin with \$ characters to indicate that I'm running programs from a command-line shell like Bash. If you're running these commands on Windows or another type of system, you may need to adjust the program names and arguments accordingly.

Where to Get the Code and Errata

It's useful to view many of the examples from this book as whole programs without interleaved prose. This also gives you a chance to tinker with the code yourself and understand why the program works as described. You can find the source code for all code snippets in this book on the book's website, https://effectivepython.com. The website also provides instructions on how to report errors. Thank you in advance for contacting me about any errors you find.

Register your copy of *Effective Python:* 125 Specific Ways to Write Better Python, 3rd Edition on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/ register and log in or create an account. Enter the product ISBN (9780138172183) and click Submit. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us. This page intentionally left blank

Acknowledgments

Thank you for reading this book. I must emphasize that this book would not have been possible without guidance, support, and encouragement from many people.

Thanks to Scott Meyers for the Effective Software Development series of books. I discovered the joy of computer programming at a young age, but when I read his book *Effective C++* when I was 15 years old, something clicked. There's no doubt that Scott's books led to my college education and first job. I'm thrilled to have had the opportunity to write all three editions of *Effective Python*. I've learned so much in the process, and I'm deeply grateful for the experience.

Thanks to the team who made this third edition a reality. Thanks to my executive editor, Debra Williams, for being so supportive throughout the process. Thanks to development editor Chris Zahn, production editor Mary Roth, copy editor Kitty Wilson, cover designer Chuti Prasertsith, and marketing manager Chike Lawrence-Mitchell. Thanks to my technical reviewers—Karry Lu, David N. Cohron, and Andy Chu for the depth and thoroughness of their feedback.

Thanks to everyone who supported me in creating the first and second editions of this book: Debra Williams, Trina MacDonald, Olivia Basegio, Mike Bayer, Titus Brown, Brett Cannon, Andy Chu, Tom Cirtin, Nick Cohron, Leah Culver, Andrew Dolan, Pamela Fox, Stephanie Geels, Adrian Holovaty, Toshiaki Kurokawa, Michael Levine, Lori Lyons, Asher Mancinelli, Wes McKinney, Julie Nahil, Stephane Nakib, Stephane Nakib, Marzia Niccolai, Ade Oshineye, Chuti Prasertsith, Brandon Rhodes, Tavis Rudd, Katrina Sostek, Mike Taylor, Simon Willison, Kitty Wilson, and Chris Zahn.

Thanks to all of the readers who reported errors and room for improvement in the book. Please keep the feedback coming! Thanks to all of the translators who made the book available around the world; nothing brings a smile to my face quite like seeing my book in other languages. Thanks to the wonderful Python programmers I've known and worked with: Anthony Baxter, Brett Cannon, Wesley Chun, Jeremy Hylton, Alex Martelli, Neal Norwitz, Guido van Rossum, Andy Smith, Greg Stein, Ka-Ping Yee, and Gregory Smith. I appreciate your tutelage and leadership. Python has an excellent community, and I feel lucky to be a part of it.

Thanks to my teammates over the years for letting me be the worst player in the band. Thanks to Kevin Gibbs for helping me take risks. Thanks to Ken Ashcraft, Ryan Barrett, and Jon McAlister for showing me how it's done. Thanks to Brad Fitzpatrick for taking it to the next level. Thanks to Paul McDonald for being an amazing co-founder. Thanks to Jeremy Ginsberg, Jack Hebert, John Skidgel, Evan Martin, Tony Chang, Troy Trimble, Tessa Pupius, Erick Armbrust, and Dylan Lorimer for helping me learn. Thanks to Sagnik Nandy, Waleed Ojeil, and Will Grannis for your mentorship.

Thanks to the inspiring programming and engineering teachers that I've had: Ben Chelf, Glenn Cowan, Vince Hugo, Russ Lewin, Jon Stemmle, Derek Thomson, Daniel Wang, Dean Nevins, Stephen Strenn, and Alex Guy. Without your instruction, I would never have pursued our craft or gained the perspective required to teach others.

Thanks to my mother for giving me a sense of purpose and encouraging me to become a programmer. Thanks to my family and friends for their support. Thanks to my wife for her love and friendship.

About the Author

Brett Slatkin has been programming with Python professionally for the past 19 years. He currently works as a principal software engineer in the Office of the CTO at Google, developing technology strategies and rapid prototypes.

His prior experience includes founding Google Surveys, an internal startup for collecting machine learning and market research data sets; launching Google App Engine, the company's first cloud computing product; scaling Google's A/B experimentation products to billions of users; co-creating PubSubHubbub, the W3C standard for real-time RSS feeds; and making various contributions to open source projects.

Brett earned a bachelor's degree in computer engineering from Columbia University in the City of New York. Outside of his day job, he enjoys playing piano, surfing, and spending time with his family. He lives in California. You can find him online at https://onebigfluke.com. This page intentionally left blank

Functions



The first organizational tool programmers use in Python is the *function*. As in other programming languages, functions enable you to break large programs into smaller, simpler components with names to represent their purpose. They improve readability and make code more approachable. They allow for reuse and refactoring.

Functions in Python have a variety of extra features that make a programmer's life easier. Some are similar to capabilities in other programming languages, but many are unique to Python. These extras can make a function's interface clearer. They can eliminate noise and reinforce the intention of callers. They can significantly reduce subtle bugs that are difficult to find.

Item 30: Know That Function Arguments Can Be Mutated

Python doesn't support pointer types (beyond interfacing with C; see Item 95: "Consider ctypes to Rapidly Integrate with Native Libraries"). But arguments passed to functions are all passed by reference. For simple types, like integers and strings, parameters appear to be passed by value because they're immutable objects. But more complex objects can be modified whenever they're passed to other functions, regardless of the caller's intent.

For example, if I pass a list to another function, that function has the ability to call mutation methods on the argument:

```
def my_func(items):
    items.append(4)
x = [1, 2, 3]
my_func(x)
print(x) # 4 is now in the list
```

>>> [1, 2, 3, 4]

In this case, you can't replace the original value of the variable x within the called function, as you might do with a C-style pointer type. But you can make modifications to the list assigned to x.

Similarly, when one variable is assigned to another, it stores a reference, or an *alias*, to the same underlying data structure. Thus, calling a function with what appears to be a separate variable actually allows for mutation of the original:

For lists and dictionaries, you can work around this issue by passing a copy of the container to insulate you from the function's behavior. Here, I create a copy by using the slice operation with no starting or ending indexes (see Item 14: "Know How to Slice Sequences"):

```
def capitalize_items(items):
    for i in range(len(items)):
        items[i] = items[i].capitalize()

my_items = ["hello", "world"]
items_copy = my_items[:] # Creates a copy
capitalize_items(items_copy)
print(items_copy)
>>>
```

['Hello', 'World']

The dictionary built-in type provides a copy method specifically for this purpose:

```
def concat_pairs(items):
    for key in items:
        items[key] = f"{key}={items[key]}"
my_pairs = {"foo": 1, "bar": 2}
pairs_copy = my_pairs.copy() # Creates a copy
concat_pairs(pairs_copy)
print(pairs_copy)
>>>
{'foo': 'foo=1', 'bar': 'bar=2'}
```

User-defined classes (see Item 29: "Compose Classes Instead of Deeply Nesting Dictionaries, Lists, and Tuples") can also be modified by callers. Any of their internal properties can be accessed or assigned by any function they're passed to (see Item 55: "Prefer Public Attributes over Private Ones"):

```
class MyClass:
    def __init__(self, value):
        self.value = value
x = MyClass(10)
def my_func(obj):
    obj.value = 20  # Modifies the object
my_func(x)
print(x.value)
>>>
20
```

When implementing a function that others will call, you shouldn't modify any mutable value provided unless that behavior is mentioned explicitly in the function name, argument names, or documentation. You might also want to make a defensive copy of any arguments you receive to avoid various pitfalls with iteration (see Item 21: "Be Defensive when Iterating over Arguments" and Item 22: "Never Modify Containers While Iterating over Them; Use Copies or Caches Instead").

When calling a function, you should be careful about passing mutable arguments because your data might get modified, which can cause difficult-to-spot bugs. For complex objects you control, it can be useful to add helper functions and methods that make it easy to create defensive copies. Alternatively, you can use a more functional style and try to leverage immutable objects and pure functions (see Item 56: "Prefer dataclasses for Creating Immutable Objects").

Things to Remember

- Arguments in Python are passed by reference, meaning their attributes can be mutated by receiving functions and methods.
- Functions should make it clear (with naming and documentation) when they will modify input arguments and avoid modifying arguments otherwise.
- Creating copies of collections and objects you receive as input is a reliable way to ensure that your functions avoid inadvertently modifying data.

Item 31: Return Dedicated Result Objects Instead of Requiring Function Callers to Unpack More Than Three Variables

One effect of the unpacking syntax (see Item 5: "Prefer Multiple-Assignment Unpacking over Indexing") is that it allows a Python function to seemingly return more than one value. For example, say that I'm trying to determine various statistics for a population of alligators. Given a list of lengths, I need to calculate the minimum and maximum lengths in the population. Here, I do this in a single function that appears to return two values:

```
def get_stats(numbers):
    minimum = min(numbers)
    maximum = max(numbers)
    return minimum, maximum
lengths = [63, 73, 72, 60, 67, 66, 71, 61, 72, 70]
minimum, maximum = get_stats(lengths)  # Two return values
print(f"Min: {minimum}, Max: {maximum}")
>>>
Min: 60, Max: 73
```

The way this works is that multiple values are returned together in a two-item tuple. The calling code then unpacks the returned tuple by assigning two variables. Here, I use an even simpler example to show how an unpacking statement and multiple-return function work the same way:

```
first, second = 1, 2
assert first == 1
assert second == 2

def my_function():
    return 1, 2

first, second = my_function()
assert first == 1
assert second == 2
```

Multiple return values can also be received by starred expressions for catch-all unpacking (see Item 16: "Prefer Catch-All Unpacking over Slicing"). For example, say I need another function that calculates

how big each alligator is relative to the population average. This function returns a list of ratios, but I can receive the longest and shortest items individually by using a starred expression for the middle portion of the list:

```
def get_avg_ratio(numbers):
    average = sum(numbers) / len(numbers)
    scaled = [x / average for x in numbers]
    scaled.sort(reverse=True)
    return scaled
longest, *middle, shortest = get_avg_ratio(lengths)
print(f"Longest: {longest:>4.0%}")
print(f"Shortest: {shortest:>4.0%}")
>>>
Longest: 108%
Shortest: 89%
```

Now, imagine that the program's requirements change, and I need to also determine the average length, median length, and total population size of the alligators. I can do this by expanding the get_stats function to also calculate these statistics and return them in the result tuple that is unpacked by the caller:

```
def get_median(numbers):
   count = len(numbers)
    sorted_numbers = sorted(numbers)
   middle = count // 2
    if count \% 2 == 0:
        lower = sorted_numbers[middle - 1]
        upper = sorted_numbers[middle]
        median = (lower + upper) / 2
    else:
        median = sorted_numbers[middle]
    return median
def get_stats_more(numbers):
   minimum = min(numbers)
   maximum = max(numbers)
    count = len(numbers)
    average = sum(numbers) / count
   median = get_median(numbers)
    return minimum, maximum, average, median, count
```

There are two problems with this code. First, all of the return values are numeric, so it is all too easy to reorder them accidentally (e.g., swapping average and median), which can cause bugs that are hard to spot later. Using a large number of return values is extremely error prone:

```
minimum, maximum, average, median, count =
    get_stats_more(lengths)
```

```
(minimum, maximum, average,
 median, count) = get_stats_more(lengths)
```

```
(minimum, maximum, average, median, count
) = get_stats_more(lengths)
```

To avoid these problems, you should never use more than three variables when unpacking the multiple return values from a function. These could be individual values from a three-tuple, two variables and one catch-all starred expression, or anything shorter.

If you need to unpack more return values than that, you're better off defining a lightweight class (see Item 29: "Compose Classes Instead

of Deeply Nesting Dictionaries, Lists, and Tuples" and Item 51: "Prefer dataclasses for Defining Lightweight Classes") and having your function return an instance of that instead. Here, I write another version of the get_stats function that returns a result object instead of a tuple:

```
from dataclasses import dataclass
@dataclass
class Stats:
    minimum: float
    maximum: float
    average: float
    median: float
    count: int
def get_stats_obj(numbers):
    return Stats(
        minimum=min(numbers),
        maximum=max(numbers).
        count=len(numbers),
        average=sum(numbers) / count,
        median=get_median(numbers),
    )
result = get_stats_obj(lengths)
print(result)
>>>
Stats(minimum=60, maximum=73, average=67.5, median=68.5,
\Rightarrowcount=10)
```

The code is clearer, less error prone, and will be easier to refactor later.

Things to Remember

- You can have functions return multiple values by putting them in a tuple and having the caller take advantage of Python's unpacking syntax.
- Multiple return values from a function can also be unpacked by catch-all starred expressions.
- Unpacking into four or more variables is error prone and should be avoided; instead, return an instance of a lightweight class.

Item 32: Prefer Raising Exceptions to Returning None

When writing utility functions, there's a draw for Python programmers to give special meaning to the return value None. It seems to make sense in some cases (see Item 26: "Prefer get over in and KeyError to Handle Missing Dictionary Keys"). For example, say I want a helper function that divides one number by another. In the case of dividing by zero, returning None seems natural because the result is undefined:

```
def careful_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return None
```

Code that uses this function can interpret the return value accordingly:

```
x, y = 1, 0
result = careful_divide(x, y)
if result is None:
    print("Invalid inputs")
```

What happens with the careful_divide function when the numerator is zero? If the denominator is not zero, then the function returns zero. The problem is that a zero return value can cause issues when you evaluate the result in a condition like an if statement. You might accidentally look for any falsey value to indicate errors instead of only looking for None (see Item 4: "Write Helper Functions Instead of Complex Expressions" and Item 7: "Consider Conditional Expressions for Simple Inline Logic"):

This misinterpretation of a False-equivalent return value is a common mistake in Python code when None has special meaning. This is why returning None from a function like careful_divide is error prone. There are two ways to reduce the chance of such errors. The first way is to split the return value into a two-tuple (see Item 31: "Return Dedicated Result Objects Instead of Requiring Function Callers to Unpack More Than Three Variables" for background). The first part of the tuple indicates that the operation was a success or failure. The second part is the actual result that was computed:

```
def careful_divide(a, b):
    try:
        return True, a / b
    except ZeroDivisionError:
        return False, None
```

Callers of this function have to unpack the tuple. That forces them to consider the status part of the tuple instead of just looking at the result of division:

```
success, result = careful_divide(x, y)
if not success:
    print("Invalid inputs")
```

The problem is that callers can easily ignore the first part of the tuple (using the underscore variable name, which is a Python convention for unused variables). The resulting code doesn't look wrong at first glance, but this can be just as error prone as returning None:

```
_, result = careful_divide(x, y)
if not result:
    print("Invalid inputs")
```

The second, better way to reduce these errors is to never return None for special cases. Instead, raise an exception up to the caller and have the caller deal with it. Here, I turn ZeroDivisionError into ValueError to indicate to the caller that the input values are bad (see Item 88: "Consider Explicitly Chaining Exceptions to Clarify Tracebacks" and Item 121: "Define a Root Exception to Insulate Callers from APIs" for details):

```
def careful_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        raise ValueError("Invalid inputs") # Changed
```

The caller no longer requires a condition on the return value of the function. Instead, it can assume that the return value is always valid and use the results immediately in the else block after try
(see Item 80: "Take Advantage of Each Block in try/except/else/ finally" for background):

```
x, y = 5, 2
try:
    result = careful_divide(x, y)
except ValueError:
    print("Invalid inputs")
else:
    print(f"Result is {result:.1f}")
>>>
Result is 2.5
```

This approach can be extended to code using type annotations (see Item 124: "Consider Static Analysis via typing to Obviate Bugs" for background). You can specify that a function's return value will always be a float and thus will never be None. However, Python's gradual typing purposely doesn't provide a way to indicate when exceptions are part of a function's interface (also known as *checked exceptions*). Instead, you have to document the exception-raising behavior and expect callers to rely on that in order to know which exceptions they should plan to catch (see Item 118: "Write Docstrings for Every Function, Class, and Module").

Pulling it all together, here's what this function should look like when using type annotations and docstrings:

```
def careful_divide(a: float, b: float) -> float:
    """Divides a by b.
    Raises:
        ValueError: When the inputs cannot be divided.
    .....
    try:
        return a / b
    except ZeroDivisionError:
        raise ValueError("Invalid inputs")
try:
    result = careful divide(1, 0)
except ValueError:
    print("Invalid inputs") # Expected
else:
    print(f"Result is {result:.1f}")
>>>
$ python3 -m mypy --strict example.py
Success: no issues found in 1 source file
```

Now the inputs, outputs, and exceptional behavior are all clear, and the chance of a caller doing the wrong thing is extremely low.

Things to Remember

- Functions that return None to indicate special meaning are error prone because None and many other values, such as zero and empty strings, evaluate to False in Boolean expressions.
- Raise exceptions to indicate special situations instead of returning None. Expect the calling code to handle exceptions properly when they're documented.
- Type annotations can be used to make it clear that a function will never return the value None, even in special situations.

Item 33: Know How Closures Interact with Variable Scope and nonlocal

Imagine that I want to sort a list of numbers but prioritize one group of numbers to come first. This pattern is useful when you're rendering a user interface and want important messages or exceptional events to be displayed before everything else. A common way to do this is to pass a helper function as the key argument to a list's sort method (see Item 100: "Sort by Complex Criteria Using the key Parameter" for details). The helper's return value will be used as the value for sorting each item in the list. The helper can check whether the given item is in the important group and can vary the sorting value accordingly:

```
def sort_priority(values, group):
    def helper(x):
        if x in group:
            return (0, x)
            return (1, x)
```

values.sort(key=helper)

This function works for simple inputs:

```
numbers = [8, 3, 1, 2, 5, 4, 7, 6]
group = {2, 3, 5, 7}
sort_priority(numbers, group)
print(numbers)
>>>
[2, 3, 5, 7, 1, 4, 6, 8]
```

There are three reasons this function operates as expected:

- Python supports *closures*—that is, functions that refer to variables from the scope in which they were defined. This is why the helper function is able to access the group argument for the sort_priority function.
- ◆ Functions are *first-class* objects in Python, which means you can refer to them directly, assign them to variables, pass them as arguments to other functions, compare them in expressions and if statements, and so on. This is how the sort method can accept a closure function as the key argument.
- ◆ Python has specific rules for comparing sequences (including tuples). It first compares items at index zero; then, if those are equal, it compares items at index one; if they are still equal, it compares items at index two, and so on. This is why the return value from the helper closure causes the sort order to have two distinct groups.

It'd be nice if this function returned whether higher-priority items were seen at all so the user interface code could act accordingly. Adding such behavior seems straightforward. There's already a closure function for deciding which group each number is in. Why not also use the closure to flip a flag when high-priority items are seen? Then, the function could return the flag value after it's modified by the closure.

Here, I try to do that in a seemingly obvious way:

I can run the function on the same inputs as before:

```
found = sort_priority2(numbers, group)
print("Found:", found)
print(numbers)
```

>>> Found: False [2, 3, 5, 7, 1, 4, 6, 8]

The sorted results are correct, which means items from group were definitely found in numbers. However, the found result returned by the function is False when it should be True. How could this happen?

When you reference a variable in an expression, the Python interpreter traverses the nested scopes to resolve the reference in this order:

- 1. The current function's scope
- 2. Any enclosing scopes (such as other containing functions)
- 3. The scope of the module that contains the code (also called the *global scope*)
- 4. The built-in scope (that contains functions like len and str)

If none of these places has defined a variable with the referenced name, then a NameError exception is raised:

```
foo = does_not_exist * 5
>>>
Traceback ...
NameError: name 'does_not_exist' is not defined
```

Assigning a value to a variable works differently. If the variable is already defined in the current scope, that name will take on the new value in that scope. If the variable doesn't exist in the current scope, Python treats the assignment as a variable definition. Critically, the scope of the newly defined variable is the function that contains the assignment, not an enclosing scope with an earlier assignment.

This assignment behavior explains the wrong return value of the sort_priority2 function. The found variable is assigned to True in the helper closure. The closure's assignment is treated as a new variable definition within the scope of helper, not as an assignment within the scope of sort_priority2:

```
return (0, x)
return (1, x)
numbers.sort(key=helper)
return found
```

This problem is sometimes called the *scoping bug* because it can be so surprising to newbies. But this behavior is the intended result: It prevents local variables in a function from polluting the containing module. Otherwise, every assignment in a function would put garbage into the global module scope. Not only would that be noise, but the interplay of the resulting global variables could cause obscure bugs.

In Python, there is special syntax for assigning data outside of a closure's scope. The nonlocal statement is used to indicate that scope traversal should happen upon assignment for a specific variable name. The only limit is that nonlocal won't traverse up to the module-level scope (to avoid polluting globals).

Here, I define the same function again, now using nonlocal:

```
def sort_priority3(numbers, group):
    found = False

    def helper(x):
        nonlocal found # Added
        if x in group:
            found = True
            return (0, x)
        return (1, x)

    numbers.sort(key=helper)
    return found
Now the found flag works as expected:
```

```
found = sort_priority3(numbers, group)
print("Found:", found)
print(numbers)
>>>
Found: True
[2, 3, 5, 7, 1, 4, 6, 8]
```

The nonlocal statement makes it clear when data is being assigned out of a closure and into another scope. It's complementary to the global statement, which indicates that a variable's assignment should go directly into the module scope. However, much as with the anti-pattern of global variables, I caution against using nonlocal for anything beyond simple functions. The side effects of nonlocal can be hard to follow. It's especially hard to understand in long functions where the nonlocal statements and assignments to associated variables are far apart.

When your usage of nonlocal starts getting complicated, it's better to wrap your state in a helper class. Here, I define a class that can be called like a function; it achieves the same result as the nonlocal approach by assigning an object's attribute during sorting (see Item 55: "Prefer Public Attributes over Private Ones"):

```
class Sorter:
    def __init__(self, group):
        self.group = group
        self.found = False
    def __call__(self, x):
        if x in self.group:
            self.found = True
            return (0, x)
        return (1, x)
```

It's a little longer than before, but it's much easier to reason about and extend if needed (see Item 48: "Accept Functions Instead of Classes for Simple Interfaces" for details on the __call__ special method). I can access the found attribute on the Sorter instance to get the result:

```
sorter = Sorter(group)
numbers.sort(key=sorter)
print("Found:", sorter.found)
print(numbers)
>>>
Found: True
[2, 3, 5, 7, 1, 4, 6, 8]
```

Things to Remember

- Closure functions can refer to variables from any of the enclosing scopes in which they were defined.
- By default, closures can't affect enclosing scopes by assigning variables.

- Use the nonlocal statement to indicate when a closure can modify a variable in its enclosing scopes. Use the global statement to do the same thing for module-level names.
- Avoid using nonlocal statements for anything beyond simple functions.

Item 34: Reduce Visual Noise with Variable Positional Arguments

Accepting a variable number of positional arguments can make a function call clearer and reduce visual noise. These positional arguments are often called *varargs* for short, or *star args*, in reference to the conventional name for the parameter *args. For example, say that I want to log some debugging information. With a fixed number of arguments, I would need a function that takes a message and a list of values:

```
def log(message, values):
    if not values:
        print(message)
    else:
        values_str = ", ".join(str(x) for x in values)
        print(f"{message}: {values_str}")
log("My numbers are", [1, 2])
log("Hi there", [])
>>>
My numbers are: 1, 2
Hi there
```

Having to pass an empty list when I have no values to log is cumbersome and noisy. It'd be better to leave out the second argument entirely. I can do this in Python by prefixing the last positional parameter name with *. The first parameter for the log message is required, and any number of subsequent positional arguments are optional. The function body doesn't need to change; only the callers do:

```
def log(message, *values): # Changed
    if not values:
        print(message)
    else:
        values_str = ", ".join(str(x) for x in values)
        print(f"{message}: {values_str}")
```

```
log("My numbers are", 1, 2)
log("Hi there")  # Changed
>>>
My numbers are: 1, 2
Hi there
```

This syntax works very similarly to the starred expressions used in unpacking assignment statements (see Item 16: "Prefer Catch-All Unpacking over Slicing" and Item 9: "Consider match for Destructuring in Flow Control; Avoid When if Statements Are Sufficient" for more examples).

If I already have a sequence (like a list) and I want to call a variadic function like log, I can do this by using the * operator. This instructs Python to pass items from the sequence as positional arguments to the function:

```
favorites = [7, 33, 99]
log("Favorite colors", *favorites)
>>>
Favorite colors: 7, 33, 99
```

There are two problems with accepting a variable number of positional arguments.

The first issue is that these optional positional arguments are always turned into a tuple before they are passed to your function. This means that if the caller of your function uses the * operator on a generator, it will be iterated until it's exhausted (see Item 43: "Consider Generators Instead of Returning Lists" for background). The resulting tuple includes every value from the generator, which could consume a lot of memory and cause the program to crash:

```
def my_generator():
    for i in range(10):
        yield i

def my_func(*args):
    print(args)

it = my_generator()
my_func(*it)
>>>
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Functions that accept *args are best for situations where you know the number of inputs in the argument list will be reasonably small. *args is ideal for function calls that pass many literals or variable names together. It's primarily for the convenience of the programmer who calls the function and the readability of the calling code.

The second issue with *args is that you can't add new positional arguments to a function in the future without migrating every caller. If you try to add a positional argument in the front of the argument list, existing callers will subtly break if they aren't updated. For example, here I add sequence as the first argument of the function and use it to render the log messages:

```
def log_seq(sequence, message, *values):
    if not values:
        print(f"{sequence} - {message}")
    else:
        values_str = ", ".join(str(x) for x in values)
        print(f"{sequence} - {message}: {values_str}")
log_seq(1, "Favorites", 7, 33)  # New with *args OK
log_seq(1, "Hi there")  # New message only OK
log_seq("Favorite numbers", 7, 33)  # Old usage breaks
>>>
1 - Favorites: 7, 33
1 - Hi there
Favorite numbers - 7: 33
```

The problem with the code above is that the third call to log used 7 as the message parameter because a sequence argument wasn't provided. Bugs like this are hard to track down because the code still runs without raising any exceptions. To avoid this possibility entirely, you should use keyword-only arguments when you want to extend functions that accept *args (see Item 37: "Enforce Clarity with Keyword-Only and Positional-Only Arguments"). To be even more defensive, you could also consider using type annotations (see Item 124: "Consider Static Analysis via typing to Obviate Bugs").

Things to Remember

- You can have functions accept a variable number of positional arguments by using *args in the def statement.
- You can use the items from a sequence as the positional arguments for a function with the * operator.

- Using the * operator with a generator may cause a program to run out of memory and crash.
- Adding new positional arguments to functions that accept *args can introduce hard-to-detect bugs.

Item 35: Provide Optional Behavior with Keyword Arguments

As in most other programming languages, in Python you may pass arguments by position when calling a function:

```
def remainder(number, divisor):
    return number % divisor
```

```
assert remainder(20, 7) = 6
```

All normal arguments to Python functions can also be passed by keyword, where the name of the argument is used in an assignment within the parentheses of a function call. Keyword arguments can be passed in any order, as long as all of the required positional arguments are specified. You can mix and match keyword and positional arguments. These calls are equivalent:

```
remainder(20, 7)
remainder(20, divisor=7)
remainder(number=20, divisor=7)
remainder(divisor=7, number=20)
```

Positional arguments must be specified before keyword arguments:

```
remainder(number=20, 7)
```

```
>>>
Traceback ...
SyntaxError: positional argument follows keyword argument
```

Each argument can be specified only once:

```
remainder(20, number=7)
```

>>>
Traceback ...
TypeError: remainder() got multiple values for argument
w'number'

If you already have a dictionary object, and you want to use its contents to call a function like remainder, you can do this by using the ** operator. This instructs Python to pass the key-value pairs

from the dictionary as the corresponding keyword arguments of the function:

```
my_kwargs = {
    "number": 20,
    "divisor": 7,
}
assert remainder(**my_kwargs) == 6
```

You can mix the ** operator with positional arguments or keyword arguments in the function call as long as no argument is repeated:

```
my_kwargs = {
    "divisor": 7,
}
assert remainder(number=20, **my_kwargs) == 6
```

You can also use the ** operator multiple times if you know that the dictionaries don't contain overlapping keys:

```
my_kwargs = {
    "number": 20,
}
other_kwargs = {
    "divisor": 7,
}
assert remainder(**my kwargs, **other kwargs) == 6
```

And if you'd like for a function to receive any named keyword argument, you can use the **kwargs catch-all parameter to collect those arguments into a dict that you can then process (see Item 38: "Define Function Decorators with functools.wraps" for when this is especially useful):

```
def print_parameters(**kwargs):
    for key, value in kwargs.items():
        print(f"{key} = {value}")
print_parameters(alpha=1.5, beta=9, gamma=4)
>>>
alpha = 1.5
beta = 9
gamma = 4
```

The flexibility of keyword arguments provides three significant benefits.

The first benefit is that keyword arguments make the function call clearer to new readers of the code. With the call remainder(20, 7), it's not evident which argument is number and which is divisor unless you look at the implementation of the remainder method. In the call with keyword arguments, number=20 and divisor=7 make it immediately obvious which parameter is being used for each purpose.

The second benefit of keyword arguments is that they can have default values specified in the function definition. This allows a function to provide additional capabilities when you need them, but you can accept the default behavior most of the time. This eliminates repetitive code and reduces noise.

For example, say that I want to compute the rate of fluid flowing into a vat. If the vat is also on a scale to measure its weight, then I could use the difference between two weight measurements at two different times to determine the flow rate:

```
def flow_rate(weight_diff, time_diff):
    return weight_diff / time_diff

weight_a = 2.5
weight_b = 3
time_a = 1
time_b = 4
weight_diff = weight_b - weight_a
time_diff = time_b - time_a
flow = flow_rate(weight_diff, time_diff)
print(f"{flow:.3} kg per second")
>>>
```

```
0.167 kg per second
```

In the typical case, it's useful to know the flow rate in kilograms per second. Other times, it'd be helpful to use the last sensor measurements to approximate larger time scales, like hours or days. I can provide this behavior in the same function by adding an argument for the time period scaling factor:

```
def flow_rate(weight_diff, time_diff, period):
    return (weight_diff / time_diff) * period
```

The problem is that now I need to specify the period argument every time I call the function, even in the common case of flow rate per second (where the period is 1):

```
flow_per_second = flow_rate(weight_diff, time_diff, 1)
```

To make this less noisy, I can give the period argument a default value:

```
def flow_rate(weight_diff, time_diff, period=1): # Changed
    return (weight_diff / time_diff) * period
```

The period argument is now optional:

```
flow_per_second = flow_rate(weight_diff, time_diff)
flow_per_hour = flow_rate(weight_diff, time_diff, period=3600)
```

This works well for simple default values that are immutable; it gets tricky for complex default values like list instances and user-defined objects (see Item 36: "Use None and Docstrings to Specify Dynamic Default Arguments" for details).

The third reason to use keyword arguments is that they provide a powerful way to extend a function's parameters while remaining backward compatible with existing callers. This means you can provide additional functionality without having to migrate a lot of existing code, which reduces the chance of introducing bugs.

For example, say that I want to extend the flow_rate function above to calculate flow rates in weight units besides kilograms. I can do this by adding a new optional parameter that provides a conversion rate to alternative measurement units:

The default argument value for units_per_kg is 1, which makes the returned weight units remain kilograms. This means that all existing callers will see no change in behavior. New callers to flow_rate can specify the new keyword argument to see the new behavior:

```
pounds_per_hour = flow_rate(
    weight_diff,
    time_diff,
    period=3600,
    units_per_kg=2.2,
```

)

Providing backward compatibility using optional keyword arguments like this is also crucial for functions that accept *args (see Item 34: "Reduce Visual Noise with Variable Positional Arguments").

The only problem with this approach is that optional keyword arguments like period and units_per_kg may still be specified as positional arguments:

pounds_per_hour = flow_rate(weight_diff, time_diff, 3600, 2.2)

Supplying optional arguments positionally can be confusing because it isn't clear what the values 3600 and 2.2 correspond to. The best practice is to always specify optional arguments using the keyword names and never pass them as positional arguments. As a function author, you can also require that all callers use this more explicit keyword style to minimize potential errors (see Item 37: "Enforce Clarity with Keyword-Only and Positional-Only Arguments").

Things to Remember

- Function arguments can be specified by position or by keyword.
- Keywords make it clear what the purpose of each argument is when it would be confusing with only positional arguments.
- Keyword arguments with default values make it easy to add new behaviors to a function without needing to migrate all existing callers.
- Optional keyword arguments should always be passed by keyword instead of by position.

Item 36: Use None and Docstrings to Specify Dynamic Default Arguments

Sometimes it can be helpful to use a function call, a newly created object, or a container type (like an empty list) as a keyword argument's default value. For example, say that I want to print logging messages that are marked with the time of the logged event. In the default case, I want the message to include the time when the function was called. I might try the following approach, which assumes that the default value for the when keyword argument is reevaluated each time the function is called:

```
from time import sleep
from datetime import datetime

def log(message, when=datetime.now()):
    print(f"{when}: {message}")
```

```
log("Hi there!")
sleep(0.1)
log("Hello again!")
>>>
2024-06-28 22:44:32.157132: Hi there!
2024-06-28 22:44:32.157132: Hello again!
```

This doesn't work as expected. The timestamps are the same because datetime.now is executed only a single time: when the function is defined at module import time. A default argument value is evaluated only once per module load, which usually happens when a program starts up (see Item 98: "Lazy-Load Modules with Dynamic Imports to Reduce Startup Time" for details). After the module containing this code is loaded, the datetime.now() default argument expression will never be evaluated again.

The convention for achieving the desired result in Python is to provide a default value of None and to document the actual behavior in the docstring (see Item 118: "Write Docstrings for Every Function, Class, and Module" for background). When your code sees that the argument value is None, you allocate the default value accordingly:

```
def log(message, when=None):
    """Log a message with a timestamp.
    Args:
        message: Message to print.
        when: datetime of when the message occurred.
        Defaults to the present time.
    """
    if when is None:
        when = datetime.now()
    print(f"{when}: {message}")
```

Now the timestamps will be different:

```
log("Hi there!")
sleep(0.1)
log("Hello again!")
>>>
2024-06-28 22:44:32.446842: Hi there!
2024-06-28 22:44:32.551912: Hello again!
```

Using None for default argument values is especially important when the arguments are mutable. For example, say that I want to load a value that's encoded as JSON data; if decoding the data fails, I want an empty dictionary to be returned by default:

```
import json
def decode(data, default={}):
    try:
        return json.loads(data)
    except ValueError:
        return default
```

The problem here is similar to the problem in the datetime.now example above. The dictionary specified for default will be shared by all calls to decode because default argument values are evaluated only once (at module load time). This can cause extremely surprising behavior:

```
foo = decode("bad data")
foo["stuff"] = 5
bar = decode("also bad")
bar["meep"] = 1
print("Foo:", foo)
print("Bar:", bar)
>>>
Foo: {'stuff': 5, 'meep': 1}
Bar: {'stuff': 5, 'meep': 1}
```

You might expect two different dictionaries, each with a single key and value. But modifying one seems to also modify the other. The culprit is that foo and bar are both equal to the default parameter to the decode function. They are the same dictionary object:

```
assert foo is bar
```

The fix is to set the keyword argument default value to None, document the actual default value in the function's docstring, and act accordingly in the function body when the argument has the value None:

```
def decode(data, default=None):
    """Load JSON data from a string.
    Args:
        data: JSON data to decode.
        default: Value to return if decoding fails.
        Defaults to an empty dictionary.
    """
```

```
try:
    return json.loads(data)
except ValueError:
    if default is None: # Check here
        default = {}
    return default
```

Now, running the same test code as before produces the expected result:

```
foo = decode("bad data")
foo["stuff"] = 5
bar = decode("also bad")
bar["meep"] = 1
print("Foo:", foo)
print("Bar:", bar)
assert foo is not bar
>>>
Foo: {'stuff': 5}
Bar: {'meep': 1}
```

This approach also works with type annotations (see Item 124: "Consider Static Analysis via typing to Obviate Bugs"). Here, the when argument is marked as having an optional value that is a datetime. Thus, the only two valid choices for when are None or a datetime object:

Things to Remember

- ◆ A default argument value is evaluated only once: during function definition at module load time. This can cause odd behaviors for dynamic values (like function calls, newly created objects, and container types).
- Use None as a placeholder default value for a keyword argument that must have its actual default value initialized dynamically.

Document the intended default for the argument in the function's docstring. Check for the None argument value in the function body to trigger the correct default behavior.

• Using None to represent keyword argument default values also works correctly with type annotations.

Item 37: Enforce Clarity with Keyword-Only and Positional-Only Arguments

Passing arguments by keyword is a powerful feature of Python functions (see Item 35: "Provide Optional Behavior with Keyword Arguments"). Keyword arguments enable you to write flexible functions that will be clear to new readers of your code for many use cases.

For example, say that I want to divide one number by another while being very careful about special cases. Sometimes, I want to ignore ZeroDivisionError exceptions and return infinity instead. Other times, I want to ignore OverflowError exceptions and return zero instead. Here, I define a function with these options:

```
def safe division(
    number.
    divisor.
    ignore overflow,
    ignore zero division.
):
    try:
        return number / divisor
    except OverflowError:
        if ignore overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float("inf")
        else:
            raise
```

Using this function is straightforward. This call ignores the float overflow from division and returns zero:

```
result = safe_division(1.0, 10**500, True, False)
print(result)
>>>
0
```

This call ignores the error from dividing by zero and returns infinity:

```
result = safe_division(1.0, 0, False, True)
print(result)
```

>>> inf

The problem is that it's easy to confuse the position of the two Boolean arguments that control the exception handling behavior. This can easily cause bugs that are hard to track down. One way to improve the readability of this code is to use keyword arguments. Using default keyword arguments (see Item 36: "Use None and Docstrings to Specify Dynamic Default Arguments"), the function can be overly cautious and can always re-raise exceptions:

```
def safe_division_b(
    number,
    divisor,
    ignore_overflow=False,  # Changed
    ignore_zero_division=False, # Changed
):
```

Then, callers can use keyword arguments to specify which of the ignore flags they want to set for specific operations, overriding the default behavior:

```
result = safe_division_b(1.0, 10**500, ignore_overflow=True)
print(result)
result = safe_division_b(1.0, 0, ignore_zero_division=True)
print(result)
>>>
0
inf
```

The problem is, because these keyword arguments are optional behavior, there's nothing forcing callers of your functions to use keyword arguments for clarity. Even with the new definition of safe_division_b, I can still call it the old way with positional arguments:

```
assert safe_division_b(1.0, 10**500, True, False) == 0
```

With complex functions like this, it's better to require that callers are clear about their intentions by defining your functions with

keyword-only arguments. These arguments can only be supplied by keyword, never by position.

Here, I redefine the safe_division function to accept keyword-only arguments. The * symbol in the argument list indicates the end of positional arguments and the beginning of keyword-only arguments (*args has the same effect; see Item 34: "Reduce Visual Noise with Variable Positional Arguments"):

```
def safe_division_c(
    number,
    divisor,
    *, # Added
    ignore_overflow=False,
    ignore_zero_division=False,
):
    ...
```

Now, calling the function with positional arguments that correspond to the keyword arguments won't work:

```
safe_division_c(1.0, 10**500, True, False)
>>>
Traceback ...
TypeError: safe_division_c() takes 2 positional arguments but 4
were given
```

But keyword arguments and their default values will work as expected (ignoring an exception in one case and raising it in another):

```
result = safe_division_c(1.0, 0, ignore_zero_division=True)
assert result == float("inf")
```

```
try:
    result = safe_division_c(1.0, 0)
except ZeroDivisionError:
    pass # Expected
```

However, a problem still remains with the safe_division_c version of this function: Callers may specify the first two required arguments (number and divisor) with a mix of positions and keywords:

```
assert safe_division_c(number=2, divisor=5) == 0.4
assert safe_division_c(divisor=5, number=2) == 0.4
assert safe_division_c(2, divisor=5) == 0.4
```

Later, I may decide to change the names of these first two arguments because of expanding needs or even just because my style preferences change:

```
def safe_division_d(
    numerator, # Changed
    denominator, # Changed
    *,
    ignore_overflow=False,
    ignore_zero_division=False
):
```

Unfortunately, this seemingly superficial change breaks all of the existing callers that specified the number or divisor arguments using keywords:

```
safe_division_d(number=2, divisor=5)
```

>>>

Traceback ...

```
TypeError: safe_division_d() got an unexpected keyword argument 

>'number'
```

This is especially problematic because I never intended for the keywords number and divisor to be part of an explicit interface for this function. These were just convenient parameter names that I chose for the implementation, and I didn't expect anyone to rely on them explicitly.

Python 3.8 introduces a solution to this problem, called *positional-only arguments*. These arguments can be supplied only by position and never by keyword (the opposite of the keyword-only arguments demonstrated above).

Here, I redefine the safe_division function to use positional-only arguments for the first two required parameters. The / symbol in the argument list indicates where positional-only arguments end:

```
def safe_division_e(
    numerator,
    denominator,
    /, # Added
    *,
    ignore_overflow=False,
    ignore_zero_division=False,
):
    ...
```

I can verify that this function works when the required arguments are provided positionally:

```
assert safe_division_e(2, 5) == 0.4
```

But an exception is raised if keywords are used for the positional-only parameters:

```
safe_division_e(numerator=2, denominator=5)
```

```
>>>
Traceback ...
TypeError: safe_division_e() got some positional-only arguments
⇔passed as keyword arguments: 'numerator, denominator'
```

Now, I can be sure that the first two required positional arguments in the definition of the safe_division_e function are decoupled from callers. I won't break anyone if I change the parameters' names again.

One notable consequence of keyword- and positional-only arguments is that any parameter name between the / and * symbols in the argument list may be passed either by position or by keyword (which is the default for all function arguments in Python). Depending on your API's style and needs, allowing both argument passing styles can increase readability and reduce noise. For example, here I've added another optional parameter to safe_division that allows callers to specify how many digits to use in rounding the result:

```
def safe_division_f(
    numerator,
    denominator,
    /,
    ndigits=10, # Changed
    * ,
    ignore_overflow=False,
    ignore_zero_division=False,
):
    trv:
        fraction = numerator / denominator # Changed
        return round(fraction, ndigits)
                                             # Changed
    except OverflowError:
        if ignore overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float("inf")
        else:
            raise
```

Now, I can call this new version of the function in all of these different ways, since ndigits is an optional parameter that may be passed either by position or by keyword:

```
result = safe_division_f(22, 7)
print(result)

result = safe_division_f(22, 7, 5)
print(result)

result = safe_division_f(22, 7, ndigits=2)
print(result)
>>>
3.1428571429
3.14286
3.14
```

Things to Remember

- Keyword-only arguments force callers to supply certain arguments by keyword (instead of by position), which makes the intention of a function call clearer. Keyword-only arguments are defined after a * in the argument list (whether on its own or as part of variable arguments like *args).
- Positional-only arguments ensure that callers can't supply certain parameters using keywords, which helps reduce coupling. Positionalonly arguments are defined before a single / in the argument list.
- Parameters between the / and * characters in the argument list may be supplied by position or keyword, which is the default for Python parameters.

Item 38: Define Function Decorators with functools.wraps

Python has special syntax for *decorators* that can be applied to functions. A decorator has the ability to run additional code before and after each call to a function it wraps. This means decorators can access and modify input arguments, return values, and raised exceptions. These capabilities can be useful for enforcing semantics, debugging, registering functions, and more.

For example, say that I want to print the arguments and return value of a function call. This can be especially helpful when debugging the stack of nested function calls from a recursive function. (Logging exceptions could be useful too; see Item 86: "Understand the Difference Between Exception and BaseException"). Here, I define such a decorator by using *args and **kwargs (see Item 34: "Reduce Visual Noise with Variable Positional Arguments" and Item 35: "Provide Optional Behavior with Keyword Arguments") to pass through all parameters to the wrapped function:

```
def trace(func):
    def wrapper(*args, **kwargs):
        args_repr = repr(args)
        kwargs_repr = repr(kwargs)
        result = func(*args, **kwargs)
        print(f"{func.__name__}"
            f"({args_repr}, {kwargs_repr}) "
            f"-> {result!r}")
        return result
```

```
return wrapper
```

I can apply this decorator to a function by using the @ symbol:

```
@trace
def fibonacci(n):
    """Return the n-th Fibonacci number"""
    if n in (0, 1):
        return n
    return fibonacci(n - 2) + fibonacci(n - 1)
```

Using the @ symbol is equivalent to calling the decorator on the function it wraps and assigning the return value to the original name in the same scope:

```
fibonacci = trace(fibonacci)
```

The decorated function runs the wrapper code before and after fibonacci runs. It prints the arguments and return value at each level in the recursive stack:

```
fibonacci(4)
>>>
fibonacci((0,), {}) -> 0
fibonacci((1,), {}) -> 1
fibonacci((2,), {}) -> 1
fibonacci((1,), {}) -> 1
fibonacci((0,), {}) -> 0
fibonacci((1,), {}) -> 1
fibonacci((2,), {}) -> 1
fibonacci((2,), {}) -> 2
fibonacci((3,), {}) -> 3
```

This works well, but it has an unintended side effect. The value returned by the decorator—the function that's called above—doesn't think it's named fibonacci:

```
print(fibonacci)
```

```
>>>
<function trace.<locals>.wrapper at 0x104a179c0>
```

The cause of this isn't hard to see. The trace function returns the wrapper defined within its body. The wrapper function is what's assigned to the fibonacci name in the containing module because of the decorator. This behavior is problematic because it undermines tools that do introspection, such as debuggers (see Item 114: "Consider Interactive Debugging with pdb").

For example, the help built-in function is useless when called on the decorated fibonacci function. It should print out the docstring defined above ("""Return the n-th Fibonacci number"""), but it doesn't:

```
help(fibonacci)
>>>
Help on function wrapper in module __main__:
```

```
wrapper(*args, **kwargs)
```

Another problem is that object serializers (see Item 107: "Make pickle Serialization Maintainable with copyreg") break because they can't determine the location of the original function that was decorated:

```
import pickle
```

```
pickle.dumps(fibonacci)
>>>
Traceback ...
AttributeError: Can't pickle local object 'trace.<locals>.
wwrapper'
```

The solution is to use the wraps helper function from the functools built-in module. This is a decorator that helps you write decorators. When you apply it to the wrapper function, it copies all of the important metadata about the inner function to the outer function. Here, I redefine the trace decorator using wraps:

```
from functools import wraps
def trace(func):
    @wraps(func) # Changed
```

• • •

Now, running the help function produces the expected result, even though the function is decorated:

```
help(fibonacci)
>>>
Help on function fibonacci in module __main__:
fibonacci(n)
    Return the n-th Fibonacci number
The idle distance by sub-
```

The pickle object serializer also works:

```
print(pickle.dumps(fibonacci))
```

>>>

```
b'\x80\x04\x95\x1a\x00\x00\x00\x00\x00\x00\x00\x8c\x08_main__\
➡x94\x8c\tfibonacci\x94\x93\x94.'
```

Beyond these examples, Python functions have many other standard attributes (e.g., __name__, __module__, __annotations__) that must be preserved to maintain the interface of functions in the language. Using wraps ensures that you'll always get the correct behavior.

Things to Remember

- Decorators in Python are syntax to allow one function to modify another function at runtime.
- Using decorators can cause strange behaviors in tools that do introspection, such as debuggers.
- Use the wraps decorator from the functools built-in module when you define your own decorators to avoid any issues.

Item 39: Prefer functools.partial over lambda Expressions for Glue Functions

Many APIs in Python accept simple functions as part of their interface (see Item 100: "Sort by Complex Criteria Using the key Parameter," Item 27: "Prefer defaultdict over setdefault to Handle Missing Items in Internal State," and Item 24: "Consider itertools for Working with Iterators and Generators"). However, these interfaces can cause friction because they might fall short of your needs.

For example, the reduce function from the functools built-in module allows you to calculate one result from a near-limitless iterable of values. Here, I use reduce to calculate the sum of many log-scaled numbers (which effectively multiplies them):

```
def log_sum(log_total, value):
    log_value = math.log(value)
    return log_total + log_value

result = functools.reduce(log_sum, [10, 20, 40], 0)
print(math.exp(result))
>>>
8000.0
```

The problem is that you don't always have a function like log_sum that exactly matches the function signature required by reduce. For example, imagine that you simply had the parameters reversed—since it's an arbitrary choice anyway—with value first and log_total second. How could you easily fit this function to the required interface?

```
def log_sum_alt(value, log_total): # Changed
    ...
```

One solution is to define a lambda function in an expression to reorder the input arguments to match what's required by reduce:

```
result = functools.reduce(
    lambda total, value: log_sum_alt(value, total), # Reordered
    [10, 20, 40],
    0,
)
```

For one-offs, creating a lambda like this is fine. But if you find yourself doing this repeatedly and copying code, it's worth defining another helper function with reordered arguments that you can call multiple times:

```
def log_sum_for_reduce(total, value):
    return log_sum_alt(value, total)
```

Another situation where function interfaces are mismatched is when you need to pass along some additional information for use in processing. For example, say I want to choose the base for the logarithm instead of always using natural log:

```
def logn_sum(base, logn_total, value): # New first parameter
    logn_value = math.log(value, base)
    return logn_total + logn_value
```

In order to pass this function to reduce, I need to somehow provide the base argument for every call. But reduce doesn't give me a way to do this easily. Again, lambda can help here by allowing me to specify one parameter and pass through the rest. Here, I always provide 10 as the first argument to logn_sum in order to calculate a base-10 logarithm:

```
result = functools.reduce(
    lambda total, value: logn_sum(10, total, value), # Changed
    [10, 20, 40],
    0,
)
print(math.pow(10, result))
>>>
8000.00000000004
```

This pattern of pinning some arguments to specific values while allowing the rest of them to be passed normally is quite common with functional-style code. This technique is often called *Currying* or *partial application*. The functools built-in module provides the partial function to make this easy and more readable. It takes the function to partially apply as the first argument followed by the pinned positional arguments:

```
result = functools.reduce(
   functools.partial(logn_sum, 10), # Changed
   [10, 20, 40],
   0,
)
```

partial also allows you to easily pin keyword arguments (see Item 35: "Provide Optional Behavior with Keyword Arguments" and Item 37: "Enforce Clarity with Keyword-Only and Positional-Only Arguments" for background). For example, imagine that the logn_sum function accepts base as a keyword-only argument, like this:

```
def logn_sum_last(logn_total, value, *, base=10): # New kwarg
    logn_value = math.log(value, base)
    return logn_total + logn_value
```

Here, I use partial to pin the value of base to Euler's number:

Achieving the same behavior is possible with a lambda expression, but it's verbose and error prone:

```
log_sum_e_alt = lambda *a, base=math.e, **kw: \
    logn_sum_last(*a, base=base, **kw)
```

partial also allows you to inspect which arguments have already been supplied, and the function being wrapped, which can be helpful for debugging:

```
print(log_sum_e.args, log_sum_e.keywords, log_sum_e.func)
>>>
() {'base': 2.718281828459045} <function logn_sum_last at
=0x1033534c0>
```

In general, you should prefer using partial when it satisfies your use case because of these extra niceties. However, partial can't be used to reorder the parameters altogether, so that's one situation where lambda is preferable.

In many cases, a lambda or partial instance is still not enough, especially if you need to access or modify state as part of a simple function interface. Luckily, Python provides additional facilities, including closures, to make this possible (see Item 33: "Know How Closures Interact with Variable Scope and nonlocal" and Item 48: "Accept Functions Instead of Classes for Simple Interfaces").

Things to Remember

- lambda expressions can succinctly make two function interfaces compatible by reordering arguments or pinning certain parameter values.
- The partial function from the functools built-in is a general tool for creating functions with pinned positional and keyword arguments.
- Use lambda instead of partial if you need to reorder the arguments of a wrapped function.

Index

Symbols

| pipe operator, 36 + operator, 43 --version flag, 1 % operator, 44, 50 * operator, 151–152 ** operator, 153-154 @ symbol, 167 _asdict method, 226 _astuple method, 225, 227, 228-229 __call__ method, 204-205 __debug__ variable, 443–444 __delattr__ method, 252 __dict__, 306-310 __eq__ method, 226 __getattr__ method, 279-281 __getattribute__ method, 281–283 __getitem__ method, 261–263 __hash__ method, 258 __init__ method, 217–219, 235–236 __init_subclass__ method, 288-299, 306-310 ____missing___ method, 124–127 @property decorator versus refactoring attributes, 270 - 274reuse, 274-279 setters and getters, 266-269 _replace method, 255 __repr__ method, 223–224 __set_name__ method, 278, 302-303 __setattr__ method, 252, 283–285

Α

accumulate function, 106–107 adjust module, 479-482 advanced string formatting, 52-55 algorithm, 493 binary search, 502–503 leaky-bucket, 270-273 alias. 136 all function, 98-100 animate function. 187 animation generator, 186–188 any function, 100–101 API exceptions, 595 hooks, 201–203 Python, 468, 471-472 Python C extension, 460, 465 stability, 590-592 append method, 182, 506 argparse module, 479-481 arguments function, 111-112, 135-137 iterating over, 87–92 keyword, 153-157, 161-162, 219-221 keyword-only, 162-164 None as default value, 157–160 passing by reference, 251 positional, 150–152 positional-only, 164-166 asdict function, 226 assert statement, 404-408, 442-443

assertAlmostEqual method, 564-565 assertNotAlmostEqual method, 565 assignment expressions, 22-23, 24 - 30, 86exceptions, 180 in list comprehensions, 178-181 slices, 68-69 AST (abstract syntax tree), 205-206, 208-210 asynchronous coroutines, 319 asynchronous I/O, 368, 376 asyncio module, 329, 366-367, 368–384. See also coroutines event loops, 389-392 thread/coroutine interoperability, 381-384 attributes, 306 annotation, 299-303 descriptor protocol, 274. See also descriptors establishing relationships between, 303-310 lazy, 279–285 modifying, 299-303 protected, 247-250 public, 245-250, 266-269 updating values, 254-256 auto-formatting, black, 5-6

B

BaseException class, 419-424 batched function. 105–106 binary operators, 44 binary search algorithm, bisect module, 502-503 binary to Unicode conversion, 42 bisect module, 501–503 black, 5-6 blocking I/O, 324-329, 348, 348–349. See also coroutines blocks, 399 else, 82-84 try, 412-414 try/except/else, 400-402 try/except/else/finally, 402-404 try/finally, 399-400 Boolean logic, De Morgan's laws, 101 break statement, 82-83, 85-86 breaking circular dependencies, 600 - 602dynamic imports, 604-605 import, configure, run, 603-604 reordering imports, 602 breakpoint function, 566–568 Bucket class, 270-273 buffer protocol, CPython, 487 built-in function. enumerate, 78-79 filter, 174–175 format, 52-53 hash, 110 help, 168 isinstance, 206 iter, 89–91 map, 174, 175 range, 77–78 super, 238-240 zip, 80-81 built-in modules, 217 bytearray, 489-491 bytecode, 324-325, 475-478 bytes, 41-46

С

C3 linearization, 238 capture patterns, 32-33 cardinality, 505-506 careful_divide function, 142 case clause, 33-34, 36-37 catch-all unpacking, 72-76, 138 - 139CFFI module, 461 chain function, 102 chained exceptions, 428-436 child processes chaining, 322-323 decoupling from parent, 321 managing, 320–324 polling, 320–321 circular dependencies, breaking, 600 - 602dynamic imports, 604–605 import, configure, run, 603-604 reordering imports, 602

class method polymorphism, 230 - 235class/es BaseException, 419-424 Bucket, 270-273 composability, 310-317 CountMissing, 204–205 Decimal, 523-525 decorators, 315–317 defaultdict, 123-124, 202-204 definition, 211–212 deque, 504–509 descriptor, 269, 275-279, 299-303, 306, 307 docstrings, 584–585 Exception, 416–424 helper, 204 hierarchy, 130 inheritance. 286-287 JsonMixin, 243-244 lightweight, 140–141 Lock. 332–333 mix-in, 240–245 OrderedDict, 112 parent, 235-236, 237-240 Point, 256-258 public attributes, 245–250 Queue, 337-344, 353-360 refactoring, 131-133 registration, 293–299 serializing to and from **JSON. 244** storing values in attributes. 131 - 133ThreadPoolExecutor, 361-363, 394 - 397user-defined. 137 versioning, 530-531 clause. case, 33-34, 36-37 finally, 436-442 if, 20, 36, 174 closures, 145-149, 203, 383 code. See also statements concurrent, 319 expressions, 4-5 formatting. 4 microbenchmarks, 453-458

modules. 5 module-scoped, 593-595 organization, 212 polymorphism, 207–210 port to coroutines and asynchronous I/O, 368-381 Pythonic, 1 refactoring, 131-133, 387 testing, 533 whitespace, 3-4 code points, 42 coin-flipping loop, 98–101 cold start, 475 collections.abc module, 91, 114 defaultdict class, 123-124 inheriting classes for custom container types, 260-264 combinations function, 108 combinations_with_replacement function, 108 command-line tool, timeit, 457 - 458commands, interactive debugger, 567 comma/s. See also syntax implicit string concatenation, 64 single-element tuple, 17-18 communicate method, 321 community-built modules, 575-576 comparing objects, 227-230 compile time, error detection, 6–8 composability, 310-317 comprehensions, 173 assignment expressions, 178–181 dictionary, 174, 179 generator expressions, 184–186 list, 79-80, 86, 98, 173-174 with more than two subexpressions, 176–177 concatenation, string, 62-66 concurrency, 319 coordinating work between threads, 333-344 coroutines, 364-368 fan-in. 348 fan-out, 348, 349-353 when to use, 344-349

concurrent.futures module, 361, 393-397 conditional expressions, 10, 19-23 containers, 261-263 iterating over, 95–98 iterator protocol, 89–92 staging modifications, 96–97 contains function, 37 context managers, 409-412 contextlib module, 409-412 converting objects to tuples, 224-225 Unicode to binary, 42 Conway's Game of Life, 344-348, 364 coordinating work between threads, 333 - 344copy method, 136 copyreg module, 528, 532 coroutines, 364-368 interoperability with threads, 381-384 moving code to, 368-381 versus threads, 364 count variable, 24-26 CountMissing class, 204-205 CPU. See also concurrency; parallelism; performance; threads parallelism, 319 pipeline, 333–337 threads, 319 CPython, 324–325 buffer protocol, 487 compiling Python without GIL. 327 extension modules, 467-474 GIL (global interpreter lock), 324-327, 329, 330 performance, 475 creating, objects, 217-218, 219-223, 250-251, 254-256 C-style formatting, 47–52 csv module, 303-306 ctypes module, 460, 462-467 Currying, 171–172 cycle function, 103 Cython, 461

D

data. semi-structured vs. encapsulated, 37-39 data races, preventing, 330-333 dataclasses module, 131, 217. See also object/s decorator, 218, 223 keyword arguments, 219-221 type checker, 218–219 datetime module, 521-523 De Morgan's laws, 101 debugging. See also exceptions/ exception handling interactive, 565-570 memory usage, 570–573 postmortem, 568–570 print function, 58-62 traceback, 424-436 Decimal class, 523-525 decorator/s, 166-169, 309, 309-310 @property, 266-269, 270-279 class, 315-317 contextmanager, 409-410 dataclasses module, 218 function, 166-169, 309, 310-312 singledispatch, 212-216 trace_func, 312-313 decoupling, child process from parent, 321 defaultdict class, 123-124, 202 - 204dependency/ies, 212, 481. See also breaking circular dependencies circular, breaking, 600–605 encapsulating, 559-562 hell, 577 injection, 604 reproducing, 580–582 deployment environment, 593-595 deque class, 504-509 descriptors, 269, 275-279, 299-303, 306, 307 deserialize function, 296-297 destructuring, 34–37 deterministic behavior, 203 development environment, 593 diamond inheritance, 237-240

dictionary/ies, 109, 306 __dict__, 306-310 comprehensions, 174, 179 converting objects to, 225-226 format strings, 50–52 handling missing keys, 117–121, 122 - 127immutable objects, 256-260 iterating over, 92–93, 109–116 KeyError exception, 117–118 key/value pairs, 12 nested, 127-130 whitespace, 4 docstrings, 158-160, 582-583 class, 584-585 function, 585-586 module. 584 dot_product function, 463–464 double-ended queue, 507 dropwhile function, 105 duck typing, 113, 500-501, 616 dumps function, 225–226 dynamic attributes, 265 dynamic imports, 604–605 dynamic inspection, 240-241

E

else blocks, 82-84, 400-402 empty sequence, looping over, 83 empty tuple, 16 encapsulated data, 37-39 encapsulating dependencies, 559 - 562encoding, open, 46 enhance module, 479-482 enumerate function. 78-79. See also iteration equivalence checking, object, 226 - 227error/s. See also exceptions/ exception handling checking, 6-8 implicit concatenation, 63–64 escaping, 63 eval function, 445–446 evaluate function, 206-208 event loop, 364-365 event loops, 389-392

exceptions/exception handling, 268, 399 assert statement, 404-408 in assignment expressions, 180 BaseException class, 419-424 chained, 428-436 Exception class, 416-424 generator, 195-199 GeneratorExit, 420, 438-442 IndexError, 336 KeyboardInterrupt, 419 KeyError, 117-118, 433 MissingError, 429-436 NameError, 147 OSError, 353 raise statement, 405-408 raising, 142–145 Reset, 197 root, 595-600 ServerMissingKeyError, 433 StopIteration, 88, 198, 438 SyntaxError, 533 traceback, 424-428 try blocks, 412-414 try/except/else blocks, 400–402 try/except/else/finally blocks, 402 - 404try/finally blocks, 399-400 ValueError, 401 variables, 414-416 exec function, 445–446 explicit string concatenation, 66 expression/s, 4–5 assignment, 22-23, 24-30, 68-69, 86, 178-181 conditional, 10, 19-23 C-style formatting, 47–52 generator, 181, 184-186 lambda, 172 starred, 73-76, 140 yield, 187-188, 409-410 extending tuples, 131 extension modules, 467–474, 625

F

fail function, 19–20 fan-in, 348 fan-out, 348, 349–353 FIFO (first in, first out) queue, 504-509. See also producerconsumer queue; queue file system cache, 477 files read mode, 46 write mode, 45 filter function, 174-175 filtering items from an iterator dropwhile function, 105 filterfalse function, 105 islice function, 104 takewhile function, 104-105 finally blocks, 399-400 finally clause, 436-442 first-class function, 202 floating point tests, 563-565 for loops, 14-15, 173-174 avoiding else blocks, 82-84 iterator protocol, 89-92 variables, 85-86 for statement, 79, 80 format code, 4 specifiers, 47-48, 53 format built-in function, 52-53 formatting, 47 advanced string, 52-55 C-style, 47-52 f-string, 56–58 functional-style programming, 250 - 251dynamic inspection, 240-241 mix-in classes, 240-245 single dispatch, 212-216 function/s, 175, 238-240, 315-317 accumulate, 106-107 all. 98-100 animate, 187 any, 100–101 arguments, 111-112, 135-137, 153-160,161-166 asdict, 226 async, 364-365 batched, 105-106 bisect_left, 502-503 breakpoint, 566-568 careful_divide, 142

chain. 102 closure, 145-149, 203, 383 combinations, 108 combinations_with replacement, 108 contains, 37 coroutines, 364-368 cycle, 103 decorator, 166-169, 309, 310-312 deserialize, 296-297 docstrings, 585-586 dot_product, 463-464 dropwhile, 105 dumps, 225-226 enumerate, 78-79 eval, 445-446 evaluate, 206-208 exec, 445-446 fail. 19-20 filter, 174-175 filterfalse, 105 finally clause, 436–437 first-class, 202 format, 52–53 generator, 87, 173, 182-184. See also generator/s get_cause, 435 get_stats, 139-140 hash, 110 help, 168 helper, 8–11, 21, 42–43, 84, 126, 145-146, 168-169, 232, 251 hooks, 201–203 index_words, 182, 183-184 isinstance, 206 islice, 104 iter, 89-91 kernel, 460 key, 494–499 lambda, 170-171 log_missing, 202 lookup, 431-436 map, 174 mapreduce, 233-235 my_print, 213 namedtuple, 259 None return value, 142 normalization, 87, 89

normalize_defensive, 91-92 pairwise, 106 partial, 171–172 patch, 555–558 permutations, 107 positional arguments, 150-152 print, 58-62 product, 107 raising exceptions, 142–145 range, 77-78 repeat, 103 returning more than one value, 138 - 141run_report, 416–418 setUpModule, 548–549 sorted, 499-501 with statement, 408–412 takewhile, 104-105 tearDownModule, 548-549 tee, 103 utility, 451 wrapper, 168 zip, 80–81 zip_longest, 81, 103–104 functools module, 170, 171-172

G

gc module, 571-572 GeneratorExit exception, 420, 438 - 442generator/s, 87, 173, 182-184, 189-191 animation, 186-188 assignment expressions, 181 compose with yield from expression, 186-188 expressions, 184–186 lazy, 80–81 passing iterators as arguments, 188-195 throw method. 195–199 wave, 189-195 get method, 9-11, 118-120 get_cause function, 435 get_stats function, 139–140 getter methods, 266-269 GIL (global interpreter lock), 324-327, 329, 330

glider, 345–346 global scope, 147 guard expression, 36

H

hash function, 110 heap, 514 heapq module, 514–519 help function, 168 helper class, 204 helper function, 8–11, 21, 42–43, 84, 126, 145–146, 168–169, 232, 251 helper method, 123 open_picture, 126–127 TestCase class, 535–541 hooks, 201–203 __getattr_, 279–281 __getattribute_, 281–283 __setattr_, 283–285

Ι

if clause, 36, 174 if statements, 19 immutable objects creating, 250–251 using in dictionaries and sets, 256 - 260implicit string concatenation, 62 - 66imports, 5, 592, 601, 604–605 in operator, 117, 119 index_words function, 182, 183-184 IndexError exception, 336 indexing, 72–73. See also slicing negative, 68 parallel, 79-80 with slicing, 67–69 strings, 10-11 inheritance class, 286-287 diamond, 237-240 multiple, 236, 239, 240 inheriting classes for custom container types, 260-264 initializing parent classes, 235–236 inline negation, 5 insert_value, profiling, 448-451
insertion ordering, dictionary, 109-116 insertion_sort, profiling, 448-451 integration tests, 541-542 interactive debugging, 565-568 interpolated format strings, 56-58 I/O asynchronous, 368, 376 blocking, 324-329, 348 threaded, porting to asyncio, 368-381 isinstance function, 206 islice function, 104 islice method. 72 iter built-in function, 89-91 iteration/iterators, 77, 182-183. See also loop/s exception, 88 generator, 87 lists, 79-80, 87 over arguments, 87-92 over containers, 95–98 over dictionaries, 92-93, 109-116 over lists, 94-95 over sets, 93, 96 passing into generators as arguments, 188–195 passing to all built-in function, 98-100 passing to any built-in function, 100-101 StopIteration exception, 88 zip generator, 80–81 iterator protocol, 89–92 itertools, 72, 185-186 accumulate function, 106-107 batched function. 105-106 chain function. 102 combinations function, 108 combinations_with_replacement function, 108 cycle function, 103 dropwhile function, 105 filterfalse function. 105 islice function, 104 pairwise function, 106 permutations function, 107 product function, 107

repeat function, 103 takewhile function, 104–105 tee function, 103 zip_longest function, 81, 103–104

J-K

JSON, 37–38, 244, 293–297 JsonMixin class, 243–244

kernel functions, 460 key function, 494–499 KeyboardInterrupt exception, 419 KeyError exception, 117–118, 433 keyword arguments, 153–157, 161–162, 219–221 -only arguments, 162–164 strict, 81

L

lambda function, 170-171, 495 lazy attributes, 279-285 lazy generator, 80-81 lazy loading, 478-485 leaky-bucket algorithm, 270-273 lightweight class, 140–141 linking iterators together chain function, 102 cycle function, 103 repeat function, 103 tee function, 103 zip_longest function, 103–104 list/s, 260–261. See also indexing cardinality, 505-506 comprehensions, 79-80, 86, 98, 173-177, 178-181, 184-186 dequeuing items, 504–509 dictionary value, 119-121 iterating over, 94–95 iteration, 78, 87 nested, 127-130 priority queue, 509-514 sorting items in, 145, 493-499 literal values. 16 Lock class, 332-333 log_missing function, 202 lookup function, 431–436

loop/s, 77 break statement, 82–83, 85–86 coin-flipping, 98–101 in comprehensions, 176–177 else block, avoiding, 82–84 event, 364–365, 389–392 for, 14–15, 85–86, 173–174 -and-a-half idiom, 29–30 iterator protocol, 89–92 range function, 77–78 while, 29, 382–383

M

managing, child processes, 320 - 324map function, 174, 175 mapreduce function, 233-235 match statement, 30–34, 370 destructuring, 34–37 semi-structured vs. encapsulated data, 37–39 memory bytecode caching, 475–478 file system cache, 477 lazy loading, 478-485 leaks, 278–279 usage, debugging, 570-573 memoryview type, 487-491 metaclasses, 265, 285-287 class registration, 293–299 modifying a class's attributes, 299 - 303subclass validation. 287-291 TraceMeta, 313-314 method/s @property. See @property decorator _asdict, 226 _astuple, 225, 227, 228-229 __call__, 204–205 __delattr__, 252 __eq__, 226 __getattr__, 279–281 __getitem__, 261–263 ___hash__, 258 __init__, 217-219, 235-236 306-310

__iter__, 89-91 _replace, 255 _repr_, 223-224 __set_name__, 278, 302-303 __setattr__, 252, 283–285 append, 182, 506 assertAlmostEqual, 564-565 assertNotAlmostEqual, 565 communicate, 321 copy, 136 get, 9–11, 118–120 getter, 266-269, 276 helper, 123 islice, 72 open_picture, 126–127 pretty, 209 print_callees, 452–453 print_callers, 452 public attributes, 246 quantize, 525 read, 230-231 register, 213 run, 198-199 send, 189–191 setdefault, 120-121, 122-123, 125 setter, 266-269, 276 sort, 145-149, 201-202, 493-499 str.format, 54-55 throw, 195-199 title. 50 update, 96 whitespace, 4 metrics. 447 microbenchmarks, 453-458, 490-491, 512-517 MissingError exception, 429–436 mix-in classes, 240–245 mocks, 550–558. See also test/s module/s, 5, 109 adjust, 479-482 argparse, 479-481 asyncio, 329, 366-367, 368-384 bisect, 501-503 built-in, 217 bytecode caching, 475-478 CFFI, 461 collections.abc, 91, 114, 260-264

community-built, 575–576 concurrent.futures, 361, 393 - 397contextlib, 409-412 copyreg, 528, 532 CProfile, 449-450 csv. 303-306 ctypes, 460, 462-467 dataclasses, 131, 217, 218-219 datetime, 521-523 decimal, 523-525 docstrings, 584 enhance, 479-482 extension, 467-474, 625 functools, 170, 171–172 gc, 571–572 heapq, 514–519 lazy-loading, 478–485 multiprocessing, 393–397 Numba, 461 Numby, 461 pdb, 566–568 pickle, 526-532 pkgutil, 624–626 root exception, 595–600 -scoped code, 593–595 subprocess, managing child processes, 320-324 threading, 332 time, 519–521 timeit. 453–458 traceback, 424-428 tracemalloc, 572–573 typing, 613-621 unittest, 533-535, 548 version, 577-578 warnings, 605–613 zipimport, 622–624 zoneinfo, 522-523 MRO (method resolution order), 238 multiple inheritance, 236, 239, 240 multiple return values, unpacking, 138 - 141multiple-assignment unpacking, 11 - 15multiprocessing module, 393–397 multithreading, preemptive, 325, 331-333

mutex, 325, 330, 332–333, 408 my_print function, 213 mypy tool, 115–116 Mypyc, 461

N

namedtuple function, 259 NameError exception, 147 negative indexing, 68 None, using as default argument value, 157–160 nonlocal statement, 148–149 normalization function, 87, 89 normalize_defensive function, 91–92 Numba module, 461 Numby module, 461

0

object/s, 60 AST (abstract syntax tree), 208 - 210converting to dictionaries, 225 - 226converting to tuples, 224–225 creating, 217-218, 219-223 creating copies with replaced attributes. 254-256 diamond inheritance, 237–240 dynamic internal state, 127 enabling for comparison, 227-230 equivalence checking, 226-227 immutable, 250-251, 256-260 -oriented polymorphism, 207-210 preventing from being modified, 251 - 254representing as strings, 223-224 serialization, 168, 528-530 sorting, 494–495 OOP (object-oriented programming) class definition, 211-212 dependencies, 212. See also dependency/ies polymorphism, 207-210 open encoding mode, 46 open source projects, 621-626 open_picture method, 126–127

operating system blocking I/O, 327, 348-349 system calls, 327–329 operator %, 44, 50 *, 151-152 **, 153-154 | pipe, 36 +, 43 binary, 44 in, 117, 119 ternary, 19-20 walrus, 24-30, 179-181. See also assignment expressions OrderedDict class, 112 **OSError** exception, 353

P

packages, 588 namespaces, 588-590 stable APIs. 590-592 pairwise function, 106 parallel indexing, 79–80 parallelism, 319 concurrent.futures, 393-397 data races, preventing, 330–333 fan-out, 350-353 managing child processes, 320 - 324parent class diamond inheritance, 237-240 initializing, 235–236 parent process, decoupling child process from, 321 parentheses assignment expressions, 22–23 single-element tuple, 16–19 partial function, 171-172 patch functions, 555–558 PEP 8 (Python Enhancement Proposal #8), 3 automation, 5–6 expressions and statements, 4-5 imports, 5 naming, 4 whitespace, 3-4 performance. See also profiling CPython, 475 engineering, 447

extension modules. 467-474 lazy-loading modules, 478–485 loading modules from zip archives, 622-623 metrics. 447 microbenchmarks, 490-491, 512-517 producer-consumer queue, 504-509 profiling, 448–453 replacing Python with another programming language, 458 - 462search. 501-503 timeit microbenchmarks, 453–458 permutations function, 107 pickle module, 526–528 default attribute values, 528–530 stable import paths, 531–532 versioning classes, 530–531 pip tool, 575–577 pipeline, 333-337, 416-417 piping data into a subprocess, 321-322 pkgutil module, 624-626 Point objects, 256-258 polymorphism, 207-210, 230-235 porting code to use coroutines and asyncio, 368-384 bottom-up approach, 387–388 top-down approach, 384–387 positional arguments, variable, 150 - 152positional-only arguments, 164-166 postmortem debugging, 568–570 preemptive multithreading, 325, 331-333 pretty method, 209 pretty print, 215 preventing objects from being modified. 251-254 print function, 58-62 print_callees method, 452-453 print_callers method, 452 priority queue, 509–519 private attributes, 246-247 producer-consumer queue, 504-509

producing combinations of items from iterators accumulate function, 106–107 batched function, 105-106 combinations function, 108 combinations_with_replacement function, 108 pairwise function, 106 permutations function, 107 product function, 107 product function, 107 profiling, 448 insertion_sort and insert_value, 448-451 print_callees method, 452–453 print_callers method, 452 utility functions, 451 programs. See also code; performance bytecode, 324-325 concurrent, 319. See also concurrency deployment environment, 593-595 development environment, 593 pipeline, 333–337 piping data into a subprocess, 321 - 322preemptive multithreading, 325 speedup, 319 protected attributes, 247-250 public attributes, 245-250, 266 - 269pylint, 6 PyPI (Python Package Index), 575 Python C extension API, 460, 465 knowing the version you're using, 1–3 replacing with another programming language, 458 - 462support for threads, 327-329 Pythonic style, 1

9

quantize method, 525 query string parameter, 8–9 queue priority, 509–519 producer-consumer, 504–509 Queue class, 337–344, 353–360 quota, leaky-bucket algorithm, 270–273

R

raise statement, 405-408 raising exceptions, 142–145. See also exceptions/exception handling assert statement, 404-408 generator, 195-199 raise statement, 405-408 try blocks, 412-414 range function, 77-78 read binary mode, 46 read method, 230-231 refactoring, 131-133, 387 register method, 213 registration, class, 293–299 repeat function, 103 replacing Python with another programming language, 458 - 462repr, difference between str and, 58 - 62reproducing dependencies, 580–582 Reset exception, 197 root exceptions, 595–600 rule of least surprise, 265 run method. 198-199 run_report function, 416–418 runtime, error checking, 6-8

S

scope, global, 147 scoping bug, 147–148 scripts, 399 security eval and exec built-in functions, 446 pickle module, 526 semi-structured data, 37–39 send method, 189–191 sequences, 261–264. *See also* list/s empty, looping over, 83

slicing, 67–70 ServerMissingKeyError exception, 433 setdefault method, 120–121, 122-123. 125 sets immutable objects, 256-260 iterating over, 93, 96 setter methods, 266–269 setUpModule function, 548–549 simulate function, 365-367 single dispatch, 212–216. See also OOP (object-oriented programming) single-element tuple parentheses, 16-17 trailing comma, 17 slicing, 67-68, 70 indexing, 67–69 memoryview, 487-488 stride syntax, 70-72 syntax, 67 sort method, 145-149, 201-202, 493 - 501sorted function, 499-501 speedup, 319 staging modifications, 96–97 star args, 150–152 starred expression, 73-76, 140 statement/s, 4–5 assert. 404–408. 442–443 break, 82-83, 85-86 for, 79, 80 if. 19 import, 5, 592 match, 30-39, 370 nonlocal, 148-149 raise. 405-408 with, 408–412 static analysis, 613–621 static error, 7 StopIteration exception, 88, 198, 438 str, 41-46, 58-62, 524 str.format method, 54–55 strict keyword, zip function, 81 striding, 70-72 string/s

advanced formatting, 52-55 concatenation, 62-66 C-style formatting, 47–52 indexing, 10-11 interpolated format, 56-58 literals, 63 repr, 58-62 representing objects as, 223-224 subclass/es protected attributes, 247-250 validation, 287-291 subprocess module, managing child processes, 320-324 super function, 238-240 superclass, 207, 297 SWIG, 461 syntax slicing, 67–68 stride. 70–72 unpacking, 12-15, 138-139 SyntaxError exception, 533 system calls, 327–329

Т

takewhile function, 104–105 as targets, enabling, 410–412 tearDownModule function, 548-549 tee function, 103 ternary operator, 19–20 TestCase subclasses, 535-541, 547 - 548test/s, 533. See also debugging encapsulating dependencies, 559 - 562floating point, 563–565 harness, 547-549 integration, 541–542 mocks. 550–558 TestCase subclasses, 535–541 unit, 542-547 unittest module, 533-535 threaded I/O, porting to asyncio, 368-381 threading module, 332 ThreadPoolExecutor class, 361-363, 394-397 thread/s, 319. See also concurrency; parallelism

avoiding for on-demand fan-out, 349-353 blocking I/O, 327-329 combining with coroutines, 381-384 coordinating work between, 333-344 versus coroutines, 364 interoperability with coroutines, 381-384 locks, 332-333 preemption, 331-333 preventing data races, 330-333 Python support for, 327–329 worker, 330-331, 335-337, 339-340, 342, 354, 358, 389 - 392throw method, 195-199 time module, 519–521 timeit microbenchmarks, 453-458 title method, 50 tools. See also itertools: modules black, 5-6 Cython, 461 error checking, 8 mypy, 115–116 Mypyc, 461 pip, 575–577 static analysis, 614–621 SWIG, 461 timeit. 457–458 venv. 578-580 trace func decorator, 312-313 traceback. 424-436 tracemalloc module, 572-573 TraceMeta metaclass, 313–314 trailing comma, single-element tuple, 17-18 tree walking, 206 try blocks, 412–414 try/except/else blocks, 400-402 try/except/else/finally blocks, 402 - 404try/finally blocks, 399-400 tuple/s, 11–12, 131 comparator methods, 496 converting objects to, 224-225 empty, 16

extending, 131 literal values, 16 nested, 127–130 single-element, 16 parentheses, 16–17 trailing comma, 17–18 two-, 143 type annotation, 115–116, 144–145, 254, 586–587 type checker, static, 613–621 type registration, 293–299 typing module, 613–621

U

underscore (_), 213 Unicode to binary conversion, 42 code points, 42 sandwich, 42–43 unit tests, 542-547 unittest module, 533-535, 548 unpacking, 12–15 catch-all, 72-76, 138-139 multiple return values, 138–141 multiple-assignment, 11–15 update method, 96 user-defined classes. 137 UTC (Coordinated Universal Time), 519 utility functions, 451

v

validation, subclass, 287–291 ValueError exception, 401 varargs, 150–152 variable/s __debug__, 443–444 alias, 136 count, 24-26 exception, 414-416 for loop, 85-86 scope, 147-148 venv, 578-580 version class, 530-531 module, 577-578 Python 2, 2 Python 3, 2-3

W

walrus operator, 24–30, 179. *See also* assignment expressions warnings module, 605–613 wave generator, 189–195 while loop, 29, 82–84, 382–383 whitespace, 3–4 with statement, 408–412 wordcode, 324–325 worker threads, 330–331, 335–337, 339–340, 342, 354, 358, 389–392 wrapper function, 168 wraps function, 168–169 write binary mode, 45 write text mode, 45

X-Y-Z

yield expressions, 187-188, 409-410

Zen of Python, The, 4 zero-copy operations, 487–491 zip function, 80–81 zip_longest function, 81, 103–104 zipimport module, 622–624 zoneinfo module, 522–523