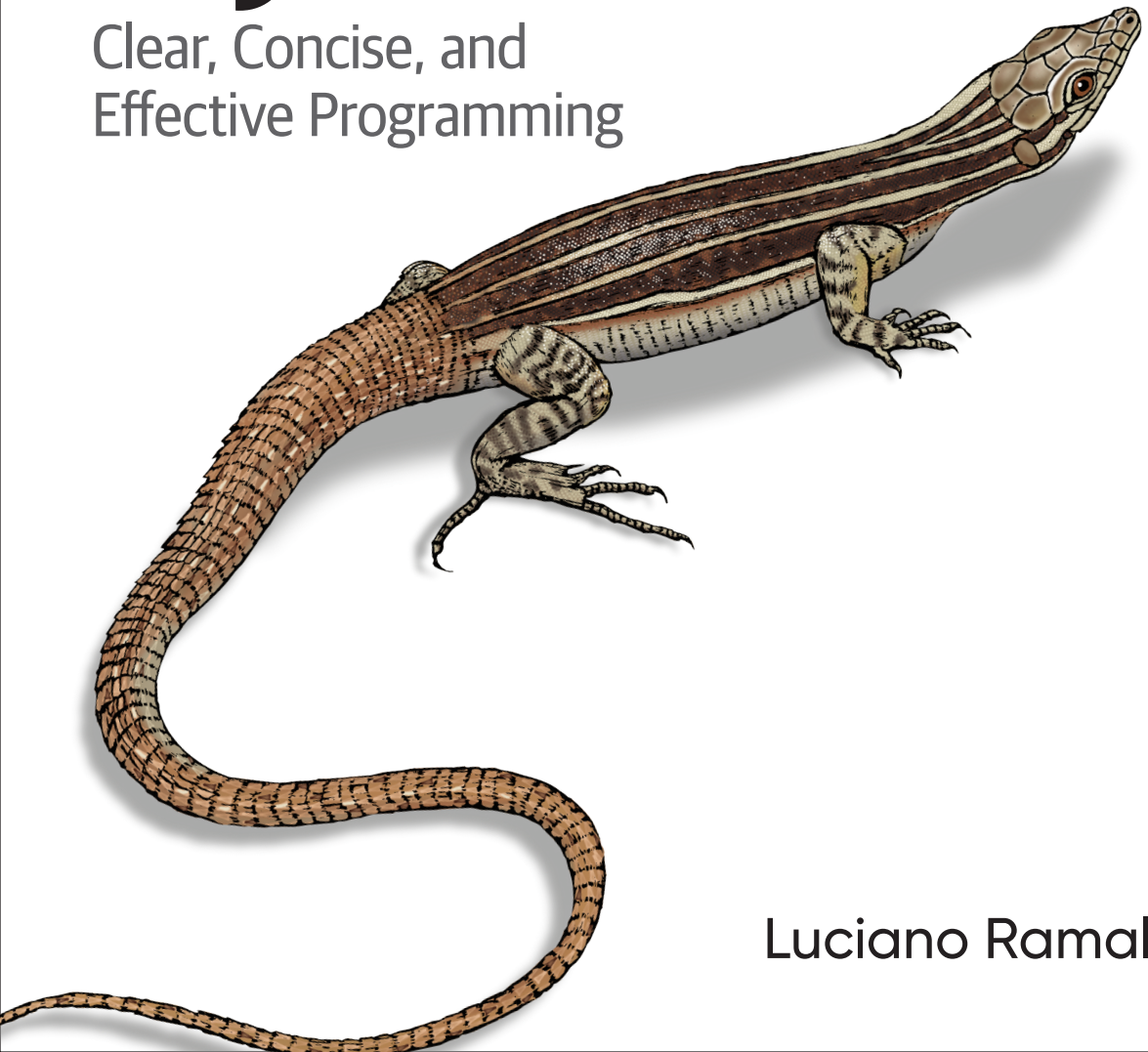
O'REILLY®

2nd Edition
Covers Python 3.10

Fluent Python

Clear, Concise, and Effective Programming



Luciano Ramalho

O'REILLY®

Fluent Python

Don't waste time bending Python to fit patterns you've learned in other languages. Python's simplicity lets you become productive quickly, but often this means you aren't using everything the language has to offer. With the updated edition of this hands-on guide, you'll learn how to write effective, modern Python 3 code by leveraging its best ideas.

Discover and apply idiomatic Python 3 features beyond your past experience. Author Luciano Ramalho guides you through Python's core language features and libraries and teaches you how to make your code shorter, faster, and more readable.

Complete with major updates throughout, this new edition features five parts that work as five short books within the book:

- **Data structures**: Sequences, dicts, sets, Unicode, and data classes
- **Functions as objects**: First-class functions, related design patterns, and type hints in function declarations
- **Object-oriented idioms**: Composition, inheritance, mixins, interfaces, operator overloading, protocols, and more static types
- **Control flow**: Context managers, generators, coroutines, async/await, and thread/process pools
- **Metaprogramming**: Properties, attribute descriptors, class decorators, and new class metaprogramming hooks that replace or simplify metaclasses

Luciano Ramalho is a principal consultant at Thoughtworks and a Python Software Foundation fellow.

"My 'go to' book when looking for detailed explanations and uses of a Python feature. Luciano's teaching and presentation are excellent. A great book for advanced beginners looking to build their knowledge."

—**Carol Willing**
Python Steering Council member (2020-2021)

"This is not the usual dry coding book, but full of useful, tested examples, and just enough humor. My colleagues and I have used this amazing, well-written book to take our Python coding to the next level."

—**Maria McKinley**
Senior Software Engineer

PROGRAMMING / PYTHON

US $69.99 CAN $87.99

ISBN: 978-1-492-05635-5

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

SECOND EDITION

Fluent Python

Clear, Concise, and Effective Programming

Luciano Ramalho

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Fluent Python
by Luciano Ramalho

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Amanda Quinn
Development Editor: Jeff Bleiel
Production Editor: Daniel Elfanbaum
Copyeditor: Sonia Saruba
Proofreader: Kim Cofer
Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

April 2022: Second Edition

Revision History for the Second Edition
2022-03-31: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492056355* for release details.

978-1-492-05635-5

[LSI]

Para Marta, com todo o meu amor.

Table of Contents

Part III. Classes and Protocols

Part V. Metaprogramming

Preface

> Here's the plan: when someone uses a feature you don't understand, simply shoot them. This is easier than learning something new, and before too long the only living coders will be writing in an easily understood, tiny subset of Python 0.9.6 <wink>.[1]
>
> —Tim Peters, legendary core developer and author of *The Zen of Python*

"Python is an easy to learn, powerful programming language." Those are the first words of the official Python 3.10 tutorial. That is true, but there is a catch: because the language is easy to learn and put to use, many practicing Python programmers leverage only a fraction of its powerful features.

An experienced programmer may start writing useful Python code in a matter of hours. As the first productive hours become weeks and months, a lot of developers go on writing Python code with a very strong accent carried from languages learned before. Even if Python is your first language, often in academia and in introductory books it is presented while carefully avoiding language-specific features.

As a teacher introducing Python to programmers experienced in other languages, I see another problem that this book tries to address: we only miss stuff we know about. Coming from another language, anyone may guess that Python supports regular expressions, and look that up in the docs. But if you've never seen tuple unpacking or descriptors before, you will probably not search for them, and you may end up not using those features just because they are specific to Python.

This book is not an A-to-Z exhaustive reference of Python. Its emphasis is on the language features that are either unique to Python or not found in many other popular languages. This is also mostly a book about the core language and some of its libraries. I will rarely talk about packages that are not in the standard library, even though the Python package index now lists more than 60,000 libraries, and many of them are incredibly useful.

1 Message to the comp.lang.python Usenet group, Dec. 23, 2002: "Acrimony in c.l.p".

Who This Book Is For

This book was written for practicing Python programmers who want to become proficient in Python 3. I tested the examples in Python 3.10—most of them also in Python 3.9 and 3.8. When an example requires Python 3.10, it should be clearly marked.

If you are not sure whether you know enough Python to follow along, review the topics of the official Python tutorial. Topics covered in the tutorial will not be explained here, except for some features that are new.

Who This Book Is Not For

If you are just learning Python, this book is going to be hard to follow. Not only that, if you read it too early in your Python journey, it may give you the impression that every Python script should leverage special methods and metaprogramming tricks. Premature abstraction is as bad as premature optimization.

Five Books in One

I recommend that everyone read Chapter 1, "The Python Data Model". The core audience for this book should not have trouble jumping directly to any part in this book after Chapter 1, but often I assume you've read preceding chapters in each specific part. Think of Parts I through V as books within the book.

I tried to emphasize using what is available before discussing how to build your own. For example, in Part I, Chapter 2 covers sequence types that are ready to use, including some that don't get a lot of attention, like `collections.deque`. Building user-defined sequences is only addressed in Part III, where we also see how to leverage the abstract base classes (ABCs) from `collections.abc`. Creating your own ABCs is discussed even later in Part III, because I believe it's important to be comfortable using an ABC before writing your own.

This approach has a few advantages. First, knowing what is ready to use can save you from reinventing the wheel. We use existing collection classes more often than we implement our own, and we can give more attention to the advanced usage of available tools by deferring the discussion on how to create new ones. We are also more likely to inherit from existing ABCs than to create a new ABC from scratch. And finally, I believe it is easier to understand the abstractions after you've seen them in action.

The downside of this strategy is the forward references scattered throughout the chapters. I hope these will be easier to tolerate now that you know why I chose this path.

How the Book Is Organized

Here are the main topics in each part of the book:

Part I, "Data Structures"

Chapter 1 introduces the Python Data Model and explains why the special methods (e.g., `__repr__`) are the key to the consistent behavior of objects of all types. Special methods are covered in more detail throughout the book. The remaining chapters in this part cover the use of collection types: sequences, mappings, and sets, as well as the `str` versus `bytes` split—the cause of much celebration among Python 3 users and much pain for Python 2 users migrating their codebases. Also covered are the high-level class builders in the standard library: named tuple factories and the `@dataclass` decorator. Pattern matching—new in Python 3.10—is covered in sections in Chapters 2, 3, and 5, which discuss sequence patterns, mapping patterns, and class patterns. The last chapter in Part I is about the life cycle of objects: references, mutability, and garbage collection.

Part II, "Functions as Objects"

Here we talk about functions as first-class objects in the language: what that means, how it affects some popular design patterns, and how to implement function decorators by leveraging closures. Also covered here is the general concept of callables in Python, function attributes, introspection, parameter annotations, and the new `nonlocal` declaration in Python 3. Chapter 8 introduces the major new topic of type hints in function signatures.

Part III, "Classes and Protocols"

Now the focus is on building classes "by hand"—as opposed to using the class builders covered in Chapter 5. Like any Object-Oriented (OO) language, Python has its particular set of features that may or may not be present in the language in which you and I learned class-based programming. The chapters explain how to build your own collections, abstract base classes (ABCs), and protocols, as well as how to cope with multiple inheritance, and how to implement operator overloading—when that makes sense. Chapter 15 continues the coverage of type hints.

Part IV, "Control Flow"

Covered in this part are the language constructs and libraries that go beyond traditional control flow with conditionals, loops, and subroutines. We start with generators, then visit context managers and coroutines, including the challenging but powerful new `yield from` syntax. Chapter 18 includes a significant example using pattern matching in a simple but functional language interpreter. Chapter 19, "Concurrency Models in Python" is a new chapter presenting an overview of alternatives for concurrent and parallel processing in Python, their limitations, and how software architecture allows Python to operate at web scale. I rewrote

the chapter about *asynchronous programming* to emphasize core language features—e.g., `await`, `async def`, `async for`, and `async with`, and show how they are used with *asyncio* and other frameworks.

Part V, "Metaprogramming"
: This part starts with a review of techniques for building classes with attributes created dynamically to handle semi-structured data, such as JSON datasets. Next, we cover the familiar properties mechanism, before diving into how object attribute access works at a lower level in Python using descriptors. The relationship among functions, methods, and descriptors is explained. Throughout Part V, the step-by-step implementation of a field validation library uncovers subtle issues that lead to the advanced tools of the final chapter: class decorators and metaclasses.

Hands-On Approach

Often we'll use the interactive Python console to explore the language and libraries. I feel it is important to emphasize the power of this learning tool, particularly for those readers who've had more experience with static, compiled languages that don't provide a read-eval-print loop (REPL).

One of the standard Python testing packages, `doctest`, works by simulating console sessions and verifying that the expressions evaluate to the responses shown. I used `doctest` to check most of the code in this book, including the console listings. You don't need to use or even know about `doctest` to follow along: the key feature of doctests is that they look like transcripts of interactive Python console sessions, so you can easily try out the demonstrations yourself.

Sometimes I will explain what we want to accomplish by showing a doctest before the code that makes it pass. Firmly establishing what is to be done before thinking about how to do it helps focus our coding effort. Writing tests first is the basis of test-driven development (TDD), and I've also found it helpful when teaching. If you are unfamiliar with `doctest`, take a look at its documentation and this book's example code repository.

I also wrote unit tests for some of the larger examples using *pytest*—which I find easier to use and more powerful than the *unittest* module in the standard library. You'll find that you can verify the correctness of most of the code in the book by typing `python3 -m doctest example_script.py` or `pytest` in the command shell of your OS. The *pytest.ini* configuration at the root of the example code repository ensures that doctests are collected and executed by the `pytest` command.

Soapbox: My Personal Perspective

I have been using, teaching, and debating Python since 1998, and I enjoy studying and comparing programming languages, their design, and the theory behind them. At the end of some chapters, I have added "Soapbox" sidebars with my own perspective about Python and other languages. Feel free to skip these if you are not into such discussions. Their content is completely optional.

Companion Website: fluentpython.com

Covering new features—like type hints, data classes, and pattern matching—made this second edition almost 30% larger than the first. To keep the book luggable, I moved some content to *fluentpython.com*. You will find links to articles I published there in several chapters. Some sample chapters are also in the companion website. The full text is available online at the O'Reilly Learning subscription service. The example code repository is on GitHub.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

  Note that when a line break falls within a `constant_width` term, a hyphen is not added—it could be misunderstood as part of the term.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Every script and most code snippets that appear in the book are available in the Fluent Python code repository on GitHub at *https://fpy.li/code*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN, e.g., "*Fluent Python*, 2nd ed., by Luciano Ramalho (O'Reilly). Copyright 2022 Luciano Ramalho, 978-1-492-05635-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://fpy.li/p-4*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For news and information about our books and courses, visit *http://oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*.

Follow us on Twitter: *https://twitter.com/oreillymedia*.

Watch us on YouTube: *http://www.youtube.com/oreillymedia*.

Acknowledgments

I did not expect updating a Python book five years later to be such a major undertaking, but it was. Marta Mello, my beloved wife, was always there when I needed her. My dear friend Leonardo Rochael helped me from the earliest writing to the final technical review, including consolidating and double-checking the feedback from the other tech reviewers, readers, and editors. I honestly don't know if I'd have made it without your support, Marta and Leo. Thank you so much!

Jürgen Gmach, Caleb Hattingh, Jess Males, Leonardo Rochael, and Miroslav Šedivý were the outstanding technical review team for the second edition. They reviewed the whole book. Bill Behrman, Bruce Eckel, Renato Oliveira, and Rodrigo Bernardo Pimentel reviewed specific chapters. Their many suggestions from different perspectives made the book much better.

Many readers sent corrections or made other contributions during the early release phase, including: Guilherme Alves, Christiano Anderson, Konstantin Baikov, K. Alex Birch, Michael Boesl, Lucas Brunialti, Sergio Cortez, Gino Crecco, Chukwuerika Dike, Juan Esteras, Federico Fissore, Will Frey, Tim Gates, Alexander Hagerman, Chen Hanxiao, Sam Hyeong, Simon Ilincev, Parag Kalra, Tim King, David Kwast, Tina Lapine, Wanpeng Li, Guto Maia, Scott Martindale, Mark Meyer, Andy McFarland, Chad McIntire, Diego Rabatone Oliveira, Francesco Piccoli, Meredith Rawls, Michael Robinson, Federico Tula Rovaletti, Tushar Sadhwani, Arthur Constantino Scardua, Randal L. Schwartz, Avichai Sefati, Guannan Shen, William Simpson, Vivek Vashist, Jerry Zhang, Paul Zuradzki—and others who did not want to be named, sent corrections after I delivered the draft, or are omitted because I failed to record their names—sorry.

During my research, I learned about typing, concurrency, pattern matching, and metaprogramming while interacting with Michael Albert, Pablo Aguilar, Kaleb Barrett, David Beazley, J. S. O. Bueno, Bruce Eckel, Martin Fowler, Ivan Levkivskyi, Alex Martelli, Peter Norvig, Sebastian Rittau, Guido van Rossum, Carol Willing, and Jelle Zijlstra.

O'Reilly editors Jeff Bleiel, Jill Leonard, and Amelia Blevins made suggestions that improved the flow of the book in many places. Jeff Bleiel and production editor Danny Elfanbaum supported me throughout this long marathon.

The insights and suggestions of every one of them made the book better and more accurate. Inevitably, there will still be bugs of my own creation in the final product. I apologize in advance.

Finally, I want to extend my heartfelt thanks to my colleagues at Thoughtworks Brazil—and especially to my sponsor, Alexey Bôas—who supported this project in many ways, all the way.

Of course, everyone who helped me understand Python and write the first edition now deserves double thanks. There would be no second edition without a successful first.

Acknowledgments for the First Edition

The Bauhaus chess set by Josef Hartwig is an example of excellent design: beautiful, simple, and clear. Guido van Rossum, son of an architect and brother of a master font designer, created a masterpiece of language design. I love teaching Python because it is beautiful, simple, and clear.

Alex Martelli and Anna Ravenscroft were the first people to see the outline of this book and encouraged me to submit it to O'Reilly for publication. Their books taught me idiomatic Python and are models of clarity, accuracy, and depth in technical writing. Alex's 6,200+ Stack Overflow posts are a fountain of insights about the language and its proper use.

Martelli and Ravenscroft were also technical reviewers of this book, along with Lennart Regebro and Leonardo Rochael. Everyone in this outstanding technical review team has at least 15 years of Python experience, with many contributions to high-impact Python projects in close contact with other developers in the community. Together they sent me hundreds of corrections, suggestions, questions, and opinions, adding tremendous value to the book. Victor Stinner kindly reviewed Chapter 21, bringing his expertise as an `asyncio` maintainer to the technical review team. It was a great privilege and a pleasure to collaborate with them over these past several months.

Editor Meghan Blanchette was an outstanding mentor, helping me improve the organization and flow of the book, letting me know when it was boring, and keeping me from delaying even more. Brian MacDonald edited chapters in Part II while Meghan was away. I enjoyed working with them, and with everyone I've contacted at O'Reilly, including the Atlas development and support team (Atlas is the O'Reilly book publishing platform, which I was fortunate to use to write this book).

Mario Domenech Goulart provided numerous, detailed suggestions starting with the first early release. I also received valuable feedback from Dave Pawson, Elias Dorneles, Leonardo Alexandre Ferreira Leite, Bruce Eckel, J. S. Bueno, Rafael Gonçalves, Alex Chiaranda, Guto Maia, Lucas Vido, and Lucas Brunialti.

Over the years, a number of people urged me to become an author, but the most persuasive were Rubens Prates, Aurelio Jargas, Rudá Moura, and Rubens Altimari. Mauricio Bussab opened many doors for me, including my first real shot at writing a book. Renzo Nuccitelli supported this writing project all the way, even if that meant a slow start for our partnership at *python.pro.br*.

The wonderful Brazilian Python community is knowledgeable, generous, and fun. The Python Brasil group has thousands of people, and our national conferences bring together hundreds, but the most influential in my journey as a Pythonista were Leonardo Rochael, Adriano Petrich, Daniel Vainsencher, Rodrigo RBP Pimentel, Bruno Gola, Leonardo Santagada, Jean Ferri, Rodrigo Senra, J. S. Bueno, David Kwast, Luiz

Irber, Osvaldo Santana, Fernando Masanori, Henrique Bastos, Gustavo Niemayer, Pedro Werneck, Gustavo Barbieri, Lalo Martins, Danilo Bellini, and Pedro Kroger.

Dorneles Tremea was a great friend (incredibly generous with his time and knowledge), an amazing hacker, and the most inspiring leader of the Brazilian Python Association. He left us too early.

My students over the years taught me a lot through their questions, insights, feedback, and creative solutions to problems. Érico Andrei and Simples Consultoria made it possible for me to focus on being a Python teacher for the first time.

Martijn Faassen was my Grok mentor and shared invaluable insights with me about Python and Neanderthals. His work and that of Paul Everitt, Chris McDonough, Tres Seaver, Jim Fulton, Shane Hathaway, Lennart Regebro, Alan Runyan, Alexander Limi, Martijn Pieters, Godefroid Chapelle, and others from the Zope, Plone, and Pyramid planets have been decisive in my career. Thanks to Zope and surfing the first web wave, I was able to start making a living with Python in 1998. José Octavio Castro Neves was my partner in the first Python-centric software house in Brazil.

I have too many gurus in the wider Python community to list them all, but besides those already mentioned, I am indebted to Steve Holden, Raymond Hettinger, A.M. Kuchling, David Beazley, Fredrik Lundh, Doug Hellmann, Nick Coghlan, Mark Pilgrim, Martijn Pieters, Bruce Eckel, Michele Simionato, Wesley Chun, Brandon Craig Rhodes, Philip Guo, Daniel Greenfeld, Audrey Roy, and Brett Slatkin for teaching me new and better ways to teach Python.

Most of these pages were written in my home office and in two labs: CoffeeLab and Garoa Hacker Clube. CoffeeLab is the caffeine-geek headquarters in Vila Madalena, São Paulo, Brazil. Garoa Hacker Clube is a hackerspace open to all: a community lab where anyone can freely try out new ideas.

The Garoa community provided inspiration, infrastructure, and slack. I think Aleph would enjoy this book.

My mother, Maria Lucia, and my father, Jairo, always supported me in every way. I wish he was here to see the book; I am glad I can share it with her.

My wife, Marta Mello, endured 15 months of a husband who was always working, but remained supportive and coached me through some critical moments in the project when I feared I might drop out of the marathon.

Thank you all, for everything.

PART I

Data Structures

CHAPTER 1
The Python Data Model

> Guido's sense of the aesthetics of language design is amazing. I've met many fine language designers who could build theoretically beautiful languages that no one would ever use, but Guido is one of those rare people who can build a language that is just slightly less theoretically beautiful but thereby is a joy to write programs in.
>
> —Jim Hugunin, creator of Jython, cocreator of AspectJ, and architect of the .Net DLR[1]

One of the best qualities of Python is its consistency. After working with Python for a while, you are able to start making informed, correct guesses about features that are new to you.

However, if you learned another object-oriented language before Python, you may find it strange to use `len(collection)` instead of `collection.len()`. This apparent oddity is the tip of an iceberg that, when properly understood, is the key to everything we call *Pythonic*. The iceberg is called the Python Data Model, and it is the API that we use to make our own objects play well with the most idiomatic language features.

You can think of the data model as a description of Python as a framework. It formalizes the interfaces of the building blocks of the language itself, such as sequences, functions, iterators, coroutines, classes, context managers, and so on.

When using a framework, we spend a lot of time coding methods that are called by the framework. The same happens when we leverage the Python Data Model to build new classes. The Python interpreter invokes special methods to perform basic object operations, often triggered by special syntax. The special method names are always written with leading and trailing double underscores. For example, the syntax

1 "Story of Jython", written as a foreword to *Jython Essentials* by Samuele Pedroni and Noel Rappin (O'Reilly).

`obj[key]` is supported by the `__getitem__` special method. In order to evaluate `my_collection[key]`, the interpreter calls `my_collection.__getitem__(key)`.

We implement special methods when we want our objects to support and interact with fundamental language constructs such as:

- Collections
- Attribute access
- Iteration (including asynchronous iteration using `async for`)
- Operator overloading
- Function and method invocation
- String representation and formatting
- Asynchronous programming using `await`
- Object creation and destruction
- Managed contexts using the `with` or `async with` statements

Magic and Dunder

The term *magic method* is slang for special method, but how do we talk about a specific method like `__getitem__`? I learned to say "dunder-getitem" from author and teacher Steve Holden. "Dunder" is a shortcut for "double underscore before and after." That's why the special methods are also known as *dunder methods*. The "Lexical Analysis" chapter of *The Python Language Reference* warns that "*Any* use of `__*__` names, in any context, that does not follow explicitly documented use, is subject to breakage without warning."

What's New in This Chapter

This chapter had few changes from the first edition because it is an introduction to the Python Data Model, which is quite stable. The most significant changes are:

- Special methods supporting asynchronous programming and other new features, added to the tables in "Overview of Special Methods" on page 15.
- Figure 1-2 showing the use of special methods in "Collection API" on page 14, including the `collections.abc.Collection` abstract base class introduced in Python 3.6.

Also, here and throughout this second edition I adopted the *f-string* syntax introduced in Python 3.6, which is more readable and often more convenient than the older string formatting notations: the `str.format()` method and the `%` operator.

One reason to still use `my_fmt.format()` is when the definition of `my_fmt` must be in a different place in the code than where the formatting operation needs to happen. For instance, when `my_fmt` has multiple lines and is better defined in a constant, or when it must come from a configuration file, or from the database. Those are real needs, but don't happen very often.

A Pythonic Card Deck

Example 1-1 is simple, but it demonstrates the power of implementing just two special methods, `__getitem__` and `__len__`.

Example 1-1. A deck as a sequence of playing cards

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

The first thing to note is the use of `collections.namedtuple` to construct a simple class to represent individual cards. We use `namedtuple` to build classes of objects that are just bundles of attributes with no custom methods, like a database record. In the example, we use it to provide a nice representation for the cards in the deck, as shown in the console session:

```
>>> beer_card = Card('7', 'diamonds')
>>> beer_card
Card(rank='7', suit='diamonds')
```

But the point of this example is the `FrenchDeck` class. It's short, but it packs a punch. First, like any standard Python collection, a deck responds to the `len()` function by returning the number of cards in it:

```
>>> deck = FrenchDeck()
>>> len(deck)
52
```

Reading specific cards from the deck—say, the first or the last—is easy, thanks to the `__getitem__` method:

```
>>> deck[0]
Card(rank='2', suit='spades')
>>> deck[-1]
Card(rank='A', suit='hearts')
```

Should we create a method to pick a random card? No need. Python already has a function to get a random item from a sequence: `random.choice`. We can use it on a deck instance:

```
>>> from random import choice
>>> choice(deck)
Card(rank='3', suit='hearts')
>>> choice(deck)
Card(rank='K', suit='spades')
>>> choice(deck)
Card(rank='2', suit='clubs')
```

We've just seen two advantages of using special methods to leverage the Python Data Model:

- Users of your classes don't have to memorize arbitrary method names for standard operations. ("How to get the number of items? Is it `.size()`, `.length()`, or what?")
- It's easier to benefit from the rich Python standard library and avoid reinventing the wheel, like the `random.choice` function.

But it gets better.

Because our `__getitem__` delegates to the `[]` operator of `self._cards`, our deck automatically supports slicing. Here's how we look at the top three cards from a brand-new deck, and then pick just the aces by starting at index 12 and skipping 13 cards at a time:

```
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'),
Card(rank='4', suit='spades')]
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

Just by implementing the `__getitem__` special method, our deck is also iterable:

```
>>> for card in deck:  # doctest: +ELLIPSIS
...   print(card)
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
Card(rank='4', suit='spades')
...
```

We can also iterate over the deck in reverse:

```
>>> for card in reversed(deck):  # doctest: +ELLIPSIS
...   print(card)
Card(rank='A', suit='hearts')
Card(rank='K', suit='hearts')
Card(rank='Q', suit='hearts')
...
```

Ellipsis in doctests

Whenever possible, I extracted the Python console listings in this book from `doctest` to ensure accuracy. When the output was too long, the elided part is marked by an ellipsis (`...`), like in the last line in the preceding code. In such cases, I used the `# doctest: +ELLIPSIS` directive to make the doctest pass. If you are trying these examples in the interactive console, you may omit the doctest comments altogether.

Iteration is often implicit. If a collection has no `__contains__` method, the `in` operator does a sequential scan. Case in point: `in` works with our `FrenchDeck` class because it is iterable. Check it out:

```
>>> Card('Q', 'hearts') in deck
True
>>> Card('7', 'beasts') in deck
False
```

How about sorting? A common system of ranking cards is by rank (with aces being highest), then by suit in the order of spades (highest), hearts, diamonds, and clubs (lowest). Here is a function that ranks cards by that rule, returning `0` for the 2 of clubs and `51` for the ace of spades:

```
suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)

def spades_high(card):
    rank_value = FrenchDeck.ranks.index(card.rank)
    return rank_value * len(suit_values) + suit_values[card.suit]
```

Given `spades_high`, we can now list our deck in order of increasing rank:

```
>>> for card in sorted(deck, key=spades_high):  # doctest: +ELLIPSIS
...     print(card)
Card(rank='2', suit='clubs')
Card(rank='2', suit='diamonds')
Card(rank='2', suit='hearts')
... (46 cards omitted)
Card(rank='A', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='A', suit='spades')
```

Although `FrenchDeck` implicitly inherits from the `object` class, most of its functionality is not inherited, but comes from leveraging the data model and composition. By implementing the special methods `__len__` and `__getitem__`, our `FrenchDeck` behaves like a standard Python sequence, allowing it to benefit from core language features (e.g., iteration and slicing) and from the standard library, as shown by the examples using `random.choice`, `reversed`, and `sorted`. Thanks to composition, the `__len__` and `__getitem__` implementations can delegate all the work to a `list` object, `self._cards`.

How About Shuffling?

As implemented so far, a `FrenchDeck` cannot be shuffled because it is *immutable*: the cards and their positions cannot be changed, except by violating encapsulation and handling the `_cards` attribute directly. In Chapter 13, we will fix that by adding a one-line `__setitem__` method.

How Special Methods Are Used

The first thing to know about special methods is that they are meant to be called by the Python interpreter, and not by you. You don't write `my_object.__len__()`. You write `len(my_object)` and, if `my_object` is an instance of a user-defined class, then Python calls the `__len__` method you implemented.

But the interpreter takes a shortcut when dealing for built-in types like `list`, `str`, `bytearray`, or extensions like the NumPy arrays. Python variable-sized collections written in C include a struct[2] called `PyVarObject`, which has an `ob_size` field holding the number of items in the collection. So, if `my_object` is an instance of one of those built-ins, then `len(my_object)` retrieves the value of the `ob_size` field, and this is much faster than calling a method.

2 A C struct is a record type with named fields.

More often than not, the special method call is implicit. For example, the statement `for i in x:` actually causes the invocation of `iter(x)`, which in turn may call `x.__iter__()` if that is available, or use `x.__getitem__()`, as in the `FrenchDeck` example.

Normally, your code should not have many direct calls to special methods. Unless you are doing a lot of metaprogramming, you should be implementing special methods more often than invoking them explicitly. The only special method that is frequently called by user code directly is `__init__` to invoke the initializer of the superclass in your own `__init__` implementation.

If you need to invoke a special method, it is usually better to call the related built-in function (e.g., `len`, `iter`, `str`, etc.). These built-ins call the corresponding special method, but often provide other services and—for built-in types—are faster than method calls. See, for example, "Using iter with a Callable" on page 598 in Chapter 17.

In the next sections, we'll see some of the most important uses of special methods:

- Emulating numeric types
- String representation of objects
- Boolean value of an object
- Implementing collections

Emulating Numeric Types

Several special methods allow user objects to respond to operators such as +. We will cover that in more detail in Chapter 16, but here our goal is to further illustrate the use of special methods through another simple example.

We will implement a class to represent two-dimensional vectors—that is, Euclidean vectors like those used in math and physics (see Figure 1-1).

The built-in `complex` type can be used to represent two-dimensional vectors, but our class can be extended to represent *n*-dimensional vectors. We will do that in Chapter 17.

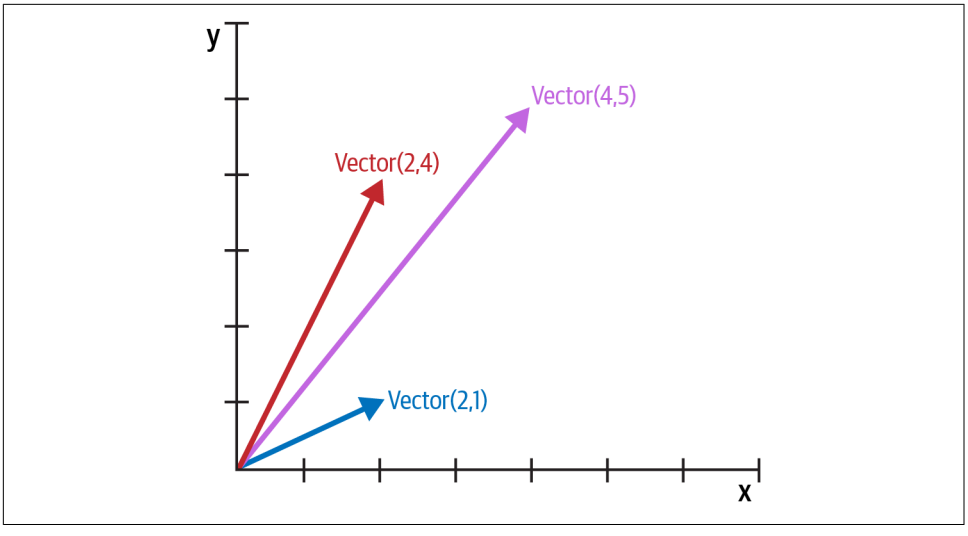


Figure 1-1. Example of two-dimensional vector addition; Vector(2, 4) + Vector(2, 1) results in Vector(4, 5).

We will start designing the API for such a class by writing a simulated console session that we can use later as a doctest. The following snippet tests the vector addition pictured in Figure 1-1:

```
>>> v1 = Vector(2, 4)
>>> v2 = Vector(2, 1)
>>> v1 + v2
Vector(4, 5)
```

Note how the `+` operator results in a new `Vector`, displayed in a friendly format at the console.

The `abs` built-in function returns the absolute value of integers and floats, and the magnitude of `complex` numbers, so to be consistent, our API also uses `abs` to calculate the magnitude of a vector:

```
>>> v = Vector(3, 4)
>>> abs(v)
5.0
```

We can also implement the `*` operator to perform scalar multiplication (i.e., multiplying a vector by a number to make a new vector with the same direction and a multiplied magnitude):

```
>>> v * 3
Vector(9, 12)
>>> abs(v * 3)
15.0
```

Example 1-2 is a `Vector` class implementing the operations just described, through the use of the special methods `__repr__`, `__abs__`, `__add__`, and `__mul__`.

Example 1-2. A simple two-dimensional vector class

```
"""
vector2d.py: a simplistic class demonstrating some special methods

It is simplistic for didactic reasons. It lacks proper error handling,
especially in the ``__add__`` and ``__mul__`` methods.

This example is greatly expanded later in the book.

Addition::

    >>> v1 = Vector(2, 4)
    >>> v2 = Vector(2, 1)
    >>> v1 + v2
    Vector(4, 5)

Absolute value::

    >>> v = Vector(3, 4)
    >>> abs(v)
    5.0

Scalar multiplication::

    >>> v * 3
    Vector(9, 12)
    >>> abs(v * 3)
    15.0

"""


import math

class Vector:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x!r}, {self.y!r})'

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
```

```
        return bool(abs(self))

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

We implemented five special methods in addition to the familiar `__init__`. Note that none of them is directly called within the class or in the typical usage of the class illustrated by the doctests. As mentioned before, the Python interpreter is the only frequent caller of most special methods.

Example 1-2 implements two operators: `+` and `*`, to show basic usage of `__add__` and `__mul__`. In both cases, the methods create and return a new instance of `Vector`, and do not modify either operand—`self` or `other` are merely read. This is the expected behavior of infix operators: to create new objects and not touch their operands. I will have a lot more to say about that in Chapter 16.

As implemented, Example 1-2 allows multiplying a `Vector` by a number, but not a number by a `Vector`, which violates the commutative property of scalar multiplication. We will fix that with the special method `__rmul__` in Chapter 16.

In the following sections, we discuss the other special methods in `Vector`.

String Representation

The `__repr__` special method is called by the `repr` built-in to get the string representation of the object for inspection. Without a custom `__repr__`, Python's console would display a `Vector` instance `<Vector object at 0x10e100070>`.

The interactive console and debugger call `repr` on the results of the expressions evaluated, as does the `%r` placeholder in classic formatting with the `%` operator, and the `!r` conversion field in the new format string syntax used in *f-strings* the `str.format` method.

Note that the *f-string* in our `__repr__` uses `!r` to get the standard representation of the attributes to be displayed. This is good practice, because it shows the crucial difference between `Vector(1, 2)` and `Vector('1', '2')`—the latter would not work in the context of this example, because the constructor's arguments should be numbers, not `str`.

The string returned by `__repr__` should be unambiguous and, if possible, match the source code necessary to re-create the represented object. That is why our `Vector` representation looks like calling the constructor of the class (e.g., `Vector(3, 4)`).

In contrast, `__str__` is called by the `str()` built-in and implicitly used by the `print` function. It should return a string suitable for display to end users.

Sometimes same string returned by `__repr__` is user-friendly, and you don't need to code `__str__` because the implementation inherited from the `object` class calls `__repr__` as a fallback. Example 5-2 is one of several examples in this book with a custom `__str__`.

Programmers with prior experience in languages with a `toString` method tend to implement `__str__` and not `__repr__`. If you only implement one of these special methods in Python, choose `__repr__`.

"What is the difference between `__str__` and `__repr__` in Python?" is a Stack Overflow question with excellent contributions from Pythonistas Alex Martelli and Martijn Pieters.

Boolean Value of a Custom Type

Although Python has a `bool` type, it accepts any object in a Boolean context, such as the expression controlling an `if` or `while` statement, or as operands to `and`, `or`, and `not`. To determine whether a value `x` is *truthy* or *falsy*, Python applies `bool(x)`, which returns either `True` or `False`.

By default, instances of user-defined classes are considered truthy, unless either `__bool__` or `__len__` is implemented. Basically, `bool(x)` calls `x.__bool__()` and uses the result. If `__bool__` is not implemented, Python tries to invoke `x.__len__()`, and if that returns zero, `bool` returns `False`. Otherwise `bool` returns `True`.

Our implementation of `__bool__` is conceptually simple: it returns `False` if the magnitude of the vector is zero, `True` otherwise. We convert the magnitude to a Boolean using `bool(abs(self))` because `__bool__` is expected to return a Boolean. Outside of `__bool__` methods, it is rarely necessary to call `bool()` explicitly, because any object can be used in a Boolean context.

Note how the special method `__bool__` allows your objects to follow the truth value testing rules defined in the "Built-in Types" chapter of *The Python Standard Library* documentation.

A faster implementation of `Vector.__bool__` is this:

```
def __bool__(self):
    return bool(self.x or self.y)
```

This is harder to read, but avoids the trip through `abs`, `__abs__`, the squares, and square root. The explicit conversion to `bool` is needed because `__bool__` must return a Boolean, and `or` returns either operand as is: `x or y` evaluates to `x` if that is truthy, otherwise the result is `y`, whatever that is.

Collection API

Figure 1-2 documents the interfaces of the essential collection types in the language. All the classes in the diagram are ABCs—*abstract base classes*. ABCs and the `collections.abc` module are covered in Chapter 13. The goal of this brief section is to give a panoramic view of Python's most important collection interfaces, showing how they are built from special methods.

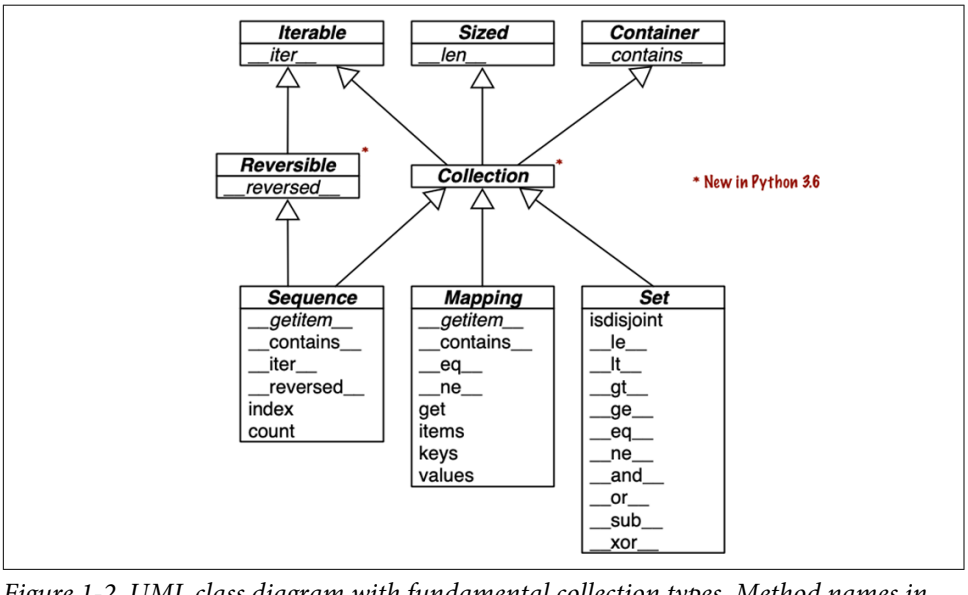


Figure 1-2. UML class diagram with fundamental collection types. Method names in italic are abstract, so they must be implemented by concrete subclasses such as `list` and `dict`. The remaining methods have concrete implementations, therefore subclasses can inherit them.

Each of the top ABCs has a single special method. The `Collection` ABC (new in Python 3.6) unifies the three essential interfaces that every collection should implement:

- `Iterable` to support `for`, unpacking, and other forms of iteration
- `Sized` to support the `len` built-in function
- `Container` to support the `in` operator

Python does not require concrete classes to actually inherit from any of these ABCs. Any class that implements `__len__` satisfies the `Sized` interface.

Three very important specializations of `Collection` are:

- `Sequence`, formalizing the interface of built-ins like `list` and `str`
- `Mapping`, implemented by `dict`, `collections.defaultdict`, etc.
- `Set`, the interface of the `set` and `frozenset` built-in types

Only `Sequence` is `Reversible`, because sequences support arbitrary ordering of their contents, while mappings and sets do not.

Since Python 3.7, the `dict` type is officially "ordered," but that only means that the key insertion order is preserved. You cannot rearrange the keys in a `dict` however you like.

All the special methods in the `Set` ABC implement infix operators. For example, `a & b` computes the intersection of sets `a` and `b`, and is implemented in the `__and__` special method.

The next two chapters will cover standard library sequences, mappings, and sets in detail.

Now let's consider the major categories of special methods defined in the Python Data Model.

Overview of Special Methods

The "Data Model" chapter of *The Python Language Reference* lists more than 80 special method names. More than half of them implement arithmetic, bitwise, and comparison operators. As an overview of what is available, see the following tables.

Table 1-1 shows special method names, excluding those used to implement infix operators or core math functions like `abs`. Most of these methods will be covered throughout the book, including the most recent additions: asynchronous special methods such as `__anext__` (added in Python 3.5), and the class customization hook, `__init_subclass__` (from Python 3.6).

Table 1-1. Special method names (operators excluded)

Category	Method names
String/bytes representation	`__repr__` `__str__` `__format__` `__bytes__` `__fspath__`
Conversion to number	`__bool__` `__complex__` `__int__` `__float__` `__hash__` `__index__`
Emulating collections	`__len__` `__getitem__` `__setitem__` `__delitem__` `__contains__`
Iteration	`__iter__` `__aiter__` `__next__` `__anext__` `__reversed__`
Callable or coroutine execution	`__call__` `__await__`
Context management	`__enter__` `__exit__` `__aexit__` `__aenter__`
Instance creation and destruction	`__new__` `__init__` `__del__`
Attribute management	`__getattr__` `__getattribute__` `__setattr__` `__delattr__` `__dir__`
Attribute descriptors	`__get__` `__set__` `__delete__` `__set_name__`
Abstract base classes	`__instancecheck__` `__subclasscheck__`
Class metaprogramming	`__prepare__` `__init_subclass__` `__class_getitem__` `__mro_entries__`

Infix and numerical operators are supported by the special methods listed in Table 1-2. Here the most recent names are `__matmul__`, `__rmatmul__`, and `__imatmul__`, added in Python 3.5 to support the use of `@` as an infix operator for matrix multiplication, as we'll see in Chapter 16.

Table 1-2. Special method names and symbols for operators

Operator category	Symbols	Method names
Unary numeric	`-` `+` `abs()`	`__neg__` `__pos__` `__abs__`
Rich comparison	`<` `<=` `==` `!=` `>` `>=`	`__lt__` `__le__` `__eq__` `__ne__` `__gt__` `__ge__`
Arithmetic	`+` `-` `*` `/` `//` `%` `@` `divmod()` `round()` `**` `pow()`	`__add__` `__sub__` `__mul__` `__truediv__` `__floordiv__` `__mod__` `__matmul__` `__divmod__` `__round__` `__pow__`
Reversed arithmetic	(arithmetic operators with swapped operands)	`__radd__` `__rsub__` `__rmul__` `__rtruediv__` `__rfloordiv__` `__rmod__` `__rmatmul__` `__rdivmod__` `__rpow__`
Augmented assignment arithmetic	`+=` `-=` `*=` `/=` `//=` `%=` `@=` `**=`	`__iadd__` `__isub__` `__imul__` `__itruediv__` `__ifloordiv__` `__imod__` `__imatmul__` `__ipow__`
Bitwise	`&` `|` `^` `<<` `>>` `~`	`__and__` `__or__` `__xor__` `__lshift__` `__rshift__` `__invert__`
Reversed bitwise	(bitwise operators with swapped operands)	`__rand__` `__ror__` `__rxor__` `__rlshift__` `__rrshift__`

Operator category	Symbols	Method names
Augmented assignment bitwise	`&=` `|=` `^=` `<<=` `>>=`	`__iand__` `__ior__` `__ixor__` `__ilshift__` `__irshift__`

Python calls a reversed operator special method on the second operand when the corresponding special method on the first operand cannot be used. Augmented assignments are shortcuts combining an infix operator with variable assignment, e.g., `a += b`.

Chapter 16 explains reversed operators and augmented assignment in detail.

Why len Is Not a Method

I asked this question to core developer Raymond Hettinger in 2013, and the key to his answer was a quote from "The Zen of Python": "practicality beats purity." In "How Special Methods Are Used" on page 8, I described how `len(x)` runs very fast when `x` is an instance of a built-in type. No method is called for the built-in objects of CPython: the length is simply read from a field in a C struct. Getting the number of items in a collection is a common operation and must work efficiently for such basic and diverse types as `str`, `list`, `memoryview`, and so on.

In other words, `len` is not called as a method because it gets special treatment as part of the Python Data Model, just like `abs`. But thanks to the special method `__len__`, you can also make `len` work with your own custom objects. This is a fair compromise between the need for efficient built-in objects and the consistency of the language. Also from "The Zen of Python": "Special cases aren't special enough to break the rules."

If you think of `abs` and `len` as unary operators, you may be more inclined to forgive their functional look and feel, as opposed to the method call syntax one might expect in an object-oriented language. In fact, the ABC language—a direct ancestor of Python that pioneered many of its features—had an `#` operator that was the equivalent of `len` (you'd write `#s`). When used as an infix operator, written `x#s`, it counted the occurrences of `x` in `s`, which in Python you get as `s.count(x)`, for any sequence `s`.

Chapter Summary

By implementing special methods, your objects can behave like the built-in types, enabling the expressive coding style the community considers Pythonic.

A basic requirement for a Python object is to provide usable string representations of itself, one used for debugging and logging, another for presentation to end users. That is why the special methods `__repr__` and `__str__` exist in the data model.

Emulating sequences, as shown with the `FrenchDeck` example, is one of the most common uses of the special methods. For example, database libraries often return query results wrapped in sequence-like collections. Making the most of existing sequence types is the subject of Chapter 2. Implementing your own sequences will be covered in Chapter 12, when we create a multidimensional extension of the `Vector` class.

Thanks to operator overloading, Python offers a rich selection of numeric types, from the built-ins to `decimal.Decimal` and `fractions.Fraction`, all supporting infix arithmetic operators. The *NumPy* data science libraries support infix operators with matrices and tensors. Implementing operators—including reversed operators and augmented assignment—will be shown in Chapter 16 via enhancements of the `Vector` example.

The use and implementation of the majority of the remaining special methods of the Python Data Model are covered throughout this book.

Further Reading

The "Data Model" chapter of *The Python Language Reference* is the canonical source for the subject of this chapter and much of this book.

Python in a Nutshell, 3rd ed. by Alex Martelli, Anna Ravenscroft, and Steve Holden (O'Reilly) has excellent coverage of the data model. Their description of the mechanics of attribute access is the most authoritative I've seen apart from the actual C source code of CPython. Martelli is also a prolific contributor to Stack Overflow, with more than 6,200 answers posted. See his user profile at Stack Overflow.

David Beazley has two books covering the data model in detail in the context of Python 3: *Python Essential Reference*, 4th ed. (Addison-Wesley), and *Python Cookbook*, 3rd ed. (O'Reilly), coauthored with Brian K. Jones.

The Art of the Metaobject Protocol (MIT Press) by Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow explains the concept of a metaobject protocol, of which the Python Data Model is one example.

Soapbox

Data Model or Object Model?

What the Python documentation calls the "Python Data Model," most authors would say is the "Python object model." Martelli, Ravenscroft, and Holden's *Python in a Nutshell*, 3rd ed., and David Beazley's *Python Essential Reference*, 4th ed. are the best books covering the Python Data Model, but they refer to it as the "object model." On Wikipedia, the first definition of "object model" is: "The properties of objects in general in a specific computer programming language." This is what the Python Data Model is about. In this book, I will use "data model" because the documentation favors that term when referring to the Python object model, and because it is the title of the chapter of *The Python Language Reference* most relevant to our discussions.

Muggle Methods

The Original Hacker's Dictionary defines *magic* as "yet unexplained, or too complicated to explain" or "a feature not generally publicized which allows something otherwise impossible."

The Ruby community calls their equivalent of the special methods *magic methods*. Many in the Python community adopt that term as well. I believe the special methods are the opposite of magic. Python and Ruby empower their users with a rich metaobject protocol that is fully documented, enabling muggles like you and me to emulate many of the features available to core developers who write the interpreters for those languages.

In contrast, consider Go. Some objects in that language have features that are magic, in the sense that we cannot emulate them in our own user-defined types. For example, Go arrays, strings, and maps support the use brackets for item access, as in `a[i]`. But there's no way to make the `[]` notation work with a new collection type that you define. Even worse, Go has no user-level concept of an iterable interface or an iterator object, therefore its `for/range` syntax is limited to supporting five "magic" built-in types, including arrays, strings, and maps.

Maybe in the future, the designers of Go will enhance its metaobject protocol. But currently, it is much more limited than what we have in Python or Ruby.

Metaobjects

The Art of the Metaobject Protocol (AMOP) is my favorite computer book title. But I mention it because the term *metaobject protocol* is useful to think about the Python Data Model and similar features in other languages. The *metaobject* part refers to the objects that are the building blocks of the language itself. In this context, *protocol* is a synonym of *interface*. So a *metaobject protocol* is a fancy synonym for object model: an API for core language constructs.

A rich metaobject protocol enables extending a language to support new programming paradigms. Gregor Kiczales, the first author of the *AMOP* book, later became a pioneer in aspect-oriented programming and the initial author of AspectJ, an extension of Java implementing that paradigm. Aspect-oriented programming is much easier to implement in a dynamic language like Python, and some frameworks do it. The most important example is *zope.interface*, part of the framework on which the Plone content management system is built.