Better PYTHON CODE

David **Mertz**

Foreword by Alex Martelli

FREE SAMPLE CHAPTER

Praise for *Better Python Code*

"You'll not just be aspiring to be an expert anymore after practicing through *Better Python Code: A Guide for Aspiring Experts*, you'll be one of them! Learn from David Mertz, who's been making experts through his writing and training for the past 20 years."
—*Iqbal Abdullah, past Chair, PyCon Asia Pacific, and past board member, PyCon Japan*

"In *Better Python Code: A Guide for Aspiring Experts*, David Mertz serves up bite-sized chapters of Pythonic wisdom in this must-have addition to any serious Python programmer's collection. This book helps bridge the gap from beginner to advanced Python user, but even the most seasoned Python programmer can up their game with Mertz's insight into the ins and outs of Python."
—*Katrina Riehl, President, NumFOCUS*

"What separates ordinary coders from Python experts? It's more than just knowing best practices—it's understanding the benefits and pitfalls of the many aspects of Python, and knowing when and why to choose one approach over another. In this book David draws on his more than 20 years of involvement in the Python ecosystem and his experience as a Python author to make sure that the readers understand both *what* to do and *why* in a wide variety of scenarios."
—*Naomi Ceder, past Chair, Python Software Foundation*

"Like a Pythonic BBC, David Mertz has been informing, entertaining, and educating the Python world for over a quarter of a century, and he continues to do so here in his own pleasantly readable style."
—*Steve Holden, past Chair, Python Software Foundation*

"Being expert means someone with a lot of experience. David's latest book provides some important but common problems that folks generally learn only after spending years of doing and fixing. I think this book will provide a much quicker way to gather those important bits and help many folks across the world to become better."
—*Kushal Das, CPython Core Developer and Director, Python Software Foundation*

"This book is for everyone: from beginners, who want to avoid hard-to-find bugs, all the way to experts looking to write more efficient code. David Mertz has compiled a great set of useful idioms that will make your life as a programmer easier and your users happier."
—*Marc-André Lemburg, past Chair, EuroPython, and past Director, Python Software Foundation*

This page intentionally left blank

Better Python Code

This page intentionally left blank

Better Python Code

A Guide for Aspiring Experts

David Mertz

Addison-Wesley
Hoboken, New Jersey

Cover image: StudioLondon/Shutterstock
Python and the Python Logo are trademarks of the Python Software Foundation.
Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.
macOS® is a trademark of Apple Inc.
Microsoft Windows is a trademark of the Microsoft group of companies.

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.

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: 2023944574

Copyright © 2024 Pearson Education, Inc.

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: 978-0-13-832094-2
ISBN-10: 0-13-832094-2

$PrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

This page intentionally left blank

This book is dedicated to my mother, Gayle Mertz, who always valued ideas and the relentless criticism of existing reality.

Contents

Foreword xvii

Preface xix

Acknowledgments xxv

About the Author xxvii

Introduction 1

1 Looping Over the Wrong Things 3

1.1 (Rarely) Generate a List for Iteration 3
1.2 Use `enumerate()` Instead of Looping Over an Index 6
1.3 Don't Iterate Over `dict.keys()` When You Want `dict.items()` 8
1.4 Mutating an Object During Iteration 9
1.5 `for` Loops Are More Idiomatic Than `while` Loops 12
1.6 The Walrus Operator for "Loop-and-a-Half" Blocks 13
1.7 `zip()` Simplifies Using Multiple Iterables 15
1.8 `zip(strict=True)` and `itertools.zip_longest()` 17
1.9 Wrapping Up 20

2 Confusing Equality with Identity 21

2.1 Late Binding of Closures 21
2.2 Overchecking for Boolean Values 25
2.3 Comparing `x == None` 28
2.4 Misunderstanding Mutable Default Arguments 29
2.4.1 First Approach, Use a Class 31
2.4.2 Second Approach, Use a `None` Sentinel 32

2.4.3 Third Approach, Take Advantage of Stateful Generators 32
2.5 Copies versus References to Mutable Objects 33
2.6 Confusing `is` with == (in the Presence of Interning) 35
2.7 Wrapping Up 37

3 A Grab Bag of Python Gotchas 39
3.1 Naming Things 39
3.1.1 Naming a File Identically to a Standard Library Module 39
3.1.2 Avoid Using `import *` 42
3.1.3 Bare or Overly Generic `except` Statements 46
3.2 Quadratic Behavior of Naive String Concatenation 52
3.3 Use a Context Manager to Open a File 56
3.3.1 First Danger 57
3.3.2 Second Danger 58
3.3.3 Correcting the Fragility 58
3.4 Optional Argument `key` to `.sort()` and `sorted()` 59
3.5 Use `dict.get()` for Uncertain Keys 62
3.6 Wrapping Up 64

4 Advanced Python Usage 67
4.1 Comparing `type(x) == type(y)` 67
4.2 Naming Things (Revisited) 71
4.2.1 Overriding Names in Built-ins 71
4.2.2 Directly Accessing a Protected Attribute 75
4.3 Keep Less-Used Features in Mind 79
4.3.1 F-String Debugging 80
4.3.2 The Elegant Magic of Decorators 83

4.3.3 Use `itertools` (Sufficiently) 90
4.3.4 The `more-itertools` Third-Party Library 95
4.4 Type Annotations Are Not Runtime Types 98
4.4.1 Type Annotations Are Not Runtime Constraints 100
4.4.2 Mistaking `typing.NewType()` for a Runtime Type 102
4.5 Wrapping Up 105

5 Just Because You Can, It Doesn't Mean You Should... 107
5.1 Metaclasses 107
5.2 Monkeypatching 112
5.3 Getters and Setters 115
5.4 It's Easier to Ask for Forgiveness Than Permission 118
5.5 Structural Pattern Matching 121
5.6 Regular Expressions and Catastrophic Backtracking 123
5.7 Wrapping Up 126

6 Picking the Right Data Structure 129
6.1 `collections.defaultdict` 129
6.2 `collections.Counter` 132
6.2.1 The Solution 132
6.2.2 The Mistake 134
6.3 `collections.deque` 135
6.3.1 The Solution 136
6.3.2 The Mistake 137
6.4 `collections.ChainMap` 138
6.4.1 The Solution 139
6.4.2 The Mistake 140
6.5 Dataclasses and Namedtuples 141
6.5.1 Using Namedtuples 142
6.5.2 Static versus Dynamic 144
6.5.3 Data Classes 145

6.6 Efficient Concrete Sequences 146
6.7 Wrapping Up 150

7 Misusing Data Structures 153
7.1 Quadratic Behavior of Repeated List Search 153
7.2 Deleting or Adding Elements to the Middle of a List 157
7.2.1 More Efficient Data Structures 161
7.3 Strings Are Iterables of Strings 163
7.4 (Often) Use `enum` Rather Than `CONSTANT` 166
7.5 Learn Less Common Dictionary Methods 169
7.5.1 The Dictionaries Defining Objects 169
7.5.2 Back to Our Regularly Scheduled Mistake 171
7.6 JSON Does Not Round-Trip Cleanly to Python 174
7.6.1 Some Background on JSON 175
7.6.2 Data That Fails to Round-Trip 176
7.7 Rolling Your Own Data Structures 178
7.7.1 When Rolling Your Own Is a Bad Idea 179
7.7.2 When Rolling Your Own Is a Good Idea 181
7.7.3 Takeaways 186
7.8 Wrapping Up 187

8 Security 189
8.1 Kinds of Randomness 190
8.1.1 Use `secrets` for Cryptographic Randomness 190
8.1.2 Reproducible Random Distributions 192
8.2 Putting Passwords or Other Secrets in “Secure” Source Code 195

8.3 "Rolling Your Own" Security Mechanisms 198
8.4 Use SSL/TLS for Microservices 201
8.5 Using the Third-Party `requests` Library 205
8.6 SQL Injection Attacks When Not Using DB-API 208
8.7 Don't Use `assert` to Check Safety Assumptions 212
8.8 Wrapping Up 215

9 Numeric Computation in Python 217

9.1 Understanding IEEE-754 Floating Point Numbers 217
9.1.1 Comparing NaNs (and Other Floating Point Numbers) 218
9.1.2 NaNs and `statistics.median()` 221
9.1.3 Naive Use of Floating Point Numbers: Associativity and Distributivity 224
9.1.4 Naive Use of Floating Point Numbers: Granularity 226
9.2 Numeric Datatypes 228
9.2.1 Avoid Floating Point Numbers for Financial Calculations 228
9.2.2 Nonobvious Behaviors of Numeric Datatypes 232
9.3 Wrapping Up 239

A Appendix: Topics for Other Books 241

A.1 Test-Driven Development 241
A.2 Concurrency 242
A.3 Packaging 243
A.4 Type Checking 243
A.5 Numeric and Dataframe Libraries 244

Index 245

This page intentionally left blank

Foreword

It was a pleasure for me to be asked to write a foreword for David's new book, as I always expect David to provide useful, insightful content.

Much as I began with high expectations, I am delighted to say that they were not just met but exceeded: The book is an engaging read, offers a great deal of insight for anyone at an intermediate or advanced level to improve their Python programming skill, and includes copious sharing of precious experience practicing and teaching the language; it is easy to read and conversational in style. In spite of all this, David manages to keep the book short and concise enough to absorb quickly and fully.

Most of the book's content reflects, and effectively teaches, what amounts to a consensus among Python experts about best practices and mistakes to avoid. In a few cases in which the author's well-explained opinions on certain issues of style differ from those of other experts, David carefully and clearly points out these cases so readers can weigh the pros and cons and come to their own decisions.

Most of the book deals with Python-related issues at intermediate levels of experience and skill. These include many instances in which programmers familiar with different languages may adopt an inferior style in Python, simply because it appears to be a direct "translation" of a style that's appropriate for the languages that they know well.

An excellent example of the latter problem is writing APIs that expose *getter* and *setter* methods: In Python, direct getting and setting of the attribute (often enabled via the `property` decorator) should take their place. Reading hypothetical code like `widgets.set_count(widgets.get_count() + 1)` — where experienced Pythonistas would instead have used the direct, readable phrasing `widgets.count += 1` — would clearly show that the hypothetical coder is ignoring or unaware of Python "best practices." David's book goes a long way toward addressing this and other common misunderstandings.

Despite its overall intermediate level, the book does not hesitate to address quite a few advanced topics, including the danger of catastrophic backtracking in regular expressions, some quirks in floating-point representations of numbers, "round-tripping" problems with serialization approaches like JSON, etc. The coverage of such issues makes studying this book definitely worthwhile, not just for Python programmers of intermediate skills, but for advanced ones too.

—*Alex Martelli*

Preface

Python is a very well-designed programming language. In surprisingly many cases, the language manages to meet one of the aphorisms in Tim Peters' *The Zen of Python*: "There should be one—and preferably only one—obvious way to do it." If there is only one way to do it, it's hard to make mistakes.

Of course, that aphorism is an aspiration that is not uniformly met. Often there are many ways to perform a task in Python, many of them simply wrong, many inelegant, many leaning heavily on idioms of other programming languages rather than being *Pythonic*, and some of them not exactly *wrong* but still grossly inefficient. All the problems described in this book are ones that I've seen in real-life code, sometimes in the wild, sometimes caught during code review, and admittedly far too often in code I wrote myself before reflecting upon its flaws.

About the Book

The sections of this book each present some mistake, pitfall, or foible that developers can easily fall into, and are accompanied by descriptions of ways to avoid making them. At times those solutions simply involve a minor change in "spelling," but in most cases they require a nuance of thought and design in your code. Many of the discussions do something else as well…

I do not hope only to show you something you did not know, but in a great many cases I hope to show you something about which you did not know there *was something* to know. I believe that the most effective writing and teaching conveys to readers or students not only information, but also good *ways of thinking* about problems and reasoning about their particular solutions. The info boxes, footnotes, and silly digressions within this book all hope to allow you to think deeper about a particular domain, or a particular task, or a particular style of programming.

There is no need to read this book from cover to cover (but I believe that readers who do so will benefit). Each chapter addresses a related cluster of concepts, but stands alone. Moreover, each section within a chapter is also self-contained. Each can be read independently of the others, and most readers will learn something interesting in each one. Some of the sections are more advanced than others, but even in those that seem introductory, I think you will find nuances you did not know. And even in those that seem advanced, I hope you will find the discussions accessible and enlightening.

Notwithstanding that each section forms a sort of vignette, the chapters are generally organized in sequence of increasing sophistication, and the sections loosely build upon

each other. Where it feels helpful, many discussions refer to other sections that might provide background, or foreshadow elaboration in later sections.

In general, I am aiming at a reader who is an intermediate-level Python developer, or perhaps an advanced beginner. I assume you know the basics of the Python programming language; these discussions do not teach the most basic syntax and semantics that you would find in a first course or first book on Python. Mostly I simply assume you have an inquisitive mind and a wish to write code that is beautiful, efficient, and correct.

This book is written with Python 3.12 in mind, which was released in October 2023. Code shown has been tested against 3.12 betas. The large majority of the code examples will work in Python 3.8, which is the earliest version that has not passed end-of-life as of mid-2023. In some cases, I note that code requires at least Python 3.10, which was released on October 4, 2021; or occasionally at least Python 3.11, released on October 24, 2022. The large majority of the mistakes discussed within this book were mistakes already in Python 3.8, although a few reflect improvements in later versions of Python.

Documents titled "*What's new in Python M.m.μ*"[1] have been maintained since at least the Python 1.4 days (in 1996).[2]

Code Samples

Most of the code samples shown in this book use the Python REPL (Read-Evaluate-Print-Loop). Or more specifically, they use the IPython (https://ipython.readthedocs.io) enhanced REPL, but using the `%doctest_mode` magic to make the prompt and output closely resemble the plain `python` REPL. One IPython "magic" that is used fairly commonly in examples is `%timeit`; this wraps the standard library `timeit` module, but provides an easy-to-use and adaptive way of timing an operation reliably. There are some *mistakes* discussed in this book where a result is not *per se* wrong, but it takes orders of magnitude longer to calculate than it should; this magic is used to illustrate that.

When you write your own code, of course, interaction within a REPL—including within Jupyter notebooks (https://jupyter.org) or other richly interactive environments—will only be a small part of what you write. But the mistakes in this book try to focus on samples of code that are as narrow as possible. An interactive shell is often a good way to illustrate these mistakes; I encourage you to borrow the lessons you learn, and copy them into full `*.py` files. Ideally these discussions can be adapted into rich codebases after starting as mere snippets.

At times when presenting commands run in the operating system shell (i.e., running a Python script to show results), I display the command prompt `[BetterPython]$` to provide a quick visual clue. This is not actually the prompt on my personal machine, but rather is something to which I could change if I wanted to do so. On Unix-like systems, the `$` is often (but not always) part of shell prompts.

1. Python does not strictly use Semantic Versioning (https://semver.org), so my implied nomenclature "major.minor.micro" is not strictly accurate.
2. See https://docs.python.org/3/whatsnew/index.html for an index of past release notes.

Note A short introduction to a REPL

Many developers who have come from other programming languages, or who are just beginning programming in general, may not appreciate how amazingly versatile and useful an interactive shell can be. More often than not, when I wish to figure out how I might go about some programming task, I jump into a Python, IPython, or Jupyter environment to get a more solid understanding of how my imagined approach to a problem will work out.

A quick example of such a session, for me within a `bash` terminal, might look like this:

```
[BetterPython]$ ipython
Python 3.11.0 | packaged by conda-forge |
    (main, Oct 25 2022, 06:24:40) [GCC 10.4.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.7.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: %doctest_mode                                   # ❶
Exception reporting mode: Plain
Doctest mode is: ON
>>> from collections import ChainMap                    # ❷
>>> ChainMap?                                           # ❸
Init signature: ChainMap(*maps)
Docstring:
A ChainMap groups multiple dicts (or other mappings) together
to create a single, updateable view.
[...]
File:           ~/miniconda3/lib/python3.11/collections/__init__.py
Type:           ABCMeta
>>> dict1 = dict(foo=1, bar=2, baz=3)
>>> dict2 = {"bar": 7, "blam": 55}
>>> chain = ChainMap(dict1, dict2)
>>> chain["blam"], chain["bar"]                         # ❹
(55, 2)
>>> !ls src/d*.adoc                                     # ❺
src/datastruct2.adoc  src/datastruct.adoc
```

❶ Use a display style similar to running just `python` with no script.

❷ I pressed <tab> to select a completed line after `collections`.

❸ I'd like information about what this object does (abridged here).

❹ Entering expressions shows their value immediately.

❺ With `!`, I can run a command within an external shell and see results.

There's much more to what REPLs can do than is shown, but this gives you a quick feel for their capabilities.

Different programming environments will treat copying/pasting code samples into them differently. Within IPython itself, using the `%paste` magic will ignore the leading `>>>` or `...` characters in an appropriate way. Various other shells, IDEs, and code editors will behave differently. Many of the code samples that are presented outside a REPL, and also many of the data files used, are available at https://gnosis.cx/better. Moreover, paths are mostly simplified for presentation; files often live within the `code/` or `data/` subdirectories of the book's website, but those paths are usually not shown. In other words, the code presented is used to explain concepts, not as reusable code I intend for you to copy directly. (That said, you *may* use it, of course.) In particular, much of the code shown is code that has *foibles* in it; for that code, I most certainly do not want you to use it in production.

All code blocks whose title includes "Source code of <filename>" are available for download from https://gnosis.cx/better. In some cases, the code shown in this book is an excerpt from a longer file named. All other code blocks, whether titled to aid navigation or untitled, are present only to explain concepts; of course, you are free to use them by copying, retyping, or adapting for your purpose.

Obtaining the Tools Used in This Book

The Python programming language is Free Software that may be obtained at the official site of the Python Software Foundation (PSF). A variety of other entities have also created customized Python distributions with additional or different capabilities bundled with the same core programming language. These include many operating system vendors. Most Linux distributions bundle Python. macOS (formerly stylized in slightly different ways, such as "Mac OS X" and "OS X") has included Python since 2001. It is available for Windows from the Microsoft Store.

To obtain the PSF distribution of Python, go to https://www.python.org/downloads/. Versions are available for many operating systems and hardware platforms. To follow some of the examples within this book, using the IPython terminal-based REPL (https://ipython.org/install.html) or Jupyter notebooks (https://docs.jupyter.org/en/latest/install.html) is advisable. These enhanced interactive environments support "magics," such as `%timeit`, that are special commands not contained in the Python language itself, but which can improve interactive exploration. Throughout the book, when interactive sessions are shown, they can be easily identified by a leading `>>>` for initial lines and leading `...` for continuation lines (when present). However, Jupyter—as well as the interactive shells in many integrated development environments (IDEs) or sophisticated code editors—mark code entered and results produced by other visual indicators. The enhanced REPLs mentioned also support adding a single or double `?` at the end of a Python name to display information about the object it refers to; this is used in some examples.

I personally use Miniconda (https://docs.conda.io/en/latest/miniconda.html) as a means of installing Python, IPython, Jupyter, and many other tools and libraries. Miniconda itself contains a version of Python, but will also allow creation of *environments* with different versions of Python, or indeed without Python at all, but rather other useful tools. You will see hints in some examples about my choice of installation, but nothing in the book depends on you following my choice.

Other Useful Tools

Most of the discussions in this book are conceptual rather than merely stylistic. However, linters will often detect mistakes that at least border on conceptual, including sometimes mistakes described in this book. A particularly good linter for Python is Flake8 (https://flake8.pycqa.org/), which actually utilizes several lower-level linters as (optional) dependencies. A good linter may very well not detect important mistakes, but you cannot go wrong in at least *understanding* why a linter is complaining about your code.

The home page for the Black code formatter (https://black.readthedocs.io/) describes itself well:

> Black is the uncompromising Python code formatter. By using it, you agree to cede control over minutiae of hand-formatting. In return, Black gives you speed, determinism, and freedom from pycodestyle nagging about formatting. You will save time and mental energy for more important matters.
>
> **— Black home page**

Opinions about using Black vary among Pythonistas. I have found that even if Black occasionally formats code in a manner I wouldn't entirely choose, enforcing consistency when working with other developers aids the readability of shared code, especially on large projects.

A very impressive recent project for linting and code formatting is Ruff (https://beta.ruff.rs/docs/). Ruff covers most of the same linting rules as Flake8 and other tools, but is written in Rust and runs several orders of magnitude faster than other linters. As well, Ruff provides auto-formatting similar to Black, but cleans up many things that Black does not address. (However, Black also cleans things that Ruff does not; they are complementary.)

In modern Python development, type annotations and type-checking tools are in relatively widespread use. The most popular of these tools are probably Mypy (http://mypy-lang.org/), Pytype (https://google.github.io/pytype/), Pyright (https://github.com/Microsoft/pyright), and Pyre (https://pyre-check.org/). All of these tools have virtues, especially for large-scale projects, but this book generally avoids discussion of the Python type-checking ecosystem. The kinds of mistakes that type checking can detect are mostly disjointed from the semantic and stylistic issues that we discuss herein.

Register your copy of *Better Python Code* 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 (9780138320942) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

A number of participants in the Python-Help Discourse board (https://discuss.python.org/c/users/7) have suggested nice ideas for these mistakes. For many of their suggestions, I had already included their idea, or some variation of it, but in other cases, their thoughts prompted an addition to or modification of the mistakes I address. I greatly thank Chris Angelico, Charles Machalow, John Melendowski, Steven D'Aprano, Ryan Duve, Alexander Bessman, Cooper Lees, Peter Bartlett, Glenn A. Richard, Ruben Vorderman, Matt Welke, Steven Rumbalski, and Marco Sulla for their suggestions.

Other friends who have made suggestions include Brad Huntting, Adam Peacock, and Mary Ann Sushinsky.

This book is far better thanks to the suggestions I received; all errors remain my own.

This page intentionally left blank

About the Author

David Mertz, Ph.D., has been a member of the Python community for a long time: about 25 years—long enough to remember what was new about Python 1.5 versus 1.4. He has followed the development of the language closely, given keynote talks about many of the changes over versions, and his writing has had a modicum of influence on the directions Python and popular libraries for it have taken. David has taught Python to scientists, developers coming from other languages, and programming neophytes.

You can find voluminous details about his publications at https://gnosis.cx/publish/resumes/david-mertz-publications.pdf. You can learn more about where he has worked at https://gnosis.cx/publish/resumes/david-mertz-resume.pdf.

This page intentionally left blank

7

Misusing Data Structures

Python has extremely well-designed data structures and data representations, many of which are discussed in the prior chapter. However, a few antipatterns, that are unfortunately common, can make the use of data structures dramatically inefficient or lead to unintended behavior in your code.

7.1 Quadratic Behavior of Repeated List Search

In Python, the `in` keyword is a very flexible way of looking for "membership" in an object, most often some sort of container. Behind the scenes, the keyword `in` is calling the `.__contains__(self, elem)` method of the object that potentially has something "inside" it.

Bear with me for a few paragraphs while I discuss the behavior of `in`, and before I get to the quadratic behavior gotcha one can encounter using lists. I believe a deeper understanding of the mechanisms of "containment" will help many developers who might have only an approximate mental model of what's going on.

A great many kinds of objects—some that might seem unexpected—respond to `in`. Here is an example.

`RegexFlag` can check for membership

```
>>> import re
>>> flags = re.VERBOSE | re.IGNORECASE | re.DOTALL | re.UNICODE
>>> type(flags)
<flag 'RegexFlag'>
>>> re.U in flags
True
>>> type(re.M)
<flag 'RegexFlag'>
```

In a commonsense way, the flag `re.U` (which is simply an alias for `re.UNICODE`) is *contained* in the mask of several flags. A single flag is simply a mask that indicates only one

operational `re` modifier. Moreover, a few special objects that are not collections but iterables also respond to `in`. For example, `range` is special in this way.

Exploring what a `range` is

```
>>> import collections
>>> r = range(1_000_000_000_000)                       # ❶
>>> isinstance(r, collections.abc.Collection)
True
>>> r[:10]                                              # ❷
range(0, 10)
>>> r[999_999_999_990:]
range(999999999990, 1000000000000)
>>> f"{r[999_999_999_990:][5]:,}"                       # ❷
'999,999,999,995'
```

❶ Ostensibly, this is a very large collection; in truth it is a very compact representation that doesn't actually *contain* a trillion integers, only the endpoints and step of the range.

❷ A number of clever shortcuts exist in the implementation of the `range` object, generally producing what we "expect."

Part of the cleverness of `range` is that it does not need to do a linear search through its items, even though it is in many respects list-like. A `range` object behaves like a realized list in most ways, but only *contains* anything in a synthetic sense. In other words, `range(start, stop, step)` has an internal representation similar to its call signature, and operations like a slice or a membership test are calculated using a few arithmetic operations. For example, `n in my_range` can simply check whether $start \leq n < stop$ and whether $(n - start) \;\%\; step = 0$.

Timing the efficiency of `range`

```
>>> %timeit 10 in r
54 ns ± 0.85 ns per loop (mean ± std. dev. of 7 runs, 10,000,000
loops each)
>>> %timeit 999_999_999_995 in r
77 ns ± 0.172 ns per loop (mean ± std. dev. of 7 runs, 10,000,000
loops each)
```

The time to check for membership of an element near the "start" of a range is almost identical to that for membership of an element near the "end" because Python is not actually *searching* the members.

Lists are a concrete and ordered collection of elements that can be appended to very quickly, and have a few other internal optimizations. However, we have to watch where the *ordered* part might bite us. The only generic way to tell if an element is contained in a

list is to do a linear search on the list. We might not find it until near the end of the search, and if it isn't there, we will have had to search the entire list.

Note When you want your lists sorted

We can use the `bisect` module in the standard library if we wish to speed this greatly for lists we are happy to keep in sorted order (which is not all of our lists, however). The `sortedcontainers` third-party library (https://grantjenks.com/docs/sortedcontainers/) also provides a similar speedup when we can live with the mentioned constraint.

We can see where checking containment within a list becomes unwieldy with a simple example. I keep a copy of the 267,752-word SOWPODS (https://en.wikipedia.org/wiki/Collins_Scrabble_Words) English wordlist on my own system. We can use that as an example of a moderately large list (of strings, in this case).

Searching the SOWPODS wordlist

```
>>> words = [w.rstrip() for w in open('data/sowpods')]
>>> len(words)
267752
>>> import random
>>> random.seed(42)
>>> most_words = random.sample(words, k=250_000)    # ❶
>>> %timeit "zygote" in most_words
2.8 ms ± 147 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> %timeit "zebra" in most_words
200 µs ± 12.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
each)
>>> %timeit "aardvark" in most_words
172 µs ± 776 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops
each)
>>> %timeit "coalfish" in most_words
10.7 ms ± 163 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

❶ The words are genuinely shuffled in the random sampling.

We can see that both "aardvark" and "zebra" take a fairly modest 200 microseconds to search. Showing that the `most_words` list really is not ordered alphabetically, "zygote" takes over 10 times as long to find (but it *is* found).

However, "coalfish" (a genuine word in the full dictionary, closely related in the Linnaean classification system to pollock) takes over 10 milliseconds because it is never found in the sampled list.

For a one-off operation, 10 milliseconds is probably fine. But imagine we want to do something slightly more complicated. The example is somewhat artificial, but one can realistically imagine wanting instead to compare lists of people's names or addresses for a degree of duplication—or, for example, of shotgun-sampled nucleotide fragments from soil—in a real-world situation.

Finding words from one collection in another collection

```
>>> random.seed(13)
>>> some_words = random.sample(words, k=10_000)
>>> sum(1 for word in some_words if word not in most_words)
649
>>> %timeit sum(1 for word in some_words if word not in most_words)
55.2 s ± 1.26 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Taking a full minute for this simple operation is terrible, and it gets worse quickly—at approximately an $O(N^2)$ rate (to be precise, it is $\Omega(N \times M)$ since it gets even worse as the hit rate goes down for specific data).[1]

What we've shown is concise and superficially intuitive code to perform one linear scan of `most_words` for every word in `some_words`. That is, we perform an O(N) scan operation M different times (where N and M are the sizes of the respective lists). A quick clue you can use in spotting such pitfalls is to look for multiple occurrences of the `in` keyword in an expression or within a suite. Whether in an `if` expression or within a loop, the complexity is similar.

Fortunately, Python gives us a very efficient way to solve exactly this problem by using sets.

Efficiently finding words from one collection in another collection

```
>>> len(set(some_words) - set(most_words))
649
>>> %timeit len(set(some_words) - set(most_words))
43.3 ms ± 1.31 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

That's better than a 1000x speedup. We can see that the result is exactly the same. Even assuming we needed to concretely look at where those words occur within our lists rather

1. The so-called big-O notation is commonly used in computer science when analyzing the complexity of an algorithm. Wikipedia has a good discussion at https://en.wikipedia.org/wiki/Big_O_notation. There are multiple symbols used for slightly different characterizations of asymptotic complexity: O, o, Ω, ω, and Θ. Big-O is used most commonly, and indicates a worst-case behavior; Big-Theta indicates an asymptote for both worst case and best case; Big-Omega indicates a best-case behavior. Small-o and Small-omega are used to express the somewhat more complex concepts of one function *dominating* another rather than *bounding* another.

than merely count them or see what they are, 649 operations of `some_words.index(word)` is *comparatively* cheap relative to the three-orders-of-magnitude difference encountered (looking through the shorter list is far faster, and typically we find the different word after searching halfway).

> **Note Trie structures for fast prefix search**
>
> If the particular problem discussed is genuinely close to the one you face, look towards the third-party module `pygtrie` (https://pypi.org/project/pygtrie/), which will probably get you even faster and more flexible behavior. For the precise problem described, `CharTrie` is the class you'd want. In general, the *trie* data structure (https://en.wikipedia.org/wiki/Trie) is very powerful for a class of string search algorithms.

7.2 Deleting or Adding Elements to the Middle of a List

An early discussion in this book, in Chapter 3, *A Grab Bag of Python Gotchas*, addresses how naive string concatenation within a loop might encounter quadratic complexity. That is to say, the overall time and computation needed to perform a sequence of N operations is $O(N^2)$.[2]

Although in many situations the solution to a slowdown in (certain) string operations is to simply "use a list instead" (perhaps followed by a final `"".join(thelist)` to get back a string), lists have their own very similar danger. The problem here is in not understanding what is "cheap" and what is "expensive" for lists. Specifically, inserting or removing items from a list anywhere other than at the end is *expensive*.

We first explore some details of exactly how lists are implemented in Python, then look at which other data structures would be good choices for which actual use cases.

Python gives you the ability to insert or remove items from anywhere within a list, and for some purposes it will seem like the obvious approach. Indeed, for a few operations on a relatively small list, the minor inefficiency is wholly unimportant.

2. *Ibid*.

Note Cost and amortized cost

For lists, accessing an item at a given numeric position is O(1). Changing the value at a numeric position is O(1). Perhaps surprisingly, `list.append()` and `list.pop()` are also *amortized* O(1).

That is, adding more items to a list will intermittently require reallocating memory to store their object references; but Python is clever enough to use pre-allocated reserve space for items that might be added. Moreover, as the size of a list grows, the pre-allocation padding also grows. The overall effect is that reallocations become rarer, and their relative cost averages out to 0% asymptotically. In CPython 3.11, we see the following behavior on an x86-64 architecture (but these details are not promised for a different Python implementation, version, or chip architecture):

```
>>> from sys import getsizeof
>>> def pre_allocate():
...     lst = []
...     size = getsizeof(lst)
...     print(" Len   Size  Alloc")
...     for i in range(1, 10_001):
...         lst.append('a')
...         newsize = getsizeof(lst)
...         if newsize > size:
...             print(f"{i:>4d}{newsize:>7d}{newsize-size:>6d}")
...             size = newsize
...
>>> pre_allocate()                                    # ❶
 Len   Size  Alloc  |   Len   Size  Alloc
   1     88    32   |   673   6136   704
   5    120    32   |   761   6936   800
   9    184    64   |   861   7832   896
  17    248    64   |   973   8856  1024
  25    312    64   |  1101  10008  1152
  33    376    64   |  1245  11288  1280
  41    472    96   |  1405  12728  1440
  53    568    96   |  1585  14360  1632
  65    664    96   |  1789  16184  1824
  77    792   128   |  2017  18232  2048
  93    920   128   |  2273  20536  2304
 109   1080   160   |  2561  23128  2592
 129   1240   160   |  2885  26040  2912
 149   1432   192   |  3249  29336  3296
 173   1656   224   |  3661  33048  3712
 201   1912   256   |  4125  37208  4160
 233   2200   288   |  4645  41880  4672
```

```
269   2520   320   |  5229  47160  5280
309   2872   352   |  5889  53080  5920
353   3256   384   |  6629  59736  6656
401   3704   448   |  7461  67224  7488
457   4216   512   |  8397  75672  8448
521   4792   576   |  9453  85176  9504
593   5432   640
```

❶ Printed output modified to show two columns of `len/size/alloc`

This general pattern of pre-allocating a larger amount each time the list grows, roughly in proportion to the length of the existing list, continues for lists of millions of items.

Inserting and removing words from middle of list

```
>>> words = [get_word() for _ in range(10)]
>>> words
['hennier', 'oughtness', 'testcrossed', 'railbus', 'ciclatoun',
'consimilitudes', 'trifacial', 'mauri', 'snowploughing', 'ebonics']
>>> del words[3]                                    # ❶
>>> del words[7]
>>> del words[3]                                    # ❶
>>> words
['hennier', 'oughtness', 'testcrossed', 'consimilitudes', 'trifacial',
'mauri', 'ebonics']
>>> words.insert(3, get_word())
>>> words.insert(1, get_word())
>>> words                                           # ❷
['hennier', 'awless', 'oughtness', 'testcrossed', 'wringings',
'consimilitudes', 'trifacial', 'mauri', 'ebonics']
```

❶ The word deleted at initial index 3 was *railbus*, but on next deletion *ciclatoun* was at that index.

❷ The word *wringings* was inserted at index 3, but got moved to index 4 when *awless* was inserted at index 1.

Note Focus on concepts, but code available at book website

The specific implementation of the `get_word()` function used here is not important. However, as with other examples ancillary to the main point of a section, or requiring larger datasets, the source code and data file can be found at https://gnosis.cx/better. All that matters for the current section is that `get_word()` returns some string each time it is called.

For the handful of items inserted and removed from the small list in the example, the relative inefficiency is not important. However, even in the small example, keeping track of *where* each item winds up by index becomes confusing.

As the number of operations gets large, this approach becomes notably painful. The following toy function performs fairly meaningless insertions and deletions, always returning five words at the end. But the general pattern it uses is one you might be tempted towards in real-world code.

Asymptotic timing for insert-and-delete from list middle

```
>>> from random import randrange
>>> def insert_then_del(n):
...     words = [get_word() for _ in range(5)]
...     for _ in range(n):
...         words.insert(randrange(0, len(words)), get_word())
...     for _ in range(n):
...         del words[randrange(0, len(words))]
...     return words
...
>>> insert_then_del(100)
['healingly', 'cognitions', 'borsic', 'rathole', 'division']
>>> insert_then_del(10_000)
['ferny', 'pleurapophyses', 'protoavis', 'unhived', 'misinform']
>>> %timeit insert_then_del(100)
109 µs ± 2.42 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops
each)
>>> %timeit insert_then_del(10_000)
20.3 ms ± 847 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
>>> %timeit insert_then_del(1_000_000)
1min 52s ± 1.51 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Going from 200 operations (counting each of insertion and deletion) to 20,000 operations takes on the order of 200x as long. At these sizes the lists themselves are small enough to matter little; the time involved is dominated by the number of calls to `get_word()`, or perhaps a bit to `randrange()`, although we still see a 2x proportional slowdown from the list operations.

However, upon increasing the number of operations by another 100x, to 2 million, linear scaling would see an increase from 20 ms to about 2 seconds. Instead it jumps to nearly 2 minutes, or about a 55x slowdown from linear scaling. I watched my memory usage during the 15 minutes that `%timeit` took to run the timing seven times, and it remained steady.

It's not that these operations actually use very much memory; rather, every time we insert one word near the middle of a 1 million word list, that requires the interpreter to move 500,000 pointers up one position in the list. Likewise, each deletion near the middle of a 1 million word list requires us to move the top 500,000 pointers back down. This gets much worse very quickly as the number of operations increases further.

7.2.1 More Efficient Data Structures

There is no one solution to the problem described here. On the other hand, there is exceedingly rarely an actual use case for the exact behavior implemented by code such as the preceding example. Trust me, code like that is not purely contrived for this book—I have encountered a great much like it in production systems (with the problem buried beneath a lot of other functionality in such code).

If you merely need to be able to insert and delete from *either* the end *or* the beginning of a concrete sequence, `collections.deque` gives you exactly what you need. This is not an arbitrary middle for insertion and deletion, but very often all you actually want is `.appendleft()` and `.popleft()` to accompany `.append()` and `.pop()`.

In some cases, `sortedcontainers` or `pyrsistent` may have closer to the performance characteristics you need, while still offering a *sequence* datatype. Generally, using these third-party containers is still only going to get you to $O(N \times \log N)$ rather than $O(N)$, but that remains strikingly better than $O(N^2)$.

Later in this chapter, in the section "Rolling Your Own Data Structures," I show an example where creating a custom data structure actually *can* make sense. My pure-Python implementation of `CountingTree` is able to do exactly the "insert into the middle" action that is described in this section, and remains relatively efficient. For this narrow and specific use case, my custom data structure is actually pretty good.

However, instead of reaching for the abovementioned collections—as excellent as each of them genuinely is—this problem is probably one in which you (or the developer before you) misunderstood what the underlying problem *actually* requires.

For example, a somewhat plausible reason you might *actually* want to keep an order for items is because they represent some sort of *priority* of actions to be performed or data to be processed. A wonderful data structure in which to maintain such priorities is simply a Python `dict`. A plausible way of using this fast data structure is to keep your "words" (per the earlier example) as keys, and their priority as values.

A priority is not exactly the same thing as an index position, but it *is* something that very quickly allows you to maintain a sequence for the data you wish to handle, while keeping insertion or deletion operations always at $O(1)$. This means, of course, that performing N such operations is $O(N)$, which is the best we might plausibly hope for. Constructing a sequence *at the end* of such operations is both cheap and easy, as the following example shows.

A collection of items with a million possible priorities

```
>>> from pprint import pprint
>>> from functools import partial
>>> priority = partial(randrange, 1, 1_000_000)
>>> words = {get_word():priority() for _ in range(100_000)}
>>> words_by_priority = sorted(words.items(), key=lambda p: p[1])
>>> pprint(words_by_priority[:10])
[('badland', 8),
 ('weakliest', 21),
```

```
 ('sowarry', 28),
 ('actinobiology', 45),
 ('oneself', 62),
 ('subpanel', 68),
 ('alarmedly', 74),
 ('marbled', 98),
 ('dials', 120),
 ('dearing', 121)]
>>> pprint(words_by_priority[-5:])
[('overslow', 999976),
 ('ironings', 999980),
 ('tussocked', 999983),
 ('beaters', 999984),
 ('tameins', 999992)]
```

It's possible—even likely—that the same priority occurs for multiple words, occasionally. It's also very uncommon that you *actually* care about *exactly* which order two individual items come in out of 100,000 of them. However, even with duplicated priorities, items are not dropped, they are merely ordered arbitrarily (but you could easily enough impose an order if you have a reason to).

Deleting items from the `words` data structure is just slightly more difficult than was `del words[n]` where it had been a list. To be safe, you'd want to do something like:

```
>>> for word in ['producibility', 'scrambs', 'marbled']:
...     if word in words:
...         print("Removing:", word, words[word])
...         del words[word]
...     else:
...         print("Not present:", word)
...
Not present: producibility
Removing: scrambs 599046
Removing: marbled 98
```

The extra `print()` calls and the `else` clause are just for illustration; presumably if this approach is relevant to your requirements, you would omit them:

```
>>> for word in ['producibility', 'scrambs', 'marbled']:
...     if word in words:
...         del words[word]
```

This approach remains fast and scalable, and is quite likely much closer to the actual requirements of your software than was misuse of a list.

7.3 Strings Are Iterables of Strings

Strings in Python are strange objects. They are incredibly useful, powerful, and well designed. But they are still strange. In many ways, strings are *scalar* objects. They are immutable and hashable, for example. We usually think of a string as a *single value*, or equivalently call it *atomic*.

However, at the same time, strings are iterable, and every item in their iteration is also a string (which is itself iterable). This oddity often leads to mistakes when we wish to decompose or flatten nested data. Sometimes in related contexts as well, as shown in the following example.

Naive attempt at `flatten()` function

```
>>> def flatten(o, items=[]):
...     try:
...         for part in o:
...             flatten(part, items)
...     except TypeError:
...         items.append(o)
...     return items
```

If you prefer LBYL (look before you leap) to EAFP (easier to ask forgiveness than permission) you could write this as follows.

Naive attempt at `flatten2()` function

```
>>> from collections.abc import Iterable
>>> def flatten2(o, items=[]):
...     if isinstance(o, Iterable):
...         for part in o:
...             flatten2(part, items)
...     else:
...         items.append(o)
...     return items
```

Either way, these are perfectly sensible functions to take a nested data structure with scalar leaves, and return a linear sequence from them. These first two functions return a concrete list, but they could equally well be written as a generator function such as the following.

Naive attempt at `flatten_gen` function

```
>>> def flatten_gen(o):
...     if isinstance(o, Iterable):
...         for part in o:
...             yield from flatten_gen(part)
```

```
...     else:
...         yield o
```

Using this function often produces what we'd like:

```
>>> nested = [
...     (1, 2, 3),
...     {(4, 5, 6), 7, 8, frozenset([9, 10, 11])},
...     [[[12, 13], [14, 15], 16], 17, 18]
... ]
>>> flatten(nested, [])                                  # ❶
[1, 2, 3, 8, 9, 10, 11, 4, 5, 6, 7, 12, 13, 14, 15, 16, 17, 18]
>>> flatten2(nested, [])                                 # ❶
[1, 2, 3, 8, 9, 10, 11, 4, 5, 6, 7, 12, 13, 14, 15, 16, 17, 18]
>>> for item in flatten_gen(nested):
...     print(item, end=" ")
... print()
1 2 3 8 9 10 11 4 5 6 7 12 13 14 15 16 17 18
```

❶ To avoid mutable-default issues, pass in initial `items` to expand.

In the examples, the iterable but unordered set in the middle happens to yield the `frozenset` first, although it is listed last in the source code. You are given no guarantee about whether that accident will hold true in a different Python version, or even on a different machine or different run.

This all breaks down terribly when strings are involved. Because strings are iterable, every item in their iteration is also a string (which is itself iterable).

How strings break recursion

```
>>> import sys
>>> sys.setrecursionlimit(10)                            # ❶
>>> flatten(nested, [])
[1, 2, 3, 8, 9, 10, 11, 4, 5, 6, 7, 12, 13, 14, 15, 16, 17, 18]
>>> flatten('abc', [])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in flatten
  File "<stdin>", line 4, in flatten
  File "<stdin>", line 4, in flatten
  [Previous line repeated 6 more times]                  # ❷
RecursionError: maximum recursion depth exceeded
```

❶ The same breakage occurs with a default depth of 1000, it just shows more lines of traceback before doing so.

❷ Recent `python` shells simplify many tracebacks, but `ipython` does not by default.

Using `flatten2()` or `flatten_gen()` will produce very similar tracebacks and exceptions (small details of their tracebacks vary, but `RecursionError` is the general result in all cases). If strings are nested within other data structures rather than top level, the result is essentially the same:

```
>>> flatten2(('a', ('b', 'c')), [])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in flatten2
  File "<stdin>", line 4, in flatten2
  File "<stdin>", line 4, in flatten2
  [Previous line repeated 2 more times]
  File "<stdin>", line 2, in flatten2
  File "<frozen abc>", line 119, in __instancecheck__
RecursionError: maximum recursion depth exceeded in comparison
```

The solution to these issues is to add some unfortunate ugliness to code, as in the examples shown here.

Ugly but safe flatten function

```
>>> def flatten_safe(o, items=[]):
...     if isinstance(o, (str, bytes)):             # ❶
...         items.append(o)
...     elif isinstance(o, Iterable):
...         for part in o:
...             flatten_safe(part, items)
...     else:
...         items.append(o)
...     return items
...
>>> flatten_safe(('a', ['b', 'c'], {'dee'}), [])
['a', 'b', 'c', 'dee']
>>> flatten_safe(nested, [])
[1, 2, 3, 8, 9, 10, 11, 4, 5, 6, 7, 12, 13, 14, 15, 16, 17, 18]

>>> flatten([b'abc', [100, 101]], [])
[97, 98, 99, 100, 101]                              # ❷
>>> flatten_safe([b'abc', [100, 101]], [])
[b'abc', 100, 101]                                  # ❸
```

❶ `bytes` has a slightly different but also annoying issue.

❷ No exception occurred, but probably not what you wanted

❸ Most likely the behavior you were hoping for

It would be nice if Python had a virtual parent class like `collections.abc.NonAtomicIterable`. Unfortunately, it does not, and it *cannot* without substantially changing the semantics of Python strings. Or perhaps, less intrusively, `isinstance()` could conceivably check for something else beyond the presence of an `.__iter__()` when deciding whether an object is an instance of this hypothetical `NonAtomicIterable` interface.

For the current Python version, 3.12 as of this writing, special case checking for string-ness is really the only approach available to handle the dual composite/atomic nature of strings.

7.4 (Often) Use `enum` Rather Than `CONSTANT`

The `enum` module was added to Python 3.4, and has grown incremental new capabilities in several versions since then. Prior to that module being added, but also simply because some developers are more accustomed to languages such as bash, C, and Java,[3] it is not uncommon to see capitalized names (usually defined at a module scope) used as constants in Python code.

Informal enumerations using capitalization

```
"This module works with sprites having colors and shapes"

RED = "RED"
GREEN = "GREEN"
BLUE = "BLUE"

CIRCLE, SQUARE, TRIANGLE = range(3)

class Sprite:
    def __init__(self, shape, color):
        self.shape = shape
        self.color = color

    # ... other methods

def process(sprite):
    if sprite.shape == TRIANGLE and sprite.color == RED:
        red_triangle_action(sprite)
    elif something_else:
        # ... other processing
```

3. C, Java, Go, Rust, C#, TypeScript, and most programming languages also have enums of varying stripes. But the `CONSTANT` convention is nonetheless often seen in code in those languages.

In a highly dynamic language like Python, we *can* potentially redefine "constants" since the capitalization is only a convention rather than in the syntax or semantics of the language. If some later line of the program redefines `SQUARE = 2`, buggy behavior is likely to emerge. More likely is that some other module that gets imported has redefined `SQUARE` to something other than the expectation of the current module. This risk is minimal if imports are within namespaces, but `from othermod import SQUARE, CUBE, TESSERACT` is not necessarily unreasonable to have within the current module.

Programs written like the preceding one are not necessarily broken, and not even necessarily mistakes, but it is certainly more elegant to use enums for constants that come in sets.

Using enums for sets of alternatives

```
>>> from enum import Enum
>>> Color = Enum("Color", ["RED", "GREEN", "BLUE"])
>>> class Shape(Enum):
...     CIRCLE = 0
...     SQUARE = 1
...     TRIANGLE = 2
...
>>> my_sprite = Sprite(Shape.TRIANGLE, Color.RED)
>>> def process(sprite):
...     if sprite.shape == Shape.TRIANGLE and sprite.color ==
Color.RED:
...         print("It is a red triangle")
...     elif something_else:
...         pass
...
>>> process(my_sprite)
It is a red triangle
>>> Color.RED = 2
Traceback (most recent call last):
[...]
AttributeError: cannot reassign member 'RED'
```

It's not *impossible* to get around the protection that an `Enum` provides, but you have to work quite hard to do so rather than break it inadvertently. In effect, the attributes of an enum are *read-only*. Therefore, reassigning to an immutable attribute raises an exception.

There are also "constants" that are not alternatives, but simply values; these likewise cannot actually be enforced as constants in Python. Enums might still be reasonable namespaces with slightly more enforcement against changes than modules have.

Overwriting constants

```
>>> import math
>>> radius = 2
>>> volume = 4/3 * math.pi * radius**3
>>> volume                                  # ❶
33.510321638291124
>>> math.pi = 3.14                          # ❷
>>> 4/3 * math.pi * radius**3
33.49333333333333
>>> from math import pi
>>> 4/3 * pi * radius**3
33.49333333333333
>>> pi = 3.1415                             # ❸
>>> 4/3 * pi * radius**3
33.50933333333333
```

❶ As good as we get with 64-bit floating point numbers

❷ Monkeypatching a bad approximation of pi

❸ A somewhat less bad approximation of pi

Using enums to "enforce" value consistency

```
>>> from enum import Enum
>>> import math
>>> class Math(Enum):
...     pi = math.pi
...     tau = math.tau
...     e = math.e
...
>>> radius = 2
>>> Math.pi.value
3.141592653589793
>>> 4/3 * Math.pi.value * radius**3
33.510321638291124
>>> math.pi = 3
>>> 4/3 * Math.pi.value * radius**3
33.510321638291124
>>> Math.pi.value = 3
Traceback (most recent call last):
[...]
AttributeError: <enum 'Enum'> cannot set attribute 'value'
```

This usage doesn't really use `Enum` as a way of enumerating distinct values, but it *does* carry with it a modest protection of "read-only" values.

7.5 Learn Less Common Dictionary Methods

Dictionaries are a wonderful data structure that in many ways make up the heart of Python. Internally, most objects, including modules, are defined by their dictionaries.

The sometimes overlooked method `dict.get()` was discussed in Chapter 3, *A Grab Bag of Python Gotchas*, but `dict`s also have a few other methods that are often overlooked, even by experienced Python programmers. As with a number of other mistakes throughout this book, the mistake here is simply one of ignorance or forgetfulness; the result is not usually broken code, but rather just code that is less fast, elegant, and expressive than it might be.

7.5.1 The Dictionaries Defining Objects

This subsection is a digression into Python's internal mechanisms. Feel free to skip it for the actual pitfall; or read it to understand Python a little bit better.

You can use Python for a long time without ever needing to think about the dictionaries at the heart of most non-`dict` objects. There are some exceptions, but many Python objects have a `.__dict__` attribute to store the dictionary providing its capabilities and behaviors.

Let's look at a couple examples.

Module dictionaries

```
>>> import re
>>> type(re.__dict__)
<class 'dict'>
>>> for key in re.__dict__.keys():
...     print(key, end=" ")
...
__name__ __doc__ __package__ __loader__ __spec__ __path__ __file__
__cached__ __builtins__ enum _constants _parser _casefix _compiler
functools __all__ __version__ NOFLAG ASCII A IGNORECASE I LOCALE L
UNICODE U MULTILINE M DOTALL S VERBOSE X TEMPLATE T DEBUG RegexFlag
error match fullmatch search sub subn split findall finditer compile
purge template _special_chars_map escape Pattern Match _cache
_MAXCACHE _compile _compile_repl _expand _subx copyreg _pickle Scanner
```

The various functions and constants in a module are simply its dictionary. Built-in types usually use a slightly different dictionary-like object.

Dictionaries of basic types

```
>>> for typ in (str, int, list, tuple, dict):
...     print(typ, type(typ.__dict__))
...
<class 'str'> <class 'mappingproxy'>
<class 'int'> <class 'mappingproxy'>
<class 'list'> <class 'mappingproxy'>
<class 'tuple'> <class 'mappingproxy'>
<class 'dict'> <class 'mappingproxy'>

>>> int.__dict__["numerator"]
<attribute 'numerator' of 'int' objects>
>>> (7).__class__.__dict__["numerator"]
<attribute 'numerator' of 'int' objects>
>>> (7).numerator
7
```

Custom classes also continue this pattern (their instances have either `.__dict__` or `.__slots__`, depending on how they are defined).

Dictionaries defining classes (and instances)

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def from_origin(self):
...         from math import sqrt
...         return sqrt(self.x**2 + self.y**2)
...
>>> point = Point(3, 4)
>>> point.from_origin()
5.0
>>> type(Point.__dict__)
<class 'mappingproxy'>
>>> type(point.__dict__)
<class 'dict'>
>>> Point.__dict__.keys()
dict_keys(['__module__', '__init__', 'from_origin', '__dict__',
'__weakref__', '__doc__'])
>>> point.__dict__
{'x': 3, 'y': 4}
```

7.5.2 Back to Our Regularly Scheduled Mistake

The Method `.setdefault()`

Of all the useful methods of dictionaries, the one I personally forget the most often is `dict.setdefault()`. I have written code like this embarrassingly often:

```
>>> point = {"x": 3, "y": 4}
>>> if 'color' in point:
...     color = point["color"]
... else:
...     color = "lime green"
...     point["color"] = color
...
>>> point
{'x': 3, 'y': 4, 'color': 'lime green'}
```

All the while, I *should* have simply written:

```
>>> point = {"x": 3, "y": 4}
>>> color = point.setdefault("color", "lime green")
>>> color
'lime green'
>>> point
{'x': 3, 'y': 4, 'color': 'lime green'}
>>> point.setdefault("color", "brick red")
'lime green'
```

The first version works, but it uses five lines where one would be slightly faster and distinctly clearer.

The Method `.update()`

The method `dict.update()` is useful to avoid writing:

```
>>> from pprint import pprint
>>> features = {
...     "shape": "rhombus",
...     "flavor": "vanilla",
...     "color": "brick red"}
>>> for key, val in features.items():
...     point[key] = val
...
>>> pprint(point)
```

```
{'color': 'brick red',
 'flavor': 'vanilla',
 'shape': 'rhombus',
 'x': 3,
 'y': 4}
```

Prior to Python 3.9, the friendlier shortcut was:

```
>>> point = {"x": 3, "y": 4, "color": "chartreuse"}
>>> point.update(features)
>>> pprint(point)
{'color': 'brick red',
 'flavor': 'vanilla',
 'shape': 'rhombus',
 'x': 3,
 'y': 4}
```

But with recent Python versions, even more elegant versions are:

```
>>> point = {"x": 3, "y": 4, "color": "chartreuse"}
>>> point | features                                     # ❶
{'x': 3, 'y': 4, 'color': 'brick red', 'shape': 'rhombus',
'flavor': 'vanilla'}
>>> point
{'x': 3, 'y': 4, 'color': 'chartreuse'}
>>> point |= features                                    # ❷
>>> point
{'x': 3, 'y': 4, 'color': 'brick red', 'shape': 'rhombus',
'flavor': 'vanilla'}
```

❶ Create a new dictionary merging `features` with `point`.

❷ Equivalent to `point.update(features)`

The Methods `.pop()` and `.popitem()`

The methods `dict.pop()` and `dict.popitem()` are also easy to forget, but extremely useful when you need them. The former is useful when you want to find and remove a specific key; the latter is useful when you want to find and remove an unspecified key/value pair:

```
>>> point.pop("color", "gray")
'brick red'
```

```
>>> point.pop("color", "gray")
'gray'
>>> point
{'x': 3, 'y': 4, 'shape': 'rhombus', 'flavor': 'vanilla'}
```

That is much friendlier than:

```
>>> point = {'x': 3, 'y': 4, 'color': 'brick red',
             'shape': 'rhombus', 'flavor': 'vanilla'}
>>> if "color" in point:
...     color = point["color"]
...     del point["color"]
... else:
...     color = "gray"
... color
'brick red'
```

Likewise, to get an arbitrary item in a dictionary, `dict.popitem()` is very quick and easy. This is often a way to process the items within a dictionary, leaving an empty dictionary when processing is complete. Since Python 3.7, "arbitrary" is always LIFO (last-in, first-out) because dictionaries maintain insertion order. Depending on your program flow, insertion order may or may not be obvious or reproducible; but you are guaranteed *some* order for successive removal:

```
>>> point = {'x': 3, 'y': 4, 'color': 'brick red',
             'shape': 'rhombus', 'flavor': 'vanilla'}
>>> while point and (item := point.popitem()):
...     print(item)
...
('flavor', 'vanilla')
('shape', 'rhombus')
('color', 'brick red')
('y', 4)
('x', 3)
>>> point
{}
```

Making Copies

Another often-overlooked method is `dict.copy()`. However, I tend to feel that this method is usually properly overlooked. The copy made by this method is a *shallow* copy, so

any mutable values might still be changed indirectly, leading to subtle and hard-to-find bugs. Chapter 2, *Confusing Equality with Identity*, is primarily about exactly this kind of mistake.

Most of the time, a much better place to look is `copy.deepcopy()`. For example:

```
>>> d1 = {"foo": [3, 4, 5], "bar": {6, 7, 8}}
>>> d2 = d1.copy()
>>> d2["foo"].extend([10, 11, 12])
>>> del d2["bar"]
>>> d1
{'foo': [3, 4, 5, 10, 11, 12], 'bar': {8, 6, 7}}
>>> d2
{'foo': [3, 4, 5, 10, 11, 12]}
```

This is confusing, and pretty much a bug magnet. Much better is:

```
>>> from copy import deepcopy
>>> d1 = {"foo": [3, 4, 5], "bar": {6, 7, 8}}
>>> d2 = deepcopy(d1)
>>> d2["foo"].extend([10, 11, 12])
>>> del d2["bar"]
>>> d1
{'foo': [3, 4, 5], 'bar': {8, 6, 7}}
>>> d2
{'foo': [3, 4, 5, 10, 11, 12]}
```

Dictionaries are an amazingly rich data structure in Python. As well as the usual efficiency that hash maps or key/value stores have in most programming languages, Python provides a moderate number of well-chosen "enhanced" methods. In principle, if dictionaries only had key/value insertion, key deletion, and a method to list keys, that would suffice to *do everything* the underlying data structure achieves. However, your code can be much cleaner and more intuitive with strategic use of the additional methods discussed.

7.6 JSON Does Not Round-Trip Cleanly to Python

A Python developer can be tempted into mistakenly thinking that arbitrary Python objects can be serialized as JSON, and relatedly that objects that can be serialized are necessarily deserialized as equivalent objects.

7.6.1 Some Background on JSON

In the modern world of microservices and "cloud-native computing," Python often needs to serialize and deserialize JavaScript Object Notation (JSON) data. Moreover, JSON doesn't only occur in the context of message exchange between small cooperating services, but is also used as a storage representation of certain structured data. For example, GeoJSON and the related TopoJSON, or JSON-LD for ontology and knowledge graph data, are formats that utilize JSON to encode domain-specific structures.

In surface appearance, JSON looks very similar to Python numbers, strings, lists, and dictionaries. The similarity is sufficient that for many JSON strings, simply writing `eval(json_str)` will deserialize a string to a valid Python object; in fact, this will *often* (but certainly not *always*) produce the same result as the correct approach of `json.loads(json_str)`. JSON looks *even more* similar to native expressions in JavaScript (as the name hints), but even there, a few valid JSON strings cannot be deserialized (meaningfully) into JavaScript.

While superficially `json.loads()` performs a similar task as `pickle.loads()`, and `json.dumps()` performs a similar task as `pickle.dumps()`, the JSON versions do distinctly *less* in numerous situations. The "type system" of JSON is less rich than is that of Python. For a large subset of all Python objects, including (deeply) nested data structures, this invariant holds:

```
obj == pickle.loads(pickle.dumps(obj))
```

There are exceptions here. File handles or open sockets cannot be sensibly serialized and deserialized, for example. But most *data structures*, including custom classes, survive this round-trip perfectly well.

In contrast, this "invariant" is very frequently violated:

```
obj = json.loads(json.dumps(obj))
```

JSON is a very useful format in several ways. It is (relatively) readable pure text; it is highly interoperable with services written in other programming languages with which a Python program would like to cooperate; deserializing JSON does not introduce code execution vulnerabilities.

Pickle (in its several protocol versions) is also useful. It is a binary serialization format that is more compact than text. Or specifically, it is protocol 0, 1, 2, 3, 4, or 5, with each successive version being improved in some respect, but all following that characterization. Almost all Python objects can be serialized in a round-trippable way using the `pickle` module. However, none of the services you might wish to interact with, written in JavaScript, Go, Rust, Kotlin, C++, Ruby, or other languages, has any idea what to do with Python pickles.

7.6.2 Data That Fails to Round-Trip

In the first place, JSON only defines a few datatypes. These are discussed in RFC 8256 (https://datatracker.ietf.org/doc/html/rfc8259), ECMA-404 (https://www.ecma-international.org/publications-and-standards/standards/ecma-404/), and ISO/IEC 21778:2017 (https://www.iso.org/standard/71616.html). Despite having "the standard" enshrined by several standards bodies in not-quite-identical language, these standards are equivalent.

We should back up for a moment. I've now twice claimed—a bit incorrectly—that JSON has a limited number of datatypes. In reality, JSON has zero datatypes, and instead is, strictly speaking, only a definition of a syntax with no semantics whatsoever. As RFC 8256 defines the highest level of its BNF (Backus–Naur form):

```
value ::= false | null | true | object | array | number | string
```

Here `false`, `null`, and `true` are literals, while object, array, number, and string are textual patterns. To simplify, a JSON object is like a Python dictionary, with curly braces, colons, and commas. An array is like a Python list, with square brackets and commas. A number can take a number of formats, but the rules are *almost* the same as what defines valid Python numbers. Likewise, JSON strings are *almost* the same as the spelling of Python strings, but always with double quotation marks. Unicode numeric codes are mostly the same between JSON and Python (edge cases concern very obscure surrogate pair handling).

Let's take a look at some edge cases. The Python standard library module `json` "succeeds" in two cases by producing output that is *not actually* JSON:

```
>>> import json
>>> import math
>>> print(json.dumps({"nan": math.nan}))          # ❶
{"nan": NaN}
>>> print(json.dumps({"inf": math.inf}))
{"inf": Infinity}
>>> json.loads(json.dumps({'nan': math.nan}))      # ❷
{'nan': nan}
>>> json.loads(json.dumps({'inf': math.inf}))
{'inf': inf}
```

❶ The result of `json.dumps()` is a string; printing it just removes the extra quotes in the echoed representation.

❷ Neither NaN nor Infinity (under any spelling variation) are in the JSON standards.

In some sense, this behavior is convenient for Python programmers, but it breaks compatibility with (many) consumers of these serializations in other programming languages. We can enforce more strictness with `json.dumps(obj, allow_nan=False)`,

which would raise `ValueError` in the preceding lines. However, *some* other libraries in *some* other programming languages also allow this almost-JSON convention.

Depending on what you mean by "round-trip," you might say this succeeded. Indeed it does strictly within Python itself; but it fails when the *round-trip* involves talking with a service written in a different programming language, and it talking back. Let's look at some failures within Python itself. The most obvious cases are in Python's more diverse collection types.

Not-quite round-tripping collections with JSON

```
>>> from collections import namedtuple
>>> Person = namedtuple("Person", "first last body_temp")
>>> david = Person("David", "Mertz", "37°C")
>>> vector1 = (4.6, 3.2, 1.5)
>>> vector2 = (9.8, -1.2, 0.4)
>>> obj = {1: david, 2: [vector1, vector2], 3: True, 4: None}
>>> obj
{1: Person(first='David', last='Mertz', body_temp='37°C'),
2: [(4.6, 3.2, 1.5), (9.8, -1.2, 0.4)], 3: True, 4: None}

>>> print(json.dumps(obj))
{"1": ["David", "Mertz", "37\u2103"], "2": [[4.6, 3.2, 1.5],
[9.8, -1.2, 0.4]], "3": true, "4": null}
>>> json.loads(json.dumps(obj))
{'1': ['David', 'Mertz', '37°C'], '2': [[4.6, 3.2, 1.5],
[9.8, -1.2, 0.4]], '3': True, '4': None}
```

In JSON, Python's `True` is spelled `true`, and `None` is spelled `null`, but those are entirely literal spelling changes. Likewise, the Unicode character DEGREE CELSIUS can perfectly well live inside a JSON string (or any Unicode character other than a quotation mark, *reverse solidus*/backslash, and the control characters U+0000 through U+001F). For some reason, Python's `json` module decided to substitute with the numeric code, but such has no effect on the round-trip.

What got lost was that some data was inside a `namedtuple` called `Person`, and other data was inside tuples. JSON only has arrays, that is, things in square brackets. The general "meaning" of the data is still there, but we've lost important type information.

Moreover, in the serialization, only strings are permitted as object keys, and hence our valid-in-Python integer keys were converted to strings. However, this is lossy since a Python dictionary could, in principle (but it's not great code), have both string and numeric keys:

```
>>> json.dumps({1: "foo", "1": "bar"})
'{"1": "foo", "1": "bar"}'
>>> json.loads(json.dumps({1: "foo", "1": "bar"}))
{'1': 'bar'}
```

Two or three things conspired against us here. Firstly, the JSON specification doesn't prevent duplicate keys from occurring. Secondly, the integer `1` is converted to the string `"1"` when it becomes JSON. And thirdly, Python dictionaries always have unique keys, so the second try at setting the `"1"` key overwrote the first try.

Another somewhat obscure edge case is that JSON itself can validly represent numbers that Python does not support:

```
>>> json_str = '[1E400, 3.14159265358979323846264338327 9]'
>>> json.loads(json_str)
[inf, 3.141592653589793]
```

This is not a case of crashing, nor failing to load numbers at all. But rather, one number overflows to infinity since it is too big for float64, and the other is approximated to fewer digits of precision than are provided.

A corner edge case is that JSON numbers that "look like Python integers" actually get cast to `int` rather than `float`:

```
>>> json_str = f'{"7"*400}'                          # ❶
>>> val = json.loads(json_str)
>>> math.log10(val)
399.8908555305749
>>> type(val)
<class 'int'>
```

❶ A string of four hundred `"7"`s in a row

However, since few other programming languages or architectures you might communicate with will support, for example, float128 either, the best policy is usually to stick with numbers float64 can represent.

7.7 Rolling Your Own Data Structures

This section covers a nuanced issue (and a long one). Readers who have come out of a college data structures course, or read a good book on the topic,[4] have learned of many powerful data structures that are neither within Python's standard library nor in the prominent third-party libraries I discuss in various parts of this book. Some of these include treaps, k-d trees, R-trees, B-trees, Fibonacci heaps, tries (prefix tree), singly-, doubly-, and multiply-linked lists, heaps, graphs, bloom filters, cons cells, and dozens of others.

4. Perhaps even Donald Knuth's "bible": *The Art of Computer Programming* (various editions among its current five volumes; but especially the 3rd edition of volume 1, Addison-Wesley, 1997).

The choice of which data structures to include as built-ins, or in the standard library, is one that language designers debate, and which often leads to in-depth discussion and analysis. Python's philosophy is to include a relatively minimal, but extremely powerful and versatile, collection of primitives with `dict`, `list`, `tuple`, `set`, `frozenset`, `bytes`, and `bytearray` in `__builtins__` (arguably, `complex` is a simple data structure as well). Modules such as `collections`, `queue`, `dataclasses`, `enum`, `array`, and a few others peripherally, include other data structures, but even there the number is much smaller than for many programming languages.

A clear contrast with Python, in this regard, is Java. Whereas Python strives for simplicity, Java strives to include every data structure users might ever want within its standard library (i.e., the `java.util` namespace). Java has hundreds of distinct data structures included in the language itself. For Pythonic programmers, this richness of choice largely leads only to "analysis paralysis" (https://en.wikipedia.org/wiki/Analysis_paralysis). Choosing among so many only-slightly-different data structures imposes a large cognitive burden, and the final decision made (after greater work) often remains sub-optimal. Giving someone more hammers can sometimes provide little other than more ways for them to hit their thumb.

Note A data structure that hasn't quite made it into Python

A really lovely example of the design discussions that go into Python is in PEP 603 (https://peps.python.org/pep-0603/), and the python-dev mailing list and Discourse thread among core developers that followed this PEP. The proposal of a new data structure has not been entirely rejected since September 2019, but it also has not been accepted so far.

Internally, CPython utilizes a data structure called a Hash Array Mapped Trie (HAMT). This isn't used widely, but there are specific places in the C code implementing CPython where it is the best choice. A HAMT is a kind of immutable dictionary, in essence. Since this structure *already exists* in the CPython codebase, it would be relatively easy to expose it under a name like `frozenmap` or `frozendict`; this would parallel the existing `frozenset` and `tuple` in being the "immutable version of built-in mutable collections."

HAMT is clearly a useful data structure for some purposes. If it were not, the very talented CPython developers would not have utilized it. However, the current tide of opinion among these developers is that HAMT is not general purpose enough to add to the cognitive load of tens of millions of Python developers who *probably won't need it*.

7.7.1 When Rolling Your Own Is a Bad Idea

Writing any of the data structures mentioned thus far is comparatively easy in Python. Doing so is often the subject of college exams and software engineering interviews, for example. Doing so is also *usually* a bad idea for most software tasks you will face. When

you reach quickly for an opportunity to use one of these data structures you have learned—each of which genuinely *does* have concrete advantages in specific contexts—it often reflects an excess of cleverness and eagerness more than it does good design instincts.

A reality is that Python itself is a relatively slow bytecode interpreter. Unlike compiled programming languages, including just-in-time (JIT) compiled languages, which produce machine-native instructions, CPython is a giant bytecode dispatch loop. Every time an instruction is executed, many levels of indirection are needed, and basic values are all relatively complex wrappers around their underlying data (remember all those methods of datatypes that you love so much?).

Note Python implementations

Several alternative implementations of Python exist besides CPython. In particular, a few of these include JIT compilation. The most widely used such implementation is PyPy (https://www.pypy.org/), which JITs everything, and does so remarkably well. Its main drawbacks are that it has fallen behind version compatibility with CPython, and that using compiled extensions created for CPython can encounter overheads that reduce the speed advantages greatly.

Less widely used attempts at JIT Python interpreters include Pyston (https://github.com/pyston/pyston), Cinder (https://github.com/facebookincubator/cinder), and Pyjion (https://github.com/tonybaloney/Pyjion). While all of these have good ideas within them—and all are derived from CPython source code (unlike PyPy)—these open source projects still largely have a focus within the private companies that developed them. Those are Dropbox, Meta, and Microsoft, respectively (Alphabet—i.e., Google—subsidiary DeepMind abandoned its similar S6 project).

Reservations mentioned, it is well possible that a custom data structure developed as pure Python but used in a JIT interpreter will achieve the speed and flexibility advantages that those developed in compiled languages have.

Accompanying the fact that Python is relatively slow, most of the built-in and standard library data structures you might reach for are written in highly optimized C. Much the same is true for the widely used library NumPy, which has a chapter of its own.

On the one hand, custom data structures such as those mentioned can have significant big-O complexity advantages over those that come with Python.[5] On the other hand, these advantages need to be balanced against what is usually a (roughly) constant

5. See note 1 on page 156.

multiplicative disadvantage to pure-Python code. That is to say, implementing the identical data structure purely in Python is likely to be 100x, or even 1000x, slower than doing so in a well-optimized compiled language like C, C++, Rust, or Fortran. At some point as a dataset grows, big-O dominates any multiplicative factor, but often that point is well past the dataset sizes you actually care about.

Plus, writing a new data structure requires actually writing it. This is prone to bugs, takes developer time, needs documentation, and accumulates technical debt. In other words, doing so might very well be a mistake.

7.7.2 When Rolling Your Own Is a Good Idea

Taking all the warnings and caveats of the first subsection of this discussion into account, there remain many times when *not* writing a custom data structure is its own mistake. Damned if you do, damned if you don't, one might think. But the real issue is more subtle; it's a mistake to make a poor judgment about which side of this decision to choose.

I present in the following subsections a "pretty good" specialized data structure that illustrates both sides. This example is inspired by the section "Deleting or Adding Elements to the Middle of a List" earlier in this chapter. To quickly summarize that section: Inserting into the middle of a Python list is inefficient, but doing so is very often a matter of *solving the wrong problem*.

For now, however, let's suppose that you genuinely *do need* to have a data structure that is concrete, strictly ordered, indexable, iterable, and into which you need to insert new items in varying middle positions. There simply is not any standard library or widely used Python library that gives you exactly this. Perhaps it's worth developing your own.

Always Benchmark When You Create a Data Structure

Before I show you the code *I* created to solve this specific requirement, I want to reveal the "punch line" by showing you performance. A testing function shows the general behavior we want to be performant.

The `insert_many()` function that exercises our use case

```
from random import randint, seed
from get_word import get_word                          # ❶
def insert_many(Collection, n, test_seed="roll-your-own"):
    seed(test_seed)                                    # ❷
    collection = Collection()
    for _ in range(n):
        collection.insert(randint(0, len(collection)), get_word())
    return collection
```

❶ The `get_word()` function available at this book's website is used in many examples. It simply returns a different word each time it is called.

❷ Using the same random seed assures that we do *exactly* the same insertions for each collection type.

The testing function performs however many insertions we ask it to, and we can time that:

```
>>> from binary_tree import CountingTree

>>> %timeit insert_many(list, 100)
92.9 µs ± 742 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops
each)
>>> %timeit insert_many(CountingTree, 100)
219 µs ± 8.17 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
each)

>>> %timeit insert_many(list, 10_000)
13.9 ms ± 193 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> %timeit insert_many(CountingTree, 10_000)
38 ms ± 755 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

>>> %timeit insert_many(list, 100_000)
690 ms ± 5.84 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
>>> %timeit insert_many(CountingTree, 100_000)
674 ms ± 20.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

>>> %timeit insert_many(list, 1_000_000)
1min 5s ± 688 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
>>> %timeit insert_many(CountingTree, 1_000_000)
9.72 s ± 321 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Without having yet said just what a `CountingTree` is, I can say that I spent more time ironing out the bugs in my code than I entirely want to admit. It's not a large amount of code, as you'll see, but the details are futzy.

Notable points are that even though I've created a data structure optimized for *exactly* this task, it does worse than `list` for 100 items. `CountingTree` does worse than `list` for 10,000 items also, even by a slightly larger margin than for 100. However, my custom data structure pulls ahead *slightly* for 100,000 items; and then *hugely* so for a million items.

It would be painful to use `list` for the million-item sequence, and increasingly worse if I needed to do even more `collection.insert()` operations.

Performing Magic in Pure Python

The source code for `binary_tree.py` is available at the book's website (https://gnosis.cx/better). But we will go through most of it here. The basic idea behind my *Counting Binary Tree* data structure is that I want to keep a binary tree, but I also want each node to keep a count of the total number of items within it and all of its descendants. Unlike some other tree data structures, we specifically *do not* want to order the node values by their inequality comparison, but rather to maintain each node exactly where it is inserted.

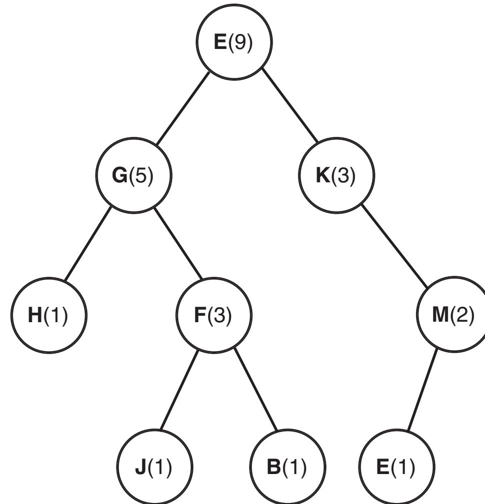

Figure 7.1 A graph of a Counting Binary Tree.

In Figure 7.1, each node contains a value that is a single letter; in parentheses we show the *length* of each node with its subtree. Identical values can occur in multiple places (unlike, e.g., for a set or a dictionary key). Finding the `len()` of this data structure is a matter of reading a single attribute. But having this length available is what guides insertions.

It is very easy to construct a *sequence* from a tree. It is simply a matter of choosing a deterministic rule for how to order the nodes. For my code, I chose to use *depth-first, left-to-right*; that's not the only possible choice, but it is an obvious and common one. In other words, every node value occurs at exactly one position in the sequence, and every sequence position (up to the length) is occupied by exactly one value. Since our use case is approximately random insertion points for new items, no extra work is needed for rebalancing or enforcing any other invariants.

The code shown *only* implements insertions, our stated use case. A natural extension to the data structure would be to implement deletions as well. Or changing values at a given position. Or other capabilities that lists and other data structures have. Most of those capabilities would remain inexpensive, but details would vary by the specific operation, of course.

The basic implementation of Counting Binary Tree

```
class CountingTree:
    def __init__(self, value=EMPTY):
        self.left = EMPTY
        self.right = EMPTY
        self.value = value
        self.length = 0 if value is EMPTY else 1
```

```
    def insert(self, index: int, value):
        if index != 0 and not 0 < index <= self.length:
            raise IndexError(
                f"CountingTree index {index} out of range")

        if self.value is EMPTY:
            self.value = value
        elif index == self.length:
            if self.right is EMPTY:
                self.right = CountingTree(value)
            else:
                self.right.insert(
                    index - (self.left.length + 1), value)
        elif index == 0 and self.left is EMPTY:
            self.left = CountingTree(value)
        else:
            if index > self.left.length:
                self.right.insert(
                    index - (self.left.length + 1), value)
            else:
                self.left.insert(index, value)

        self.length += 1
```

This much is all we actually need to run the benchmarks performed here. Calling `CountingTree.insert()` repeatedly creates trees much like that in the figure. The `.left` and `.right` attributes at each level might be occupied by the sentinel `EMPTY`, which the logic can utilize for nodes without a given child.

It's useful also to define a few other behaviors we'd like a collection to have.

Additional methods within Counting Binary Tree

```
    def append(self, value):
        self.insert(len(self), value)

    def __iter__(self):
        if self.left is not EMPTY:
            yield from self.left
        if self.value is not EMPTY:
            yield self.value
        if self.right is not EMPTY:
            yield from self.right

    def __repr__(self):
        return f"CountingTree({list(self)})"
```

```
    def __len__(self):
        return self.length

    def tree(self, indent=0):
        print(f"{'· '*indent}{self.value}")
        if self.left is not EMPTY or self.right is not EMPTY:
            self.left.tree(indent+1)
            self.right.tree(indent+1)
```

These other methods largely just build off of `.insert()`. A `CountingBinaryTree` is iterable, but along with `.__iter__()` it would be natural to define `.__getitem__()` or `.__contains__()` to allow use of square bracket indexing and the `in` operator. These would be straightforward.

For the `.tree()` method we need our sentinel to have a couple specific behaviors. This method is just for visual appeal in viewing the data structure, but it's nice to have.

The `EMPTY` sentinel

```
# Sentinel for an unused node
class Empty:
    length = 0

    def __repr__(self):
        return "EMPTY"

    def tree(self, indent=0):
        print(f"{'· '*indent}EMPTY")

EMPTY = Empty()
```

Observing the Behavior of Our Data Structure

By no means am I advocating the general use of this specific skeletal data structure implementation. It's shown merely to illustrate the general way you might go about creating something analogous for well-understood use cases and with a knowledge of the theoretical advantages of particular data structures. Let's look at a few behaviors, though:

```
>>> insert_many(CountingTree, 10)
CountingTree(['secedes', 'poss', 'killcows', 'unpucker',
'gaufferings', 'funninesses', 'trilingual', 'nihil', 'bewigging',
'reproachably'])
>>> insert_many(list, 10)                              # ❶
['secedes', 'poss', 'killcows', 'unpucker', 'gaufferings',
```

```
'funninesses', 'trilingual', 'nihil', 'bewigging', 'reproachably']

>>> ct = insert_many(CountingTree, 1000, "david")
>>> lst = insert_many(list, 1000, "david")
>>> list(ct) == lst                                # ❷
True

>>> insert_many(CountingTree, 9, "foobar").tree()   # ❸
loaf
· acknown
· · spongily
· · · saeculums
· · · EMPTY
· · EMPTY
· fecundities
· · EMPTY
· · input
· · · boddle
· · · · sots
· · · · shrifts
· · · EMPTY
```

❶ Insertions into `list` or `CountingTree` preserve the same order.

❷ Equivalence for some operations between `list` and `CountingTree`

❸ Display the underlying tree implementing the sequence.

The tree is fairly balanced, and sometimes a given subtree fills only one or the other of its left and right children. This balance would be lost if, for example, we always used `.append()` (it would degenerate to a singly-linked list).

7.7.3 Takeaways

This section has had a long discussion. The takeaway you should leave with isn't a simple one. The lesson is "be subtle and accurate in your judgments" about when to create and when to avoid creating custom data structures. It's not a recipe, but more vaguely an advocacy of a nuanced attitude.

As a general approach to making the right choice, I'd suggest following a few steps in your thinking:

1. Try implementing the code using a widely used, standard Python data structure.
2. Run benchmarks to find out if any theoretical sub-optimality *genuinely* matters for the use case your code is put to.
3. Research the wide range of data structures that exist in the world to see which, if any, are theoretically optimal for your use case.

4. Research whether someone else has already written a well-tested Python implementation of the less common data structure you are considering. Such a library might not be widely used simply because the niche it fulfills is relatively narrow. On the other hand, it is also easy to put partially developed, poorly tested, and buggy libraries on PyPI, conda-forge, GitHub, GitLab, Bitbucket, or other public locations.
5. Assuming you are writing your own after considering the preceding steps, create both tests and benchmarks either in conjunction with—or even before—the implementation of the data structure.
6. If your well-tested implementation of a new data structure makes your code better, ask your boss for a raise or a bonus… and then share the code with the Python community under an open source license.

7.8 Wrapping Up

Sometimes a powerful object method or general technique can also lead you in the wrong direction, even in seemingly ordinary uses. This wrong direction might cause bad complexity behavior; at times it might work for initial cases but then fail in cases you had not yet considered.

In this chapter we probed at some operations on lists—generally one of the best optimized and flexible data structures Python has—where a different data structure is simply better. We also looked at how recursive algorithms need to remember that strings are both scalar and iterable, which means they often need to be special-cased in program flow.

Two more mistakes in this chapter looked at "sins of omission" where a facility that may be less familiar provides a more convenient and more readable approach to common tasks. Specifically, two mistakes served as reminders of the `enum` module and of some of the less widely used methods of dictionaries.

In the penultimate mistake of this chapter, the capabilities and limitations of the widely used JSON format were explored. In particular, we saw how Python developers might forget the (relatively minor) lossiness of JSON representations of data.

The final mistake discussed is one of nuance and complex decision-making. Often, creating custom data structures is premature optimization; but at other times they can significantly improve your code. The (long) discussion provides some guidance about making this choice wisely.

This page intentionally left blank

Index

Note: Italic letters following page numbers refer to special formats—*f* for figures, *n* for footnotes, and *t* for tables.

A

append operation, 11, 56
Applied Cryptography: Protocols, Algorithms, and Source Code in C (Schneier), 189
`approxEqualNums()`, 220–221
`array.array` data structure, 147–148, 149–150
The Art of Computer Programming (Knuth), 225
`AssertionError`, 215
assertions
  as safeguards, 212
  sample algorithm, 212–213
associativity, mathematical, 224
async event loops, 242
`async` keyword, 242
`asyncio` module, 242
atomic groups, Python feature, 126
authentication, 203
averaging, floating point numbers, 226
`await` keyword, 242

B

backtracking
  exponential behavior, 124
  solutions to, 126
  visualizing, 125
baseline iteration, 9
benchmarking, 181–182, 184–185
big-O complexity
  and custom data structures, 180–181
  decorate-sort-undecorate pattern for, 61
  and lazy computation solutions, 90–91
  and list insertion/deletion operations, 161
  notation explained, 156*n*
  and runtime behavior, 52
binary format, 147
bit width, 227, 228
BNF (Backus–Naur form), of JSON, 176
Boolean values
  checking for, 25–28
  and three-valued logic, 27–28
"Borg idiom," 29
Borges, Jorge Luís, 232*n*
brute-force attacks, defense against, 191
bugs
  caused by Boolean overchecking, 25
  and writing new data structures, 81
built-in (`builtin`) functions
  exception hierarchy, 47*f*
  overriding names in, 71–75
`bytearray`
  and append operations, 11
  as data structure for mutable sequences, 147–149
  iterable and mutable properties of, 9

C

capitalization format, use for constants, 166–167
catastrophic backtracking, 123–126
certificates, 202–205
classes
  and aspect-oriented programming, 84
  and monkeypatching, 112
  numeric, parent–child relationships between, 232–233, 232*f*

and protected names in implementations, 75–77
single-instance class vs. class Borg, 29
truthiness of, 25
cleanup. *See* file closing
closure function
behavior, 24
definition, 23
Cloudflare network, 202
code samples
assertions
sample algorithm, 212–213
sample algorithm after optimization, 213–214
avoiding filename conflicts with relative imports, 40, 41–42
backtracking in regular expression matching, 124
backward compatibility in Python, illustration, 41
Boolean checks
NumPy and Pandas, 25
in Python, 26
in SQL, 27
Boolean function with a sentinel, 27–28
built-ins, in Python 3.11, 72–73
calculation of data values with magic, 4–5, 5*f*
class implementation of an LCG, 76–78, 79
closure behavior
in JavaScript, 23–24
in Python, 22–23, 24
comparing types
isinstance() implementation, 69
vector_op() flawed implementation, 68–69
vector_op() good implementation, 70
concatenation
in-place string, 52–53, 54
using a constructed list, 54–55
using io.StringIO, 55
constants
defined by capitalization, 166
defined by enum, 167
overwriting, 168
using enum for read-only values, 168
CountingTree custom data structure
benchmarks, 182, 184–185
implementation, 183–184
performance, 186
cryptography
custom encryption/decryption, 198–199
using templates, 200–201
data structure
bytearray methods for mutable sequences, 147–149
collections.defaultdict, 130–131
collections.ChainMap, 139–141
collections.Counter, 132–134
collections.deque, 136–138
collections.namedtuple, 142–145
in collections.Sequence runtimes, 149–150
dataclass, 145–146
decorator
attaching attributes, 87–88
@register_plugin() for metaclass registration, 111
factory use, 85–86
functionality, 83
functionality, dual, 86–87
using functools.total_ordering, 88–89
writing good and bad, 84–85
dictionaries
examples of, 169
of basic types, 170
of classes, 170
use of dict.copy(), 174
use of dict.pop(), 172–173
use of dict.popitem(), 173
use of dict.setdefault(), 171
use of dict.update(), 171–172
EAFP and LBYL coding style comparison, 119–121
EAFP coding style example, 120–121
enumerate() use, 7
equality and identity comparisons, 36–37
except statements
generic, outputs, 48
specific, outputs, 48–50, 51
exception hierarchy, 46

FASTA file retrieval, 91–92, 93–94, 95–96
file handling
  limits for temporary files, 57–58
  limits of operating system, 57
  opening without closure, 56
  safe, 58–59
  unsafe, 58
flattening, strings
  and broken recursion, 164
  safely, 165
flattening nested data, naive attempts, 163–164, 165
floating point numbers
  associativity and distributivity, 224–225
  bit patterns of, 219
  comparing `statistics.mean()` with `statistics.fmean()`, 222
  comparing with `approxEqualNums()`, 221
  comparison including NaNs, 220–221
  finding median in presence of NaNs, 223
  granularity, 226–227
  mean, 226
  NaN bit patterns, 219
  NaN-aware comparison, 222
`for` loops vs. `while` loops, 12–13
f-string debugging using `f"{var=}`," 81, 82–83
"getter" and "setter" antipattern, 116
getting and setting using properties, 117–118
HTTPS page retrieval
  failure using `urllib`, 206–207
  using `requests`, 206
`if` suite, with/without inline initialization, 15
`import *`
  from `itertools`, 45
  inputs, 42–43
  outputs, 44
imports, naming sources, 44
integer interning, 22
iterable class returning an iterator, 90
iterables multiple
  unpythonic use, 15–16
  `zip()` use, 16–17
`itertools` combining function `accumulate()`, 95–96
`itertools.zip_longest()` to impute missing data, 19
JSON data processing
  round-trip failures, 177–178
  with `json` module, 176
keywords, Python 3.11, 72
lazy calculation within a generator, 5, 6*f*
LBYL coding style example, 119–120
list creation from generated items, 4
list of lists
  mistaken creation of, 34
  solution for creation of, 35
lists
  deleting items from `word` data structure, 161
  `insert_then_del()`, 160
  maintaining a data sequence, 161–162
  memory allocation and amortization, 158–159
loop-and-a-half patterns, 13–14
looping over `my_dict.keys()`, 8–9
matching structural patterns
  with `match` and `case` statements, 121–122
  with `requests.get()`, 122–123
metaclass
  created with `tuple.__new__()`, 110
  registration, using decorator `@register_plugin()`, 111
  use, with `PluginLoggerMeta(type)`, 108–109
`metaclass.py` source code, 109
monkeypatching scenario with `re.match()` calls, 112–115
mutable default arguments
  class-based solution, 31
  generator-based solution, 32–33
  illustration, 30
  `None` solution, 32
mutating an object during iteration, 9–11
name mangling, 78
`namedtuple` attributes and methods, 80

naming
  assigning to keywords vs. built-ins, 73
  overwriting built-ins, 73–75
numeric datatypes
  class parent–child relationships, 233–234
  `fractions.Fraction` class, 236
  interest compounding, 229–231
  operations combining number types, 235–236
  `Ratio(Fraction)` class, 237–238
  rounding modes, 231
object creation as a filter of a sequence, 11
randomness
  cryptographic, generate token length, 191
  reproducible, 192–193
  reproducible, creating test sequences, 194–195
  reproducible, saving a Mersenne Twister state, 193
run-length functions, 93–94
search
  one collection to another collection, 156
  with `get_word()`, 159
  with `range`, 153
  with `RegexFlag`, 153
  time, 155
secure source code
  insecure credential storage, 196
  using `dotenv`, 197
  using environment variables, 196
  using `keyring`, 197–198
single-instance class vs. class `Borg`, 29
sorting
  of custom objects, 59–60
  of heterogenous iterables, 60
  using `operator.itemgetter` and `dict` key, 62
  using the key function, 61
  with unnecessary use of subclass, 61
SQL injection attack
  example, 209–210
  and DB-API solutions, 208–209
  and sanitized inputs, 210, 211, 212
SSL/TLS
  authentication, 203–205
    Flask application, 202
    localhost access, 202
  string interning, 22
  testing function, 181
  type annotations, example of, 99
  type checking
    static and runtime, 100–101
    using `typing.Sequence`, 102–103
  `TypeError` exception, 101
  typing, using `class` statements, 104–105
  `typing.NewType()` misapplication, 104
  uncertain keys
    EAFP approach, 63
    just-use-default approach, 64
    LBYL approach, 63
  `zip(strict=True)` for data consistency, 18
code smells
  Boolean overchecking, 27
  utilizing the index position within a loop, 7
`collapse()`, 97–98
collections
  looping over, 13
  ordered, 10
`collections.ChainMap`
  mistake code sample, 140–141
  solution code sample, 139–140
  and virtual mapping, 138
`collections.Counter`
  mistake code sample, 134–135
  solution code sample, 132–134
`collections.defaultdict`
  mistake code sample, 130
  solution code sample, 130–132
`collections.deque`
  deque characteristics, 135
  list insertions and deletions, 161
  mistake code sample, 137–138
  solution code sample, 136–137
`collections.namedtuple`
  dynamic code sample, 142–144
  static code sample, 144–145
commutative property, mathematical, 225

concatenation
  naive, encountering quadratic complexity, 157
  of strings, 52–56
  using a constructed list, 54–55
  using `io.StringIO`, 55
concurrency, 3, 138, 224, 242
`concurrent.futures` module, 242
constants
  capitalization use to define, 166
  defined by `enum`, 167–168
  overwriting, 168
containerization approach to packaging, 243
containment mechanisms, 153–157
context managers
  cleanup loss mitigation, 56–59
  placing around `open()`, 48
  writing custom, 59
copies, making, 173–174
`copy.deepcopy()`, 174
CountingTree (custom implementation), 161, 182–186, 183*f*
cryptography
  amateur challenge, 200*n*
  primitives, in Python library, 198–199
  scope of, 189
  templates, 200–201
Cryptography Engineering: Design Principles and Practical Applications (Schneier, Ferguson, and Kohno), 189
`csv` module, 142
custom approaches
  to averaging floating point numbers, 226
  to writing data structures, 161, 179–186
  to writing security mechanisms, 198–201

D

data classes, 142, 145–146
data flaw remedies
  `itertools.zip_longest()`, 19–20
  `zip()`, 16–20
  `zip(strict=True)`, 17–18
data loss mitigation, 59
data structures, custom
  benchmarking, 181–182
  guidelines for, 181–182, 186–187
  implementation example, 181–186
  Python library limitations, 178–179
  rationales against, 179–181
data structures. *See also specific data structures*
  for antipatterns
    dictionary solutions, 169–174
    enums for sets of constants, 166–168
    JSON serialization/deserialization failures, 174–178
    list insertions and deletions, 157–162
    membership tests, 153–157
    strings and flatten functions, 163–166
  library modules
    collections.ChainMap, 138–141
    collections.Counter, 132–135
    collections.defaultdict, 129–132
    collections.deque, 135–138
    collections.namedtuple, 142–145
    dataclass, 145–146
    for record-oriented data, 141–142, 145
    inclusion criteria, 179
    sequence data types, 146–150
`dataclass`, 142, 145–146
dataframes, 25, 244
datum, calculating a value, 4
DB-API (database application programming interface) Python library, 208–209, 211
debugging, 80–83
`decimal` module, 231
`decimal.Decimal`, 235
decorate-sort-undecorate pattern, 61
decorator factories, 86–87
decorators
  and decorator factories, 86–87
  for data class definition, 146
  for metaclass registration, 110–111
  standard, 88–89
  value of, 83–84, 89
  writing, 84–88
`deque` characteristics, 135, 135*t*
deserialized JSON data, 62

Design Patterns: Elements of Reusable Object-Oriented Software (Gamma, Helm, Johnson, and Vlissides), 29
`dict`, 8, 9, 10, 161
`.__dict__` attribute storage of dictionary characteristics, 169
`dict.get()` function, 62–64
dictionary data structures
  `copy.deepcopy()` preferred to `dict.copy()`, 173–174
  defining classes, 170
  `.__dict__` attribute storage of characteristics, 169
  `dict.pop()`, 172–173
  `dict.popitem()`, 172–173
  `dict.setdefault()` for precision, 169
  `dict.update()`, 171–172
  `point`, 172
  `point.update()`, 172
  of types, 170
`dict.items()`, 9
`dict.keys()`, 8
`dict.setdefault()`, 171
`dict.update()`, 171
distributivity, mathematical, 224
Django, 108
Django REST framework, 202
`doctest` module, 242
domain-specific language (DSL), 244
duck typing, 67–68
dunder methods
  to control rounding, 238
  to define private attributes, 75–76
  format of "dunder" names, 52*n*

E

EAFP (easier-to-ask-forgiveness-than-permission), 63, 67, 118–121
encryption/decryption, 198–199
`enum` module, 166–168
`enumerate()`, 6–7
enumerations, 166–168
environment variables, 196–197
equality and identity, of objects
  comparisons, 35–37
  importance of distinguishing, 21
  `is` vs. `==`, 37
  JavaScript conceptual diagram, 37*f*
equivalency of objects. *See* equality and identity, of objects
Erlang programming language, 3
exceptions
  built-in, 46–51
  f-string debugging, 80–83
  narrowing for best results, 46
  Python hierarchy of, 47*f*
exponential behavior, 124

F

`f"{var=}"` pattern, 80
"fail fast" approach, 19
FASTA, 91–92, 94–95
FastAPI framework, 201–203
faster CPython project, 35
FIFO (first in, first out) operations, 135
file closure
  for data loss mitigation, 59
  explicit via `fh.close()`, 57
  file handle limit safety, 57–58
  queued changes safety, 58
file handling
  limits, for temporary files, 57–58
  opening without closing, 56
  safe and unsafe, 58–59
`fillvalue` argument, 19
financial calculations, 228–232
FIFO (first in, first out) operations, 135
Flask framework, 201–203
`flatten()` function, 163–166
`float` numeric type, 235, 237
floating point numbers
  avoiding for financial calculations, 228–232
  comparing, 220–221
  granularity of, 226–228
  IEEE-754 standard NaNs, 218–223, 218*n*
  NaN bit patterns, 219–220

plausibility arguments alternative, 224–225
precision of, 229
statistics module treatment of, 227
`for` and `while` equivalence, 13
`for` loop, 3, 12
`fractions.Fraction` class, 236–237
f-strings, debugging, 80–83

G

`get data()` function, 12, 15
getters and setters
Java pattern to be avoided, 116
Pythonic alternative, 116–118
`get_word()` function, 4
GIL (global interpreter lock), 242
granularity, mathematical, 226–228
Grimm, Avdi, 112

H

Haldane, J. B. S., 217–218
"hand-rolled" approaches. *See* custom approaches
Hash Array Mapped Trie (HAMT) data structures, 179
`hashlib` module, 198
Haskell programming language, 3
hidden failure in mutation of iterables, 10
`hmac` module, 198
Hopper, Grace, 89, 118
HTTP vs. HTTPS, 206

I

identity, of mutable objects
comparison through interning, 35
vs. equality, 21
`is` vs. `==`, 35, 37
IEEE-754-1985 standard, 218, 226, 236
`if obj is None`, 28
`if` statements, 15
`if/elif` checks, 215
immutable objects, 9, 21, 22, 163, 168, 179
`import *` function, 42–45
import options
complexity of, 39–40
`import *` ambiguity, 42–45
relative imports, 40
imputed values configurations, 19
infinities, excluding, 224
injection attack
code example, 209–210
DB-API solution, 208–209, 211–212
SQL vulnerability to, 208
integers
interning, 22, 148–149
and list structure, 148
interest compounding, 228–232
internal-break-style loop-and-a-half pattern, 14
interning, 22, 35–36
`isinstance()` function, 69–70
iterables
and data consistency, 17–20
emphasis on, in Python, 3, 61
vs. iterators, 90
looping over, 13
multiple, 17
sortability of, 61
iteration
and iterator length consistency, 18
list generation for, 3–6
and mutating objects, 9–11
over `dict.keys()`, 8
"iterator algebra," 45, 94–95, 96
`itertools` module
combining function `accumulate()`, 95–96
distinctive names of, 45
flattening function `collapse()`, 97, 98
functions most used, 94
`import *` function, 45
impute missing data with `itertools.zip_longest()`, 19
purpose of functions in, 91
and reading FASTA, 91–92
and run-length encoding (RLE), 92–94
`itertools.zip_longest()` function, 19–20

J

JavaScript
- closures behavior, 23–24
- equality and identity conceptual diagram, 37*f*
- JSON. *See* JSON data-interchange format
- packaging, of software, 243

JIT (just-in-time) Python interpreter, 180

JSON data-interchange format
- characteristics, 175–176
- `json` Python library module, 176–177
- and nested data structures, 97
- serialization/deserialization failures, 176–178
- sorting deserialized data, 62

K

Kahan, William, 218, 225

`key` functions, 56–59, 61

`keyring` module, 197–198

`in` keyword, for checking containment in lists, 153–156

Knuth, Donald, 225

L

lambda function objects, 21, 24, 62

lazy computation
- constructing a generator for, 5
- emphasis on, in Python, 90–91
- `itertools` functions for, 91
- and run-length encoding (RLE), 92–94

lazy iterable, 3

LBYL (look-before-you-leap) approach
- code sample, 63
- EAFP comparison, 118–121

LCGs (linear congruential generators), 76–79, 190, 212–215

LEGB (local, enclosing, global, and built-in) rule, 74

Let's Encrypt certificate authority, 202

LIFO (last in, first out) operations, 135

linear congruential generators (LCGs), 76–79, 190, 212–215

Lisp-family languages, 3

list, 146–150

lists
- boxed structure, 148
- efficiency of common operations, 135*t*
- insertions and deletions, 157–162
- listing, right and wrong ways, 33–35
- memory allocation behavior, 158–159
- and memory consumption, 3
- sorting, 155

local, enclosing, global, and built-in (LEGB) rule, 74

Lodash library, 96

look-before-you-leap (LBYL) approach. *See* LBYL approach

loop-and-a-half pattern, 13–15

looping, 3–20

loops
- `for item in iterable`, 12
- over index elements, 6
- and recursion, 3
- unpythonic, 6
- `while True`, 12

M

mapping, index to values, 8

Martelli, Alex, 29, 119*n*, 225

`match/case` checks, 215

matching
- by NFAs in backtracking, 125–126
- with `re.match()` calls, 112–115
- structural patterns, 121–123

`math.isclose()` function, 224–225

`math.isfinite()` function, 224

mean and median, of NaN values, 222–223

membership tests
- with `in` keyword, 153–156
- with `range`, 154
- with `RegexFlag`, 153–154
- with `set()`, 156–157

memory, consumption of
- floating point numbers as conserving, 236
- lazy calculation to conserve, 5

- and lists in Python, 3
- in numeric domains, 238

Mersenne Twister, 12*n*, 76, 190, 191, 193, 212
metaclasses
- behavior of, 110
- for logging registered plugins, 108–111
- fragility of, 111

microservices, 201
ML-family languages, 3
"Monkey Patching Is Destroying Ruby" (Grimm), 112
monkeypatching, 112–115
`more-itertools` third-party library, 95–98
`multiprocessing` module, 242
Munroe, Randall, 210
mutable default arguments
- class-based approach, 31–32
- generator-based approach, 32–33
- `None` sentinel approach, 32
- Python behavior, 29–31

mutation
- of iterables, 10
- of objects, 9

Mypy type analyzer, 100–102, 244

N

name mangling, 78, 116
`namedtuple` attributes, 80, 142
names and naming concerns
- exception statement vagueness, 46–51
- filename conflicts, 40–42
- import complexity, 39–40, 42–45
- overriding names in `_builtins_`, 71–75
- private and protected attributes use, 77–79

namespace, 17, 167
NaN (Not a Number) values
- bit patterns, 219–220
- description and roles, 218, 220
- effects on `statistics.median`, 221–223
- excluding with `math.isfinite()`, 224
- mean and median calculations and, 222–223
- propagation and stripping of, 223

nested data structures, 97–98
new-style loop-and-a-half pattern, 14
NFAs (nondeterministic finite automata), 125
nominative typing, 68
nondeterministic finite automata (NFAs), 125
Not a Number (NaN) values. *See* NaN values
`NullBase` class, 111
numeric datatypes
- abstract base classes, 232
- class parent–child relationships of, 233–234
- for financial calculations, 228
- for interest compounding, 229–231
- for operations combining number types, 235–236
- for setting numeric domains, 238
- inheritance diagram of base classes, 232*f*

NumPy
- averaging, performance of, 226–228
- and choosing a numeric datatype, 226–228
- floating point numbers in, 218, 228
- for large-scale numeric computation, 217, 224–225
- for tabular data handling, 34*n*
- library development of, 244
- and naming conflicts, 41
- NaN behavior in, 223
- and NaN propagation, 223
- and truthiness, 25
- usage by Python programmers, 96, 149
- vectorized properties of, 67, 88, 244

`numpy.isclose()` function, 224–225

O

object-oriented programming, 21
objects
- creating as a filter of a sequence, 11
- dictionary storage of characteristics, 169
- dual-role, 102–103
- equality vs. identity, 21, 35, 37
- immutability of, 21
- memory-consuming, 3, 5
- and polymorphic functions, 67–71

`open()` function, 48
optimizations
- interning strategies, 22
- and removal of assertions, 212, 213–215

P

packaging, of software, 243
Pandas
  access to dataframes via, 244
  bit width errors, 228
  Boolean context of, 25
  DataFrames, 218, 244
  for extensive numeric analysis, 142
  for tabular data handling, 34*n*
  and naming conflicts, 41
  NaN roles in, 218
  and NaN stripping, 223
  usage by Python programmers, 96
  vectorized properties of, 67
parallel computing, 3, 242
parameters
  passing safely, 208
  specification styles, 211
parentage of classes, 232, 232*f*, 233–234
pass by object reference, 21
passwords and secure information
  `dotenv` for semi-secure storage, 197
  environment variables for storage, 196–197
  examples
    insecure code, 196
    semi-secure code, 196
  `keyring` for secure storage, 197–198
  rationales for putting in source code, 195
PCGs (permuted congruential generators), 190
PEPs (Python Enhancement Proposals)
  PEP 249, 211
  PEP 484, 99
  PEP 603, 179
Peters, Tim, 108, 225
`pickle` module
  characteristics, 175
  unpickling concerns, 215
plausibility arguments, 224, 225
`Plugin` class, 108–111
polymorphism, and function creation, 67–71
posits and unums, 236*n*
possessive quantifier Python feature, 126
power-of-two modulus, 79
precision, of numbers, 229, 231, 235
`predicate()`, 12–13
primitives in Python library
  cryptographic, 198–199
  data structures, 179
“private” and “protected” attributes, 77–79
PRNGs (pseudo-random number generators), 12*n*, 76, 79, 190
programming languages, 3
propagation, of NaN values, 223
“protected” and “private” attributes, 77–79
pseudo-random number generators (PRNGs), 12*n*, 76, 79, 190
`_pth` files and import options, 39
PuDB debugger, 81*n*
PyCharm debugger, 81*n*
PyPy implementation, 180
PyPy program, 35
pyrsistent library, 129, 161
pytest, 241
Python documentation URLs
  `collections.OrderedDict` and `collections.UserDict`, 130
  concurrency approaches, 242
  on context managers, 59
  EAFP description, 119
  mistakes, descriptions of, 215–216
  type-checking tool, 244
  `urllib.request` module description, 205
  using a deque for a moving average, 136
Python Enhancement Proposals (PEPs). *See* PEPs
Python Packaging Authority, 243
`PYTHONPATH` and import options, 39

Q

quadratic complexity, 11, 52, 54, 135, 153–157

R

`raise` exception, 215
randomness
  for cryptographic purposes, 190–191
  for testing purposes, 192–195
range keyword, 154

RDBMSs (relational database management systems), 208
recursion
  and problem-solving, 3
  string breakage of, 164
`RecursionError` exception, 165
regex. *See* regular expressions
`RegexFlag` class, 153–154
regular expressions, 123–126
relative imports, 40, 42
`re.match()` function in monkeypatching, 112–115
reproducibility, of random values
  coding solution, 192–193
  `random` module use for, 192–195
`requests` third-party library, 121, 205–208
REST (representational state transfer), 201
Roper, James, 191
rounding issues
  control with `Ratio(Fraction)`, 237–238
  control with dunder methods, 238
  in floating point number operations, 220, 224
  rules for, in financial regulations, 228–232
rounding modes
  in Python, 231–232
  `ROUND_HALF_DOWN`, 231
  `ROUND_HALF_EVEN`, 231
Ruby programming language, 112
run-length encoding (RLE), 92–94
runtime
  and data quadratic complexity, 52
  and type annotations, 99–102
Rust programming language, 3

S

sanitizing database inputs, 210–212
Scala programming language, 3
scaling, linear, 54, 55
scoping
  behavior in Python, 21
  and garbage collection, 57
  LEGB rule, 74
  by using keyword binding, 24
searching
  complexity of, in lists, 153–157
  with `get_word()`, 159
  one collection to another collection, 156
  with `range`, 153
  with `RegexFlag`, 153
`secrets` module
  brute-force and timing attacks, defense against, 191
  for cryptographic purposes, 191–192
  function and purpose, 190
`secrets.compare_digest()` function, 191
secure sockets layer (SSL) protocol, 201, 202, 205
security concerns
  assertions used as safeguards, 212–215
  cryptographic randomness, 190–191
  custom cryptographic protocols, 198–201
  passwords in source code, 195–198
  scope of, 189
  SQL injection attacks and DB-API solutions, 208–212
  SSL/TLS for secure microservices, 201–208
  temp files created with `temp.mktemp()`, 216
  unpickling pickles, 215
  URL access with `requests` third-party library, 205–208
  XML loading, 216
  YAML loading, 215
sentinel values, 19
sequences
  filters for, 11
  key to good design, 13
serialization/deserialization, 175–177
session keys. *See* passwords and secure information
`set`, 9
sets, 156–157
"singleton pattern," 29
sortedcontainers library, 129, 155, 161

sorting
  using `key` arguments, 59–61
  using `operator.itemgetter` or `operator.attrgetter`, 62
SOWPODS word list, 98
SQL (structured query language)
  and Boolean values, 27
  injection attack vulnerability, 208–212
SSL (secure sockets layer) protocol, 201, 202, 205
stateful behavior, 12, 31–33
`statistics` module, 221–223, 226, 227
`statistics.mode()` function, 222
strings
  append operation, 56
  concatenation, 52–56, 157
  dual composite/atomic nature of, 166
  and flatten functions, 163–166
  interning, 22
  and recursion, 164
  as scalar objects, 163
stripping, of NaN values, 223
structured query language (SQL). *See* SQL
structural pattern matching, 121–123
subnormal numbers, 218*n*
`sys.path` and import options, 39

T

tail-call optimization, absent in Python, 3
TDD (test-driven development) recommendations, 241–242
`tempfile.mkstemp`, 216
test-driven development (TDD) recommendations, 241–242
`threading` module, 242
threat modeling, 189, 189*n*
Timsort algorithm, 60, 225
TLS (transport layer security) protocol, 201, 202, 205
tokens. *See* passwords and secure information
Torvalds, Linus, 150
Tower of Hanoi puzzle, 126
tracebacks, 164–165
transport layer security (TLS) protocol, 201, 202, 205
trie data structures, 157
truthiness
  defined by `._bool_()` dunder method, 25
  Python checks, 26
`tuple`, 9, 69–70, 102, 110, 142, 143, 147, 150
type annotations
  in data classes, 144
  and gradual typing, 99
  potential to mislead programmers, 105
  pros and cons, 243
  and runtime behavior, 77, 87, 99–102
`typeError`, 101
types (of values)
  implementations that compare, 67–71
  type checking tools, 100–105, 243–244
  typing an object, 68
`type(x) == type(y)` function, 69–70
`typing.NewType()` function
  generally, 102
  misapplication, 104–105

U

uncertain keys, 63–64
`unittest` module, 241
`urlib` module, 205–208

V

van Rossum, Guido, 89, 108
VS Code debugger, 81*n*

W

walrus operator, 14–15, 121
web server gateway interface (WSGI) server, 202
`while`
  expressed as `for`, 12
  loop, 3
  statement, 14

`while` and `for` equivalence, 13
`while True` statement, 14
WSGI (web server gateway interface) server, 202

X

`x == None` error, 28–29, 36
XKCD comic strip (Munroe), 210
XML loading, and denial-of-service attacks, 216

Y

YAML, loading, 215

Z

The Zen of Python (Peters), 17, 97, 108, 123
`zip()` function, to remedy data flaws, 16–20
`zip(strict=True)` function, to remedy data flaws, 17–18
`zip_longest()` function, 19