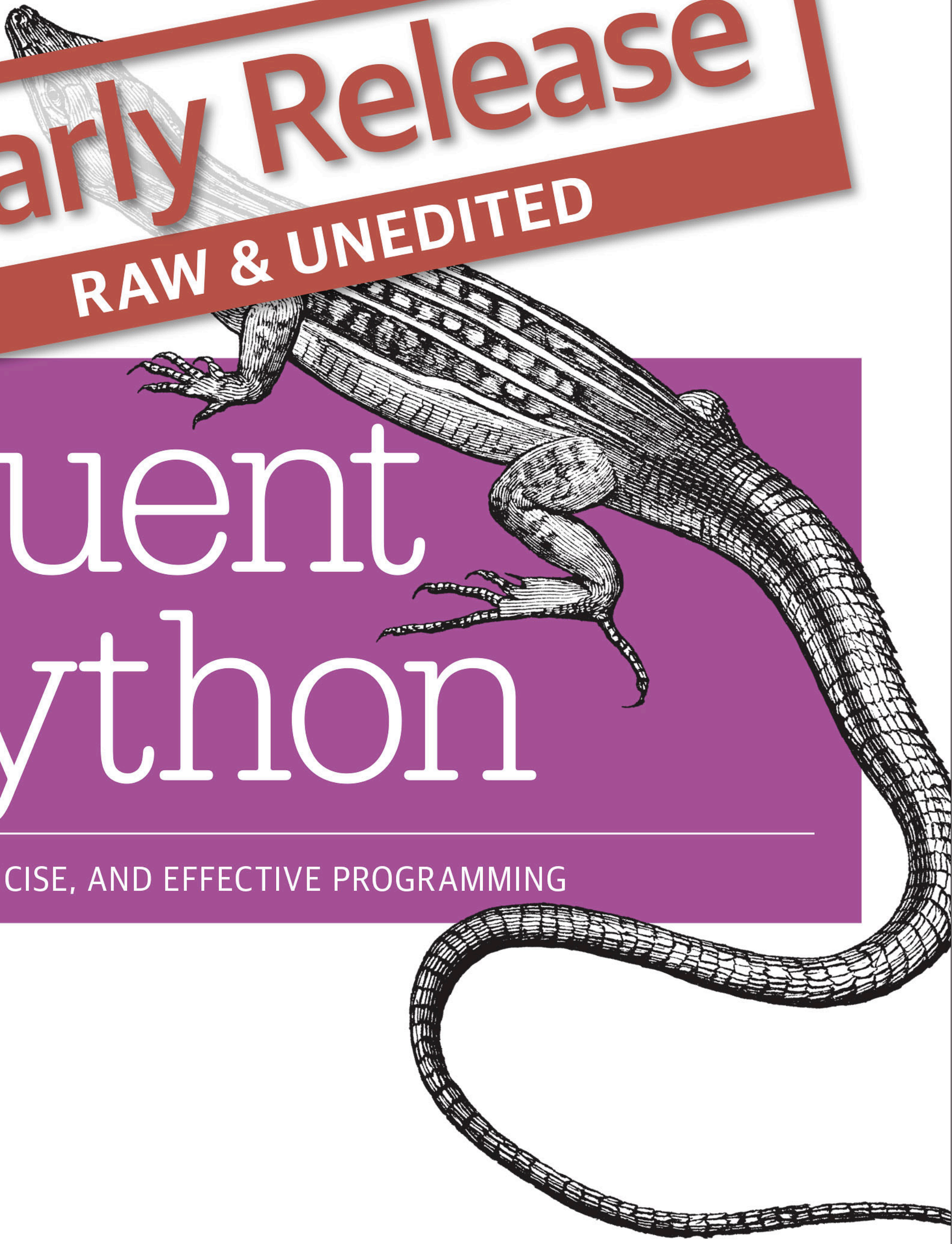
O'REILLY

Early Release

RAW & UNEDITED



Fluent Python

CLEAR, CONCISE, AND EFFECTIVE PROGRAMMING

Luciano Ramalho

Fluent Python

Luciano Ramalho

Beijing • Cambridge • Farnham • Köln • 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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Meghan Blanchette and Rachel Roumeliotis
Production Editor: FIX ME!
Copyeditor: FIX ME!
Proofreader: FIX ME!

Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

March 2015: First Edition

Revision History for the First Edition:

2014-09-30:	Early release revision 1
2014-12-05:	Early release revision 2
2014-12-18:	Early release revision 3
2015-01-27:	Early release revision 4
2015-02-27:	Early release revision 5
2015-04-15:	Early release revision 6
2015-04-21:	Early release revision 7

See *http://oreilly.com/catalog/errata.csp?isbn=9781491946008* for release details.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-491-94600-8

[?]

Para Marta, com todo o meu amor.

Table of Contents

Part I. Prologue

Part II. Data structures

Part III. Functions as objects

Part IV. Object Oriented Idioms

Part V. Control flow

Part VI. 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 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 ot 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 may end up not using those features just because they are specific to Python.

This book is not an A-Z exhaustive reference of Python. My emphasis is in 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 53.000 libraries and many of them are incredibly useful.

[1] Message to `comp.lang.python`, Dec. 23, 2002: "Acrimony in c.l.p."

XXXtodo

Who This Book Is For

This book was written for practicing Python programmers who want to become proficient in Python 3. If you know Python 2 but are willing to migrate to Python 3.4 or later, you should be fine. At this writing the majority of professional Python programmers are using Python 2, so I took special care to highlight Python 3 features that may be new to that audience.

However, Fluent Python is about making the most of Python 3.4, and I do not spell out the fixes needed to make the code work in earlier versions. Most examples should run in Python 2.7 with little or no changes, but in some cases backporting would require significant rewriting.

Having said that, I believe this book may be useful even if you must stick with Python 2.7, because the core concepts are still the same. Python 3 is not a new language, and most differences can be learned in an afternoon. What's New In Python 3.0 is a good starting point. Of course, there have been changes after Python 3.0 was released in 2009, but none as important as those in 3.0.

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 in Python 3.

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.

How This Book is Organized

The core audience for this book should not have trouble jumping directly to any chapter in this book. However, I did put some thought into their ordering.

I tried to emphasize using what is available before discussing how to build your own. For example, Chapter 2 in Part II 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 IV, where we also see how to leverage the Abstract Base Classes (ABC) from `collections.abc`. Creating your own ABCs is discussed even later in Part IV, 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 are the forward references scattered throughout the chapters. I hope these will be easier to tolerate now that you know why I chose this path.

The chapters are split in 6 parts. This is the idea behind each of them:

Part I: Prologue
: A single chapter about the Python Data Model explaining how the special methods (e.g. `__repr__`) are the key to the consistent behavior of objects of all types — in a language that is admired for its consistency. Understanding various facets of the data model is the subject of most of the rest of the book, but Chapter 1 provides a high-level overview.

Part II: Data structures
: The chapters in this part cover the use of collection types: sequences, mappings and sets, as well as the `str` versus `bytes` split — the reason for much joy for Python 3 users and much pain for Python 2 users who have not yet migrated their code bases. The main goals are to recall what is already available and to explain some behavior that is sometimes surprising, like the reordering of `dict` keys when we are not looking, or the caveats of locale-dependent Unicode string sorting. To achieve these goals, the coverage is sometimes high level and wide — when many variations of sequences and mappings are presented — and sometimes deep, for example when we dive into the hash tables underneath the `dict` and `set` types.

Part III: 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.

Part IV: Object Oriented Idioms
: Now the focus is on building classes. In part II the `class` declaration appears in few examples; part IV presents many classes. Like any OO language, Python has its particular set of features that may or may not be present in the language where you and I learned class-based programming. The chapters explain how references work, what mutability really means, the lifecycle of instances, how to build your own collections and ABCs, how to cope with multiple inheritance and how to implement operator overloading — when that makes sense.

Part V: Control flow

Covered in this part are the language constructs and libraries that go beyond sequential 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. Part V closes with high level a introduction to modern concurrency in Python with `collections.futures` — using threads and processes under the covers with the help of futures — and doing event-oriented I/O with `asyncio` — leveraging futures on top of coroutines and `yield from`.

Part VI: 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 between functions, methods and descriptors is explained. Throughout Part VI, the step by step implementation of a field validation library uncovers subtle issues the lead to the use of the advanced tools of the last 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 REPL — read-eval-print-loop.

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 TDD (Test Driven Development) 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 source code repository. 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` in the command shell of your OS.

Hardware used for timings

The book has some simple benchmarks and timings. Those tests were performed on one or the other laptop I used to write the book: a 2011 MacBook Pro 13” with a 2.7 GHz Intel Core i7 CPU, 8MB of RAM and a spinning hard disk, and a 2014 MacBook Air 13” with a 1.4 GHZ Intel Core i5 CPU, 4MB of RAM and a solid state disk. The MacBook Air has a slower CPU and less RAM, but its RAM is faster (1600 vs. 1333 MHz) and the SSD is much faster than the HD. In daily usage I can’t tell which machine is faster.

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 a section called `Soapbox` with my own perspective about Python and other languages. Feel free to skip that if you are not into such discussions. Their content is completely optional.

Python Jargon

I wanted this to be a book not only about Python but also about the culture around it. Over more than 20 years of communications, the Python community has developed its own particular lingo and acronyms. The Python jargon collects terms that have special meaning among Pythonistas.

Python version covered

I tested all the code in the book using Python 3.4 — that is, CPython 3.4, i.e. the most popular Python implementation written in C. There is only one excpetion: the sidebar “The new `@` infix operator in Python 3.5” on page 385 shows the `@` operator which is only supported by Python 3.5.

Almost all code in the book should work with any Python 3.x compatible interpreter, including PyPy3 2.4.0 which is compatible with Python 3.2.5. A notable exception are the examples using `yield from` and `asyncio`, which are only available in Python 3.3 or later.

Most code should also work with Python 2.7 with minor changes, except the Unicode-related examples in Chapter 4, and the exceptions already noted for Python 3 versions earlier than 3.3.

Conventions Used in This Book

XXXrevise

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 `con stant_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.

«Guillemets»
: Indicate that the enclosed parameter is optional, eg.: `dir(«object»)` [2]

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

[2] In the Python docs, square brackets are used for this purpose, but I have seen people confuse them with list displays.

Using Code Examples

Every script and most code snippets that appear in the book are available in the `fluentpython/example-code` repository on Github.

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/oreillymedia/title_title*.

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 a CD-ROM of 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 do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

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*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

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 *http://www.oreilly.com/catalog/<catalog page>*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

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

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

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

Acknowledgments

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 5000+ answers in Stack Overflow are a fountain of insights about the language and its proper use.

Martelli and Ravescroft 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 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 18, 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 last 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 III while Meghan was away. I enjoyed working with him, 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, giving 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. Erico 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

Prologue

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[1].
>
> — Jim Hugunin
> *creator of Jython, co-creator of AspectJ, architect of the .Net DLR*

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 have found it strange to spell `len(collection)` instead of `collection.len()`. This apparent oddity is the tip of an iceberg which, when properly understood, is the key to everything we call *Pythonic*. The iceberg is called the Python Data Model, and it describes the API that you can use to make your 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, iterators, functions, classes, context managers and so on.

While coding with any framework, you spend a lot of time implementing methods that are called by the framework. The same happens when you leverage the Python Data Model. The Python interpreter invokes special methods to perform basic object operations, often triggered by special syntax. The special method names are always spelled with leading and trailing double underscores, i.e. `__getitem__`. For example, the syntax

[1] Story of Jython, written as a foreword to *Jython Essentials* (O'Reilly, 2002), by Samuele Pedroni and Noel Rappin.

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

The special method names allow your objects to implement, support and interact with basic language constructs such as:

- iteration;
- collections;
- attribute access;
- operator overloading;
- function and method invocation;
- object creation and destruction;
- string representation and formatting;
- managed contexts (i.e. **`with`** blocks);

Magic and dunder

The term *magic method* is slang for special method, but when talking about a specific method like `__getitem__`, some Python developers take the shortcut of saying "under-under-getitem" which is ambiguous, since the syntax `__x` has another special meaning[2]. But being precise and pronouncing "under-under-getitem-under-under" is tiresome, so I follow the lead of author and teacher Steve Holden and say "dunder-getitem". All experienced Pythonistas understand that shortcut. As a result, the special methods are also known as *dunder methods* [3].

A Pythonic Card Deck

The following is a very simple example, but it demonstrates the power of implementing just two special methods, `__getitem__` and `__len__`.

Example 1-1 is a class to represent a deck of playing cards:

[2] See "Private and "protected" attributes in Python" on page 263

[3] I personally first heard "dunder" from Steve Holden. The English language Wikipedia credits Mark Johnson and Tim Hochberg for the first written records of "dunder" in responses to the question "How do you pronounce __ (double underscore)?" in the python-list in September 26, 2002: Johnson's message; Hochberg's (11 minutes later).

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

```python
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. Since Python 2.6, `namedtuple` can be used 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, should be as easy as `deck[0]` or `deck[-1]`, and this is what the `__getitem__` method provides.

```
>>> 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 just 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:

1. The 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?")
2. 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 on 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')
...
```

The deck can also be iterated 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, the Python console listings in this book were extracted from doctests to insure accuracy. When the output was too long, the elided part is marked by an ellipsis `...` like in the last line above. In such cases, we 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 directives 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: spades (highest), then 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 ommitted)
Card(rank='A', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='A', suit='spades')
```

Although FrenchDeck implicitly inherits from `object`[4], 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 — like iteration and slicing — and from the standard library, as shown by the examples using `ran`

[4] In Python 2 you'd have to be explicit and write `FrenchDeck(object)`, but that's the default in Python 3.

`dom.choice`, `reversed` and `sorted`. Thanks to composition, the `__len__` and `__getitem__` implementations can hand off 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 11 that will be fixed 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__` instance method you implemented.

But for built-in types like `list`, `str`, `bytearray` etc., the interpreter takes a shortcut: the CPython implementation of `len()` actually returns the value of the `ob_size` field in the `PyVarObject` C struct that represents any variable-sized built-in object in memory. This is much faster than calling a method.

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.

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, such as `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 "A closer look at the `iter` function" on page 438 in Chapter 14.

Avoid creating arbitrary, custom attributes with the `__foo__` syntax because such names may acquire special meanings in the future, even if they are unused today.

Emulating numeric types

Several special methods allow user objects to respond to operators such as +. We will cover that in more detail in Chapter 13, but here our goal is to further illustrate the use of special methods through another simple example.

We will implement a class to represent 2-dimensional vectors, i.e. Euclidean vectors like those used in math and physics (see Figure 1-1).

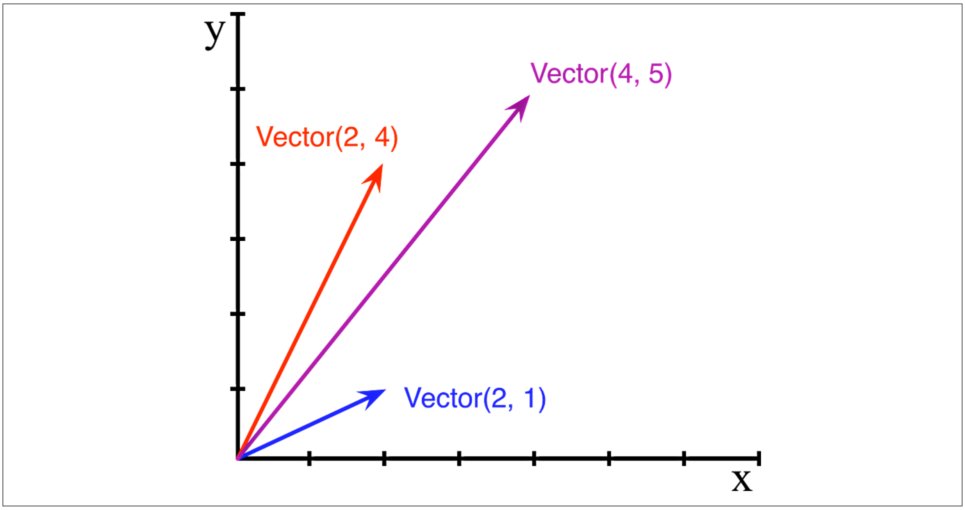


Figure 1-1. Example of 2D vector addition. `Vector(2, 4) + Vector(2, 1)` *results in* `Vector(4, 5)`*.*

The built-in `complex` type can be used to represent 2D vectors, but our class can be extended to represent n-dimensional vectors. We will do that in Chapter 14.

We will start by designing the API for such a class by writing a simulated console session which we can use later as 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 produces a `Vector` result which is displayed in a friendly manner in 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 produce 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 2D vector class.

```
from math import hypot

class Vector:

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

    def __repr__(self):
        return 'Vector(%r, %r)' % (self.x, self.y)

    def __abs__(self):
        return 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)
```

Note that although we implemented four special methods (apart from `__init__`), none of them is directly called within the class or in the typical usage of the class illustrated by the console listings. As mentioned before, the Python interpreter is the only frequent caller of most special methods. In the next sections we discuss the code for each special method.

String representation

The `__repr__` special method is called by the `repr` built-in to get string representation of the object for inspection. If we did not implement `__repr__`, vector instances would be shown in the console like `<Vector object at 0x10e100070>`.

The interactive console and debugger call `repr` on the results of the expressions evaluated, as does the `'%r'` place holder in classic formatting with `%` operator, and the `!r` conversion field in the new Format String Syntax used in the `str.format` method[5].

Note that in our `__repr__` implementation we used `%r` to obtain the standard representation of the attributes to be displayed. This is good practice, as 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 constructors arguments must be numbers, not `str`.

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

Contrast `__repr__` with with `__str__`, which is called by the `str()` constructor and implicitly used by the `print` function. `__str__` should return a string suitable for display to end-users.

If you only implement one of these special methods, choose `__repr__`, because when no custom `__str__` is available, Python will call `__repr__` as a fallback.

Difference between `__str__` and `__repr__` in Python is a StackOverflow question with excellent contributions from Pythonistas Alex Martelli and Martijn Pieters.

Arithmetic operators

Example 1-2 implements two operators: `+` and `*`, to show basic usage of `__add__` and `__mul__`. Note that in both cases, the methods create and return a new instance of

[5] Speaking of the `%` operator and the `str.format` method, the reader will notice I use both in this book, as does the Python community at large. I am increasingly favoring the more powerful `str.format`, but I am aware many Pythonistas prefer the simpler `%`, so we'll probably see both in Python source code for the foreseeable future.

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 13.

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 multiplication. We will fix that with the special method __rmul__ in Chapter 13.

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 always returns 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.

Note how the special method __bool__ allows your objects to be consistent with 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.

Overview of special methods

The Data Model page of the Python Language Reference lists 83 special method names, 47 of which are used to implement arithmetic, bitwise and comparison operators.

As an overview of what is available, see Table 1-1 and Table 1-2.

The grouping shown in the following tables is not exactly the same as in the official documentation.

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

category	method names
string/bytes representation	__repr__, __str__, __format__, __bytes__
conversion to number	__abs__, __bool__, __complex__, __int__, __float__, __hash__, __index__
emulating collections	__len__, __getitem__, __setitem__, __delitem__, __contains__
iteration	__iter__, __reversed__, __next__
emulating callables	__call__
context management	__enter__, __exit__
instance creation and destruction	__new__, __init__, __del__
attribute management	__getattr__, __getattribute__, __setattr__, __delattr__, __dir__
attribute descriptors	__get__, __set__, __delete__
class services	__prepare__, __instancecheck__, __subclasscheck__

Table 1-2. Special method names for operators.

category	method names and related operators
unary numeric operators	__neg__ -, __pos__ +, __abs__ abs()
rich compartison operators	__lt__ >, __le__ <=, __eq__ ==, __ne__ !=, __gt__ >, __ge__ >=
arithmetic operators	__add__ +, __sub__ -, __mul__ *, __truediv__ /, __floordiv__ //, __mod__ %, __divmod__ divmod() , __pow__ ** or pow(), __round__ round()
reversed arithmetic operators	__radd__, __rsub__, __rmul__, __rtruediv__, __rfloordiv__, __rmod__, __rdivmod__, __rpow__
augmented assignment arithmetic operators	__iadd__, __isub__, __imul__, __itruediv__, __ifloordiv__, __imod__, __ipow__
bitwise operators	__invert__ ~, __lshift__ <<, __rshift__ >>, __and__ &, __or__ |, __xor__ ^
reversed bitwise operators	__rlshift__, __rrshift__, __rand__, __rxor__, __ror__
augmented assignment bitwise operators	__ilshift__, __irshift__, __iand__, __ixor__, __ior__

The reversed operators are fallbacks used when operands are swapped (`b * a` instead of `a * b`), while augmented assignment are shortcuts combining an infix operator with variable assignment (`a = a * b` becomes `a *= b`). Chapter 13 explains both 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` etc.

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 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 a OO language. In fact, the ABC language — a direct ancestor of Python which 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 widely used applications of the special methods. Making the most of sequence types is the subject of Chapter 2, and implementing your own sequence will be covered in Chapter 10 we will create a multi-dimensional 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. Implementing operators, including reversed operators and augmented assignment will be shown in Chapter 13 via enhancements of the `Vector` example.

The use and implementation of the majority of the remaining special methods of the Python Data Model is 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, 2nd Edition, by Alex Martelli, has excellent coverage of the Data Model. As I write this, the most recent edition of the *Nutshell* book is from 2006 and focuses on Python 2.5, but there were very few changes in the Data Model since then, and Martelli's 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 5000 answers posted. See his user profile at *http://stackoverflow.com/users/95810/alex-martelli*.

David Beazley has two books covering the Data Model in detail in the context of Python 3: *Python Essential Reference, 4th Edition*, and *Python Cookbook, 3rd Edition*, co-authored with Brian K. Jones.

The Art of the Metaobject Protocol (AMOP), by Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow explains the concept of a MOP (Meta Object 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". Alex Martelli's *Python in a Nutshell 2e* and David Beazley's *Python Essential Reference 4e* are the best books covering the "Python Data Model", but they always refer to it as the "object model". In the English language 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 that is how the Python object model is called in the documentation, and is the title of the chapter of the Python Language Reference most relevant to our discussions.

Magic methods

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 actually the opposite of magic. Python and Ruby are the same in this regard: both empower their users with a rich metaobject protocol which is not magic, but enables users to leverage the same tools available to core developers.

In contrast, consider JavaScript. Objects in that language have features that are magic, in the sense that you cannot emulate them in your own user-defined objects. For example, before JavaScript 1.8.5 you could not define read-only attributes in your JavaScript objects, but some built-in objects always had read-only attributes. In JavaScript, read-only attributes were “magic”, requiring supernatural powers that a user of the language did not have until ECMAScript 5.1 came out in 2009. The metaobject protocol of JavaScript is evolving, but historically it has been more limited than those of Python and Ruby.

Metaobjects

The Art of The Metaobject Procotol (AMOP) is my favorite computer book title. Less subjectively, 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, *prococol* 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 several frameworks do it, but the most important is `zope.interface`, which is briefly discussed in the **Further reading** section of Chapter 11.

PART II

Data structures

CHAPTER 2

An array of sequences

> As you may have noticed, several of the operations mentioned work equally for texts, lists and tables. Texts, lists and tables together are called *trains*. […] The FOR command also works generically on trains[1].
>
> — Geurts, Meertens and Pemberton
> *ABC Programmer's Handbook*

Before creating Python, Guido was a contributor to the ABC language — a 10-year research project to design a programming environment for beginners. ABC introduced many ideas we now consider "Pythonic": generic operations on sequences, built-in tuple and mapping types, structure by indentation, strong typing without variable declarations etc. It's no accident that Python is so user friendly.

Python inherited from ABC the uniform handling of sequences. Strings, lists, byte sequences, arrays, XML elements and database results share a rich set of common operations including iteration, slicing, sorting and concatenation.

Understanding the variety of sequences available in Python saves us from reinventing the wheel, and their common interface inspires us to create APIs that properly support and leverage existing and future sequence types.

Most of the discussion in this chapter applies to sequences in general, from the familiar `list` to the `str` and `bytes` types that are new in Python 3. Specific topics on lists, tuples, arrays and queues are also covered here, but the focus on Unicode strings and byte sequences is deferred to Chapter 4. Also, the idea here is to cover sequence types that are ready to use. Creating your own sequence types is the subject of Chapter 10.

[1] Leo Geurts, Lambert Meertens and Steven Pemberton, *ABC Programmer's Handbook*, p. 8.

Overview of built-in sequences

The standard library offers a rich selection of sequence types implemented in C:

Container sequences
: `list`, `tuple` and `collections.deque` can hold items of different types.

Flat sequences
: `str`, `bytes`, `bytearray`, `memoryview` and `array.array` hold items of one type.

Container sequences hold references to the objects they contain, which may be of any type, while *flat sequences* physically store the value of each item within its own memory space, and not as distinct objects. Thus, flat sequences are more compact, but they are limited to holding primitive values like characters, bytes and numbers

Another way of grouping sequence types is by mutability:

Mutable sequences
: `list`, `bytearray`, `array.array`, `collections.deque` and `memoryview`

Immutable sequences
: `tuple`, `str` and `bytes`

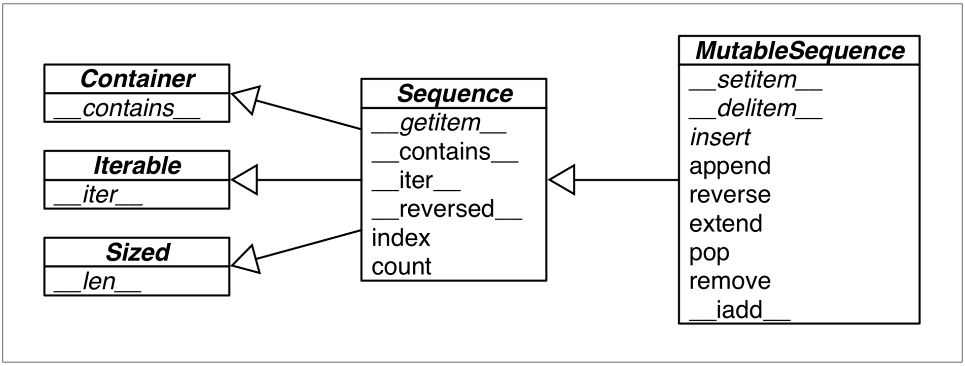


Figure 2-1. UML class diagram for some classes from `collections.abc`. Superclasses are on the left; inheritance arrows point from subclasses to superclasses. Names in italic are abstract classes and abstract methods.

Figure 2-1 helps visualize how mutable sequences differ from immutable ones, while also inheriting several methods from them. Note that the built-in concrete sequence types do not actually subclass the `Sequence` and `MutableSequence` *ABCs* (Abstract Base Classes) depicted, but the ABCs are still useful as a formalization of what functionality to expect from a full-featured sequence type.

Keeping in mind these common traits — mutable versus immutable; container versus flat — is helpful to extrapolate what you know about one sequence type to others.

The most fundamental sequence type is the `list` — mutable and mixed-type. I am sure you are comfortable handling them, so we'll jump right into list comprehensions, a powerful way of building lists which is somewhat underused because the syntax may be unfamiliar. Mastering list comprehensions opens the door to generator expressions, which — among other uses — can produce elements to fill-up sequences of any type. Both are the subject of the next section.

List comprehensions and generator expressions

A quick way to build a sequence is using a list comprehension (if the target is a `list`) or a generator expression (for all other kinds of sequences). If you are not using these syntactic forms on a daily basis, I bet you are missing opportunities to write code that is more readable and often faster at the same time.

If you doubt my claim that these constructs are "more readable", please read on. I'll try to convince you.

For brevity, many Python programmers call list comprehensions *listcomps*, and generator expressions *genexps*. I will use these words as well.

List comprehensions and readability

Here is a test: which do you find easier to read, Example 2-1 or Example 2-2?

Example 2-1. Build a list of Unicode codepoints from a string.

```
>>> symbols = '$¢£¥€¤'
>>> codes = []
>>> for symbol in symbols:
...     codes.append(ord(symbol))
...
>>> codes
[36, 162, 163, 165, 8364, 164]
```

Example 2-2. Build a list of Unicode codepoints from a string, take two.

```
>>> symbols = '$¢£¥€¤'
>>> codes = [ord(symbol) for symbol in symbols]
>>> codes
[36, 162, 163, 165, 8364, 164]
```

Anybody who knows a little bit of Python can read Example 2-1. However, after learning about listcomps I find Example 2-2 more readable because its intent is explicit.

A for loop may be used to do lots of different things: scanning a sequence to count or pick items, computing aggregates (sums, averages), or any number of other processing. The code in Example 2-1 is building up a list. In contrast, a listcomp is meant to do one thing only: to build a new list.

Of course, it is possible to abuse list comprehensions to write truly incomprehensible code. I've seen Python code with listcomps used just to repeat a block of code for its side effects. Please don't use that syntax if you are not doing something with the produced list. Also, try to keep it short. If the list comprehension spans more than two lines, it is probably best to break it apart or rewrite as a plain old for loop. Use your best judgment: for Python as for English, there are no hard and fast rules for clear writing.

Syntax tip
In Python code, line breaks are ignored inside pairs of [], {} or (). So you can build multi-line lists, listcomps, genexps, dictionaries etc. without using the ugly \ line continuation escape.

Listcomps no longer leak their variables

In Python 2.x, variables assigned in the for clauses in list comprehensions were set in the surrounding scope, sometimes with tragic consequences. See the following Python 2.7 console session:

```
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 'my precious'
>>> dummy = [x for x in 'ABC']
>>> x
'C'
```

As you can see, the initial value of x was clobbered. This no longer happens in Python 3.

List comprehensions, generator expressions and their siblings set and dict comprehensions now have their own local scope, like functions. Variables assigned within the expression are local, but variables in the surrounding scope can still be referenced. Even better, the local variables do not mask the variables from the surrounding scope.

This is Python 3:

```
>>> x = 'ABC'
>>> dummy = [ord(x) for x in x]
>>> x  ❶
```

```
'ABC'
>>> dummy  ❷
[65, 66, 67]
>>>
```

❶ The value of x is preserved.

❷ The list comprehension produces the expected list.

List comprehensions build lists from sequences or any other iterable type by filtering and transforming items. The filter and map built-ins can be composed to do the same, but readability suffers, as we will see next.

Listcomps versus map and filter

Listcomps do everything the map and filter functions do, without the contortions of the functionally challenged Python lambda. Consider Example 2-3.

Example 2-3. The same list built by a listcomp and a map/filter composition.

```
>>> symbols = '$¢£¥€¤'
>>> beyond_ascii = [ord(s) for s in symbols if ord(s) > 127]
>>> beyond_ascii
[162, 163, 165, 8364, 164]
>>> beyond_ascii = list(filter(lambda c: c > 127, map(ord, symbols)))
>>> beyond_ascii
[162, 163, 165, 8364, 164]
```

I used to believe that map and filter were faster than the equivalent listcomps, but Alex Martelli pointed out that's not the case — at least not in the examples above[2].

I'll have more to say about map and filter in Chapter 5. Now we turn to the use of listcomps to compute cartesian products: a list containing tuples build from all items from two or more lists.

Cartesian products

Listcomps can generate lists from the cartesian product of two or more iterables. The items that make up the cartesian product are tuples made from items from every input iterable. The resulting list has a length equal to the lengths of the input iterables multiplied. See Figure 2-2.

[2] The 02-array-seq/listcomp_speed.py script in the Fluent Python code repository is a simple speed test comparing listcomp with filter/map.

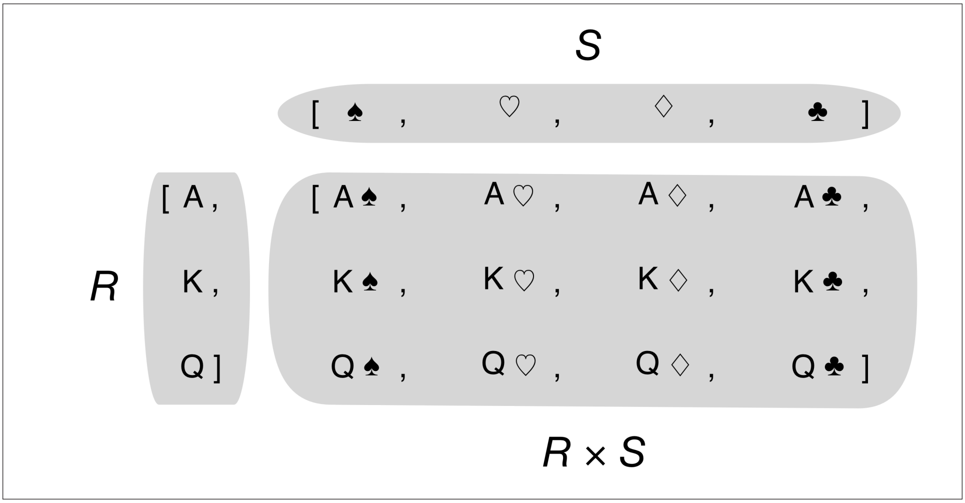


Figure 2-2. The cartesian product of a sequence of three card ranks and a sequence of four suits results in a sequence of twelve pairings.

For example, imagine you need to produce a list of t-shirts available in two colors and three sizes. Example 2-4 shows how to produce that list using a listcomp. The result has 6 items.

Example 2-4. Cartesian product using a list comprehension.

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> tshirts = [(color, size) for color in colors for size in sizes]  ❶
>>> tshirts
[('black', 'S'), ('black', 'M'), ('black', 'L'), ('white', 'S'),
 ('white', 'M'), ('white', 'L')]
>>> for color in colors:  ❷
...     for size in sizes:
...         print((color, size))
...
('black', 'S')
('black', 'M')
('black', 'L')
('white', 'S')
('white', 'M')
('white', 'L')
>>> tshirts = [(color, size) for size in sizes      ❸
...                          for color in colors]
>>> tshirts
[('black', 'S'), ('white', 'S'), ('black', 'M'), ('white', 'M'),
 ('black', 'L'), ('white', 'L')]
```

❶ This generates a list of tuples arranged by color, then size.

❷ Note how the resulting list is arranged as if the for loops were nested in the same order as they appear in the listcomp.

❸ To get items arranged by size, then color, just rearrange the `for` clauses; adding a line break to the listcomp makes it easy to see how the result will be ordered.

In Example 1-1 (Chapter 1), the following expression was used to initialize a card deck with a list made of 52 cards from all 13 suits and 4 ranks, grouped by suit:

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

Listcomps are a one-trick pony: they build lists. To fill-up other sequence types, a genexp is the way to go. The next section is a brief look at genexps in the context of building non-list sequences.

Generator expressions

To initialize tuples, arrays and other types of sequences, you could also start from a listcomp but a genexp saves memory because it yields items one by one using the iterator protocol instead of building a whole list just to feed another constructor.

Genexps use the same syntax as listcomps, but are enclosed in parenthesis rather than brackets.

Example 2-5 shows basic usage of genexps to build a `tuple` and an `array`.

Example 2-5. Initializing a tuple and an array from a generator expression.

```
>>> symbols = '$¢£¥€¤'
>>> tuple(ord(symbol) for symbol in symbols)  ❶
(36, 162, 163, 165, 8364, 164)
>>> import array
>>> array.array('I', (ord(symbol) for symbol in symbols))  ❷
array('I', [36, 162, 163, 165, 8364, 164])
```

❶ If the generator expression is the single argument in a function call, there is no need to duplicate the enclosing parenthesis.

❷ The `array` constructor takes two arguments, so the parenthesis around the generator expression are mandatory[3].

Example 2-6 uses a genexp with a cartesian product to print out a roster of t-shirts of two colors in three sizes. In contrast with Example 2-4, here the 6-item list of t-shirts is never built in memory: the generator expression feeds the `for` loop producing one item at a time. If the two lists used in the cartesian product had a thousand items each, using

[3] The first argument of the `array` constructor defines the storage type used for the numbers in the array, as we'll see in "Arrays" on page 48

a generator expression would save the expense of building a list with a million items just to feed the for loop.

Example 2-6. Cartesian product in a generator expression.

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> for tshirt in ('%s %s' % (c, s) for c in colors for s in sizes):  ❶
...     print(tshirt)
...
black S
black M
black L
white S
white M
white L
```

❶ The generator expression yields items one by one; a list with all 6 t-shirt variations is never produced in this example.

Chapter 14 is devoted to explaining how generators work in detail. Here the idea was just to show the use of generator expressions to initialize sequences other than lists, or to produce output that you don't need to keep in memory.

Now we move on to the other fundamental sequence type in Python: the `tuple`.

Tuples are not just immutable lists

Some introductory texts about Python present tuples as "immutable lists", but that is short selling them. Tuples do double-duty: they can be used as immutable lists and also as records with no field names. This use is sometimes overlooked, so we will start with that.

Tuples as records

Tuples hold records: each item in the tuple holds the data for one field and the position of the item gives its meaning.

If you think of a tuple just as an immutable list, the quantity and the order of the items may or may not be important, depending on the context. But when using a tuple as a collection of fields, the number of items is often fixed and their order is always vital.

Example 2-7 shows tuples being used as records. Note that in every expression below, sorting the tuple would destroy the information because the meaning of each data item is given by its position in the tuple.

Example 2-7. Tuples used as records.

```
>>> lax_coordinates = (33.9425, -118.408056)  ❶
>>> city, year, pop, chg, area = ('Tokyo', 2003, 32450, 0.66, 8014)  ❷
>>> traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'),  ❸
...     ('ESP', 'XDA205856')]
>>> for passport in sorted(traveler_ids):  ❹
...     print('%s/%s' % passport)   ❺
...
BRA/CE342567
ESP/XDA205856
USA/31195855
>>> for country, _ in traveler_ids:  ❻
...     print(country)
...
USA
BRA
ESP
```

❶ Latitude and longitude of the Los Angeles International Airport.

❷ Data about Tokyo: name, year, population (millions), population change (%), area (km^2).

❸ A list of tuples of the form (country_code, passport_number).

❹ As we iterate over the list, `passport` is bound to each tuple.

❺ The `%` formatting operator understands tuples and treats each item as a separate field.

❻ The `for` loop knows how to retrieve the items of a tuple separately — this is called “unpacking”. Here we are not interested in the second item, so it’s assigned to _, a dummy variable.

Tuples work well as records because of the tuple unpacking mechanism — our next subject.

Tuple unpacking

In Example 2-7 we assigned `('Tokyo', 2003, 32450, 0.66, 8014)` to `city, year, pop, chg, area` in a single statement. Then, in the last line, the `%` operator assigned each item in the `passport` tuple to one slot in the format string in the `print` argument. Those are two examples of *tuple unpacking*.

Tuple unpacking works with any iterable object. The only requirement is that the iterable yields exactly one item per variable in the receiving tuple, unless you use a star `*` to capture excess items as explained in "Using * to grab excess items" on page 29. The term *tuple unpacking* is widely used by Pythonistas, but *iterable unpacking* is gaining traction, as in the title of PEP 3132 — Extended Iterable Unpacking.

The most visible form of tuple unpacking is *parallel assignment*, that is, assigning items from an iterable to a tuple of variables, as you can see in this example:

```
>>> lax_coordinates = (33.9425, -118.408056)
>>> latitude, longitude = lax_coordinates  # tuple unpacking
>>> latitude
33.9425
>>> longitude
-118.408056
```

An elegant application of tuple unpacking is swapping the values of variables without using a temporary variable:

```
>>> b, a = a, b
```

Another example of tuple unpacking is prefixing an argument with a star when calling a function:

```
>>> divmod(20, 8)
(2, 4)
>>> t = (20, 8)
>>> divmod(*t)
(2, 4)
>>> quotient, remainder = divmod(*t)
>>> quotient, remainder
(2, 4)
```

The code above also shows a further use of tuple unpacking: enabling functions to return multiple values in a way that is convenient to the caller. For example, the `os.path.split()` function builds a tuple `(path, last_part)` from a filesystem path.

```
>>> import os
>>> _, filename = os.path.split('/home/luciano/.ssh/idrsa.pub')
>>> filename
'idrsa.pub'
```

Sometimes when we only care about certain parts of a tuple when unpacking, a dummy variable like _ is used as placeholder, as in the example above.

If you write internationalized software, _ is not a good dummy variable because it is traditionally used as an alias to the `gettext.gettext` function, as recommended in the `gettext` module documentation. Otherwise, it's a nice name for placeholder variable.

Another way of focusing on just some of the items when unpacking a tuple is to use the *, as we'll see right away.

Using * to grab excess items

Defining function parameters with `*args` to grab arbitrary excess arguments is a classic Python feature.

In Python 3 this idea was extended to apply to parallel assignment as well.

```
>>> a, b, *rest = range(5)
>>> a, b, rest
(0, 1, [2, 3, 4])
>>> a, b, *rest = range(3)
>>> a, b, rest
(0, 1, [2])
>>> a, b, *rest = range(2)
>>> a, b, rest
(0, 1, [])
```

In the context of parallel assignment, the * prefix can be applied to exactly one variable, but it can appear in any position:

```
>>> a, *body, c, d = range(5)
>>> a, body, c, d
(0, [1, 2], 3, 4)
>>> *head, b, c, d = range(5)
>>> head, b, c, d
([0, 1], 2, 3, 4)
```

Finally, a powerful feature of tuple unpacking is that it works with nested structures.

Nested tuple unpacking

The tuple to receive an expression to unpack can have nested tuples, like `(a, b, (c, d))` and Python will do the right thing if the expression matches the nesting structure.

Example 2-8. Unpacking nested tuples to access the longitude.

```
metro_areas = [
    ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),   # ❶
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
    ('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
]

print('{:15} | {:^9} | {:^9}'.format('', 'lat.', 'long.'))
fmt = '{:15} | {:9.4f} | {:9.4f}'
for name, cc, pop, (latitude, longitude) in metro_areas:  # ❷
    if longitude <= 0:  # ❸
        print(fmt.format(name, latitude, longitude))
```

❶ Each tuple holds a record with four fields, the last of which is a coordinate pair.

❷ By assigning the last field to a tuple, we unpack the coordinates.

❸ `if longitude <= 0:` limits the output to metropolitan areas in the Western hemisphere.

The output of Example 2-8 is:

```
                |   lat.    |   long.
Mexico City     |   19.4333 |  -99.1333
New York-Newark |   40.8086 |  -74.0204
Sao Paulo       |  -23.5478 |  -46.6358
```

Before Python 3, it used to be possible to define functions with nested tuples in the formal parameters, eg. `def fn(a, (b, c), d):`. This is no longer supported in Python 3 function definitions, for practical reasons explained in PEP 3113 — Removal of Tuple Parameter Unpacking. To be clear: nothing changed from the perspective of users calling a function. The restriction applies only to the definition of functions.

As designed, tuples are very handy. But there is a missing feature when using them as records: sometimes it is desirable to name the fields. That is why the `namedtuple` function was invented. Read on.

Named tuples

The `collections.namedtuple` function is a factory that produces subclasses of `tuple` enhanced with field names and a class name — which helps debugging.

Instances of a class that you build with `namedtuple` take exactly the same amount of memory as tuples because the field names are stored in the class. They use less memory than a regular object because they do store attributes in a per-instance `__dict__`.

Recall how we built the `Card` class in Example 1-1 in Chapter 1:

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

Example 2-9 shows how we could define a named tuple to hold information about a city:

Example 2-9. Defining and using a named tuple type

```
>>> from collections import namedtuple
>>> City = namedtuple('City', 'name country population coordinates')  ❶
>>> tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667))  ❷
>>> tokyo
City(name='Tokyo', country='JP', population=36.933, coordinates=(35.689722, 139.691667))
>>> tokyo.population  ❸
36.933
>>> tokyo.coordinates
(35.689722, 139.691667)
>>> tokyo[1]
'JP'
```

❶ Two parameters are required to create a named tuple: a class name and a list of field names, which can be given as an iterable of strings or as a single space-delimited string.

❷ Data must be passed as positional arguments to the constructor (in contrast, the `tuple` constructor takes a single iterable).

❸ You can access the fields by name or position.

A named tuple type has a few attributes in addition to those inherited from `tuple`. Example 2-10 shows the most useful: the `_fields` class attribute, the class method `_make(iterable)` and the `_asdict()` instance method.

Example 2-10. Named tuple attributes and methods (continued from Example 2-9)

```
>>> City._fields  ❶
('name', 'country', 'population', 'coordinates')
>>> LatLong = namedtuple('LatLong', 'lat long')
>>> delhi_data = ('Delhi NCR', 'IN', 21.935, LatLong(28.613889, 77.208889))
>>> delhi = City._make(delhi_data)  ❷
>>> delhi._asdict()  ❸
OrderedDict([('name', 'Delhi NCR'), ('country', 'IN'), ('population',
21.935), ('coordinates', LatLong(lat=28.613889, long=77.208889))])
>>> for key, value in delhi._asdict().items():
        print(key + ':', value)

name: Delhi NCR
country: IN
population: 21.935
coordinates: LatLong(lat=28.613889, long=77.208889)
>>>
```

❶ `_fields` is a tuple with the field names of the class.

❷ `_make()` lets you instantiate a named tuple from an iterable; `City(*delhi_data)` would do the same.

❸ `_asdict()` returns a `collections.OrderedDict` built from the named tuple instance. That can be used to produce a nice display of city data.

Now that we've explored the power of tuples as records, we can consider their second role as an immutable variant of the `list` type.

Tuples as immutable lists

When using a `tuple` as an immutable variation of `list`, it helps to know how similar they actually are. As you can see in Table 2-1, all `list` methods that do not involve adding or removing items are supported by `tuple` with one exception — `tuple` lacks the `__reversed__` method, but that is just for optimization; `reversed(my_tuple)` works without it.

Table 2-1. Methods and attributes found in `list` or `tuple` (methods implemented by `object` are omitted for brevity).

	list	tuple	
`s.__add__(s2)`	●	●	`s + s2` — concatenation
`s.__iadd__(s2)`	●		`s += s2` — in-place concatenation
`s.append(e)`	●		append one element after last
`s.clear()`	●		delete all items
`s.__contains__(e)`	●	●	`e in s`
`s.copy()`	●		shallow copy of the list
`s.count(e)`	●	●	count occurrences of an element
`s.__delitem__(p)`	●		remove item at position `p`
`s.extend(it)`	●		append items from iterable `it`
`s.__getitem__(p)`	●	●	`s[p]` — get item at position
`s.__getnewargs__()`		●	support for optimized serialization with `pickle`
`s.index(e)`	●	●	find position of first occurrence of `e`
`s.insert(p, e)`	●		insert element `e` before the item at position `p`
`s.__iter__()`	●	●	get iterator
`s.__len__()`	●	●	`len(s)` — number of items
`s.__mul__(n)`	●	●	`s * n` — repeated concatenation
`s.__imul__(n)`	●		`s *= n` — in-place repeated concatenation
`s.__rmul__(n)`	●	●	`n * s` — reversed repeated concatenation[a]

	list	tuple	
s.pop(«p»)	●		remove and return last item or item at optional position p
s.remove(e)	●		remove first occurrence of element e by value
s.reverse()	●		reverse the order of the items in-place
s.__reversed__()	●		get iterator to scan items from last to first
s.__setitem__(p, e)	●		s[p] = e — put e in position p, overwriting existing item
s.sort(«key», «reverse»)	●		sort items in place with optional keyword arguments key and reverse

[a] Reversed operators are explained in Chapter 13.

Every Python programmer knows that sequences can be sliced using the s[a:b] syntax. We now turn to some less well-known facts about slicing.

Slicing

A common feature of list, tuple, str and all sequence types in Python is the support of slicing operations, which are more powerful than most people realize.

In this section we describe the *use* of these advanced forms of slicing. Their implementation in a user-defined class will be covered in Chapter 10, in keeping with our philosophy of covering ready-to-use classes in this part of the book, and creating new classes in Part IV.

Why slices and range exclude the last item

The Pythonic convention of excluding the last item in slices and ranges works well with the zero-based indexing used in Python, C and many other languages. Some convenient features of the convention are:

- It's easy to see the length of a slice or range when only the stop position is given: range(3) and my_list[:3] both produce three items.
- It's easy to compute the length of a slice or range when start and stop are given: just subtract stop - start.
- It's easy to split a sequence in two parts at any index x, without overlapping: simply get my_list[:x] and my_list[x:]. See for example:

```
>>> l = [10, 20, 30, 40, 50, 60]
>>> l[:2]  # split at 2
[10, 20]
>>> l[2:]
[30, 40, 50, 60]
>>> l[:3]  # split at 3
[10, 20, 30]
```

```
>>> l[3:]
[40, 50, 60]
```

But the best arguments for this convention were written by the Dutch computer scientist Edsger W. Dijkstra. Please see the last reference in "Further reading" on page 58.

Now let's take a close look at how Python interprets slice notation.

Slice objects

This is no secret, but worth repeating just in case: `s[a:b:c]` can be used to specify a stride or step `c`, causing the resulting slice to skip items. The stride can also be negative, returning items in reverse. Three examples make this clear:

```
>>> s = 'bicycle'
>>> s[::3]
'bye'
>>> s[::-1]
'elcycib'
>>> s[::-2]
'eccb'
```

Another example was shown in Chapter 1 when we used `deck[12::13]` to get all the aces in the unshuffled deck:

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

The notation `a:b:c` is only valid within `[]` when used as the indexing or subscript operator, and it produces a slice object: `slice(a, b, c)`. As we will see in "How slicing works" on page 283, to evaluate the expression `seq[start:stop:step]`, Python calls `seq.__getitem__(slice(start, stop, step))`. Even if you are not implementing your own sequence types, knowing about slice objects is useful because it lets you assign names to slices, just like spreadsheets allow naming of cell ranges.

Suppose you need to parse flat-file data like the invoice on top of Example 2-11. Instead of filling your code with hard-coded slices, you can name them. See how readable this makes the for loop at the end of the example:

Example 2-11. Line items from a flat file invoice

```
>>> invoice = """
... 0.....6.................................40........52...55........
... 1909  Pimoroni PiBrella                     $17.50    3    $52.50
... 1489  6mm Tactile Switch x20                 $4.95    2     $9.90
... 1510  Panavise Jr. - PV-201                 $28.00    1    $28.00
... 1601  PiTFT Mini Kit 320x240                $34.95    1    $34.95
... """
>>> SKU = slice(0, 6)
>>> DESCRIPTION = slice(6, 40)
>>> UNIT_PRICE = slice(40, 52)
>>> QUANTITY =  slice(52, 55)
>>> ITEM_TOTAL = slice(55, None)
>>> line_items = invoice.split('\n')[2:]
>>> for item in line_items:
...     print(item[UNIT_PRICE], item[DESCRIPTION])
...
    $17.50   Pimoroni PiBrella
     $4.95   6mm Tactile Switch x20
    $28.00   Panavise Jr. - PV-201
    $34.95   PiTFT Mini Kit 320x240
```

When we discuss creating your own collections in "Vector take #2: a sliceable sequence" on page 282, we'll come back to slice objects. Meanwhile, from a user perspective slicing includes additional features such as multi-dimensional slices and ellipsis (...) notation. Read on.

Multi-dimensional slicing and ellipsis

The [] operator can also take multiple indexes or slices separated by commas. This is used, for instance, in the external NumPy package, where items of a 2-dimensional numpy.ndarray can be fetched using the syntax a[i, j] and a 2-dimensional slice obtained with an expression like a[m:n, k:l]. Example 2-22 later in this chapter shows the use of this notation. The __getitem__ and __setitem__ special methods which handle the [] operator simply receive the indices in a[i, j] as a tuple. In other words, to evaluate a[i, j], Python calls a.__getitem__((i, j)).

The built-in sequence types in Python are one-dimensional, so they support only one index or slice, and not a tuple of them.

The ellipsis — written with three full stops ... and not … (Unicode U+2026) — is recognized as a token by the Python parser. It is an alias to the Ellipsis object, the single instance of the ellipsis class[4]. As such, it can be passed as an argument to functions and as part of a slice specification, as in f(a, ..., z) or a[i:...]. NumPy uses ... as a shortcut when slicing arrays of many dimensions, for example, if x is a 4-dimensional array, x[i, ...] is a shortcut for x[i, :, :, :,]. See the Tentative NumPy Tutorial to learn more about this.

I am unaware of uses of Ellipsis or multi-dimensional indexes and slices in the Python standard library at this writing. If you spot one, please let me know. These syntactic features exist to support user-defined types and extensions such as NumPy.

[4] No, I did not get this backwards: the ellipsis class name is really all lowercase and the instance is a built-in named Ellipsis, just like bool is lowercase but its instances are True and False.

Slices are not just useful to extract information from sequences, they can also be used to change mutable sequences in-place, that is, without rebuilding them from scratch.

Assigning to slices

Mutable sequences can be grafted, excised and otherwise modified in-place using slice notation on the left side of an assignment statement or as the target of a `del` statement. The next few examples give an idea of the power of this notation.

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2:5] = [20, 30]
>>> l
[0, 1, 20, 30, 5, 6, 7, 8, 9]
>>> del l[5:7]
>>> l
[0, 1, 20, 30, 5, 8, 9]
>>> l[3::2] = [11, 22]
>>> l
[0, 1, 20, 11, 5, 22, 9]
>>> l[2:5] = 100  ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable
>>> l[2:5] = [100]
>>> l
[0, 1, 100, 22, 9]
```

❶ When the target of the assignment is a slice, the right-hand side must be an iterable object, even if it has just one item.

Everybody knows that concatenation is a common operation with sequences of any type. Any introductory Python text explains the use of `+` and `*` for that purpose, but there are some subtle details on how they work, which we cover next.

Using + and * with sequences

Python programmers expect that sequences support `+` and `*`. Usually both operands of `+` must be of the same sequence type, and neither of them is modified but a new sequence of the same type is created as result of the concatenation.

To concatenate multiple copies of the same sequence, multiply it by an integer. Again, a new sequence is created:

```
>>> l = [1, 2, 3]
>>> l * 5
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> 5 * 'abcd'
'abcdabcdabcdabcdabcd'
```

Both + and * always create a new object, and never change their operands.

Beware of expressions like `a * n` when `a` is a sequence containing mutable items because the result may surprise you. For example, trying to initialize a list of lists as `my_list = [[]] * 3` will result in a list with three references to the same inner list, which is probably not what you want.

The next section covers the pitfalls of trying to use * to initialize a list of lists.

Building lists of lists

Sometimes we need to initialize a list with a certain number of nested lists, for example, to distribute students in a list of teams or to represent squares on a game board. The best way of doing so is with a list comprehension, like this:

Example 2-12. A list with 3 lists of length 3 can represent a Tic-tac-toe board.

```
>>> board = [['_'] * 3 for i in range(3)]  ❶
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[1][2] = 'X'  ❷
>>> board
[['_', '_', '_'], ['_', '_', 'X'], ['_', '_', '_']]
```

❶ Create a list of with 3 lists of 3 items each. Inspect the structure.

❷ Place a mark in row 1, column 2 and check the result.

A tempting but wrong shortcut is doing it like Example 2-13

Example 2-13. A list with with three references to the same list is useless.

```
>>> weird_board = [['_'] * 3] * 3  ❶
>>> weird_board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> weird_board[1][2] = 'O' ❷
>>> weird_board
[['_', '_', 'O'], ['_', '_', 'O'], ['_', '_', 'O']]
```

❶ The outer list is made of three references to the same inner list. While it is unchanged, all seems right.

❷ Placing a mark in row 1, column 2 reveals that all rows are aliases referring to the same object.

The problem with Example 2-13 is that, in essence, it behaves like this code:

```
row = ['_'] * 3
board = []
for i in range(3):
    board.append(row)  ❶
```

❶ The same `row` is appended 3 times to `board`.

On the other hand, the list comprehension from Example 2-12 is equivalent to this code:

```
>>> board = []
>>> for i in range(3):
...     row = ['_'] * 3  # ❶
...     board.append(row)
...
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[2][0] = 'X'
>>> board  # ❷
[['_', '_', '_'], ['_', '_', '_'], ['X', '_', '_']]
```

❶ Each iteration builds a new `row` and appends it to `board`.

❷ Only row `2` is changed, as expected.

If either the problem or the solution in this section are not clear to you, relax. The entire Chapter 8 was written to clarify the mechanics and pitfalls of references and mutable objects.

So far we have discussed the use of the plain + and * operators with sequences, but there are also the += and *= operators, which produce very different results depending on the mutability of the target sequence. The following section explains how that works.

Augmented assignment with sequences

The augmented assignment operators += and *= behave very differently depending on the first operand. To simplify the discussion, we will focus on augmented addition first (+=), but the concepts also apply to *= and to other augmented assignment operators.

The special method that makes += work is `__iadd__` (for "in-place addition"). However, if `__iadd__` is not implemented, Python falls back to calling `__add__`. Consider this simple expression:

```
>>> a += b
```

If a implements __iadd__, that will be called. In the case of mutable sequences like list, bytearray, array.array, a will be changed in-place, i.e. the effect will be similar to a.extend(b). However, when a does not implement __iadd__, the expression a += b has the same effect as a = a + b: the expression a + b is evaluated first, producing a new object which is then bound to a. In other words, the identity of the object bound to a may or may not change, depending on the availability of __iadd__.

In general, for mutable sequences it is a good bet that __iadd__ is implemented and that += happens in-place. For immutable sequences, clearly there is no way for that to happen.

What I just wrote about += also applies to *=, which is implemented via __imul__. The __iadd__ and __imul__ special methods are discussed in Chapter 13.

Here is a demonstration of *= with a mutable sequence and then an immutable one:

```
>>> l = [1, 2, 3]
>>> id(l)
4311953800  ❶
>>> l *= 2
>>> l
[1, 2, 3, 1, 2, 3]
>>> id(l)
4311953800  ❷
>>> t = (1, 2, 3)
>>> id(t)
4312681568  ❸
>>> t *= 2
>>> id(t)
4301348296  ❹
```

❶ id of the initial list;

❷ after multiplication, the list is the same object, with new items appended;

❸ id of the initial tuple;

❹ after multiplication, a new tuple was created.

Repeated concatenation of immutable sequences is inefficient, because instead of just appending new items, the interpreter has to copy the whole target sequence to create a new one with the new items concatenated[5].

We've seen common use cases for +=. The next section shows an intriguing corner case that highlights what "immutable" really means in the context of tuples.

[5] str is an exception to this description. Because string building with += in loops is so common in the wild, CPython is optimized for this use case. str instances are allocated in memory with room to spare, so that concatenation does not require copying the whole string every time.

A += assignment puzzler

Try to answer without using the console: what is the result of evaluating these two expressions[6]?

Example 2-14. A riddle.

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
```

What happens next? Choose the best answer:

a) `t` becomes `(1, 2, [30, 40, 50, 60])`.

b) `TypeError` is raised with the message `'tuple' object does not support item assignment`.

c) Neither.

d) Both **a** and **b**.

When I saw this I was pretty sure the answer was **b** but it's actually **d**, "Both **a** and **b**."! Here is the actual output from a Python 3.4 console (actually the result is the same in a Python 2.7 console):

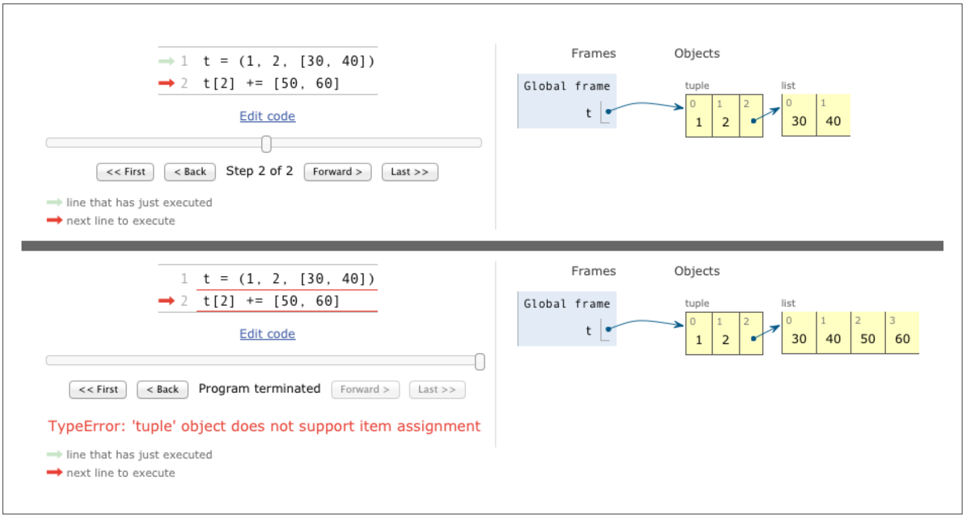
Example 2-15. The unexpected result: item `t2` is changed and an exception is raised[7].

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, [30, 40, 50, 60])
```

Online Python Tutor is an awesome online tool to visualize how Python works in detail. Figure 2-3 is a composite of two screenshots showing the initial and final states of the tuple `t` from Example 2-15.

[6] Thanks to Leonardo Rochael and Cesar Kawakami for sharing this riddle at the 2013 PythonBrasil Conference.

[7] *A reader suggested that the operation in the example can be performed with `t[2].extend([50,60])`, without errors. We're aware of that, but the intent of the example is to discuss the odd behavior of the += operator.*



```
t = (1, 2, [30, 40])
t[2] += [50, 60]
```

TypeError: 'tuple' object does not support item assignment

Figure 2-3. Initial and final state of the tuple assignment puzzler. (Diagram generated by Online Python Tutor)

If you look at the bytecode Python generates for the expression `s[a] += b` (Example 2-16), it becomes clear how that happens.

Example 2-16. Bytecode for the expression `s[a] += b`.

```
>>> dis.dis('s[a] += b')
  1           0 LOAD_NAME                0 (s)
              3 LOAD_NAME                1 (a)
              6 DUP_TOP_TWO
              7 BINARY_SUBSCR                      ❶
              8 LOAD_NAME                2 (b)
             11 INPLACE_ADD                        ❷
             12 ROT_THREE
             13 STORE_SUBSCR                       ❸
             14 LOAD_CONST               0 (None)
             17 RETURN_VALUE
```

❶ Put the value of `s[a]` on `TOS` (Top Of Stack).

❷ Perform `TOS += b`. This succeeds if `TOS` refers to a mutable object (it's a list, in Example 2-15).

❸ Assign `s[a] = TOS`. This fails if `s` is immutable (the `t` tuple in Example 2-15).

This example is quite a corner case — in 15 years of using Python I have never seen this strange behavior actually bite somebody.

I take three lessons from this:

- Putting mutable items in tuples is not a good idea.
- Augmented assignment is not an atomic operation — we just saw it throwing an exception after doing part of its job.
- Inspecting Python bytecode is not too difficult, and is often helpful to see what is going on under the hood.

After witnessing the subtleties of using `+` and `*` for concatenation, we can change the subject to another essential operation with sequences: sorting.

`list.sort` and the `sorted` built-in function

The `list.sort` method sorts a list in-place, that is, without making a copy. It returns `None` to remind us that it changes the target object, and does not create a new list. This is an important Python API convention: functions or methods that change an object in-place should return `None` to make it clear to the caller that the object itself was changed, and no new object was created. The same behavior can be seen, for example, in the `random.shuffle` function.

> The convention of returning `None` to signal in-place changes has a drawback: you cannot cascade calls to those methods. In contrast, methods that return new objects (eg. all `str` methods) can be cascaded in the fluent interface style. See Fluent interface on Wikipedia for a description of fluent interfaces.

In contrast, the built-in function `sorted` creates a new list and returns it. In fact, `sorted` accepts any iterable object as argument, including immutable sequences and generators (see Chapter 14). Regardless of the type of iterable given to `sorted`, it always returns a newly created list.

Both `list.sort` and `sorted` take two optional, keyword-only arguments: `key` and `reverse`.

`reverse`

  If `True`, the items are returned in descending order, i.e. by reversing the comparison of the items. The default is `False`.

`key`

  A one-argument function that will be applied to each item to produce its sorting key. For example, when sorting a list of strings, `key=str.lower` can be used to perform a case-insensitive sort, and `key=len` will sort the strings by character length. The default is the identity function, i.e. the items themselves are compared.

The `key` optional keyword parameter can also be used with the `min()` and `max()` built-ins and with other functions from the standard library such as `itertools.groupby()` and `heapq.nlargest()`.

Here are a few examples to clarify the use of these functions and keyword arguments. The examples also demonstrate that Timsort — the sorting algorithm used in Python — is stable, i.e. it preserves the relative ordering of items that compare equal[8].

```
>>> fruits = ['grape', 'raspberry', 'apple', 'banana']
>>> sorted(fruits)
['apple', 'banana', 'grape', 'raspberry']  ❶
>>> fruits
['grape', 'raspberry', 'apple', 'banana']  ❷
>>> sorted(fruits, reverse=True)
['raspberry', 'grape', 'banana', 'apple']  ❸
>>> sorted(fruits, key=len)
['grape', 'apple', 'banana', 'raspberry']  ❹
>>> sorted(fruits, key=len, reverse=True)
['raspberry', 'banana', 'grape', 'apple']  ❺
>>> fruits
['grape', 'raspberry', 'apple', 'banana']  ❻
>>> fruits.sort()                          ❼
>>> fruits
['apple', 'banana', 'grape', 'raspberry']  ❽
```

❶ This produces a new list of strings sorted alphabetically.

❷ Inspecting the original list, we see it is unchanged.

❸ This is simply reverse alphabetical ordering.

❹ A new list of strings, now sorted by length. Since the sorting algorithm is stable, “grape” and “apple”, both of length 5, are in the original order.

❺ These are the strings sorted in descending order of length. It is not the reverse of the previous result because the sorting is stable, so again “grape” appears before “apple”.

❻ So far the ordering of the original `fruits` list has not changed.

❼ This sorts the list in-place, and returns `None` (which the console omits).

❽ Now `fruits` is sorted.

Once your sequences are sorted, they can be very efficiently searched. Fortunately, the standard binary search algorithm is already provided in `bisect` module of the Python

[8] The **Soapbox** at the end of this chapter has more about Timsort.

Standard Library. We discuss its essential features next, including the convenient bisect.insort function which you can use to make sure that your sorted sequences stay sorted.

Managing ordered sequences with bisect

The bisect module offers two main functions — bisect and insort — that use the binary search algorithm to quickly find and insert items in any sorted sequence.

Searching with bisect

bisect(haystack, needle) does a binary search for needle in haystack — which must be a sorted sequence — to locate the position where needle can be inserted while maintaining haystack in ascending order. In other words, all items appearing up to that position are less or equal to needle. You could use the result of bisect(haystack, needle) as the index argument to haystack.insert(index, needle), but using insort does both steps, and is faster.

Raymond Hettinger — a prolific Python contributor — has a Sorted Collection recipe which leverages the bisect module but is easier to use than these stand-alone functions.

Example 2-17 uses a carefully chosen set of "needles" to demonstrate the insert positions returned by bisect. Its output is in Figure 2-4.

Example 2-17. bisect finds insertion points for items in a sorted sequence.

```
import bisect
import sys

HAYSTACK = [1, 4, 5, 6, 8, 12, 15, 20, 21, 23, 23, 26, 29, 30]
NEEDLES = [0, 1, 2, 5, 8, 10, 22, 23, 29, 30, 31]

ROW_FMT = '{0:2d} @ {1:2d}    {2}{0:<2d}'

def demo(bisect_fn):
    for needle in reversed(NEEDLES):
        position = bisect_fn(HAYSTACK, needle)  ❶
        offset = position * '  |'  ❷
        print(ROW_FMT.format(needle, position, offset))  ❸

if __name__ == '__main__':

    if sys.argv[-1] == 'left':    ❹
```

```
        bisect_fn = bisect.bisect_left
    else:
        bisect_fn = bisect.bisect

    print('DEMO:', bisect_fn.__name__)  ❺
    print('haystack ->', ' '.join('%2d' % n for n in HAYSTACK))
    demo(bisect_fn)
```

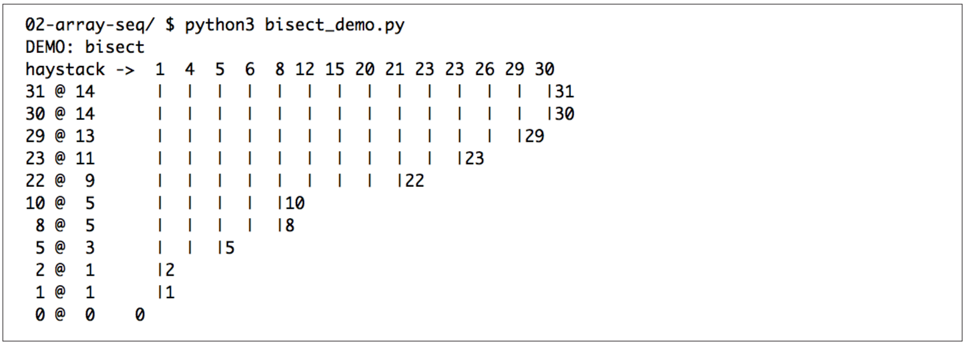
❶ Use the chosen `bisect` function to get the insertion point.

❷ Build a pattern of vertical bars proportional to the `offset`.

❸ Print formatted row showing needle and insertion point.

❹ Choose the `bisect` function to use according to the last command line argument.

❺ Print header with name of function selected.



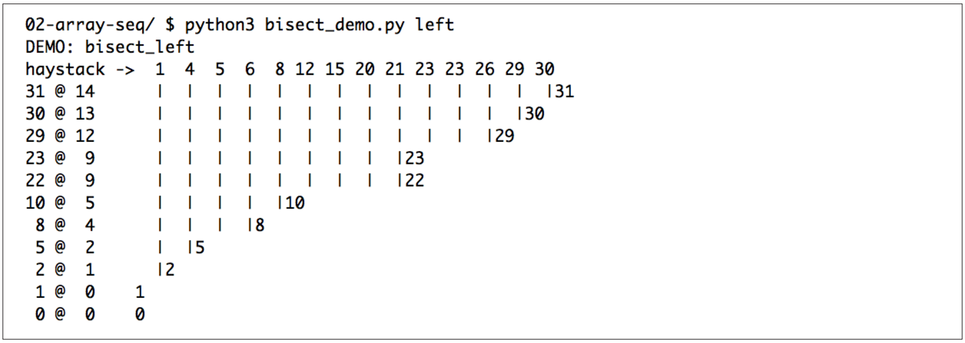
```
02-array-seq/ $ python3 bisect_demo.py
DEMO: bisect
haystack ->  1  4  5  6  8 12 15 20 21 23 23 26 29 30
31 @ 14      |  |  |  |  |  |  |  |  |  |  |  |  |  |31
30 @ 14      |  |  |  |  |  |  |  |  |  |  |  |  |  |30
29 @ 13      |  |  |  |  |  |  |  |  |  |  |  |  |29
23 @ 11      |  |  |  |  |  |  |  |  |  |  |23
22 @  9      |  |  |  |  |  |  |  |  |22
10 @  5      |  |  |  |  |  |10
 8 @  5      |  |  |  |  |  |8
 5 @  3      |  |  |  |5
 2 @  1      |  |2
 1 @  1      |  |1
 0 @  0    0
```

Figure 2-4. Output of Example 2-17 with `bisect` in use. Each row starts with the notation `needle @ position` and the `needle` value appears again below its insertion point in the `haystack`.

The behavior of `bisect` can be fine-tuned in two ways.

First, a pair of optional arguments `lo` and `hi` allow narrowing the region in the sequence to be searched when inserting. `lo` defaults to 0 and `hi` to the `len()` of the sequence.

Second, `bisect` is actually an alias for `bisect_right`, and there is a sister function called `bisect_left`. Their difference is apparent only when the needle compares equal to an item in the list: `bisect_right` returns an insertion point after the existing item, and `bisect_left` returns the position of the existing item, so insertion would occur before it. With simple types like `int` this makes no difference, but if the sequence contains objects that are distinct yet compare equal, then it may be relevant. For example, `1` and `1.0` are distinct, but `1 == 1.0` is `True`.



```
02-array-seq/ $ python3 bisect_demo.py left
DEMO: bisect_left
haystack ->  1  4  5  6  8 12 15 20 21 23 23 26 29 30
31 @ 14      |  |  |  |  |  |  |  |  |  |  |  |  |  |31
30 @ 13      |  |  |  |  |  |  |  |  |  |  |  |  |30
29 @ 12      |  |  |  |  |  |  |  |  |  |  |  |29
23 @  9      |  |  |  |  |  |  |  |  |23
22 @  9      |  |  |  |  |  |  |  |  |22
10 @  5      |  |  |  |  |10
 8 @  4      |  |  |  |8
 5 @  2      |  |5
 2 @  1      |2
 1 @  0    1
 0 @  0    0
```

Figure 2-5. Output of Example 2-17 with `bisect_left` in use. Compare with Figure 2-4 and note the insertion points for the values 1, 8, 23, 29 and 30 to the left of the same numbers in the `haystack`.

An interesting application of `bisect` is to perform table lookups by numeric values, for example to convert test scores to letter grades, as in Example 2-18.

Example 2-18. Given a test score, `grade` returns the corresponding letter grade.

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect.bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

The code in Example 2-18 is from the `bisect` module documentation, which also lists functions to use `bisect` as a faster replacement for the `index` method when searching through long ordered sequences of numbers.

These functions are not only used for searching, but also for inserting items in sorted sequences, as the following section shows.

Inserting with `bisect.insort`

Sorting is expensive, so once you have a sorted sequence, it's good to keep it that way. That is why `bisect.insort` was created.

`insort(seq, item)` inserts `item` into `seq` so as to keep `seq` in ascending order. See Example 2-19 and its output in Figure 2-6.

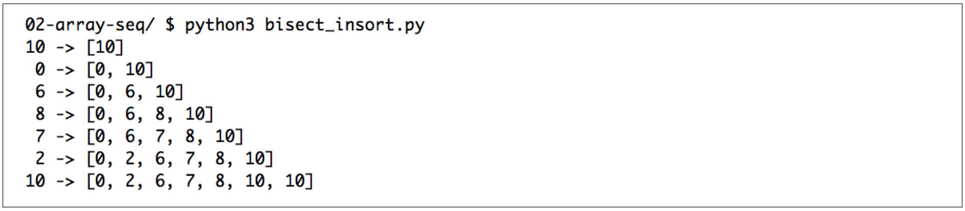
Example 2-19. Insort keeps a sorted sequence always sorted.

```
import bisect
import random

SIZE = 7
```

```
random.seed(1729)

my_list = []
for i in range(SIZE):
    new_item = random.randrange(SIZE*2)
    bisect.insort(my_list, new_item)
    print('%2d ->' % new_item, my_list)
```



```
02-array-seq/ $ python3 bisect_insort.py
10 -> [10]
 0 -> [0, 10]
 6 -> [0, 6, 10]
 8 -> [0, 6, 8, 10]
 7 -> [0, 6, 7, 8, 10]
 2 -> [0, 2, 6, 7, 8, 10]
10 -> [0, 2, 6, 7, 8, 10, 10]
```

Figure 2-6. Output of Example 2-19.

Like bisect, insort takes optional lo, hi arguments to limit the search to a subsequence. There is also an insort_left variation that uses bisect_left to find insertion points.

Much of what we have seen so far in this chapter applies to sequences in general, not just lists or tuples. Python programmers sometimes overuse the list type because it is so handy — I know I've done it. If you are handling lists of numbers, arrays are the way to go. The remainder of the chapter is devoted to them.

When a list is not the answer

The list type is flexible and easy to use, but depending on specific requirements there are better options. For example, if you need to store 10 million of floating point values an array is much more efficient, because an array does not actually hold full-fledged float objects, but only the packed bytes representing their machine values — just like an array in the C language. On the other hand, if you are constantly adding and removing items from the ends of a list as a FIFO or LIFO data structure, a deque (double-ended queue) works faster.

> If your code does a lot of containment checks — e.g. item in my_collection, consider using a set for my_collection, especially if it holds a large number of items. Sets are optimized for fast membership checking. But they are not sequences (their content is unordered). We cover them in Chapter 3.

For the remainder of this chapter we discuss mutable sequence types that can replace lists in many cases, starting with arrays.

Arrays

If all you want to put in the list are numbers, an `array.array` is more efficient than a `list`: it supports all mutable sequence operations (including `.pop`, `.insert` and `.extend`), and additional methods for fast loading and saving such as `.frombytes` and `.tofile`.

A Python array is as lean as a C array. When creating an `array` you provide a typecode, a letter to determine the underlying C type used to store each item in the array. For example, *b* is the typecode for `signed char`. If you create an `array('b')` then each item will be stored in a single byte and interpreted as an integer from -128 to 127. For large sequences of numbers, this saves a lot of memory. And Python will not let you put any number that does not match the type for the array.

Example 2-20 shows creating, saving and loading an array of 10 million floating-point random numbers:

Example 2-20. Creating, saving and loading a large array of floats.

```
>>> from array import array  ❶
>>> from random import random
>>> floats = array('d', (random() for i in range(10**7)))  ❷
>>> floats[-1]  ❸
0.07802343889111107
>>> fp = open('floats.bin', 'wb')
>>> floats.tofile(fp)  ❹
>>> fp.close()
>>> floats2 = array('d')  ❺
>>> fp = open('floats.bin', 'rb')
>>> floats2.fromfile(fp, 10**7)  ❻
>>> fp.close()
>>> floats2[-1]  ❼
0.07802343889111107
>>> floats2 == floats  ❽
True
```

❶ import the `array` type;

❷ create an array of double-precision floats (typecode `'d'`) from any iterable object, in this case a generator expression;

❸ inspect the last number in the array;

❹ save the array to a binary file;

❺ create an empty array of doubles;

❻ read 10 million numbers from the binary file;

❼ inspect the last number in the array;

❽ verify that the contents of the arrays match;

As you can see, `array.tofile` and `array.fromfile` are easy to use. If you try the example, you'll notice they are also very fast. A quick experiment show that it takes about 0.1s for `array.fromfile` to load 10 million double-precision floats from a binary file created with `array.tofile`. That is nearly 60 times faster than reading the numbers from a text file, which also involves parsing each line with the `float` built-in. Saving with `array.tofile` is about 7 times faster than writing one float per line in a text file. In addition, the size of the binary file with 10 million doubles is 80,000,000 bytes (8 bytes per double, zero overhead), while the text file has 181,515,739 bytes, for the same data.

> Another fast and more flexible way of saving numeric data is the `pickle` module for object serialization. Saving an array of floats with `pickle.dump` is almost as fast as with `array.tofile`, but `pickle` handles almost all built-in types, including `complex` numbers, nested collections and even instances of user defined classes automatically — if they are not too tricky in their implementation.

For the specific case of numeric arrays representing binary data, such as raster images, Python has the `bytes` and `bytearray` types discussed in Chapter 4.

We wrap-up this section on arrays with Table 2-2, comparing the features of `list` and `array.array`.

Table 2-2. Methods and attributes found in `list` or `array` (deprecated `array` methods and those also implemented by `object` were omitted for brevity).

	list	array	
`s.__add__(s2)`	●	●	`s + s2` — concatenation
`s.__iadd__(s2)`	●	●	`s += s2` — in-place concatenation
`s.append(e)`	●	●	append one element after last
`s.byteswap()`		●	swap bytes of all items in array for endianess conversion
`s.clear()`	●		delete all items
`s.__contains__(e)`	●	●	`e in s`
`s.copy()`	●		shallow copy of the list
`s.__copy__()`		●	support for `copy.copy`
`s.count(e)`	●	●	count occurrences of an element
`s.__deepcopy__()`		●	optimized support for `copy.deepcopy`
`s.__delitem__(p)`	●	●	remove item at position p

	list	array	
s.extend(it)	●	●	append items from iterable it
s.frombytes(b)		●	append items from byte sequence interpreted as packed machine values
s.fromfile(f, n)		●	append n items from binary file f interpreted as packed machine values
s.fromlist(l)		●	append items from list; if one causes TypeError, none are appended
s.__getitem__(p)	●	●	s[p] — get item at position
s.index(e)	●	●	find position of first occurrence of e
s.insert(p, e)	●	●	insert element e before the item at position p
s.itemsize		●	length in bytes of each array item
s.__iter__()	●	●	get iterator
s.__len__()	●	●	len(s) — number of items
s.__mul__(n)	●	●	s * n — repeated concatenation
s.__imul__(n)	●	●	s *= n — in-place repeated concatenation
s.__rmul__(n)	●	●	n * s — reversed repeated concatenation[a]
s.pop(«p»)	●	●	remove and return item at position p (default: last)
s.remove(e)	●	●	remove first occurrence of element e by value
s.reverse()	●	●	reverse the order of the items in-place
s.__reversed__()	●		get iterator to scan items from last to first
s.__setitem__(p, e)	●	●	s[p] = e — put e in position p, overwriting existing item
s.sort(«key», «reverse»)	●		sort items in place with optional keyword arguments key and reverse
s.tobytes()		●	return items as packed machine values in a bytes object
s.tofile(f)		●	save items as packed machine values to binary file f
s.tolist()		●	return items as numeric objects in a list
s.typecode		●	one-character string identifying the C type of the items

[a] Reversed operators are explained in Chapter 13.

As of Python 3.4, the array type does not have an in-place sort method like list.sort(). If you need to sort an array, use the sorted function to rebuild it sorted:

```
a = array.array(a.typecode, sorted(a))
```

To keep a sorted array sorted while adding items to it, use the bisect.insort function as seen in "Inserting with bisect.insort" on page 46.

If you do a lot of work with arrays and don't know about memoryview, you're missing out. See the next topic.

Memory views

The built-in memorview class is a shared-memory sequence type that lets you handle slices of arrays without copying bytes. It was inspired by the NumPy library (which we'll visit shortly in "NumPy and SciPy" on page 52). Travis Oliphant, lead author of NumPy, answers When should a memoryview be used? like this:

> A memoryview is essentially a generalized NumPy array structure in Python itself (without the math). It allows you to share memory between data-structures (things like PIL images, SQLlite databases, NumPy arrays, etc.) without first copying. This is very important for large data sets.

Using notation similar to the array module, the memoryview.cast method lets you change the way multiple bytes are read or written as units without moving bits around — just like the C cast operator. memoryview.cast returns yet another memoryview object, always sharing the same memory.

See Example 2-21 for an example of changing a single byte of an array of 16-bit integers.

Example 2-21. Changing the value of an array item by poking one of its bytes.

```
>>> numbers = array.array('h', [-2, -1, 0, 1, 2])
>>> memv = memoryview(numbers)  ❶
>>> len(memv)
5
>>> memv[0]  ❷
-2
>>> memv_oct = memv.cast('B')  ❸
>>> memv_oct.tolist()  ❹
[254, 255, 255, 255, 0, 0, 1, 0, 2, 0]
>>> memv_oct[5] = 4  ❺
>>> numbers
array('h', [-2, -1, 1024, 1, 2])  ❻
```

❶ Build memoryview from array of 5 short signed integers (typecode 'h').

❷ memv sees the same 5 items in the array.

❸ Create memv_oct by casting the elements of memv to typecode 'B' (unsigned char).

❹ Export elements of memv_oct as a list, for inspection.

❺ Assign value 4 to byte offset 5.

❻ Note change to numbers: a 4 in the most significant byte of a 2-byte unsigned integer is 1024.

We'll see another short example with `memoryview` in the context of binary sequence manipulations with `struct` (Chapter 4, Example 4-4).

Meanwhile, if you are doing advanced numeric processing in arrays, then you should be using the NumPy and SciPy libraries. We'll take a brief look at them right away.

NumPy and SciPy

Throughout this book I make a point of highlighting what is already in the Python Standard Library so you can make the most of it. But NumPy and SciPy are so awesome that a detour is warranted.

For advanced array and matrix operations, NumPy and SciPy are the reason why Python became mainstream in scientific computing applications[9]. NumPy implements multi-dimensional, homogeneous arrays and matrix types that hold not only numbers but also user defined records, and provides efficient elementwise operations.

SciPy is a library, written on top of NumPy, offering many scientific computing algorithms from linear algebra, numerical calculus and statistics. SciPy is fast and reliable because it leverages the widely-used C and Fortran codebase from the Netlib Repository. In other words, SciPy gives scientists the best of both worlds: an interactive prompt and high-level Python APIs, together with industrial-strength number crunching functions optimized in C and Fortran.

As a very brief demo, Example 2-22 shows some basic operations with 2-dimensional arrays in NumPy.

Example 2-22. Basic operations with rows and columns in a `numpy.ndarray`.

```
>>> import numpy  ❶
>>> a = numpy.arange(12)  ❷
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> type(a)
<class 'numpy.ndarray'>
>>> a.shape  ❸
(12,)
>>> a.shape = 3, 4  ❹
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a[2]  ❺
array([ 8,  9, 10, 11])
```

[9] Of course, if Python did not have an expressive and user-friendly syntax and semantics, and was not easy to extend in C, then the creators of NumPy and SciPy may not have invested in it. So it is fair to say these libraries are part of the reason why Python became successful among scientists in many fields; the rest of the credit goes to Guido van Rossum and the Python core team.

```
>>> a[2, 1]  ❻
9
>>> a[:, 1]  ❼
array([1, 5, 9])
>>> a.transpose()  ❽
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

❶ import Numpy, after installing (it's not in the Python Standard Library);

❷ build and inspect a `numpy.ndarray` with integers 0 to 11

❸ inspect the dimensions of the array: this is a one-dimensional, 12-element array;

❹ change the shape of the array, adding one dimension, then inspecting the result;

❺ get row at index 2;

❻ get element at index `2, 1`;

❼ get column at index `1`;

❽ create a new array by transposing (swapping columns with rows).

NumPy also supports high-level operations for loading, saving and operating on all elements of a `numpy.ndarray`.

```
>>> import numpy
>>> floats = numpy.loadtxt('floats-10M-lines.txt')  ❶
>>> floats[-3:]  ❷
array([ 3016362.69195522,   535281.10514262,  4566560.44373946])
>>> floats *= .5  ❸
>>> floats[-3:]
array([ 1508181.34597761,   267640.55257131,  2283280.22186973])
>>> from time import perf_counter as pc ❹
>>> t0 = pc(); floats /= 3; pc() - t0 ❺
0.03690556302899495
>>> numpy.save('floats-10M', floats)  ❻
>>> floats2 = numpy.load('floats-10M.npy', 'r+')  ❼
>>> floats2 *= 6
>>> floats2[-3:]  ❽
memmap([ 3016362.69195522,   535281.10514262,  4566560.44373946])
```

❶ load 10 million floating point numbers from a text file;

❷ use sequence slicing notation to inspect the last 3 numbers;

❸ multiply every element in the `floats` array by `.5` and inspect last 3 elements again;

❹ import the high-resolution performance measurement timer (available since Python 3.3);

❺ divide every element by 3; the elapsed time for 10 million floats is less than 40 milliseconds;

❻ save the array in a .npy binary file;

❼ load the data as a memory-mapped file into another array; this allows efficient processing of slices of the array even if it does not fit entirely in memory;

❽ inspect the last 3 elements after multiplying every element by 6.

Installing NumPy and SciPy from source is not a breeze. The Installing the SciPy Stack page on SciPy.org recommends using special scientific Python distributions such as Anaconda, Enthought Canopy and WinPython, among others. These are large downloads, but come ready to use. Users of popular GNU/Linux distributions can usually find NumPy and SciPy in the standard package repositories. For example, installing them on Debian or Ubuntu is as easy as:

```
$ sudo apt-get install python-numpy python-scipy
```

This was just an appetizer. NumPy and SciPy are formidable libraries, and are the foundation of other awesome tools such as the Pandas and Blaze data analysis libraries, which provide efficient array types that can to hold non-numeric data as well as import/export functions compatible with many different formats like .csv, .xls, SQL dumps, HDF5 etc. These packages deserve entire books about them. This is not one of those books. But no overview of Python sequences would be complete without at least a quick look at NumPy arrays.

Having looked at flat sequences — standard arrays and NumPy arrays — we now turn to a completely different set of replacements for the plain old list: queues.

Deques and other queues

The .append and .pop methods make a list usable as a stack or a queue (if you use .append and .pop(0), you get LIFO behavior). But inserting and removing from the left of a list (the 0-index end) is costly because the entire list must be shifted.

The class collections.deque is a thread-safe double-ended queue designed for fast inserting and removing from both ends. It is also the way to go if you need to keep a list of "last seen items" or something like that, because a deque can be bounded — i.e. created with a maximum length and then, when it is full, it discards items from the opposite end when you append new ones.

```
>>> from collections import deque
>>> dq = deque(range(10), maxlen=10)  ❶
>>> dq
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
```

```
>>> dq.rotate(3)  ❷
>>> dq
deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)
>>> dq.rotate(-4)
>>> dq
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], maxlen=10)
>>> dq.appendleft(-1)  ❸
>>> dq
deque([-1, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.extend([11, 22, 33])  ❹
>>> dq
deque([3, 4, 5, 6, 7, 8, 9, 11, 22, 33], maxlen=10)
>>> dq.extendleft([10, 20, 30, 40])  ❺
>>> dq
deque([40, 30, 20, 10, 3, 4, 5, 6, 7, 8], maxlen=10)
```

❶ the optional `maxlen` argument set the maximum number of items allowed in this instance of `deque`; this sets a read-only `maxlen` instance attribute;

❷ rotating with `n > 0` takes items from the right end and prepends them to the left; when `n < 0` items are taken from left and appended to the right;

❸ appending to a `deque` that is full (`len(d) == d.maxlen`) discards items from the other end; note in the next line that the `0` is dropped;

❹ adding three items to the right pushes out the leftmost `-1`, `1` and `2`;

❺ note that `extendleft(iter)` works by appending each successive item of the `iter` argument to the left of the deque, therefore the final position of the items is reversed.

Table 2-3 compares the methods that are specific to `list` and `deque` (removing those that also appear in `object`).

Note that `deque` implements most of the `list` methods, and adds a few specific to its design, like `popleft` and `rotate`. But there is a hidden cost: removing items from the middle of a `deque` is not as fast. It is really optimized for appending and popping from the ends.

The `append` and `popleft` operations are atomic, so `deque` is safe to use as a LIFO-queue in multi-threaded applications without the need for using locks.

Table 2-3. Methods implemented in `list` or `deque` (those that are also implemented by `object` were omitted for brevity).

	list	deque	
`s.__add__(s2)`	●		`s + s2` concatenation
`s.__iadd__(s2)`	●	●	`s += s2` in-place concatenation
`s.append(e)`	●	●	append one element to the right (after last)

	list	deque	
s.appendleft(e)		●	append one element to the left (before first)
s.clear()	●	●	delete all items
s.__contains__(e)	●		e in s
s.copy()	●		shallow copy of the list
s.__copy__()		●	support for copy.copy (shallow copy)
s.count(e)	●	●	count occurrences of an element
s.__delitem__(p)	●	●	remove item at position p
s.extend(i)	●	●	append items from iterable i to the right
s.extendleft(i)		●	append items from iterable i to the left
s.__getitem__(p)	●	●	s[p] — get item at position
s.index(e)	●		find position of first occurrence of e
s.insert(p, e)	●		insert element e before the item at position p
s.__iter__()	●	●	get iterator
s.__len__()	●	●	len(s) — number of items
s.__mul__(n)	●		s * n — repeated concatenation
s.__imul__(n)	●		s *= n — in-place repeated concatenation
s.__rmul__(n)	●		n * s — reversed repeated concatenation[a]
s.pop()	●	●	remove and return last item[b]
s.popleft()		●	remove and return first item
s.remove(e)	●	●	remove first occurrence of element e by value
s.reverse()	●	●	reverse the order of the items in-place
s.__reversed__()	●	●	get iterator to scan items from last to first
s.rotate(n)		●	move n items from one end to the other
s.__setitem__(p, e)	●	●	s[p] = e — put e in position p, overwriting existing item
s.sort(«key», «reverse»)	●		sort items in place with optional keyword arguments key and reverse

[a] Reversed operators are explained in Chapter 13.

[b] a_list.pop(p) allows removing from position p but deque does not support that option.

Besides deque, other Python Standard Library packages implement queues.

queue

Provides the synchronized (i.e. thread-safe) classes Queue, LifoQueue and PriorityQueue. These are used for safe communication between threads. All three classes can be bounded by providing a maxsize argument greater than 0 to the constructor. However, they don't discard items to make room as deque does. Instead, when the queue is full the insertion of a new item blocks — i.e. it waits until some other thread makes room by taking an item from the queue, which is useful to throttle the number of live threads.

`multiprocessing`
: Implements its own bounded `Queue`, very similar to `queue.Queue` but designed for inter-process communication. There is also has a specialized `multiprocessing.JoinableQueue` for easier task management.

`asyncio`
: Newly added to Python 3.4, `asyncio` provides `Queue`, `LifoQueue`, `PriorityQueue` and `JoinableQueue` with APIs inspired by the classes in `queue` and `multiprocessing`, but adapted for managing tasks in asynchronous programming.

`heapq`
: In contrast to the previous three modules, `heapq` does not implement a queue class, but provides functions like `heappush` and `heappop` that let you use a mutable sequence as a heap queue or priority queue.

This ends our overview of alternatives to the `list` type, and also our exploration of sequence types in general — except for the particulars of `str` and binary sequences which have their own Chapter 4.

Chapter summary

Mastering the standard library sequence types is a pre-requisite for writing concise, effective and idiomatic Python code.

Python sequences are often categorized as mutable or immutable, but it is also useful to consider a different axis: flat sequences and container sequences. The former are more compact, faster and easier to use, but are limited to storing atomic data such as numbers, characters and bytes. Container sequences are more flexible, but may surprise you when they hold mutable objects, so you need to be careful to use them correctly with nested data structures.

List comprehensions and generator expressions are powerful notations to build and initialize sequences. If you are not yet comfortable with them, take the time to master their basic usage. It is not hard, and soon you will be hooked.

Tuples in Python play two roles: as records with unnamed fields and as immutable lists. When a tuple is used as a record, tuple unpacking is the safest, most readable way of getting at the fields. The new `*` syntax makes tuple unpacking even better by making it easier to ignore some fields and to deal with optional fields. Named tuples are not so new, but deserve more attention: like tuples, they have very little overhead per instance, yet provide convenient access to the fields by name and a handy `._asdict()` to export the record as an `OrderedDict`.

Sequence slicing is a favorite Python syntax feature, and it is even more powerful than many realize. Multi-dimensional slicing and ellipsis `...` notation, as used in NumPy, may also be supported by user-defined sequences. Assigning to slices is a very expressive way of editing mutable sequences.

Repeated concatenation as in `seq * n` is convenient and, with care, can be used to initialize lists of lists containing immutable items. Augmented assignment with += and *= behaves differently for mutable and immutable sequences. In the latter case, these operators necessarily build new sequences. But if the target sequence is mutable, it is usually changed in-place — but not always, depending on how the sequence is implemented.

The `sort` method and the `sorted` built-in function are easy to use and flexible, thanks to the `key` optional argument they accept, with a function to calculate the ordering criterion. By the way, `key` can also be used with the `min` and `max` built-in functions. To keep a sorted sequence in order, always insert items into it using `bisect.insort`; to search it efficiently, use `bisect.bisect`.

Beyond lists and tuples, the Python standard library provides `array.array`. Although NumPy and SciPy are not part of the standard library, if you do any kind of numerical processing on large sets of data, studying even a small part of these libraries can take you a long way.

We closed by visiting the versatile and thread-safe `collections.deque`, comparing its API with that of `list` in Table 2-3 and mentioning other queue implementations in the standard library.

Further reading

Chapter 1 — Data Structures — of the Python Cookbook 3e. by David Beazley and Brian K. Jones has many recipes focusing on sequences, including recipe 1.11 — Naming a Slice — where I learned the trick of assigning slices to variables to improve readability, illustrated in our Example 2-11.

The 2nd edition of the Python Cookbook was written for Python 2.4 but much of its code works with Python 3, and a lot of the recipes in chapters 5 and 6 deal with sequences. The Python Cookbook 2e was edited by Alex Martelli, Anna Martelli Ravenscroft, and David Ascher, and it includes contributions by dozens of Pythonistas. The third edition was rewritten from scratch, and focuses more on the semantics of the language — particularly what has changed in Python 3 — while the older volume emphasizes pragmatics, i.e. how to apply the language to real-world problems. Even though some of the 2nd edition solutions are no longer the best approach, I honestly think it is worthwhile to have both editions of the Python Cookbook on hand.

The official Python Sorting HOW TO has several examples of advanced tricks for using sorted and list.sort.

PEP 3132 — Extended Iterable Unpacking is the canonical source to read about the new use of *extra as a target in parallel assignments. If you'd like a glimpse of Python evolving, Missing *-unpacking generalizations is a bug tracker issue proposing even wider use of iterable unpacking notation. PEP 448 — Additional Unpacking Generalizations resulted from the discussions in that issue. As of this writing it seems likely the proposed changes will be merged to Python, perhaps in version 3.5.

Eli Bendersky has a short tutorial on memoriview at Less copies in Python with the buffer protocol and memoryviews.

There are numerous books covering NumPy in the market, even some don't mention "NumPy" in the title. Wes McKinney's "Python for Data Analysis" (O'Reilly, 2012) is one such title.

Scientists love the combination of an interactive prompt with the power of NumPy and SciPy so much that they developed IPython, an incredibly powerful replacement for the Python console which also provides a GUI, integrated inline graph plotting, literate programming support (interleaving text with code), and rendering to PDF. Interactive, multimedia IPython sessions can even be shared over HTTP as IPython notebooks. See screenshots and video at The IPython Notebook page. IPython is so hot that in 2012 its core developers, most of whom are researchers at UC Berkeley, received a $1.15 million dollar grant from the Sloan Foundation for enhancements to be implemented over the 2013-2014 period.

Chapter 8.3. collections — Container datatypes of the Python Standard Library documentation has short examples and practical recipes using deque (and other collections).

The best defense of the Python convention of excluding the last item in ranges and slices was written by Edsger W. Dijkstra himself, in a short memo titled "Why numbering should start at zero". The subject of the memo is mathematical notation, but it's relevant to Python because Prof. Dijkstra explains with rigor and humor why the sequence 2, 3, …, 12 should always be expressed as 2 ≤ i < 13. All other reasonable conventions are refuted, as is the idea of letting each user choose a convention. The title refers to zero-based indexing, but the memo is really about why it is desirable that 'ABCDE'[1:3] means 'BC' and not 'BCD' and why it makes perfect sense to spell 2, 3, …, 12 as range(2, 13). By the way, the memo is a hand-written note, but it's beautiful and totally readable. Somebody should create a Dijkstra font. I'd buy it.

Soapbox

The nature of tuples

In 2012 I presented a poster about the ABC language at PyCon US. Before creating Python, Guido had worked on the ABC interpreter, so he came to see my poster. Among other things, we talked about the ABC *compounds* which are clearly the predecessors of Python tuples. Compounds also support parallel assignment and are used as composite keys in dictionaries (or *tables*, in ABC parlance). However, compounds are not sequences. They are not iterable and you cannot retrieve a field by index, much less slice them. You either handle the compound as whole or extract the individual fields using parallel assignment, that's all.

I told Guido that these limitations make the main purpose of compounds very clear: they are just records without field names. His response: “Making tuples behave as sequences was a hack.”

This illustrates the pragmatic approach that makes Python so much better and more successful than ABC. From a language implementer perspective, making tuples behave as sequences costs little. As a result, tuples may not be as “conceptually pure” as compounds, but we have many more ways of using them. They can even be used as immutable lists, of all things!

It is really useful to have immutable lists in the language, even if their type is not called `frozenlist` but is really `tuple` behaving as sequence.

“Elegance begets simplicity”

The use of the syntax `*extra` to assign multiple items to a parameter started with function definitions a long time ago (I have a book about Python 1.4 from 1996 that covers that). Starting with Python 1.6, the form `*extra` can be used in the context of function calls to unpack an iterable into multiple arguments, a complementary operation. This is elegant, makes intuitive sense, and made the `apply` function redundant (it's now gone). Now with Python 3 the `*extra` notation also works on the left of parallel assignments to grab excess items, enhancing what was already a handy language feature.

With each of these changes the language became more flexible, more consistent and simpler at the same time. “Elegance begets simplicity” is the motto on my favorite PyCon t-shirt from Chicago, 2009. It is decorated with a painting by Bruce Eckel depicting hexagram 22 from the I Ching — ⍰ (bì), “Adorning" — sometimes translated as “Grace” or “Beauty”.

Flat versus container sequences

To highlight the different memory models of the sequence types I used the terms *container sequence* and *flat sequence*. The “container” word is from the Data Model documentation:

> Some objects contain references to other objects; these are called containers.

I used the term “container sequence” to be specific, because there are containers in Python that are not sequences, like `dict` and `set`. Container sequences can be nested because they may contain objects of any type, including their own type.

On the other hand, *flat sequences* are sequence types that cannot be nested because they only hold simple atomic types like integers, floats or characters.

I adopted the term *flat sequence* because I needed something to contrast with "container sequence". I can't cite a reference to support the use of *flat sequence* in this specific context: as the category of Python sequence types that are not containers. In the Wikipedia this usage would be tagged "original research". I prefer to call it "our term", hoping you'll find it useful and adopt it too.

Mixed bag lists

Introductory Python texts emphasize that lists can contain objects of mixed types, but in practice that feature is not very useful: we put items in a list to process them later, which implies that all items should support at least some operation in common, i.e. they should all "quack" whether or not they are genetically 100% ducks. For example, you can't sort a list in Python 3 unless the items in it are comparable[10].

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

Unlike lists, tuples often hold items of different types. That is natural, considering that each item in a tuple is really a field, and each field type is independent on the others.

Key is brilliant

The `key` optional argument of `list.sort`, `sorted`, `max` and `min` is a great idea. Other languages force you to provide a two-argument comparison function like the deprecated `cmp(a, b)` function in Python 2. Using `key` is both simpler and more efficient. It's simpler because you just define a one-argument function that retrieves or calculates whatever criterion you want to use to sort your objects; this is easier than writing a two-argument function to return -1, 0, 1. It is also more efficient because the key function is invoked only once per item, while the two-argument comparison is called every time the sorting algorithm needs to compare two items. Of course, while sorting Python also has to compare the keys, but that comparison is done in optimized C code and not in a Python function that you wrote.

By the way, using `key` actually lets us sort a mixed bag of numbers and number-like strings. You just need to decide whether you want to treat all items as integers or strings:

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l, key=int)
[0, '1', 5, 6, '9', 14, 19, '23', 28, '28']
```

[10] Python 2 lets you sort mixed lists, but you get the items grouped by type; within each group the items are sorted, but the order of the groups themselves is arbitrary and implementation-dependent, which means, "not dependable".

```
    >>> sorted(l, key=str)
    [0, '1', 14, 19, '23', 28, '28', 5, 6, '9']
```

Oracle, Google and the Timbot conspiracy

The sorting algorithm used in `sorted` and `list.sort` is Timsort, an adaptive algorithm that switches from insertion sort to merge sort strategies, depending on how ordered the data is. This is efficient because real world data tends to have runs of sorted items. There is a Wikipedia article about it.

Timsort was first deployed in CPython, in 2002. Since 2009, Timsort is also used sort arrays in both standard Java and Android, a fact that became widely known when Oracle used some of the code related to Timsort as evidence of Google infringement of Sun's intellectual property. See Oracle v. Google - Day 14 Filings.

Timsort was invented by Tim Peters, a Python core developer so prolific that he is believed to be an AI, the Timbot. You can read about that conspiracy theory in Python Humor. Tim also wrote the Zen of Python: `import this`.

CHAPTER 3
Dictionaries and sets

> Any running Python program has many dictionaries active at the same time, even if the user's program code doesn't explicitly use a dictionary.
>
> — A. M. Kuchling
> *Beautiful Code, Chapter 18, Python's Dictionary Implementation*

The `dict` type is not only widely used in our programs but also a fundamental part of the Python implementation. Module namespaces, class and instance attributes and function keyword arguments are some of the fundamental constructs where dictionaries are deployed. The built-in functions live in `__builtins__.__dict__`.

Because of their crucial role, Python dicts are highly optimized. *Hash tables* are the engines behind Python's high performance dicts.

We also cover sets in this chapter because they are implemented with hash tables as well. Knowing how a hash table works is key to making the most of dictionaries and sets.

Here is a brief outline of this chapter:

- Common dictionary methods
- Special handling for missing keys
- Variations of `dict` in the standard library
- The `set` and `frozenset` types
- How hash tables work
- Implications of hash tables: key type limitations, unpredictable ordering etc.

Generic mapping types

The `collections.abc` module provides the `Mapping` and `MutableMapping` ABCs to formalize the interfaces of `dict` and similar types[1]. See Figure 3-1.

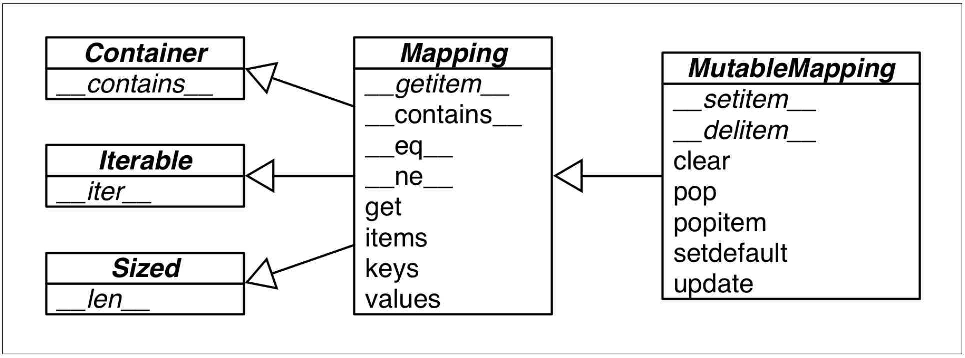


Figure 3-1. UML class diagram for the `MutableMapping` and its superclasses from `collections.abc`. Inheritance arrows point from subclasses to superclasses. Names in italic are abstract classes and abstract methods.

Implementations of specialized mappings often extend `dict` or `collections.UserDict`, instead of these ABCs. The main value of the ABCs is documenting and formalizing the minimal interfaces for mappings, and serving as criteria for `isinstance` tests in code that needs to support mappings in a broad sense:

```
>>> my_dict = {}
>>> isinstance(my_dict, abc.Mapping)
True
```

Using `insinstace` is better than checking whether a function argument is of `dict` type, because then alternative mapping types can be used.

All mapping types in the standard library use the basic `dict` in their implementation, so they share the limitation that the keys must be *hashable* (the values need not be hashable, only the keys).

What is hashable?

Here is part of the definition of hashable from the Python Glossary:

[1] In Python 2.6 to 3.2 these classes are imported from the `collections` module, and not from `collections.abc`.

> An object is hashable if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value. […]

The atomic immutable types (`str`, `bytes`, numeric types) are all hashable. A `frozenset` is always hashable, because its elements must be hashable by definition. A `tuple` is hashable only if all its items are hashable. See tuples `tt`, `tl` and `tf`:

```
>>> tt = (1, 2, (30, 40))
>>> hash(tt)
8027212646858338501
>>> tl = (1, 2, [30, 40])
>>> hash(tl)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> tf = (1, 2, frozenset([30, 40]))
>>> hash(tf)
-4118419923444501110
```

At this writing the Python Glossary states: “All of Python’s immutable built-in objects are hashable” but that is inaccurate because a `tuple` is immutable, yet it may contain references to unhashable objects.

User-defined types are hashable by default because their hash value is their `id()` and they all compare not equal. If an object implements a custom `__eq__` that takes into account its internal state, it may be hashable only if all its attributes are immutable.

Given these ground rules, you can build dictionaries in several ways. The Built-in Types page in the Library Reference has this example to show the various means of building a dict:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

In addition to the literal syntax and the flexible `dict` constructor, we can use *dict comprehensions* to build dictionaries. See the next section.

`dict` comprehensions

Since Python 2.7 the syntax of listcomps and genexps was applied to `dict` comprehensions (and `set` comprehensions as well, which we'll soon visit). A *dictcomp* builds a `dict` instance by producing `key:value` pair from any iterable.

Example 3-1. Examples of `dict` comprehensions

```
>>> DIAL_CODES = [                          ❶
...         (86, 'China'),
...         (91, 'India'),
...         (1, 'United States'),
...         (62, 'Indonesia'),
...         (55, 'Brazil'),
...         (92, 'Pakistan'),
...         (880, 'Bangladesh'),
...         (234, 'Nigeria'),
...         (7, 'Russia'),
...         (81, 'Japan'),
...     ]
>>> country_code = {country: code for code, country in DIAL_CODES}  ❷
>>> country_code
{'China': 86, 'India': 91, 'Bangladesh': 880, 'United States': 1,
'Pakistan': 92, 'Japan': 81, 'Russia': 7, 'Brazil': 55, 'Nigeria':
234, 'Indonesia': 62}
>>> {code: country.upper() for country, code in country_code.items()  ❸
...  if code < 66}
{1: 'UNITED STATES', 55: 'BRAZIL', 62: 'INDONESIA', 7: 'RUSSIA'}
```

❶ A list of pairs can be used directly with the `dict` constructor.

❷ Here the pairs are reversed: `country` is the key, and `code` is the value.

❸ Reversing the pairs again, values upper-cased and items filtered by `code < 66`.

If you're used to liscomps, dictcomps are a natural next step. If you aren't, the spread of the listcomp syntax means it's now more profitable than ever to become fluent in it.

We now move to a panoramic view of the API for mappings.

Overview of common mapping methods

The basic API for mappings is quite rich. Table 3-1 shows the methods implemented by `dict` and two of its most useful variations: `defaultdict` and `OrderedDict`, both defined in the `collections` module.

Table 3-1. Methods of the mapping types `dict`, `collections.defaultdict` and `collections.OrderedDict` (common `object` methods omitted for brevity). Optional arguments are enclosed in «…».

	`dict`	`default dict`	`Ordered Dict`	
`d.clear()`	●	●	●	remove all items
`d.__contains__(k)`	●	●	●	`k in d`
`d.copy()`	●	●	●	shallow copy
`d.__copy__()`		●		support for `copy.copy`
`d.default_factory`		●		callable invoked by `__missing__` to set missing values[a]
`d.__delitem__(k)`	●	●	●	`del d[k]` — remove item with key `k`
`d.fromkeys(it, «initial»)`	●	●	●	new mapping from keys in iterable, with optional initial value (defaults to `None`)
`d.get(k, «default»)`	●	●	●	get item with key `k`, return `default` or `None` if missing
`d.__getitem__(k)`	●	●	●	`d[k]` — get item with key `k`
`d.items()`	●	●	●	get *view* over items — `(key, value)` pairs
`d.__iter__()`	●	●	●	get iterator over keys
`d.keys()`	●	●	●	get *view* over keys
`d.__len__()`	●	●	●	`len(d)` — number of items
`d.__missing__(k)`		●		called when `__getitem__` cannot find the key
`d.move_to_end(k, «last»)`			●	move `k` first or last position (`last` is `True` by default)
`d.pop(k, «default»)`	●	●	●	remove and return value at `k`, or `default` or `None` if missing
`d.popitem()`	●	●	●	remove and return an arbitrary `(key, value)` item[b]
`d.__reversed__()`			●	get iterator for keys from last to first inserted
`d.setdefault(k, «default»)`	●	●	●	if `k in d`, return `d[k]`; else set `d[k] = default` and return it
`d.__setitem__(k, v)`	●	●	●	`d[k] = v` — put `v` at `k`
`d.update(m, «**kargs»)`	●	●	●	update `d` with items from mapping or iterable of `(key, value)` pairs
`d.values()`	●	●	●	get *view* over values

	dict	default dict	Ordered Dict

[a] `default_factory` is not a method, but a callable instance attribute set by the end user when `defaultdict` is instantiated.

[b] `OrderedDict.popitem()` removes the first item inserted (FIFO); an optional `last` argument, if set to `True`, pops the last item (LIFO).

The way `update` handles its first argument `m` is a prime example of *duck typing*: it first checks whether `m` has a `keys` method and, if it does, assumes it is a mapping. Otherwise, `update` falls back to iterating over `m`, assuming its items are `(key, value)` pairs. The constructor for most Python mappings uses the logic of `update` internally, which means they can be initialized from other mappings or from any iterable object producing `(key, value)` pairs.

A subtle mapping method is `setdefault`. We don't always need it, but when we do, it provides a significant speedup by avoiding redundant key lookups. If you are not comfortable using it, the following section explains how, through a practical example.

Handling missing keys with `setdefault`

In line with the *fail-fast* philosophy, `dict` access with `d[k]` raises an error when `k` is not an existing key. Every Pythonista knows that `d.get(k, default)` is an alternative to `d[k]` whenever a default value is more convenient than handling `KeyError`. However, when updating the value found (if it is mutable), using either `__getitem__` or `get` is awkward and inefficient. Consider Example 3-2, a sub-optimal script written just to show one case where `dict.get` is not the best way to handling a missing key.

Example 3-2 is adapted from an example by Alex Martelli[2] which generates an index like that in Example 3-3.

Example 3-2. `index0.py` uses `dict.get` to fetch and update a list of word occurrences from the `index`. A better solution is in Example 3-4.

```
"""Build an index mapping word -> list of occurrences"""

import sys
import re

WORD_RE = re.compile('\w+')

index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
```

[2] The original script appears in slide 41 of Martelli's "Re-learning Python" presentation. His script is actually a demonstration of `dict.setdefault`, as shown in our Example 3-4.

```
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start()+1
            location = (line_no, column_no)
            # this is ugly; coded like this to make a point
            occurrences = index.get(word, [])  ❶
            occurrences.append(location)       ❷
            index[word] = occurrences          ❸

# print in alphabetical order
for word in sorted(index, key=str.upper):  ❹
    print(word, index[word])
```

❶ Get the list of occurrences for `word`, or `[]` if not found.

❷ Append new location to `occurrences`.

❸ Put changed `occurrences` into to `index` dict; this entails a second search through the `index`.

❹ In the `key=` argument of `sorted` I am not calling `str.upper`, just passing a reference to that method so the `sorted` function can use it to normalize the words for sorting[3].

Example 3-3. Partial output from Example 3-2 processing the Zen of Python. Each line shows a word and a list of occurrences coded as pairs: (line-number, column-number).

```
$ python3 index0.py ../../data/zen.txt
a [(19, 48), (20, 53)]
Although [(11, 1), (16, 1), (18, 1)]
ambiguity [(14, 16)]
and [(15, 23)]
are [(21, 12)]
aren [(10, 15)]
at [(16, 38)]
bad [(19, 50)]
be [(15, 14), (16, 27), (20, 50)]
beats [(11, 23)]
Beautiful [(3, 1)]
better [(3, 14), (4, 13), (5, 11), (6, 12), (7, 9), (8, 11),
(17, 8), (18, 25)]
...
```

The three lines dealing with `occurrences` in Example 3-2 can be replaced by a single line using `dict.setdefault`. Example 3-4 is closer to Alex Martelli's original example.

[3] This is an example of using a method as a first class function, the subject of Chapter 5.

Example 3-4. `index.py` uses `dict.setdefault` to fetch and update a list of word occurrences from the `index` in a single line Example 3-4.

```
"""Build an index mapping word -> list of occurrences"""

import sys
import re

WORD_RE = re.compile('\w+')

index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start()+1
            location = (line_no, column_no)
            index.setdefault(word, []).append(location)   ❶

# print in alphabetical order
for word in sorted(index, key=str.upper):
    print(word, index[word])
```

❶ Get the list of occurrences for `word`, or set it to `[]` if not found; `setdefault` returns the value, so it can be updated without requiring a second search.

In other words, the end result of this line…

```
my_dict.setdefault(key, []).append(new_value)
```

…is the same as running…

```
if key not in my_dict:
    my_dict[key] = []
my_dict[key].append(new_value)
```

…except that the latter code performs at least two searches for `key` — three if it's not found — while `setdefault` does it all with a single lookup.

A related issue, handling missing keys on any lookup (and not only when inserting), is the subject of the next section.

Mappings with flexible key lookup

Sometimes it is convenient to have mappings that return some made-up value when a missing key is searched. There are two main approaches to this: one is to use a `defaultdict` instead of a plain `dict`. The other is to subclass `dict` or any other mapping type and add a `__missing__` method. Both solutions are covered next.

`defaultdict`: another take on missing keys

Example 3-5 uses `collections.defaultdict` to provide another elegant solution to the problem in Example 3-4. A `defaultdict` is configured to create items on demand whenever a missing key is searched.

Here is how it works: when instantiating a `defaultdict`, you provide a callable which is used to produce a default value whenever `__getitem__` is passed a non-existent key argument.

For example, given an empty `defaultdict` created as `dd = defaultdict(list)`, if `'new-key'` is not in `dd` then the expression `dd['new-key']` does the following steps:

1. calls `list()` to create a new list;
2. inserts the list into `dd` using `'new-key'` as key;
3. returns a reference to that list.

The callable that produces the default values is held in an instance attribute called `default_factory`.

Example 3-5. `index.py` uses `dict.setdefault` to fetch and update a list of word occurrences from the `index` in a single line Example 3-4.

```
"""Build an index mapping word -> list of occurrences"""

import sys
import re
import collections

WORD_RE = re.compile('\w+')

index = collections.defaultdict(list)     ❶
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start()+1
            location = (line_no, column_no)
            index[word].append(location)  ❷

# print in alphabetical order
for word in sorted(index, key=str.upper):
    print(word, index[word])
```

❶ Create a `defaultdict` with the `list` constructor as `default_factory`.

❷ If `word` is not initially in the `index`, the `default_factory` is called to produce the missing value, which in this case is an empty `list` that is then assigned to `index[word]` and returned, so the `.append(location)` operation always succeeds.

If no `default_factory` is provided, the usual `KeyError` is raised for missing keys.

> The `default_factory` of a `defaultdict` is only invoked to provide default values for `__getitem__` calls, and not for the other methods. For example, if `dd` is a `defaultdict`, and `k` is a missing key, `dd[k]` will call the `default_factory` to create a default value, but `dd.get(k)` still returns `None`.

The mechanism that makes `defaultdict` work by calling `default_factory` is actually the `__missing__` special method, a feature supported by all standard mapping types that we discuss next.

The `__missing__` method

Underlying the way mappings deal with missing keys is the aptly named `__missing__` method. This method is not defined in the base `dict` class, but `dict` is aware of it: if you subclass `dict` and provide a `__missing__` method, the standard `dict.__getitem__` will call it whenever a key is not found, instead of raising `KeyError`.

> The `__missing__` method is just called by `__getitem__`, i.e. for the `d[k]` operator. The presence of a `__missing__` method has no effect on the behavior of other methods that look up keys, such as `get` or `__contains__` (which implements the `in` operator). This is why the `default_factory` of `defaultdict` works only with `__getitem__`, as noted in the warning at the end of the previous section.

Suppose you'd like a mapping where keys are converted to `str` when looked up. A concrete use case is the Pingo.io project, where a programmable board with GPIO pins — like the Raspberry Pi or the Arduino — is represented by a `board` object with a `board.pins` attribute which is a mapping of physical pin locations to pin objects, and the physical location may be just a number or a string like `"A0"` or `"P9_12"`. For consistency, it is desirable that all keys in `board.pins` are strings, but it is also convenient that looking up `my_arduino.pin[13]` works as well, so beginners are not tripped when they want to blink the LED on pin 13 of their Arduinos. Example 3-6 shows how such a mapping would work.

Example 3-6. When searching for a non-string key, `StrKeyDict0` converts it to `str` when it is not found.

```
Tests for item retrieval using `d[key]` notation::

    >>> d = StrKeyDict0([('2', 'two'), ('4', 'four')])
    >>> d['2']
    'two'
    >>> d[4]
    'four'
    >>> d[1]
    Traceback (most recent call last):
      ...
    KeyError: '1'

Tests for item retrieval using `d.get(key)` notation::

    >>> d.get('2')
    'two'
    >>> d.get(4)
    'four'
    >>> d.get(1, 'N/A')
    'N/A'


Tests for the `in` operator::

    >>> 2 in d
    True
    >>> 1 in d
    False
```

Example 3-7 implements a class `StrKeyDict0` that passes the tests above.

A better way to create a user-defined mapping type is to subclass `collections.UserDict` instead of `dict`. We'll do that in Example 3-8. Here we subclass `dict` just to show that `__missing__` is supported by the built-in `dict.__getitem__` method.

Example 3-7. `StrKeyDict0` converts non-string keys to `str` on lookup. See tests in Example 3-6

```
class StrKeyDict0(dict):  ❶

    def __missing__(self, key):
        if isinstance(key, str):  ❷
            raise KeyError(key)
        return self[str(key)]  ❸

    def get(self, key, default=None):
        try:
            return self[key]  ❹
        except KeyError:
            return default  ❺

    def __contains__(self, key):
        return key in self.keys() or str(key) in self.keys()  ❻
```

❶ `StrKeyDict0` inherits from `dict`.

❷ Check whether `key` is already a `str`. If it is, and it's missing, raise `KeyError`.

❸ Build `str` from `key` and look it up.

❹ The `get` method delegates to `__getitem__` by using the `self[key]` notation; that gives the opportunity for our `__missing__` to act.

❺ If a `KeyError` was raised, `__missing__` already failed, so we return the `default`.

❻ Search for unmodified key (the instance may contain non-str keys), then for a `str` built from the key.

Consider for a moment why the test `isinstance(key, str)` is needed in the `__missing__` implementation.

Without that test, our `__missing__` method would work OK for any key `k` — `str` or not `str` — whenever `str(k)` produced an existing key. But if `str(k)` is not an existing key, then we'd have an infinite recursion. The last line, `self[str(key)]` would call `__getitem__` passing that `str` key, which in turn would call `__missing__` again.

The `__contains__` method is also needed for consistent behavior in this example, because the operation `k in d` calls it, but the method inherited from `dict` does not fall back to invoking `__missing__`. There is a subtle detail in our implementation of `__contains__`: we do not check for the key in the usual Pythonic way — `k in my_dict` — because `str(key) in self` would recursively call `__contains__`. We avoid this by explicitly looking up the key in `self.keys()`.

> A search like `k in my_dict.keys()` is efficient in Python 3 even for very large mappings because `dict.keys()` returns a view, which is similar to a set, and containment checks in sets are as fast as in dicts. Details are documented in Dictionary view objects. In Python 2, `dict.keys()` returns a `list`, so our solution also works there, but it is not efficient for large dicts, because `k in my_list` must scan the list.

The check for the unmodified key — `key in self.keys()` — is necessary for correctness because `StrKeyDict0` does not enforce that all keys in the dict must be of type str. Our only goal with this simple example is to make searching “friendlier” and not enforce types.

So far we have covered the dict and defaultdict mapping types, but the standard library comes with other mapping implementations, which we discuss next.

Variations of dict

In this section we summarize the various mapping types included in collections module of the standard library, besides defaultdict.

collections.OrderedDict
: maintains keys in insertion order, allowing iteration over items in a predictable order. The popitem method of an OrderedDict pops the first item by default, but if called as my_odict.popitem(last=True), it pops the last item added.

collections.ChainMap
: holds a list of mappings which can be searched as one. The lookup is performed on each mapping in order, and succeeds if the key is found in any of them. This is useful to interpreters for languages with nested scopes, where each mapping represents a scope context. The ChainMap section in the collections docs has several examples of ChainMap usage, including this snippet inspired by the basic rules of variable lookup in Python:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

collections.Counter
: a mapping that holds an integer count for each key. Updating an existing key adds to its count. This can be used to count instances of hashable objects (the keys) or as a multiset — a set that can hold several occurrences of each element. Counter implements the + and - operators to combine tallies, and other useful methods such as most_common([n]), which returns an ordered list of tuples with the n most common items and their counts; see the documentation. Here is Counter used to count letters in words:

```
>>> ct = collections.Counter('abracadabra')
>>> ct
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.update('aaaaazzz')
>>> ct
Counter({'a': 10, 'z': 3, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.most_common(2)
[('a', 10), ('z', 3)]
```

collections.UserDict
: a pure Python implementation of a mapping that works like a standard dict.

While OrderedDict, ChainMap and Counter come ready to use, UserDict is designed to be subclassed, as we'll do next.

Subclassing UserDict.

It's almost always easier to create a new mapping type by extending UserDict than dict. Its value can be appreciated as we extend our StrKeyDict0 from Example 3-7 to make sure that any keys added to the mapping are stored as str.

The main reason why it's preferable to subclass from UserDict than dict is that the built-in has some implementation shortcuts that end up forcing us to override methods that we can just inherit from UserDict with no problems[4].

Note that UserDict does not inherit from dict, but has an internal dict instance, called data, which holds the actual items. This avoids undesired recursion when coding special methods like __setitem__, and simplifies the coding of __contains__, compared to Example 3-7.

Thanks to UserDict, StrKeyDict (Example 3-8) is actually shorter than StrKeyDict0 (Example 3-7), but it does more: it stores all keys as str, avoiding unpleasant surprises if the instance is built or updated with data containing non-string keys.

Example 3-8. StrKeyDict always converts non-string keys to str — on insertion, update and lookup.

```
import collections


class StrKeyDict(collections.UserDict):  ❶

    def __missing__(self, key):  ❷
        if isinstance(key, str):
            raise KeyError(key)
        return self[str(key)]

    def __contains__(self, key):
        return str(key) in self.data  ❸

    def __setitem__(self, key, item):
        self.data[str(key)] = item  ❹
```

❶ StrKeyDict extends UserDict

❷ __missing__ is exactly as in Example 3-7

[4] The exact problem with subclassing dict and other built-ins is covered in "Subclassing built-in types is tricky" on page 350.

❸ `__contains__` is simpler: we can assume all stored keys are `str` and we can check on `self.data` instead of invoking `self.keys()` as we did in `StrKeyDict0`.

❹ `__setitem__` converts any `key` to a `str`. This method is easier to overwrite when we can delegate to the `self.data` attribute.

Because `UserDict` subclasses `MutableMapping`, the remaining methods that make `StrKeyDict` a full-fledged mapping are inherited from `UserDict`, `MutableMapping` or `Mapping`. The latter have several useful concrete methods, in spite of being ABCs (abstract base classes). Worth noting:

`MutableMapping.update`

This powerful method can be called directly but is also used by `__init__` to load the instance from other mappings, from iterables of `(key, value)` pairs and keyword arguments. Because it uses `self[key] = value` to add items, it ends up calling our implementation of `__setitem__`.

`Mapping.get`

In `StrKeyDict0` (Example 3-7) we had to code our own `get` to obtain results consistent with `__getitem__`, but in Example 3-8 we inherited `Mapping.get` which is implemented exactly like `StrKeyDict0.get` (see Python source code).

> After I wrote `StrKeyDict` I discovered that Antoine Pitrou authored PEP 455 — Adding a key-transforming dictionary to collections and a patch to enhance the `collections` module with a `TransformDict`. The patch is attached to issue18986 and may land in Python 3.5. To experiment with `TransformDict` I extracted it into a stand-alone module `03-dict-set/transformdict.py` in the Fluent Python code repository. `TransformDict` is more general than `StrKeyDict`, and is complicated by the requirement to preserve the keys as they were originally inserted.

We know there are several immutable sequence types, but how about an immutable dictionary? Well, there isn't a real one in the standard library, but a stand-in is available. Read on.

Immutable mappings

The mapping types provided by the standard library are all mutable, but you may need to guarantee that a user cannot change a mapping by mistake. A concrete use case can be found, again, in the Pingo.io project I described in "The `__missing__` method" on page 72: the `board.pins` mapping represents the physical GPIO pins on the device. As such, it's nice to prevent inadvertent updates to `board.pins` because the hardware can't possibly be changed via software, so any change in the mapping would make it inconsistent with the physical reality of the device.

Since Python 3.3 the `types` module provides a wrapper class `MappingProxyType` which, given a mapping, returns a `mappingproxy` instance that is a read-only but dynamic view of the original mapping. This means that updates to the original mapping can be seen in the `mappingproxy`, but changes cannot be made through it. See Example 3-9 for a brief demonstration.

Example 3-9. `MappingProxyType` builds a read-only `mappingproxy` instance from a `dict`.

```
>>> from types import MappingProxyType
>>> d = {1: 'A'}
>>> d_proxy = MappingProxyType(d)
>>> d_proxy
mappingproxy({1: 'A'})
>>> d_proxy[1]  ❶
'A'
>>> d_proxy[2] = 'x'  ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'mappingproxy' object does not support item assignment
>>> d[2] = 'B'
>>> d_proxy  ❸
mappingproxy({1: 'A', 2: 'B'})
>>> d_proxy[2]
'B'
>>>
```

❶ Items in `d` can be seen through `d_proxy`.

❷ Changes cannot be made through `d_proxy`.

❸ `d_proxy` is dynamic: any change in `d` is reflected.

Here is how this could be used in practice in the Pingo.io scenario: the constructor in a concrete `Board` subclass would fill a private mapping with the pin objects, and expose it to clients of the API via a public `.pins` attribute implemented as a `mappingproxy`. That way the clients would not be able to add, remove or change pins by accident[5].

After covering most mapping types in the standard library and when to use them, we now move to the set types.

[5] We are not actually using `MappingProxyType` in Pingo.io because it is new in Python 3.3 and we need to support Python 2.7 at this time.

Set theory

Sets are a relatively new addition in the history of Python, and somewhat underused. The `set` type and its immutable sibling `frozenset` first appeared in a module in Python 2.3 and were promoted to built-ins in Python 2.6.

> In this book, the word "set" is used to refer both to `set` and `frozenset`. When talking specifically about the `set` class, its name appears in the constant width font used for source code: `set`.

A set is a collection of unique objects. A basic use case is removing duplication:

```
>>> l = ['spam', 'spam', 'eggs', 'spam']
>>> set(l)
{'eggs', 'spam'}
>>> list(set(l))
['eggs', 'spam']
```

Set elements must be hashable. The `set` type is not hashable, but `frozenset` is, so you can have `frozenset` elements inside a `set`.

In addition to guaranteeing uniqueness, the set types implement the essential set operations as infix operators, so, given two sets `a` and `b`, `a | b` returns their union, `a & b` computes the intersection, and `a - b` the difference. Smart use of set operations can reduce both the line count and the run time of Python programs, at the same time making code easier to read and reason about — by removing loops and lots of conditional logic.

For example, imagine you have a large set of e-mail addresses (the `haystack`) and a smaller set of addresses (the `needles`) and you need to count how many of the `needles` occur in the `haystack`. Thanks to `set` intersection (the `&` operator) you can code that in a simple line:

Example 3-10. Count occurrences `needles` in a `haystack`, both of type `set`

```
found = len(needles & haystack)
```

Without the intersection operator, you'd have write Example 3-11 to accomplish the same task as Example 3-10:

Example 3-11. Count occurrences of `needles` in a `haystack` (same end result as Example 3-10).

```
found = 0
for n in needles:
    if n in haystack:
        found += 1
```

Example 3-10 runs slightly faster than Example 3-11. On the other hand, Example 3-11 works for any iterable objects `needles` and `haystack`, while Example 3-10 requires that both be sets. But, if you don't have sets on hand, you can always build them on the fly:

Example 3-12. Count occurrences `needles` in a `haystack`; these lines work for any iterable types.

```
found = len(set(needles) & set(haystack))

# another way:
found = len(set(needles).intersection(haystack))
```

Of course, there is an extra cost involved in building the sets in Example 3-12, but if either the `needles` or the `haystack` is already a set, the alternatives in Example 3-12 may be cheaper than Example 3-11.

Any one of the above examples are capable of searching 1,000 values in a `haystack` of 10,000,000 items in a little over 3 milliseconds — that's about 3 microseconds per needle.

Besides the extremely fast membership test (thanks to the underlying hash table), the `set` and `frozenset` built-in types provide a rich selection of operations to create new sets or — in the case of `set` — to change existing ones. We will discuss the operations shortly, but first a note about syntax.

`set` literals

The syntax of `set` literals — `{1}`, `{1, 2}`, etc. — looks exactly like the math notation, with one important exception: there's no literal notation for the empty `set`, we must remember to write `set()`.

Syntax quirk

Don't forget: to create an empty `set`, use the constructor without an argument: `set()`. If you write `{}`, you're creating an empty `dict` — this hasn't changed.

In Python 3, the standard string representation of sets always uses the `{...}` notation, except for the empty set:

```
>>> s = {1}
>>> type(s)
<class 'set'>
>>> s
{1}
>>> s.pop()
1
>>> s
set()
```

Literal set syntax like {1, 2, 3} is both faster and more readable than calling the constructor e.g. set([1, 2, 3]). The latter form is slower because, to evaluate it, Python has to look up the set name to fetch the constructor, then build a list and finally pass it to the constructor. In contrast, to process the a literal like {1, 2, 3}, Python runs a specialized BUILD_SET bytecode.

Comparing bytecodes for {1} and set([1])

Take a look at the bytecode for the two operations, as output by dis.dis (the disassembler function):

```
>>> from dis import dis
>>> dis('{1}')                                        ❶
  1           0 LOAD_CONST               0 (1)
              3 BUILD_SET                1            ❷
              6 RETURN_VALUE
>>> dis('set([1])')                                   ❸
  1           0 LOAD_NAME                0 (set) ❹
              3 LOAD_CONST               0 (1)
              6 BUILD_LIST               1
              9 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             12 RETURN_VALUE
```

❶ Disassemble bytecode for literal expression {1}.

❷ Special BUILD_SET bytecode does almost all the work.

❸ Bytecode for set([1]).

❹ Three operations instead of BUILD_SET: LOAD_NAME, BUILD_LIST, CALL_FUNCTION.

There is no special syntax to represent frozenset literals — they must be created by calling the constructor. The standard string representation in Python 3 looks like a frozenset constructor call. Note the output in the console session:

```
>>> frozenset(range(10))
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

Speaking of syntax, the familiar shape of listcomps was adapted to build sets as well.

set comprehensions

Set comprehensions (*setcomps*) were added in Python 2.7, together with the dictcomps which we saw in "dict comprehensions" on page 66. Here is a simple example:

Example 3-13. Build a `set` of Latin-1 characters that have the word "SIGN" in their Unicode names.

```
>>> from unicodedata import name  ❶
>>> {chr(i) for i in range(32, 256) if 'SIGN' in name(chr(i),'')}  ❷
{'§', '=', '¢', '#', '¤', '<', '¥', 'µ', '×', '$', '¶', '£', '©',
'°', '+', '÷', '±', '>', '¬', '®', '%'}
```

❶ Import `name` function from `unicodedata` to obtain character names.

❷ Build set of characters with codes from 32 to 255 that have the word `'SIGN'` in their names.

Syntax matters aside, let's now overview the rich assortment of operations provided by sets.

Set operations

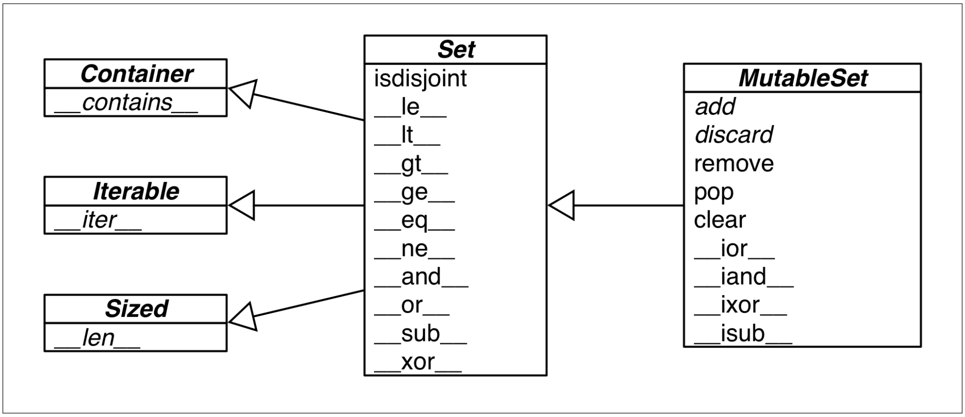


Figure 3-2. UML class diagram for `MutableSet` and its superclasses from `collections.abc`. Names in italic are abstract classes and abstract methods. Reverse operator methods omitted for brevity.

Figure 3-2 gives an overview of the methods you can expect from mutable and immutable sets. Many of them are special methods for operator overloading. Table 3-2 shows the math set operators that have corresponding operators or methods in Python. Note that some operators and methods perform in-place changes on the target set (e.g. `&=`, `difference_update` etc.). Such operations make no sense in the ideal world of mathematical sets, and are not implemented in `frozenset`.

The infix operators in Table 3-2 require that both operands be sets, but all other methods take one or more iterable arguments. For example, to produce the union of four collections a, b, c and d, you can call a.union(b, c, d), where a must be a set, but b, c and d can be iterables of any type.

Table 3-2. Mathematical set operations: these methods either produce a new set or update the target set in place, if it's mutable.

Math symbol	Python operator	method	description
S∩Z	s & z	s.__and__(z)	intersection of s and z
	z & s	s.__rand__(z)	reversed & operator
		s.intersection(it, …)	intersection of s and all sets built from iterables it etc.
	s &= z	s.__iand__(z)	s updated with intersection of s and z
		s.intersection_update(it, …)	s updated with intersection of s and all sets built from iterables it etc.
S∪Z	s | z	s.__or__(z)	union of s and z
	z | s	s.__ror__(z)	reversed |
		s.union(it, …)	union of s and all sets built from iterables it etc.
	s |= z	s.__ior__(z)	s updated with union of s and z
		s.update(it, …)	s updated with union of s and all sets built from iterables it etc.
S\Z	s - z	s.__sub__(z)	relative complement or difference between s and z
	z - s	s.__rsub__(z)	reversed - operator
		s.difference(it, …)	difference between s all sets built from iterables it etc.
	s -= z	s.__isub__(z)	s updated with difference between s and z
		s.difference_update(it, …)	s updated with difference between s all sets built from iterables it etc.
		s.symmetric_difference(it)	complement of s & set(it)
SΔZ	s ^ z	s.__xor__(z)	symmetric difference (the complement of the intersection s & z)
	z ^ s	s.__rxor__(z)	reversed ^ operator
		s.symmetric_difference_update(it, …)	s updated with symmetric difference of s and all sets built from iterables it etc.
	s ^= z	s.__ixor__(z)	s updated with symmetric difference of s and z

As I write this, there is a Python bug report (issue 8743) that says: "The set() operators (*or*, *and*, *sub*, *xor*, and their in-place counterparts) require that the parameter also be an instance of set().", with the undesired side-effect these operators don't work with `collections.abc.Set` subclasses. The bug is already fixed in trunk for Python 2.7 and 3.4, and should be history by the time you read this.

Table 3-3 lists set predicates: operators and methods that return `True` or `False`.

Table 3-3. Set comparison operators and methods that return a ***`bool`****.*

Math symbol	Python operator	method	description
		`s.isdisjoint(z)`	`s` and `z` are disjoint (have no elements in common)
e ∈ S	`e in s`	`s.__contains__(e)`	element `e` is a member of `s`
S ⊆ Z	`s <= z`	`s.__le__(z)`	`s` is a subset of the `z` set
		`s.issubset(it)`	`s` is a subset of the set built from the iterable `it`
S ⊂ Z	`s < z`	`s.__lt__(z)`	`s` is a proper subset of the `z` set
S ⊇ Z	`s >= z`	`s.__ge__(z)`	`s` is a superset of the `z` set
		`s.issuperset(it)`	`s` is a superset of the set built from the iterable `it`
S ⊃ Z	`s > z`	`s.__gt__(z)`	`s` is a proper superset of the `z` set

In addition to the operators and methods derived from math set theory, the set types implement other methods of practical use, summarized in Table 3-4.

Table 3-4. Additional set methods.

	`set`	`frozenset`	
`s.add(e)`	●		Add element `e` to `s`.
`s.clear()`	●		Remove all elements of `s`.
`s.copy()`	●	●	Shallow copy of `s`.
`s.discard(e)`	●		Remove element `e` from `s` if it is present.
`s.__iter__()`	●	●	Get iterator over `s`.
`s.__len__()`	●	●	`len(s)`
`s.pop()`	●		Remove and return an element from `s`, raising `KeyError` if `s` is empty.
`s.remove(e)`	●		Remove element `e` from `s`, raising `KeyError` if `e not in s`.

This completes our overview of the features of sets.

We now change gears to a discussion of how dictionaries and sets are implemented with hash tables. After reading the rest of this chapter you will no longer be surprised by the apparently unpredictable behavior sometimes exhibited by `dict`, `set` and their brethren.

`dict` and `set` under the hood

Understanding how Python dictionaries and sets are implemented using hash tables is helpful to make sense of their strengths and limitations.

Here are some questions this section will answer:

- How efficient are Python `dict` and `set`?
- Why are they unordered?
- Why can't we use any Python object as a `dict` key or `set` element?
- Why does the order of the `dict` keys or `set` elements depend on insertion order, and may change during the lifetime of the structure?
- Why is it bad to add items to a `dict` or `set` while iterating through it?

To motivate the study of hash tables, we start by showcasing the amazing performance of `dict` and `set` with a simple test involving millions of items.

A performance experiment

From experience, all Pythonistas know that dicts and sets are fast. We'll confirm that with a controlled experiment.

To see how the size of a `dict`, `set` or `list` affects the performance of search using the `in` operator, I generated an array of 10 million distinct double-precision floats, the "haystack". I then generated an array of needles: 1,000 floats, with 500 picked from the haystack and 500 verified not to be in it.

For the `dict` benchmark I used `dict.fromkeys()` to create a `dict` named `haystack` with 1,000 floats. This was the setup for the `dict` test. The actual code I clocked with the `timeit` module is Example 3-14 (like Example 3-11).

Example 3-14. Search for `needles` in `haystack` and count those found.

```
found = 0
for n in needles:
    if n in haystack:
        found += 1
```

The benchmark was repeated another 4 times, each time increasing tenfold the size of `haystack`, to reach a size of 10,000,000 in the last test. The result of the `dict` performance test is in Table 3-5.

Table 3-5. Total time for using `in` operator to search for 1,000 `needles` in `haystack` dicts of five sizes on a Core i7 laptop running Python 3.4.0. Tests timed the loop in Example 3-14.

len of haystack	factor	dict time	factor
1,000	1x	0.000202s	1.00x
10,000	10x	0.000140s	0.69x
100,000	100x	0.000228s	1.13x
1,000,000	1,000x	0.000290s	1.44x
10,000,000	10,000x	0.000337s	1.67x

In concrete terms, on my laptop the time to check for the presence of 1,000 floating point keys in a dictionary with 1,000 items was 0.000202s, and the same search in a `dict` with 10,000,000 items took 0.000337s. In other words: the time per search in the haystack with 10 million items was 0.337µs for each needle — yes, that is about one third of a microsecond per needle.

To compare, I repeated the benchmark, with the same haystacks of increasing size, but storing the `haystack` as a `set` or as `list`. For the `set` tests, in addition to timing the `for` loop in Example 3-14, I also timed the one-liner in Example 3-15, which produces the same result: count the number of elements from `needles` that are also in `haystack`.

Example 3-15. Use set intersection to count the `needles` that occur in `haystack`.

```
found = len(needles & haystack)
```

Table 3-6 shows the tests side-by-side. The best times are in the “set& time” column, which displays results for the set `&` operator using the code from Example 3-15. The worst times are — as expected — in the `list time` column, because there is no hash table to support searches with the `in` operator on a `list`, so a full scan must be made, resulting in times that grow linearly with the size of the haystack.

Table 3-6. Total time for using `in` operator to search for 1,000 keys in haystacks of 5 sizes, stored as dicts, sets and lists on a Core i7 laptop running Python 3.4.0. Tests timed the loop in Example 3-14 except the `set&` which uses Example 3-15.

len of haystack	factor	dict time	factor	set time	factor	set& time	factor	list time	factor
1,000	1x	0.000202s	1.00x	0.000143s	1.00x	0.000087s	1.00x	0.010556s	1.00x
10,000	10x	0.000140s	0.69x	0.000147s	1.03x	0.000092s	1.06x	0.086586s	8.20x
100,000	100x	0.000228s	1.13x	0.000241s	1.69x	0.000163s	1.87x	0.871560s	82.57x
1,000,000	1,000x	0.000290s	1.44x	0.000332s	2.32x	0.000250s	2.87x	9.189616s	870.56x
10,000,000	10,000x	0.000337s	1.67x	0.000387s	2.71x	0.000314s	3.61x	97.948056s	9,278.90x

If your program does any kind of I/O, the lookup time for keys in dicts or sets is negligible, regardless of the dict or set size (as long as it does fit in RAM). See the code used to generate Table 3-6 and accompanying discussion in Appendix A, Example A-1.

Now that we have concrete evidence of the speed of dicts and sets, let's explore how that is achieved. The discussion of the hash table internals explains, for example, why the key ordering is apparently random and unstable.

Hash tables in dictionaries

> To simplify the ensuing presentation, we will focus on the internals of dict first, and later transfer the concepts to sets.

This is a high level view of how Python uses a hash table to implement a dict. Many details are omitted — the CPython code has some optimization tricks[6] — but the overall description is accurate.

A hash table is a sparse array, i.e. an array which always has empty cells. In standard data structure texts, the cells in a hash table are often called "buckets". In a dict hash table, there is a bucket for each item, and it contains two fields: a reference to the key and a reference to the value of the item. Because all buckets have the same size, access to an individual bucket is done by offset.

Python tries to keep at least 1/3 of the buckets empty; if the hash table becomes too crowded, it is copied to a new location with room for more buckets.

To put an item in a hash table the first step is to calculate the *hash value* of the item key, which is done with the hash() built-in function, explained next.

Hashes and equality

The hash() built-in function works directly with built-in types and falls back to calling __hash__ for user-defined types. If two objects compare equal, their hash values must also be equal, otherwise the hash table algorithm does not work. For example, because 1 == 1.0 is true, hash(1) == hash(1.0) must also be true, even though the internal representation of an int and a float are very different[7].

[6] The source code for the CPython dictobject.c module is rich in comments. See also the reference for the Beautiful Code book in "Further reading" on page 94.

[7] Since we just mentioned int, here is a CPython implementation detail: the hash value of an int that fits in a machine word is the value of the int itself.

Also, to be effective as hash table indexes, hash values should scatter around the index space as much as possible. This means that, ideally, objects that are similar but not equal should have hash values that differ widely. Example 3-16 is the output of a script to compare the bit patterns of hash values. Note how the hashes of 1 and 1.0 are the same, but those of 1.0001, 1.0002 and 1.0003 are very different.

Example 3-16. Comparing hash bit patterns of 1, 1.0001, 1.0002 and 1.0003 on a 32-bit build of Python. Bits that are different in the hashes above and below are highlighted with !. The right column shows the number of bits that differ.

```
32-bit Python build
1        00000000000000000000000000000001
                                          != 0
1.0      00000000000000000000000000000001
------------------------------------------------
1.0      00000000000000000000000000000001
           ! !!! ! !! ! !    ! ! !! !!!   != 16
1.0001   00101110101101010000101011011101
------------------------------------------------
1.0001   00101110101101010000101011011101
          !!!  !!!! !!!!!   !!!!! !!  !   != 20
1.0002   01011101011010100001010110111001
------------------------------------------------
1.0002   01011101011010100001010110111001
          ! !   ! !!! ! !  !! ! !  ! !!!! != 17
1.0003   00001100000111110010000010010110
------------------------------------------------
```

The code to produce Example 3-16 is in Appendix A. Most of it deals with formatting the output, but it is listed as Example A-3 for completeness.

Starting with Python 3.3, a random salt value is added to the hashes of `str`, `bytes` and `datetime` objects. The salt value is constant within a Python process but varies between interpreter runs. The random salt is a security measure to prevent a DOS attack. Details are in a note in the documentation for the `__hash__` special method.

With this basic understanding of object hashes, we are ready to dive into the algorithm that makes hash tables operate.

The hash table algorithm

To fetch the value at `my_dict[search_key]`, Python calls `hash(search_key)` to obtain the *hash value* of `search_key` and uses the least significant bits of that number as an offset to look up a bucket in the hash table (the number of bits used depends on the current size of the table). If the found bucket is empty, `KeyError` is raised. Otherwise, the found bucket has an item — a `found_key:found_value` pair — and then Python checks whether `search_key == found_key`. If they match, that was the item sought: `found_value` is returned.

However, if `search_key` and `found_key` do not match, this is a *hash collision*. This happens because a hash function maps arbitrary objects to a small number of bits, and — in addition — the hash table is indexed with a subset of those bits. To resolve the collision, the algorithm then takes different bits in the hash, massages them in a particular way and uses the result as an offset to look up a different bucket[8]. If that is empty, `KeyError` is raised; if not, either the keys match and the item value is returned, or the collision resolution process is repeated. See Figure 3-3 for a diagram of this algorithm.

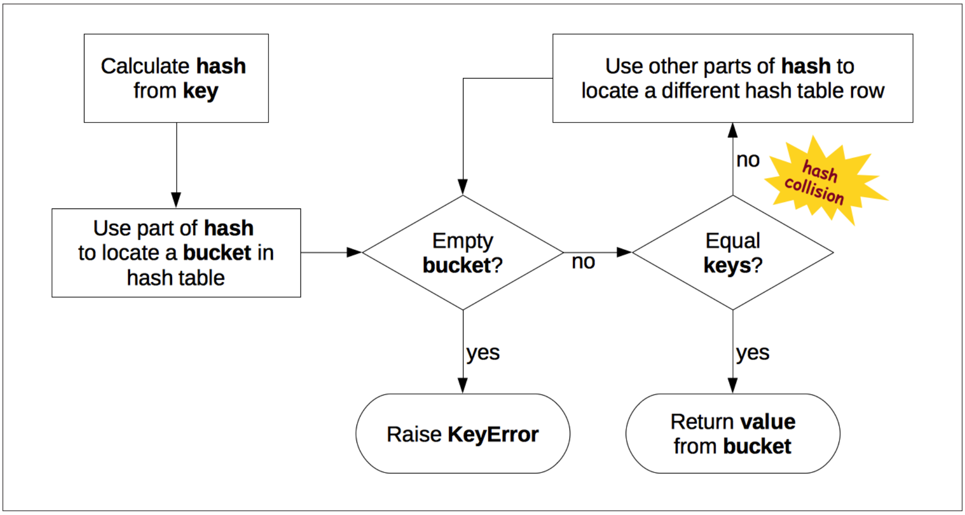

Calculate **hash** from **key**
Use part of **hash** to locate a **bucket** in hash table
Empty **bucket?**
no
Equal **keys?**
no
hash collision
Use other parts of **hash** to locate a different hash table row
yes
yes
Raise **KeyError**
Return **value** from **bucket**

Figure 3-3. Flowchart for retrieving an item from a `dict`. Given a `key`, this procedure either returns a `value` or raises `KeyError`

The process to insert or update an item is the same, except that when an empty bucket is located, the new item is put there, and when a bucket with a matching key is found, the value in that bucket is overwritten with the new value.

Additionally, when inserting items Python may determine that the hash table is too crowded and rebuild it to a new location with more room. As the hash table grows, so does the number of hash bits used as bucket offsets, and this keeps the rate of collisions low.

[8] The C function that shuffles the hash bits in case of collision has a curious name: `perturb`. See `dictobject.c` in the CPython source code for all the details.

This implementation may seem like a lot of work, but even with millions of items in a `dict`, many searches happen with no collisions, and the average number of collisions per search is between one and two. Under normal usage, even the unluckiest keys can be found after a handful of collisions are resolved.

Knowing the internals of the `dict` implementation we can explain the strengths and limitations of this data structure and all the others derived from it in Python. We are now ready to consider why Python `dict` behave as they do.

Practical consequences of how `dict` works

In the next five sections, we'll discuss the limitations and benefits that the underlying hash table implementation brings to `dict` usage.

#1: Keys must be hashable objects

An object is hashable if all of these requirements are met:

1. It supports the `hash()` function via a `__hash__()` method that always returns the same value over the lifetime of the object.
2. It supports equality via an `__eq__()` method.
3. If `a == b` is `True` then `hash(a) == hash(b)` must also be `True`.

User-defined types are hashable by default because their hash value is their `id()` and they all compare not equal.

> If you implement a class with a custom `__eq__` method then you must also implement a suitable `__hash__`, because you must always make sure that if `a == b` is `True` then `hash(a) == hash(b)` is also `True`. Otherwise you are breaking an invariant of the hash table algorithm, with the grave consequence that dicts and sets will not deal reliably with your objects. If a custom `__eq__` depends on mutable state, then `__hash__` must raise `TypeError` with a message like `unhashable type: 'MyClass'`.

#2: `dicts` have significant memory overhead

Because a `dict` uses a hash table internally, and hash tables must be sparse to work, they are not space efficient. For example, if you are handling a large quantity of records it makes sense to store them in a list of tuples or named tuples instead of using a list of dictionaries in JSON style, with one `dict` per record. Replacing dicts with tuples reduces the memory usage in two ways: by removing the overhead of one hash table per record and by not storing the field names again with each record.

For user-defined types, the `__slots__` class attribute changes the storage of instance attributes from a `dict` to a tuple in each instance. This will be discussed in "Saving space with the `__slots__` class attribute" on page 265 (Chapter 9).

Keep in mind we are talking about space optimizations. If you are dealing with a few million objects and your machine has gigabytes of RAM, you should postpone such optimizations until they are actually warranted. Optimization is the altar where maintainability is sacrificed.

#3: Key search is very fast

The `dict` implementation is an example of trading space for time: dictionaries have significant memory overhead, but they provide fast access regardless of the size of the dictionary — as long as it fits in memory. As Table 3-5 shows, when we increased the size of a `dict` from 1,000 to 10,000,000 elements, the time to search grew by a factor of 2.8, from 0.000163s to 0.000456s. The latter figure means we could search more than 2 million keys per second in a `dict` with 10 million items.

#4: Key ordering depends on insertion order

When a hash collision happens, the second key ends up in a position that it would not normally occupy if it had been inserted first. So, a `dict` built as `dict([(key1, value1), (key2, value2)])` compares equal to `dict([(key2, value2), (key1, value1)])`, but their key ordering may not be the same if the hashes of `key1` and `key2` collide.

Example 3-17 demonstrates the effect of loading three dicts with the same data, just in different order. The resulting dictionaries all compare equal, even if their order is not the same.

Example 3-17. `dialcodes.py` fills three dictionaries with the same data sorted in different ways. See output in Example 3-18

```
# dial codes of the top 10 most populous countries
DIAL_CODES = [
        (86, 'China'),
        (91, 'India'),
        (1, 'United States'),
        (62, 'Indonesia'),
        (55, 'Brazil'),
        (92, 'Pakistan'),
        (880, 'Bangladesh'),
        (234, 'Nigeria'),
        (7, 'Russia'),
        (81, 'Japan'),
    ]

d1 = dict(DIAL_CODES)  ❶
print('d1:', d1.keys())
```

```
d2 = dict(sorted(DIAL_CODES))   ❷
print('d2:', d2.keys())
d3 = dict(sorted(DIAL_CODES, key=lambda x:x[1]))   ❸
print('d3:', d3.keys())
assert d1 == d2 and d2 == d3   ❹
```

❶ `d1`: built from the tuples in descending order of country population.

❷ `d2`: filled with tuples sorted by dial code.

❸ `d3`: loaded with tuples sorted by country name.

❹ The dictionaries compare equal, because they hold the same key:value pairs.

Example 3-18. Output from `dialcodes.py` shows three distinct key orderings.

```
d1: dict_keys([880, 1, 86, 55, 7, 234, 91, 92, 62, 81])
d2: dict_keys([880, 1, 91, 86, 81, 55, 234, 7, 92, 62])
d3: dict_keys([880, 81, 1, 86, 55, 7, 234, 91, 92, 62])
```

#5: Adding items to a `dict` may change the order of existing keys

Whenever you add a new item to a `dict`, the Python interpreter may decide that the hash table of that dictionary needs to grow. This entails building a new, bigger hash table, and adding all current items to the new table. During this process, new (but different) hash collisions may happen, with the result that the keys are likely to be ordered differently in the new hash table. All of this is implementation-dependent, so you cannot reliably predict when it will happen. If you are iterating over the dictionay keys and changing them at the same time, your loop may not scan all the items as expected — not even the items that were already in the dictionary before you added to it.

This is why modifying the contents of a `dict` while iterating through it is a bad idea. If you need to scan and add items to a dictionary, do it in two steps: read the `dict` from start to finish and collect the needed additions in a second `dict`. Then update the first one with it.

In Python 3, the `.keys()`, `.items()` and `.values()` methods return dictionary views, which behave more like sets than the lists returned by these methods in Python 2. Such views are also dynamic: they do not replicate the contents of the `dict`, and they immediately reflect any changes to the `dict`.

We can now apply what we know about hash tables to sets.

How sets work — practical consequences

The set and frozenset types are also implemented with a hash table, except that each bucket holds only a reference to the element (as if it were a key in a dict, but without a value to go with it). In fact, before set was added to the language, we often used dictionaries with dummy values just to perform fast membership tests on the keys.

Everything said in "Practical consequences of how dict works" on page 90 about how the underlying hash table determines the behavior of a dict applies to a set. Without repeating the previous section, we can summarize it for sets with just a few words:

1. Set elements must be hashable objects.
2. Sets have a significant memory overhead.
3. Membership testing is very efficient.
4. Element ordering depends on insertion order.
5. Adding elements to a set may change the order of other elements.

Chapter summary

Dictionaries are a keystone of Python. Beyond the basic dict, the standard library offers handy, ready-to-use specialized mappings like defaultdict, OrderedDict, ChainMap and Counter, all defined in the collections module. The same module also provides the easy to extend UserDict class.

Two powerful methods available in most mappings are setdefault and update. The setdefault method is used to update items holding mutable values, for example, in a dict of list values, to avoid redundant searches for the same key. The update method allows bulk insertion or overwriting of items from any other mapping, from iterables providing (key, value) pairs and from keyword arguments. Mapping constructors also use update internally, allowing instances to be initialized from mappings, iterables or keyword arguments.

A clever hook in the mapping API is the __missing__ method, that lets you customize what happens when a key is not found.

The collections.abc module provides the Mapping and MutableMapping abstract base classes for reference and type checking. The little-known MappingProxyType from the types module creates immutable mappings. There are also ABCs for Set and MutableSet.

The hash table implementation underlying `dict` and `set` is extremely fast. Understanding its logic explains why items are apparently unordered and may even be reordered behind our backs. There is a price to pay for all this speed, and the price is in memory.

Further reading

Chapter 8.3. collections — Container datatypes of the Python Standard Library documentation has examples and practical recipes with several mapping types. The Python source code for that module — `Lib/collections/__init__.py` is a great reference for anyone who wants to create a new mapping type or grok the logic of the existing ones.

Chapter 1 of Python Cookbook, 3rd ed. (O'Reilly, 2013) by David Beazley and Brian K. Jones has 20 handy and insightful recipes with data structures — the majority using `dict` in clever ways.

The book Beautiful Code (O'Reilly) has a chapter titled "Python's Dictionary Implementation: Being All Things to All People" with a detailed explanation of the inner workings of the Python `dict`, written by A. M. Kuchling — a Python core contributor and author of many pages of the official Python docs and how-tos. Also, there are lots of comments in the source code of the CPython `dictobject.c` module. Brandon Craig Rhodes' presentation The Mighty Dictionary is excellent and shows how hash tables work by using lots of slides with… tables!

The rationale for adding sets to the language is documented in PEP 218 — Adding a Built-In Set Object Type. When PEP 218 was approved, no special literal syntax was adopted for sets. The `set` literals where created for Python 3 and backported to Python 2.7, along with `dict` and `set` comprehensions. PEP 274 — Dict Comprehensions is the birth certificate of dictcomps. I could not find a PEP for setcomps; apparently they were adopted because they get along well with their siblings — a jolly good reason.

Soapbox

My friend Geraldo Cohen once remarked that Python is "simple and correct".

The `dict` type is an example of simplicity and correctness. It's highly optimized to do one thing well: retrieve arbitrary keys. It's fast and robust enough to be used all over the Python interpreter itself. If you need predictable ordering, use `OrderedDict`. That is not a requirement in most uses of mappings, so it makes sense to keep the core implementation simple and offer variations in the standard library.

Contrast with PHP, where arrays are described like this in the official PHP Manual:

> An array in PHP is actually an ordered map. A map is a type that associates values to keys. This type is optimized for several different uses; it can be treated as an array, list

> (vector), hash table (an implementation of a map), dictionary, collection, stack, queue, and probably more.

From that description, I don't know what is the real cost of using PHP's `list`/`OrderedDict` hybrid.

The goal of this and the previous chapter in this book was to showcase the Python collection types optimized for particular uses. I made the point that beyond the trusty `list` and `dict` there are specialized alternatives for different use cases.

Before finding Python I had done Web programming using Perl, PHP and JavaScript. I really enjoyed having a literal syntax for mappings in these languages, and I badly miss it whenever I have to use Java or C. A good literal syntax for mappings makes it easy to do configuration, table-driven implementations and to hold data for prototyping and testing. The lack of it pushed the Java community to adopt the verbose and overly complex XML as a data format.

JSON was proposed as "The Fat-Free Alternative to XML" and became a huge success, replacing XML in many contexts. A concise syntax for lists and dictionaries makes an excellent data interchange format.

PHP and Ruby imitated the hash syntax from Perl, using `=>` to link keys to values. JavaScript followed the lead of Python and uses `:`. Of course JSON came from JavaScript, but it also happens to be an almost exact subset of Python syntax. JSON is compatible with Python except for the spelling of the values `true`, `false` and `null`. The syntax everybody now uses for exchanging data is the Python `dict` and `list` syntax.

Simple and correct.

CHAPTER 4
Text versus bytes

> Humans use text. Computers speak bytes[1].
>
> — Esther Nam and Travis Fischer
> *Character encoding and Unicode in Python*

Python 3 introduced a sharp distinction between strings of human text and sequences of raw bytes. Implicit conversion of byte sequences to Unicode text is a thing of the past. This chapter deals with Unicode strings, binary sequences and the encodings used to convert between them.

Depending on your Python programming context, a deeper understanding of Unicode may or may not be of vital importance to you. In the end, most of the issues covered in this chapter do not affect programmers who deal only with ASCII text. But even if that is your case, there is no escaping the `str` versus `byte` divide. As a bonus, you'll find that the specialized binary sequence types provide features that the "all-purpose" Python 2 `str` type does not have.

In this chapter we will visit the following topics:

- characters, code points and byte representations;
- unique features of binary sequences: `bytes`, `bytearray` and `memoryview`;
- codecs for full Unicode and legacy character sets;
- avoiding and dealing with encoding errors;
- best practices when handling text files;
- the default encoding trap and standard I/O issues;
- safe Unicode text comparisons with normalization;

[1] Slyde 12 of PyCon 2014 talk *Character encoding and Unicode in Python* (slides, video).

- utility functions for normalization, case folding and brute-force diacritic removal;
- proper sorting of Unicode text with `locale` and the PyUCA library;
- character metadata in the Unicode database;
- dual-mode APIs that handle `str` and `bytes`;

Let's start with the characters, code points and bytes.

Character issues

The concept of "string" is simple enough: a string is a sequence of characters. The problem lies in the definition of "character".

In 2014 the best definition of "character" we have is a Unicode character. Accordingly, the items you get out of a Python 3 `str` are Unicode characters, just like the items of a `unicode` object in Python 2 — and not the raw bytes you get from a Python 2 `str`.

The Unicode standard explicitly separates the identity of characters from specific byte representations.

- The identity of a character — its *code point* — is a number from 0 to 1,114,111 (base 10), shown in the Unicode standard as 4 to 6 hexadecimal digits with a "U+" prefix. For example, the code point for the letter A is U+0041, the Euro sign is U+20AC and the musical symbol G clef is assigned to code point U+1D11E. About 10% of the valid code points have characters assigned to them in Unicode 6.3, the standard used in Python 3.4.
- The actual bytes that represent a character depend on the *encoding* in use. An encoding is an algorithm that converts code points to byte sequences and vice-versa. The code point for A (U+0041) is encoded as the single byte `\x41` in the UTF-8 encoding, or as the bytes `\x41\x00` in UTF-16LE encoding. As another example, the Euro sign (U+20AC) becomes three bytes in UTF-8 — `\xe2\x82\xac` — but in UTF-16LE it is encoded as two bytes: `\xac\x20`.

Converting from code points to bytes is *encoding*; from bytes to code points is *decoding*. See Example 4-1.

Example 4-1. Encoding and decoding.

```
>>> s = 'café'
>>> len(s)  # ❶
4
>>> b = s.encode('utf8')  # ❷
>>> b
b'caf\xc3\xa9'  # ❸
>>> len(b)  # ❹
5
```

```
>>> b.decode('utf8')  # ❺
'café'
```

❶ The `str` `'café'` has 4 Unicode characters.

❷ Encode `str` to `bytes` using UTF-8 encoding.

❸ `bytes` literals start with a `b` prefix.

❹ `bytes` `b` has five bytes (the code point for "é" is encoded as two bytes in UTF-8).

❺ Decode `bytes` to `str` using UTF-8 encoding.

> If you need a memory aid to distinguish `.decode()` from `.encode()`, convince yourself that byte sequences can be cryptic machine core dumps while Unicode `str` objects are "human" text . Therefore, it makes sense that we **decode** `bytes` to `str` to get human readable text, and we **encode** `str` to `bytes` for storage or transmission.

Although the Python 3 `str` is pretty much the Python 2 `unicode` type with a new name, the Python 3 `bytes` is not simply the old `str` renamed, and there is also the closely related `bytearray` type. So it is worthwhile to take a look at the binary sequence types before advancing to encoding/decoding issues.

Byte essentials

The new binary sequence types are unlike the Python 2 `str` in many regards. The first thing to know is that there are two basic built-in types for binary sequences: the immutable `bytes` type introduced in Python 3 and the mutable `bytearray`, added in Python 2.6[2].

Each item in `bytes` or `bytearray` is an integer from 0 to 255, and not a 1-character string like in the Python 2 `str`. However a slice of a binary sequence always produces a binary sequence of the same type — including slices of length 1. See Example 4-2.

Example 4-2. A five-byte sequence as `bytes` and as `bytearray`.

```
>>> cafe = bytes('café', encoding='utf_8')  ❶
>>> cafe
b'caf\xc3\xa9'
>>> cafe[0]  ❷
99
>>> cafe[:1]  ❸
```

[2] Python 2.6 also introduced `bytes`, but it's just an alias to the `str` type, and does not behave like the Python 3 `bytes` type.

```
b'c'
>>> cafe_arr = bytearray(cafe)
>>> cafe_arr  ❹
bytearray(b'caf\xc3\xa9')
>>> cafe_arr[-1:]  ❺
bytearray(b'\xa9')
```

❶ `bytes` can be built from a `str`, given an encoding.

❷ Each item is an integer in `range(256)`.

❸ Slices of `bytes` are also `bytes` — even slices of a single byte.

❹ There is no literal syntax for `bytearray`: they are shown as `bytearray()` with a `bytes` literal as argument.

❺ A slice of `bytearray` is also a `bytearray`.

> The fact that `my_bytes[0]` retrieves an `int` but `my_bytes[:1]` returns a `bytes` object of length 1 should not be surprising. The only sequence type where `s[0] == s[:1]` is the `str` type. Although practical, this behavior of `str` is exceptional. For every other sequence, `s[i]` returns one item, and `s[i:i+1]` returns a sequence of the same type with the `s[1]` item inside it.

Although binary sequences are really sequences of integers, their literal notation reflects the fact that ASCII text is often embedded in them. Therefore, three different displays are used, depending on each byte value:

- For bytes in the printable ASCII range — from space to ~ — the ASCII character itself is used.
- For bytes corresponding to tab, newline, carriage return and `\`, the escape sequences `\t`, `\n`, `\r` and `\\` are used.
- For every other byte value, an hexadecimal escape sequence is used, e.g. `\x00` is the null byte.

That is why in Example 4-2 you see `b'caf\xc3\xa9'`: the first three bytes `b'caf'` are in the printable ASCII range, the last two are not.

Both `bytes` and `bytearray` support every `str` method except those that do formatting (`format`, `format_map`) and a few others that depend on Unicode data: `casefold`, `isdecimal`, `isidentifier`, `isnumeric`, `isprintable` and `encode`. This means that you can use familiar string methods like `endswith`, `replace`, `strip`, `translate`, `upper` and dozens of others with binary sequences — only using `bytes` and not `str` arguments. In addition, the regular expression functions in the `re` module also work on binary sequences, if the regex is compiled from a binary sequence instead of a `str`. The `%` operator does not work with binary sequences in Python 3.0 to 3.4, but should be supported in version 3.5 according to PEP 461 — Adding % formatting to bytes and bytearray.

Binary sequences have a class method that `str` doesn't have: `fromhex`, which builds a binary sequence by parsing pairs of hex digits optionally separated by spaces:

```
>>> bytes.fromhex('31 4B CE A9')
b'1K\xce\xa9'
```

The other ways of building `bytes` or `bytearray` instances are calling their constructors with:

- a `str` and an `encoding` keyword argument.
- an iterable providing items with values from 0 to 255.
- a single integer, to create a binary sequence of that size initialized with null bytes[3].
- an object that implements the buffer protocol (eg. `bytes`, `bytearray`, `memoryview`, `array.array`); this copies the bytes from the source object to the newly created binary sequence.

Building a binary sequence from a buffer-like object is a low-level operation that may involve type casting. See a demonstration in Example 4-3.

Example 4-3. Initializing `bytes` from the raw data of an `array`.

```
>>> import array
>>> numbers = array.array('h', [-2, -1, 0, 1, 2])  ❶
>>> octets = bytes(numbers)  ❷
>>> octets
b'\xfe\xff\xff\xff\x00\x00\x01\x00\x02\x00'  ❸
```

❶ Typecode `'h'` creates an `array` of short integers (16 bits).

❷ `octets` holds a copy of the bytes that make up `numbers`.

❸ These are the 10 bytes that represent the five short integers.

Creating a `bytes` or `bytearray` object from a any buffer-like source will always copy the bytes. In contrast, `memoryview` objects let you share memory between binary data structures. To extract structured information from binary sequences, the `struct` module is invaluable. We'll see it working along with `bytes` and `memoryview` in the next section.

[3] This signature will be deprecated in Python 3.5 and removed in Python 3.6. See PEP 467 — Minor API improvements for binary sequences.

Structs and memory views

The struct module provides functions to parse packed bytes into a tuple of fields of different types and to perform the opposite conversion, from a tuple into packed bytes. struct is used with bytes, bytearray and memoryview objects.

As we've seen in "Memory views" on page 51, memoryview class does not let you create or store byte sequences, but provides shared memory access to slices of data from other binary sequences, packed arrays and buffers such as PIL images[4], without copying the bytes.

Example 4-4 shows the use of memoryview and struct together to extract the width and height of a GIF image.

Example 4-4. Using memoryview and struct to inspect a GIF image header.

```
>>> import struct
>>> fmt = '<3s3sHH'  # ❶
>>> with open('filter.gif', 'rb') as fp:
...     img = memoryview(fp.read())  # ❷
...
>>> header = img[:10]  # ❸
>>> bytes(header)  # ❹
b'GIF89a+\x02\xe6\x00'
>>> struct.unpack(fmt, header)  # ❺
(b'GIF', b'89a', 555, 230)
>>> del header  # ❻
>>> del img
```

❶ struct format: < little-endian; 3s3s two sequences of 3 bytes; HH two 16-bit integers.

❷ Create memoryview from file contents in memory…

❸ …then another memoryview by slicing the first one; no bytes are copied here.

❹ Convert to bytes for display only; 10 bytes are copied here.

❺ Unpack memoryview into tuple of: type, version, width and height.

❻ Delete references to release the memory associated with the memoryview instances.

Note that slicing a memoryview returns a new memoryview, without copying bytes[5].

[4] PIL is the Python Imaging Library, and Pillow is its most active fork.

[5] Leonardo Rochael — one of the technical reviewers — pointed out that even less byte copying would happen if I used the mmap module to open the image as a memory-mapped file. I will not cover mmap in this book, but if read and change binary files frequently, learning more about mmap — Memory-mapped file support will be very fruitful.

We will not go deeper into memoryview or the struct module in this book, but if you work with binary data, you'll find it worthwhile to study their docs: Built-in Types » Memory Views and struct — Interpret bytes as packed binary data.

After this brief exploration of binary sequence types in Python, let us see how they are converted to/from strings.

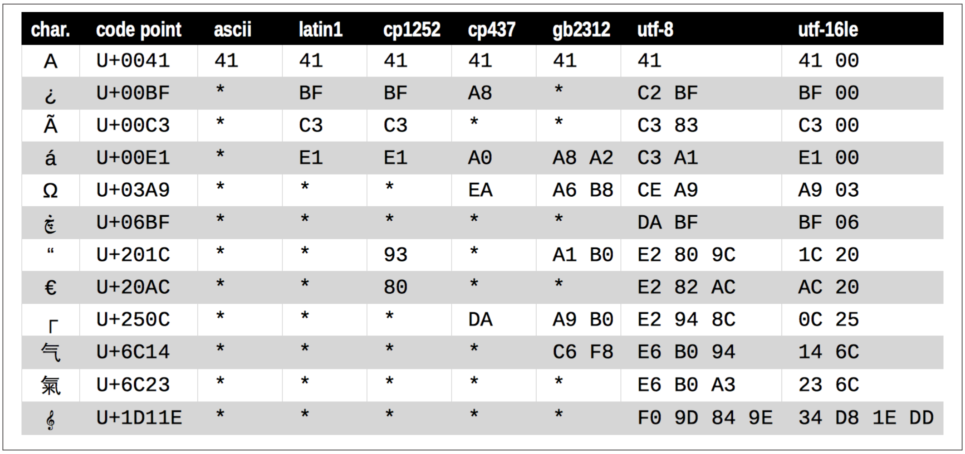
Basic encoders/decoders

The Python distribution bundles more than 100 *codecs* (encoder/decoder) for text to byte conversion and vice-versa. Each codec has a name, like 'utf_8', and often aliases, such as 'utf8', 'utf-8' and 'U8', which you can use as the encoding` argument in functions like open(), str.encode(), bytes.decode() and so on.

Example 4-5. The string "El Niño" encoded with three codecs producing very different byte sequences.

```
>>> for codec in ['latin_1', 'utf_8', 'utf_16']:
...     print(codec, 'El Niño'.encode(codec), sep='\t')
...
latin_1 b'El Ni\xf1o'
utf_8   b'El Ni\xc3\xb1o'
utf_16  b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

Figure 4-1 demonstrates a variety of codecs generating bytes from characters like the letter "A" through the G-clef musical symbol. Note that the last three encodings are variable-length, multi-byte encodings.



char.	code point	ascii	latin1	cp1252	cp437	gb2312	utf-8	utf-16le
A	U+0041	41	41	41	41	41	41	41 00
¿	U+00BF	*	BF	BF	A8	*	C2 BF	BF 00
Ã	U+00C3	*	C3	C3	*	*	C3 83	C3 00
á	U+00E1	*	E1	E1	A0	A8 A2	C3 A1	E1 00
Ω	U+03A9	*	*	*	EA	A6 B8	CE A9	A9 03
ڿ	U+06BF	*	*	*	*	*	DA BF	BF 06
“	U+201C	*	*	93	*	A1 B0	E2 80 9C	1C 20
€	U+20AC	*	*	80	*	*	E2 82 AC	AC 20
┌	U+250C	*	*	*	DA	A9 B0	E2 94 8C	0C 25
气	U+6C14	*	*	*	*	C6 F8	E6 B0 94	14 6C
氣	U+6C23	*	*	*	*	*	E6 B0 A3	23 6C
𝄞	U+1D11E	*	*	*	*	*	F0 9D 84 9E	34 D8 1E DD

*Figure 4-1. Twelve characters, their code points and their byte representation (in hex) in seven different encodings. * means the character cannot be represented in that encoding.*

All those stars `*` in Figure 4-1 make clear that some encodings, like ASCII and even the multi-byte GB2312, cannot represent every Unicode character. The UTF encodings, however, are designed to handle every Unicode code point.

The encodings shown in Figure 4-1 were chosen as a representative sample:

`latin1` *a.k.a.* `iso8859_1`
: Important because it is the basis for other encodings, such as `cp1252` and Unicode itself (note how the `latin1` byte values appear in the `cp1252` bytes and even in the code points).

`cp1252`
: A `latin1` superset by Microsoft, adding useful symbols like curly quotes and the € (euro); some Windows apps call it "ANSI", but it was never a real ANSI standard.

`cp437`
: The original character set of the IBM PC, with box drawing characters. Incompatible with `latin1`, which appeared later.

`gb2312`
: Legacy standard to encode the simplified Chinese ideographs used in mainland China; one of several widely deployed multi-byte encodings for Asian languages.

`utf-8`
: The most common 8-bit encoding on the Web, by far[6]; backward-compatible with ASCII (pure ASCII text is valid UTF-8).

`utf-16le`
: One form of the UTF-16 16-bit encoding scheme; all UTF-16 encodings support code points beyond U+FFFF through escape sequences called "surrogate pairs".

> UTF-16 superseded the original 16-bit Unicode 1.0 encoding — UCS-2 — way back in 1996. UCS-2 is still deployed in many systems, but it only supports code points up to U+FFFF. As of Unicode 6.3, more than 50% of the allocated code points are above U+10000, including the increasingly popular emoji pictographs.

After this overview of common encodings, we now move to handling issues in encoding and decoding operations.

[6] As of September, 2014, W3Techs: Usage of character encodings for websites claims that 81.4% of sites use UTF-8, while Built With: Encoding Usage Statistics estimates 79.4%.

Understanding encode/decode problems

Although there is a generic `UnicodeError` exception, almost always the error reported is more specific: either an `UnicodeEncodeError`, when converting `str` to binary sequences or an `UnicodeDecodeError` when reading binary sequences into `str`. Loading Python modules may also generate `SyntaxError` when the source encoding is unexpected. We'll show how to handle all of these errors in the next sections.

The first thing to note when you get a Unicode error is the exact type of the exception. Is it an `UnicodeEncodeError`, an `UnicodeDecodeError` or some other error (eg. `SyntaxError`) that mentions an encoding problem? To solve the problem you have to understand it first.

Coping with `UnicodeEncodeError`

Most non-UTF codecs handle only a small subset of the Unicode characters. When converting text to bytes, if a character is not defined in the target encoding, `UnicodeEncodeError` will be raised, unless special handling is provided by passing an `errors` argument to the encoding method or function. The behavior of the error handlers is shown in Example 4-6.

Example 4-6. Encoding to `bytes`: success and error handling

```
>>> city = 'São Paulo'
>>> city.encode('utf_8')  ❶
b'S\xc3\xa3o Paulo'
>>> city.encode('utf_16')
b'\xff\xfeS\x00\xe3\x00o\x00 \x00P\x00a\x00u\x00l\x00o\x00'
>>> city.encode('iso8859_1')  ❷
b'S\xe3o Paulo'
>>> city.encode('cp437')  ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/.../lib/python3.4/encodings/cp437.py", line 12, in encode
    return codecs.charmap_encode(input,errors,encoding_map)
UnicodeEncodeError: 'charmap' codec can't encode character '\xe3' in
position 1: character maps to <undefined>
>>> city.encode('cp437', errors='ignore')  ❹
b'So Paulo'
>>> city.encode('cp437', errors='replace')  ❺
b'S?o Paulo'
>>> city.encode('cp437', errors='xmlcharrefreplace')  ❻
b'S&#227;o Paulo'
```

❶ The `'utf_?'` encodings handle any `str`.

❷ `'iso8859_1'` also works for the `'São Paulo'` `str`.

❸ 'cp437' can't encode the 'ã' ("a" with tilde). The default error handler — 'strict' — raises UnicodeEncodeError.

❹ The error='ignore' handler silently skips characters that cannot be encoded; this is usually a very bad idea.

❺ When encoding, error='replace' substitutes unencodable characters with '?'; data is lost, but users will know something is amiss.

❻ 'xmlcharrefreplace' replaces unencodable characters with a XML entity.

The codecs error handling is extensible. You may register extra strings for the errors argument by passing a name and an error handling function to the codecs.register_error function. See the codecs.register_error documentation.

Coping with UnicodeDecodeError

Not every byte holds a valid ASCII character, and not every byte sequence is valid UTF-8 or UTF-16, therefore when you assume one of these encodings while converting a binary sequence to text, you will get a UnicodeDecodeError if unexpected bytes are found.

On the other hand, many legacy 8-bit encodings like 'cp1252', 'iso8859_1', 'koi8_r' are able to decode any stream of bytes, including random noise, without generating errors. Therefore, if your program assumes the wrong 8-bit encoding, it will silently decode garbage.

Garbled characters are known as gremlins or mojibake (⍰⍰⍰⍰ - Japanese for "transformed text")[7].

Example 4-7. Decoding from `str` to `bytes`: success and error handling

```
>>> octets = b'Montr\xe9al'  ❶
>>> octets.decode('cp1252')  ❷
'Montréal'
>>> octets.decode('iso8859_7')  ❸
'Montrιal'
>>> octets.decode('koi8_r')  ❹
'MontrИal'
```

[7] Ironically, the rendered PDF of this book currently shows gremlins where the Japanese characters for "mojibake" should be. The English language Wikipedia article for Mojibake shows the word in Japanese, if you're curious.

```
>>> octets.decode('utf_8')  ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 5:
invalid continuation byte
>>> octets.decode('utf_8', errors='replace')  ❻
'Montr�al'
```

❶ These bytes are the characters for “Montréal” encoded as `latin1`; `'\xe9'` is the byte for “é”.

❷ Decoding with `'cp1252'` (Windows 1252) works because it is a proper superset of `latin1`.

❸ ISO-8859-7 is intended for Greek, so the `'\xe9'` byte is misinterpreted, and no error is issued.

❹ KOI8-R is for Russian. Now `'\xe9'` stands for the Cyrillic letter “И”.

❺ The `'utf_8'` codec detects that `octets` is not valid UTF-8, and raises `UnicodeDecodeError`

❻ Using `'replace'` error handling, the `\xe9` is replaced by “�” (code point U+FFFD), the official Unicode `REPLACEMENT CHARACTER` intended to represent unknown characters.

`SyntaxError` when loading modules with unexpected encoding

UTF-8 is the default source encoding for Python 3, just as ASCII was the default for Python 2 (starting with 2.5). If you load a `.py` module containing non-UTF-8 data and no encoding declaration, you get a message like this:

```
SyntaxError: Non-UTF-8 code starting with '\xe1' in file ola.py on line
  1, but no encoding declared; see http://python.org/dev/peps/pep-0263/
  for details
```

Because UTF-8 is widely deployed in GNU/Linux and OSX systems, a likely scenario is opening a `.py` file created on Windows with `cp1252`. Note that this error happens even in Python for Windows, because the default encoding for Python 3 is UTF-8 across all platforms.

To fix this problem, add a magic `coding` comment at the top of the file:

Example 4-8. `'ola.py'`: “Hello, World!” in Portuguese.

```
# coding: cp1252

print('Olá, Mundo!')
```

Now that Python 3 source code is no longer limited to ASCII and defaults to the excellent UTF-8 encoding, the best "fix" for source code in legacy encodings like `'cp1252'` is to convert them to UTF-8 already, and not bother with the `coding` comments. If your editor does not support UTF-8, it's time to switch.

Non-ASCII names in source code: should you use them?

Python 3 allows non-ASCII identifiers in source code:

```
>>> ação = 'PBR'   # ação = stock
>>> ε = 10**-6     # ε = epsilon
```

Some people dislike the idea. The most common argument to stick with ASCII identifiers is to make it easy for everyone to read and edit code. That argument misses the point: you want your source code to be readable and editable by its intended audience, and that may not be "everyone". If the code belongs to a multi-national corporation or is Open Source and you want contributors from around the World, the identifiers should be in English, and then all you need is ASCII.

But if you are a teacher in Brazil, your students will find it easier to read code that uses Portuguese variable and function names, correctly spelled. And they will have no difficulty typing the cedillas and accented vowels on their localized keyboards.

Now that Python can parse Unicode names and UTF-8 is the default source encoding, I see no point in coding identifiers in Portuguese without accents, as we used to do in Python 2 out of necessity — unless you need the code to run on Python 2 also. If the names are in Portuguese, leaving out the accents won't make the code more readable to anyone.

This is my point of view as a Portuguese-speaking Brazilian, but I believe it applies across borders and cultures: choose the human language that makes the code easier to read by the team, then use the characters needed for correct spelling.

Suppose you have a text file, be it source code or poetry, but you don't know its encoding. How to detect the actual encoding? The next section answers that with a library recommendation.

How to discover the encoding of a byte sequence

Short answer: you can't. You must be told.

Some communication protocols and file formats, like HTTP and XML, contain headers that explicitly tell us how the content is encoded. You can be sure that some byte streams are not ASCII because they contain byte values over 127, and the way UTF-8 and UTF-16 are built also limits the possible byte sequences. But even then, you can never be 100% positive that a binary file is ASCII or UTF-8 just because certain bit patterns are not there.

However, considering that human languages also have their rules and restrictions, once you assume that a stream of bytes is human *plain text* it may be possible to sniff out its encoding using heuristics and statistics. For example, if `b'\x00'` bytes are common, it is probably a 16 or 32-bit encoding, and not an 8-bit scheme, because null characters in plain text are bugs; when the byte sequence `b'\x20\x00'` appears often, it is likely to be the space character (U+0020) in a UTF-16LE encoding, rather than the obscure U +2000 `EN QUAD` character — whatever that is.

That is how the package Chardet — The Universal Character Encoding Detector works to identify one of 30 supported encodings. Chardet is a Python library that you can use in your programs, but also includes a command-line utility, `chardetect`. Here is what it reports on the source file for this chapter:

```
$ chardetect 04-text-byte.asciidoc
04-text-byte.asciidoc: utf-8 with confidence 0.99
```

Although binary sequences of encoded text usually don't carry explicit hints of their encoding, the UTF formats may prepend a byte order mark to the textual content. That is explained next.

BOM: a useful gremlin

In Example 4-5 you may have noticed a couple of extra bytes at the beginning of an UTF-16 encoded sequence. Here they are again:

```
>>> u16 = 'El Niño'.encode('utf_16')
>>> u16
b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

The bytes are: `b'\xff\xfe'`. That is a *BOM* — byte-order mark — denoting the "little-endian" byte ordering of the Intel CPU where the encoding was performed.

On a little-endian machine, for each code point the least significant byte comes first: the letter `'E'`, code point U+0045 (decimal 69), is encoded in byte offsets 2 and 3 as 69 and 0:

```
>>> list(u16)
[255, 254, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
```

On a big-endian CPU, the encoding would be reversed, `'E'` would be encoded as 0 and 69.

To avoid confusion, the UTF-16 encoding prepends the text to be encoded with the special character `ZERO WIDTH NO-BREAK SPACE` (U+FEFF) which is invisible. On a little-endian system, that is encoded as `b'\xff\xfe'` (decimal 255, 254). Because, by design, there is no U+FFFE character, the byte sequence `b'\xff\xfe'` must mean the `ZERO WIDTH NO-BREAK SPACE` on a little-endian encoding, so the codec knows which byte ordering to use.

There is a variant of UTF-16 — UTF-16LE — that is explicitly little endian, and another one explicitly big-endian, UTF-16BE. If you use them, a BOM is not generated:

```
>>> u16le = 'El Niño'.encode('utf_16le')
>>> list(u16le)
[69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
>>> u16be = 'El Niño'.encode('utf_16be')
>>> list(u16be)
[0, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111]
```

If present, the BOM is supposed to be filtered by the UTF-16 codec, so that you only get the actual text contents of the file without the leading `ZERO WIDTH NO-BREAK SPACE`. The standard says that if a file is UTF-16 and has no BOM, it should be assumed to be UTF-16BE (big-endian). However, the Intel x86 architecture is little-endian, so there is plenty of little-endian UTF-16 with no BOM in the wild.

This whole issue of endianness only affects encodings that use words of more than one byte, like UTF-16 and UTF-32. One big advantage of UTF-8 is that it produces the same byte sequence regardless of machine endianness, so no BOM is needed. Nevertheless, some Windows applications (notably Notepad) add the BOM to UTF-8 files anyway — and Excel depends on the BOM to detect an UTF-8 file, otherwise it assumes the content is encoded with a Windows codepage. The character U+FEFF encoded in UTF-8 is the three-byte sequence `b'\xef\xbb\xbf'`. So if a file starts with those three bytes, it is likely to be a UTF-8 file with a BOM. However, Python does not automatically assume a file is UTF-8 just because it starts with `b'\xef\xbb\xbf'`.

We now move to handling text files in Python 3.

Handling text files

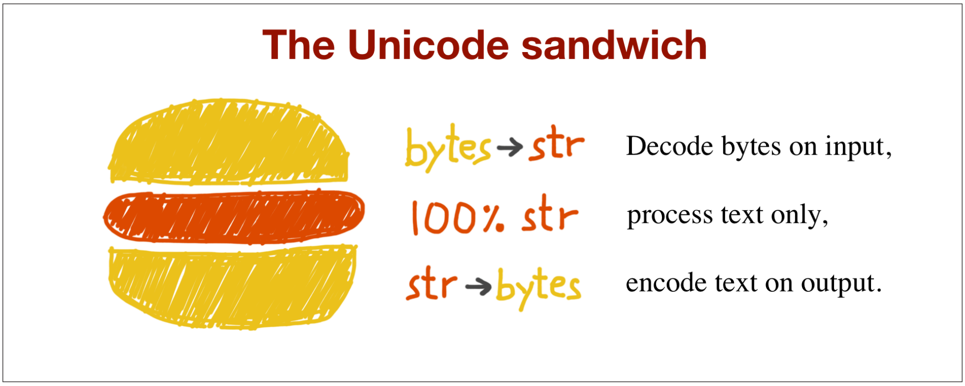


Figure 4-2. Unicode sandwich: current best practice for text processing.

The best practice for handling text is the "Unicode sandwich" (Figure 4-2)[8]. This means that `bytes` should be decoded to `str` as early as possible on input, e.g. when opening a file for reading. The "meat" of the sandwich is the business logic of your program, where text handling is done exclusively on `str` objects. You should never be encoding or decoding in the middle of other processing. On output, the `str` are encoded to `bytes` as late as possible. Most Web frameworks work like that, and we rarely touch `bytes` when using them. In Django, for example, your views should output Unicode `str`; Django itself takes care of encoding the response to `bytes`, using UTF-8 by default.

Python 3 makes it easier to follow the advice of the Unicode sandwich, because the `open` built-in does the necessary decoding when reading and encoding when writing files in text mode, so all you get from `my_file.read()` and pass to `my_file.write(text)` are `str` objects[9].

Therefore, using text files is simple. But if you rely on default encodings you will get bitten.

Consider the console session in Example 4-9. Can you spot the bug?

Example 4-9. A platform encoding issue. If you try this on your machine, you may or may not see the problem.

```
>>> open('cafe.txt', 'w', encoding='utf_8').write('café')
4
>>> open('cafe.txt').read()
'cafÃ©'
```

[8] I first saw the term "Unicode sandwich" in Ned Batchelder's excellent *Pragmatic Unicode* talk at US PyCon 2012

[9] Python 2.6 or 2.7 users have to use `io.open()` the get automatic decoding/encoding when reading/writing.

The bug: I specified UTF-8 encoding when writing the file but failed to do so when reading it, so Python assumed the system default encoding — Windows 1252 — and the trailing bytes in the file were decoded as characters `'Ã©'` instead of `'é'`.

I ran Example 4-9 on a Windows 7 machine. The same statements running on recent GNU/Linux or Mac OSX work perfectly well because their default encoding is UTF-8, giving the false impression that everything is fine. If the encoding argument was omitted when opening the file to write, the locale default encoding would be used, and we'd read the file correctly using the same encoding. But then this script would generate files with different byte contents depending on the platform or even depending on locale settings in the same platform, creating compatibility problems.

> Code that has to run on multiple machines or on multiple occasions should never depend on encoding defaults. Always pass an explicit `encoding=` argument when opening text files, because the default may change from one machine to the next, or from one day to the next.

A curious detail in Example 4-9 is that the `write` function in the first statement reports that 4 characters were written, but in the next line 5 characters are read. Example 4-10 is an extended version of Example 4-9, explaining that and other details.

Example 4-10. Closer inspection of Example 4-9 running on Windows reveals the bug and how to fix it.

```
>>> fp = open('cafe.txt', 'w', encoding='utf_8')
>>> fp  ❶
<_io.TextIOWrapper name='cafe.txt' mode='w' encoding='utf_8'>
>>> fp.write('café')
4  ❷
>>> fp.close()
>>> import os
>>> os.stat('cafe.txt').st_size
5  ❸
>>> fp2 = open('cafe.txt')
>>> fp2  ❹
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='cp1252'>
>>> fp2.encoding  ❺
'cp1252'
>>> fp2.read()
'cafÃ©'  ❻
>>> fp3 = open('cafe.txt', encoding='utf_8')  ❼
>>> fp3
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='utf_8'>
>>> fp3.read()
'café'  ❽
>>> fp4 = open('cafe.txt', 'rb')  ❾
>>> fp4
```

```
<_io.BufferedReader name='cafe.txt'>  ❿
>>> fp4.read()  ⓫
b'caf\xc3\xa9'
```

❶ By default, open operates in text mode and returns a TextIOWrapper object.

❷ The write method on a TextIOWrapper returns the number of Unicode characters written.

❸ os.stat reports that the file holds 5 bytes; UTF-8 encodes 'é' as two bytes, 0xc3 and 0xa9.

❹ Opening a text file with no explicit encoding returns a TextIOWrapper with the encoding set to a default from the locale.

❺ A TextIOWrapper object has an encoding attribute that you can inspect: cp1252 in this case.

❻ In the Windows cp1252 encoding, the byte 0xc3 is an "Ã" (A with tilde) and 0xa9 is the copyright sign.

❼ Opening the same file with the correct encoding.

❽ The expected result: the same 4 Unicode characters for 'café'.

❾ The 'rb' flag opens a file for reading in binary mode.

❿ The returned object is a BufferedReader and not a TextIOWrapper.

⓫ Reading that returns bytes, as expected.

> Do not open text files in binary mode unless you need to analyze the file contents to determine the encoding — even then, you should be using Chardet instead of reinventing the wheel (see "How to discover the encoding of a byte sequence" on page 108). Ordinary code should only use binary mode to open binary files, like raster images.

The problem in Example 4-10 has to do with relying on a default setting while opening a text file. There are several sources for such defaults, as the next section shows.

Encoding defaults: a madhouse

Several settings affect the encoding defaults for I/O in Python. See the default_encodings.py script in Example 4-11.

Example 4-11. Exploring encoding defaults

```
import sys, locale

expressions = """
        locale.getpreferredencoding()
        type(my_file)
        my_file.encoding
        sys.stdout.isatty()
        sys.stdout.encoding
        sys.stdin.isatty()
        sys.stdin.encoding
        sys.stderr.isatty()
        sys.stderr.encoding
        sys.getdefaultencoding()
        sys.getfilesystemencoding()
    """

my_file = open('dummy', 'w')

for expression in expressions.split():
    value = eval(expression)
    print(expression.rjust(30), '->', repr(value))
```

The output of Example 4-11 on GNU/Linux (Ubuntu 14.04) and OSX (Mavericks 10.9) is identical, showing that `UTF-8` is used everywhere in these systems:

```
    $ python3 default_encodings.py
     locale.getpreferredencoding() -> 'UTF-8'
                     type(my_file) -> <class '_io.TextIOWrapper'>
                  my_file.encoding -> 'UTF-8'
               sys.stdout.isatty() -> True
               sys.stdout.encoding -> 'UTF-8'
                sys.stdin.isatty() -> True
                sys.stdin.encoding -> 'UTF-8'
               sys.stderr.isatty() -> True
               sys.stderr.encoding -> 'UTF-8'
          sys.getdefaultencoding() -> 'utf-8'
       sys.getfilesystemencoding() -> 'utf-8'
```

On Windows, however, the output is Example 4-12.

Example 4-12. Default encodings on Windows 7 (SP 1) `cmd.exe` localized for Brazil; PowerShell gives same result.

```
Z:\>chcp  ❶
Página de código ativa: 850
Z:\>python default_encodings.py  ❷
 locale.getpreferredencoding() -> 'cp1252'   ❸
                 type(my_file) -> <class '_io.TextIOWrapper'>
              my_file.encoding -> 'cp1252'   ❹
           sys.stdout.isatty() -> True       ❺
           sys.stdout.encoding -> 'cp850'    ❻
            sys.stdin.isatty() -> True
            sys.stdin.encoding -> 'cp850'
           sys.stderr.isatty() -> True
           sys.stderr.encoding -> 'cp850'
      sys.getdefaultencoding() -> 'utf-8'
   sys.getfilesystemencoding() -> 'mbcs'
```

❶ `chcp` shows the active codepage for the console: 850.

❷ Running `default_encodings.py` with output to console.

❸ `locale.getpreferredencoding()` is the most important setting.

❹ Text files use `locale.getpreferredencoding()` by default.

❺ The output is going to the console, so `sys.stdout.isatty()` is `True`.

❻ Therefore, `sys.stdout.encoding` is the same as the console encoding.

If the output is redirected to a file, like this:

```
Z:\>python default_encodings.py > encodings.log
```

Then the value of `sys.stdout.isatty()` becomes `False`, and `sys.stdout.encoding` is set by `locale.getpreferredencoding()`, `'cp1252'` in that machine.

Note that there are 4 different encodings in Example 4-12:

- If you omit the `encoding` argument when opening a file, the default is given by `locale.getpreferredencoding()` (`'cp1252'` in Example 4-12).
- The encoding of `sys.stdout/stdin/stderr` is given by the *https://docs.python.org/3/using/cmdline.html#envvar-PYTHONIOENCODING* [`PYTHONIOENCODING`] environment variable, if present, otherwise it is either inherited from the console or defined by `locale.getpreferredencoding()` if the output/input is redirected to/from a file.
- `sys.getdefaultencoding()` is used internally by Python to convert binary data to/from `str`; this happens less often in Python 3, but still happens[10]. Changing this setting is not supported[11].
- `sys.getfilesystemencoding()` is used to encode/decode file names (not file contents). It is used when `open()` gets a `str` argument for the file name; if the file name is given as a `bytes` argument, it is passed unchanged to the OS API. The Python Unicode HOWTO says: "on Windows, Python uses the name `mbcs` to refer to whatever the currently configured encoding is." The acronym MBCS stands for Multi Byte Character Set, which for Microsoft are the legacy variable-width encodings

[10] While researching this subject I did not find a list of situations when Python 3 internally converts `bytes` to `str`. Python core developer Antoine Pitrou says on the `comp.python.devel` list that CPython internal functions that depend on such conversions "don't get a lot of use in py3k" (*http://article.gmane.org/gmane.comp.python.devel/110036*)

[11] The Python 2 `sys.setdefaultencoding` function was misused and is no longer documented in Python 3. It was intended for use by the core developers when the internal default encoding of Python was still undecided. In the same `comp.python.devel` thread, Marc-André Lemburg states that the `sys.setdefaultencoding` must never be called by user code and the only values supported by CPython are `'ascii'` in Python 2 and `'utf-8'` in Python 3 (see *http://article.gmane.org/gmane.comp.python.devel/109916*).

like `gb2312` or `Shift_JIS`, but not UTF-8footnote:[On this topic, a useful answer on StackOverflow is Difference between MBCS and UTF-8 on Windows..

On GNU/Linux and OSX all of these encodings are set to UTF-8 by default, and have been for several years, so I/O handles all Unicode characters. On Windows, not only are different encodings used in the same system, but they are usually codepages like `'cp850'` or `'cp1252'` that support only ASCII with 127 additional characters that are not the same from one encoding to the other. Therefore, Windows users are far more likely to face encoding errors unless they are extra careful.

To summarize, the most important encoding setting is that returned by `locale.getpreferredencoding()`: it is the default for opening text files and for `sys.stdout/stdin/stderr` when they are redirected to files. However, the documentation reads (in part):

> `locale.getpreferredencoding(do_setlocale=True)`
>
> Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess. […]

Therefore, the best advice about encoding defaults is: do not rely on them.

If you follow the advice of the Unicode sandwich and always are explicit about the encodings in your programs, you will avoid a lot of pain. Unfortunately Unicode is painful even if you get your `bytes` correctly converted to `str`. The next two sections cover subjects that are simple in ASCII-land, but get quite complex on planet Unicode: text normalization — i.e. converting text to a uniform representation for comparisons — and sorting.

Normalizing Unicode for saner comparisons

String comparisons are complicated by the fact that Unicode has combining characters: diacritics and other marks that attach to the preceding character, appearing as one when printed.

For example, the word “café” may be composed in two ways, using 4 or 5 code points, but the result looks exactly the same:

```
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> len(s1), len(s2)
(4, 5)
>>> s1 == s2
False
```

The code point U+0301 is the `COMBINING ACUTE ACCENT`. Using it after "e" renders "é". In the Unicode standard, sequences like `'é'` and `'e\u0301'` are called "canonical equivalents", and applications are supposed to treat them as the same. But Python sees two different sequences of code points, and considers them not equal.

The solution is to use Unicode normalization, provided by the `unicodedata.normalize` function. The first argument to that function is one of four strings: `'NFC'`, `'NFD'`, `'NFKC'` and `'NFKD'`. Let's start with the first two.

NFC (Normalization Form C) composes the code points to produce the shortest equivalent string, while NFD decomposes, expanding composed characters into base characters and separate combining characters. Both of these normalizations make comparisons work as expected:

```
>>> from unicodedata import normalize
>>> s1 = 'café'  # composed "e" with acute accent
>>> s2 = 'cafe\u0301'  # decomposed "e" and acute accent
>>> len(s1), len(s2)
(4, 5)
>>> len(normalize('NFC', s1)), len(normalize('NFC', s2))
(4, 4)
>>> len(normalize('NFD', s1)), len(normalize('NFD', s2))
(5, 5)
>>> normalize('NFC', s1) == normalize('NFC', s2)
True
>>> normalize('NFD', s1) == normalize('NFD', s2)
True
```

Western keyboards usually generate composed characters, so text typed by users will be in NFC by default. But to be safe it may be good to sanitize strings with `normalize('NFC', user_text)` before saving. NFC is also the normalization form recommended by the W3C in Character Model for the World Wide Web: String Matching and Searching.

Some single characters are normalized by NFC into another single character. The symbol for the ohm Ω unit of electrical resistance is normalized to the Greek uppercase omega. They are visually identical, but they compare unequal so it is essential to normalize to avoid surprises:

```
>>> from unicodedata import normalize, name
>>> ohm = '\u2126'
>>> name(ohm)
'OHM SIGN'
>>> ohm_c = normalize('NFC', ohm)
>>> name(ohm_c)
'GREEK CAPITAL LETTER OMEGA'
>>> ohm == ohm_c
False
>>> normalize('NFC', ohm) == normalize('NFC', ohm_c)
True
```

The letter K in the acronym for the other two normalization forms — NFKC and NFKD — stands for "compatibility". These are stronger forms of normalization, affecting the so called "compatibility characters". Although one goal of Unicode is to have a single "canonical" code point for each character, some characters appear more than once for compatibility with preexisting standards. For example, the micro sign, `'µ'` (`U+00B5`) was added to Unicode to support round-trip conversion to `latin1`, even though the same character is part of the Greek alphabet with code point is `U+03BC` (`GREEK SMALL LETTER MU`). So, the micro sign is considered a "compatibility character".

In the NFKC and NFKD forms, each compatibility character is replaced by a "compatibility decomposition" of one or more characters that are considered a "preferred" representation, even if there is some formatting loss — ideally, the formatting should be the responsibility of external markup, not part of Unicode. To exemplify, the compatibility decomposition of the one half fraction `'½'` (`U+00BD`) is the sequence of three characters `'1/2'`, and the compatibility decomposition of the micro sign `'µ'` (`U+00B5`) is the lowercase mu `'μ'` (`U+03BC`)[12].

Here is how the NFKC works in practice:

```
>>> from unicodedata import normalize, name
>>> half = '½'
>>> normalize('NFKC', half)
'1/2'
>>> four_squared = '4²'
>>> normalize('NFKC', four_squared)
'42'
>>> micro = 'µ'
>>> micro_kc = normalize('NFKC', micro)
>>> micro, micro_kc
('µ', 'μ')
>>> ord(micro), ord(micro_kc)
(181, 956)
>>> name(micro), name(micro_kc)
('MICRO SIGN', 'GREEK SMALL LETTER MU')
```

Although `'1/2'` is a reasonable substitute for `'½'`, and the micro sign is really a lowercase Greek mu, converting `'4²'` to `'42'` changes the meaning[13]. An application could store `'4²'` as `'4<sup>2</sup>'`, but the `normalize` function knows nothing about formatting. Therefore, NFKC or NFKD may lose or distort information, but they can produce

[12] Curiously, the micro sign is considered a "compatibility character" but the ohm symbol is not. The end result is that NFC doesn't touch the micro sign but changes the ohm symbol to capital omega, while NFKC and NFKD change both the ohm and the micro into other characters.

[13] This could lead some to believe that 16 is The Answer to the Ultimate Question of Life, The Universe, and Everything.

convenient intermediate representations for searching and indexing: users may be pleased that a search for '1/2 inch' also finds documents containing '½ inch'.

NFKC and NFKD normalization should be applied with care and only in special cases — e.g. search and indexing — and not for permanent storage, as these transformations cause data loss.

When preparing text for searching or indexing, another operation is useful: case folding, our next subject.

Case folding

Case folding is essentially converting all text to lowercase, with some additional transformations. It is supported by the str.casefold() method (new in Python 3.3).

For any string s containing only latin1 characters, s.casefold() produces the same result as s.lower(), with only two exceptions: the micro sign 'µ' is changed to the Greek lower case mu (which looks the same in most fonts) and the German Eszett or "sharp s" (ß) becomes "ss".

```
>>> micro = 'µ'
>>> name(micro)
'MICRO SIGN'
>>> micro_cf = micro.casefold()
>>> name(micro_cf)
'GREEK SMALL LETTER MU'
>>> micro, micro_cf
('µ', 'μ')
>>> eszett = 'ß'
>>> name(eszett)
'LATIN SMALL LETTER SHARP S'
>>> eszett_cf = eszett.casefold()
>>> eszett, eszett_cf
('ß', 'ss')
```

As of Python 3.4 there are 116 code points for which str.casefold() and str.lower() return different results. That's 0.11% of a total of 110,122 named characters in Unicode 6.3.

As usual with anything related to Unicode, case folding is a complicated issue with plenty of linguistic special cases, but the Python core team made an effort to provide a solution that hopefully works for most users.

In the next couple of sections, we'll put our normalization knowledge to use developing utility functions.

Utility functions for normalized text matching

As we've seen, NFC and NFD are safe to use and allow sensible comparisons between Unicode strings. NFC is the best normalized form for most applications. `str.casefold()` is the way to go for case-insensitive comparisons.

If you work with text in many languages, a pair of functions like `nfc_equal` and `fold_equal` in Example 4-13 are useful additions to your toolbox.

Example 4-13. `normeq.py`: normalized Unicode string comparison.

```
"""
Utility functions for normalized Unicode string comparison.

Using Normal Form C, case sensitive:

    >>> s1 = 'café'
    >>> s2 = 'cafe\u0301'
    >>> s1 == s2
    False
    >>> nfc_equal(s1, s2)
    True
    >>> nfc_equal('A', 'a')
    False

Using Normal Form C with case folding:

    >>> s3 = 'Straße'
    >>> s4 = 'strasse'
    >>> s3 == s4
    False
    >>> nfc_equal(s3, s4)
    False
    >>> fold_equal(s3, s4)
    True
    >>> fold_equal(s1, s2)
    True
    >>> fold_equal('A', 'a')
    True

"""

from unicodedata import normalize

def nfc_equal(str1, str2):
    return normalize('NFC', str1) == normalize('NFC', str2)

def fold_equal(str1, str2):
    return (normalize('NFC', str1).casefold() ==
            normalize('NFC', str2).casefold())
```

Beyond Unicode normalization and case folding — both part of the Unicode standard — sometimes it makes sense to apply deeper transformations, like changing `'café'` into `'cafe'`. We'll see when and how in the next section.

Extreme "normalization": taking out diacritics

The Google Search secret sauce involves many tricks, but one of them apparently is ignoring diacritics (e.g. accents, cedillas etc.), at least in some contexts. Removing diacritics is not a proper form of normalization because if often changes the meaning of words and may produce false positives when searching. But it helps coping with some facts of life: people sometimes are lazy or ignorant about the correct use of diacritics, and spelling rules change over time, meaning that accents come and go in living languages.

Outside of searching, getting rid of diacritics also makes for more readable URLs, at least in Latin based languages. Take a look at the URL for the English language Wikipedia article about the city of São Paulo:

```
http://en.wikipedia.org/wiki/S%C3%A3o_Paulo
```

The *%C3%A3* part is the URL-escaped, UTF-8 rendering of the single letter "ã" ("a" with tilde). The following is much friendlier, even if it is not the right spelling:

```
http://en.wikipedia.org/wiki/Sao_Paulo
```

To remove all diacritics from a `str`, you can use a function like Example 4-14.

Example 4-14. Function to remove all combining marks (module `sanitize.py`)

```
import unicodedata
import string


def shave_marks(txt):
    """Remove all diacritic marks"""
    norm_txt = unicodedata.normalize('NFD', txt)  ❶
    shaved = ''.join(c for c in norm_txt
                     if not unicodedata.combining(c))  ❷
    return unicodedata.normalize('NFC', shaved)  ❸
```

❶ Decompose all characters into base characters and combining marks.

❷ Filter out all combining marks.

❸ Recompose all characters.

Example 4-15 shows a couple of uses of `shave_marks`.

Example 4-15. Two examples using `shave_marks` from Example 4-14.

```
>>> order = '“Herr Voß: • ½ cup of Œtker™ caffè latte • bowl of açaí.”'
>>> shave_marks(order)
'“Herr Voß: • ½ cup of Œtker™ caffe latte • bowl of acai.”'  ❶
>>> Greek = 'Ζέφυρος, Zéfiro'
>>> shave_marks(Greek)
'Ζεφυρος, Zefiro'  ❷
```

❶ Only the letters “è”, “ç” and “í” were replaced.

❷ Both “έ” and “é” were replaced.

The function `shave_marks` from Example 4-14 works all right, but maybe it goes too far. Often the reason to remove diacritics is to change Latin text to pure ASCII, but `shave_marks` also changes non-Latin characters — like Greek letters — which will never become ASCII just by losing their accents. So it makes sense to analyze each base character and to remove attached marks only if the base character is a letter from the Latin alphabet. This is what Example 4-16 does.

Example 4-16. Function to remove combining marks from Latin characters. `import` statements are omitted as this is part of the `sanitize.py` module from Example 4-14.

```
def shave_marks_latin(txt):
    """Remove all diacritic marks from Latin base characters"""
    norm_txt = unicodedata.normalize('NFD', txt)  ❶
    latin_base = False
    keepers = []
    for c in norm_txt:
        if unicodedata.combining(c) and latin_base:   ❷
            continue  # ignore diacritic on Latin base char
        keepers.append(c)                             ❸
        # if it isn't combining char, it's a new base char
        if not unicodedata.combining(c):              ❹
            latin_base = c in string.ascii_letters
    shaved = ''.join(keepers)
    return unicodedata.normalize('NFC', shaved)   ❺
```

❶ Decompose all characters into base characters and combining marks.

❷ Skip over combining marks when base character is Latin.

❸ Otherwise, keep current character.

❹ Detect new base character and determine if it’s Latin.

❺ Recompose all characters.

An even more radical step would be to replace common symbols in Western texts, like curly quotes, em-dashes, bullets etc. into `ASCII` equivalents. This is what the function `asciize` does in Example 4-17.

Example 4-17. Transform some Western typographical symbols into `ASCII`. This snippet is also part of `sanitize.py` from Example 4-14.

```
single_map = str.maketrans("""‚ƒ„†ˆ‹‘’“”•–—˜›""",   ❶
                           """'f"*^<''""---~>""")

multi_map = str.maketrans({   ❷
    '€': '<euro>',
    '…': '...',
    'Œ': 'OE',
    '™': '(TM)',
    'œ': 'oe',
    '‰': '<per mille>',
    '‡': '**',
})

multi_map.update(single_map)   ❸


def dewinize(txt):
    """Replace Win1252 symbols with ASCII chars or sequences"""
    return txt.translate(multi_map)   ❹


def asciize(txt):
    no_marks = shave_marks_latin(dewinize(txt))     ❺
    no_marks = no_marks.replace('ß', 'ss')          ❻
    return unicodedata.normalize('NFKC', no_marks)  ❼
```

❶ Build mapping table for char to char replacement.

❷ Build mapping table for char to string replacement.

❸ Merge mapping tables.

❹ `dewinize` does not affect `ASCII` or `latin1` text, only the Microsoft additions in to `latin1` in `cp1252`.

❺ Apply `dewinize` and remove diacritical marks.

❻ Replace the Eszett with "ss" (we are not using case fold here because we want to preserve the case).

❼ Apply NFKC normalization to compose characters with their compatibility code points.

Example 4-18 shows `asciize` in use.

Example 4-18. Two examples using `asciize` from Example 4-17.

```
>>> order = '“Herr Voß: • ½ cup of Œtker™ caffè latte • bowl of açaí.”'
>>> dewinize(order)
'"Herr Voß: - ½ cup of OEtker(TM) caffè latte - bowl of açaí."'   ❶
```

```
>>> asciize(order)
'"Herr Voss: - 1/2 cup of OEtker(TM) caffe latte - bowl of acai."'  ❷
```

❶ `dewinize` replaces curly quotes, bullets, and ™ (trade mark symbol).

❷ `asciize` applies `dewinize`, drops diacritics and replaces the `'ß'`.

Different languages have their own rules for removing diacritics. For example, Germans change the `'ü'` into `'ue'`. Our `asciize` function is not as refined, so it may or not be suitable for your language. It works acceptably for Portuguese, though.

To summarize, the functions in `sanitize.py` go way beyond standard normalization and perform deep surgery on the text, with a good chance of changing its meaning. Only you can decide whether to go so far, knowing the target language, your users and how the transformed text will be used.

This wraps up our discussion of normalizing Unicode text.

The next Unicode matter to sort out is… sorting.

Sorting Unicode text

Python sorts sequences of any type by comparing the items in each sequence one by one. For strings, this means comparing the code points. Unfortunately, this produces unacceptable results for anyone who uses non-ASCII characters.

Consider sorting a list of fruits grown in Brazil:

```
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted(fruits)
['acerola', 'atemoia', 'açaí', 'caju', 'cajá']
```

Sorting rules vary for different locales, but in Portuguese and many languages that use the Latin alphabet, accents and cedillas rarely make a difference when sorting[14]. So "cajá" is sorted as "caja", and must come before "caju".

The sorted `fruits` list should be:

```
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

The standard way to sort non-ASCII text in Python is to use the `locale.strxfrm` function which, according to the `locale` module docs, "transforms a string to one that can be used in locale-aware comparisons".

[14] Diacritics affect sorting only in the rare case when they are the only difference between two words — in that case the word with a diacritic is sorted after the plain word.

To enable `locale.strxfrm` you must first set a suitable locale for your application, and pray that the OS supports it. On GNU/Linux (Ubuntu 14.04) with the `pt_BR` locale, the sequence of commands in Example 4-19 works:

Example 4-19. Using the `locale.strxfrm` function as sort key.

```
>>> import locale
>>> locale.setlocale(locale.LC_COLLATE, 'pt_BR.UTF-8')
'pt_BR.UTF-8'
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted_fruits = sorted(fruits, key=locale.strxfrm)
>>> sorted_fruits
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

So you need to call `setlocale(LC_COLLATE, «your_locale»)` before using `locale.strxfrm` as the key when sorting.

There are a few caveats, though:

- Because locale settings are global, calling `setlocale` in a library is not recommended. Your application or framework should set the locale when the process starts, and should not change it afterwards.
- The locale must be installed on the OS, otherwise `setlocale` raises a `locale.Error: unsupported locale setting` exception.
- You must know how to spell the locale name. They are pretty much standardized in the Unix derivatives as `'language_code.encoding'` but on Windows the syntax is more complicated: `Language Name-Language Variant_Region Name.code page>`. Note that the Language Name, Language Variant and Region Name parts can have spaces inside them, but the parts after the first are prefixed with special different characters: an hyphen, an underline character and a dot. All parts seem to be optional except the language name. For example: `English_United States.850` means Language Name "English", region "United States" and codepage "850". The language and region names Windows understands are listed in the MSDN article Language Identifier Constants and Strings, while Code Page Identifiers lists the numbers for the last part[15].
- The locale must be correctly implemented by the makers of the OS. I was successful on Ubuntu 14.04, but not on OSX (Mavericks 10.9). On two different Macs, the call `setlocale(LC_COLLATE, 'pt_BR.UTF-8')` returns the string `'pt_BR.UTF-8'` with no complaints. But `sorted(fruits, key=locale.strxfrm)` produced the same

[15] Thanks to Leonardo Rochael who went beyond his duties as tech reviewer and researched these Windows details, even though he is a GNU/Linux user himself.

incorrect result as sorted(fruits) did. I also tried the fr_FR, es_ES and de_DE locales on OSX, but locale.strxfrm never did its job[16].

So the standard library solution to internationalized sorting works, but seems to be well supported only on GNU/Linux (perhaps also on Windows, if you are an expert). Even then, it depends on locale settings, creating deployment headaches.

Fortunately there is a simpler solution: the PyUCA library, available on *PyPI*.

Sorting with the Unicode Collation Algorithm

James Tauber, prolific Django contributor, must have felt the pain and created PyUCA, a pure-Python implementation of UCA — the Unicode Collation Algorithm. Example 4-20 shows how easy it is to use.

Example 4-20. Using the pyuca.Collator.sort_key method.

```
>>> import pyuca
>>> coll = pyuca.Collator()
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted_fruits = sorted(fruits, key=coll.sort_key)
>>> sorted_fruits
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

This is friendly and just works. I tested it on GNU/Linux, OSX and Windows. Only Python 3.X is supported at this time.

PyUCA does not take the locale into account. If you need to customize the sorting, you can provide the path to a custom collation table to the Collator() constructor. Out of the box, it uses allkeys.txt which is bundled with the project. That's just a copy of the Default Unicode Collation Element Table from Unicode 6.3.0.

By the way, that table is one of the many that comprise the Unicode database, our next subject.

The Unicode database

The Unicode standard provides an entire database — in the form of numerous structured text files — that includes not only the table mapping code points to character names, but also lot of metadata about the individual characters and how they are related. For example, the Unicode database records whether a character is printable, is a letter, is a decimal digit or is some other numeric symbol. That's how the str methods isi

[16] Again, I could not find a solution but did find other people reporting the same problem. Alex Martelli, one of the tech reviewers, had no problem using setlocale and locale.strxfrm on his Mac with OSX 10.9. In summary: your mileage may vary.

dentifier, isprintable, isdecimal and isnumeric work. str.casefold also uses information from a Unicode table.

The unicodedata module has functions that return character metadata, for instance, its official name in the standard, whether it is a combining character (e.g. diacritic like a combining tilde) and the numeric value of the symbol for humans (not its code point). Example 4-21 shows the use of unicodedata.name() unicodedata.numeric() along with the .isdecimal() and .isnumeric() methods of str.

Example 4-21. Demo of Unicode database numerical character metadata. Callouts describe each column in the output.

```
import unicodedata
import re

re_digit = re.compile(r'\d')

sample = '1\xbc\xb2\u0969\u136b\u216b\u2466\u2480\u3285'

for char in sample:
    print('U+%04x' % ord(char),                       ❶
          char.center(6),                             ❷
          're_dig' if re_digit.match(char) else '-',  ❸
          'isdig' if char.isdigit() else '-',         ❹
          'isnum' if char.isnumeric() else '-',       ❺
          format(unicodedata.numeric(char), '5.2f'),  ❻
          unicodedata.name(char),                     ❼
          sep='\t')
```

❶ Code point in U+0000 format.

❷ Character centralized in a str of length 6.

❸ Show re_dig if character matches the r'\d' regex.

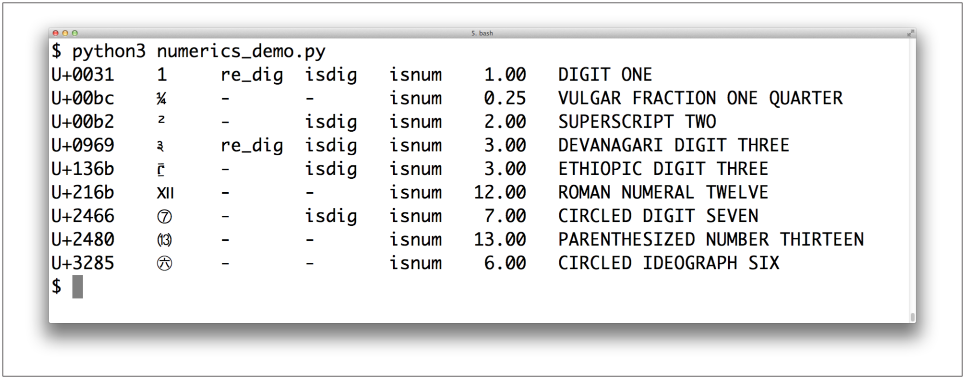
❹ Show isdig if char.isdigit() is True.

❺ Show isnum if char.isnumeric() is True.

❻ Numeric value formated with width 5 and 2 decimal places.

❼ Unicode character name.

Running Example 4-21 you get Figure 4-3.



```
$ python3 numerics_demo.py
U+0031    1     re_dig  isdig   isnum    1.00   DIGIT ONE
U+00bc    ¼     -       -       isnum    0.25   VULGAR FRACTION ONE QUARTER
U+00b2    ²     -       isdig   isnum    2.00   SUPERSCRIPT TWO
U+0969    ३     re_dig  isdig   isnum    3.00   DEVANAGARI DIGIT THREE
U+136b    ፫     -       isdig   isnum    3.00   ETHIOPIC DIGIT THREE
U+216b    Ⅻ     -       -       isnum   12.00   ROMAN NUMERAL TWELVE
U+2466    ⑦     -       isdig   isnum    7.00   CIRCLED DIGIT SEVEN
U+2480    ⒀     -       -       isnum   13.00   PARENTHESIZED NUMBER THIRTEEN
U+3285    ㊅     -       -       isnum    6.00   CIRCLED IDEOGRAPH SIX
$ 
```

Figure 4-3. Nine numeric characters and metadata about them; `re_dig` means the character matches the regular expression `r'\d'`.

The sixth column of Figure 4-3 is the result of calling `unicodedata.numeric(char)` on the character. It shows that Unicode knows the numeric value of symbols that represent numbers. So if you want to create a spreadsheet application that supports tamil digits or roman numerals, go for it!

Figure 4-3 shows that the regular expression `r'\d'` matches the digit "1" and the Devanagari digit three, but not some other characters that are considered digits by the `isdigit` function. The `re` module is not as savvy about Unicode as it could be. The new `regex` module available in PyPI was designed to eventually replace `re` and provides better Unicode support[17]. We'll come back to the `re` module in the next section.

Throughout this chapter we've used several `unicodedata` functions, but there are many more we did not cover. See the standard library documentation for the `unicodedata` module.

We will wrap up our tour of `str` versus `bytes` with a quick look at a new trend: dual mode APIs offering functions that accept `str` or `bytes` arguments with special handling depending on the type.

Dual mode `str` and `bytes` APIs

The standard library has functions that accept `str` or `bytes` arguments and behave differently depending on the type. Some examples are in the `re` and `os` modules.

[17] Although it was not better than `re` at identifying digits in this particular sample.

`str` versus `bytes` in regular expressions

If you build a regular expression with `bytes`, patterns such as `\d` and `\w` only match ASCII characters; in contrast, if these patterns are given as `str`, they match Unicode digits or letters beyond ASCII. Example 4-22 and Figure 4-4 compare how letters, ASCII digits, superscripts and Tamil digits are matched by `str` and `bytes` patterns.

Example 4-22. `ramanujan.py`: compare behavior of simple `str` and `bytes` regular expressions.

```
import re

re_numbers_str = re.compile(r'\d+')      ❶
re_words_str = re.compile(r'\w+')
re_numbers_bytes = re.compile(rb'\d+')   ❷
re_words_bytes = re.compile(rb'\w+')

text_str = ("Ramanujan saw \u0be7\u0bed\u0be8\u0bef"   ❸
            " as 1729 = 1³ + 12³ = 9³ + 10³.")         ❹

text_bytes = text_str.encode('utf_8')   ❺

print('Text', repr(text_str), sep='\n  ')
print('Numbers')
print('  str  :', re_numbers_str.findall(text_str))      ❻
print('  bytes:', re_numbers_bytes.findall(text_bytes))  ❼
print('Words')
print('  str  :', re_words_str.findall(text_str))        ❽
print('  bytes:', re_words_bytes.findall(text_bytes))    ❾
```

❶ The first two regular expressions are of the `str` type.

❷ The last two are of the `bytes` type.

❸ Unicode text to search, containing the Tamil digits for 1729 (the logical line continues until the right parenthesis token).

❹ This string is joined to the previous one at compile time (see String literal concatenation in the Language Reference)

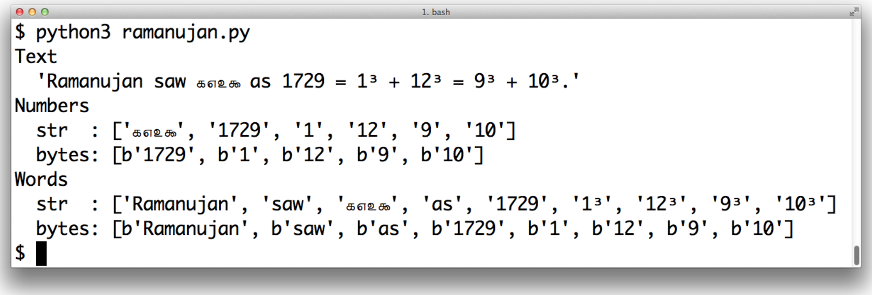
❺ A `bytes` string is needed to search with the `bytes` regular expressions.

❻ The `str` pattern `r'\d+'` matches the Tamil and ASCII digits.

❼ The `bytes` pattern `rb'\d+'` matches only the ASCII bytes for digits.

❽ The `str` pattern `r'\w+'` matches the letters, superscripts, Tamil and ASCII digits.

❾ The `bytes` pattern `rb'\w+'` matches only the ASCII bytes for letters and digits.



```
$ python3 ramanujan.py
Text
  'Ramanujan saw ௧௭௨௯ as 1729 = 1³ + 12³ = 9³ + 10³.'
Numbers
  str  : ['௧௭௨௯', '1729', '1', '12', '9', '10']
  bytes: [b'1729', b'1', b'12', b'9', b'10']
Words
  str  : ['Ramanujan', 'saw', '௧௭௨௯', 'as', '1729', '1³', '12³', '9³', '10³']
  bytes: [b'Ramanujan', b'saw', b'as', b'1729', b'1', b'12', b'9', b'10']
$
```

Figure 4-4. Screenshot of running `ramanujan.py` from Example 4-22.

Example 4-22 is a trivial example to make one point: you can use regular expressions on `str` and `bytes` but in the second case bytes outside of the ASCII range are treated as non-digits and non-word characters.

For `str` regular expressions there is a `re.ASCII` flag that makes `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` perform ASCII-only matching. See the documentation of the `re` module for full details.

Another important dual mode module is `os`.

`str` versus `bytes` on os functions

The GNU/Linux kernel is not Unicode savvy, so in the real world you may find file names made of byte sequences that are not valid in any sensible encoding scheme, and cannot be decoded to `str`. File servers with clients using a variety of OSes are particularly prone to this problem.

To work around this issue, all `os` module functions that accept file names or path names take arguments as `str` or `bytes`. If one such function is called with a `str` argument, the argument will be automatically converted using the codec named by `sys.getfilesystemencoding()`, and the OS response will be decoded with the same codec. This is almost always what you want, in keeping with the Unicode sandwich best practice.

But if you must deal with (and perhaps fix) file names that cannot be handled in that way, you can pass `bytes` arguments to the `os` functions to get `bytes` return values. This feature lets you deal with any file or path name, no matter how many gremlins you may find. See Example 4-23.

Example 4-23. `listdir` with `str` and `bytes` arguments and results.

```
>>> os.listdir('.')  # ①
['abc.txt', 'digits-of-π.txt']
```

```
>>> os.listdir(b'.')  # ❷
[b'abc.txt', b'digits-of-\xcf\x80.txt']
```

❶ The second filename is "digits-of-π.txt" (with the Greek letter pi).

❷ Given a `byte` argument, `listdir` returns filenames as bytes: `b'\xcf\x80'` is the UTF-8 encoding of the Greek letter pi).

To help with manual handling of `str` or `bytes` sequences that are file or path names, the `os` module provides special encoding and decoding functions:

`fsencode(filename)`
: Encodes `filename` (can be `str` or `bytes`) to `bytes` using the codec named by `sys.getfilesystemencoding()` if `filename` is of type `str`, otherwise return the `filename` `bytes` unchanged.

`fsdecode(filename)`
: Decodes `filename` (can be `str` or `bytes`) to `str` using the codec named by `sys.getfilesystemencoding()` if `filename` is of type `bytes`, otherwise return the `filename` `str` unchanged.

On Unix-derived platforms, these functions use the `surrogateescape` error handler (see next sidebar) to avoid choking on unexpected bytes. On Windows, the `strict` error handler is used.

Using `surrogateescape` to deal with gremlins

A trick to deal with unexpected bytes or unknown encodings is the `surrogateescape` codec error handler described in PEP 383 — Non-decodable Bytes in System Character Interfaces introduced in Python 3.1.

The idea of this error handler is to replace each non-decodable byte with a code point in the Unicode range from U+DC00 to U+DCFF that lies in the so-called "Low Surrogate Area" of the standard — a code space with no characters assigned, reserved for internal use in applications. On encoding, such code points are converted back to the byte values they replaced.

Example 4-24. `listdir` with `str` and `bytes` arguments and results.

```
>>> os.listdir('.')  ❶
['abc.txt', 'digits-of-π.txt']
>>> os.listdir(b'.')  ❷
[b'abc.txt', b'digits-of-\xcf\x80.txt']
>>> pi_name_bytes = os.listdir(b'.')[1]  ❸
>>> pi_name_str = pi_name_bytes.decode('ascii', 'surrogateescape')  ❹
>>> pi_name_str  ❺
'digits-of-\udccf\udc80.txt'
```

```
>>> pi_name_str.encode('ascii', 'surrogateescape')  ❻
b'digits-of-\xcf\x80.txt'
```

❶ List directory with a non-ASCII file name.

❷ Let's pretend we don't know the encoding and get file names as `bytes`.

❸ `pi_names_bytes` is the file name with the pi character.

❹ Decode it to `str` using the `'ascii'` codec with `'surrogateescape'`.

❺ Each non-ASCII byte is replaced by a surrogate code point: `'\xcf\x80'` becomes `'\udccf\udc80'`

❻ Encode back to ASCII bytes: each surrogate code point is replaced by the byte it replaced.

This ends our exploration of `str` and `bytes`. If you are still with me, congratulations!

Chapter summary

We started the chapter by dismissing the notion that `1 character == 1 byte`. As the world adopts Unicode (80% of Web sites already use UTF-8), we need to keep the concept of text strings separated from the binary sequences that represent them in files, and Python 3 enforces this separation.

After a brief overview of the binary sequence data types — `bytes`, `bytearray` and `memoryview` — we jumped into encoding and decoding, with a sampling of important codecs, followed by approaches to prevent or deal with the infamous `UnicodeEncodeError`, `UnicodeDecodeError` and the `SyntaxError` caused by wrong encoding in Python source files.

While on the subject of source code, I presented my position on the debate about non-ASCII identifiers: if the maintainers of the code base want to use a human language that has non-ASCII characters, the identifiers should follow suit — unless the code needs to run on Python 2 as well. But if the project aims to attract an international contributor base, identifiers should be made from English words, and then ASCII suffices.

We then considered the theory and practice of encoding detection in the absence of metadata: in theory, it can't be done, but in practice the Chardet package pulls it off pretty well for a number of popular encodings. Byte order marks were then presented as the only encoding hint commonly found in UTF-16 and UTF-32 files — sometimes in UTF-8 files as well.

In the next section we demonstrated opening text files, an easy task except for one pitfall: the `encoding=` keyword argument is not mandatory when you open a text file, but it should be. If you fail to specify the encoding, you end up with a program that manages to generate "plain text" that is incompatible across platforms, due to conflicting default encodings. We then exposed the different encoding settings that Python uses as defaults and how to detect them: `locale.getpreferredencoding()`, `sys.getfilesystemencoding()`, `sys.getdefaultencoding()` and the encodings for the standard I/O files (e.g. `sys.stdout.encoding`). A sad realization for Windows users is that these settings often have distinct values within the same machine, and the values are mutually incompatible; GNU/Linux and OSX users, in contrast, live in a happier place where `UTF-8` is the default pretty much everywhere.

Text comparisons are surprisingly complicated because Unicode provides multiple ways of representing some characters, so normalizing is a prerequisite to text matching. In addition to explaining normalization and case folding, we presented some utility functions that you may adapt to your needs, including drastic transformations like removing all accents. We then saw how to sort Unicode text correctly by leveraging the standard `locale` module — with some caveats — and an alternative that does not depend on tricky locale configurations: the external PyUCA package.

Finally we glanced at the Unicode database — a source of metadata about every character — and wrapped up with brief discussion of dual mode APIs — e.g. the `re` and `os` modules — where some functions can be called with `str` or `bytes` arguments, prompting different yet fitting results.

Further reading

Ned Batchelder's 2012 PyCon US talk "Pragmatic Unicode — or — How Do I Stop the Pain?" was outstanding. Ned is so professional that he provides a full transcript of the talk along with the slides and video. Esther Nam and Travis Fischer gave an excellent PyCon 2014 talk "Character encoding and Unicode in Python: How to (╯°□°)╯︵ ┻━┻ with dignity" (slides, video) from which I quoted this chapter's short and sweet epigraph: "Humans use text. Computers speak bytes". Lennart Regebro — one of this book's technical reviewers — presents his "Useful Mental Model of Unicode (UMMU)" in the short post Unconfusing Unicode: What is Unicode?. Unicode is a complex standard, so Lennart's UMMU is a really useful starting point.

The official Unicode HOWTO in the Python docs approaches the subject from several different angles, from a good historic intro to syntax details, codecs, regular expressions, file names and best practices for Unicode-aware I/O (i.e. the Unicode sandwich), with plenty of additional reference links from each section. Chapter 4 — Strings of Mark Pilgrim's awesome book "Dive into Python 3" also provides a very good intro to Unicode support in Python 3. In the same book, Chapter 15 describes how the Chardet library was ported from Python 2 to Python 3, a valuable case study given that the switch from old the `str` to the new `bytes` is the cause of most migration pains, and that is a central concern in a library designed to detect encodings.

If you know Python 2 but are new to Python 3, Guido van Rossum's What's New In Python 3.0 has 15 bullet points that summarize what changed, with lots of links. Guido starts with the blunt statement: "Everything you thought you knew about binary data and Unicode has changed." Armin Ronacher's blog post The Updated Guide to Unicode on Python is deep and highlights some of the pitfalls of Unicode in Python 3 (Armin is not a big fan of Python 3).

Chapter 2 — Strings and Text — of the Python Cookbook, 3rd. edition (O'Reilly, 2013), by David Beazley and Brian K. Jones, has several recipes dealing with Unicode normalization, sanitizing text, and performing text-oriented operations on byte sequences. Chapter 5 covers files and I/O, and it includes recipe 5.17. — Writing Bytes to a Text File — showing that underlying any text file there is always a binary stream that may be accessed directly when needed. Later in the cookbook the `struct` module is put to use in recipe 6.11 — Reading and Writing Binary Arrays of Structures.

Nick Coghlan's Python Notes blog has two posts very relevant to this chapter: Python 3 and ASCII Compatible Binary Protocols and Processing Text Files in Python 3. Highly recommended.

Binary sequences are about to gain new constructors and methods in Python 3.5, with one of the current constructor signatures being deprecated (see PEP 467 — Minor API improvements for binary sequences). Python 3.5 should also see the implementation of PEP 461 — Adding % formatting to bytes and bytearray.

A list of encodings supported by Python is available at Standard Encodings in the `codecs` module documentation. If you need to get that list programmatically, see how it's done in the `/Tools/unicode/listcodecs.py` script that comes with the CPython source code.

Martijn Faassen's Changing the Python default encoding considered harmful and Tarek Ziadé's sys.setdefaultencoding is evil explain why the default encoding you get from `sys.getdefaultencoding()` should never be changed, even if you discover how.

The books Unicode Explained by Jukka K. Korpela (O'Reilly, 2006) and Unicode Demystified by Richard Gillam (Addison-Wesley, 2003) are not Python-specific but were very helpful as I studied Unicode concepts. Programming with Unicode by Victor Stinner is a free self-published book (Creative Commons BY-SA) covering Unicode in general as well as tools and APIs in the context of the main operating systems and a few programming languages, including Python.

The W3C page Case Folding: An Introduction and Character Model for the World Wide Web: String Matching and Searching covers normalization concepts, with the former being a gentle introduction and the latter a working draft written in dry standard-speak — the same tone of the Unicode Standard Annex #15 — Unicode Normalization Forms. The Frequently Asked Questions / Normalization from Unicode.org is more readable, as is the NFC FAQ by Mark Davis — author of several Unicode algorithms and president of the Unicode Consortium at the time of this writing.

Soapbox

What is “plain text”?

For anyone who deals non-English text on a daily basis, “plain text” does not imply “ASCII”. The Unicode Glossary defines *plain text* like this:

> Computer-encoded text that consists only of a sequence of code points from a given standard, with no other formatting or structural information.

That definition starts very well, but I don’t agree with the part after the comma. HTML is a great example of a plain text format that carries formatting and structural information. But it’s still plain text because every byte in such a file is there to represent a text character, usually using UTF-8. There are no bytes with non-text meaning, as you can find in a `.png` or `.xls` document where most bytes represent packed binary values like RGB values and floating point numbers. In plain text, numbers are represented as sequences of digit characters.

I am writing this book in a plain text format called — ironically — `.asciidoc`. AsciiDoc is part of the toolchain of the excellent Atlas book publishing platform created by O’Reilly Media. AsciiDoc source files are plain text, but they are UTF-8, not ASCII. Otherwise writing this chapter would have been really painful. Despite the name, AsciiDoc is just great.

The world of Unicode is constantly expanding and, at the edges, tool support is not always there. That’s why I had to use images for Figure 4-1, Figure 4-3 and Figure 4-4: not all characters I wanted to show were available in the fonts used to render the book. On the other hand, the Ubuntu 14.04 and OSX 10.9 terminals display them perfectly well — including the Japanese characters for the word “mojibake”: ⍰⍰⍰⍰.

Unicode riddles

Imprecise qualifiers such as “often”, “most” and “usually” seem to pop up whenever I write about Unicode normalization. I regret the lack of more definitive advice, but there are so many exceptions to the rules in Unicode that it is hard to be absolutely positive.

For example, the µ (micro sign) is considered a “compatibility character” but the Ω (ohm) and Å (Ångström) symbols are not. The difference has practical consequences: NFC normalization — recommended for text matching — replaces the Ω (ohm) by Ω (uppercase Grek omega) and the Å (Ångström) by Å (uppercase A with ring above). But as a “compatibility character” the µ (micro sign) is not replaced by the visually identical μ (lowercase Greek mu), except when the stronger NFKC or NFKD normalizations are applied, and these transformations are lossy.

I understand the µ (micro sign) is in Unicode because it appears in the latin1 encoding and replacing it with the Greek mu would break round-trip conversion. After all, that's why the micro sign is a "compatibility character". But if the ohm and Ångström symbols are not in Unicode for compatibility reasons, then why have them at all? There are already code points for the GREEK CAPITAL LETTER OMEGA and the LATIN CAPITAL LETTER A WITH RING ABOVE which look the same and replace them on NFC normalization. Go figure.

My take after many hours studying Unicode: it is hugely complex and full of special cases, reflecting the wonderful variety of human languages and the politics of industry standards.

How are str represented in RAM?

The official Python docs avoid the issue of how the code points of a str are stored in memory. This is, after all, an implementation detail. In theory it doesn't matter: whatever the internal representation, every str must be encoded to bytes on output.

In memory, Python 3 stores each str as a sequence of code points using a fixed number of bytes per code point, to allow efficient direct access to any character or slice.

Before Python 3.3, CPython could be compiled to use either 16 or 32 bits per code point in RAM; the former was a "narrow build", the latter a "wide build". To know which you have, check the value of sys.maxunicode: 65535 implies a "narrow build" that can't handle code points above U+FFFF transparently. A "wide build" doesn't have this limitation, but consumes a lot of memory: 4 bytes per character, even while the vast majority of code points for Chinese ideographs fit in 2 bytes. Neither option was great, so you had to choose depending on your needs.

Since Python 3.3, when creating a new str object, the interpreter checks the characters in it and chooses the most economic memory layout that is suitable for that particular str: if there are only characters in the latin1 range, that str will use just one byte per code point. Otherwise 2 or 4 bytes per code point may be used, depending on the str. This is a simplification, for the full details look up PEP 393 — Flexible String Representation.

The flexible string representation is similar to the way the int type works in Python 3: if the integer fits in a machine word, it is stored in one machine word. Otherwise, the interpreter switches to a variable-length representation like that of the Python 2 long type. It is nice to see the spread of good ideas.

PART III

Functions as objects

CHAPTER 5

First-class functions

> I have never considered Python to be heavily influenced by functional languages, no matter what people say or think. I was much more familiar with imperative languages such as C and Algol 68 and although I had made functions first-class objects, I didn't view Python as a functional programming language[1].
>
> — Guido van Rossum
> *Python BDFL*

Functions in Python are first-class objects. Programming language theorists define a "first-class object" as a program entity that can be:

- created at runtime;
- assigned to a variable or element in a data structure;
- passed as an argument to a function;
- returned as the result of a function.

Integers, strings and dictionaries are other examples of first-class objects in Python — nothing fancy here. But, if you came to Python from a language where functions are not first-class citizens, this chapter and the rest of Part III of the book focuses on the implications and practical applications of treating functions as objects.

The term "first-class functions" is widely used as shorthand for "functions as first class objects". It's not perfect because it seems to imply an "elite" among functions. In Python, all functions are first-class.

[1] Origins of Python's "Functional" Features, in Guido's *Python History* blog

Treating a function like an object

The following console session shows that Python functions are objects. Here we create a function, call it, read its `__doc__` attribute, and check that the function object itself is an instance of the `function` class:

Example 5-1. Create and test a function, then read its `__doc__` and check its type.

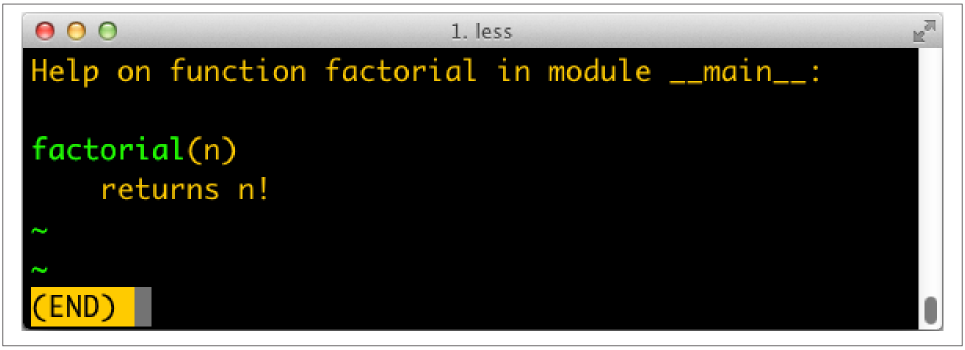
```
>>> def factorial(n):  ❶
...     '''returns n!'''
...     return 1 if n < 2 else n * factorial(n-1)
...
>>> factorial(42)
1405006117752879898543142606244511569936384000000000
>>> factorial.__doc__  ❷
'returns n!'
>>> type(factorial)  ❸
<class 'function'>
```

❶ This is a console session, so we're creating a function in "run time".

❷ `__doc__` is one of several attributes of function objects.

❸ `factorial` is an instance of the `function` class.

The `__doc__` attribute is used to generate the help text of an object. In the Python interactive console, the command `help(factorial)` will display a screen like that in Figure 5-1:



```
Help on function factorial in module __main__:

factorial(n)
    returns n!
~
~
(END)
```

Figure 5-1. Help screen for the factorial function. The text is from the `__doc__` attribute of the function object.

The next snippet shows the "first class" nature of a function object. We can assign it a variable `fact` and call it through that name. We can also pass `factorial` as an argument to `map`. The `map` function returns an iterable where each item is the the result of the application of the first argument (a function) to succesive elements of the second argument (an iterable), `range(10)` in this example.

Example 5-2. Use function through a different name, and pass function as argument.

```
>>> fact = factorial
>>> fact
<function factorial at 0x...>
>>> fact(5)
120
>>> map(factorial, range(11))
<map object at 0x...>
>>> list(map(fact, range(11)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Having first-class functions enables programming in a functional style. One of the hallmarks of functional programming is the use of higher-order functions, our next topic.

Higher-order functions

A function that takes a function as argument or returns a function as result is a *higher-order function*. One example is `map`, shown in Example 5-2. Another is the `sorted` built-in function: an optional `key` argument lets you provide a function to be applied to each item for sorting, as seen in "`list.sort` and the `sorted` built-in function" on page 42.

For example, to sort a list of words by length, simply pass the `len` function as the `key`, as in Example 5-3.

Example 5-3. Sorting a list of words by length.

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
>>> sorted(fruits, key=len)
['fig', 'apple', 'cherry', 'banana', 'raspberry', 'strawberry']
>>>
```

Any one-argument function can be used as key. For example, to create a rhyme dictionary it might be useful to sort each word spelled backwards. In the next snippet, note that the words in the list are not changed at all, only their reversed spelling is used as the sort criterion, so that the berries appear together.

Example 5-4. Sorting a list of words by their reversed spelling.

```
>>> def reverse(word):
...     return word[::-1]
>>> reverse('testing')
'gnitset'
>>> sorted(fruits, key=reverse)
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

In the functional programming paradigm, some of the best known higher-order functions are `map`, `filter`, `reduce` and `apply`. The `apply` function was deprecated in Python 2.3 and removed in Python 3 because it's no longer necessary. If you need to call a function with a dynamic set of arguments, you can just write `fn(*args, **keywords)` instead of `apply(fn, args, kwargs)`.

The `map`, `filter` and `reduce` higher-order functions are still around, but better alternatives are available for most of their use cases, as the next section shows.

Modern replacements for `map`, `filter` and `reduce`

Functional languages commonly offer the `map`, `filter` and `reduce` higher-order functions (sometimes with different names). The `map` and `filter` functions are still built-ins in Python 3, but since the introduction of list comprehensions and generator expressions, they are not as important. A listcomp or a genexp does the job of `map` and `filter` combined, but is more readable. Consider this:

Example 5-5. Lists of factorials produced with `map` and `filter` compared to alternatives coded as list comprehensions.

```
>>> list(map(fact, range(6)))  ❶
[1, 1, 2, 6, 24, 120]
>>> [fact(n) for n in range(6)]  ❷
[1, 1, 2, 6, 24, 120]
>>> list(map(factorial, filter(lambda n: n % 2, range(6))))  ❸
[1, 6, 120]
>>> [factorial(n) for n in range(6) if n % 2]  ❹
[1, 6, 120]
>>>
```

❶ Build a list of factorials from 0! to 5!

❷ Same operation, with a list comprehension.

❸ List of factorials of odd numbers up to 5!, using both `map` and `filter`.

❹ List comprehension does the same job, replacing `map` and `filter`, and making `lambda` unnecessary.

In Python 3, `map` and `filter` return generators — a form of iterator — so their direct substitute is now a generator expression (in Python 2 these functions returned lists, therefore their closest alternative is a listcomp).

The `reduce` function was demoted from a built-in in Python 2 to the `functools` module in Python 3. Its most common use case, summation, is better served by the `sum` built-in available since Python 2.3 was released in 2003. This is a big win in terms of readability and performance (see Example 5-5).

Example 5-6. Sum of integers up to 99 performed with `reduce` and `sum`.

```
>>> from functools import reduce  ❶
>>> from operator import add  ❷
>>> reduce(add, range(100))  ❸
4950
>>> sum(range(100))  ❹
4950
>>>
```

❶ Starting with Python 3.0, `reduce` is not a built-in.

❷ Import `add` to avoid creating a function just to add two numbers.

❸ Sum integers up to 99.

❹ Same task using `sum`; import or adding function not needed.

The common idea of `sum` and `reduce` is to apply some operation to successive items in a sequence, accumulating previous results, thus reducing a sequence of values to a single value.

Other reducing built-ins are `all` and `any`:

`all(iterable)`
: return `True` if every element of the `iterable` is truthy; `all([])` returns `True`.

`any(iterable)`
: return `True` if any element of the `iterable` is truthy; `all([])` returns `False`.

I give a fuller explanation of `reduce` in "`Vector` take #4: hashing and a faster ==" on page 290 where an ongoing example provides a meaningful context for the use of this function. The reducing functions are summarized later in the book when iterables are in focus, in "Iterable reducing functions" on page 436.

To use a higher-order function sometimes it is convenient to create a small, one-off function. That is why anonymous functions exist. We'll cover them next.

Anonymous functions

The `lambda` keyword creates an anonymous function within a Python expression.

However, the simple syntax of Python limits the body of lambda functions to be pure expressions. In other words, the body of a `lambda` cannot make assignments or use any other Python statement such as `while`, `try` etc.

The best use of anonymous functions is in the context of an argument list. For example, here is the rhyme index example from Example 5-4 rewritten with `lambda`, without defining a `reverse` function:

Example 5-7. Sorting a list of words by their reversed spelling using `lambda`.

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
>>> sorted(fruits, key=lambda word: word[::-1])
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

Outside the limited context of arguments to higher-order functions, anonymous functions are rarely useful in Python. The syntactic restrictions tend to make non-trivial lambdas either unreadable or unworkable.

Lundh's `lambda` refactoring recipe

If you find a piece of code hard to understand because of a `lambda`, Fredrik Lundh suggests this refactoring procedure:

1. Write a comment explaining what the heck that lambda does.
2. Study the comment for a while, and think of a name that captures the essence of the comment.
3. Convert the lambda to a def statement, using that name.
4. Remove the comment.

The steps above are quoted from the Functional Programming HOWTO, a must read.

The `lambda` syntax is just syntactic sugar: a `lambda` expression creates a function object just like the `def` statement. That is just one of several kinds of callable objects in Python. The following section overviews all of them.

The seven flavors of callable objects

The call operator, i.e. `()`, may be applied to other objects beyond user-defined functions. To determine whether an object is callable, use the `callable()` built-in function. The Python Data Model documentation lists seven callable types:

User-defined functions
: created with `def` statements or `lambda` expressions.

Built-in functions
: a function implemented in C (for CPython), like `len` or `time.strftime`.

Built-in methods
: methods implemented in C, like `dict.get`.

Methods

functions defined in the body of a class.

Classes

when invoked, a class runs its `__new__` method to create an instance, then `__init__` to initialize it, and finally the instance is returned to the caller. Because there is no `new` operator in Python, calling a class is like calling a function[2].

Class instances

if a class defines a `__call__` method, then its instances may be invoked as functions. See "User defined callable types" on page 145 below.

Generator functions

functions or methods that use the `yield` keyword. When called, generator functions return a generator object.

Generator functions are unlike other callables in many respects. Chapter 14 is devoted to them. They can also be used as coroutines, which are covered in Chapter 16.

Given the variety of existing callable types in Python, the safest way to determine whether an object is callable is to use the `callable()` built-in:

```
>>> abs, str, 13
(<built-in function abs>, <class 'str'>, 13)
>>> [callable(obj) for obj in (abs, str, 13)]
[True, True, False]
```

We now move to building class instances that work as callable objects.

User defined callable types

Not only are Python functions real objects, but arbitrary Python objects may also be made to behave like functions. Implementing a `__call__` instance method is all it takes.

Example 5-8 implements a `BingoCage` class. An instance is built from any iterable, and stores an internal `list` of items, in random order. Calling the instance pops an item.

Example 5-8. `bingocall.py`: A `BingoCage` does one thing: picks items from a shuffled list.

```
import random

class BingoCage:
```

[2] Usually calling a class creates an instance of the same class, but other behaviors are possible by overriding `__new__`. We'll see an example of this in "Flexible object creation with `__new__`" on page 594

```
    def __init__(self, items):
        self._items = list(items)  ①
        random.shuffle(self._items)  ②

    def pick(self):  ③
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')  ④

    def __call__(self):  ⑤
        return self.pick()
```

① `__init__` accepts any iterable; building a local copy prevents unexpected side-effects on any `list` passed as an argument.

② `shuffle` is guaranteed to work because `self._items` is a `list`.

③ The main method.

④ Raise exception with custom message if `self._items` is empty.

⑤ Shortcut to `bingo.pick()`: `bingo()`.

Here is a simple demo of Example 5-8. Note how a `bingo` instance can be invoked as a function, and the `callable(…)` built-in recognizes it as a callable object.

```
>>> bingo = BingoCage(range(3))
>>> bingo.pick()
1
>>> bingo()
0
>>> callable(bingo)
True
```

A class implementing `__call__` is an easy way to create function-like objects that have some internal state that must be kept across invocations, like the remaining items in the `BingoCage`. An example is a decorator. Decorators must be functions, but it is sometimes convenient to be able to "remember" something between calls of the decorator, for example for memoization — caching the results of expensive computations for later use.

A totally different approach to creating functions with internal state is to use closures. Closures, as well as decorators, are the subject of Chapter 7.

We now move to another aspect of handling functions as objects: run-time introspection.

Function introspection

Function objects have many attributes beyond `__doc__`. See below what the `dir` function reveals about our `factorial`:

```
>>> dir(factorial)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
'__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__get__', '__getattribute__', '__globals__',
'__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__',
'__module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__']
>>>
```

Most of these attributes are common to Python objects in general. In this section we cover those which are especially relevant to treating functions as objects, starting with `__dict__`.

Like the instances of a plain user-defined class, a function uses the `__dict__` attribute to store user attributes assigned to it. This is useful as a primitive form of annotation. Assigning arbitrary attributes to functions is not a very common practice in general, but Django is one framework that uses it. See, for example, the `short_description`, `boolean` and `allow_tags` attributes described in The Django admin site documentation. In the Django docs this example shows attaching a `short_description` to a method, to determine the description that will appear in record listings in the Django admin when that method is used:

```
def upper_case_name(obj):
    return ("%s %s" % (obj.first_name, obj.last_name)).upper()
upper_case_name.short_description = 'Customer name'
```

Now let us focus on the attributes that are specific to functions and are not found in a generic Python user-defined object. Computing the difference of two sets quickly gives us a list of the function-specific attributes:

Example 5-9. Listing attributes of functions that don't exist in plain instances.

```
>>> class C: pass  # ❶
>>> obj = C()  # ❷
>>> def func(): pass  # ❸
>>> sorted(set(dir(func)) - set(dir(obj))) # ❹
['__annotations__', '__call__', '__closure__', '__code__', '__defaults__',
'__get__', '__globals__', '__kwdefaults__', '__name__', '__qualname__']
>>>
```

❶ Create bare user-defined class.

❷ Make an instance of it.

❸ Create a bare function.

❹ Using set difference, generate a sorted list of the attributes that exist in a function but not in an instance of a bare class.

Table 5-1 shows a summary of the attributes listed by Example 5-9.

Table 5-1. Attributes of user-defined functions

name	type	description
`__annotations__`	`dict`	parameter and return annotations
`__call__`	`method-wrapper`	implementation of the () operator; a.k.a. the callable object protocol
`__closure__`	`tuple`	the function closure, i.e. bindings for free variables (often is None)
`__code__`	`code`	function metadata and function body compiled into bytecode
`__defaults__`	`tuple`	default values for the formal parameters
`__get__`	`method-wrapper`	implementation of the read-only descriptor protocol (see XREF)
`__globals__`	`dict`	global variables of the module where the function is defined
`__kwdefaults__`	`dict`	default values for the keyword-only formal parameters
`__name__`	`str`	the function name
`__qualname__`	`str`	the qualified function name, ex.: `Random.choice` (see PEP-3155)

In coming sections we will discuss the `__defaults__`, `__code__` and `__annotations__`, used by IDEs and frameworks to extract information about function signatures. But to fully appreciate these attributes, we will make a detour to explore the powerful syntax Python offers to declare function parameters and to pass arguments into them.

From positional to keyword-only parameters

One of the best features of Python functions is the extremely flexible parameter handling mechanism, enhanced with keyword-only arguments in Python 3. Closely related are the use of `*` and `**` to "explode" iterables and mappings into separate arguments when we call a function. To see these features in action, see the code for Example 5-10 and tests showing its use in Example 5-11.

Example 5-10. `tag` generates HTML. A keyword-only argument `cls` is used to pass `"class"` attributes as a work-around because `class` is a keyword in Python.

```
def tag(name, *content, cls=None, **attrs):
    """Generate one or more HTML tags"""
    if cls is not None:
        attrs['class'] = cls
    if attrs:
        attr_str = ''.join(' %s="%s"' % (attr, value)
                           for attr, value
                           in sorted(attrs.items()))
    else:
        attr_str = ''
    if content:
        return '\n'.join('<%s%s>%s</%s>' %
                         (name, attr_str, c, name) for c in content)
    else:
        return '<%s%s />' % (name, attr_str)
```

The `tag` function can be invoked in many ways, as Example 5-11 shows.

Example 5-11. Some of the many ways of calling the `tag` function from Example 5-10.

```
>>> tag('br')  ❶
'<br />'
>>> tag('p', 'hello')  ❷
'<p>hello</p>'
>>> print(tag('p', 'hello', 'world'))
<p>hello</p>
<p>world</p>
>>> tag('p', 'hello', id=33)  ❸
'<p id="33">hello</p>'
>>> print(tag('p', 'hello', 'world', cls='sidebar'))  ❹
<p class="sidebar">hello</p>
<p class="sidebar">world</p>
>>> tag(content='testing', name="img")  ❺
'<img content="testing" />'
>>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
...           'src': 'sunset.jpg', 'cls': 'framed'}
>>> tag(**my_tag)  ❻
'<img class="framed" src="sunset.jpg" title="Sunset Boulevard" />'
```

❶ A single positional argument produces an empty tag with that name.

❷ Any number of arguments after the first are captured by `*content` as a `tuple`.

❸ Keyword arguments not explicitly named in the `tag` signature are captured by `**attrs` as a `dict`.

❹ The `cls` parameter can only be passed as a keyword argument.

❺ Even the first positional argument can be passed as a keyword when `tag` is called.

❻ Prefixing the `my_tag` `dict` with `**` passes all its items as separate arguments which are then bound to the named parameters, with the remaining caught by `**attrs`.

Keyword-only arguments are a new feature in Python 3. In Example 5-10 the `cls` parameter can only be given as a keyword argument — it will never capture unnamed positional arguments. To specify keyword-only arguments when defining a function, name them after the argument prefixed with `*`. If you don't want to support variable positional arguments but still want keyword-only arguments, put a `*` by itself in the signature, like this:

```
>>> def f(a, *, b):
...     return a, b
...
>>> f(1, b=2)
(1, 2)
```

Note that keyword-only arguments do not need to have a default value: they can be mandatory, like `b` in the example above.

We now move to the introspection of function parameters, starting with a motivating example from a web framework, and on through introspection techniques.

Retrieving information about parameters

An interesting application of function introspection can be found in the Bobo HTTP micro-framework. To see that in action, consider a variation of the Bobo tutorial "Hello world" application in Example 5-12.

Example 5-12. Bobo knows that `hello` requires a `person` argument, and retrieves it from the HTTP request.

```
import bobo

@bobo.query('/')
def hello(person):
    return 'Hello %s!' % person
```

The `bobo.query` decorator integrates a plain function such as `hello` with the request handling machinery of the framework. We'll cover decorators in Chapter 7 — that's not the point of this example here. The point is that Bobo introspects the `hello` function and finds out it needs one parameter named `person` to work, and it will retrieve a parameter with that name from the request and pass it to `hello`, so the programmer does not need to touch the request object at all.

If you install Bobo and point its development server to the code above (e.g. `bobo -f hello.py`), a hit on the URL `http://localhost:8080/` will produce the message "Missing form variable person" with a 403 HTTP code. This happens because Bobo understands that the `person` argument is required to call `hello`, but no such name was found in the request. Example 5-13 is a shell session using `curl` to show this behavior.

Example 5-13. Bobo issues a 403 forbidden response if there are missing function arguments in the request; `curl -i` is used to dump the headers to standard output.

```
$ curl -i http://localhost:8080/
HTTP/1.0 403 Forbidden
Date: Thu, 21 Aug 2014 21:39:44 GMT
```

```
Server: WSGIServer/0.2 CPython/3.4.1
Content-Type: text/html; charset=UTF-8
Content-Length: 103

<html>
<head><title>Missing parameter</title></head>
<body>Missing form variable person</body>
</html>
```

However if you get `http://localhost:8080/?person=Jim`, the response will be the string `'Hello Jim!'`. See Example 5-14.

Example 5-14. Passing

```
$ curl -i http://localhost:8080/?person=Jim
HTTP/1.0 200 OK
Date: Thu, 21 Aug 2014 21:42:32 GMT
Server: WSGIServer/0.2 CPython/3.4.1
Content-Type: text/html; charset=UTF-8
Content-Length: 10

Hello Jim!
```

How does Bobo know what are the parameter names required by the function, and whether they have default values or not?

Within a function object, the `__defaults__` attribute holds a tuple with the default values of positional and keyword arguments. The defaults for keyword-only arguments appear in `__kwdefaults__`. The names of the arguments, however, are found within the `__code__` attribute, which is a reference to a `code` object with many attributes of its own.

To demonstrate the use of these attributes, we will inspect the function `clip` in a module `clip.py`, listed in Example 5-15.

Example 5-15. Function to shorten a string by clipping at a space near the desired length.

```python
def clip(text, max_len=80):
    """Return text clipped at the last space before or after max_len
    """
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ', 0, max_len)
        if space_before >= 0:
            end = space_before
        else:
            space_after = text.rfind(' ', max_len)
            if space_after >= 0:
                end = space_after
    if end is None:  # no spaces were found
```

```
        end = len(text)
    return text[:end].rstrip()
```

Example 5-16 shows the values of `__defaults__`, `__code__.co_varnames` and `__code__.co_argcount` for the `clip` function listed above.

Example 5-16. Extracting information about the function arguments.

```
>>> from clip import clip
>>> clip.__defaults__
(80,)
>>> clip.__code__  # doctest: +ELLIPSIS
<code object clip at 0x...>
>>> clip.__code__.co_varnames
('text', 'max_len', 'end', 'space_before', 'space_after')
>>> clip.__code__.co_argcount
2
```

As you can see, this is not the most convenient arrangement of information. The argument names appear in `__code__.co_varnames`, but that also includes the names of the local variables created in the body of the function. Therefore the argument names are the first N strings, where N is given by `__code__.co_argcount` which — by the way — does not include any variable arguments prefixed with `*` or `**`. The default values are identified only by their position in the `__defaults__` tuple, so to link each with the respective argument you have to scan from last to first. In the example, we have two arguments, `text` and `max_len`, and one default, `80`, so it must belong to the last argument, `max_len`. This is awkward.

Fortunately there is a better way: the `inspect` module.

Take a look at Example 5-17.

Example 5-17. Extracting the function signature.

```
>>> from clip import clip
>>> from inspect import signature
>>> sig = signature(clip)
>>> sig  # doctest: +ELLIPSIS
<inspect.Signature object at 0x...>
>>> str(sig)
'(text, max_len=80)'
>>> for name, param in sig.parameters.items():
...     print(param.kind, ':', name, '=', param.default)
...
POSITIONAL_OR_KEYWORD : text = <class 'inspect._empty'>
POSITIONAL_OR_KEYWORD : max_len = 80
```

Much better: `inspect.signature` returns an `inspect.Signature` object, which has a `parameters` attribute that lets you read an ordered mapping of names to `inspect.Parameter` objects. Each `Parameter` instance has attributes such as `name`, `default` and kind. The special value inspect._empty denotes parameters with no default — which makes sense considering that None is a valid — and popular — default value.

The kind attribute holds one of five possible values from _ParameterKind class:

POSITIONAL_OR_KEYWORD
: a parameter that may be passed as a positional or as a keyword argument (most Python function parameters are of this kind).

VAR_POSITIONAL
: a tuple of positional parameters.

VAR_KEYWORD
: a dict of keyword parameters.

KEYWORD_ONLY
: a keyword-only parameter (new in Python 3).

POSITIONAL_ONLY
: a positional-only parameter; currently unsupported by Python function declaration syntax, but exemplified by existing functions implemented in C — like divmod — that do not accept parameters passed by keyword.

Besides name, default and kind, inspect.Parameter objects have an annotation attribute which is usually inspect._empty but may contain function signature metadata provided via the new annotations syntax in Python 3 (annotations are covered in the next section).

An inspect.Signature object has a bind method that takes any number of arguments and binds them to the parameters in the signature, applying the usual rules for matching actual arguments to formal parameters. This can be used by a framework to validate arguments prior to the actual function invocation.

Example 5-18. Binding the function signature from the tag function in Example 5-10 do a dict of arguments.

```
>>> import inspect
>>> sig = inspect.signature(tag)  ❶
>>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
...           'src': 'sunset.jpg', 'cls': 'framed'}
>>> bound_args = sig.bind(**my_tag)  ❷
>>> bound_args
<inspect.BoundArguments object at 0x...>  ❸
>>> for name, value in bound_args.arguments.items():  ❹
...     print(name, '=', value)
...
name = img
cls = framed
attrs = {'title': 'Sunset Boulevard', 'src': 'sunset.jpg'}
>>> del my_tag['name']  ❺
>>> bound_args = sig.bind(**my_tag)  ❻
Traceback (most recent call last):
  ...
TypeError: 'name' parameter lacking default value
```

❶ Get the signature from `tag` function in Example 5-10.

❷ Pass a `dict` of arguments to `.bind()`.

❸ An `inspect.BoundArguments` object is produced.

❹ Iterate over the items in `bound_args.arguments`, which is an `OrderedDict`, to display the names and values of the arguments.

❺ Remove the mandatory argument `name` from `my_tag`.

❻ Calling `sig.bind(**my_tag)` raises a `TypeError` complaining of the missing `name` parameter.

This is shows how the Python Data Model, with the help of `inspect`, exposes the same machinery the interpreter uses to bind arguments to formal parameters in function calls.

Frameworks and tools like IDEs can use this information to validate code. Another feature of Python 3, function annotations, enhances the possible uses of this, as we will see next.

Function annotations

Python 3 provides syntax to attach metadata to the parameters of a function declaration and its return value. Example 5-19 is an annotated version of Example 5-15. The only differences are in the first line.

Example 5-19. Annotated `clip` function.

```
def clip(text:str, max_len:'int > 0'=80) -> str:  ❶
    """Return text clipped at the last space before or after max_len
    """
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ', 0, max_len)
        if space_before >= 0:
            end = space_before
        else:
            space_after = text.rfind(' ', max_len)
            if space_after >= 0:
                end = space_after
    if end is None:  # no spaces were found
        end = len(text)
    return text[:end].rstrip()
```

❶ The annotated function declaration.

Each argument in the function declaration may have an annotation expression preceded by `:`. If there is a default value, the annotation goes between the argument name and the `=` sign. To annotate the return value, add `->` and another expression between the `)` and the `:` at the tail of the function declaration. The expressions may be of any type. The most common types used in annotations are classes, like `str` or `int`, or strings, like `'int > 0'`, as seen in the annotation for `max_len` in Example 5-19.

No processing is done with the annotations. They are merely stored in the `__annotations__` attribute of the function, a `dict`:

```
>>> from clip_annot import clip
>>> clip.__annotations__
{'text': <class 'str'>, 'max_len': 'int > 0', 'return': <class 'str'>}
```

The item with key `'return'` holds the return value annotation marked with `->` in the function declaration in Example 5-19.

The only thing Python does with annotations is to store them in the `__annotations__` attribute of the function. Nothing else: no checks, enforcement, validation, or any other action is performed. In other words, annotations have no meaning to the Python interpreter. They are just metadata that may be used by tools, such as IDEs, frameworks and decorators. At this writing no tools that use this metadata exist in the standard library, except that `inspect.signature()` knows how to extract the annotations, as Example 5-20 shows.

Example 5-20. Extracting annotations from the function signature.

```
>>> from clip_annot import clip
>>> from inspect import signature
>>> sig = signature(clip)
>>> sig.return_annotation
<class 'str'>
>>> for param in sig.parameters.values():
...     note = repr(param.annotation).ljust(13)
...     print(note, ':', param.name, '=', param.default)
<class 'str'> : text = <class 'inspect._empty'>
'int > 0'     : max_len = 80
```

The `signature` function returns a `Signature` object which has a `return_annotation` attribute and a `parameters` dictionary mapping parameter names to `Parameter` objects. Each `Parameter` object has its own `annotation` attribute. That's how Example 5-20 works.

In the future, frameworks such as Bobo could support annotations to further automate request processing. For example, an argument annotated as `price:float` may be automatically converted from query string to the `float` expected by the function; a string annotation like quantity:'int > 0' might be parsed to perform conversion and validation of a parameter.

The biggest impact of functions annotations will probably not be dynamic settings such as Bobo, but in providing optional type information for static type checking in tools like IDEs and linters.

After this deep dive into the anatomy of functions, the remaining of this chapter covers the most useful packages in the standard library supporting functional programming.

Packages for functional programming

Although Guido makes it clear that Python does not aim to be a functional programming language, a functional coding style can be used to good extent, thanks to the support of packages like operator and functools, which we cover in the next two sections.

The operator module

Often in functional programming it is convenient to use an arithmetic operator as a function. For example, suppose you want to multiply a sequence of numbers to calculate factorials without using recursion. To perform summation you can use sum, but there is no equivalent function for multiplication. You could use reduce — as we saw in "Modern replacements for map, filter and reduce" on page 142 — but this requires a function to multiply two items of the sequence. Here is how to solve this using lambda:

Example 5-21. Factorial implemented with reduce and an anonymous function.

```
from functools import reduce

def fact(n):
    return reduce(lambda a, b: a*b, range(1, n+1))
```

To save you the trouble of writing trivial anonymous functions like lambda a, b: a*b, the operator module provides function equivalents for dozens of arithmetic operators. With it, we can rewrite Example 5-21 as Example 5-22.

Example 5-22. Factorial implemented with reduce and operator.mul.

```
from functools import reduce
from operator import mul

def fact(n):
    return reduce(mul, range(1, n+1))
```

Another group of one-trick lambdas that `operator` replaces are functions to pick items from sequences or read attributes from objects: `itemgetter` and `attrgetter` actually build custom functions to do that.

Example 5-23 shows a common use of `itemgetter`: sorting a list of tuples by the value of one field. In the example, the cities are printed sorted by country code (field 1). Essentially, `itemgetter(1)` does the same as `lambda fields: fields[1]`: create a function that, given a collection, returns the item at index 1.

Example 5-23. Demo of `itemgetter` to sort a list of tuples (data from Example 2-8).

```
>>> metro_data = [
...     ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
...     ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
...     ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
...     ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
...     ('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
... ]
>>>
>>> from operator import itemgetter
>>> for city in sorted(metro_data, key=itemgetter(1)):
...     print(city)
...
('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833))
('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889))
('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
('Mexico City', 'MX', 20.142, (19.433333, -99.133333))
('New York-Newark', 'US', 20.104, (40.808611, -74.020386))
```

If you pass multiple index arguments to `itemgetter`, the function it builds will return tuples with the extracted values:

```
>>> cc_name = itemgetter(1, 0)
>>> for city in metro_data:
...     print(cc_name(city))
...
('JP', 'Tokyo')
('IN', 'Delhi NCR')
('MX', 'Mexico City')
('US', 'New York-Newark')
('BR', 'Sao Paulo')
>>>
```

Because `itemgetter` uses the `[]` operator, it supports not only sequences but also mappings and any class that implements `__getitem__`.

A sibling of `itemgetter` is `attrgetter`, which creates functions to extract object attributes by name. If you pass `attrgetter` several attribute names as arguments, it also returns a tuple of values. In addition, if any argument name contains a `.` (dot), `attrgetter` navigates through nested objects to retrieve the attribute. These behaviors are shown in Example 5-24. This is not the shortest console session because we need to build a nested structure to showcase the handling of dotted attributes by `attrgetter`.

Example 5-24. Demo of `attrgetter` to process a previously defined list of `namedtuple` called `metro_data` (the same list that appears in Example 5-23).

```
>>> from collections import namedtuple
>>> LatLong = namedtuple('LatLong', 'lat long')  # ❶
>>> Metropolis = namedtuple('Metropolis', 'name cc pop coord')  # ❷
>>> metro_areas = [Metropolis(name, cc, pop, LatLong(lat, long))  # ❸
...     for name, cc, pop, (lat, long) in metro_data]
>>> metro_areas[0]
Metropolis(name='Tokyo', cc='JP', pop=36.933, coord=LatLong(lat=35.689722, long=139.691667))
>>> metro_areas[0].coord.lat  # ❹
35.689722
>>> from operator import attrgetter
>>> name_lat = attrgetter('name', 'coord.lat')  # ❺
>>>
>>> for city in sorted(metro_areas, key=attrgetter('coord.lat')):  # ❻
...     print(name_lat(city))  # ❼
...
('Sao Paulo', -23.547778)
('Mexico City', 19.433333)
('Delhi NCR', 28.613889)
('Tokyo', 35.689722)
('New York-Newark', 40.808611)
```

❶ Use `namedtuple` to define `LatLong`.

❷ Also define `Metropolis`.

❸ Build `metro_areas` list with `Metropolis` instances; note the nested tuple unpacking to extract `(lat, long)` and use them o build the `LatLong` for the `coord` attribute of `Metropolis`.

❹ Reach into element `metro_areas[0]` to get its latitude.

❺ Define an `attrgetter` to retrieve the `name` and the `coord.lat` nested attribute.

❻ Use `attrgetter` again to sort list of cities by latitude.

❼ Use the attrgetter defined in <5> to show only city name and latitude.

Here is a partial list of functions defined in `operator` (names starting with _ are omitted, as they are mostly implementation details):

```
>>> [name for name in dir(operator) if not name.startswith('_')]
['abs', 'add', 'and_', 'attrgetter', 'concat', 'contains',
'countOf', 'delitem', 'eq', 'floordiv', 'ge', 'getitem', 'gt',
'iadd', 'iand', 'iconcat', 'ifloordiv', 'ilshift', 'imod', 'imul',
'index', 'indexOf', 'inv', 'invert', 'ior', 'ipow', 'irshift',
'is_', 'is_not', 'isub', 'itemgetter', 'itruediv', 'ixor', 'le',
'length_hint', 'lshift', 'lt', 'methodcaller', 'mod', 'mul', 'ne',
'neg', 'not_', 'or_', 'pos', 'pow', 'rshift', 'setitem', 'sub',
'truediv', 'truth', 'xor']
```

Most of the 52 names above are self-evident. The group of names prefixed with `i` and the name of another operator — e.g. `iadd`, `iand` etc. — correspond to the augmented assignment operators — e.g. `+=`, `&=` etc. These change their first argument in-place, if it is mutable; if not, the function works like the one without the `i` prefix: it simply returns the result of the operation.

Of the remaining `operator` functions, `methodcaller` is the last we will cover. It is somewhat similar to `attrgetter` and `itemgetter` in that it creates a function on-the-fly. The function it creates calls a method by name on the object given as argument.

Example 5-25. Demo of `methodcaller`: second test shows the binding of extra arguments.

```
>>> from operator import methodcaller
>>> s = 'The time has come'
>>> upcase = methodcaller('upper')
>>> upcase(s)
'THE TIME HAS COME'
>>> hiphenate = methodcaller('replace', ' ', '-')
>>> hiphenate(s)
'The-time-has-come'
```

The first test in Example 5-25 is there just to show `methodcaller` at work, but if you need to use the `str.upper` as a function you can just call it on the `str` class and pass a string as argument, like this:

```
>>> str.upper(s)
'THE TIME HAS COME'
```

The second test in Example 5-25 shows that `methodcaller` can also do a partial application to freeze some arguments, like the `functools.partial` function does. That is our next subject.

Freezing arguments with `functools.partial`

The `functools` module brings together a handful of higher-order functions. The best known of them is probably `reduce`, which was covered in "Modern replacements for `map`, `filter` and `reduce`" on page 142. Of the remaining functions in `functools` the most useful is `partial` and its variation, `partialmethod`.

The `functools.partial` is a higher-order function that allows partial application of a function. Given a function, a partial application produces a new callable with some of the arguments of the original function fixed. This is useful to adapt a function that takes one or more arguments to an API that requires a callback with less arguments. Example 5-26 is a trivial demonstration.

Example 5-26. Using `partial` to use a 2-argument function where a 1-argument callable is required.

```
>>> from operator import mul
>>> from functools import partial
>>> triple = partial(mul, 3)  ❶
>>> triple(7)  ❷
21
>>> list(map(triple, range(1, 10)))  ❸
[3, 6, 9, 12, 15, 18, 21, 24, 27]
```

❶ Create new `triple` function from `mul`, binding first positional argument to `3`.

❷ Test it.

❸ Use `triple` with `map`; `mul` would not work with `map` in this example.

A more useful example involves the `unicode.normalize` function that we saw in "Normalizing Unicode for saner comparisons" on page 116. If you work with text from many languages, you may want to apply `unicode.normalize('NFC', s)` to any string `s` before comparing or storing it. If you do that often, its handy to have an `nfc` function to do that, as in Example 5-27.

Example 5-27. Building a convenient Unicode normalizing function with `partial`.

```
>>> import unicodedata, functools
>>> nfc = functools.partial(unicodedata.normalize, 'NFC')
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> s1 == s2
False
>>> nfc(s1) == nfc(s2)
True
```

`partial` takes a callable as first argument, followed by an arbitrary number of positional and keyword arguments to bind.

Example 5-28 shows the use of `partial` with the `tag` function from Example 5-10, to freeze one positional argument and one keyword argument.

Example 5-28. Demo of `partial` applied to the function `tag` from Example 5-10.

```
>>> from tagger import tag
>>> tag
<function tag at 0x10206d1e0>  ❶
>>> from functools import partial
>>> picture = partial(tag, 'img', cls='pic-frame')  ❷
>>> picture(src='wumpus.jpeg')
'<img class="pic-frame" src="wumpus.jpeg" />'  ❸
```

```
>>> picture
functools.partial(<function tag at 0x10206d1e0>, 'img', cls='pic-frame')  ❹
>>> picture.func  ❺
<function tag at 0x10206d1e0>
>>> picture.args
('img',)
>>> picture.keywords
{'cls': 'pic-frame'}
```

❶ Import `tag` from Example 5-10 and show its id.

❷ Create `picture` function from `tag` by fixing the first positional argument with `'img'` and the `cls` keyword argument with `'pic-frame'`.

❸ `picture` works as expected.

❹ `partial()` returns a `functools.partial` object[3].

❺ A `functools.partial` object has attributes providing access to the original function and the fixed arguments.

The `functools.partialmethod` function (new in Python 3.4) does the same job as `partial`, but is designed to work with methods.

An impressive `functools` function is `lru_cache`, which does memoization — a form of automatic optimization that works by storing the results of function calls to avoid expensive recalculations. We will cover it in Chapter 7, where decorators are explained, along with other higher-order functions designed to be used as decorators: `singledispatch` and `wraps`.

Chapter summary

The goal of this chapter was to explore the first-class nature of functions in Python. The main ideas are that you can assign functions to variables, pass them to other functions, store them in data structures and access function attributes, allowing frameworks and tools to act on that information. Higher-order functions, a staple of functional programming, are common in Python — even if the use of `map`, `filter` and `reduce` is not as frequent as it was — thanks to list comprehensions (and similar constructs like generator expressions) and the appearance of reducing built-ins like `sum`, `all` and `any`. The `sorted`, `min`, `max` built-ins, and `functools.partial` are examples of commonly used higher-order functions in the language.

[3] The source code for `functools.py` reveals that the `functools.partial` class is implemented in C and is used by default. If that is not available, a pure-Python implementation of `partial` is available since Python 3.4.in the `functools` module.

Callables come in seven different flavors in Python, from the simple functions created with `lambda` to instances of classes implementing `__call__`. They can all be detected by the `callable()` built-in. Every callable supports the same rich syntax for declaring formal parameters, including keyword-only parameters and annotations — both new features introduced with Python 3.

Python functions and their annotations have a rich set of attributes that can be read with the help of the `inspect` module, which includes the `Signature.bind` method to apply the flexible rules that Python uses to bind actual arguments to declared parameters.

Lastly, we covered some functions from the `operator` module and `functools.partial`, which facilitate functional programming by minimizing the need for the functionally-challenged `lambda` syntax.

Further reading

The next two chapters continue our exploration of programming with function objects. Chapter 6 shows how first-class functions can simplify some classic Object Oriented design patterns, while Chapter 7 dives into function decorators — a special kind of higher-order function — and the closure mechanism that makes them work.

Chapter 7 of the Python Cookbook, 3rd. edition (O'Reilly, 2013), by David Beazley and Brian K. Jones, is an excellent complement to the current chapter as well as Chapter 7 of this book, covering mostly the same concepts with a different approach.

In the Python Language Reference, chapter "3. Data model", section 3.2 The standard type hierarchy presents the seven callable types, along with all the other built-in types.

The Python-3-only features discussed in this chapter have their own PEPs: PEP 3102 — Keyword-Only Arguments and PEP 3107 — Function Annotations.

For more about the current (as of mid-2014) use of annotations, two Stack Overflow questions are worth reading: What are good uses for Python3's "Function Annotations", has a practical answer and insightful comments by Raymond Hettinger, and the answer for What good are Python function annotations? quotes extensively from Guido van Rossum.

PEP 362 — Function Signature Object is worth reading if you intend to use the `inspect` module which implements that feature.

A great introduction to functional programming in Python is A. M. Kuchling's Python Functional Programming HOWTO. The main focus of that text, however, is on the use of iterators and generators, which are the subject of Chapter 14.

fn.py is a package to support Functional programming in Python 2 and 3. According to its author, Alexey Kachayev, fn.py provides "implementation of missing features to enjoy FP" in Python. It includes a @recur.tco decorator that implements tail-call optimization for unlimited recursion in Python, among many other functions, data structures and recipes.

The StackOverflow question Python: Why is functools.partial necessary? has a highly informative (and funny) reply by Alex Martelli, author of the classic *Python in a Nutshell*.

Jim Fulton's Bobo was probably the first Web framework that deserved to be called Object Oriented. If you were intrigued by it and want to learn more about its modern rewrite, start at its Introduction. A little of the early history of Bobo appears in a comment by Phillip J. Eby in a discussion at Joel Spolsky's blog.

Soapbox

About Bobo

I owe my Python career to Bobo. I used it in my first Python Web project, in 1998. I discovered Bobo while looking for an object-oriented way to code web applications, after trying Perl and Java alternatives.

In 1997 Bobo had pioneered the object publishing concept: direct mapping from URLs to a hierarchy of objects, with no need to configure routes. I was hooked when I saw the beauty of this. Bobo also featured automatic HTTP query handling based on analysis of the signatures of the methods or functions used to handle requests.

Bobo was created by Jim Fulton, known as "The Zope Pope" thanks to his leading role in the development of the Zope framework, the foundation of the Plone CMS, SchoolTool, ERP5 and other large-scale Python projects. Jim is also the creator of ZODB — the Zope Object Database — a transactional object database that provides ACID (atomicity, consistency, isolation and durability), designed for ease of use from Python.

Jim has since rewritten Bobo from scratch to support WSGI and modern Python (including Python 3). As of this writing, Bobo uses the six library to do the function introspection, in order to be compatible with Python 2 and Python 3 in spite of the changes in function objects and related APIs.

Is Python a functional language?

Around the year 2000 I was at a training in the US when Guido van Rossum dropped by the classroom (he was not the instructor). In the Q&A that followed, somebody asked him which features of Python were borrowed from other languages. His answer: "Everything that is good in Python was stolen from other languages."

Shriram Krishnamurthi, professor of Computer Science at Brown University, starts his Teaching Programming Languages in a Post-Linnaean Age paper with this:

> Programming language "paradigms" are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them?

In that paper, Python is mentioned by name in this passage:

> What else to make of a language like Python, Ruby, or Perl? Their designers have no patience for the niceties of these Linnaean hierarchies; they borrow features as they wish, creating melanges that utterly defy characterization.

Krishnamurthi submits that instead of trying to classify languages in some taxonomy, it's more useful to consider them as aggregations of features.

Even if it was not Guido's goal, endowing Python with first-class functions opened the door to functional programming. In his post Origins of Python's "Functional" Features, he tells that `map`, `filter` and `reduce` were the motivation for adding `lambda` to Python in the first place. All of these features were contributed together by Amrit Prem for Python 1.0 in 1994 (according to Misc/HISTORY in the CPython source code).

`lambda`, `map`, `filter` and `reduce` first appeared in Lisp, the original functional language. However, Lisp does not limit what can be done inside a `lambda`, because everything in Lisp is an expression. Python uses a statement-oriented syntax in which expressions cannot contain statements, and many language constructs are statements — including `try/catch`, which is what I miss most often when writing `lambdas`. This is the price to pay for Python's highly readable syntax[4]. Lisp has many strengths, but readability is not one of them.

Ironically, stealing the list comprehension syntax from another functional language — Haskell — significantly diminished the need for `map` and `filter`, and also for `lambda`.

Besides the limited anonymous function syntax, the biggest obstacle to wider adoption of functional programming idioms in Python is the lack of tail-recursion elimination, an optimization that allows memory-efficient computation of a function that makes a recursive call at the "tail" of its body. In another blog post, Tail Recursion Elimination Guido gives several reasons why such optimization is not a good fit for Python. That post is a great read for the technical arguments, but even more so because the first three and most important reasons given are usability issues. It is no accident that Python is a pleasure to use, learn and teach. Guido made it so.

So there you have it: Python is, by design, not a functional language — whatever that means. Python just borrows a few good ideas from functional languages.

The problem with anonymous functions

Beyond the Python-specific syntax constraints, anonymous functions have a serious drawback in every language: they have no name.

[4] There also the problem of lost indentation when pasting code to Web forums, but I digress.

I am only half joking here. Stack traces are easier to read when functions have names. Anonymous functions are a handy shortcut, people have fun coding with them, but sometimes they get carried away — especially if the language and environment encourage deep nesting of anonymous functions, like JavaScript on Node.js. Lots of nested anonymous functions make debugging and error handling hard. Asynchronous programming in Python is more structured, perhaps because the limited `lambda` demands it. I promise to write more about asynchronous programming in the future, but this subject must be deferred to Chapter 18. By the way, promises, futures and deferreds are concepts used in modern asynchronous APIs. Along with coroutines, they provide an escape from the so called "callback hell". We'll see how callback-free asynchronous programming works in "From callbacks to futures and coroutines" on page 564.

CHAPTER 6
Design patterns with first-class functions

> Conformity to patterns is not a measure of goodness[1].
>
> — Ralph Johnson
> *co-author of the Design Patterns classic*

Although design patterns are language-independent, that does not mean every pattern applies to every language. In his 1996 presentation Design Patterns in Dynamic Languages, Peter Norvig states that 16 out of the 23 patterns in the original Design Patterns book by Gamma et.al. become either "invisible or simpler" in a dynamic language (slide 9). He was talking about Lisp and Dylan, but many of the relevant dynamic features are also present in Python.

The authors of the Design Patterns book acknowledge in their Introduction that the implementation language determines which patterns are relevant:

> The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called "Inheritance," "Encapsulation," and "Polymorphism." Similarly, some of our patterns are supported directly by the less common object-oriented languages. CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor.[2]

In particular, in the context of languages with first-class functions, Norvig suggests rethinking the Strategy, Command, Template Method and Visitor patterns. The general idea is: you can replace instances of some participant class in these patterns with simple functions, reducing a lot of boilerplate code. In this chapter we will refactor Strategy

[1] From a slide in the talk *Root cause analysis of some faults in Design Patterns* presented by Ralph Johnson at IME/CCSL, Universidade de São Paulo, Nov. 15. 2014.

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides — *Design Patterns: Elements of Reusable Object-Oriented Software.* (Addison-Wesley, 1995), p. 4

using function objects, and discuss a similar approach to simplifying the Command pattern.

Case study: refactoring Strategy

Strategy is a good example of a design pattern that can be simpler in Python if you leverage functions as first class objects. In the following section we describe and implement Strategy using the "classic" structure described in the Design patterns book. If you are familiar with the classic pattern, you may skip to "Function-oriented Strategy" on page 172 where we refactor the code using functions, significantly reducing the line count.

Classic Strategy

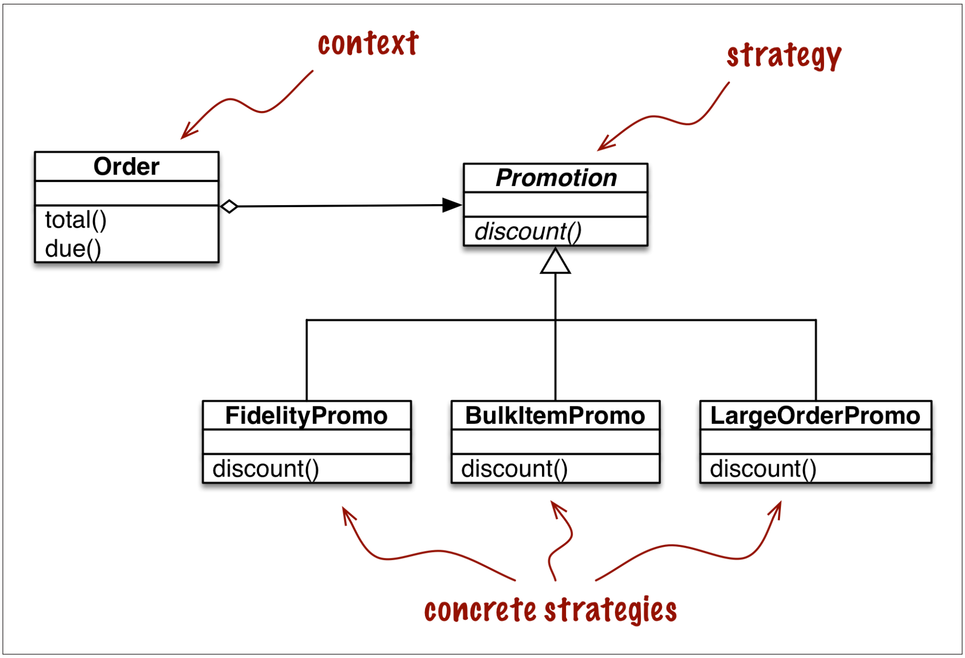


Figure 6-1. UML class diagram for order discount processing implemented with the Strategy design pattern.

The Strategy pattern is summarized like this in the Design Patterns book:

> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.[3]

A clear example of Strategy applied in the e-commerce domain is computing discounts to orders according to the attributes of the customer or inspection of the ordered items.

Consider an online store with these discount rules:

- customers with 1000 or more fidelity points get a global 5% discount per order;
- a 10% discount is applied to each line item with 20 or more units in the same order;
- orders with at least 10 distinct items get a 7% global discount.

For brevity, let us assume that only one discount may be applied to an order.

The UML class diagram for the Strategy pattern is depicted in Figure 6-1. Its participants are:

Context
: Provides a service by delegating some computation to interchangeable components that implement alternative algorithms. In the e-commerce example, the context is an `Order`, which is configured to apply a promotional discount according to one of several algorithms.

Strategy
: The interface common to the components that implement the different algorithms. In our example, this role is played by an abstract class called `Promotion`.

Concrete Strategy
: One of the concrete subclasses of Strategy. `FidelityPromo`, `BulkPromo` and `LargeOrderPromo` are the three concrete strategies implemented.

The code in Example 6-1 follows the blueprint in Figure 6-1. As described in the Design Patterns book, the concrete strategy is chosen by the client of the context class. In our example, before instantiating an order, the system would somehow select a promotional discount strategy and pass it to the `Order` constructor. The selection of the strategy is outside of the scope of the pattern.

Example 6-1. Implementation `Order` class with pluggable discount strategies.

```
from abc import ABC, abstractmethod
from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')
```

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides — *Design Patterns: Elements of Reusable Object-Oriented Software.* (Addison-Wesley, 1995), p. 9

```
class LineItem:

    def __init__(self, product, quantity, price):
        self.product = product
        self.quantity = quantity
        self.price = price

    def total(self):
        return self.price * self.quantity


class Order:  # the Context

    def __init__(self, customer, cart, promotion=None):
        self.customer = customer
        self.cart = list(cart)
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion.discount(self)
        return self.total() - discount

    def __repr__(self):
        fmt = '<Order total: {:.2f} due: {:.2f}>'
        return fmt.format(self.total(), self.due())


class Promotion(ABC):  # the Strategy: an Abstract Base Class

    @abstractmethod
    def discount(self, order):
        """Return discount as a positive dollar amount"""


class FidelityPromo(Promotion):  # first Concrete Strategy
    """5% discount for customers with 1000 or more fidelity points"""

    def discount(self, order):
        return order.total() * .05 if order.customer.fidelity >= 1000 else 0


class BulkItemPromo(Promotion):  # second Concrete Strategy
    """10% discount for each LineItem with 20 or more units"""
    def discount(self, order):
        discount = 0
        for item in order.cart:
            if item.quantity >= 20:
                discount += item.total() * .1
        return discount


class LargeOrderPromo(Promotion):  # third Concrete Strategy
    """7% discount for orders with 10 or more distinct items"""

    def discount(self, order):
        distinct_items = {item.product for item in order.cart}
        if len(distinct_items) >= 10:
            return order.total() * .07
        return 0
```

Note that in Example 6-1, I coded `Promotion` as an ABC (Abstract Base Class), to be able to use the `@abstractmethod` decorator, thus making the pattern more explicit.

> In Python 3.4 the simplest way to declare an ABC is to subclass `abc.ABC`, as I did in Example 6-1. From Python 3.0 to 3.3 you must use the `metaclass=` keyword in the `class` statement, e.g. `class Promotion(metaclass=ABCMeta):`.

Example 6-2 shows doctests used to demonstrate and verify the operation of a module implementing the rules above.

Example 6-2. Sample usage of `Order` class with different promotions applied.

```
    >>> joe = Customer('John Doe', 0)  ❶
    >>> ann = Customer('Ann Smith', 1100)
    >>> cart = [LineItem('banana', 4, .5),  ❷
    ...         LineItem('apple', 10, 1.5),
    ...         LineItem('watermellon', 5, 5.0)]
    >>> Order(joe, cart, FidelityPromo())  ❸
    <Order total: 42.00 due: 42.00>
    >>> Order(ann, cart, FidelityPromo())  ❹
    <Order total: 42.00 due: 39.90>
    >>> banana_cart = [LineItem('banana', 30, .5),  ❺
    ...                LineItem('apple', 10, 1.5)]
    >>> Order(joe, banana_cart, BulkItemPromo())  ❻
    <Order total: 30.00 due: 28.50>
    >>> long_order = [LineItem(str(item_code), 1, 1.0)  ❼
    ...               for item_code in range(10)]
    >>> Order(joe, long_order, LargeOrderPromo())  ❽
    <Order total: 10.00 due: 9.30>
    >>> Order(joe, cart, LargeOrderPromo())
    <Order total: 42.00 due: 42.00>
```

❶ Two customers: joe has 0 fidelity points, ann has 1100.

❷ One shopping cart with three line items.

❸ The FidelityPromo promotion gives no discount to joe.

❹ ann gets a 5% discount because she has at least 1000 points.

❺ The banana_cart has 30 units of the "banana" product and 10 apples.

❻ Thanks to the BulkItemPromo, joe gets a $1.50 discount on the bananas.

❼ long_order has 10 different items at $1.00 each.

❽ joe gets a 7% discount on the whole order because of LargerOrderPromo.

Example 6-1 works perfectly well, but the same functionality can be implemented with less code in Python by using functions as objects. The next section shows how.

Function-oriented Strategy

Each concrete strategy in Example 6-1 is a class with a single method, discount. Furthermore, the strategy instances have no state (no instance attributes). You could say they look a lot like plain functions, and you would be right. Example 6-3 is a refactoring of Example 6-1 replacing the concrete strategies with simple functions, and removing the Promo abstract class.

Example 6-3. Order class with discount strategies implemented as functions.

```
from collections import namedtuple

Customer = namedtuple('Customer', 'name fidelity')


class LineItem:

    def __init__(self, product, quantity, price):
        self.product = product
        self.quantity = quantity
        self.price = price

    def total(self):
        return self.price * self.quantity


class Order:  # the Context

    def __init__(self, customer, cart, promotion=None):
        self.customer = customer
        self.cart = list(cart)
        self.promotion = promotion

    def total(self):
        if not hasattr(self, '__total'):
            self.__total = sum(item.total() for item in self.cart)
        return self.__total

    def due(self):
        if self.promotion is None:
            discount = 0
        else:
            discount = self.promotion(self)  ❶
        return self.total() - discount

    def __repr__(self):
        fmt = '<Order total: {:.2f} due: {:.2f}>'
        return fmt.format(self.total(), self.due())

 ❷

def fidelity_promo(order):  ❸
    """5% discount for customers with 1000 or more fidelity points"""
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0


def bulk_item_promo(order):
    """10% discount for each LineItem with 20 or more units"""
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * .1
    return discount


def large_order_promo(order):
    """7% discount for orders with 10 or more distinct items"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * .07
    return 0
```

❶ To compute a discount, just call the `self.promotion()` function.

❷ No abstract class.

❸ Each strategy is a function.

The code in Example 6-3 is 12 lines shorter than Example 6-1. Using the new `Order` is also a bit simpler, as shown in the Example 6-4 doctests.

Example 6-4. Sample usage of `Order` class with promotions as functions.

```
>>> joe = Customer('John Doe', 0)  ❶
>>> ann = Customer('Ann Smith', 1100)
>>> cart = [LineItem('banana', 4, .5),
...         LineItem('apple', 10, 1.5),
...          LineItem('watermellon', 5, 5.0)]
>>> Order(joe, cart, fidelity_promo)  ❷
<Order total: 42.00 due: 42.00>
>>> Order(ann, cart, fidelity_promo)
<Order total: 42.00 due: 39.90>
>>> banana_cart = [LineItem('banana', 30, .5),
...                LineItem('apple', 10, 1.5)]
>>> Order(joe, banana_cart, bulk_item_promo)  ❸
<Order total: 30.00 due: 28.50>
>>> long_order = [LineItem(str(item_code), 1, 1.0)
...               for item_code in range(10)]
>>> Order(joe, long_order, large_order_promo)
<Order total: 10.00 due: 9.30>
>>> Order(joe, cart, large_order_promo)
<Order total: 42.00 due: 42.00>
```

❶ Same test fixtures as Example 6-1.

❷ To apply a discount strategy to an `Order`, just pass the promotion function as an argument.

❸ A different promotion function is used here and in the next test.

Note the callouts in Example 6-4: there is no need to instantiate a new promotion object with each new order: the functions are ready to use.

It is interesting to note that in the Design Patterns book the authors suggest: "Strategy objects often make good flyweights"[4]. A definition of the Flyweight in another part of that work states: "A flyweight is a shared object that can be used in multiple contexts simultaneously"[5]. The sharing is recommended to reduce the cost of creating a new concrete strategy object when the same strategy is applied over and over again with every new context — with every new `Order` instance, in our example. So, to overcome a drawback of the Strategy pattern — its runtime cost — the authors recommend applying yet another pattern. Meanwhile, the line count and maintenance cost of your code are piling up.

A thornier use case, with complex concrete strategies holding internal state may require all the pieces of the Strategy and Flyweight design patterns combined. But often concrete strategies have no internal state, they only deal with data from the context. If that is the case, then by all means use plain old functions instead of coding single-method classes implementing a single-method interface declared in yet another class. A function is more lightweight than an instance of a user-defined class, and there is no need for Flyweight as each strategy function is created just once by Python when it compiles the

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides — *Design Patterns: Elements of Reusable Object-Oriented Software.* (Addison-Wesley, 1995), p. 323

[5] *idem*, p. 196

module. A plain function is also "a shared object that can be used in multiple contexts simultaneously."

Now that we have implemented the Strategy pattern with functions, other possibilities emerge. Suppose you want to create a "meta-strategy" that selects the best available discount for a given `Order`. In the next sections we present additional refactorings that implement this requirement using a variety of approaches that leverage functions and modules as objects.

Choosing the best strategy: simple approach

Given the same customers and shopping carts from the tests in Example 6-4, we now add three additional tests in Example 6-5:

Example 6-5. The `best_promo` function applies all discounts and returns the largest.

```
>>> Order(joe, long_order, best_promo)  ❶
<Order total: 10.00 due: 9.30>
>>> Order(joe, banana_cart, best_promo)  ❷
<Order total: 30.00 due: 28.50>
>>> Order(ann, cart, best_promo)  ❸
<Order total: 42.00 due: 39.90>
```

❶ `best_promo` selected the `larger_order_promo` for customer `joe`.

❷ Here `joe` got the discount from `bulk_item_promo` for ordering lots of bananas.

❸ Checking out with a simple cart, `best_promo` gave loyal customer `ann` the discount for the `fidelity_promo`.

The implementation of `best_promo` is very simple. See Example 6-6.

Example 6-6. `best_promo` finds the maximum discount iterating over a list of functions.

```
promos = [fidelity_promo, bulk_item_promo, large_order_promo]  ❶

def best_promo(order):  ❷
    """Select best discount available
    """
    return max(promo(order) for promo in promos)  ❸
```

❶ `promos`: list of the strategies implemented as functions.

❷ `best_promo` takes an instance of `Order` as argument, as do the other `*_promo` functions.

❸ Using a generator expression, we apply each of the functions from `promos` to the `order`, and return the maximum discount computed.

Example 6-6 is straightforward: promos is a list of functions. Once you get used to the idea that functions are first class objects, it naturally follows that building data structures holding functions often makes sense.

Although Example 6-6 works and is easy to read, there is some duplication that could lead to a subtle bug: to add a new promotion strategy we need to code the function and remember to add it to the promos list. Otherwise the new promotion will work when explicitly passed as an argument to Order, but will not be considered by best_promotion.

Read on for a couple of solutions to this issue.

Finding strategies in a module

Modules in Python are also first class objects, and the Standard Library provides several functions to handle them. The built-in globals is described as follows in the Python docs:

globals()
: Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

Here is a somewhat hackish way of using globals to help best_promo automatically find the other available *_promo functions:

Example 6-7. The promos list is built by introspection of the module global namespace.

```
promos = [globals()[name] for name in globals()   ❶
            if name.endswith('_promo')   ❷
            and name != 'best_promo']    ❸

def best_promo(order):
    """Select best discount available
    """
    return max(promo(order) for promo in promos)   ❹
```

❶ Iterate over each name in the dictionary returned by globals().

❷ Select only names that end with the _promo suffix.

❸ Filter out best_promo itself, to avoid an infinite recursion.

❹ No changes inside best_promo.

Another way of collecting the available promotions would be to create a module and put all the strategy functions there, except for best_promo.

In Example 6-8 the only significant change is that the list of strategy functions is built by introspection of a separate module called `promotions`. Note that Example 6-8 depends on importing the `promotions` module as well as `inspect`, which provides high-level introspection functions (the imports are not shown for brevity, as they would normally be at the top of the file).

Example 6-8. The `promos` list is built by introspection of a new `promotions` module.

```
promos = [func for name, func in
                inspect.getmembers(promotions, inspect.isfunction)]

def best_promo(order):
    """Select best discount available
    """
    return max(promo(order) for promo in promos)
```

The function `inspect.getmembers` returns the attributes of an object — the `promotions` module here — optionally filtered by a predicate (a boolean function). We use `inspect.isfunction` to get only the functions from the module.

Example 6-8 works regardless of the names given to the functions; all that matters is that the `promotions` module contains only functions that calculate discounts given orders. Of course, this is an implicit assumption of the code. If someone would create a function with a different signature in the `promotions` module, then `best_promo` would break while trying to apply it to an order.

We could add more stringent tests to filter the functions, by inspecting their arguments for instance. The point of Example 6-8 is not to offer a complete solution, but to highlight one possible use of module introspection.

A more explicit alternative for dynamically collecting promotional discount functions would be to use a simple decorator. We'll show yet another version of our e-commerce Strategy example in Chapter 7 which deals with function decorators.

In the next section we discuss Command — another design pattern which is sometimes implemented via single-method classes when plain functions would do.

Command

Command is another design pattern that can be simplified by the use of functions passed as arguments.

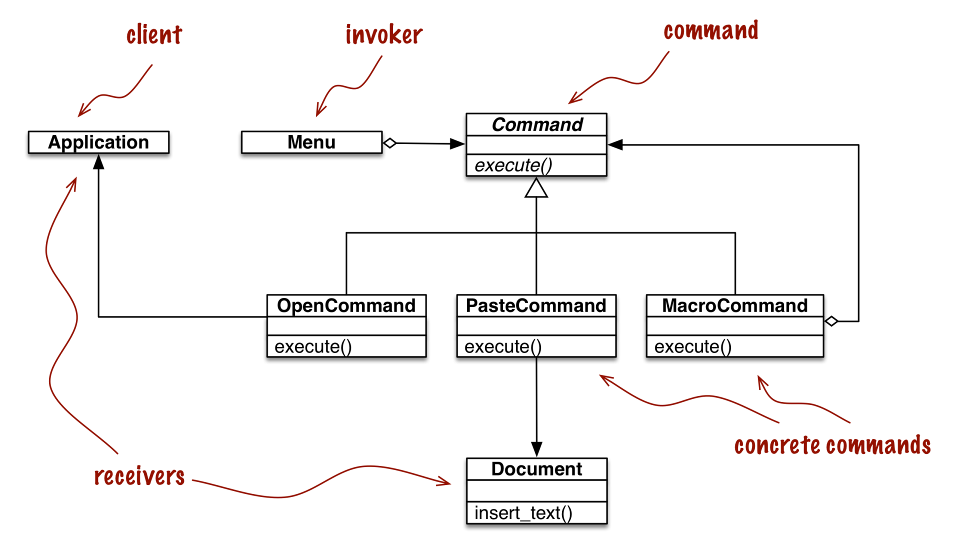


Figure 6-2. UML class diagram for menu-driven text editor implemented with the Command design pattern. Each command may have a different receiver: the object that implements the action. For PasteCommand, the receiver is the Document. For OpenCommand the receiver is the application.

The goal of Command is to decouple an object that invokes an operation (the Invoker) from the provider object that implements it (the Receiver). In the example from the Design Patterns book, each invoker is a menu item in a graphical application, and the receivers are the document being edited or the application itself.

The idea is to put a Command object between the two, implementing an interface with a single method, `execute`, which calls some method in the Receiver to perform the desired operation. That way the Invoker does not need to know the interface of the Receiver, and different receivers can be adapted through different Command subclasses. The Invoker is configured with a concrete command and calls its `execute` method to operate it. Note in Figure 6-2 that MacroCommand may store a sequence of commands; its `execute()` method calls the same method in each command stored.

Quoting from Gamma et.al. "Commands are an object-oriented replacement for callbacks." The question is: do we need a an object-oriented replacement for callbacks? Sometimes yes, but not always.

Instead of giving the Invoker a Command instance, we can simply give it a function. Instead of calling `command.execute()`, the Invoker can just call `command()`. The `MacroCommand` can be implemented with a class implementing `__call__`. Instances of `MacroCommand` would be callables, each holding a list of functions for future invocation, as implemented in Example 6-9.

Example 6-9. Each instance of `MacroCommand` has an internal `list` of commands.

```
class MacroCommand:
    """A command that executes a list of commands"""

    def __init__(self, commands):
        self.commands = list(commands)  # ❶

    def __call__(self):
        for command in self.commands:  # ❷
            command()
```

❶ Building a list from the `commands` arguments ensures that it is iterable and keeps a local copy of the command references in each `MacroCommand` instance.

❷ When an instance of `MacroCommand` is invoked, each command in `self.commands` is called in sequence.

More advanced uses of the Command pattern, to support undo, for example, may require more than a simple callback function. Even then, Python provides a couple of alternatives that deserve consideration:

- a callable instance like MacroCommand in Example 6-9 can keep whatever state is necessary, and provide extra methods in addition to `__call__`.
- a closure can be used to hold the internal state of a function between calls.

This concludes our rethinking of the Command pattern with first-class functions. At a high level, the approach here was similar to the one we applied to Strategy: replacing with callables the instances of a participant class that implemented a single-method interface. After all, every Python callable implements a single-method interface, and that method is named `__call__`.

Chapter summary

As Peter Norvig pointed out a couple of years after the classic Design Patterns book appeared, "16 of 23 patterns have qualitatively simpler implementation in Lisp or Dylan than in C++ for at least some uses of each pattern" (slide 9 of Norvig's Design Patterns in Dynamic Languages presentation). Python shares some of the dynamic features of the Lisp and Dylan languages, in particular first-class functions, our focus in this part of the book.

Reflecting on the 20th anniversary of *Design Patterns: Elements of Reusable Object-Oriented Software.*, Ralph Johnson has stated that one of the failings of the book is "Too much emphasis on patterns as end-points instead of steps in the design patterns[6].” In this chapter we used the Strategy pattern as a starting point: a working solution which we could simplify using first-class functions.

In many cases, functions or callable objects provide a more natural way of implementing callbacks in Python than mimicking the Strategy or the Command patterns as described by the Gamma, Helm, Johnson and Vlissides. The refactoring of Strategy and the discussion of Command in this chapter are examples of a more general insight: sometimes you may encounter a design pattern or an API that requires that components implement an interface with a single method, and that method has a generic sounding name such as “execute”, “run” or “doIt”. Such patterns or APIs often can be implemented with less boilerplate code in Python using first-class functions or other callables.

The message from Peter Norvig’s design patterns slides is that the Command and Strategy patterns — along with Template Method and Visitor — can be made simpler or even “invisible” with first class functions, at least for some applications of these patterns.

Further reading

Our discussion of Strategy ended with a suggestion that function decorators could be used to improve on Example 6-8. We also mentioned the use of closures a couple of times in this chapter. Decorators as well as closures are the focus of Chapter 7. That chapter starts with a refactoring of the e-commerce example using a decorator to register available promotions.

Recipe 8.21. “Implementing the Visitor Pattern” in the Python Cookbook, 3rd. edition (O’Reilly, 2013), by David Beazley and Brian K. Jones, presents an elegant implementation of the Visitor pattern in which a `NodeVisitor` class handles methods as first class objects.

On the general topic of Design Patterns, the choice of readings for the Python programmer is not as broad as what is available to other language communities.

As far as I know, Learning Python Design Patterns, by Gennadiy Zlobin (Packt, 2013), is the only book entirely devoted to patterns in Python — as of June 2014. But Zlobin’s work is quite short (100 pages) and covers eight of the original 23 design patterns.

Expert Python Programming, by Tarek Ziadé (Packt, 2008) is one of the best intermediate-level Python books in the market, and its final chapter — Useful Design Patterns — presents seven of the classic patterns from a Pythonic perspective.

[6] From the same talk quoted at the start of this chapter: *Root cause analysis of some faults in Design Patterns* presented by Johnson at IME-USP, Nov. 15. 2014.

Alex Martelli has given several talks about Python Design Patterns. There is a video of his EuroPython 2011 presentation and a set of slides in his personal Web site. I've found different slide decks and videos over the years, of varying lengths, so it is worthwhile to do a thorough search for his name with the words "Python Design Patterns".

Around 2008 Bruce Eckel — author of the excellent Thinking in Java — started a book titled Python 3 Patterns, Recipes and Idioms. It was to be written by a community of contributors led by Eckel, but six years later it's still incomplete and apparently stalled (as I write this, the last change to the repository is two years old).

There are many books about design patterns in the context of Java, but among them the one I like most is Head First Design Patterns by Eric Freeman, Bert Bates, Kathy Sierra and Elisabeth Robson (O'Reilly, 2004). It explains 16 of the 23 classic patterns. If you like the wacky style of the Head First series and need an introduction to this topic, you will love that work. However, it is Java-centric.

For a fresh look at patterns from the point of view of a dynamic language with duck typing and first-class functions, Design Patterns in Ruby by Russ Olsen (Addison-Wesley, 2007) has many insights that are also applicable to Python. In spite of many the syntactic differences, at the semantic level Python and Ruby are closer to each other than to Java or C++.

In Design Patterns in Dynamic Languages (slides), Peter Norvig shows how first-class functions (and other dynamic features) make several of the original design patterns either simpler or unnecessary.

Of course, the original Design Patterns book by Gamma et al. is mandatory reading if you are serious about this subject. The Introduction by itself is worth the price. That is the source of the often quoted design principles "Program to an interface, not an implementation." and "Favor object composition over class inheritance."

Soapbox

Python has first-class functions and first-class types, features that Norvig claims affect 10 of the 23 patterns (slide 10 of Design Patterns in Dynamic Languages). In the next chapter we'll see that Python also has generic functions ("Generic functions with single dispatch" on page 202), similar to the CLOS multi-methods that Gamma et.al. suggest as a simpler way to implement the classic Visitor pattern. Norvig, on the other hand, says that multi-methods simplify the Builder pattern (slide 10). Matching design patterns to language features is not an exact science.

In classrooms around the world Design Patterns are frequently taught using Java examples. I've heard more than one student claim that they were led to believe that the original design patterns are useful in any implementation language. It turns out that the "classic" 23 patterns from the Gamma et.al. book apply to "classic" Java very well in spite of being originally presented mostly in the context of C++ — a few have Smalltalk examples in the book. But that does not mean every one of those patterns applies equally well in any language. The authors are explicit right at the beginning of their book that "some of our patterns are supported directly by the less common object-oriented languages" (recall full quote on first page of this chapter).

The Python bibliography about design patterns is very thin, compared to that of Java, C++ or Ruby. In "Further reading" on page 180 I mentioned "Learning Python Design Patterns", by Gennadiy Zlobin, which was published as recently as November 2013. In contrast, Russ Olsen's Design Patterns in Ruby was published in 2007 and has 384 pages — 284 more than Zlobin's work.

Now that Python is becoming increasingly popular in academia, let's hope more will be written about design patterns in the context of this language. Also, Java 8 introduced method references and anonymous functions, and those highly anticipated features are likely to prompt fresh approaches to patterns in Java — recognizing that as languages evolve, so must our understanding of how to apply the classic design patterns.

CHAPTER 7
Function decorators and closures

> There's been a number of complaints about the choice of the name "decorator" for this feature. The major one is that the name is not consistent with its use in the GoF book[1]. The name *decorator* probably owes more to its use in the compiler area — a syntax tree is walked and annotated.
>
> — PEP 318 — Decorators for Functions and Methods

Function decorators let us "mark" functions in the source code to enhance their behavior is some way. This is powerful stuff, but mastering it requires understanding closures.

One of the newest reserved keywords in Python is `nonlocal`, introduced in Python 3.0. You can have a profitable life as a Python programmer without ever using it if you adhere to a strict regimen of class-centered object orientation. However, if you want to implement your own function decorators, you must know closures inside out, and then the need for `nonlocal` becomes obvious.

Aside from their application in decorators, closures are also essential for effective asynchronous programming with callbacks, and for coding in a functional style whenever it makes sense.

The end goal of this chapter is to explain exactly how function decorators work, from the simplest registration decorators to the rather more complicated parametrized ones. However, before we reach that goal we need to cover:

- How Python evaluates decorator syntax
- How Python decides whether a variable is local
- Why closures exist and how they work

[1] That's the 1995 "Design Patterns" book by the so-called Gang of Four.

- What problem is solved by `nonlocal`

With this grounding we can tackle further decorator topics:

- Implementing a well-behaved decorator
- Interesting decorators in the standard library
- Implementing a parametrized decorator

We start with a very basic introduction to decorators, and then proceed with the rest of the items above[2].

Decorators 101

A decorator is a callable that takes another function as argument (the decorated function)[3]. The decorator may perform some processing with the decorated function, and returns it or replaces it with another function or callable object.

In other words, this code:

```
@decorate
def target():
    print('running target()')
```

Has the same effect as writing this:

```
def target():
    print('running target()')

target = decorate(target)
```

The end result is the same: at the end of either of these snippets, the `target` name does not necessarily refer to the original `target` function, but to whatever function is returned by `decorate(target)`.

To confirm that the decorated function is replaced, see the console session in Example 7-1.

Example 7-1. A decorator usually replaces a function with a different one.

```
>>> def deco(func):
...     def inner():
...         print('running inner()')
...     return inner  ❶
...
```

[2] The next section is named after the academic tradition of naming introductory courses like “Calculus 101”. Function decorators are way easier than calculus, though.

[3] Python also supports class decorators. They are covered in Chapter 21

```
>>> @deco
... def target():  ❷
...     print('running target()')
...
>>> target()  ❸
running inner()
>>> target  ❹
<function deco.<locals>.inner at 0x10063b598>
```

❶ deco returns its `inner` function object.

❷ `target` is decorated by `deco`.

❸ Invoking the decorated `target` actually runs `inner`.

❹ Inspection reveals that `target` is a now a reference to `inner`.

Strictly speaking, decorators are just syntactic sugar. As we just saw, you can always simply call a decorator like any regular callable, passing another function. Sometimes that is actually convenient, especially when doing *metaprogramming* — changing program behavior at run-time.

To summarize: the first crucial fact about decorators is that they have the power to replace the decorated function with a different one. The second crucial fact is that they are executed immediately when a module is loaded. This is explained next.

When Python executes decorators

A key feature of decorators is that they run right after the decorated function is defined. That is usually at *import time*, i.e. when a module is loaded by Python. Consider `registration.py` in Example 7-2.

Example 7-2. The `registration.py` module

```
registry = []  ❶

def register(func):  ❷
    print('running register(%s)' % func)  ❸
    registry.append(func)  ❹
    return func  ❺

@register  ❻
def f1():
    print('running f1()')

@register
def f2():
    print('running f2()')

def f3():  ❼
```

```
    print('running f3()')

def main():  ❽
    print('running main()')
    print('registry ->', registry)
    f1()
    f2()
    f3()

if __name__=='__main__':
    main()  ❾
```

❶ `registry` will hold references to functions decorated by `@register`

❷ `register` takes a function as argument.

❸ Display what function is being decorated, for demonstration.

❹ Include `func` in `registry`.

❺ Return `func`: we must return a function, here we return the same received as argument.

❻ `f1` and `f2` are decorated by `@register`.

❼ `f3` is not decorated.

❽ `main` displays the `registry`, then calls `f1()`, `f2()` and `f3()`.

❾ `main()` is only invoked if `registration.py` runs as a script.

The output of running `registration.py` as a script looks like this:

```
$ python3 registration.py
running register(<function f1 at 0x100631bf8>)
running register(<function f2 at 0x100631c80>)
running main()
registry -> [<function f1 at 0x100631bf8>, <function f2 at 0x100631c80>]
running f1()
running f2()
running f3()
```

Note that `register` runs (twice) before any other function in the module. When `register` is called, it receives as argument the function object being decorated, eg. `<function f1 at 0x100631bf8>`.

After the module is loaded, the `registry` holds references to the two decorated functions: `f1` and `f2`. These functions, as well as `f3`, are only executed when explicitly called by `main`.

If `registration.py` is imported (and not run as a script), the output is this:

```
>>> import registration
running register(<function f1 at 0x10063b1e0>)
running register(<function f2 at 0x10063b268>)
```

At this time, if you look at the `registry`, here is what you get:

```
>>> registration.registry
[<function f1 at 0x10063b1e0>, <function f2 at 0x10063b268>]
```

The main point of Example 7-2 is to emphasize that function decorators are executed as soon as the module is imported, but the decorated functions only run when they are explicitly invoked. This highlights the difference between what Pythonistas call *import time* and *run time*.

Considering how decorators are commonly employed in real code, Example 7-2 is unusual in two ways:

- The decorator function is defined in the same module as the decorated functions. A real decorator is usually defined in one module and applied to functions in other modules.
- The `register` decorator returns the same function passed as argument. In practice, most decorators define an inner function and return it.

Even though the `register` decorator in Example 7-2 returns the decorated function unchanged, that technique is not useless. Similar decorators are used in many Python Web frameworks to add functions to some central registry, for example, a registry mapping URL patterns to functions that generate HTTP responses. Such registration decorators may or may not change the decorated function. The next section shows a practical example.

Decorator-enhanced Strategy pattern

A registration decorator is a good enhancement to the e-commerce promotional discount from "Case study: refactoring Strategy" on page 168.

Recall that our main issue with Example 6-6 is the repetition of the function names in their definitions and then in the `promos` list used by the `best_promo` function to determine the highest discount applicable. The repetition is problematic because someone may add a new promotional strategy function and forget to manually add it to the `promos` list — in which case `best_promo` will silently ignore the new strategy, introducing a subtle bug in the system. Example 6-8 solves this problem with a registration decorator.

Example 7-3. The `promos` list is filled by the `promotion` decorator.

```
promos = []  ❶

def promotion(promo_func):  ❷
```

```
    promos.append(promo_func)
    return promo_func

@promotion    ❸
def fidelity(order):
    """5% discount for customers with 1000 or more fidelity points"""
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0

@promotion
def bulk_item(order):
    """10% discount for each LineItem with 20 or more units"""
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * .1
    return discount

@promotion
def large_order(order):
    """7% discount for orders with 10 or more distinct items"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * .07
    return 0

def best_promo(order):    ❹
    """Select best discount available
    """
    return max(promo(order) for promo in promos)
```

❶ The `promos` list starts empty.

❷ `promotion` decorator returns `promo_func` unchanged, after adding it to the `promos` list.

❸ Any function decorated by `@promotion` will be added to `promos`.

❹ No changes needed to `best_promos`, as it relies on the `promos` list.

This solution has several advantages over the others presented in "Case study: refactoring Strategy" on page 168:

- The promotion strategy functions don't have to use special names (like the `_promo` suffix).
- The `@promotion` decorator highlights the purpose of the decorated function, and also makes it easy to temporarily disable a promotion: just comment out the decorator.

- Promotional discount strategies may be defined in other modules, anywhere in the system, as long as the `@promotion` decorator is applied to them.

Most decorators do change the decorated function. They usually do it by defining an inner function and returning it to replace the decorated function. Code that uses inner functions almost always depends on closures to operate correctly. To understand closures, we need to take a step back a have a close look at how variable scopes work in Python.

Variable scope rules

In Example 7-4 we define and test a function which reads two variables: a local variable `a`, defined as function parameter, and variable `b` that is not defined anywhere in the function:

Example 7-4. Function reading a local and a global variable.

```
>>> def f1(a):
...     print(a)
...     print(b)
...
>>> f1(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f1
NameError: global name 'b' is not defined
```

The error we got is not surprising. Continuing from Example 7-4, if we assign a value to a global `b` and then call `f1`, it works:

```
>>> b = 6
>>> f1(3)
3
6
```

Now, let's see an example that may surprise you.

Take a look at the `f2` function in Example 7-5. Its first two lines are the same as `f1` in Example 7-4, then it makes an assignment to `b`, and prints its value. But it fails at the second `print`, before the assignment is made:

Example 7-5. Variable `b` is local, because it is assigned a value in the body of the function.

```
>>> b = 6
>>> def f2(a):
...     print(a)
...     print(b)
...     b = 9
```

```
...
>>> f2(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f2
UnboundLocalError: local variable 'b' referenced before assignment
```

Note that the output starts with 3, which proves that the print(a) statement was executed. But the second one, print(b) never runs. When I first saw this I was surprised, thinking that 6 should be printed, because there is a global variable b and the assignment to the local b is made after print(b).

But the fact is, when Python compiles the body of the function, it decides that b is a local variable because it is assigned within the function. The generated bytecode reflects this decision and will try to fetch b from the local environment. Later, when the call f2(3) is made, the body of f2 fetches and prints the value of the local variable a, but when trying to fetch the value of local variable b it discovers that b is unbound.

This is not a bug, but a design choice: Python does not require you to declare variables, but assumes that a variable assigned in the body of a function is local. This is much better than the behavior of JavaScript, which does not require variable declarations either, but if you do forget to declare that a variable is local (with var), you may clobber a global variable without knowing.

If we want the interpreter to treat b as a global variable in spite of the assignment within the function, we use the global declaration:

```
>>> def f3(a):
...     global b
...     print(a)
...     print(b)
...     b = 9
...
>>> f3(3)
3
6
>>> b
9

>>> f3(3)
a = 3
b = 8
b = 30
>>> b
30
>>>
```

After this closer look at how variable scopes work in Python, we can tackle closures in the next section, "Closures" on page 192. If you are curious about the bytecode differences between the functions in Example 7-4 and Example 7-5, see the following sidebar.

Comparing bytecodes

The `dis` module provides an easy way to disassemble the bytecode of Python functions. Read Example 7-6 and Example 7-7 to see the bytecodes for `f1` and `f2` from Example 7-4 and Example 7-5.

Example 7-6. Disassembly of the `f1` function from Example 7-4.

```
>>> from dis import dis
>>> dis(f1)
  2           0 LOAD_GLOBAL              0 (print)  ❶
              3 LOAD_FAST                0 (a)  ❷
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 POP_TOP

  3          10 LOAD_GLOBAL              0 (print)
             13 LOAD_GLOBAL              1 (b)  ❸
             16 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             19 POP_TOP
             20 LOAD_CONST               0 (None)
             23 RETURN_VALUE
```

❶ Load global name `print`.

❷ Load local name `a`.

❸ Load global name `b`.

Contrast the bytecode for `f1` shown in Example 7-6 with the bytecode for `f2` in Example 7-6.

Example 7-7. Disassembly of the `f2` function from Example 7-5.

```
>>> dis(f2)
  2           0 LOAD_GLOBAL              0 (print)
              3 LOAD_FAST                0 (a)
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 POP_TOP

  3          10 LOAD_GLOBAL              0 (print)
             13 LOAD_FAST                1 (b)  ❶
             16 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             19 POP_TOP

  4          20 LOAD_CONST               1 (9)
             23 STORE_FAST               1 (b)
             26 LOAD_CONST               0 (None)
             29 RETURN_VALUE
```

❶ Load *local* name `b`. This shows that the compiler considers `b` a local variable, even if the assignment to `b` occurs later, because the nature of the variable — whether it is local or not — cannot change body of the function.

The CPython VM that runs the bytecode is a stack machine, so the operations `LOAD` and `POP` refer to the stack. It is beyond the scope of this book to further describe the Python opcodes, but they are documented along with the `dis` module in dis — Disassembler for Python bytecode

Closures

In the blogosphere closures are sometimes confused with anonymous functions. The reason why many confuse them is historic: defining functions inside functions is not so common, until you start using anonymous functions. And closures only matter when you have nested functions. So a lot of people learn both concepts at the same time.

Actually, a closure is function with an extended scope that encompasses non-global variables referenced in the body of the function but not defined there. It does not matter whether the function is anonymous or not, what matters is that it can access non-global variables that are defined outside of its body.

This is a challenging concept to grasp, and is better approached through an example.

Consider an `avg` function to compute the mean of an ever-increasing series of values, for example, the average closing price of a commodity over its entire history. Every day a new price is added, and the average is computed taking into account all prices so far.

Starting with a clean slate, this is how `avg` could be used:

```
    >>> avg(10)
    10.0
    >>> avg(11)
    10.5
    >>> avg(12)
    11.0
```

Where does `avg` come from, and where does it keep the history of previous values?

For starters, Example 7-8 is a class-based implementation:

Example 7-8. `average_oo.py`: A class to calculate a running average.

```
class Averager():

    def __init__(self):
        self.series = []

    def __call__(self, new_value):
        self.series.append(new_value)
        total = sum(self.series)
        return total/len(self.series)
```

The `Averager` class creates instances that are callable:

```
    >>> avg = Averager()
    >>> avg(10)
    10.0
    >>> avg(11)
    10.5
    >>> avg(12)
    11.0
```

Now, Example 7-9 is a functional implementation, using a higher order function `make_averager`:

Example 7-9. `average.py`: a higher-order function to calculate a running average.

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```

When invoked, `make_averager` returns an `averager` function object. Each time an `averager` is called, it appends the passed argument to the series, and computes the current average:

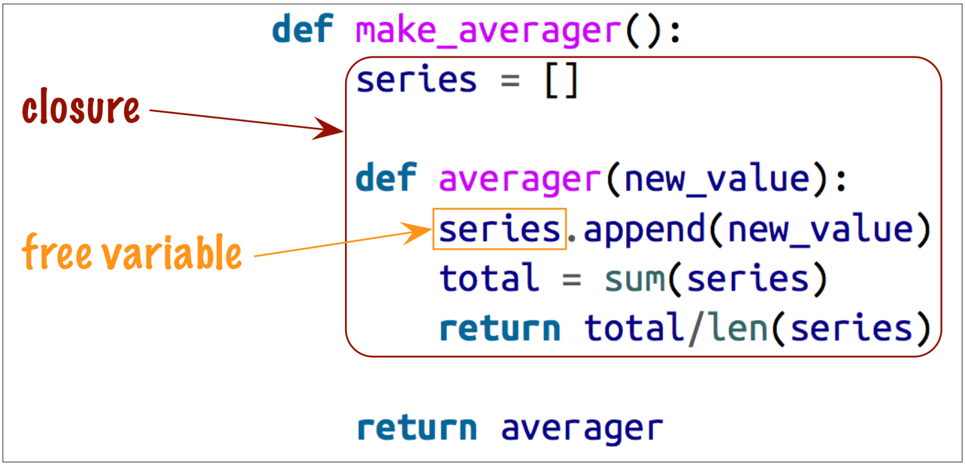
Example 7-10. Testing Example 7-9

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

Note the similarities of the examples: we call `Averager()` or `make_averager()` to get a callable object `avg` that will update the historical series and calculate the current mean. In Example 7-8, `avg` is an instance of `Averager`, and in Example 7-9 it is the inner function, `averager`. Either way, we just call `avg(n)` to include `n` in the series and get the updated mean.

It's obvious where the `avg` of the `Averager` class keeps the history: the `self.series` instance attribute. But where does the `avg` function in the second example find the `series`?

Note that `series` is a local variable of `make_averager` because the initialization `series = []` happens in the body of that function. But when `avg(10)` is called, `make_averager` has already returned, its local scope is long gone.



```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```

Figure 7-1. The closure for `averager` extends the scope of that function to include the binding for the free variable `series`.

Within `averager`, `series` is a *free variable*. This is a technical term meaning a variable that is not bound in the local scope. See Figure 7-1.

Inspecting the returned `averager` object shows how Python keeps the names of local and free variables in the `__code__` attribute that represents the compiled body of the function:

Example 7-11. Inspecting the function created by `make_averager` in Example 7-9.

```
>>> avg.__code__.co_varnames
('new_value', 'total')
>>> avg.__code__.co_freevars
('series',)
```

The binding for `series` is kept in the `__closure__` attribute of the returned function `avg`. Each item in `avg.__closure__` corresponds to a name in `avg.__code__.co_freevars`. These items are `cells`, and they have an attribute `cell_contents` where the actual value can be found:

Example 7-12. Continuing from Example 7-10.

```
>>> avg.__code__.co_freevars
('series',)
>>> avg.__closure__
(<cell at 0x107a44f78: list object at 0x107a91a48>,)
>>> avg.__closure__[0].cell_contents
[10, 11, 12]
```

To summarize: a closure is function that retains the bindings of the free variables that exist when the function is defined, so that they can be used later when the function is invoked and the defining scope is no longer available.

Note that the only situation in which a function may need to deal with external variables that are non-global is when it is nested in another function.

The `nonlocal` declaration

Our previous implementation of `make_averager` was not efficient. In Example 7-9 we stored all the values in the historical series and computed their `sum` every time `averager` was called. A better implementation would just store the total and the number of items so far, and compute the mean from these two numbers.

Example 7-13 is a broken implementation, just to make a point. Can you see where it breaks?

Example 7-13. A broken higher-order function to calculate a running average without keeping all history.

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        count += 1
        total += new_value
        return total / count

    return averager
```

If you try Example 7-13, here is what you get:

```
    >>> avg = make_averager()
    >>> avg(10)
    Traceback (most recent call last):
      ...
    UnboundLocalError: local variable 'count' referenced before assignment
    >>>
```

The problem is that the statement `count += 1` actually means the same as `count = count + 1`, when `count` is a number or any immutable type. So we are actually assigning to `count` in the body of `averager`, and that makes it a local variable. The same problem affects the `total` variable.

We did not have this problem in Example 7-9 because we never assigned to the `series` list, we only called `series.append` and invoked `sum` and `len` on it. So we took advantage of the fact that lists are mutable.

But with immutable types like numbers, strings, tuples etc., all you can is read, but never update. If you try to rebind them, as in `count = count + 1`, then you are implicitly creating a local variable `count`. It is no longer a free variable, therefore it is not saved in the closure.

To work around this the `nonlocal` declaration was introduced in Python 3. It lets you flag a variable as a free variable even when it is assigned a new value within the function. If a new value is assigned to a `nonlocal` variable, the binding stored in the closure is changed. A correct implementation of our newest `make_averager` looks like Example 7-14:

Example 7-14. Calculate a running average without keeping all history. Fixed with the use of `nonlocal`.

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal count, total
        count += 1
        total += new_value
        return total / count

    return averager
```

Getting by without `nonlocal` in Python 2

The lack of `nonlocal` in Python 2 requires workarounds, one of which is described in the third code snippet of PEP 3104 — Access to Names in Outer Scopes, which introduced `nonlocal`. Essentially the idea is to store the variables the inner functions need to change (eg. `count`, `total`) as items or attributes of some mutable object, like a `dict` or a simple instance, and bind that object to a free variable.

Now that we got Python closures covered, we can effectively implement decorators with nested functions.

Implementing a simple decorator

Example 7-15 is a decorator that clocks every invocation of the decorated function and prints the elapsed time, the arguments passed and the result of the call.

Example 7-15. A simple decorator to output the running time of functions.

```
import time

def clock(func):
    def clocked(*args):  # ❶
        t0 = time.perf_counter()
        result = func(*args)  # ❷
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_str = ', '.join(repr(arg) for arg in args)
        print('[%0.8fs] %s(%s) -> %r' % (elapsed, name, arg_str, result))
        return result
    return clocked  # ❸
```

❶ Define inner function `clocked` to accept any number of positional arguments.

❷ This line only works because the closure for `clocked` encompasses the `func` free variable.

❸ Return the inner function to replace the decorated function.

This script demonstrates the use of the `clock` decorator:

Example 7-16. A simple decorator to output the running time of functions.

```
# clockdeco_demo.py

import time
from clockdeco import clock

@clock
def snooze(seconds):
    time.sleep(seconds)

@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

if __name__=='__main__':
    print('*' * 40, 'Calling snooze(.123)')
    snooze(.123)
    print('*' * 40, 'Calling factorial(6)')
    print('6! =', factorial(6))
```

The output of running Example 7-16 looks like:

```
$ python3 clockdeco_demo.py
**************************************** Calling snooze(123)
[0.12405610s] snooze(.123) -> None
**************************************** Calling factorial(6)
[0.00000191s] factorial(1) -> 1
[0.00004911s] factorial(2) -> 2
[0.00008488s] factorial(3) -> 6
[0.00013208s] factorial(4) -> 24
[0.00019193s] factorial(5) -> 120
[0.00026107s] factorial(6) -> 720
6! = 720
```

How it works

Remember that this code:

```
@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)
```

Actually does this:

```
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

factorial = clock(factorial)
```

So, in both examples above, `clock` gets the `factorial` function as its `func` argument (see Example 7-15). It then creates and returns the `clocked` function, which the Python interpreter assigns to `factorial` behind the scenes. In fact, if you import the `clockdeco_demo` module and check the `__name__` of `factorial`, this is what you get:

```
>>> import clockdeco_demo
>>> clockdeco_demo.factorial.__name__
'clocked'
>>>
```

So `factorial` now actually holds a reference to the `clocked` function. From now on, each time `factorial(n)` is called, `clocked(n)` gets executed. In essence, `clocked` does the following:

1. records the initial time `t0`
2. calls the original `factorial`, saving the result
3. computes the elapsed time
4. formats and prints the collected data
5. returns the result saved in step 2

This is the typical behavior of a decorator: it replaces the decorated function with a new function that accepts the same arguments and (usually) returns whatever the decorated function was supposed to return, while also doing some extra processing.

> In the Design Patterns book by Gamma et.al., the short description of the Decorator pattern starts with: "Attach additional responsibilities to an object dynamically." Function decorators fit that description. But at the implementation level, Python decorators bear little resemblance to the classic Decorator described in the original Design Patterns work. The **Soapbox** section at the end of this chapter has more on this subject (see "Soapbox" on page 213).

The `clock` decorator implemented in Example 7-15 has a few shortcomings: it does not support keyword arguments, and it masks the `__name__` and `__doc__` of the decorated function. Example 7-17 uses the `functools.wraps` decorator to copy the relevant attributes from `func` to `clocked`. Also, in this new version keyword arguments are correctly handled.

Example 7-17. An improved `clock` decorator.

```python
# clockdeco2.py

import time
import functools

def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        t0 = time.time()
        result = func(*args, **kwargs)
        elapsed = time.time() - t0
        name = func.__name__
        arg_lst = []
        if args:
            arg_lst.append(', '.join(repr(arg) for arg in args))
        if kwargs:
            pairs = ['%s=%r' % (k, w) for k, w in sorted(kwargs.items())]
            arg_lst.append(', '.join(pairs))
        arg_str = ', '.join(arg_lst)
        print('[%0.8fs] %s(%s) -> %r ' % (elapsed, name, arg_str, result))
        return result
    return clocked
```

`functools.wraps` is just one of the ready-to-use decorators in the standard library. In the next section we'll meet two of the most impressive decorators that `functools` provides: `lru_cache` and `singledispatch`.

Decorators in the standard library

Python has three built-in functions that are designed to decorate methods: property, classmethod and staticmethod. We will discuss property in "Using a property for attribute validation" on page 606 and the others in "classmethod versus staticmethod" on page 252.

Another frequently seen decorator is functools.wraps, a helper for building well-behaved decorators. We used it in Example 7-17. Two of the most interesting decorators in the standard library are lru_cache and the brand-new singledispatch (added in Python 3.4). Both are defined in the functools module. We'll cover them next.

Memoization with functools.lru_cache

A very practical decorator is functools.lru_cache. It implements memoization: an optimization technique which works by saving the results of previous invocations of an expensive function, avoiding repeat computations on previously used arguments. The letters LRU stand for Least Recently Used, meaning that the growth of the cache is limited by discarding the entries that have not been read for a while.

A good demonstration is to apply lru_cache to the painfully slow recursive function to generate the Nth number in the Fibonacci sequence.

Example 7-18. The very costly recursive way to compute the Nth number in the Fibonacci series.

```
from clockdeco import clock

@clock
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

if __name__=='__main__':
    print(fibonacci(6))
```

Below is result of running fibo_demo.py. Except for the last line, all output is generated by clock decorator:

```
$ python3 fibo_demo.py
[0.00000095s] fibonacci(0) -> 0
[0.00000095s] fibonacci(1) -> 1
[0.00007892s] fibonacci(2) -> 1
[0.00000095s] fibonacci(1) -> 1
[0.00000095s] fibonacci(0) -> 0
[0.00000095s] fibonacci(1) -> 1
[0.00003815s] fibonacci(2) -> 1
[0.00007391s] fibonacci(3) -> 2
```

```
    [0.00018883s] fibonacci(4) -> 3
    [0.00000000s] fibonacci(1) -> 1
    [0.00000095s] fibonacci(0) -> 0
    [0.00000119s] fibonacci(1) -> 1
    [0.00004911s] fibonacci(2) -> 1
    [0.00009704s] fibonacci(3) -> 2
    [0.00000000s] fibonacci(0) -> 0
    [0.00000000s] fibonacci(1) -> 1
    [0.00002694s] fibonacci(2) -> 1
    [0.00000095s] fibonacci(1) -> 1
    [0.00000095s] fibonacci(0) -> 0
    [0.00000095s] fibonacci(1) -> 1
    [0.00005102s] fibonacci(2) -> 1
    [0.00008917s] fibonacci(3) -> 2
    [0.00015593s] fibonacci(4) -> 3
    [0.00029993s] fibonacci(5) -> 5
    [0.00052810s] fibonacci(6) -> 8
    8
```

The waste is obvious: `fibonacci(1)` is called 8 times, `fibonacci(2)` 5 times etc. But if we just add two lines to use `lru_cache`, performance is much improved. See Example 7-19.

Example 7-19. Faster implementation using caching.

```
import functools

from clockdeco import clock

@functools.lru_cache() # ❶
@clock  # ❷
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

if __name__=='__main__':
    print(fibonacci(6))
```

❶ Note that `lru_cache` must be invoked as a regular function — note the parenthesis in the line: `@functools.lru_cache()`. The reason is that it accepts configuration parameters, as we'll see shortly.

❷ This is an example of stacked decorators: `@lru_cache()` is applied on the function returned by `@clock`.

Execution time is halved, and the function is called only once for each value of `n`.

```
    $ python3 fibo_demo_lru.py
    [0.00000119s] fibonacci(0) -> 0
    [0.00000119s] fibonacci(1) -> 1
    [0.00010800s] fibonacci(2) -> 1
```

```
[0.00000787s] fibonacci(3) -> 2
[0.00016093s] fibonacci(4) -> 3
[0.00001216s] fibonacci(5) -> 5
[0.00025296s] fibonacci(6) -> 8
```

In another test, to compute `fibonacci(30)`, Example 7-19 made the 31 calls needed in 0.0005s, while the uncached Example 7-18 called `fibonacci` 2,692,537 times and took 17.7 seconds in an Intel Core i7 notebook.

Besides making silly recursive algorithms viable, `lru_cache` really shines in applications that need to fetch information from Web.

It's important to note that `lru_cache` can be tuned by passing two optional arguments. Its full signature is:

```
functools.lru_cache(maxsize=128, typed=False)
```

The `maxsize` arguments determines how many call results are stored. After the cache is full, older results are discarded to make room. For optimal performance, `maxsize` should be a power of 2. The `typed` argument, if set to `True`, stores results of different argument types separately, i.e. distinguishing between float and integer arguments that are normally considered equal, like `1` and `1.0`. By the way, because `lru_cache` uses a `dict` to store the results, and the keys are made from the positional and keyword arguments used in the calls, all the arguments taken by the decorated function must be *hashable*.

Now let us consider the intriguing `functools.singledispatch` decorator.

Generic functions with single dispatch

Imagine we are creating a tool to debug Web applications. We want to be able to generate HTML displays for different types of Python objects.

We could start with a function like this:

```
import html

def htmlize(obj):
    content = html.escape(repr(obj))
    return '<pre>{}</pre>'.format(content)
```

That will work for any Python type, but now we want to extend it generate custom displays for some types:

- `str`: replace embedded newline characters with `'<br>\n'` and use `<p>` tags instead of `<pre>`.
- `int`: show the number in decimal and hexadecimal.

- `list`: output an HTML list, formatting each item according to its type.

The behavior we want is shown in Example 7-20.

Example 7-20. `htmlize` generates HTML tailored to different object types.

```
>>> htmlize({1, 2, 3})   ❶
'<pre>{1, 2, 3}</pre>'
>>> htmlize(abs)
'<pre>&lt;built-in function abs&gt;</pre>'
>>> htmlize('Heimlich & Co.\n- a game')   ❷
'<p>Heimlich &amp; Co.<br>\n- a game</p>'
>>> htmlize(42)   ❸
'<pre>42 (0x2a)</pre>'
>>> print(htmlize(['alpha', 66, {3, 2, 1}]))   ❹
<ul>
<li><p>alpha</p></li>
<li><pre>66 (0x42)</pre></li>
<li><pre>{1, 2, 3}</pre></li>
</ul>
```

❶ By default, the HTML-escaped `repr` of an object is shown enclosed in `<pre></pre>`.

❷ `str` objects are also HTML-escaped but wrapped in `<p></p>` with `<br>` line breaks.

❸ An `int` is shown in decimal and hexadecimal, inside `<pre></pre>`

❹ Each list item is formatted according to its type, and the whole sequence rendered as an HTML list.

Since we don't have method or function overloading in Python, we can't create variations of `htmlize` with different signatures for each data type we want to handle differently. A common solution in Python would be to turn `htmlize` into a dispatch function, with a chain of `if/elif/elif` calling specialized functions like `htmlize_str`, `htmlize_int` etc. This is not extensible by users of our module, and is unwieldy: over time, the `htmlize` dispatcher would become too big, and the coupling between it and the specialized functions would be very tight.

The new `functools.singledispatch` decorator in Python 3.4 allows each module to contribute to the overall solution, and lets you easily provide a specialized function even for classes that you can't edit. If you decorate a plain function with `@singledispatch` it becomes a *generic function*: a group of functions to perform the same operation in different ways, depending on the type of the first argument[4]. Example 7-21 shows how.

[4] This is what is meant by the term single-dispatch. If more arguments were used to select the specific functions, we'd have multiple-dispatch.

`functools.singledispatch` was added in Python 3.4, but the `singledispatch` package available on PyPI is a backport compatible with Python 2.6 to 3.3.

Example 7-21. `singledispatch` creates a custom `htmlize.register` to bundle several functions into a generic function.

```
from functools import singledispatch
from collections import abc
import numbers
import html

@singledispatch  ❶
def htmlize(obj):
    content = html.escape(repr(obj))
    return '<pre>{}</pre>'.format(content)

@htmlize.register(str)  ❷
def _(text):            ❸
    content = html.escape(text).replace('\n', '<br>\n')
    return '<p>{0}</p>'.format(content)

@htmlize.register(numbers.Integral)  ❹
def _(n):
    return '<pre>{0} (0x{0:x})</pre>'.format(n)

@htmlize.register(tuple)  ❺
@htmlize.register(abc.MutableSequence)
def _(seq):
    inner = '</li>\n<li>'.join(htmlize(item) for item in seq)
    return '<ul>\n<li>' + inner + '</li>\n</ul>'
```

❶ `@singledispatch` marks the base function which handles the `object` type.

❷ Each specialized function is decorated with `@«base_function».register(«type»)`.

❸ The name of the specialized functions is irrelevant; `_` is a good choice to make this clear.

❹ For each additional type to receive special treatment, register a new function. `numbers.Integral` is a virtual superclass of `int` (see below).

❺ You can stack several `register` decorators to support different types with the same function.

When possible, register the specialized functions to handle ABCs (abstract classes) such as `numbers.Integral` and `abc.MutableSequence` instead of concrete implementations like `int` and `list`. This allows your code to support a greater variety of compatible types. For example, a Python extension can provide alternatives to the `int` type with fixed bit lengths as subclasses of `numbers.Integral`.

> Using ABCs for type checking allows your code to support existing or future classes that are either actual or virtual subclasses of those ABCs. The use of ABCs and the concept of a virtual subclass are subjects of Chapter 11.

A notable quality of the `singledispatch` mechanism is that you can register specialized functions anywhere in the system, in any module. If you later add a module with a new user-defined type, you can easily provide a new custom function to handle that type. And you can write custom functions for classes that you did not write and can't change.

`singledispatch` is a well thought-out addition to the standard library, and it offers more features than we can describe here. The best documentation for it is PEP 443 — Single-dispatch generic functions.

> `@singledispatch` is not designed to bring Java-style method overloading to Python. A single class with many overloaded variations of a method better than a single function with a lengthy stretch of `if/elif/elif/elif` blocks. But both solutions are flawed because they concentrate too much responsibility in a single code unit — the class or the function. The advantage of `@singledispath` is supporting modular extension: each module can register a specialized function for each type it supports.

Decorators are functions, therefore they may be composed, i.e. you can apply a decorator to a function that is already decorated, as shown in Example 7-21. The next session explains how that works.

Stacked decorators

Example 7-19 demonstrated the use of stacked decorators: `@lru_cache` is applied on the result of `@clock` over `fibonacci`. In Example 7-21 the `@htmelize.register` decorator was applied twice to the last function in the module.

When two decorators `@d1` and `@d2` are applied to a function `f` in that order, the result is the same as `f = d1(d2(f))`.

In other words, this:

```
    @d1
    @d2
    def f():
        print('f')
```

Is the same as:

```
    def f():
        print('f')

    f = d1(d2(f))
```

Besides stacked decorators, this chapter has shown some decorators that take arguments, for example, `@lru_cache()` and the `htmlize.register(«type»)` produced by `@singledispatch` in Example 7-21. The next section shows how to build decorators that accept parameters.

Parametrized Decorators

When parsing a decorator in source code, Python takes the decorated function and passes it as the first argument to the decorator function. So how do you make a decorator accept other arguments? The answer is: make a decorator factory that takes those arguments and returns a decorator, which is then applied to the function to be decorated. Confusing? Sure. Let's start with an example based on the simplest decorator we've seen: `register` in Example 7-22.

Example 7-22. Abridged `registration.py` module from Example 7-2, repeated here for convenience.

```
registry = []

def register(func):
    print('running register(%s)' % func)
    registry.append(func)
    return func

@register
def f1():
    print('running f1()')

print('running main()')
print('registry ->', registry)
f1()
```

A parametrized registration decorator

In order to make it easy to enable or disable the function registration performed by `register`, we'll make it accept an optional `active` parameter which, if `False` skips registering the decorated function. Example 7-23 shows how. Conceptually, the new register function is not a decorator but a decorator factory. When called, it returns the actual decorator that will be applied to the target function.

Example 7-23. To accept parameters, the new register decorator must be called as a function.

```
registry = set()  ❶

def register(active=True):  ❷
    def decorate(func):  ❸
        print('running register(active=%s)->decorate(%s)'
              % (active, func))
        if active:   ❹
            registry.add(func)
        else:
            registry.discard(func)  ❺

        return func  ❻
    return decorate  ❼

@register(active=False)  ❽
def f1():
    print('running f1()')

@register()  ❾
def f2():
    print('running f2()')

def f3():
    print('running f3()')
```

❶ registry is now a set, so adding and removing functions is faster.

❷ register takes an optional keyword argument.

❸ The decorate inner function is the actual decorator; note how it takes a function as argument.

❹ Register func only if the active argument (retrieved from the closure) is True.

❺ If not active and func in registry, remove it.

❻ Because decorate is a decorator, it must return a function.

❼ register is our decorator factory, so it returns decorate.

❽ The @register factory must be invoked as a function, with the desired parameters.

❾ If no parameters are passed, register must still be called as a function — @register() — to return the actual decorator, decorate.

The main point is that `register()` returns `decorate` which is then applied to the decorated function.

The code in Example 7-23 is in a `registration_param.py` module. If we import it, this is what we get:

```
>>> import registration_param
running register(active=False)->decorate(<function f1 at 0x10063c1e0>)
running register(active=True)->decorate(<function f2 at 0x10063c268>)
>>> registration_param.registry
[<function f2 at 0x10063c268>]
```

Note how only the `f2` function appears in the `registry`; `f1` is not there because `active=False` was passed to the `register` decorator factory, so the `decorate` that was applied to `f1` did not add it to the `registry`.

If, instead of using the `@` syntax, we used `register` as a regular function, the syntax needed to decorate a function `f` would be `register()(f)` to add `f` to the `registry`, or `register(active=False)(f)` to not add it (or remove it). See Example 7-24 for a demo of adding and removing functions to the `registry`.

Example 7-24. Using the `registration_param` module listed in Example 7-23.

```
>>> from registration_param import *
running register(active=False)->decorate(<function f1 at 0x10073c1e0>)
running register(active=True)->decorate(<function f2 at 0x10073c268>)
>>> registry  # ❶
{<function f2 at 0x10073c268>}
>>> register()(f3)  # ❷
running register(active=True)->decorate(<function f3 at 0x10073c158>)
<function f3 at 0x10073c158>
>>> registry  # ❸
{<function f3 at 0x10073c158>, <function f2 at 0x10073c268>}
>>> register(active=False)(f2)  # ❹
running register(active=False)->decorate(<function f2 at 0x10073c268>)
<function f2 at 0x10073c268>
>>> registry  # ❺
{<function f3 at 0x10073c158>}
```

❶ When the module is imported, `f2` is in the `registry`.

❷ The `register()` expression returns `decorate`, which is then applied to `f3`.

❸ The previous line added `f3` to the `registry`.

❹ This call removes `f2` from the `registry`.

❺ Confirm that only `f3` remains in the `registry`.

The workings of parametrized decorators are fairly involved, and the one we've just discussed is simpler than most. Parametrized decorators usually replace the decorated function, and their construction requires yet another level of nesting. Touring such function pyramids is our next adventure.

The parametrized `clock` decorator

In this section we'll revisit the `clock` decorator, adding a feature: users may pass a format string to control the output of the decorated function. See Example 7-25.

> For simplicity, Example 7-25 is based on the initial `clock` implementation from Example 7-15, and not the improved one from Example 7-17 which uses `@functools.wraps`, adding yet another function layer.

Example 7-25. Module `clockdeco_param.py`: the parametrized `clock` decorator.

```
import time

DEFAULT_FMT = '[{elapsed:0.8f}s] {name}({args}) -> {result}'

def clock(fmt=DEFAULT_FMT):   ❶
    def decorate(func):       ❷
        def clocked(*_args):  ❸
            t0 = time.time()
            _result = func(*_args)   ❹
            elapsed = time.time() - t0
            name = func.__name__
            args = ', '.join(repr(arg) for arg in _args)   ❺
            result = repr(_result)   ❻
            print(fmt.format(**locals()))   ❼
            return _result   ❽
        return clocked   ❾
    return decorate   ❿

if __name__ == '__main__':

    @clock()   ⓫
    def snooze(seconds):
        time.sleep(seconds)

    for i in range(3):
        snooze(.123)
```

❶ `clock` is our parametrized decorator factory.

❷ `decorate` is the actual decorator.

❸ `clocked` wraps the decorated function.

❹ `_result` is the actual result of the decorated function.

❺ `_args` holds the actual arguments of `clocked`, while `args` is `str` used for display.

❻ `result` is the `str` representation of `_result`, for display.

❼ Using `**locals()` here allows any local variable of `clocked` to be referenced in the `fmt`.

❽ `clocked` will replace the decorated function, so it should return whatever that function returns.

❾ `decorate` returns `clocked`.

❿ `clock` returns `decorate`.

⓫ In this self test, `clock()` is called without arguments, so the decorator applied will use the default format `str`.

If you run Example 7-25 from the shell, this is what you get:

```
$ python3 clockdeco_param.py
[0.12412500s] snooze(0.123) -> None
[0.12411904s] snooze(0.123) -> None
[0.12410498s] snooze(0.123) -> None
```

To exercise the new functionality, here are two other modules using `clockdeco_param`, and the outputs they generate.

Example 7-26. `clockdeco_param_demo1.py`

```
import time
from clockdeco_param import clock

@clock('{name}: {elapsed}s')
def snooze(seconds):
    time.sleep(seconds)

for i in range(3):
    snooze(.123)
```

Output of Example 7-26:

```
$ python3 clockdeco_param_demo1.py
snooze: 0.12414693832397461s
snooze: 0.1241159439086914s
snooze: 0.12412118911743164s
```

Example 7-27. `clockdeco_param_demo2.py`

```
import time
from clockdeco_param import clock

@clock('{name}({args}) dt={elapsed:0.3f}s')
def snooze(seconds):
    time.sleep(seconds)
```

```
for i in range(3):
    snooze(.123)
```

Output of Example 7-27:

```
$ python3 clockdeco_param_demo2.py
snooze(0.123) dt=0.124s
snooze(0.123) dt=0.124s
snooze(0.123) dt=0.124s
```

This ends our exploration of decorators as far as time and space permits within the scope of this book. See "Further reading" on page 212, in particular Graham Dumpleton's blog and `wrapt` module for industrial-strength techniques when building decorators.

> Graham Dumpleton and Lennart Regebro — one of this book's technical reviewers — argue that decorators are best coded as classes implementing `__call__`, and not as functions like the examples in this chapter. I agree that approach is better for non-trivial decorators, but to explain the basic idea of this language feature, functions are easier to understand.

Chapter summary

We covered a lot of ground in this chapter, but I tried to make the journey as smooth as possible even if the terrain is rugged. After all, we did enter the realm of metaprogramming.

We started with a simple `@register` decorator without an inner function, and finished with a parametrized `@clock()` involving two levels of nested functions.

Registration decorators, though simple in essence, have real applications in advanced Python frameworks. We applied the registration idea to an improvement of our Strategy design pattern refactoring from Chapter 6.

Parametrized decorators almost aways involve at least two nested functions, maybe more if you want to use `@functools.wraps` to produce a decorator that provides better support for more advanced techniques. One such technique is stacked decorators, which we briefly covered.

We also visited two awesome function decorators provided in the `functools` module of standard library: `@lru_cache()` and `@singledispatch`.

Understanding how decorators actually work required covering the difference between *import time* and *run time*, then diving into variable scoping, closures, and the new `nonlocal` declaration. Mastering closures and `nonlocal` is valuable not only to build decorators, but also to code event-oriented programs for GUIs or asynchronous I/O with callbacks.

Further reading

Chapter 9 — Metaprogramming — of the Python Cookbook, 3rd. edition (O'Reilly, 2013), by David Beazley and Brian K. Jones, has several recipes from elementary decorators to very sophisticated ones, including one that can be called as regular decorator or as a decorator factory, eg. `@clock` or `@clock()`. That's recipe "9.6. Defining a Decorator That Takes an Optional Argument", page 339 in that book.

Graham Dumpleton has a series of in-depth blog posts about techniques for implementing well-behaved decorators, starting with How you implemented your Python decorator is wrong. His deep expertise in this matter is also nicely packaged in the wrapt module he wrote to simplify the implementation of decorators and dynamic function wrappers which support introspection and behave correctly when further decorated, when applied to methods and when used as descriptors[5].

Michele Simionato authored a package aiming to "simplify the usage of decorators for the average programmer, and to popularize decorators by showing various non-trivial examples", according to the docs. It's available on PyPI as the decorator package.

Created when decorators where still a new feature in Python, the Python Decorator Library wiki page has dozens of examples. Because that page started years ago, some of the techniques shown have been superseded, but the page is still an excellent source of inspiration.

PEP 443 provides the rationale and a detailed description of the single-dispatch generic functions facility. An old (March, 2005) blog post by Guido van Rossum, Five-minute Multimethods in Python walks through an implementation of generic functions (a.k.a. multimethods) using decorators. His code supports multiple-dispatch, ie. dispatch based on more than one positional argument. Guido's multimethods code is interesting, but it's a didactic example. For a modern, production-ready implementation of multiple-dispatch generic functions, check out Reg by Martijn Faassen — author of the model-driven and REST-savvy Morepath Web framework.

Closures in Python is a short blog post by Fredrik Lundh that explains the terminology of closures.

PEP 3104 — Access to Names in Outer Scopes describes the introduction of the `nonlocal` declaration to allow rebinding of names that are neither local nor global. It also includes an excellent overview of how this issue is resolved in other dynamic languages

[5] Descriptors are the subject of chapter Chapter 20

(Perl, Ruby, JavaScript etc.) and the pros and cons of the design options available to Python.

On a more theoretical level, PEP 227 — Statically Nested Scopes documents the introduction of lexical scoping in as an option in Python 2.1 and as a standard in Python 2.2, explaining the rationale and design choices for the implementation of closures in Python.

Soapbox

The designer of any language with first-class functions faces this issue: being first-class objects, functions are defined in a certain scope but may be invoked in other scopes. The question is: how to evaluate the free variables? The first and simplest answer is "dynamic scope". This means that free variables are evaluated by looking into the environment where the function is invoked.

If Python had dynamic scope and no closures, we could improvise `avg` — similar to Example 7-9 — like this:

```
>>> ### this is not a real Python console session! ###
>>> avg = make_averager()
>>> series = []  # ❶
>>> avg(10)
10.0
>>> avg(11)  # ❷
10.5
>>> avg(12)
11.0
>>> series = [1]  # ❸
>>> avg(5)
3.0
```

❶ Before using `avg` we have to define `series = []` ourselves, so we must know that `averager` (inside `make_averager`) refers to a `list` by that name.

❷ Behind the scenes, `series` is used to accumulate the values to be averaged.

❸ When `series = [1]` is executed, the previous list is lost. This could happen by accident, when handling two independent running averages at the same time.

Functions should be black boxes, with their implementation hidden from users. But with dynamic scope, if a function uses free variables, the programmer has to know its internals to set up an environment where it works correctly.

On the other hand, dynamic scope is easier to implement, which is probably why it was the path taken by John McCarthy when he created Lisp, the first language to have first-class functions. Paul Graham's article The roots of Lisp is an accessible explanation of John McCarthy's original paper about the Lisp language: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. McCarthy's paper is a masterpiece as great as Beethoven's 9th Symphony. Paul Graham translated it for the rest of us, from mathematics to English and running code.

Paul Graham's commentary also shows how tricky dynamic scoping is. Quoting from "The Roots of Lisp":

> It's an eloquent testimony to the dangers of dynamic scope that even the very first example of higher-order Lisp functions was broken because of it. It may be that McCarthy was not fully aware of the implications of dynamic scope in 1960. Dynamic scope remained in Lisp implementations for a surprisingly long time — until Sussman and Steele developed Scheme in 1975. Lexical scope does not complicate the definition of `eval` very much, but it may make compilers harder to write.

Today, lexical scope is the norm: free variables are evaluated considering the environment where the function is defined. Lexical scope complicates the implementation of languages with first-class functions, because it requires the support of closures. On the other hand, lexical scope makes source code easier to read. Most languages invented since Algol have lexical scope.

For many years Python lambdas did not provide closures, contributing to the bad name of this feature among functional-programming geeks in the blogosphere. This was fixed in Python 2.2 (December 2001), but the blogosphere has a long memory. Since then, `lambda` is embarrassing only because of its limited syntax.

Python decorators and the Decorator design pattern

Python function decorators fit the general description of Decorator given by Gamma et.al. in the Design Patterns book: "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."

At the implementation level, Python decorators do not resemble the classic Decorator design pattern, but an analogy can be made.

In the design pattern, Decorator and Component are abstract classes. An instance of a concrete decorator wraps an instance of a concrete component in order to add behaviors to it. Quoting from the Design Patterns book:

> The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities." (p. 175)

In Python, the decorator function plays the role of a concrete Decorator subclass, and the inner function it returns is a decorator instance. The returned function wraps the function to be decorated, which is analogous to the component in the design pattern. The returned function is transparent because it conforms to the interface of the component by accepting the same arguments. It forwards calls to the component and may perform additional actions either before or after it. Borrowing from the previous citation, we can adapt the last sentence to say that "Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added behaviors." That is what enable stacked decorators to work.

Note that I am not suggesting that function decorators should be used to implement the Decorator pattern in Python programs. Although this can be done in specific situations, in general the Decorator pattern is best implemented with classes to represent the Decorator and the components it will wrap.

PART IV
Object Oriented Idioms

CHAPTER 8

Object references, mutability and recycling

> ‘You are sad,’ the Knight said in an anxious tone: ‘let me sing you a song to comfort you. […] The name of the song is called “HADDOCKS’ EYES”.’
>
> ‘Oh, that’s the name of the song, is it?’ Alice said, trying to feel interested.
>
> ‘No, you don’t understand,’ the Knight said, looking a little vexed. ‘That’s what the name is CALLED. The name really IS “THE AGED AGED MAN."’ (adapted from Chapter VIII. ‘It’s my own Invention’)
>
> — Lewis Carroll
> *Through the Looking-Glass, and What Alice Found There*

Alice and the Knight set the tone of what we will see in this chapter. The theme is the distinction between objects and their names. A name is not the object; a name is a separate thing.

We start the chapter by presenting a metaphor for variables in Python: variables are labels, not boxes. If reference variables are old news you, the analogy may still be handy if you need to explain aliasing issues to others.

We then discuss the concepts of object identity, value and aliasing. A surprising trait of tuples is revealed: they are immutable but their values may change. This leads to a discussion of shallow and deep copies. References and function parameters are our next theme: the problem with mutable parameter defaults and the safe handling of mutable arguments passed by clients of our functions.

The last sections of the chapter cover garbage collection, the `del` command, and how to use weak references to “remember” objects without keeping them alive.

This is a rather dry chapter, but its topics lie at the heart of many subtle bugs in real Python programs.

Let’s start by unlearning that a variable is like a box where you store data.

Variables are not boxes

In 1997 I took a summer course on Java at MIT. The professor, Lynn Andrea Stein[1] — an award-winning computer science educator — made the point that the usual "variables as boxes" metaphor actually hinders the understanding of reference variables in OO languages. Python variables are like reference variables in Java, so it's better to think of them as labels attached to objects.

Example 8-1 is a simple interaction that the "variables as boxes" idea cannot explain.

Example 8-1. Variables `a` and `b` hold references to the same list, not copies of the list.

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> b
[1, 2, 3, 4]
```

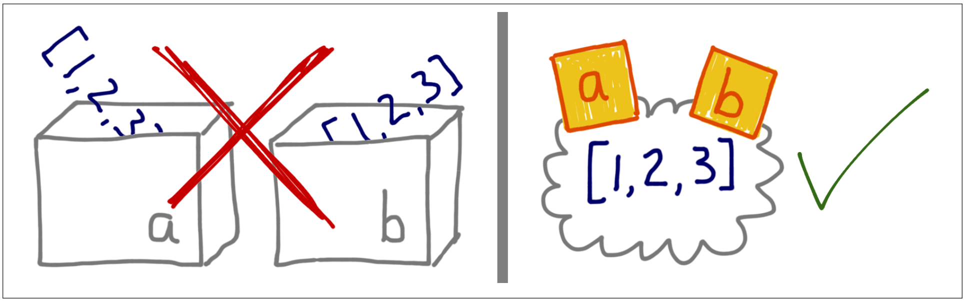


Figure 8-1. If you imagine variables are like boxes, you can't make sense of assignment in Python. Think of variables as Post-it® notes. Then Example 8-1 becomes easy to explain.

Prof. Stein also spoke about assignment in a very deliberate way. For example, when talking about a seesaw object in a simulation, she would say: "Variable *s* is assigned to the seesaw", but never "The seesaw is assigned to variable *s* ". With reference variables it makes much more sense to say that the variable is assigned to an object, and not the other way around. After all, the object is created before the assignment. Example 8-2 proves that the right-hand side of an assignment happens first:

Example 8-2. Variables are assigned to objects only after the objects are created.

```
>>> class Gizmo:
...     def __init__(self):
...         print('Gizmo id: %d' % id(self))
```

[1] Prof. Stein is now at Olin College of Engineering

```
...
>>> x = Gizmo()
Gizmo id: 4301489152  ❶
>>> y = Gizmo() * 10  ❷
Gizmo id: 4301489432  ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'Gizmo' and 'int'
>>>
>>> dir()  ❹
['Gizmo', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'x']
```

❶ The output `Gizmo id: ...` is a side effect of creating a `Gizmo` instance.

❷ Multiplying a `Gizmo` instance will raise an exception.

❸ Here is proof that a second `Gizmo` was actually instantiated before the multiplication was attempted.

❹ But variable `y` was never created, because the exception happened while the right-hand side of the assignment was being evaluated.

To understand an assignment in Python, always read the right-hand side first: that's where the object is created or retrieved. After that, the variable on the left is bound to the object, like a label stuck to it. Just forget about the boxes.

Since variables are mere labels, nothing prevents an object from having several labels assigned to it. When that happens, you have *aliasing*, our next topic.

Identity, equality and aliases

Lewis Carroll is the pen name of Prof. Charles Lutwidge Dodgson. Mr. Carroll is not only equal to Prof. Dodgson: they are one and the same. Example 8-3 expresses this idea in Python:

Example 8-3. `charles` and `lewis` refer to the same object.

```
>>> charles = {'name': 'Charles L. Dodgson', 'born': 1832}
>>> lewis = charles  ❶
>>> lewis is charles
True
>>> id(charles), id(lewis)  ❷
(4300473992, 4300473992)
>>> lewis['balance'] = 950  ❸
>>> charles
{'name': 'Charles L. Dodgson', 'balance': 950, 'born': 1832}
```

❶ `lewis` is an alias for `charles`.

❷ The `is` operator and the `id` function confirm it.

❸ Adding an item to `lewis` is the same as adding an item to `charles`.

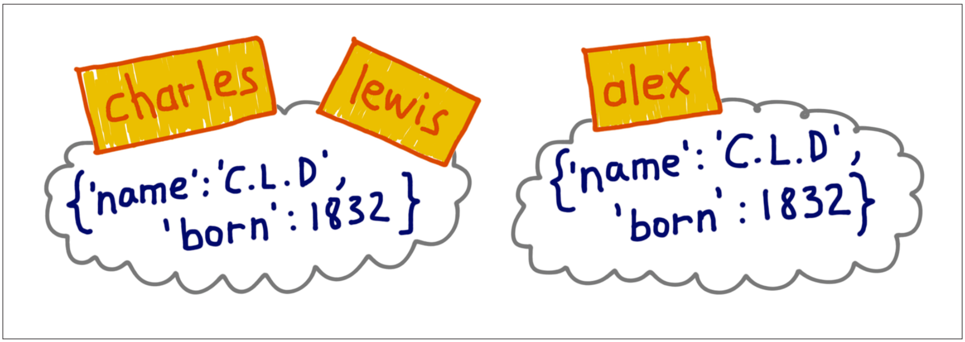


Figure 8-2. `charles` and `lewis` are bound to the same object; `alex` is bound to a separate object of equal contents.

However, suppose an impostor — let's call him Dr. Alexander Pedachenko — claims he is Charles L. Dodgson, born in 1832. His credentials may be the same, but Dr. Pedachenko is not Prof. Dodgson. Figure 8-2 illustrates this scenario.

Example 8-4. `alex` and `charles` compare equal, but `alex` is not `charles`.

```
>>> alex = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950}  ❶
>>> alex == charles  ❷
True
>>> alex is not charles  ❸
True
```

❶ `alex` refers to an object that is a replica of the object assigned to `charles`.

❷ The objects compare equal, because of the `__eq__` implementation in the `dict` class.

❸ But they are distinct objects. This is the Pythonic way of writing the negative identity comparison: `a is not b`.

Example 8-3 is an example of *aliasing*. In that code, `lewis` and `charles` are aliases: two variables bound to the same object. On the other hand, `alex` is not an alias for `charles`: these variables are bound to distinct objects. The objects bound to `alex` and `charles` have the same *value* — that's what `==` compares — but they have different identities.

In the Python Language Reference — Data model — section 3.1. Objects, values and types states:

> Every object has an identity, a type and a value. An object's identity never changes once it has been created; you may think of it as the object's address in memory. The `is` operator compares the identity of two objects; the `id()` function returns an integer representing its identity.

The real meaning of an object's id is implementation-dependent. In CPython, `id()` returns the memory address of the object, but it may be something else in another Python interpreter. The key point is that the id is guaranteed to be a unique numeric label, and it will never change during the life of the object.

In practice, we rarely use the `id()` function while programming. Identity checks are most often done with the `is` operator, and not by comparing ids. We'll talk about `is` versus == next.

Choosing between == and `is`

The == operator compares the values of objects (the data they hold), while `is` compares their identities.

We often care about values and not identities, so == appears more frequently than `is` in Python code.

However, if you are comparing a variable to a singleton, then it makes sense to use `is`. By far, the most common case is checking whether a variable is bound to `None`. This is the recommended way to do it:

```
x is None
```

And the proper way to write its negation is:

```
x is not None
```

The `is` operator is faster than ==, because it cannot be overloaded, so Python does not have to find and invoke special methods to evaluate it, and computing is as simple as comparing two integer ids. In contrast, `a == b` is syntactic sugar for `a.__eq__(b)`. The `__eq__` method inherited from `object` compares object ids, so it produces the same result as `is`. But most built-in types override `__eq__` with more meaningful implementations that actually take into account the values of the object attributes. Equality may involve a lot of processing — for example, when comparing large collections or deeply nested structures.

To wrap up this discussion of identity versus equality, we'll see that the famously immutable `tuple` is not as rigid as you may expect.

The relative immutability of tuples

Tuples, like most Python collections — lists, dicts, sets etc. — hold references to objects[2]. If the referenced items are mutable, they may change even if the tuple itself does not. In other words, the immutability of tuples really refers to the physical contents of the `tuple` data structure (ie. the references it holds), and does not extend to the referenced objects.

Example 8-5 illustrates the situation in which the value of a tuple changes as result of changes to a mutable object referenced in it. What can never change in a tuple is the identity of the items it contains.

Example 8-5. `t1` and `t2` initially compare equal, but changing a mutable item inside tuple `t1` makes it different.

```
>>> t1 = (1, 2, [30, 40])  ❶
>>> t2 = (1, 2, [30, 40])  ❷
>>> t1 == t2  ❸
True
>>> id(t1[-1])  ❹
4302515784
>>> t1[-1].append(99)  ❺
>>> t1
(1, 2, [30, 40, 99])
>>> id(t1[-1])  ❻
4302515784
>>> t1 == t2  ❼
False
```

❶ `t1` is immutable, but `t1[-1]` is mutable.

❷ Build a tuple `t2` whose items are equal to those of `t1`.

❸ Although distinct objects, `t1` and `t2` compare equal, as expected.

❹ Inspect the identity of the list at `t1[-1]`.

❺ Modify the `t1[-1]` list in place.

❻ The identity of `t1[-1]` has not changed, only its value.

❼ `t1` and `t2` are now different.

This relative immutability of tuples is behind the riddle "A += assignment puzzler" on page 40. It's also the reason why some tuples are unhashable, as we've seen in "What is hashable?" on page 64.

[2] On the other hand, single-type sequences like `str`, `bytes` and `array.array` are flat: they don't contain references but physically hold their data — characters, bytes and numbers — in contiguous memory.

The distinction between equality and identity has further implications when you need to copy an object. A copy is an equal object with a different id. But if an object contains other objects, should the copy also duplicate the inner objects, or is it ok to share them? There's no single answer. Read on for a discussion.

Copies are shallow by default

The easiest way to copy a list (or most built-in mutable collections) is to use the built-in constructor for the type itself, for example:

```
>>> l1 = [3, [55, 44], (7, 8, 9)]
>>> l2 = list(l1)  ❶
>>> l2
[3, [55, 44], (7, 8, 9)]
>>> l2 == l1  ❷
True
>>> l2 is l1  ❸
False
```

❶ `list(l1)` creates a copy of `l1`;

❷ the copies are equal;

❸ but refer to two different objects.

For lists and other mutable sequences, the shortcut `l2 = l1[:]` also makes a copy.

However, using the constructor or `[:]` produces a *shallow copy*, i.e. the outermost container is duplicated, but the copy is filled with references to the same items held by the original container. This saves memory and causes no problems if all the items are immutable. But if there are mutable items, this may lead to unpleasant surprises.

In Example 8-6, we create a shallow copy of a list containing another list and a tuple, and then make changes to see how they affect the referenced objects.

> If you have a connected computer on hand, I highly recommend watching the interactive animation for Example 8-6 at the Online Python Tutor. As I write this, direct linking to a prepared example at pythontutor.com is not working reliably, but the tool is awesome, so taking the time to copy and paste the code is worthwhile.

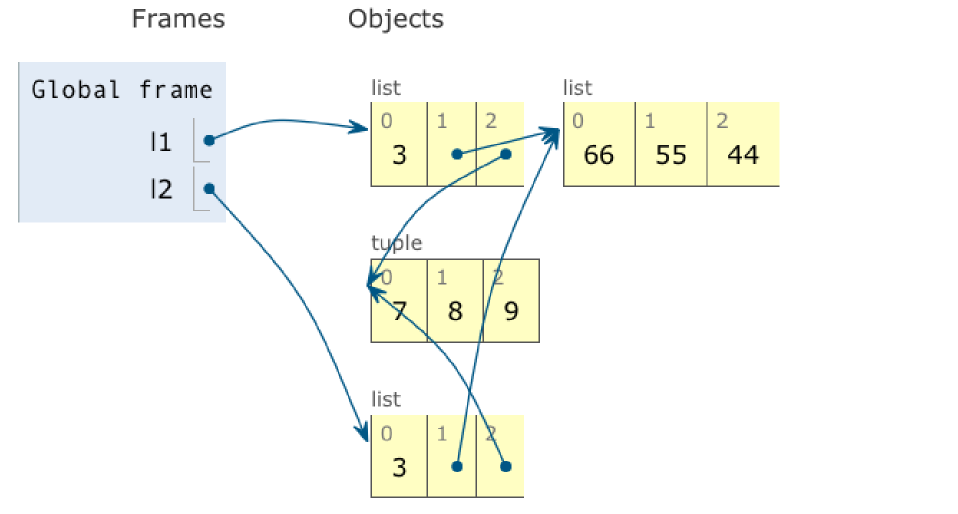


Figure 8-3. Program state immediately after the assignment `l2 = list(l1)` in Example 8-6. `l1` and `l2` refer to distinct lists, but the lists share references to the same inner list object `[66, 55, 44]` and tuple `(7, 8, 9)`. (Diagram generated by Online Python Tutor)

Example 8-6. Making a shallow copy of a list containing another list. Copy and paste this code to see it animated at Online Python Tutor.

```
l1 = [3, [66, 55, 44], (7, 8, 9)]
l2 = list(l1)      # ❶
l1.append(100)     # ❷
l1[1].remove(55)   # ❸
print('l1:', l1)
print('l2:', l2)
l2[1] += [33, 22]  # ❹
l2[2] += (10, 11)  # ❺
print('l1:', l1)
print('l2:', l2)
```

❶ `l2` is a shallow copy of `l1`. This state is depicted in Figure 8-3.

❷ Appending `100` to `l1` has no effect on `l2`.

❸ Here we remove `55` from the inner list `l1[1]`. This affects `l2` because `l2[1]` is bound to the same list as `l1[1]`.

❹ For a mutable object like the list referred by `l2[1]`, the operator `+=` changes the list in-place. This change is visible at `l1[1]`, which is an alias for `l2[1]`.

❺ += on a tuple creates a new tuple and rebinds the variable l2[2] here. This is the same as doing l2[2] = l2[2] + (10, 11). Now the tuples in the last position of l1 and l2 are no longer the same object. See Figure 8-4.

The output of Example 8-6 is Example 8-7, and the final state of the objects is depicted in Figure 8-4.

Example 8-7. Output of Example 8-6.

```
l1: [3, [66, 44], (7, 8, 9), 100]
l2: [3, [66, 44], (7, 8, 9)]
l1: [3, [66, 44, 33, 22], (7, 8, 9), 100]
l2: [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]
```

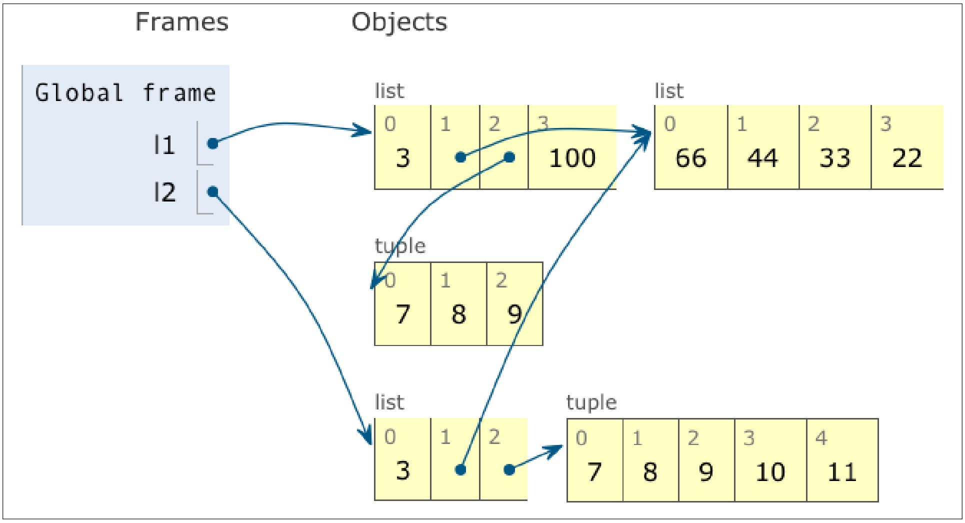


Figure 8-4. Final state of l1 and l2: they still share references to the same list object, now containing [66, 44, 33, 22], but the operation l2[2] += (10, 11) created a new tuple with content (7, 8, 9, 10, 11), unrelated to the tuple (7, 8, 9) referenced by l1[2]. (Diagram generated by Online Python Tutor)

It should be clear now that shallow copies are easy to make, but they may or may not be what you want. How to make deep copies is our next topic.

Deep and shallow copies of arbitrary objects

Working with shallow copies is not always a problem, but sometimes you need to make deep copies, i.e. duplicates that do not share references of embedded objects. The copy module provides the deepcopy and copy functions that return deep and shallow copies of arbitrary objects.

To illustrate the use of copy() and deepcopy(), Example 8-8 defines a simple class Bus, representing a school bus that is loaded with passengers and then picks or drops passengers on its route.

Example 8-8. Bus picks and drops passengers.

```
class Bus:

    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []
        else:
            self.passengers = list(passengers)

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name)
```

Now in the interactive Example 8-9 we will create a Bus instance, bus1 and two clones: a shallow copy (bus2) and a deep copy (bus3), to observe what happens as bus1 drops a student.

Example 8-9. Effects of using copy versus deepcopy

```
>>> import copy
>>> bus1 = Bus(['Alice', 'Bill', 'Claire', 'David'])
>>> bus2 = copy.copy(bus1)
>>> bus3 = copy.deepcopy(bus1)
>>> id(bus1), id(bus2), id(bus3)
(4301498296, 4301499416, 4301499752)  ❶
>>> bus1.drop('Bill')
>>> bus2.passengers
['Alice', 'Claire', 'David']          ❷
>>> id(bus1.passengers), id(bus2.passengers), id(bus3.passengers)
(4302658568, 4302658568, 4302657800)  ❸
>>> bus3.passengers
['Alice', 'Bill', 'Claire', 'David']  ❹
```

❶ Using copy and deepcopy we create three distinct Bus instances.

❷ After bus1 drops 'Bill', he is also missing from bus2.

❸ Inspection of the passengers atributes shows that bus1 and bus2 share the same list object, because bus2 is a shallow copy of bus1.

❹ bus3 is a deep copy of bus1, so its passengers attribute refers to another list.

Note that making deep copies is not a simple matter in the general case. Objects may have cyclic references which would cause a naïve algorithm to enter an infinite loop. The deepcopy function remembers the objects already copied to handle cyclic references gracefully. This is demonstrated in Example 8-10.

Example 8-10. Cyclic references: b refers to a, and then is appended to a; deepcopy still manages to copy a.

```
>>> a = [10, 20]
>>> b = [a, 30]
>>> a.append(b)
>>> a
[10, 20, [[...], 30]]
>>> from copy import deepcopy
>>> c = deepcopy(a)
>>> c
[10, 20, [[...], 30]]
```

Also, a deep copy may be too deep in some cases. For example, objects may refer to external resources or singletons that should not be copied. You may control the behavior of both copy and deepcopy by implementing the __copy__() and __deepcopy__() special methods as described in the copy module documentation.

The sharing of objects through aliases also explains how parameter passing works in Python, and the problem of using mutable types as parameter defaults. These issues will be covered next.

Function parameters as references

The only mode of parameter passing in Python is *call by sharing*. That is the same mode used in most OO languages, including Ruby, SmallTalk and Java[3]. Call by sharing means that each formal parameter of the function gets a copy of each reference in the arguments. In other words, the parameters inside the function become aliases of the actual arguments.

The result of this scheme is that a function may change any mutable object passed as a parameter, but it cannot change the identity of those objects, i.e. it cannot replace altogether an object with another. Example 8-11 shows a simple function using += on one of its parameters. As we pass numbers, lists and tuples to the function, the actual arguments passed are affected in different ways.

Example 8-11. A function may change any mutable object it receives.

```
>>> def f(a, b):
...     a += b
```

[3] This applies to Java reference types; primitive types use call by value.

```
...     return a
...
>>> x = 1
>>> y = 2
>>> f(x, y)
3
>>> x, y  ❶
(1, 2)
>>> a = [1, 2]
>>> b = [3, 4]
>>> f(a, b)
[1, 2, 3, 4]
>>> a, b  ❷
([1, 2, 3, 4], [3, 4])
>>> t = (10, 20)
>>> u = (30, 40)
>>> f(t, u)  ❸
(10, 20, 30, 40)
>>> t, u
((10, 20), (30, 40))
```

❶ The number `x` is unchanged.

❷ The list `a` is changed.

❸ The tuple `t` is unchanged.

Another issue related to function parameters is the use of mutable values for defaults. See next.

Mutable types as parameter defaults: bad idea

Optional parameters with default values are a great feature of Python function definitions, allowing our APIs to evolve while remaining backward-compatible. However, you should avoid mutable objects as default values for parameters.

To illustrate this point, we take the `Bus` class from Example 8-8 and change its `__init__` method to create `HauntedBus`. Here we tried to be clever and instead of having a default value of `passengers=None` we have `passengers=[]`, thus avoiding the `if` in the previous `__init__`. This "cleverness" gets us into trouble.

Example 8-12. A simple class to illustrate the danger of a mutable default.

```
class HauntedBus:
    """A bus model haunted by ghost passengers"""

    def __init__(self, passengers=[]):  ❶
        self.passengers = passengers  ❷

    def pick(self, name):
        self.passengers.append(name)  ❸
    def drop(self, name):
        self.passengers.remove(name)
```

❶ When the `passengers` argument is not passed, this parameter is bound to the default list object, which is initially empty.

❷ This assignment makes `self.passengers` an alias for `passengers` which is itself an alias for the default list, when no `passengers` argument is given.

❸ When the methods `.remove()` and `.append()` are used with `self.passengers` we are actually mutating the default list, which is an attribute of the function object.

Example 8-13 shows the eerie behavior of the `HauntedBus`.

Example 8-13. Buses haunted by ghost passengers.

```
>>> bus1 = HauntedBus(['Alice', 'Bill'])
>>> bus1.passengers
['Alice', 'Bill']
>>> bus1.pick('Charlie')
>>> bus1.drop('Alice')
>>> bus1.passengers  ❶
['Bill', 'Charlie']
>>> bus2 = HauntedBus()  ❷
>>> bus2.pick('Carrie')
>>> bus2.passengers
['Carrie']
>>> bus3 = HauntedBus()  ❸
>>> bus3.passengers  ❹
['Carrie']
>>> bus3.pick('Dave')
>>> bus2.passengers  ❺
['Carrie', 'Dave']
>>> bus2.passengers is bus3.passengers  ❻
True
>>> bus1.passengers  ❼
['Bill', 'Charlie']
```

❶ So far, so good: no surprises with `bus1`.

❷ `bus2` starts empty, so the default empty list is assigned to `self.passengers`.

❸ `bus3` also starts empty, again the default list is assigned.

❹ The default is no longer empty!

❺ Now `Dave`, picked by `bus3`, appears in `bus2`.

❻ The problem: `bus2.passengers` and `bus3.passengers` refer to the same list.

❼ But `bus1.passengers` is a distinct list.

The problem is that Bus instances that don't get an initial passenger list end up sharing the same passenger list among themselves.

Such bugs may be subtle. As Example 8-13 demonstrates, when a HauntedBus is instantiated with passengers, it works as expected. Strange things happen only when a HauntedBus starts empty, because then self.passengers becomes an alias for the default value of the passengers parameter. The problem is that each default value is evaluated when the function is defined — i.e. usually when the module is loaded — and the default values become attributes of the function object. So if a default value is a mutable object, and you change it, the change will affect every future call of the function.

After running the lines in Example 8-13 you can inspect the HauntedBus.__init__ object and see the ghost students haunting its __defaults__ attribute.

```
>>> dir(HauntedBus.__init__)  # doctest: +ELLIPSIS
['__annotations__', '__call__', ..., '__defaults__', ...]
>>> HauntedBus.__init__.__defaults__
(['Carrie', 'Dave'],)
```

Finally, we can verify that bus2.passengers is an alias bound to the first element of the HauntedBus.__init__.__defaults__ attribute:

```
>>> HauntedBus.__init__.__defaults__[0] is bus2.passengers
True
```

The issue with mutable defaults explains why None is often used as the default value for parameters that may receive mutable values. In Example 8-8, __init__ checks whether the passengers argument is None, and assigns a new empty list to self.passengers. If passengers is not None, the correct implementation assigns a copy of it to self.passengers. The following section explains why.

Defensive programming with mutable parameters

When you are coding a function that receives a mutable parameter you should carefully consider whether the caller expects the argument passed to be changed.

For example, if your function receives a dict and needs to modify it while processing it, should this side effect be visible outside of the function or not? Actually it depends on the context. It's really a matter of aligning the expectation of the coder of the function and that of the caller.

The last bus example in this chapter shows how a TwilightBus breaks expectations by sharing its passenger list with its clients. Before studying the implementation, see in Example 8-14 how the TwilightBus class works from the perspective of a client of the class.

Example 8-14. Passengers disappear when dropped by a `TwilightBus`.

```
>>> basketball_team = ['Sue', 'Tina', 'Maya', 'Diana', 'Pat']  ❶
>>> bus = TwilightBus(basketball_team)  ❷
>>> bus.drop('Tina')  ❸
>>> bus.drop('Pat')
>>> basketball_team  ❹
['Sue', 'Maya', 'Diana']
```

❶ `basketball_team` holds five student names.

❷ A `TwilightBus` is loaded with the team.

❸ The `bus` drops one student, then another.

❹ The dropped passengers vanished from the basketball team!

`TwilightBus` violates the "Principle of least astonishment", a best practice of interface design. It surely is astonishing that when the bus drops a student, her name is removed from the basketball team roster.

Example 8-15 is the implementation `TwilightBus` and an explanation of the problem.

Example 8-15. A simple class to show the perils of mutating received arguments.

```
class TwilightBus:
    """A bus model that makes passengers vanish"""

    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []   ❶
        else:
            self.passengers = passengers  ❷

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name)   ❸
```

❶ Here we are careful to create a new empty list when `passengers` is `None`.

❷ However, this assignment makes `self.passengers` an alias for `passengers` which is itself an alias for the actual argument passed to `__init__` — `basketball_team` in Example 8-14.

❸ When the methods `.remove()` and `.append()` are used with `self.passengers` we are actually mutating the original list received as argument to the constructor.

The problem here is that the bus is aliasing the list that is passed to the constructor. Instead, it should keep its own passenger list. The fix is simple: in `__init__`, when the `passengers` parameter is provided, `self.passengers` should be initialized with a copy of it, as we did correctly in Example 8-8 ("Deep and shallow copies of arbitrary objects" on page 227):

```
    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []
        else:
            self.passengers = list(passengers) ❶
```

❶ Make a copy of the passengers list, or convert it to a list if it's not one.

Now our internal handling of the passenger list will not affect the argument used to initialize the bus. As a bonus, this solution is more flexible: now the argument passed to the passengers parameter may be a tuple or any other iterable, like a set or even database results, because the list constructor accepts any iterable. As we create our own list to manage, we make sure that it supports the necessary .remove() and .append() operations we use in the .pick() and .drop() methods.

Unless a method is explicitly intended to mutate an object received as argument, you should think twice before aliasing the argument object by simply assigning it to an instance variable in your class. If in doubt, make a copy. Your clients will often be happier.

del and garbage collection

> Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected.
>
> — Python Language Reference: Data model

The del statement deletes names, not objects. An object may be garbage collected as result of a del command, but only if the variable deleted holds the last reference to the object, or if the object becomes unreachable[4]. Rebinding a variable may also cause the number of references to an object reach zero, causing its destruction.

[4] If two objects refer to each other, as in Example 8-10, they may be destroyed if the garbage collector determines that they are otherwise unreachable because their only references are their mutual references.

There is a `__del__` special method, but it does not cause the disposal of the instance, and should not be called by your code. `__del__` is invoked by the Python interpreter when the instance is about to be destroyed to give it a chance to release external resources. You will seldom need to implement `__del__` in your own code, yet some Python beginners spend time coding it for no good reason. The proper use of `__del__` is rather tricky. See the `__del__` special method documentation in the Python data model chapter of the Language Reference.

In CPython the primary algorithm for garbage collection is reference counting. Essentially, each object keeps count of how many references point to it. As soon as that *refcount* reaches zero, the object is immediately destroyed: CPython calls the `__del__` method on the object (if defined) and then frees the memory allocated to the object. In CPython 2.0 a generational garbage collection algorithm was added to detect groups of objects involved in reference cycles — which may be unreachable even with outstanding references to them, when all the mutual references are contained within the group. Other implementations of Python have more sophisticated garbage collector that do not rely of reference counting, which means the `__del__` method may not be called immediately when there are no more references to the object. See the PyPy, Garbage Collection, And A Deadlock by A. Jesse Jiryu Davis for discussion of improper and proper use of `__del__`.

In order to demonstrate the end of an object's life, Example 8-16 uses `weakref.finalize` to register a callback function to be called when an object is destroyed.

Example 8-16. Watching the end of an object when no more references point to it.

```
>>> import weakref
>>> s1 = {1, 2, 3}
>>> s2 = s1         ❶
>>> def bye():      ❷
...     print('Gone with the wind...')
...
>>> ender = weakref.finalize(s1, bye)  ❸
>>> ender.alive  ❹
True
>>> del s1
>>> ender.alive  ❺
True
>>> s2 = 'spam'  ❻
Gone with the wind...
>>> ender.alive
False
```

❶ `s1` and `s2` are aliases referring to the same set, `{1, 2, 3}`.

❷ This function must not be a bound method the object about to be destroyed or otherwise hold a reference to it.

❸ Register the `bye` callback on the object referred by `s1`.

❹ The `.alive` attribute is `True` before the `finalize` object is called.

❺ As discussed, `del` does not delete an object, just a reference to it.

❻ Rebinding the last reference, `s2`, makes `{1, 2, 3}` unreachable. It is destroyed, the `bye` callback is invoked and `ender.alive` becomes `False`.

The point of Example 8-16 is to make explicit that `del` does not delete objects, but objects may be deleted as a consequence of being unreachable after `del` is used.

You may be wondering why the `{1, 2, 3}` object was destroyed in Example 8-16. After all, the `s1` reference was passed to the `finalize` function, which must have held on to it in order to monitor the object and invoke the callback. This works because `finalize` holds a *weak reference* to `{1, 2, 3}`, as explained in the next section.

Weak references

The presence of references is what keeps an object alive in memory. When the reference count of an object reaches zero, the garbage collector disposes of it. But sometimes it is useful to have a reference to an object that does not keep it around longer than necessary. A common use case is a cache.

Weak references to an object do not increase its reference count. The object that is the target of a reference is called the *referent*. Therefore, we say that a weak reference does not prevent the referent from being garbage collected.

Weak references are useful in caching applications because you don't want the cached objects to be kept alive just because they are referenced by the cache.

Example 8-17 shows how a `weakref.ref` instance can be called to reach its referent. If the object is alive, calling the weak reference returns it, otherwise `None` is returned.

> Example 8-17 is a console session, and the Python console automatically binds the _ variable to the result of expressions that are not `None`. This interfered with my intended demonstration but also highlights a practical matter: when trying to micro-manage memory we are often surprised by hidden, implicit assignments that create new references to our objects. The _ console variable is one example. Traceback objects are another common source of unexpected references.

Example 8-17. A weak reference is a callable that returns the referenced object or None *if the referent is no more.*

```
>>> import weakref
>>> a_set = {0, 1}
>>> wref = weakref.ref(a_set)  ❶
>>> wref
<weakref at 0x100637598; to 'set' at 0x100636748>
>>> wref()  ❷
{0, 1}
>>> a_set = {2, 3, 4}  ❸
>>> wref()  ❹
{0, 1}
>>> wref() is None  ❺
False
>>> wref() is None  ❻
True
```

❶ The `wref` weak reference object is created and inspected in the next line.

❷ Invoking `wref()` returns the referenced object, `{0, 1}`. Because this is a console session, the result `{0, 1}` is bound to the _ variable.

❸ `a_set` no longer refers to the `{0, 1}` set, so its reference count is decreased. But the _ variable still refers to it.

❹ Calling `wref()` still returns `{0, 1}`.

❺ When this expression is evaluated, `{0, 1}` lives, therefore `wref()` is not `None`. But _ is then bound to the resulting value, `False`. Now there are no more strong references to `{0, 1}`.

❻ Because the `{0, 1}` object is now gone, this last call to `wref()` returns `None`.

The `weakref` module documentation makes the point that the `weakref.ref` class is actually a low-level interface intended for advanced uses, and that most programs are better served by the use of the `weakref` collections and `finalize`. In other words, consider using `WeakKeyDictionary`, `WeakValueDictionary`, `WeakSet` and `finalize` — which use weak references internally — instead of creating and handling your own `weakref.ref` instances by hand. We just did that in Example 8-17 in the hope that showing a single `weakref.ref` in action could take away some of the mistery around them. But in practice, most of the time Python programs use the `weakref` collections.

The next subsection briefly discusses the `weakref` collections.

The `WeakValueDictionary` skit

The class `WeakValueDictionary` implements a mutable mapping where the values are weak references to objects. When a referred object is garbage collected elsewhere in the program, the corresponding key is automatically removed from the `WeakValueDictionary`. This is commonly used for caching.

Our demonstration of a `WeakValueDictionary` is inspired by the classic *Cheese Shop* skit by Monty Python, in which a customer asks for more than 40 kinds of cheese, including cheddar and mozzarella, but none are in stock[5].

Example 8-18 implements a trivial class to represent each kind of cheese.

Example 8-18. `Cheese` has a `kind` attribute and a standard representation.

```
class Cheese:

    def __init__(self, kind):
        self.kind = kind

    def __repr__(self):
        return 'Cheese(%r)' % self.kind
```

In Example 8-19 each cheese is loaded from a `catalog` to a `stock` implemented as a `WeakValueDictionary`. However, all but one disappear from the `stock` as soon as the `catalog` is deleted. Can you explain why the Parmesan cheese lasts longer than the others[6]? The tip after the code has the answer.

Example 8-19. Customer: “Have you in fact got any cheese here at all?"”

```
>>> import weakref
>>> stock = weakref.WeakValueDictionary()  ❶
>>> catalog = [Cheese('Red Leicester'), Cheese('Tilsit'),
...                 Cheese('Brie'), Cheese('Parmesan')]
...
>>> for cheese in catalog:
...     stock[cheese.kind] = cheese  ❷
...
>>> sorted(stock.keys())
['Brie', 'Parmesan', 'Red Leicester', 'Tilsit']  ❸
>>> del catalog
>>> sorted(stock.keys())
['Parmesan']  ❹
>>> del cheese
>>> sorted(stock.keys())
[]
```

[5] `cheeseshop.python.org` is also an alias for PyPI — the Python Package Index software repository — which started its life quite empty. At this writing the Python Cheese Shop has 41,426 packages. Not bad, but still far from the more than 131,000 modules available in CPAN — the Comprehensive Perl Archive Network — the envy of all dynamic language communities.

[6] Parmesan cheese is aged at least a year at the factory, so it is more durable than fresh cheese, but this is not the answer we are looking for.

❶ stock is a WeakValueDictionary

❷ The stock maps the name of the cheese to a weak reference to the cheese instance in the catalog.

❸ The stock is complete.

❹ After the catalog is deleted, most cheeses are gone from the stock, as expected in WeakValueDictionary. Why not all, in this case?

> A temporary variable may cause an object to last longer than expected by holding a reference to it. This is usually not a problem with local variables: they are destroyed when the function returns. But in Example 8-19, the for loop variable cheese is a global variable and will never go away unless explicitly deleted.

A counterpart to the WeakValueDictionary is the WeakKeyDictionary in which the keys are weak references. The weakref.WeakKeyDictionary documentation hints on possible uses:

> [A WeakKeyDictionary] can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

The weakref module also provides a WeakSet, simply described in the docs as "Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more." If you need to build a class that is aware of every one of its instances, a good solution is to create a class attribute with a WeakSet to hold the references to the instances. Otherwise, if a regular set was used, the instances would never be garbage collected, because the class itself would have strong references to them, and classes live as long as the Python process unless you deliberately delete them.

These collections, and weak references in general are limited in the kinds of objects they can handle. The next section explains.

Limitations of weak references

Not every Python object may be the target, or referent, of a weak reference. Basic list and dict instances may not be referents, but a plain subclass of either can solve this problem easily:

```python
class MyList(list):
    """list subclass whose instances may be weakly referenced"""

a_list = MyList(range(10))
```

```
    # a_list can be the target of a weak reference
    wref_to_a_list = weakref.ref(a_list)
```

A `set` instance can be a referent, that's why a `set` was used in Example 8-17. User-defined types also pose no problem, which explains why the silly `Cheese` class was needed in Example 8-19. But `int` and `tuple` instances cannot be targets of weak references, even if subclasses of those types are created.

Most of these limitations are implementation details of CPython that may not apply to other Python iterpreters. They are the result of internal optimizations, some of which are discussed in the following (highly optional) section.

Tricks Python plays with immutables

> You may safely skip this section. It discusses some Python implementation details that are not really important for *users* of Python. They are shortcuts and optimizations done by the CPython core developers which should not bother you when using the language, and that may not apply to other Python implementations or even future versions of CPython. Nevertheless, while experimenting with aliases and copies you may stumble upon these tricks, so I felt they were worth mentioning.

I was surprised to learn that, for a tuple `t`, `t[:]` does not make a copy, but returns a reference to the same object. You also get a reference to the same tuple if you write `tuple(t)`.[7]

Example 8-20. A tuple built from another is actually the same exact tuple.

```
>>> t1 = (1, 2, 3)
>>> t2 = tuple(t1)
>>> t2 is t1  ❶
True
>>> t3 = t1[:]
>>> t3 is t1  ❷
True
```

❶ `t1` and `t2` are bound to the same object.

❷ And so is `t3`.

The same behavior can be observed with instances of `str`, `bytes` and `frozenset`. Note that a `frozenset` is not a sequence, so `fs[:]` does not work if `fs` is a `frozenset`. But

[7] This is clearly documented, type `help(tuple)` in the Python console to read: "If the argument is a tuple, the return value is the same object." I thought I knew everything about tuples before writing this book.

`fs.copy()` has the same effect: it cheats and returns a reference to the same object, and not a copy at all[8].

Example 8-21. String literals may create shared objects.

```
>>> t1 = (1, 2, 3)
>>> t3 = (1, 2, 3)  # ❶
>>> t3 is t1  # ❷
False
>>> s1 = 'ABC'
>>> s2 = 'ABC'  # ❸
>>> s2 is s1 # ❹
True
```

❶ Creating a new tuple from scratch.

❷ `t1` and `t3` are equal, but not the same object.

❸ Creating a second `str` from scratch.

❹ Surprise: `a` and `b` refer to the same `str`!

The sharing of string literals is an optimization technique called *interning*. CPython uses the same technique with small integers to avoid unnecessary duplication of "popular" numbers like 0, -1 and 42. Please note that CPython does not intern all strings or integers, and the criteria it uses to do so is an undocumented implementation detail.

Never depend on `str` or `int` interning! Always use == and not `is` to compare them for equality. Interning is a feature for internal use of the Python interpreter.

The tricks discussed in this section, including the behavior of `frozenset.copy()` are "white lies": they save memory and make the interpreter faster. Do not worry about them, they should not give you any trouble because they only apply to immutable types. Probably the best use of these bits of trivia is to win bets with fellow Pythonistas.

[8] The white lie of having the `copy` method not copying anything can be explained by interface compatibility: it makes `frozenset` more compatible with `set`. Anyway, it makes no difference to the end user whether two identical immutable objects are the same or are copies.

Chapter summary

Every Python object has an identity, a type and a value. Only the value of an object changes over time[9].

If two variables refer to immutable objects that have equal values (`a == b` is `True`), in practice it rarely matters if they refer to copies or are aliases referring to the same object because the value of an immutable object does not change, with one exception. The exception is immutable collections such as tuples and frozensets: if an immutable collection holds references to mutable items, then its value may actually change when the value of a mutable item changes. In practice, this scenario is not so common. What never changes in an immutable collection are the identities of the objects within.

The fact that variables hold references has many practical consequences in Python programming.

1. Simple assignment does not create copies.
2. Augmented assignment with `+=`, `*=` creates new objects if the left-hand variable is bound to an immutable object, but may modify a mutable object in-place.
3. Assigning a new value to an existing variable does not change the object previously bound to it. This is called a rebinding: the variable is now bound to a different object. If that variable was the last reference to the previous object, that object will be garbage collected.
4. Function parameters are passed as aliases, which means the function may change any mutable object received as an argument. There is no way to prevent this, except making local copies or using immutable objects (eg. passing a tuple instead of a list).
5. Using mutable objects as default values for function parameters is dangerous because if the parameters are changed in-place then the default is changed, affecting every future call that relies on the default.

In CPython, objects are discarded as soon as the number of references to them reaches zero. They may also be discarded if they form groups with cyclic references but no outside references. In some situations it may be useful to hold a reference to an object that will not — by itself — keep an object alive. One example is a class that wants to keep track of all its current instances. This can be done with weak references, a low level mechanism underlying the more useful collections `WeakValueDictionary`, `WeakKeyDictionary`, `WeakSet`, and the `finalize` function from the `weakref` module.

[9] Actually the type of an object may be changed by merely assigning a different class to its `__class__` attribute, but that is pure evil and I regret writing this footnote.

Further reading

The Data Model chapter of the Python Language Reference starts with a clear explanation of object identities and values.

Wesley Chun, author of the Core Python series of books, made a great presentation about many of the topics covered in this chapter during OSCON 2013. You can download the slides from the Python 103: Memory Model & Best Practices talk page. There is also a You Tube video of a longer presentation Wesley gave at EuroPython 2011, covering not only the theme of this chapter but also the use of special methods.

Doug Hellmann wrote a long series of excellent blog posts titled Python Module of the Week which became a book, The Python Standard Library by Example. His posts "copy – Duplicate objects" and weakref – Garbage-collectable references to objects cover some of the topics we just discussed.

More information on the CPython generational garbage collector can be found in the gc module documentation, which starts with the sentence "This module provides an interface to the optional garbage collector." The "optional" qualifier here may be surprising, but the the Data Model chapter also states:

> An implementation is allowed to postpone garbage collection or omit it altogether — it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable.

Fredrik Lundh — creator of key libraries like ElementTree, Tkinter and the PIL image library — has a short post about the Python garbage collector titled How does Python manage memory? He emphasizes that the garbage collector is an implementation feature that behaves differently across Python interpreters. For example, Jython uses the Java garbage collector.

The CPython 3.4 garbage collector improved handling of objects with a `__del__` method, as described in PEP 442 — Safe object finalization.

The English language Wikipedia has an article about string interning, mentioning the use of this technique in several languages, including Python.

Soapbox

Equal treatment to all objects

I learned Java before I discovered Python. The == operator in Java never felt right for me. It is much more common for programmers to care about equality than identity, but for objects (not primitive types) the Java == compares references, and not object values. Even for something as basic as comparing strings, Java forces you to use the `.equals` method. Even then, there is another catch: if you write `a.equals(b)` and `a` is `null` you get a null pointer exception. The Java designers felt the need to overload + for strings, so why not go ahead and overload == as well?

Python gets this right. The == operator compares object values and `is` compares references. And because Python has operator overloading, == works sensibly with all objects in the standard library, including `None`, which is a proper object, unlike Java's `null`.

And of course, you can define `__eq__` in your own classes to decide what == means for your instances. If you don't override `__eq__`, the method inherited from `object` compares object ids, so the fallback is that every instance of a user-defined class is considered different.

These are some of the things that made me switch from Java to Python as soon as I finished reading the Python Tutorial one afternoon in September, 1998.

Mutability

This chapter would be redundant if all Python objects were immutable. When you are dealing with unchanging objects, it makes no difference whether variables hold the actual objects or references to shared objects. If `a == b` is true, and neither object can change, they might as well be the same. That's why string interning is safe. Object identity becomes important only when objects are mutable.

In "pure" functional programming, all data is immutable: appending to a collection actually creates a new collection. Python, however, is not a functional language, much less a pure one. Instances of user-defined classes are mutable by default in Python — as in most object-oriented languages. When creating your own objects, you have to be extra careful to make them immutable, if that is a requirement. Every attribute of the object must also be immutable, otherwise you end up with something like the `tuple`: immutable as far as object ids go, but the value of a `tuple` may change if it holds a mutable object.

Mutable objects are also the main reason why programming with threads is so hard to get right: threads mutating objects without proper synchronization produce corrupted data. Excessive synchronization, on the other hand, causes deadlocks.

Object destruction and garbage collection

There is no mechanism is Python to directly destroy an object, and this omission is actually a great feature: if you could destroy an object at any time, what would happen to existing strong references pointing to it?

Garbage collection in CPython is done primarily by reference counting, which is easy to implement, but is prone to memory leaking when there are reference cycles, so with version 2.0 (Oct. 2000) a generational garbage collector was implemented, and it is able to dispose of unreachable objects kept alive by reference cycles.

But the reference counting is still there as a baseline, and it causes the immediate disposal of objects with zero references. This means that, in CPython — at least for now — it's safe to write this:

```
open('test.txt', 'wt', encoding='utf-8').write('1, 2, 3')
```

That code is safe because the reference count of the file object will be zero after the `write` method returns, and Python will immediately close the file before destroying the object representing it in memory. However, the same line is not safe in Jython or IronPython that use the garbage collector of their host runtimes (the Java VM and the .net CLR), which are more sophisticated but do not rely on reference counting and may take longer to destroy the object and close the file. In all cases, including CPython, the best practice is to explicitly close the file, and the most reliable way of doing it is using the `with` statement, which guarantees that the file will be closed even if exceptions are raised while it is open. Using `with`, the previous snippet becomes:

```
with open('test.txt', 'wt', encoding='utf-8') as fp:
    fp.write('1, 2, 3')
```

If you are into the subject of garbage collectors, you may want to read Thomas Perl's paper Python Garbage Collector Implementations: CPython, PyPy and GaS where I learned the bit about the safety of the `open().write()` in CPython.

Parameter passing: call by sharing

A popular way of explaining how parameter passing works in Python is the phrase: "Parameters are passed by value, but the values are references". This not wrong, but causes confusion because the most common parameter passing modes in older languages are *call by value* (the function gets a copy of the argument) and *call by reference* (the function gets a pointer to the argument). In Python, the function gets a copy of the arguments, but the arguments are always references. So the value of the referenced objects may be changed, if they are mutable, but their identity cannot. Also, because the function gets a copy of the reference in an argument, rebinding it has no effect outside of the function. I adopted the term *call by sharing* after reading up on the subject in Programming Language Pragmatics, 3e, Michael L. Scott, section 8.3.1: Parameter modes (p. 396).

The full quote of Alice and the Knights's song

I love this passage, but it was too long as a chapter opener. So here is the complete dialog about the Knight's song, its name, and how the song and its name are called:

> 'You are sad,' the Knight said in an anxious tone: 'let me sing you a song to comfort you.'
>
> 'Is it very long?' Alice asked, for she had heard a good deal of poetry that day.
>
> 'It's long,' said the Knight, 'but very, VERY beautiful. Everybody that hears me sing it—either it brings the TEARS into their eyes, or else—'
>
> 'Or else what?' said Alice, for the Knight had made a sudden pause.
>
> 'Or else it doesn't, you know. The name of the song is called "HADDOCKS' EYES".'

‘Oh, that’s the name of the song, is it?’ Alice said, trying to feel interested.

‘No, you don’t understand,’ the Knight said, looking a little vexed. ‘That’s what the name is CALLED. The name really IS “THE AGED AGED MAN”.’

‘Then I ought to have said “That’s what the SONG is called”?’ Alice corrected herself.

‘No, you oughtn’t: that’s quite another thing! The SONG is called “WAYS AND MEANS”: but that’s only what it’s CALLED, you know!’

‘Well, what IS the song, then?’ said Alice, who was by this time completely bewildered.

‘I was coming to that,’ the Knight said. ‘The song really IS “A-SITTING ON A GATE”: and the tune’s my own invention.’

— Through The Looking-glass -- Chapter VIII. ‘It’s my own Invention’
Lewis Carroll

CHAPTER 9
A Pythonic object

> Never, ever use two leading underscores. This is annoyingly private[1].
>
> — Ian Bicking
> *creator of pip, virtualenv, Paste and many other projects*

Thanks to the Python data model, your user-defined types can behave as naturally as the built-in types. And this can be accomplished without inheritance, in the spirit of *duck typing*: you just implement the methods needed for your objects to behave as expected.

In previous chapters we presented the structure and behavior of many built-in objects. We will now build user-defined classes that behave as real Python objects.

This chapter starts where Chapter 1 ended, by showing how to implement several special methods that are commonly seen in Python objects of many different types.

In this chapter we will see how to:

- Support the built in functions that produce alternate object representations (e.g. `repr()`, `bytes()` etc).
- Implement an alternate constructor as a class method.
- Extend the format mini-language used by the `format()` built-in and the `str.format()` method.
- Provide read-only access to attributes.
- Make an object hashable for use in sets and as `dict` keys.
- Save memory with the use of `__slots__`.

[1] From the Paste Style Guide

We'll do all that as we develop a simple 2D Euclidean vector type.

The evolution of the example will be paused to discuss two conceptual topics:

- How and when to use the `@classmethod` and `@staticmethod` decorators.
- Private and protected attributes in Python: usage, conventions and limitations.

Let's get started with the object representation methods.

Object representations

Every object-oriented language has at least one standard way of getting a string representation from any object. Python has two:

`repr()`
: Return a string representing the object as the developer wants to see it.

`str()`
: Return a string representing the object as the user wants to see it.

As you know, we implement the special methods `__repr__` and `__str__` to support `repr()` and `str()`.

There are two additional special methods to support alternate representations of objects: `__bytes__` and `__format__`. The `__bytes__` method is analogous to `__str__`: it's called by `bytes()` to get the object represented as a byte sequence. Regarding `__format__`, both the built-in function `format()` and the `str.format()` method call it to get string displays of objects using special formatting codes. We'll cover `__bytes__` in the next example, and `__format__` after that.

> If you're coming from Python 2, remember that in Python 3 `__repr__`, `__str__` and `__format__` must always return Unicode strings (type `str`). Only `__bytes__` is supposed to return a byte sequence (type `bytes`).

`Vector` class redux

To demonstrate the many methods used to generate object representations, we'll use a 2D `Vector2d` class similar to the one we saw in Chapter 1. We will build on it in this and future sections. Example 9-1 shows the basic behavior we expect from a `Vector2d` instance.

Example 9-1. `Vector2d` instances have several representations

```
>>> v1 = Vector2d(3, 4)
>>> print(v1.x, v1.y)  ❶
3.0 4.0
>>> x, y = v1  ❷
>>> x, y
(3.0, 4.0)
>>> v1  ❸
Vector2d(3.0, 4.0)
>>> v1_clone = eval(repr(v1))  ❹
>>> v1 == v1_clone  ❺
True
>>> print(v1)  ❻
(3.0, 4.0)
>>> octets = bytes(v1)  ❼
>>> octets
b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x10@'
>>> abs(v1)  ❽
5.0
>>> bool(v1), bool(Vector2d(0, 0))  ❾
(True, False)
```

❶ The components of a `Vector2d` can be accessed directly as attributes (no getter method calls).

❷ A `Vector2d` can be unpacked to a tuple of variables.

❸ The `repr` of a `Vector2d` emulates the source code for constructing the instance.

❹ Using `eval` here shows that the `repr` of a `Vector2d` is a faithful representation of its constructor call[2].

❺ `Vector2d` supports comparison with `==`; this is useful for testing.

❻ `print` calls `str`, which for `Vector2d` produces an ordered pair display.

❼ `bytes` uses the `__bytes__` method to produce a binary representation.

❽ `abs` uses the `__abs__` method to return the magnitude of the `Vector2d`.

❾ `bool` uses the `__bool__` method to return `False` for a `Vector2d` of zero magnitude or `True` otherwise.

`Vector2d` from Example 9-1 is implemented in `vector2d_v0.py` (Example 9-2). The code is based on Example 1-2, but the infix operators will be implemented in Chapter 13 — except for `==` which is useful for testing. At this point, `Vector2d` uses several special methods to provide operations that a Pythonista expects in a well designed object.

[2] I used `eval` to clone the object here just to make a point about `repr`; to clone an instance the `copy.copy` function is safer and faster.

Example 9-2. `vector2d_v0.py`: methods so far are all special methods.

```
from array import array
import math


class Vector2d:
    typecode = 'd'  ❶

    def __init__(self, x, y):
        self.x = float(x)    ❷
        self.y = float(y)

    def __iter__(self):
        return (i for i in (self.x, self.y))  ❸

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)  ❹

    def __str__(self):
        return str(tuple(self))  ❺

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +  ❻
                bytes(array(self.typecode, self)))  ❼

    def __eq__(self, other):
        return tuple(self) == tuple(other)  ❽

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

    def __bool__(self):
        return bool(abs(self))  ❿
```

❶ `typecode` is a class attribute we'll use when converting `Vector2d` instances to/from `bytes`.

❷ Converting `x` and `y` to `float` in `__init__` catches errors early, in case `Vector2d` is called with unsuitable arguments.

❸ `__iter__` makes a `Vector2d` iterable; this is what makes unpacking work, e.g, `x, y = my_vector`. We implement it simply by using a generator expression to yield the components one after the other[3].

[3] This line could also be written as `yield self.x; yield.self.y`. I have a lot more to say about the `__iter__` special method, generator expressions and the `yield` keyword in Chapter 14.

❹ `__repr__` builds a string by interpolating the components with `{!r}` to get their `repr`; because `Vector2d` is iterable, `*self` feeds the `x` and `y` components to `format`.

❺ From an iterable `Vector2d` it's easy to build a `tuple` for display as an ordered pair.

❻ To generate `bytes`, we convert the typecode to `bytes` and concatenate…

❼ …`bytes` converted from an `array` built by iterating over the instance.

❽ To quickly compare all components, build tuples out of the operands. This works for operands that are instances of `Vector2d`, but has issues. See warning below.

❾ The magnitude is the length of the hypotenuse of the triangle formed by the `x` and `y` components.

❿ `__bool__` uses `abs(self)` to compute the magnitude, then converts it to `bool`, so `0.0` becomes `False`, non-zero is `True`.

> Method `__eq__` in Example 9-2 works for `Vector2d` operands but also returns `True` when comparing `Vector2d` instances to other iterables holding the same numeric values, e.g. `Vector(3, 4) == [3, 4]`. This may be considered a feature or a bug. Further discussion needs to wait until Chapter 13, when we cover operator overloading.

We have a fairly complete set of basic methods, but one obvious operation is missing: rebuilding a `Vector2d` from the binary representation produced by `bytes()`.

An alternative constructor

Since we can export a `Vector2d` as bytes, naturally we need a method that imports a `Vector2d` from a binary sequence. Looking at the standard library for inspiration, we find that `array.array` has a class method named `.frombytes` that suits our purpose — we saw it in "Arrays" on page 48. We adopt its name and use its functionality in a class method for `Vector2d` in `vector2d_v1.py` (Example 9-3).

Example 9-3. Part of `vector2d_v1.py`: this snippet shows only the `frombytes` class method, added to the `Vector2d` definition in `vector2d_v0.py` Example 9-2.

```
    @classmethod  ❶
    def frombytes(cls, octets):  ❷
        typecode = chr(octets[0])  ❸
        memv = memoryview(octets[1:]).cast(typecode)  ❹
        return cls(*memv)  ❺
```

❶ Class method is modified by the classmethod decorator.

❷ No self argument; instead, the class itself is passed as cls.

❸ Read the typecode from the first byte.

❹ Create a memoryview from the octets binary sequence and use the typecode to cast it[4].

❺ Unpack the memoryview resulting from the cast into the pair of arguments needed for the constructor.

Since we just used a classmethod decorator, and it is very Python-specific, let's have a word about it.

classmethod versus staticmethod

The classmethod decorator is not mentioned in the Python tutorial and neither is staticmethod. Anyone who has learned OO in Java may wonder why Python has both of these decorators and not just one of them.

Let's start with classmethod. Example 9-3 shows its use: to define a method that operates on the class and not on instances. classmethod changes the way the method is called, so it receives the class itself as the first argument, instead of an instance. Its most common use is for alternate constructors, like frombytes in Example 9-3. Note how the last line of frombytes actually uses the cls argument by invoking it to build a new instance: cls(*memv). By convention, the first parameter of a class method should be named cls (but Python doesn't care how it's named).

In contrast, the staticmethod decorator changes a method so that it receives no special first argument. In essence, a static method is just like a plain function that happens to live in a class body, instead of being defined at the module level. Example 9-4 contrasts the operation of classmethod and staticmethod.

Example 9-4. Comparing behaviors or classmethod and staticmethod.

```
>>> class Demo:
...     @classmethod
...     def klassmeth(*args):
...         return args  # ❶
...     @staticmethod
...     def statmeth(*args):
...         return args  # ❷
...
>>> Demo.klassmeth()  # ❸
(<class '__main__.Demo'>,)
```

[4] We had a brief introduction to memoryview, explaining its .cast method in "Memory views" on page 51.

```
>>> Demo.klassmeth('spam')
(<class '__main__.Demo'>, 'spam')
>>> Demo.statmeth()   # ❹
()
>>> Demo.statmeth('spam')
('spam',)
```

❶ `klassmeth` just returns all positional arguments.

❷ `statmeth` does the same.

❸ No matter how you invoke it, `Demo.klassmeth` receives the `Demo` class as the first argument.

❹ `Demo.statmeth` behaves just like a plain old function.

> The `classmethod` decorator is clearly useful, but I've never seen a compelling use case for `staticmethod`. If you want do define a function that does not interact with the class, just define it in the module. Maybe the function is closely related even if it never touches the class, so you want to them nearby in the code. Even so, defining the function right before or after the class in the same module is close enough for all practical purposes[5]

Now that we've seen what `classmethod` is good for (and that `staticmethod` is not very useful), let's go back to the issue of object representation and see how to support formatted output.

Formatted displays

The `format()` built-in function and the `str.format()` method delegate the actual formatting to each type by calling their `.__format__(format_spec)` method. The `format_spec` is a formatting specifier, which is either:

- The second argument in `format(my_obj, format_spec)`, or
- whatever appears after the colon in a replacement field delimited with `{}` inside a format string used with `str.format()`.

For example:

[5] Leonardo Rochael, one of the technical reviewers of this book disagrees with my low opinion of `staticmethod` and recommends the blog post The definitive guide on how to use static, class or abstract methods in Python by Julien Danjou as a counter-argument. Danjou's post is very good, I do recommend it. But it wasn't enough to change my mind about `staticmethod`. You'll have to make your own mind about it.

```
>>> brl = 1/2.43  # BRL to USD currency conversion rate
>>> brl
0.4115226337448559
>>> format(brl, '0.4f')  # ❶
'0.4115'
>>> '1 BRL = {rate:0.2f} USD'.format(rate=brl)  # ❷
'1 BRL = 0.41 USD'
```

❶ Formatting specifier is `'0.4f'`.

❷ Formatting specifier is `'0.2f'`. The `'rate'` substring in the replacement field is called the field name. It's unrelated to the formatting specifier, but determines which argument of `.format()` goes into that replacement field.

The last callout above makes an important point: a format string such as `'{0.mass:5.3e}'` actually uses two separate notations. The `'0.mass'` to the left of the colon is the `field_name` part of the replacement field syntax; the `'5.3e'` after the colon is the formatting specifier. The notation used in the formatting specifier is called the Format Specification Mini-Language.

> If `format()` and `str.format()` are new to you, classroom experience has shown that it's best to study the `format()` function first, which uses just the Format Specification Mini-Language. After you get the gist of that, read Format String Syntax to learn about the `{:}` replacement field notation, used in the `str.format()` method — including the `!s`, `!r` and `!a` conversion flags.

A few built-in types have their own presentation codes in the Format Specification Mini-Language. For example — among several other codes — the `int` type supports `b` and `x` for base 2 and base 16 output, respectively, while `float` implements `f` for a fixed-point display and `%` for a percentage display:

```
>>> format(42, 'b')
'101010'
>>> format(2/3, '.1%')
'66.7%'
```

The Format Specification Mini-Language is extensible because each class gets to interpret the `format_spec` argument as it likes. For instance, the classes in the `datetime` module use the same format codes in the `strftime()` functions and in their `__format__` methods. Here are a couple of examples using the `format()` built-in and the `str.format()` method.

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> format(now, '%H:%M:%S')
'18:49:05'
```

```
>>> "It's now {:%I:%M %p}".format(now)
"It's now 06:49 PM"
```

If a class has no `__format__`, the method inherited from `object` returns `str(my_object)`. Since `Vector2d` has a `__str__`, this works:

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
```

However, if you pass a format specifier, `object.__format__` raises `TypeError`:

```
>>> format(v1, '.3f')
Traceback (most recent call last):
  ...
TypeError: non-empty format string passed to object.__format__
```

We will fix that by implementing our own format mini-language. The first step will be to assume the format specifier provided by the user is intended to format each `float` component of the vector. This is the result we want:

```
>>> v1 = Vector2d(3, 4)
>>> format(v1)
'(3.0, 4.0)'
>>> format(v1, '.2f')
'(3.00, 4.00)'
>>> format(v1, '.3e')
'(3.000e+00, 4.000e+00)'
```

Example 9-5 implements `__format__` to produce the displays just shown.

Example 9-5. `Vector2d.__format__` method, take #1

```
# inside the Vector2d class

def __format__(self, fmt_spec=''):
    components = (format(c, fmt_spec) for c in self)  # ❶
    return '({}, {})'.format(*components)  # ❷
```

❶ Use the `format` built-in to apply the `fmt_spec` to each vector component, building an iterable of formatted strings.

❷ Plug the formatted strings in the formula `'(x, y)'`.

Now let's add a custom formatting code to our mini-language: if the format specifier ends with a `'p'`, we'll display the vector in polar coordinates: `<r, θ>`, where `r` is the magnitude and θ (theta) is the angle in radians. The rest of the format specifier (whatever comes before the `'p'`) will be used as before.

When choosing the letter for the custom format code I avoided overlapping with codes used by other types. In Format Specification Mini-Language we see that integers use the codes 'bcdoxXn', floats use 'eEfFgGn%' and strings use 's'. So I picked 'p' for polar coordinates. Since each class interprets these codes independently, reusing a code letter in a custom format for a new type is not an error, but may be confusing to users.

To generate polar coordinates we already have the `__abs__` method for the magnitude, and we'll code a simple `angle` method using the `math.atan2()` function to get the angle. This is the code:

```
    # inside the Vector2d class

    def angle(self):
        return math.atan2(self.y, self.x)
```

With that, we can enhance our `__format__` to produce polar coordinates. See Example 9-6.

Example 9-6. `Vector2d.__format__` method, take #2, now with polar coordinates.

```
def __format__(self, fmt_spec=''):
    if fmt_spec.endswith('p'):  ❶
        fmt_spec = fmt_spec[:-1]  ❷
        coords = (abs(self), self.angle())  ❸
        outer_fmt = '<{}, {}>'  ❹
    else:
        coords = self  ❺
        outer_fmt = '({}, {})'  ❻
    components = (format(c, fmt_spec) for c in coords)  ❼
    return outer_fmt.format(*components)  ❽
```

❶ Format ends with `'p'`: use polar coordinates.

❷ Remove `'p'` suffix from `fmt_spec`.

❸ Build `tuple` of polar coordinates: `(magnitude, angle)`.

❹ Configure outer format with angle brackets.

❺ Otherwise, use `x, y` components of `self` for rectangular coordinates.

❻ Configure outer format with parenthesis.

❼ Generate iterable with components as formatted strings.

❽ Plug formatted strings into outer format.

With Example 9-6 we get results similar to these:

```
>>> format(Vector2d(1, 1), 'p')
'<1.4142135623730951, 0.7853981633974483>'
>>> format(Vector2d(1, 1), '.3ep')
'<1.414e+00, 7.854e-01>'
>>> format(Vector2d(1, 1), '0.5fp')
'<1.41421, 0.78540>'
```

As this section shows, it's not hard to extend the format specification mini-language to support user-defined types.

Now let's move to a subject that's not just about appearances: we will make our Vector2d hashable, so we can build sets of vectors, or use them as dict keys. But before we can do that, we must make vectors immutable. We'll do what it takes next.

A hashable Vector2d

As defined, so far our Vector2d instances are unhashable, so we can't put them in a set:

```
>>> v1 = Vector2d(3, 4)
>>> hash(v1)
Traceback (most recent call last):
  ...
TypeError: unhashable type: 'Vector2d'
>>> set([v1])
Traceback (most recent call last):
  ...
TypeError: unhashable type: 'Vector2d'
```

To make a Vector2d hashable we must implement __hash__ (__eq__ is also required, and we already have it). We also need to make vector instances immutable, as we've seen in "What is hashable?" on page 64.

Right now, anyone can do v1.x = 7 and there is nothing in the code to suggest that changing a Vector2d is forbidden. This is the behavior we want:

```
>>> v1.x, v1.y
(3.0, 4.0)
>>> v1.x = 7
Traceback (most recent call last):
  ...
AttributeError: can't set attribute
```

We'll do that by making the x and y components read-only properties in Example 9-7.

Example 9-7. vector2d_v3.py: only the changes needed to make Vector2d immutable are shown here; see full listing in Example 9-9.

```
class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x)   ❶
        self.__y = float(y)

    @property   ❷
    def x(self):   ❸
        return self.__x   ❹

    @property   ❺
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y))   ❻

    # remaining methods follow (omitted in book listing)
```

❶ Use exactly two leading underscores (with zero or one trailing underscore) to make an attribute private[6].

❷ The `@property` decorator marks the getter method of a property.

❸ The getter method is named after the public property it exposes: `x`.

❹ Just return `self.__x`.

❺ Repeat same formula for `y` property.

❻ Every method that just reads the `x`, `y` components can stay as they were, reading the public properties via `self.x` and `self.y` instead of the private attribute, so this listing omits the rest of the code for the class.

> `Vector.x` and `Vector.y` are examples of read-only properties. Read-write properties will be covered in Chapter 19, where we dive deeper into the `@property`.

Now that our vectors are reasonably immutable, we can implement the `__hash__` method. It should return an `int` and ideally take into account the hashes of the object attributes that are also used in the `__eq__` method, because objects that compare equal should have the same hash. The `__hash__` special method documentation suggests using the bitwise xor operator (`^`) to mix the hashes of the components, so that's what we do. The code for our `Vector2d.__hash__` method is really simple:

[6] This is not how Ian Bicking would do it; recall the quote at the start of the chapter. The pros and cons of private attributes are the subject of the upcoming "Private and "protected" attributes in Python" on page 263.

Example 9-8. `vector2d_v3.py`: implementation of `__hash__`.

```
# inside class Vector2d:

def __hash__(self):
    return hash(self.x) ^ hash(self.y)
```

With the addition of the `__hash__` method, we now have hashable vectors:

```
>>> v1 = Vector2d(3, 4)
>>> v2 = Vector2d(3.1, 4.2)
>>> hash(v1), hash(v2)
(7, 384307168202284039)
>>> set([v1, v2])
{Vector2d(3.1, 4.2), Vector2d(3.0, 4.0)}
```

It's not strictly necessary to implement properties or otherwise protect the instance attributes to create a hashable type. Implementing `__hash__` and `__eq__` correctly is all it takes. But the hash value of an instance is never supposed to change, so this provides an excellent opportunity to talk about read-only properties.

If you are creating a type that has a sensible scalar numeric value, you may also implement the `__int__` and `__float__` methods, invoked by the `int()`, `float()` constructors — which are used for type coercion in some contexts. There's also a `__complex__` method to support the `complex()` built-in constructor. Perhaps `Vector2d` should provide `__complex__`, but I'll leave that as an exercise for you.

We have been working on `Vector2d` for a while, showing just snippets, so Example 9-9 is a consolidated, full listing of `vector2d_v3.py`, including all the doctests I used when developing it.

Example 9-9. `vector2d_v3.py`: the full monty.

```
"""
A 2-dimensional vector class

    >>> v1 = Vector2d(3, 4)
    >>> print(v1.x, v1.y)
    3.0 4.0
    >>> x, y = v1
    >>> x, y
    (3.0, 4.0)
    >>> v1
    Vector2d(3.0, 4.0)
    >>> v1_clone = eval(repr(v1))
    >>> v1 == v1_clone
    True
    >>> print(v1)
    (3.0, 4.0)
    >>> octets = bytes(v1)
    >>> octets
    b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x10@'
    >>> abs(v1)
    5.0
    >>> bool(v1), bool(Vector2d(0, 0))
    (True, False)


Test of ``.frombytes()`` class method:

    >>> v1_clone = Vector2d.frombytes(bytes(v1))
    >>> v1_clone
    Vector2d(3.0, 4.0)
    >>> v1 == v1_clone
    True


Tests of ``format()`` with Cartesian coordinates:

    >>> format(v1)
    '(3.0, 4.0)'
    >>> format(v1, '.2f')
    '(3.00, 4.00)'
    >>> format(v1, '.3e')
    '(3.000e+00, 4.000e+00)'


Tests of the ``angle`` method::

    >>> Vector2d(0, 0).angle()
    0.0
    >>> Vector2d(1, 0).angle()
    0.0
    >>> epsilon = 10**-8
    >>> abs(Vector2d(0, 1).angle() - math.pi/2) < epsilon
    True
    >>> abs(Vector2d(1, 1).angle() - math.pi/4) < epsilon
    True


Tests of ``format()`` with polar coordinates:

    >>> format(Vector2d(1, 1), 'p')  # doctest:+ELLIPSIS
    '<1.414213..., 0.785398...>'
    >>> format(Vector2d(1, 1), '.3ep')
    '<1.414e+00, 7.854e-01>'
    >>> format(Vector2d(1, 1), '0.5fp')
    '<1.41421, 0.78540>'
```

```
Tests of `x` and `y` read-only properties:

    >>> v1.x, v1.y
    (3.0, 4.0)
    >>> v1.x = 123
    Traceback (most recent call last):
      ...
    AttributeError: can't set attribute


Tests of hashing:

    >>> v1 = Vector2d(3, 4)
    >>> v2 = Vector2d(3.1, 4.2)
    >>> hash(v1), hash(v2)
    (7, 384307168202284039)
    >>> len(set([v1, v2]))
    2

"""

from array import array
import math

class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x)
        self.__y = float(y)

    @property
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y))

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(array(self.typecode, self)))
    def __eq__(self, other):
        return tuple(self) == tuple(other)

    def __hash__(self):
        return hash(self.x) ^ hash(self.y)

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

    def __bool__(self):
        return bool(abs(self))

    def angle(self):
        return math.atan2(self.y, self.x)

    def __format__(self, fmt_spec=''):
        if fmt_spec.endswith('p'):
            fmt_spec = fmt_spec[:-1]
            coords = (abs(self), self.angle())
            outer_fmt = '<{}, {}>'
        else:
            coords = self
            outer_fmt = '({}, {})'
        components = (format(c, fmt_spec) for c in coords)
        return outer_fmt.format(*components)

    @classmethod
    def frombytes(cls, octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(*memv)
```

To recap, in this and the previous sections we saw some essential special methods that you may want to implement to have a full-fledged object. Of course, it is a bad idea to implement all of these methods if your application has no real use for them. Customers don't care if your objects are "Pythonic" or not.

As coded in Example 9-9, `Vector2d` is a didactic example with a laundry list of special methods related to object representation, not a template for every user-defined class.

In the next section we'll take a break from `Vector2d` to discuss the design and drawbacks of the private attribute mechanism in Python — the double-underscore prefix in `self.__x`.

Private and "protected" attributes in Python

In Python there is no way to create private variables in the strong sense of the private modifier in Java. What we have in Python is a simple mechanism to prevent accidental overwriting of a "private" attribute in a subclass.

Consider this scenario: someone wrote a class Dog which uses a mood instance attribute internally, without exposing it. You need to subclass Dog as Beagle. If you create your own instance attribute mood without being aware of the name clash, you will clobber the mood attribute used by the methods inherited from Dog. This would be a pain to debug.

To prevent this, if you name an instance attribute in the form __mood (two leading underscores and zero or at most one trailing underscore), Python stores the name in the instance __dict__ prefixed with a leading underscore and the class name, so in the Dog class, __mood becomes _Dog__mood, and in Beagle it's _Beagle__mood. This language feature goes by the lovely name of *name mangling*.

In the Vector2d class from Example 9-7 this is the result:

Example 9-10. Private attribute names are "mangled" by prefixing the _ and the class name.

```
>>> v1 = Vector2d(3, 4)
>>> v1.__dict__
{'_Vector2d__y': 4.0, '_Vector2d__x': 3.0}
>>> v1._Vector2d__x
3.0
```

Name mangling is about safety, not security: it's designed to prevent accidental access and not intentional wrongdoing (Figure 9-1 illustrates another safety device).

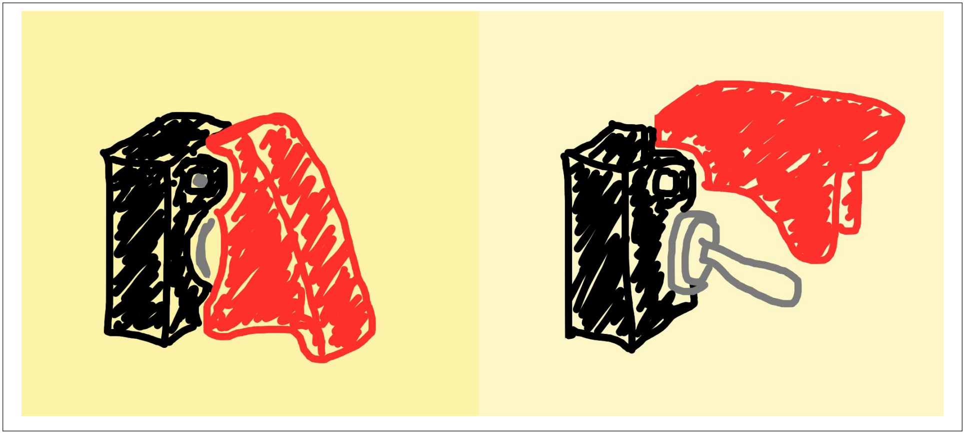


Figure 9-1. A cover on a switch is a safety device, not a security one: it prevents accidental activation, not malicious use.

Anyone who knows how private names are mangled can read the private attribute directly, as the last line of Example 9-10 shows — that's actually useful for debugging and serialization. They can also directly assign a value to a private component of a `Vector2d` by simply writing `v1._Vector__x = 7`. But if you are doing that in production code, you can't complain if something blows up.

The name mangling functionality is not loved by all Pythonistas, neither is the skewed look of names written as `self.__x`. Some prefer to avoid this syntax and use just one underscore prefix to "protect" attributes by convention, eg. `self._x`. Critics of the automatic double-underscore mangling suggest that concerns about accidental attribute clobbering should be addressed by naming conventions. This is the full quote from the prolific Ian Bicking, cited at the top of this chapter:

> Never, ever use two leading underscores. This is annoyingly private. If name clashes are a concern, use explicit name mangling instead (e.g., `_MyThing_blahblah`). This is essentially the same thing as double-underscore, only it's transparent where double underscore obscures[7].

The single underscore prefix has no special meaning to the Python interpreter when used in attribute names, but it's a very strong convention among Python programmers that you should not access such attributes from outside the class[8]. It's easy to respect the

[7] From the Paste Style Guide

[8] In modules, a single _ in front of a top-level name does have an effect: if you write `from mymod import *` the names with a _ prefix are not imported from `mymod`. But you can still write `from mymod import _privatefunc`. This is explained in the Python Tutorial, section 6.1. More on Modules

privacy of an object that marks its attributes with a single `_`, just as it's easy respect the convention that variables in `ALL_CAPS` should be treated as constants.

Attributes with a single `_` prefix are called "protected" in some corners of the Python documentation[9]. The practice of "protecting" attributes by convention with the form `self._x` is widespread, but calling that a "protected" attribute is not so common. Some even call that a "private" attribute.

To conclude: the `Vector2d` components are "private" and our `Vector2d` instances are "immutable" — with scare quotes — because there is no way to make them really private and immutable[10].

We'll now come back to our `Vector2d` class. In this final section we cover a special attribute (not a method) that affects the internal storage of an object, with potentially huge impact on the use of memory but little effect on its public interface: `__slots__`.

Saving space with the `__slots__` class attribute

By default, Python stores instance attributes in a per-instance `dict` named `__dict__`. As we saw in "Practical consequences of how `dict` works" on page 90, dictionaries have a significant memory overhead because of the underlying hash table used to provide fast access. If you are dealing with millions of instances with few attributes, the `__slots__` class attribute can save a lot of memory, by letting the interpreter store the instance attributes in a `tuple` instead of a `dict`.

A `__slots__` attribute inherited from a superclass has no effect. Python only takes into account `__slots__` attributes defined in each class individually.

To define `__slots__` you create a class attribute with that name and assign it an iterable of `str` with identifiers for the instance attributes. I like to use a `tuple` for that, because it conveys the message that the `__slots__` definition cannot change. See Example 9-11.

Example 9-11. `vector2d_v3_slots.py`: the `__slots__` attribute is the only addition to `Vector2d`.

```python
class Vector2d:
    __slots__ = ('__x', '__y')
```

[9] One example is in the gettext module docs.

[10] If this state of affairs depresses you, and makes you wish Python was more like Java in this regard, don't read my discussion of the relative strength of the Java `private` modifier in "Soapbox" on page 272.

```
    typecode = 'd'

    # methods follow (omitted in book listing)
```

By defining `__slots__` in the class, you are telling the interpreter: “These are all the instance attributes in this class.” Python then stores them in a tuple-like structure in each instance, avoiding the memory overhead of the per-instance `__dict__`. This can make a huge difference in memory usage if your have millions of instances active at the same time.

If you are handling millions of objects with numeric data, you should really be using NumPy arrays (“NumPy and SciPy” on page 52), which are not only memory-efficient but have highly optimized functions for numeric processing, many of which operate on the entire array at once. I designed the `Vector2d` class just to provide context when discussing special methods, because I try to avoid vague `foo` and `bar` examples when I can.

Example 9-12 shows two runs of a script that simply builds a `list`, using a list comprehension, with 10,000,000 instances of `Vector2d`. The `mem_test.py` script takes the name of a module with a `Vector2d` class variant as command-line argument. In the first run I am using `vector2d_v3.Vector2d` (from Example 9-7), in the second run the `__slots__` version of `vector2d_v3_slots.Vector2d` is used.

Example 9-12. `mem_test.py` creates 10 million `Vector2d` instances using the class defined in the named module (eg. `vector2d_v3.py`).

```
$ time python3 mem_test.py vector2d_v3.py
Selected Vector2d type: vector2d_v3.Vector2d
Creating 10,000,000 Vector2d instances
Initial RAM usage:      5,623,808
  Final RAM usage:  1,558,482,944

real  0m16.721s
user  0m15.568s
sys 0m1.149s
$ time python3 mem_test.py vector2d_v3_slots.py
Selected Vector2d type: vector2d_v3_slots.Vector2d
Creating 10,000,000 Vector2d instances
Initial RAM usage:      5,718,016
  Final RAM usage:    655,466,496

real  0m13.605s
user  0m13.163s
sys 0m0.434s
```

As Example 9-12 demo reveals, the RAM footprint of the script grows to 1,5 GB when instance `__dict__` is used in each of the 10 million `Vector2d` instances, but that is reduced to 655 MB when `Vector2d` has a `__slots__` attribute. The `__slots__` version is also faster. The `mem_test.py` script in this test basically deals with loading a module, checking memory usage and formatting results. The code is not really relevant here so it's in Appendix A, Example A-4.

> When `__slots__` is specified in a class, its instances will not be allowed to have any other attributes apart from those named in `__slots__`. This is really a side-effect, and not the reason why `__slots__` exists. It's considered bad form to use `__slots__` just to prevent users of your class to create new attributes in the instances if they want to. `__slots__` should used for optimization, not for programmer restraint.

It may be possible, however, to "save memory and eat it too": if you add the `'__dict__'` name to the `__slots__` list, your instances will keep attributes named in `__slots__` in the per-instance tuple, but will also support dynamically created attributes, which will be stored in the usual `__dict__`. Of course, having `'__dict__'` in `__slots__` may entirely defeat its purpose, depending on the number of static and dynamic attributes in each instance and how they are used. Careless optimization is even worse than premature optimization.

There is another special per-instance attribute that you may want to keep: the `__weakref__` attribute is necessary for an object to support weak references (covered in "Weak references" on page 236). That attribute is present by default in instances of user-defined classes. However, if the class defines `__slots__`, and you need the instances to be targets of weak references, then you need to include `'__weakref__'` among the attributes named in `__slots__`.

To summarize, `__slots__` is has some caveats and should not be abused just for the sake of limiting what attributes can be assigned by users. It is mostly useful when working with tabular data such as database records where the schema is fixed by definition and the data sets may be very large. However, if you do this kind of work often, you must check out not only NumPy, but also the pandas data analysis library, which can handle non-numeric data and import/export to many different tabular data formats.

The problems with `__slots__`

To summarize, if well used `__slots__` may provide significant memory savings, but there are a few caveats:

- You must remember to redeclare `__slots__` in each subclass, since the inherited attribute is ignored by the interpreter.
- Instances will only be able to have the attributes listed in `__slots__`, unless you include `'__dict__'` in `__slots__` — but doing so may negate the memory savings.
- Instances cannot be targets of weak references unless you remember to include `'__weakref__'` in `__slots__`.

If your program is not handling millions of instances, it's probably not worth the trouble of creating a somewhat unusual and tricky class whose instances may not accept dynamic attributes or may not support weak references. Like any optimization, `__slots__` should be used only if justified by a present need and when its benefit is proven by careful profiling.

The last topic in this chapter has to do with overriding a class attribute in instances and subclasses.

Overriding class attributes

A distinctive feature of Python is how class attributes can be used as default values for instance attributes. In `Vector2d` there is the `typecode` class attribute. It's used twice in the `__bytes__` method, but we read it as `self.typecode` — by design. Because `Vector2d` instances are created without a `typecode` attribute of their own, `self.typecode` will get the `Vector2d.typecode` class attribute by default.

But if you write to an instance attribute that does not exist, you create a new instance attribute — e.g. a `typecode` instance attribte — the class attribute by the same name is untouched. However, from then on whenever code handling that instance reads `self.typecode`, the instance `typecode` will be retrieved, effectively shadowing the class attribute by the same name. This opens the possibility of customizing an individual instance with a different `typecode`.

The default `Vector2d.typecode` is `'d'`, meaning each vector component will be represented as an 8-byte double precision float when exporting to `bytes`. If we set the `typecode` of a `Vector2d` instance to `'f'` prior to exporting, each component will exported as a 4-byte single precision float. Example 9-13 demonstrates.

We are discussing adding a custom instance attribute, therefore Example 9-13 uses the `Vector2d` implementation without `__slots__` as listed in Example 9-9.

Example 9-13. Customizing an instance by setting the `typecode` attribute that was formerly inherited from the class.

```
>>> from vector2d_v3 import Vector2d
>>> v1 = Vector2d(1.1, 2.2)
>>> dumpd = bytes(v1)
>>> dumpd
b'd\x9a\x99\x99\x99\x99\x99\xf1?\x9a\x99\x99\x99\x99\x99\x01@'
>>> len(dumpd)  # ❶
17
>>> v1.typecode = 'f'  # ❷
>>> dumpf = bytes(v1)
>>> dumpf
b'f\xcd\xcc\x8c?\xcd\xcc\x0c@'
>>> len(dumpf)  # ❸
9
>>> Vector2d.typecode  # ❹
'd'
```

❶ Default `bytes` representation is 17 bytes long.

❷ Set `typecode` to `'f'` in the `v1` instance.

❸ Now the `bytes` dump is 9 bytes long.

❹ `Vector2d.typecode` is unchanged, only the `v1` instance uses `typecode` `'f'`.

Now it should be clear why the `bytes` export of a `Vector2d` is prefixed by the `typecode`: we wanted to support different export formats.

If you want to change a class attribute you must set it on the class directly, not through an instance. You could change the default `typecode` for all instances (that don't have their own `typecode`) by doing this:

```
    >>> Vector2d.typecode = 'f'
```

However, there is an idiomatic Python way of achieving a more permanent effect, and being more explicit about the change. Since class attributes are public, they are inherited by subclasses, so it's common practice to subclass just to customize a class data attribute. The Django class-based views use this technique extensively. Example 9-14 shows how:

Example 9-14. The `ShortVector2d` is a subclass of `Vector2d` which only overwrites the default `typecode`.

```
>>> from vector2d_v3 import Vector2d
>>> class ShortVector2d(Vector2d):  # ❶
...     typecode = 'f'
...
>>> sv = ShortVector2d(1/11, 1/27)  # ❷
>>> sv
ShortVector2d(0.09090909090909091, 0.037037037037037035)  # ❸
```

```
>>> len(bytes(sv))  # ❹
9
```

❶ Create `ShortVector2d` as a `Vector2d` subclass just to overwrite the `typecode` class attribute.

❷ Build `ShortVector2d` instance `sv` for demonstration.

❸ Inspect the `repr` of `sv`.

❹ Check that the length of the exported bytes is 9, not 17 as before.

This example also explains why I did not hardcode the `class_name` in `Vector2d.__repr__`, but instead got it from `type(self).__name__`, like this:

```
    # inside class Vector2d:

    def __repr__(self):
        class_name = type(self).__name__
        return '{}({!r}, {!r})'.format(class_name, *self)
```

If I had hardcoded the `class_name`, subclasses of `Vector2d` like `ShortVector2d` would have to overwrite `__repr__` just to change the `class_name`. By reading the name from the `type` of the instance, I made `__repr__` safer to inherit.

This ends our coverage of implementing a simple class that leverages the data model to play well with the rest of Python — offering different object representations, implementing a custom formatting code, exposing read-only attributes, and supporting `hash()` to integrate with sets and mappings.

Chapter summary

The aim of this chapter was to demonstrate the use of special methods and conventions in the construction of a well-behaved Pythonic class.

Is `vector2d_v3.py` (Example 9-9) more Pythonic than `vector2d_v0.py` (Example 9-2)? The `Vector2d` class in `vector2d_v3.py` certainly exhibits more Python features. But whether the first or the last `Vector2d` implementation is more idiomatic depends on the context where it would be used. Tim Peter's Zen of Python says:

> Simple is better than complex.

A Pythonic object should be as simple as the requirements allow — and not a parade of language features.

But my goal in expanding the `Vector2d` code was to provide context for discussing Python special methods and coding conventions. If you look back at Table 1-1, the several listings in this chapter demonstrated:

- All string/bytes representation methods: `__repr__`, `__str__`, `__format__` and `__bytes__`.
- Several methods for converting an object to a number: `__abs__`, `__bool__`, `__hash__`.
- The `__eq__` operator, to test `bytes` conversion and to enable hashing (along with `__hash__`).

While supporting conversion to `bytes` we also implemented an alternate constructor, `Vector2d.frombytes()` which provided the context for discussing the decorators `@classmethod` (very handy) and `@staticmethod` (not so useful, module level functions are simpler). The `frombytes` method was inspired by it's namesake in the `array.array` class.

We saw that the Format Specification Mini-Language is extensible by implementing a `__format__` method that does some minimal parsing of `format_spec` provided to the `format(obj, format_spec)` built-in or within replacement fields `'{:«format_spec»}'` in strings used with the `str.format` method.

In preparation to make `Vector2d` instances hashable, we made an effort to make them immutable, at least preventing accidental changes by coding the `x` and `y` attributes as private, and exposing them as read-only properties. We then implemented `__hash__` using the recommended technique of xor-ing the hashes of the instance attributes.

We then discussed the memory savings and the caveats of declaring a `__slots__` attribute in `Vector2d`. Because using `__slots__` is somewhat tricky, it really makes sense only when handling a very large number of instances — think millions of instances, not just thousands.

The last topic we covered was the overriding of a class attribute accessed via the instances, e.g. `self.typecode`. We did that first by creating an instance attribute, and then by subclassing and overwriting at the class level.

Throughout the chapter I mentioned how design choices in the examples were informed by studying the API of standard Python objects. If this chapter can be summarized in one sentence, this is it:

> To build Pythonic objects, observe how real Python objects behave.
>
> — Ancient Chinese proverb

Further reading

This chapter covered several special methods of the data model, so naturally the primary references are the same given in Chapter 1 which gave a high-level view of the same topic. For convenience, I'll repeat those four earlier recommendations here, and add a few ones.

Data Model page of the Python Language Reference
: Most of the methods we used in this chapter are documented in section 3.3.1. Basic customization of that page.

Python in a Nutshell, 2e, by Alex Martelli
: Excellent coverage of the Data Model, even if only Python 2.5 is covered (in the second edition), the fundamental concepts are all the same and most of the Data Model APIs haven't changed at all since Python 2.2, when built-in types and user-defined classes became more compatible.

Python Cookbook, 3rd Edition, by David Beazley and Brian K. Jones
: Very modern coding practices demonstrated through recipes. Chapter 8, *Classes and Objects* in particular has several solutions related to discussions in this chapter.

Python Essential Reference, 4th Edition, by David Beazley
: Covers the Data Model in detail in the context of Python 2.6 and Python 3.

In this chapter we covered every special method related to object representation, except `__index__`. It's used to coerce an object to an integer index in the specific context of sequence slicing, and was created to solve a need in NumPy. In practice, you and I are not likely to need to implement `__index__` unless we decide to write a new numeric data type, and we want it to be usable as arguments to `__getitem__`. If you are curious about it, A.M. Kuchling's What's New in Python 2.5 has a short explanation, and PEP 357 — Allowing Any Object to be Used for Slicing details the need for `__index__`, from the perspective of an implementor of a C-extension, Travis Oliphant, the lead author of NumPy.

An early realization of the need for distinct string representations for objects appeared in Smalltalk. The 1996 article How to display an object as a string: printString and displayString by Bobby Woolf discusses the implementation of the `printString` and `displayString` methods in that language. From that article I borrowed the pithy descriptions "the way the developer wants to see it" and "the way the user wants to see it" I used when defining `repr()` and `str()` in "Object representations" on page 248.

Soapbox

Properties help reduce upfront costs

In the initial versions of `Vector2d` the `x` and `y` attributes were public, as are all Python instance and class attributes by default. Naturally users of vectors need to be able to access its components. Although our vectors are iterable and can be unpacked into a pair of variables, it's also desirable to be able to write `my_vector.x` and `my_vector.y` to get each component.

When we felt the need to avoid accidental updates to the `x` and `y` attributes, we implemented properties but nothing changed elsewhere in the code and in the public interface of `Vector2d`, as verified by the doctests. We are still able to access `my_vector.x` and `my_vector.y`.

This shows that we can always start our classes in the simplest possible way, with public attributes, because when (or if) later we need to impose more control with getters and setters, these can be implemented through properties without changing any of the code that already interacts with our objects through the names (eg. `x` and `y`) that were initially simple public attributes.

This approach is the opposite of that encouraged by the Java language: a Java programmer cannot start with simple public attributes and only later, if needed, implement properties, because they don't exist in the language. Therefore, writing getters and setters is the norm in Java — even when those methods do nothing useful — because the API cannot evolve from simple public attributes to getters and setters without breaking all code that uses those attributes.

In addition, as our technical reviewer Alex Martelli points out, typing getter/setter calls everywhere is goofy. You have to write stuff like:

```
---
>>> my_object.set_foo(my_object.get_foo() + 1)
---
```

Just to do this:

```
---
>>> my_object.foo += 1
---
```

Ward Cunningham, inventor of the wiki and an Extreme Programming pioneer, recommends asking "What's the simplest thing that could possibly work?". The idea is to focus on the goal[11]. Implementing setters and getters up front is a distraction from the goal. In Python, we can simply use public attributes knowing we can change them to properties later, if the need arises.

Safety versus security in private attributes

> Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun.
>
> — Larry Wall
> *creator of Perl*

[11] See Simplest Thing that Could Possibly Work: A Conversation with Ward Cunningham, Part V.

Python and Perl are polar opposites in many regards, but Larry and Guido seem to agree on object privacy.

Having taught Python to many Java programmers over the years, I've found a lot of them put too much faith in the privacy guarantees that Java offers. As it turns out, the Java `private` and `protected` modifiers normally provide protection against accidents only — i.e. safety. They can only guarantee security against malicious intent if the application is deployed with a security manager, and that seldom happens in practice, even in corporate settings.

To prove my point, I like to show this Java class:

Example 9-15. `Confidential.java`: a Java class with a private field named `secret`.

```
public class Confidential {

    private String secret = "";

    public Confidential(String text) {
        secret = text.toUpperCase();
    }
}
```

In Example 9-15 I store the `text` in the `secret` field after converting it to uppercase, just to make it obvious that whatever is in that field will be in all caps.

The actual demonstration consists of running `expose.py` with Jython. That script uses introspection ("reflection" in Java parlance) to get the value of a private field. The code is in Example 9-16.

Example 9-16. `expose.py`: Jython code to read the content of a private field in another class.

```
import Confidential

message = Confidential('top secret text')
secret_field = Confidential.getDeclaredField('secret')
secret_field.setAccessible(True)  # break the lock!
print 'message.secret =', secret_field.get(message)
```

If you run Example 9-16, this is what you get:

```
$ jython expose.py
message.secret = TOP SECRET TEXT
```

The string `'TOP SECRET TEXT'` was read from the `secret` private field of the `Confidential` class.

There is no black magic here: `expose.py` uses the Java reflection API to get a reference to the private field named `'secret'`, and then calls `'secret_field.setAccessible(True)'` to make it readable. The same thing can be done with Java code, of course (but it takes more than three times as many lines to do it; see the file `Expose.java` in the Fluent Python code repository).

The crucial call `.setAccessible(True)` will fail only if the Jython script or the Java main program (e.g. `Expose.class`) is running under the supervision of a SecurityManager. But in the real world, Java applications are rarely deployed with a SecurityManager — except for Java Applets (remember those?).

My point is: in Java too, access control modifiers are mostly about safety and not security, at least in practice. So relax and enjoy the power Python gives you. Use it responsibly.

CHAPTER 10
Sequence hacking, hashing and slicing

> Don't check whether it *is*-a duck: check whether it *quacks*-like-a duck, *walks*-like-a duck, etc, etc, depending on exactly what subset of duck-like behavior you need to play your language-games with. (`comp.lang.python`, Jul. 26, 2000)
>
> — Alex Martelli

In this chapter we will create a class to represent a multi-dimensional `Vector` class — a significant step up from the 2-dimensional `Vector2d` of Chapter 9. `Vector` will behave like a standard Python immutable flat sequence. Its elements will be floats, and it will support the following by the end of this chapter:

- Basic sequence protocol: `__len__` and `__getitem__`.
- Safe representation of instances with many items.
- Proper slicing support, producing new `Vector` instances.
- Aggregate hashing taking into account every contained element value.
- Custom formatting language extension.

We'll also implement dynamic attribute access with `__getattr__` as a way of replacing the read-only properties we used in `Vector2d` — although this is not typical of sequence types.

The code-intensive presentation will be interrupted by a conceptual discussion about the idea of protocols as an informal interface. We'll talk about how protocols and *duck typing* are related, and its practical implications when you create your own types.

Let's get started.

Who needs a vector with 1000-dimensions?

Hint: not 3D artists! N-dimensional vectors (with large values of N) are widely used in information retrieval, where documents and text queries are represented as vectors, with one dimension per word. This is called the Vector space model. In this model, a key relevance metric is the cosine similarity, i.e. the cosine of the angle between a query vector and a document vector. As the angle decreases the cosine approaches the maximum value of 1, and so does the relevance of the document to the query.

Having said that, the `Vector` class in this chapter is a didactic example and we'll not do much math here. Our goal is just to demonstrate some Python special methods in the context of a sequence type.

NumPy and SciPy are the tools you need for real world vector math. The PyPI package gemsim, by Radim Rehurek, implements vector space modeling for natural language processing and information retrieval, using NumPy and SciPy.

`Vector`: a user-defined sequence type

Our strategy to implement `Vector` will be to use composition, not inheritance. We'll store the components in an `array` of floats, and will implement the methods needed for our `Vector` to behave like an immutable flat sequence.

But before we implement the sequence methods, let's make sure we have a baseline implementation of `Vector` that is compatible with our earlier `Vector2d` class — except where such compatibility would not make sense.

`Vector` take #1: `Vector2d` compatible

The first version of `Vector` should be as compatible as possible with our earlier `Vector2d` class.

However, by design, the `Vector` constructor is not compatible with the `Vector2d` constructor. We could make `Vector(3, 4)` and `Vector(3, 4, 5)` work, by taking arbitrary arguments with `*args` in `__init__`, but the best practice for a sequence constructor is to take the data as an iterable argument in the constructor, like all built-in sequence types do. Example 10-1 shows some ways of instantiating our new `Vector` objects.

Example 10-1. Tests of `Vector.__init__` and `Vector.__repr__`.

```
>>> Vector([3.1, 4.2])
Vector([3.1, 4.2])
>>> Vector((3, 4, 5))
Vector([3.0, 4.0, 5.0])
```

```
>>> Vector(range(10))
Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
```

Apart from new constructor signature, I made sure every test I did with `Vector2d` like `Vector2d(3, 4)` passed and produced the same result with a 2-component `Vector([3, 4])`.

> When a `Vector` has more than 6 components, the string produced by `repr()` is abbreviated with `...` as seen in the last line of Example 10-1. This is crucial in any collection type that may contain a large number of items, because `repr` is used for debugging — and you don't want a single large object to span thousands of lines in your console or log. Use the `reprlib` module to produce limited-length representations, as in Example 10-2. The `reprlib` module is called `repr` in Python 2. The `2to3` tool rewrites imports from `repr` automatically.

Example 10-2 lists the implementation of our first version of `Vector`.

Example 10-2. `vector_v1.py`: derived from `vector2d_v1.py` (See Example 9-2 and Example 9-3).

```
from array import array
import reprlib
import math


class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)    ❶

    def __iter__(self):
        return iter(self._components)   ❷

    def __repr__(self):
        components = reprlib.repr(self._components)   ❸
        components = components[components.find('['):-1]   ❹
        return 'Vector({})'.format(components)

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(self._components))   ❺

    def __eq__(self, other):
        return tuple(self) == tuple(other)

    def __abs__(self):
        return math.sqrt(sum(x * x for x in self))   ❻

    def __bool__(self):
        return bool(abs(self))

    @classmethod
    def frombytes(cls, octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(memv)   ❼
```

❶ The `self._components` instance "protected" attribute will hold an `array` with the `Vector` components.

❷ To allow iteration, we return an iterator over `self._components`[1].

❸ Use `reprlib.repr()` to get a limited-length representation of `self._components`, e.g. `array('d', [0.0, 1.0, 2.0, 3.0, 4.0, ...])`

❹ Remove the `array('d',` prefix and the trailing `)` before plugging the string into a `Vector` constructor call.

❺ Build a `bytes` object directly from `self._components`.

❻ We can't use `hypot` anymore, so we sum the squares of the components and compute the `sqrt` of that.

❼ The only change needed from the earlier `frombytes` is in the last line: we pass the `memoryview` directly to the constructor, without unpacking with `*` as we did before.

The way I used `reprlib.repr` deserves some elaboration. That function produces safe representations of large or recursive structures by limiting the length of the output string and marking the cut with `'...'`. I wanted the `repr` of a `Vector` to look like `Vector([3.0, 4.0, 5.0])` and not `Vector(array('d', [3.0, 4.0, 5.0]))`, because the fact that there is an `array` inside a `Vector` is an implementation detail. Since these constructor calls build identical `Vector` objects, I prefer the simpler syntax using a `list` argument.

When coding `__repr__` I could have produced the simplified `components` display with this expression: `reprlib.repr(list(self._components))`. However, this would be wasteful because I'd be copying every item from `self._components` to a `list` just to use the `list` `repr`. Instead, I decided to apply `reprlib.repr` to the `self._compo`

[1] The `iter()` function is covered in Chapter 14, along with the `__iter__` method.

nents array directly, and then chop off the characters outside of the []. That's what the second line of __repr__ does in Example 10-2.

> Because of its role in debugging, calling repr() on an object should never raise an exception. If something goes wrong inside your implementation of __repr__ you must deal with the issue and do your best to produce some serviceable output that gives the user a chance of identifying the target object.

Note that the __str__, __eq__ and __bool__ methods are unchanged from Vector2d, and only one character was changed in frombytes (a * was removed in the last line). This is one of the benefits of making the original Vector2d iterable.

By the way, we could have sub-classed Vector from Vector2d but I chose not to do it for two reasons. First, the incompatible constructors really make sub-classing not advisable. I could work around that with some clever parameter handling in __init__, but the second reason is more important: I want Vector to be a stand-alone example of a class implementing the sequence protocol. That's what we'll do next, after a discussion of the term "protocol".

Protocols and duck typing

As early as Chapter 1 we saw that you don't need to inherit from any special class to create a fully functional sequence type in Python, you just need to implement the methods that fulfill the sequence protocol. But what kind of protocol are we talking about?

In the context of Object Oriented Programming, a protocol is an informal interface, defined only in documentation and not in code. For example, the sequence protocol in Python entails just the __len__ and __getitem__ methods. Any class Spam that implements those methods with the standard signature and semantics can be used anywhere a sequence is expected. Whether Spam is a subclass of this or that is irrelevant, all that matters is that it provides the necessary methods.

Example 10-3. Code from Example 1-1, reproduced here for convenience.

```
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 `FrenchDeck` class in Example 10-3 takes advantage of many Python facilities because it implements the sequence protocol, even if that is not declared anywhere in the code. Any experienced Python coder will look at it and understand that it *is* a sequence, even if it subclasses `object`. We say it *is* a sequence because it *behaves* like one, and that is what matters.

This became known as *duck typing*, after Alex Martelli's post quoted at the top of this chapter.

Because protocols are informal and unenforced, you can often get away implementing just part of a protocol, if you know the specific context where a class will be used. For example, to support iteration, only `__getitem__` is required, there is no need to provide `__len__`.

We'll now implement the sequence protocol in `Vector`, initially without proper support for slicing, but later adding that.

`Vector` take #2: a sliceable sequence

As we saw with the `FrenchDeck` example, supporting the sequence protocol is really easy if you can delegate to a sequence attribute in your object, like our `self._components` array. These `__len__` and `__getitem__` one-liners are a good start:

```
class Vector:
    # many lines omitted
    # ...

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

    def __getitem__(self, index):
        return self._components[index]
```

With these additions, all of these operations now work:

```
>>> v1 = Vector([3, 4, 5])
>>> len(v1)
3
>>> v1[0], v1[-1]
(3.0, 5.0)
>>> v7 = Vector(range(7))
>>> v7[1:4]
array('d', [1.0, 2.0, 3.0])
```

As you can see, even slicing is supported — but not very well. It would be better if a slice of a `Vector` was also a `Vector` instance and not a `array`. The old `FrenchDeck` class has a similar problem: when you slice it, you get a `list`. In the case of `Vector`, a lot of functionality is lost when slicing produces plain arrays.

Consider the built-in sequence types: every one of them, when sliced, produces a new instance of its own type, and not of some other type.

To make `Vector` produce slices as `Vector` instance, we can't just delegate the slicing to `array`. We need to analyze the arguments we get in `__getitem__` and do the right thing.

Now, let's see how Python turns the syntax `my_seq[1:3]` into arguments for `my_seq.__getitem__(...)`.

How slicing works

A demo is worth a thousand words, so take a look at Example 10-4.

Example 10-4. Checking out the behavior of `__getitem__` and slices.

```
>>> class MySeq:
...     def __getitem__(self, index):
...         return index  # ❶
...
>>> s = MySeq()
>>> s[1]  # ❷
1
>>> s[1:4]  # ❸
slice(1, 4, None)
>>> s[1:4:2]  # ❹
slice(1, 4, 2)
>>> s[1:4:2, 9]  # ❺
(slice(1, 4, 2), 9)
>>> s[1:4:2, 7:9]  # ❻
(slice(1, 4, 2), slice(7, 9, None))
```

❶ For this demonstration, `__getitem__` merely returns whatever is passed to it.

❷ A single index, nothing new.

❸ The notation `1:4` becomes `slice(1, 4, None)`.

❹ `slice(1, 4, 2)` means start at 1, stop at 4, step by 2.

❺ Surprise: the presence of commas inside the `[]` means `__getitem__` receives a tuple.

❻ The tuple may even hold several slice objects.

Now let's take a closer look at `slice` itself:

Example 10-5. Checking out the behavior of `__getitem__` and slices.

```
>>> slice  # ❶
<class 'slice'>
>>> dir(slice) # ❷
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__',
 '__hash__', '__init__', '__le__', '__lt__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'indices', 'start', 'step', 'stop']
```

❶ `slice` is a built-in type (we saw it first in "Slice objects" on page 34).

❷ Inspecting a `slice` we find the data attributes `start`, `stop` and `step`, and an `indices` method.

In Example 10-5, calling `dir(slice)` reveals an `indices` attribute, which turns out to be a very interesting but little known method. Here is what `help(slice.indices)` reveals:

`S.indices(len) -> (start, stop, stride)`

> Assuming a sequence of length `len`, calculate the `start` and `stop` indices, and the `stride` length of the extended slice described by `S`. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

In other words, `indices` exposes the tricky logic that's implemented in the built-in sequences to gracefully handle missing or negative indices and slices that are longer than the target sequence. This method produces "normalized" tuples of non-negative `start`, `stop` and `stride` integers adjusted to fit within the bounds of a sequence of the given length.

Here are a couple of examples, considering a sequence of `len == 5`, e.g. `'ABCDE'`:

```
    >>> slice(None, 10, 2).indices(5)  # ❶
    (0, 5, 2)
    >>> slice(-3, None, None).indices(5)  # ❷
    (2, 5, 1)
```

❶ `'ABCDE'[:10:2]` is the same as `'ABCDE'[0:5:2]`

❷ `'ABCDE'[-3:]` is the same as `'ABCDE'[2:5:1]`

As I write this, the `slice.indices` method is apparently not documented in the online Python Library Reference. The Python Python/C API Reference Manual documents a similar C-level function, PySlice_GetIndicesEx. I discovered `slice.indices` while exploring slice objects in the Python console, using `dir()` and `help()`. Yet another evidence of the value of the interactive console as a discovery tool.

In our `Vector` code we'll not need the `slice.indices()` method because when we get a slice argument we'll delegate its handling to the `_components array`. But if you can't count on the services of an underlying sequence, this method can be a huge time saver.

Now that we know how to handle slices, let's see how the improved `Vector.__getitem__` implementation looks like.

A slice-aware `__getitem__`

Example 10-6 lists the two methods needed to make `Vector` behave as a sequence: `__len__` and `__getitem__` — the latter now implemented to handle slicing correctly.

Example 10-6. Part of vector_v2.py: `__len__` and `__getitem__` methods added to `Vector` class from vector_v1.py (See Example 10-2).

```
    def __len__(self):
        return len(self._components)

    def __getitem__(self, index):
        cls = type(self)  ❶
        if isinstance(index, slice):  ❷
            return cls(self._components[index])  ❸
        elif isinstance(index, numbers.Integral):  ❹
            return self._components[index]  ❺
        else:
            msg = '{cls.__name__} indices must be integers'
            raise TypeError(msg.format(cls=cls))  ❻
```

❶ Get the class of the instance (i.e. `Vector`) for later use.

❷ If the `index` argument is a `slice`…

❸ …invoke the class to build another `Vector` instance from a slice of the `_components` array.

❹ If the `index` is an `int` or some other kind of integer…

❺ …just return the specific item from `_components`.

❻ Otherwise, raise an exception.

Excessive use of isinstance may be a sign of bad OO design, but handling slices in __getitem__ is a justified use case. Note in Example 10-6 the test against numbers.Integral — an Abstract Base Class. Using ABCs in insinstance tests makes an API more flexible and future-proof. Chapter 11 explains why. Unfortunately, there is no ABC for slice in the Python 3.4 standard library.

To discover which exception to raise in the else clause of __getitem__, I used the interactive console to check the result of 'ABC'[1, 2]. I then learned that Python raises a TypeError, and I also copied the wording from the error message: "indices must be integers". To create Pythonic objects, mimic Python's own objects.

Once the code in Example 10-6 is added to the Vector class, we have proper slicing behavior, as Example 10-7 demonstrates.

Example 10-7. Tests of enhanced Vector.__getitem__ from Example 10-6.

```
>>> v7 = Vector(range(7))
>>> v7[-1]  ❶
6.0
>>> v7[1:4]  ❷
Vector([1.0, 2.0, 3.0])
>>> v7[-1:]  ❸
Vector([6.0])
>>> v7[1,2]  ❹
Traceback (most recent call last):
  ...
TypeError: Vector indices must be integers
```

❶ An integer index retrieves just one component value as a float.

❷ A slice index creates a new Vector.

❸ A slice of len == 1 also creates a Vector.

❹ Vector does not support multi-dimensional indexing, so a tuple of indices or slices raises an error.

Vector take #3: dynamic attribute access

In the evolution from Vector2d to Vector we lost the ability to access vector components by name, e.g. v.x, v.y. We are now dealing with vectors that may have large number of components. Still, it may be convenient to access the first few components with shortcut letters such as x, y, z instead of v[0], v[1] and v[2].

Here is the alternate syntax we want to provide for reading the first four components of a vector:

```
>>> v = Vector(range(10))
>>> v.x
0.0
>>> v.y, v.z, v.t
(1.0, 2.0, 3.0)
```

In Vector2d we provided read-only access to x and y using the @property decorator (Example 9-7). We could write four properties in Vector, but it would be tedious. The __getattr__ special method provides a better way.

The __getattr__ method is invoked by the interpreter when attribute lookup fails. In simple terms, given the expression my_obj.x, Python checks if the my_obj instance has an attribute named x; if not, the search goes to the class (my_obj.__class__), and then up the inheritance graph[2]. If the x attribute is not found, then the __getattr__ method defined in the class of my_obj is called with self and the name of the attribute as a string, e.g. 'x'.

Example 10-8 lists our __getattr__ method. Essentially it checks whether the attribute being sought is one of the letters xyzt and if so, returns the corresponding vector component.

Example 10-8. Part of vector_v3.py: __getattr__ method added to Vector class from vector_v2.py.

```
    shortcut_names = 'xyzt'

    def __getattr__(self, name):
        cls = type(self)  ❶
        if len(name) == 1:  ❷
            pos = cls.shortcut_names.find(name)  ❸
            if 0 <= pos < len(self._components):  ❹
                return self._components[pos]
        msg = '{.__name__!r} object has no attribute {!r}'  ❺
        raise AttributeError(msg.format(cls, name))
```

❶ Get the Vector class for later use.

❷ If the name is one character, it may be one of the shortcut_names.

❸ Find position of 1-letter name; str.find would also locate 'yz' and we don't want that, this is the reason for the test above.

❹ If the position is within range, return the array element.

❺ If either test failed, raise AttributeError with a standard message text.

[2] Attribute lookup is more complicated than this; we'll see the gory details in Part VI. For now, this simplified explanation will do.

It's not hard to implement `__getattr__`, but in this case it's not enough. Consider the bizarre interaction in Example 10-9.

Example 10-9. Inappropriate behavior: assigning to `v.x` raises no error, but introduces an inconsistency.

```
>>> v = Vector(range(5))
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0])
>>> v.x  # ❶
0.0
>>> v.x = 10  # ❷
>>> v.x  # ❸
10
>>> v
Vector([0.0, 1.0, 2.0, 3.0, 4.0])  # ❹
```

❶ Access element `v[0]` as `v.x`.

❷ Assign new value to `v.x`. This should raise an exception.

❸ Reading `v.x` shows the new value, `10`.

❹ However the vector components did not change.

Can you explain what is happening? In particular, why the second time `v.x` returns `10` if that value is not in the vector components array? If you don't know right off the bat, study the explanation of `__getattr__` given right before Example 10-8. It's a bit subtle, but a very important foundation to understand a lot of what comes later in the book.

The inconsistency in Example 10-9 was introduced because of the way `__getattr__` works: Python only calls that method as a fall back, when the object does not have the named attribute. However, after we assign `v.x = 10`, the `v` object now has an `x` attribute, so `__getattr__` will no longer be called to retrieve `v.x`: the interpreter will just return the value `10` that is bound to `v.x`. On the other hand, our implementation of `__getattr__` pays no attention to instance attributes other than `self._components`, from where it retrieves the values of the "virtual attributes" listed in `shortcut_names`.

We need to customize the logic for setting attributes in our `Vector` class in order do avoid this inconsistency.

Recall that in the latest `Vector2d` examples from Chapter 9, trying to assign to the `.x` or `.y` instance attributes raised `AttributeError`. In `Vector` we want the same exception with any attempt at assigning to all single-letter lower case attribute names, just to avoid confusion. To do that, we'll implement `__setattr__` as listed in Example 10-10.

Example 10-10. Part of `vector_v3.py`: `__setattr__` method in `Vector` class.

```
    def __setattr__(self, name, value):
        cls = type(self)
        if len(name) == 1:  ❶
            if name in cls.shortcut_names:  ❷
                error = 'readonly attribute {attr_name!r}'
            elif name.islower():  ❸
                error = "can't set attributes 'a' to 'z' in {cls_name!r}"
            else:
                error = ''  ❹
            if error:  ❺
                msg = error.format(cls_name=cls.__name__, attr_name=name)
                raise AttributeError(msg)
        super().__setattr__(name, value)  ❻
```

❶ Special handling for single-character attribute names.

❷ If name is one of xyzt, set specific error message.

❸ If name is lower case, set error message about all single-letter names.

❹ Otherwise, set blank error message.

❺ If there is a non-blank error message, raise AttributeError.

❻ Default case: call __setattr__ on superclass for standard behavior.

> The super() function provides a way to access methods of superclasses dynamically, a necessity in a dynamic language supporting multiple inheritance like Python. It's used to delegate some task from a method in a subclass to a suitable method in a superclass, as seen in Example 10-10. There is more about super in "Multiple inheritance and method resolution order" on page 353.

While choosing the error message to display with AttributeError, my first check was the behavior of the built-in complex type, since they are immutable and have a pair of data attributes real and imag. Trying to change either of those in a complex instance raises AttributeError with the message "can't set attribute". On the other hand, trying to set a read-only attribute protected by a property as we did in "A hashable Vector2d" on page 257 produces the message "readonly attribute". I drew inspiration from both wordings to set the error string in __setitem__, but was more explicit about the forbidden attributes.

Note that we are not disallowing setting all attributes, only single-letter, lowercase ones, to avoid confusion with the supported read-only attributes x, y, z and t.

Knowing that declaring `__slots__` at the class level prevents setting new instance attributes, it's tempting use that feature instead of implementing `__setattr__` as we did. However, because of all the caveats discussed in "The problems with `__slots__`" on page 267, using `__slots__` just to prevent instance attribute creation is not recommended. `__slots__` should be used only to save memory, and only if that is a real issue.

Even without supporting writing to the `Vector` components, here is an important take away from this example: very often when you implement `__getattr__` you need to code `__setattr__` as well, to avoid inconsistent behavior in your objects.

If we wanted to allow changing components we could implement `__setitem__` to enable `v[0] = 1.1` and/or `__setattr__` to make `v.x = 1.1` work. But `Vector` will remain immutable because we want to make it hashable in the coming section.

`Vector` take #4: hashing and a faster ==

Once more we get to implement a `__hash__` method. Together with the existing `__eq__`, this will make `Vector` instances hashable.

The `__hash__` in Example 9-8 simply computed `hash(self.x) ^ hash(self.y)`. We now would like to apply the `^` (xor) operator to the hashes of every component, in succession, like this: `v[0] ^ v[1] ^ v[2]`… That is what the `functools.reduce` function is for. Previously I said that `reduce` is not as popular as before[3], but computing the hash of all vector components is a perfect job for it.

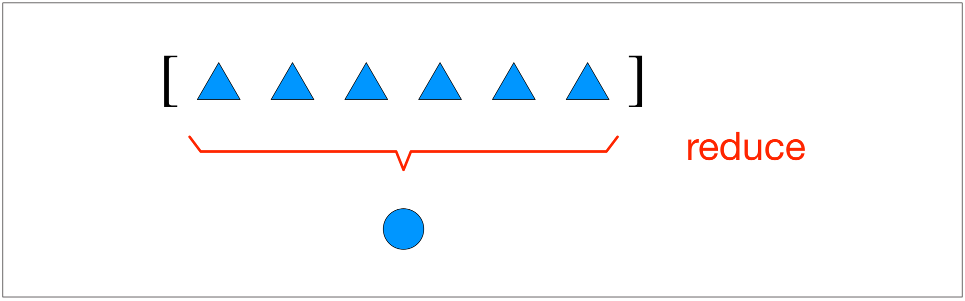


Figure 10-1. Reducing functions — `reduce`, `sum`, `any`, `all` — produce a single aggregate result from a sequence or from any finite iterable object.

[3] The `sum`, `any` and `all` cover the most common uses of `reduce`. See discussion in "Modern replacements for `map`, `filter` and `reduce`" on page 142.

So far we've seen that `functools.reduce()` can be replaced by `sum()`, but now let's properly explain how it works. The key idea is to reduce a series of values to a single value. The first argument to `reduce()` is a two-argument function, and the second argument is an iterable. Let's say we have a two-argument function `fn` and a list `lst`. When you call `reduce(fn, lst)`, `fn` will be applied to the first pair of elements — `fn(lst[0], lst[1])` — producing a first result, `r1`. Then `fn` is applied to `r1` and the next element — `fn(r1, lst[2])` — producing a second result `r2`. Now `fn(r2, lst[3])` is called to produce `r3` … and so on until the last element, when a single result `rN` is returned.

Here is how you could use `reduce` to compute 5! (the factorial of 5):

```
>>> 2 * 3 * 4 * 5  # the result we want: 5! == 120
120
>>> import functools
>>> functools.reduce(lambda a,b: a*b, range(1, 6))
120
```

Back to our hashing problem, Example 10-11 shows the idea computing the aggregate xor by doing it in three ways: with a `for` loop and two `reduce` calls.

Example 10-11. Three ways of calculating the accumulated xor of integers from 0 to 5.

```
>>> n = 0
>>> for i in range(1, 6):  # ❶
...     n ^= i
...
>>> n
1
>>> import functools
>>> functools.reduce(lambda a, b: a^b, range(6))  # ❷
1
>>> import operator
>>> functools.reduce(operator.xor, range(6))  # ❸
1
```

❶ Aggregate xor with a `for` loop and an accumulator variable.

❷ `functools.reduce` using an anonymous function.

❸ `functools.reduce` replacing custom `lambda` with `operator.xor`.

From the alternatives in Example 10-11, the last one is my favorite, and the `for` loop comes second. What is your preference?

As seen in "The `operator` module" on page 156, `operator` provides the functionality of all Python infix operators in function form, lessening the need for `lambda`.

To code `Vector.__hash__` in my preferred style, we need to import the `functools` and `operator` modules. Example 10-12 shows the relevant changes.

Example 10-12. Part of `vector_v4.py`: two imports and `__hash__` method added to `Vector` class from `vector_v3.py`.

```
from array import array
import reprlib
import math
import functools  # ❶
import operator  # ❷


class Vector:
    typecode = 'd'

    # many lines omitted in book listing...

    def __eq__(self, other):  # ❸
        return tuple(self) == tuple(other)

    def __hash__(self):
        hashes = (hash(x) for x in self._components)  # ❹
        return functools.reduce(operator.xor, hashes, 0)  # ❺

    # more lines omitted...
```

❶ Import `functools` to use `reduce`.

❷ Import `operator` to use `xor`.

❸ No change to `__eq__`; I listed it here because it's good practice to keep `__eq__` and `__hash__` close in source code, since they need to work together.

❹ Create a generator expression to lazily compute the hash of each component.

❺ Feed `hashes` to `reduce` with the `xor` function to compute the aggregate hash value; the third argument, `0`, is the initializer (see next warning).

When using `reduce` it's good practice to provide the third argument, `reduce(function, iterable, initializer)`, to prevent this exception: `TypeError: reduce() of empty sequence with no initial value` (excellent message: explains the problem and how to fix it). The `initializer` is the value returned if the sequence is empty and is used as the first argument in the reducing loop, so it should be the identity value of the operation. As examples, for `+`, `|`, `^` the `initializer` should be `0`, but for `*`, `&` it should be `1`.

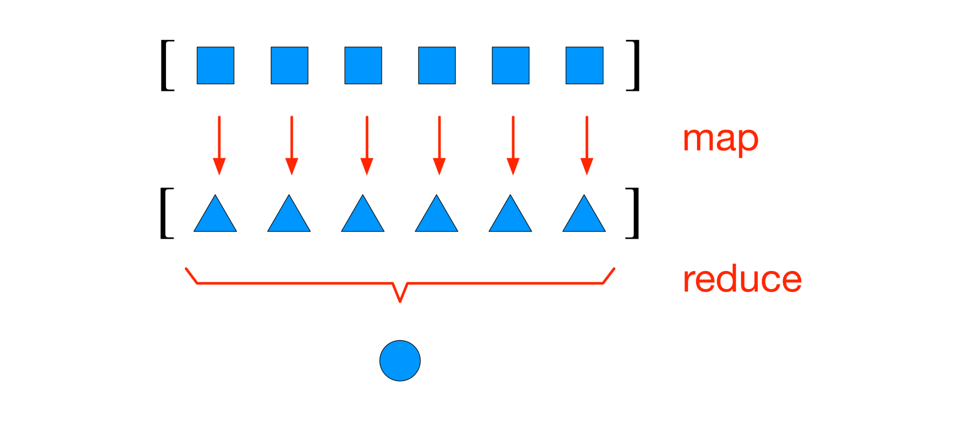


Figure 10-2. Map-reduce: apply function to each item to generate a new series (map), then compute aggregate (reduce)

As implemented, the `__hash__` method in Example 10-8 is a perfect example of a map-reduce computation (Figure 10-2). The mapping step produces one hash for each component, and the reduce step aggregates all hashes with the xor operator. Using `map` instead of a *genexp* makes the mapping step even more visible:

```
def __hash__(self):
    hashes = map(hash, self._components)
    return functools.reduce(operator.xor, hashes)
```

The solution with `map` would be less efficient in Python 2, where the `map` function builds a new `list` with the results. But in Python 3, `map` is lazy: it creates a generator that yields the results on demand, thus saving memory — just like the generator expression we used in the `__hash__` method of Example 10-8.

While we are on the topic of reducing functions, we can replace our quick implementation of `__eq__` with another one that will be cheaper in terms of processing and memory, at least for large vectors. Since Example 9-2 we have this very concise implementation of `__eq__`:

```
def __eq__(self, other):
    return tuple(self) == tuple(other)
```

This works for `Vector2d` and for `Vector` — it even considers `Vector([1, 2])` equal to `(1, 2)` which may be a problem, but we'll overlook that for now[4]. But for `Vector` instances that may have thousands of components, it's very inefficient. It builds two tuples copying the entire contents of the operands just to use the `__eq__` of the `tuple` type. For `Vector2d` — with only two components — it's a good shortcut, but not for the large multi-dimensional vectors. A better way of comparing one `Vector` to another `Vector` or iterable would be this:

Example 10-13. `Vector.__eq__` using `zip` in a `for` loop for more efficient comparison.

```
def __eq__(self, other):
    if len(self) != len(other):  # ❶
        return False
    for a, b in zip(self, other):  # ❷
        if a != b:  # ❸
            return False
    return True  # ❹
```

❶ If the `len` of the objects are different, they are not equal.

❷ `zip` produces a generator of tuples made from the items in each iterable argument. See "The awesome `zip`" on page 295 if `zip` is new to you. The `len` comparison above is needed because `zip` stops producing values without warning as soon as one of the inputs is exhausted.

❸ As soon as two components are different, exit returning `False`.

❹ Otherwise, the objects are equal.

Example 10-13 is efficient, but the `all` function can produce the same aggregate computation of the `for` loop in one line: if all comparisons between corresponding components in the operands are `True`, the result is `True`. As soon as one comparison is `False`, `all` returns `False`. This is how `__eq__` looks like using `all`:

Example 10-14. `Vector.__eq__` using `zip` and `all`: same logic as Example 10-13.

```
def __eq__(self, other):
    return len(self) == len(other) and all(a == b for a, b in zip(self, other))
```

Note that we first check that the operands have equal length, because `zip` will stop at the shortest operand.

Example 10-14 is the implementation we choose for `__eq__` in `vector_v4.py`.

[4] We'll seriously consider the matter of `Vector([1, 2]) == (1, 2)` in "Operator overloading 101" on page 374.

We wrap up this chapter by bringing back the `__format__` method from `Vector2d` to `Vector`.

The awesome `zip`

Having a `for` loop that iterates over items without fiddling with index variables is great and prevents lots of bugs, but demands some special utility functions. One of them is the `zip` built-in, which makes it easy to iterate in parallel over two or more iterables by returning tuples what you can unpack into variables, one for each item in the parallel inputs. See Example 10-15.

The `zip` function is named after the zipper fastener because the physical device works by interlocking pairs of teeth taken from both zipper sides, a good visual analogy for what `zip(left, right)` does. No relation with compressed files.

Example 10-15. The `zip` built-in at work.

```
>>> zip(range(3), 'ABC')  # ❶
<zip object at 0x10063ae48>
>>> list(zip(range(3), 'ABC'))  # ❷
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(zip(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3]))  # ❸
[(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2)]
>>> from itertools import zip_longest  # ❹
>>> list(zip_longest(range(3), 'ABC', [0.0, 1.1, 2.2, 3.3], fillvalue=-1))
[(0, 'A', 0.0), (1, 'B', 1.1), (2, 'C', 2.2), (-1, -1, 3.3)]
```

❶ `zip` returns generator that produces tuples on demand.

❷ Here we build a `list` from it just for display; usually we iterate over the generator.

❸ `zip` has a surprising trait: it stops without warning when one of the iterables is exhausted[5].

❹ The `itertools.zip_longest` function behaves differently: it uses an optional `fillvalue` (`None` by default) to complete missing values so it can generate tuples until the last iterable is exhausted.

The `enumerate` built-in is another generator function often used in `for` loops to avoid manual handling of index variables. If you are not familiar with `enumerate` you should definitely check it out in the Built-in function documentation. The `zip` and `enumer`

[5] That's surprising to me at least. I think `zip` should raise `ValueError` if the sequences are not all of the same length, which is what happens when unpacking an iterable to a tuple of variables of different length.

`ate` built-ins, along with several other generator functions in the standard library are covered in "Generator functions in the standard library" on page 426.

`Vector` take #5: formatting

The `__format__` method of `Vector` will resemble that of `Vector2d`, but instead of providing a custom display in polar coordinates, `Vector` will use spherical coordinates — also known as "hyperspherical" coordinates, because now we support N-dimensions, and spheres are "hyperspheres" in in 4D and beyond[6]. Accordingly, we'll change the custom format suffix from `'p'` to `'h'`.

As we saw in "Formatted displays" on page 253, when extending the Format Specification Mini-Language it's best to avoid reusing format codes supported by built-in types. In particular, our extended mini-language also uses the float formatting codes `'eEfFgGn%'` in their original meaning, so we definitely must avoid these. Integers use `'bcdoxXn'` and strings use `'s'`. I picked `'p'` for `Vector2d` polar coordinates. Code `'h'` for hyperspherical coordinates is a good choice.

For example, given a `Vector` object in 4D space (`len(v) == 4`), the `'h'` code will produce a display like <r, Φ_1, Φ_2, Φ_3> where `r` is the magnitude (`abs(v)`) and the remaining numbers are the angular coordinates Φ_1, Φ_2, Φ_3.

Here are some samples of the spherical coordinate format in 4D, taken from the doctests of `vector_v5.py` (see Example 10-16):

```
>>> format(Vector([-1, -1, -1, -1]), 'h')
'<2.0, 2.0943951023931957, 2.186276035465284, 3.9269908169872414>'
>>> format(Vector([2, 2, 2, 2]), '.3eh')
'<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
>>> format(Vector([0, 1, 0, 0]), '0.5fh')
'<1.00000, 1.57080, 0.00000, 0.00000>'
```

Before we can implement the minor changes required in `__format__`, we need to code a pair of support methods: `angle(n)` to compute one of the angular coordinates (eg. Φ_1), and `angles()` to return an iterable of all angular coordinates. I'll not describe the math here; if you're curious, the Wikipedia article on n-sphere has the formulas I used to calculate the spherical coordinates from the Cartesian coordinates in the `Vector` components array.

[6] The Wolfram Mathworld site has an article on Hypersphere; on the English Wikipedia, "hypersphere" redirects to n-sphere.

Example 10-16 is a full listing of `vector_v5.py` consolidating all we've implemented since "`Vector` take #1: `Vector2d` compatible" on page 278 and introducing custom formatting.

Example 10-16. `vector_v5.py`: doctests and all code for final `Vector` class. Callouts highlight additions needed to support `__format__`.

```
"""
A multi-dimensional ``Vector`` class, take 5

A ``Vector`` is built from an iterable of numbers::

    >>> Vector([3.1, 4.2])
    Vector([3.1, 4.2])
    >>> Vector((3, 4, 5))
    Vector([3.0, 4.0, 5.0])
    >>> Vector(range(10))
    Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])


Tests with 2-dimensions (same results as ``vector2d_v1.py``)::

    >>> v1 = Vector([3, 4])
    >>> x, y = v1
    >>> x, y
    (3.0, 4.0)
    >>> v1
    Vector([3.0, 4.0])
    >>> v1_clone = eval(repr(v1))
    >>> v1 == v1_clone
    True
    >>> print(v1)
    (3.0, 4.0)
    >>> octets = bytes(v1)
    >>> octets
    b'd\\x00\\x00\\x00\\x00\\x00\\x00\\x08@\\x00\\x00\\x00\\x00\\x00\\x00\\x10@'
    >>> abs(v1)
    5.0
    >>> bool(v1), bool(Vector([0, 0]))
    (True, False)


Test of ``.frombytes()`` class method:

    >>> v1_clone = Vector.frombytes(bytes(v1))
    >>> v1_clone
    Vector([3.0, 4.0])
    >>> v1 == v1_clone
    True


Tests with 3-dimensions::
    >>> v1 = Vector([3, 4, 5])
    >>> x, y, z = v1
    >>> x, y, z
    (3.0, 4.0, 5.0)
    >>> v1
    Vector([3.0, 4.0, 5.0])
    >>> v1_clone = eval(repr(v1))
    >>> v1 == v1_clone
    True
    >>> print(v1)
    (3.0, 4.0, 5.0)
    >>> abs(v1)  # doctest:+ELLIPSIS
    7.071067811...
    >>> bool(v1), bool(Vector([0, 0, 0]))
    (True, False)


Tests with many dimensions::

    >>> v7 = Vector(range(7))
    >>> v7
    Vector([0.0, 1.0, 2.0, 3.0, 4.0, ...])
    >>> abs(v7)  # doctest:+ELLIPSIS
    9.53939201...


Test of ``.__bytes__`` and ``.frombytes()`` methods::

    >>> v1 = Vector([3, 4, 5])
    >>> v1_clone = Vector.frombytes(bytes(v1))
    >>> v1_clone
    Vector([3.0, 4.0, 5.0])
    >>> v1 == v1_clone
    True


Tests of sequence behavior::

    >>> v1 = Vector([3, 4, 5])
    >>> len(v1)
    3
    >>> v1[0], v1[len(v1)-1], v1[-1]
    (3.0, 5.0, 5.0)


Test of slicing::

    >>> v7 = Vector(range(7))
    >>> v7[-1]
    6.0
    >>> v7[1:4]
    Vector([1.0, 2.0, 3.0])
    >>> v7[-1:]
    Vector([6.0])
    >>> v7[1,2]
    Traceback (most recent call last):
      ...
    TypeError: Vector indices must be integers


Tests of dynamic attribute access::

    >>> v7 = Vector(range(10))
    >>> v7.x
    0.0
    >>> v7.y, v7.z, v7.t
    (1.0, 2.0, 3.0)

Dynamic attribute lookup failures::

    >>> v7.k
    Traceback (most recent call last):
      ...
    AttributeError: 'Vector' object has no attribute 'k'
    >>> v3 = Vector(range(3))
    >>> v3.t
    Traceback (most recent call last):
      ...
    AttributeError: 'Vector' object has no attribute 't'
    >>> v3.spam
    Traceback (most recent call last):
      ...
    AttributeError: 'Vector' object has no attribute 'spam'


Tests of hashing::

    >>> v1 = Vector([3, 4])
    >>> v2 = Vector([3.1, 4.2])
    >>> v3 = Vector([3, 4, 5])
    >>> v6 = Vector(range(6))
    >>> hash(v1), hash(v3), hash(v6)
    (7, 2, 1)


Most hash values of non-integers vary from a 32-bit to 64-bit CPython build::

    >>> import sys
    >>> hash(v2) == (384307168202284039 if sys.maxsize > 2**32 else 357915986)
    True


Tests of ``format()`` with Cartesian coordinates in 2D::
    >>> v1 = Vector([3, 4])
    >>> format(v1)
    '(3.0, 4.0)'
    >>> format(v1, '.2f')
    '(3.00, 4.00)'
    >>> format(v1, '.3e')
    '(3.000e+00, 4.000e+00)'


Tests of ``format()`` with Cartesian coordinates in 3D and 7D::

    >>> v3 = Vector([3, 4, 5])
    >>> format(v3)
    '(3.0, 4.0, 5.0)'
    >>> format(Vector(range(7)))
    '(0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0)'


Tests of ``format()`` with spherical coordinates in 2D, 3D and 4D::

    >>> format(Vector([1, 1]), 'h')  # doctest:+ELLIPSIS
    '<1.414213..., 0.785398...>'
    >>> format(Vector([1, 1]), '.3eh')
    '<1.414e+00, 7.854e-01>'
    >>> format(Vector([1, 1]), '0.5fh')
    '<1.41421, 0.78540>'
    >>> format(Vector([1, 1, 1]), 'h')  # doctest:+ELLIPSIS
    '<1.73205..., 0.95531..., 0.78539...>'
    >>> format(Vector([2, 2, 2]), '.3eh')
    '<3.464e+00, 9.553e-01, 7.854e-01>'
    >>> format(Vector([0, 0, 0]), '0.5fh')
    '<0.00000, 0.00000, 0.00000>'
    >>> format(Vector([-1, -1, -1, -1]), 'h')  # doctest:+ELLIPSIS
    '<2.0, 2.09439..., 2.18627..., 3.92699...>'
    >>> format(Vector([2, 2, 2, 2]), '.3eh')
    '<4.000e+00, 1.047e+00, 9.553e-01, 7.854e-01>'
    >>> format(Vector([0, 1, 0, 0]), '0.5fh')
    '<1.00000, 1.57080, 0.00000, 0.00000>'
"""

from array import array
import reprlib
import math
import numbers
import functools
import operator
import itertools  ❶


class Vector:
    typecode = 'd'
    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __iter__(self):
        return iter(self._components)

    def __repr__(self):
        components = reprlib.repr(self._components)
        components = components[components.find('['):-1]
        return 'Vector({})'.format(components)

    def __str__(self):
        return str(tuple(self))

    def __bytes__(self):
        return (bytes([ord(self.typecode)]) +
                bytes(self._components))

    def __eq__(self, other):
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))

    def __hash__(self):
        hashes = (hash(x) for x in self)
        return functools.reduce(operator.xor, hashes, 0)

    def __abs__(self):
        return math.sqrt(sum(x * x for x in self))

    def __bool__(self):
        return bool(abs(self))

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

    def __getitem__(self, index):
        cls = type(self)
        if isinstance(index, slice):
            return cls(self._components[index])
        elif isinstance(index, numbers.Integral):
            return self._components[index]
        else:
            msg = '{.__name__} indices must be integers'
            raise TypeError(msg.format(cls))

    shortcut_names = 'xyzt'

    def __getattr__(self, name):
        cls = type(self)
        if len(name) == 1:
            pos = cls.shortcut_names.find(name)
            if 0 <= pos < len(self._components):
                return self._components[pos]
        msg = '{.__name__!r} object has no attribute {!r}'
        raise AttributeError(msg.format(cls, name))

    def angle(self, n):  ❷
        r = math.sqrt(sum(x * x for x in self[n:]))
        a = math.atan2(r, self[n-1])
        if (n == len(self) - 1) and (self[-1] < 0):
            return math.pi * 2 - a
        else:
            return a

    def angles(self):  ❸
        return (self.angle(n) for n in range(1, len(self)))

    def __format__(self, fmt_spec=''):
        if fmt_spec.endswith('h'):  # hyperspherical coordinates
            fmt_spec = fmt_spec[:-1]
            coords = itertools.chain([abs(self)],
                                     self.angles())  ❹
            outer_fmt = '<{}>'  ❺
        else:
            coords = self
            outer_fmt = '({})'  ❻
        components = (format(c, fmt_spec) for c in coords)  ❼
        return outer_fmt.format(', '.join(components))  ❽

    @classmethod
    def frombytes(cls, octets):
        typecode = chr(octets[0])
        memv = memoryview(octets[1:]).cast(typecode)
        return cls(memv)
```

❶ Import `itertools` to use `chain` function in `__format__`.

❷ Compute one of the angular coordinates, using formulas adapted from the n-sphere article.

❸ Create generator expression to compute all angular coordinates on demand.

❹ Use `itertools.chain` to produce genexp to iterate seamlessly over the magnitude and the angular coordinates.

❺ Configure spherical coordinate display with angular brackets.

❻ Configure Cartesian coordinate display with parenthesis.

❼ Create generator expression to format each coordinate item on demand.

❽ Plug formatted components separated by commas inside brackets or parenthesis.

We are making heavy use of generator expressions in `__format__`, `angle` and `angles` but our focus here is in providing `__format__` to bring `Vector` to the same implementation level as `Vector2d`. When we cover generators in Chapter 14 we'll use some of the code in `Vector` as examples, and then the generator tricks will be explained in detail.

This concludes our mission for this chapter. The `Vector` class will be enhanced with infix operators in Chapter 13, but our goal here was to explore techniques for coding special methods that are useful in a wide variety of collection classes.

Chapter summary

The `Vector` example in this chapter was designed to be compatible with `Vector2d`, except for the use of a different constructor signature accepting a single iterable argument, just like the built-in sequence types do. The fact that `Vector` behaves as a sequence just by implementing `__getitem__` and `__len__` prompted a discussion of protocols, the informal interfaces used in duck-typed languages.

We then looked at how the `my_seq[a:b:c]` syntax works behind the scenes, by creating a `slice(a, b, c)` object and handing it to `__getitem__`. Armed with this knowledge, we made `Vector` respond correctly to slicing, by returning new `Vector` instances, just like a Pythonic sequence is expected to do.

The next step was to provide read-only access to the first few `Vector` components using notation such as `my_vec.x`. We did it by implementing `__getattr__`. Doing that opened the possibility of tempting the user to assign to those special components by writing `my_vec.x = 7`, revealing a potential bug. We fixed it by implementing `__setattr__` as well, to forbid assigning values to single-letter attributes. Very often, when you code a `__getattr__` you need to add `__setattr__` too, in order to avoid inconsistent behavior.

Implementing the `__hash__` function provided the perfect context for using `functools.reduce`, as we needed to apply the xor operator `^` in succession to the hashes of all `Vector` components to produce an aggregate hash value for the whole `Vector`. After applying `reduce` in `__hash__`, we used the `all` reducing built-in to create a more efficient `__eq__` method.

The last enhancement to `Vector` was to re-implement the `__format__` method from `Vector2d` by supporting spherical coordinates as an alternative to the default Cartesian coordinates format. We used quite a bit of math and several generators to code `__format__` and its auxiliary functions, but these are implementation details — and we'll come back to the generators in Chapter 14. The goal of that last section was to support a custom format, thus fulfilling the promise of `Vector` that could do everything a `Vector2d` did, and more.

As we did in Chapter 9, here we often looked at how standard Python objects behave, to emulate them and provide a "Pythonic" look-and-feel to `Vector`.

In Chapter 13 we will implement several infix operators on `Vector`. The math will much simpler than that in the `angle()` method here, but exploring how infix operators work in Python is a great lesson in OO design. But before we get to operator overloading, we'll step back from working on one class and look at organizing multiple classes with interfaces and inheritance, the subjects of Chapter 11 and Chapter 12.

Further reading

Most special methods covered in the `Vector` example also appear in the `Vector2d` example from Chapter 9, so the references in "Further reading" on page 271 are all relevant here.

The powerful `reduce` higher-order function is also known as fold, accumulate, aggregate, compress and inject. The Wikipedia article about it is titled Fold, and presents applications of that higher-order function with emphasis on functional programming with recursive data structures. The article also includes a table listing fold-like functions in dozens of programming languages.

Soapbox

Protocols as informal interfaces

Protocols are not an invention of Python. The Smalltalk team, who also coined the expression "object oriented", used "protocol" as synonym for what we now call interfaces. Some Smalltalk programming environments allowed programmers to tag a group of methods as a protocol, but that was merely a documentation and navigation aid, and not enforced by the language. That's why I believe "informal interface" is a reasonable short explanation for "protocol" when I speak to an audience that is more familiar with formal (and compiler enforced) interfaces.

Established protocols naturally evolve in any language that uses dynamic typing, that is, when type-checking done at run time because there is no static type information in method signatures and variables. Ruby is another important OO language that has dynamic typing and uses protocols.

In the Python documentation, you can often tell when a protocol is being discussed when you see language like "a file-like object". This is a quick way of saying "something that behaves sufficiently like a file, by implementing the parts of the file interface that are relevant in the context".

You may think that implementing only part of a protocol is sloppy but it has the advantage of keeping things simple. Section 3.3 of the Data Model chapter suggests:

> When implementing a class that emulates any built-in type, it is important that the emulation only be implemented to the degree that it makes sense for the object being modeled. For example, some sequences may work well with retrieval of individual elements, but extracting a slice may not make sense.
>
> — Python Language Reference: Data Model

When we don't need to code non-sense methods just to fulfill some over-designed interface contract and keep the compiler happy, it becomes easier to follow the KISS principle.

I'll have more to say about protocols and interfaces in Chapter 11, where that is actually the main focus.

Origins of duck-typing

I believe the Ruby community, more than any other, helped popularize the term "duck typing", as they preached to the Java masses. But the expression has been used in Python discussions before either Ruby or Python were "popular". According to the English-language Wikipedia, an early example of the duck analogy in object-oriented programming, is a July, 26th, 2000 message to the Python-list by Alex Martelli: polymorphism (was Re: Type checking in python?). That's where the quote at the top of this chapter came from. If you are curious about the literary origins of the "duck typing" term, and the applications of this OO concept in many languages, check out the Wikipedia article Duck typing.

A safe `__format__`, with enhanced usability

While implementing `__format__` we did not take any precautions regarding `Vector` instances with a very large number of components, as we did in `__repr__` using `reprlib`. The reasoning is that `repr()` is for debugging and logging, so it must always generate some serviceable output, while `__format__` is used to display output to end-users who presumably want to see the entire `Vector`. If you think this is dangerous, then it would be cool to implement a further extension to the format specifier mini-language.

Here is how I'd do it: by default, any formatted `Vector` would display a reasonable but limited number of components, say 30. If there are more elements than that, the default behavior would be similar to what the `reprlib` does: chop the excess and put `...` in its place. However, if the format specifier ended with the special `*` code, meaning "all", then the size limitation would be disabled. So a user that's unaware of the problem of very long displays will not be bitten by it by accident. But if the default limitation becomes a nuisance, then the presence of the `...` should prompt the user to research the documentation and discover the `*` formatting code.

Send a pull request to the Fluent Python repository on Github if you implement this!

The search for a Pythonic sum

There's no single answer to "What is Pythonic?", just as there's no single answer to "What is beautiful?". Saying, as I often do, that it means using "idiomatic Python" is not 100%

satisfactory, because what may be “idiomatic” for you may not be for me. One thing I know: “idiomatic” does not mean using the most obscure language features.

In the Python-list there’s a thread from April, 2003 titled “Pythonic way to sum n-th list element?”. It’s relevant to our discussion of reduce in this chapter.

The original poster, Guy Middleton, asked for an improvement on this solution, stating he did not like to use `lambda` [7]:

```
>>> my_list = [[1, 2, 3], [40, 50, 60], [9, 8, 7]]
>>> import functools
>>> functools.reduce(lambda a, b: a+b, [sub[1] for item in my_list])
60
```

That code uses lots of idioms: `lambda`, `reduce` and a list comprehension. It would probably come last in a popularity contest, as it offends people who hate `lambda` and those who despise list comprehensions — pretty much both sides of a divide.

If you’re going to use `lambda`, there’s probably no reason to use a list comprehension — except for filtering, which is not the case here.

Here is a solution of my own that will please the lambda-lovers:

```
>>> functools.reduce(lambda a, b: a + b[1], my_list, 0)
60
```

I did not take part in the original thread, and I wouldn’t use that in real code, as I don’t like `lambda` too much myself, but I wanted to show an example without a list comprehension.

The first answer came from Fernando Perez, creator of IPython, highlighting that NumPy supports N-dimensional arrays and N-dimensional slicing:

```
>>> import numpy as np
>>> my_array = np.array(my_list)
>>> np.sum(my_array[:, 1])
60
```

I think Perez’s solution is cool, but Guy Middleton praised this next solution, by Paul Rubin and Skip Montanaro:

```
>>> import operator
>>> functools.reduce(operator.add, [sub[1] for sub in my_list], 0)
60
```

Then Evan Simpson asked, “What’s wrong with this?”:

```
>>> t = 0
>>> for sub in my_list:
...     total += sub[1]
```

[7] I adapted the code for this presentation: in 2003 `reduce` was a built-in, but in Python 3 we need to import it; also I replaced the names `x` and `y` with `my_list` and `sub`, for sub-list.

```
>>> t
60
```

Lots of people agreed that was quite Pythonic, Alex Martelli went as far as saying that's probably how Guido would code it.

I like Evan Simpsons's code but I also like David Eppstein's comment on it:

> If you want the sum of a list of items, you should write it in a way that looks like "the sum of a list of items", not in a way that looks like "loop over these items, maintain another variable t, perform a sequence of additions". Why do we have high level languages if not to express our intentions at a higher level and let the language worry about what low-level operations are needed to implement it?

Then Alex Martelli comes back to suggest:

> "The sum" is so frequently needed that I wouldn't mind at all if Python singled it out as a built-in. But "reduce(operator.add, ..." just isn't a great way to express it, in my opinion (and yet as an old APL'er, and FP-liker, I *should* like it — but I don't).

Alex goes on to suggest a `sum()` function, which he contributed. It became a built-in in Python 2.3, released only three months after that conversation took place. So Alex's preferred syntax became the norm:

```
>>> sum([sub[1] for sub in my_list])
60
```

By the end of the next year (November, 2004), Python 2.4 was launched with generator expressions, providing what is now in my opinion the most Pythonic answer to Guy Middleton's original question:

```
>>> sum(sub[1] for sub in my_list)
60
```

This is not only more readable than `reduce` but also avoids the trap of the empty sequence: `sum([])` is `0`, simple as that.

In the same conversation, Alex Martelli suggests the `reduce` built-in in Python 2 was more trouble than it was worth, as it encouraged coding idioms that were hard to explain. He was most convincing: the function was demoted to the `functools` module in Python 3.

Still, `functools.reduce` has its place. It solved the problem of our `Vector.__hash__` in a way that I would call Pythonic.

CHAPTER 11
Interfaces: from protocols to ABCs

> An abstract class represents an interface[1].
>
> — Bjarne Stroustrup
> *Creator of C++*

Interfaces are the subject of this chapter: from the dynamic protocols that are the hallmark of *duck typing* to ABCs (Abstract Base Classes) that make interfaces explicit and verify implementations for conformance.

If you have a Java, C# or similar background, the novelty here is in the informal protocols of duck typing. But for the long-time Pythonista or Rubyist, that is the "normal" way of thinking about interfaces, and the news is the formality and type-checking of ABCs. The language was 15 years old when ABCs were introduced in Python 2.6.

We'll start the chapter reviewing how the Python community traditionally understood interfaces as somewhat loose — in the sense that a partially implemented interface is often acceptable. We'll make that clear through a couple examples that highlight the dynamic nature of duck typing.

Then, a guest essay by Alex Martelli will introduce ABCs and give name to a new trend in Python programming. The rest of the chapter will be devoted to ABCs, starting with their common use as superclasses when you need to implement an interface. We'll then see when an ABC checks concrete subclasses for conformance to the interface it defines, and how a registration mechanism lets developers declare that a class implements an interface without subclassing. Finally, we'll see how an ABC can be programmed to automatically "recognize" arbitrary classes that conform to its interface — without subclassing or explicit registration.

[1] Bjarne Stroustrup, *The Design and Evolution of C++* (Addison-Wesley, 1994), p. 278.

We will implement a new ABC to see how that works, but Alex Martelli and I don't want to encourage you to start writing your own ABCs left and right. The risk of over-engineering with ABCs is very high.

> ABCs, like descriptors and metaclasses, are tools for building frameworks. Therefore, only a very small minority of Python developers can create ABCs without imposing unreasonable limitations and needless work on fellow programmers.

Let's get started with the Pythonic view of interfaces.

Interfaces and protocols in Python culture

Python was already highly successful before ABCs were introduced, and most existing code does not use them at all. Since Chapter 1 we've been talking about *duck typing* and protocols. In "Protocols and duck typing" on page 281 protocols are defined as the informal interfaces that make polymorphism work in languages with dynamic typing like Python.

How do interfaces work in a dynamic-typed language? First, the basics: even without an `interface` keyword in the language, and regardless of ABCs, every class has an interface: the set public attributes (methods or data attributes) implemented or inherited by the class. This includes special methods, like `__getitem__` or `__add__`.

By definition, protected and private attributes are not part of an interface, even if "protected" is merely a naming convention (the single leading underscore) and private attributes are easily accessed (recall "Private and "protected" attributes in Python" on page 263). It is bad form to violate these conventions.

On the other hand, it's not a sin to have public data attributes as part of the interface of an object, because — if necessary — a data attribute can always be turned into a property implementing getter/setter logic without breaking client code that uses the plain `obj.attr` syntax. We did that in the `Vector2d` class: in Example 11-1 we see the first implementation with public `x` and `y` attributes.

Example 11-1. `vector2d_v0.py`: `x` and `y` are public data attributes (same code as Example 9-2).

```
class Vector2d:
    typecode = 'd'

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

    def __iter__(self):
        return (i for i in (self.x, self.y))

    # more methods follow (omitted in this listing)
```

Later in Example 9-7 we turned x and y into read-only properties (Example 11-2). This is a significant refactoring, but an essential part of the interface of Vector2d is unchanged: users can still read my_vector.x and my_vector.y.

Example 11-2. vector2d_v3.py: x and y reimplemented as properties (see full listing in Example 9-9).

```
class Vector2d:
    typecode = 'd'

    def __init__(self, x, y):
        self.__x = float(x)
        self.__y = float(y)

    @property
    def x(self):
        return self.__x

    @property
    def y(self):
        return self.__y

    def __iter__(self):
        return (i for i in (self.x, self.y))

    # more methods follow (omitted in this listing)
```

A useful complementary definition of interface is: the subset of an object's public methods that enable it to play a specific role in the system. That's what is implied when the Python documentation mentions "a file-like object" or "an iterable", without specifying a class. An interface seen as a set of methods to fulfill a role is what Smalltalkers called a *procotol*, and the term spread to other dynamic language communities. Protocols are independent of inheritance. A class may implement several protocols, enabling its instances to fulfill several roles.

Protocols are interfaces, but because they are informal — defined only by documentation and conventions — protocols cannot be enforced like formal interfaces can (we'll see how ABCs enforce interface conformance later in this chapter). A protocol may be partially implemented in a particular class, and that's OK. Sometimes all a specific API requires from "a file-like object" is that it has a .read() method that returns bytes. The remaining file methods may or may not be relevant in the context.

As I write this, the Python 3 documentation of memoryview says that it works with objects that "support the buffer protocol", which is only documented at the C API level. The bytearray constructor accepts an "an object conforming to the buffer interface". Now there is a move to adopt "bytes-like object" as a friendlier term[2]. I point this out to emphasize that "X-like object", "X protocol" and "X interface" are synonyms in the minds of Pythonistas.

One of the most fundamental interfaces in Python is the sequence protocol. The interpreter goes out of its way to handle objects that provide even a minimal implementation of that protocol, as the next section demonstrates.

Python digs sequences

The philosophy of the Python Data Model is to cooperate with essential protocols as much as possible. When it comes to sequences, Python tries hard to work with even the simplest implementations.

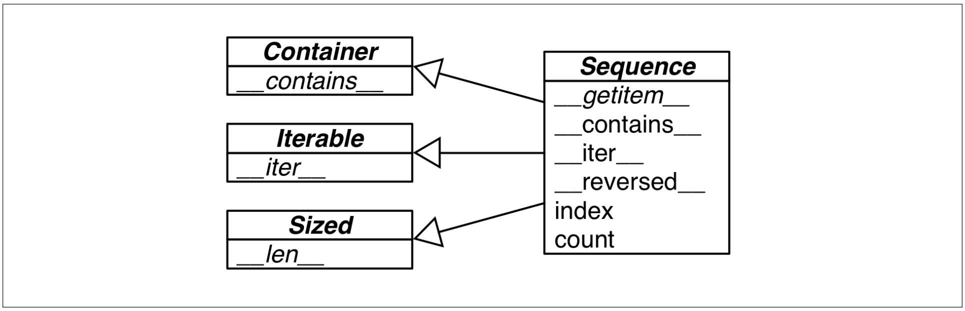


Figure 11-1. UML class diagram for the `Sequence` ABC and related abstract classes from `collections.abc`. Inheritance arrows point from subclass to its superclasses. Names in italic are abstract methods.

Figure 11-1 shows how the formal `Sequence` interface is defined as an ABC. Now, take a look at the `Foo` class in Example 11-3. It does not inherit from `abc.Sequence`, and it only implements one method of the sequence protocol: `__getitem__` (`__len__` is missing).

Example 11-3. Partial sequence protocol implementation with `__getitem__`: enough for item access, iteration and the `in` operator.

```
>>> class Foo:
...     def __getitem__(self, pos):
...         return range(0, 30, 10)[pos]
...
```

[2] Issue16518: add "buffer protocol" to glossary was actually resolved by replacing many mentions of "object that supports the buffer protocol/interface/API" with "bytes-like object"; a follow-up issue is Other mentions of the buffer protocol.

```
>>> f[1]
10
>>> f = Foo()
>>> for i in f: print(i)
...
0
10
20
>>> 20 in f
True
>>> 15 in f
False
```

There is no method `__iter__` yet `Foo` instances are iterable because — as a fall-back — when Python sees a `__getitem__` method, it tries to iterate over the object by calling that method with integer indexes starting with `0`. Since Python is smart enough to iterate over `Foo` instances, it can also make the `in` operator work even if `Foo` has no `__contains__` method: it does a full scan to check if an item is present.

In summary, given the importance of the sequence protocol, in the absence `__iter__` and `__contains__` Python still manages to make iteration and the `in` operator work by invoking `__getitem__`.

Our original `FrenchDeck` from Chapter 1 does not subclass from `abc.Sequence` either, but it does implement both methods of the sequence protocol: `__getitem__` and `__len__`. See Example 11-4.

Example 11-4. A deck as a sequence of cards (same as Example 1-1).

```
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]
```

A good part of the demos in Chapter 1 work because of the special treatment Python gives to anything vaguely resembling a sequence. Iteration in Python represents an extreme form of duck typing: the interpreter tries two different methods to iterate over objects.

Now let’s study another example emphasizing the dynamic nature of protocols.

Monkey-patching to implement a protocol at run time

The `FrenchDeck` class from Example 11-4 has a major flaw: it cannot be shuffled. Years ago when I first wrote the `FrenchDeck` example I did implement a `shuffle` method. Later I had a Pythonic insight: if a `FrenchDeck` acts like a sequence, then it doesn’t need its own `shuffle` method, because there is already `random.shuffle`, documented as “Shuffle the sequence *x* in place”.

> When you follow established protocols you improve your chances of leveraging existing standard library and third-party code, thanks to duck typing.

The standard `random.shuffle` function is used like this:

```
>>> from random import shuffle
>>> l = list(range(10))
>>> shuffle(l)
>>> l
[5, 2, 9, 7, 8, 3, 1, 4, 0, 6]
```

However, if we try to shuffle a `FrenchDeck` instance, we get an exception, as in Example 11-5.

Example 11-5. `random.shuffle` cannot handle `FrenchDeck`.

```
>>> from random import shuffle
>>> from frenchdeck import FrenchDeck
>>> deck = FrenchDeck()
>>> shuffle(deck)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../python3.3/random.py", line 265, in shuffle
    x[i], x[j] = x[j], x[i]
TypeError: 'FrenchDeck' object does not support item assignment
```

The error message is quite clear: `"'FrenchDeck'` object does not support item assignment”. The problem is that shuffle operates by swapping items inside the collection, and `FrenchDeck` only implements the *immutable* sequence protocol. Mutable sequences must also provide a `__setitem__` method.

Because Python is dynamic, we can fix this at runtime, even at the interactive console. Here is how to do it:

Example 11-6. Monkey patching `FrenchDeck` to make it mutable and compatible with `random.shuffle` (continuing from Example 11-5).

```
>>> def set_card(deck, position, card):  ❶
...     deck._cards[position] = card
...
>>> FrenchDeck.__setitem__ = set_card  ❷
>>> shuffle(deck)  ❸
>>> deck[:5]
[Card(rank='3', suit='hearts'), Card(rank='4', suit='diamonds'), Card(rank='4',
suit='clubs'), Card(rank='7', suit='hearts'), Card(rank='9', suit='spades')]
```

❶ create a function that takes `deck`, `position` and `card` as arguments.

❷ assign that function to an attribute named `__setitem__` in the `FrenchDeck` class.

❸ `deck` can now be sorted because `FrenchDeck` now implements the necessary method of the mutable sequence protocol.

The signature of the `__setitem__` special method is defined in the Emulating container types section of the Data model chapter of the Python Language Reference. Here we named the arguments `deck, position, card` — and not `self, key, value` as in the Language Reference — to show that every Python method starts life as a plain function, and naming the first argument `self` is merely a convention. This is OK in a console session, but in a Python source file it's much better to use `self`, `key` and `value` as documented.

The trick is that `set_card` knows that the `deck` object has an attribute named `_cards`, and `_cards` must be a mutable sequence. The `set_card` function is then attached to the `FrenchDeck` class as the `__setitem__` special method. This is an example of *monkey patching*: changing a class or module at run time, without touching the source code. Monkey patching is powerful, but the code that does the actual patching is very tightly coupled with the program to be patched, often handling private and undocumented parts.

Besides being an example of monkey patching, Example 11-6 highlights that protocols are dynamic: `random.shuffle` doesn't care what type of argument it gets, it only needs the object to implement part of the mutable sequence protocol. It doesn't even matter if the object was "born" with the necessary methods or if they were somehow acquired later.

The theme of this chapter so far has been "duck typing": operating with objects regardless of their types, as long as they implement certain protocols.

When we did present diagrams with ABCs, the intent was to show how the protocols are related to the explicit interfaces documented in the abstract classes, but we did not actually inherit from any ABC so far.

In the next sections we will leverage ABCs directly, and not just as documentation.

Waterfowl and ABCs

After reviewing the usual protocol-style interfaces of Python, we move to ABCs. But before diving into examples and details, Alex Martelli explains in a guest essay why ABCs were a great addition to Python.

I am very grateful to Alex Martelli. He was already the most cited person in this book before he became one of the technical editors. His insights have been invaluable, and then he offered to write this essay. We are incredibly lucky to have him. Take it away, Alex!

Waterfowl and ABCs

by Alex Martelli

I've been credited on Wikipedia for helping spread the helpful meme and sound-bite "*duck typing*“ — i.e, ignoring an object's actual type, focusing instead on ensuring that the object implements the method names, signatures, and semantics, required for its intended use.

In Python, this mostly boils down to avoiding the use of `isinstance` to check the object's type (not to mention the even worse approach of checking, e.g, whether `type(foo) is bar` — which is rightly anathema as it inhibits even the simplest forms of inheritance!).

The overall *duck typing* approach remains quite useful in many contexts — and yet, in many others, an often preferable one has evolved over time. And herein lies a tale…

In recent generations, the taxonomy of genus and species (including but not limited to the family of waterfowl known as Anatidae) has mostly been driven by *phenetics* — an approach focused on similarities of morphology and behavior… chiefly, **observable** traits. The analogy to "duck typing" was strong.

However, parallel evolution can often produce similar traits, both morphological and behavioral ones, among species that are actually unrelated, but just happened to evolve in similar, though separate, ecological niches. Similar "accidental similarities" happen in programming, too — e.g, consider the classic OOP example:

```
class Artist:
    def draw(self): ...
```

```
class Gunslinger:
    def draw(self): ...

class Lottery:
    def draw(self): ...
```

Clearly, the mere existence of a method called `draw`, callable without arguments, is far from sufficient to assure us that two objects `x` and `y` such that `x.draw()` and `y.draw()` can be called are in any way exchangeable or abstractly equivalent — nothing about the similarity of the semantics resulting from such calls can be inferred. Rather, we need a knowledgeable programmer to somehow positively **assert** that such an equivalence holds at some level!

In biology (and other disciplines) this issue has led to the emergence (and, on many facets, the dominance) of an approach that's an alternative to phenetics, known as *cladistics* — focusing taxonomical choices on characteristics that are inherited from common ancestors, rather than ones that are independently evolved. (Cheap and rapid DNA sequencing can make cladistics highly practical in many more cases, in recent years).

For example, sheldgeese (once classified as being closer to other geese) and shelducks (once classified as being closer to other ducks) are now grouped together within the subfamily Tadornidae (implying they're closer to each other than to any other Anatidae — sharing a closer common ancestor). Furthermore, DNA analysis has shown, in particular, that the White-winged Wood Duck is not as close to the Muscovy Duck (the latter being a shelduck) as similarity in looks and behavior had long suggested — so the wood duck was reclassified into its own genus, and entirely out of the subfamily!

Does this matter? It depends on the context! For such purposes as deciding how best to cook a waterfowl once you've bagged it, for example, specific observable traits (not all of them — plumage, for example, is de minimis in such a context :-), mostly texture and flavor (old-fashioned phenetics!), may be far more relevant than cladistics. But for other issues, such as susceptibility to different pathogens (whether you're trying to raise waterfowl in captivity, or preserve them in the wild), DNA closeness can matter much more…

So, by very loose analogy with these taxonomic revolutions in the world of waterfowls, I'm recommending supplementing (not entirely replacing — in certain contexts it shall still serve) good old *duck typing* with… *goose typing*!

What **goose typing** means is: `isinstance(obj, cls)` is now just fine… as long as `cls` is an Abstract Base Class — in other words, `cls`'s metaclass is `abc.ABCMeta`.

You can find many useful existing abstract classes in `collections.abc` — others yet in the `numbers` module of the Python Standard Library[3].

Among the many conceptual advantages of ABCs over concrete classes (such as, Scott Meyer's “all non-leaf classes should be abstract" — see Item 33 in his book, More Effective C++), Python's ABCs add one major practical advantage: the `register` class method, which lets end-user code “declare” that a certain class becomes a “virtual” subclass of an ABC (for this purpose the registered class must meet the ABC's method name and signature requirements, and more importantly the underlying semantic contract — but it need not have been developed with any awareness of the ABC, and in particular need not inherit from it!). This goes a long way towards breaking the rigidity and strong coupling that make inheritance something to use with much more caution than typically practiced by most OOP programmers…

Sometimes you don't even need to register a class for an ABC to recognize it as a subclass!

That's the case for the ABCs whose essence boils down to a few special methods. For example:

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
```

As you see, `abc.Sized` recognizes `Struggle` as “a subclass”, with no need for registration — because implementing the special method named `__len__` is all it takes (it's supposed to be implemented with the proper syntax — callable without arguments — and semantics — returning a non-negative integer denoting an object's “length”; any code that implements a specially named method, such as `__len__`, with arbitrary, non-compliant syntax and semantics has much-worse problems anyway :-).

So, here's my valediction…: whenever you're implementing a class embodying any of the concepts represented in the ABCs in `numbers`, `collections.abc`, or other framework you may be using, be sure (if needed) to subclass it from, or register it into, the corresponding ABC. At the start of your programs using some library or framework defining classes which have omitted to do that, perform the registrations yourself; then, when you must check for (most typically) an argument being, e.g, “a sequence”, check whether:

```
isinstance(the_arg, collections.abc.Sequence)
```

[3] You can also, of course, define your own ABCs — but I would discourage all but the most advanced Pythonistas from going that route, just as I would discourage them from defining their own custom metaclasses… and even for said “most advanced Pythonistas”, those of us sporting deep mastery of every fold and crease in the language, these are not tools for frequent use: such “deep metaprogramming”, if ever appropriate, is intended for authors of broad frameworks meant to be independently extended by vast numbers of separate development teams… less than 1% of “most advanced Pythonistas” may ever need that! — *A.M.*

And, **don’t** define custom ABCs (or metaclasses) in production code… if you feel the urge to do so, I’d bet it’s likely to be a case of “all problems look like a nail”-syndrome for somebody who just got a shiny new hammer — you (and future maintainers of your code) will be much happier sticking with straightforward and simple code, eschewing such depths. *Valē!*

Besides coining the “goose typing”, Alex makes the point that inheriting from an ABC is more than implementing the required methods: it’s also a clear declaration of intent by the developer. That intent can also be made explicit through registering a virtual subclass.

In addition, the use of `isinstance` and `issubclass` becomes more acceptable to test against ABCs. In the past these functions worked against duck typing, but with ABCs they become more flexible. After all, if a component does not implement an ABC by subclassing, it can always be registered after the fact so it passes those explicit type checks.

However, even with ABCs, you should beware that excessive use of `isinstance` checks may be a *code smell* — a symptom of bad OO design. It’s usually **not** OK to have a chain of `if/elif/elif` with `insinstance` checks performing different actions depending on the type of an object: you should be using polymorphism for that — i.e. designing your classes so that the interpreter dispatches calls to the proper methods, instead of you hard coding the dispatch logic in `if/elif/elif` blocks.

> There is a common, practical exception to the recommendation above: some Python APIs accept a single `str` or a sequence of `str` items; if it’s just a single `str`, you want to wrap it in a `list`, to ease processing. Since `str` is a sequence type, the simplest way to distinguish it from any other immutable sequence is to do an explicit `isinstance(x, str)` check[4].

On the other hand, it’s usually OK to perform an `insinstance` check against an ABC if you must enforce an API contract: “Dude, you have to implement this if you want to call me”, as technical reviewer Lennart Regebro put it. That’s particularly useful in systems that have a plug-in architecture. Outside of frameworks, duck typing is often simpler and more flexible than type checks.

[4] Unfortunately in Python 3.4 there is no ABC that helps distinguish a `str` from `tuple` or other immutable sequences, so we must test against `str`. In Python 2 the `basestr` type exists to help with tests like these. It’s not an ABC, but it’s a superclass of both `str` and `unicode`, but in Python 3 `basestr` is gone. Curiously, there is in Python 3 a `collections.abc.ByteString` type, but it only helps detecting `bytes` and `bytearray`.

For example, in several classes in this book when I needed to take a sequence of items and process them as a `list`, instead of requiring a `list` argument by type checking, I simply took the argument and immediately built a `list` from it: that way I can accept any iterable, and if the argument is not iterable, the call will fail soon enough with a very clear message. One example of this code pattern is in the `__init__` method in Example 11-13, later in this chapter. Of course, this approach wouldn't work if the sequence argument shouldn't be copied, either because it's too large or because my code needs to change it in-place. Then an `insinstance(x, abc.MutableSequence)` would be better. If any iterable is acceptable, then calling `iter(x)` to obtain an iterator would be the way to go, as we'll see in "Why sequences are iterable: the `iter` function" on page 406.

Another example is how you might imitate the handling of the `field_names` argument in `collections.namedtuple`: `field_names` accepts a single string with identifiers separated by spaces or commas, or a sequence of identifiers. It might be tempting to use `isinstance`, but Example 11-7 shows how I'd do it using duck typing[5].

Example 11-7. Duck typing to handle a string or an iterable of strings.

```
try:  ❶
    field_names = field_names.replace(',', ' ').split()  ❷
except AttributeError:  ❸
    pass  ❹
field_names = tuple(field_names)  ❺
```

❶ Assume it's a string (EAFP = it's easier to ask forgiveness than permission).

❷ Convert commas to spaces and split the result into a list of names.

❸ Sorry, `field_names` doesn't quack like a `str`… there's either no `.replace`, or it returns something we can't `.split`.

❹ Now we assume it's already an iterable of names.

❺ To make sure it's an iterable and to keep our own copy, create a tuple out of what he have.

Finally, in his essay Alex reinforces more than once the need for restraint in the creation of ABCs. An ABC epidemic would be disastrous, imposing excessive ceremony in a language that became popular because it's practical and pragmatic. During the *Fluent Python* review process, Alex wrote:

> ABCs are meant to encapsulate very general concepts, abstractions, introduced by a framework — things like "a sequence" and "an exact number". [Readers] most likely don't need to write any new ABCs, just use existing ones correctly, to get 99.9% of the benefits without serious risk of mis-design.

[5] This snippet was extracted from Example 21-2.

Now let's see goose typing in practice.

Subclassing an ABC

Following Martelli's advice we'll leverage an existing ABC, collections.MutableSequence, before daring to invent our own. In Example 11-8 FrenchDeck2 is explicitly declared a sub-class of collections.MutableSequence.

Example 11-8. frenchdeck2.py: FrenchDeck2, a subclass of collections.MutableSequence.

```
import collections

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

class FrenchDeck2(collections.MutableSequence):
    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]

    def __setitem__(self, position, value):  # ❶
        self._cards[position] = value

    def __delitem__(self, position):  # ❷
        del self._cards[position]

    def insert(self, position, value):  # ❸
        self._cards.insert(position, value)
```

❶ __setitem__ is all we need to enable shuffling…

❷ But subclassing MutableSequence forces us to implement __delitem__, an abstract method of that ABC.

❸ We are also required to implement insert, the third abstract method of MutableSequence.

Python does not check for the implementation of the abstract methods at import time (when the frenchdeck2.py module is loaded and compiled), but only at runtime when we actually try to instantiate FrenchDeck2. Then, if we fail to implement any abstract method we get a `TypeError` exception with a message such as `"Can't instantiate abstract class FrenchDeck2 with abstract methods __delitem__, insert"`. That's why we must implement `__delitem__` and `insert`, even if our `FrenchDeck2` examples to not need those behaviors: the `MutableSequence` ABC demands them.

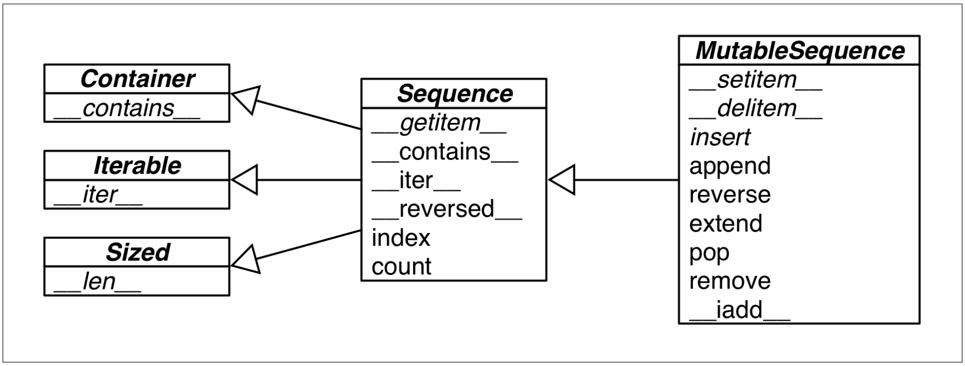


Figure 11-2. UML class diagram for the `MutableSequence` ABC and its superclasses from `collections.abc`. Inheritance arrows point from subclasses to ancestors. Names in italic are abstract classes and abstract methods.

As Figure 11-2 shows, not all methods of the `Sequence` and `MutableSequence` ABCs are abstract. From `Sequence`, `FrenchDeck2` inherits the ready-to-use concrete methods `__contains__`, `__iter__`, `__reversed__`, `index`, and `count`; from `MutableSequence`, it gets `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`.

The concrete methods in each `collections.abc` ABC are implemented in terms of the public interface of the class, so they work without any knowledge of the internal structure of instances.

As the coder of a concrete subclass, you may be able to override methods inherited from ABCs with more efficient implementations. For example, `__contains__` works by doing a full scan of the sequence, but if your concrete sequence keeps its items sorted, you can write a faster `__contains__` that does a binary search using `bisect` function (see "Managing ordered sequences with `bisect`" on page 44).

To use ABCs well you need to know what's available. We'll go over the collections ABCs next.

ABCs in the standard library.

Since Python 2.6 ABCs are available in the standard library. Most are defined in the `collections.abc` module, but there are others. You can find ABCs in the `numbers` and `io` packages, for example. But the most widely used is `collections.abc`. Let's see what is available there.

ABCs in `collections.abc`

There are two modules named `abc` in the standard library. Here we are talking about `collections.abc`. To reduce loading time, in Python 3.4 it's implemented outside of the `collections` package, in Lib/_collections_abc.py), so it's imported separately from `collections`. The other `abc` module is just `abc`, i.e. `Lib/abc.py`, where the `abc.ABC` class is defined. Every ABC depends on it, but we don't need to import it ourselves except to create a new ABC.

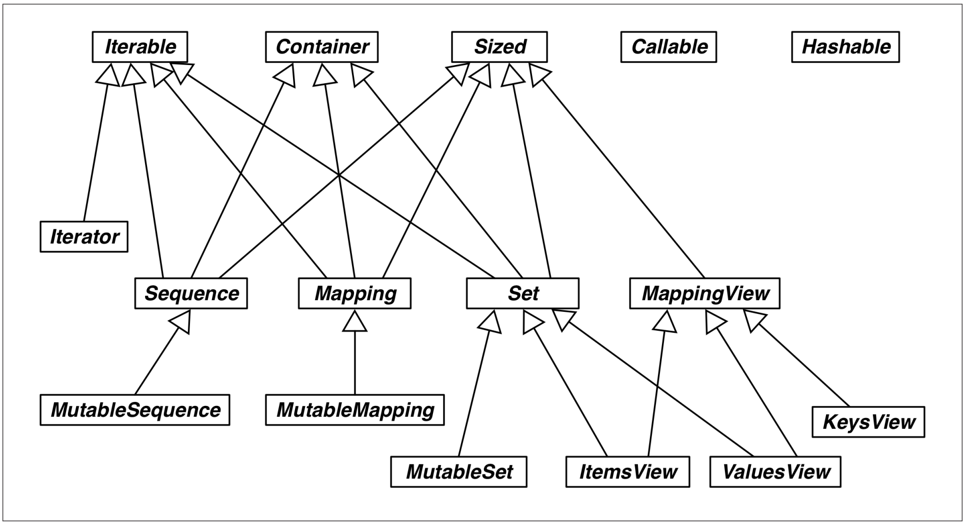


Figure 11-3. UML class diagram for ABCs in `collections.abc`.

Figure 11-3 is a summary UML class diagram (without attribute names) of all 16 ABCs defined in `collections.abc` as of Python 3.4. The official documentation of `collections.abc` has a nice table summarizing the ABCs, their relationships and their abstract and concrete methods (called "mixin methods"). There is plenty of multiple inheritance going on in Figure 11-3. We'll devote most of Chapter 12 to multiple inheritance, but for now it's enough to say that it is usually not a problem when ABCs are concerned[6].

Let's go over the clusters in Figure 11-3:

`Iterable`, `Container` *and* `Sized`
: Every collection should either inherit from these ABCs or at least implement compatible protocols. `Iterable` supports iteration with `__iter__`, `Container` supports the `in` operator with `__contains__` and `Sized` supports `len()` with `__len__`.

`Sequence`, `Mapping` *and* `Set`
: These are the main immutable collection types, and each has a mutable subclass. A detailed diagram for `MutableSequence` is in Figure 11-2; for `MutableMapping` and `MutableSet` there are diagrams in Chapter 3 (Figure 3-1 and Figure 3-2).

`MappingView`
: In Python 3, the objects returned from the mapping methods `.items()`, `.keys()` and `.values()` inherit from `ItemsView`, `ValuesView` and `ValuesView`, respectively. The first two also inherit the rich interface of `Set`, with all the operators we saw in "Set operations" on page 82.

`Callable` *and* `Hashable`
: These ABCs are not so closely related to collections, but `collections.abc` was the first package to define ABCs in the standard library, and these two were deemed important enough to be included. I've never seen subclasses of `Callable` or `Hashable`. Their main use is to support the `insinstance` built-in as a safe way of determining whether an object is callable or hashable[7].

`Iterator`
: Note that iterator subclasses `Iterable`. We discuss this further in Chapter 14.

After the `collections.abc` package, the most useful package of ABCs in the standard library is `numbers`, covered next.

The `numbers` tower of ABCs

The `numbers` package defines the so called "numerical tower", i.e. this linear hierarchy of ABCs, where `Number` is the topmost superclass, `Complex` is its immediate subclass and so on, down to `Integral`:

[6] Multiple inheritance was *considered harmful* and excluded from Java, except for interfaces: Java interfaces can extend multiple interfaces, and Java classes can implement multiple interfaces.

[7] For callable detection there is the `callable()` built-in function — but there is no equivalent `hashable()` function, so `isinstance(my_obj, Hashable)` is the preferred way to test for a hashable object.

- `Number`
- `Complex`
- `Real`
- `Rational`
- `Integral`

So if you need to check for an integer, use `isinstance(x, numbers.Integral)` to accept `int`, `bool` (which subclasses `int`) or other integer types that may be provided by external libraries which register their types with the `numbers` ABCs. And you or the users of your API may always register any compatible type as a virtual subclass of `numbers.Integral` to satisfy your check.

If, on the other hand, a value can be a floating point type, you write `isinstance(x, numbers.Real)`, and your code will be happy to take `bool`, `int`, `float`, `fractions.Fraction` or any other non-complex numerical type provided by an external library, such as NumPy which is suitably registered.

Somewhat surprisingly, `decimal.Decimal` is not registered as a virtual subclass of `numbers.Real`. The reason is that, if you need the precision of `Decimal` in your program, then you want to be protected from accidental mixing of decimals with other less precise numeric types, particularly floats.

After looking at some existing ABCs, let's practice goose typing by implementing an ABC from scratch and putting it to use. The goal here is not to encourage everyone to start coding ABCs left and right, but to learn how to read the source code of the ABCs you'll find in the standard library and other packages.

Defining and using an ABC

To justify creating an ABC we need to come up with a context for using it as an extension point in a framework. So here is our our context: imagine you need to to display advertisements in a web site or a mobile app in random order, but without repeating an ad before the full inventory of ads is shown. Now let's assume we are building an ad management framework called `ADAM`. One of its requirements is to support user-provided non-repeating random-picking classes[8]. To make it clear to `ADAM` users what is expected of a "non-repeating random-picking" component, we'll define an ABC.

[8] Perhaps the client needs to audit the randomizer; or the agency wants to provide a rigged one. You never know…

Taking a clue from “stack” and “queue" — which describe abstract interfaces in terms of physical arrangements of objects, I will use a real world metaphor to name our ABC: bingo cages and lottery blowers are machines designed to pick items at random from a finite set, without repeating until the set is exhausted. The ABC will be named `Tombola`, after the Italian name of bingo and the tumbling container which mixes the numbers[9].

The `Tombola` ABC has four methods. The two abstract methods are:

- `.load(…)`: put items into the container.
- `.pick()`: remove one item at random from the container, returning it.

The concrete methods are:

- `.loaded()`: return `True` if there is at least one item in the container.
- `.inspect()`: return a sorted `tuple` built from the items currently in the container, without changing its contents (its internal ordering is not preserved).

Figure 11-4 shows the `Tombola` ABC and three concrete implementations.

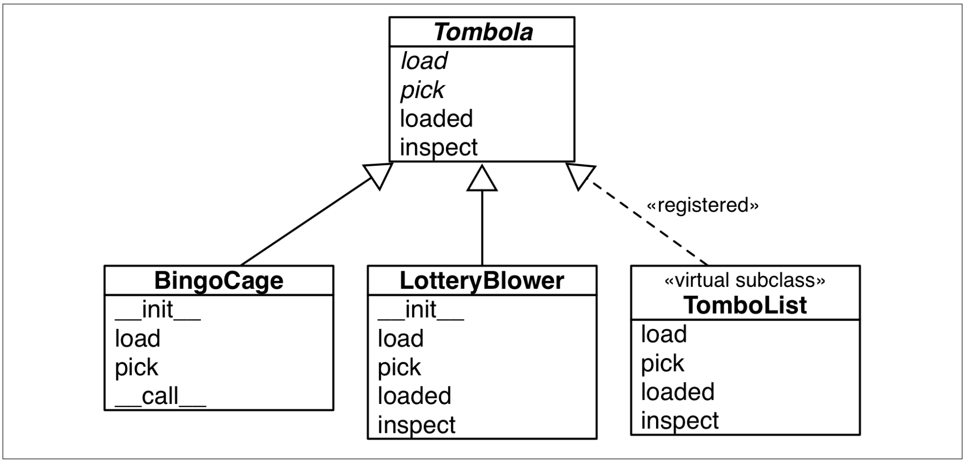


Figure 11-4. UML diagram for an ABC and three subclasses. The name of the `Tombola` ABC and its abstract methods are written in italics, per UML conventions. The dashed arrow is used for interface implementation, here we are using it to show that `Tombo`

[9] The Oxford English Dictionary defines tombola as “A kind of lottery resembling lotto.”

`List` is a virtual subclass of `Tombola` because it is registered, as we will see later in this chapter[10]*.*

Example 11-9 shows the definition of the `Tombola` ABC:

Example 11-9. `tombola.py`: `Tombola` is an ABC with two abstract methods and two concrete methods.

```
import abc

class Tombola(abc.ABC):  ❶

    @abc.abstractmethod
    def load(self, iterable):  ❷
        """Add items from an iterable."""

    @abc.abstractmethod
    def pick(self):  ❸
        """Remove item at random, returning it.

        This method should raise `LookupError` when the instance is empty.
        """

    def loaded(self):  ❹
        """Return `True` if there's at least 1 item, `False` otherwise."""
        return bool(self.inspect())  ❺


    def inspect(self):
        """Return a sorted tuple with the items currently inside."""
        items = []
        while True:  ❻
            try:
                items.append(self.pick())
            except LookupError:
                break
        self.load(items)  ❼
        return tuple(sorted(items))
```

❶ To define an ABC, subclass `abc.ABC`.

❷ An abstract method is marked with the `@abstractmethod` decorator, and often its body is empty except for a docstring[11].

[10] *«registered» and «virtual subclass» are not standard UML words, we are using them to represent a class relationship that is specific to Python.*

[11] Before ABCs existed, abstract methods would use the statement `raise NotImplementedError` to signal that subclasses were responsible for their implementation.

❸ The docstring instructs implementers to raise `LookupError` if there are no items to pick.

❹ An ABC may include concrete methods.

❺ Concrete methods in an ABC must rely only on the interface defined by the ABC — i.e. other concrete or abstract methods or properties of the ABC.

❻ We can't know how concrete subclasses will store the items, but we can build the `inspect` result by emptying the `Tombola` with successive calls `.pick()`…

❼ …then use `.load(…)` to put everything back.

> An abstract method can actually have an implementation. Even if it does, subclasses will still be forced to override it, but they will be able to invoke the abstract method with `super()`, adding functionality to it instead of implementing from scratch. See the abc module documentation. for details on `@abstractmethod` usage.

The `.inspect()` method in Example 11-9 is perhaps a silly example, but it shows that, given `.pick()` and `.load(…)` we can inspect what's inside the `Tombola` by picking all items and loading them back. The point of this example is to highlight that it's OK to provide concrete methods in ABCs, as long as they only depend on other methods in the interface. Being aware of their internal data structures, concrete subclasses of `Tombola` may always override `.inspect()` with a smarter implementation, but they don't have to.

The `.loaded()` method in Example 11-9 may not be as silly, but it's expensive: it calls `.inspect()` to build the sorted `tuple` just to apply `bool()` on it. This works, but a concrete subclass can do much better, as we'll see.

Note that our roundabout implementation of `.inspect()` requires that we catch a `LookupError` thrown by `self.pick()`. The fact that `self.pick()` may raise `LookupError` is also part of its interface, but there is no way to declare this in Python, except in the documentation (see the docstring for the abstract `pick` method in Example 11-9.)

I chose the `LookupError` exception because of its place in the Python hierarchy of exceptions in relation to `IndexError` and `KeyError`, the most likely exceptions to be raised by the data structures used to implement a concrete `Tombola`. Therefore implementations can raise `LookupError`, `IndexError` or `KeyError` to comply See Example 11-10.

Example 11-10. Part of the `Exception` class hierarchy. Complete tree at the Built-in Exceptions page of the Python Standard Library documentation.

```
BaseException
 ├── SystemExit
 ├── KeyboardInterrupt
 ├── GeneratorExit
 └── Exception
      ├── StopIteration
      ├── ArithmeticError
      |    ├── FloatingPointError
      |    ├── OverflowError
      |    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ImportError
      ├── LookupError  ❶
      |    ├── IndexError  ❷
      |    └── KeyError  ❸
      ├── MemoryError
      ... etc.
```

❶ `LookupError` is the exception we handle in `Tombola.inspect`.

❷ `IndexError` is the `LookupError` subclass raised when we try to get a item from a sequence with an index beyond the last position.

❸ `KeyError` is raised when we use a non-existent key to get an item from a mapping.

We now have our very own `Tombola` ABC. To witness the interface checking performed by an ABC, let's try to fool `Tombola` with a defective implementation in Example 11-11.

Example 11-11. A fake `Tombola` doesn't go undetected.

```
>>> from tombola import Tombola
>>> class Fake(Tombola):  # ❶
...     def pick(self):
...         return 13
...
>>> Fake  # ❷
<class '__main__.Fake'>
<class 'abc.ABC'>, <class 'object'>)
>>> f = Fake()  # ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Fake with abstract methods load
```

❶ Declare `Fake` as a subclass of `Tombola`.

❷ The class was created, no errors so far.

❸ `TypeError` is raised when we try to instantiate `Fake`. The message is very clear: `Fake` is considered abstract because it failed to implement `load`, one of the abstract methods declared in the `Tombola` ABC.

So we have our first ABC defined, and we put it to work validating a class. We'll soon subclass the `Tombola` ABC, but first we must cover some ABC coding rules.

ABC syntax details

The best way to declare an ABC is to subclass `abc.ABC` or any other ABC.

However, the `abc.ABC` class is new in Python 3.4, so if you are using an earlier version of Python — and it does not make sense to subclass another existing ABC — then you must use the `metaclass=` keyword in the `class` statement, pointing to `abc.ABCMeta` (not `abc.ABC`). In Example 11-9 we would write:

```
class Tombola(metaclass=abc.ABCMeta):
    # ...
```

The `metaclass=` keyword argument was introduced in Python 3. In Python 2, you must use the `__metaclass__` class attribute:

```
class Tombola(object):  # this is Python 2!!!
    __metaclass__ = abc.ABCMeta
    # ...
```

We'll explain metaclasses in Chapter 21, for now let's accept that a metaclass as a special kind of class, and agree that an ABC is a special kind of class; for example, "regular" classes don't check subclasses, so this is a special behavior of ABCs.

Besides the `@abstractmethod`, the `abc` module defines the `@abstractclassmethod`, `@abstractstaticmethod` and `@abstractproperty` decorators. However, these last three are deprecated since Python 3.3, when it became possible to stack decorators on top of `@abstractmethod`, making the others redundant. For example, the preferred way to declare an abstract class method is:

```
class MyABC(abc.ABC):
    @classmethod
    @abc.abstractmethod
    def an_abstract_classmethod(cls, ...):
        pass
```

The order of stacked function decorators usually matters, and in the case of `@abstractmethod` the documentation is explicit:

> When abstractmethod() is applied in combination with other method descriptors, it should be applied as the innermost decorator, …[12]

In other words, no other decorator may appear between `@abstractmethod` and the `def` statement).

[12] `@abc.abstractmethod` entry in the abc module documentation.

Now that we got these ABC syntax issues covered, let’s put `Tombola` to use by implementing some full-fledged concrete descendants of it.

Subclassing the `Tombola` ABC

Given the `Tombola` ABC, we’ll now develop two concrete subclasses that satisfy it’s interface. These classes were pictured in Figure 11-4, along with the virtual subclass to be discussed in the next section.

The `BingoCage` class in Example 11-12 is a variation of Example 5-8 using a better randomizer. This `BingoCage` implements the required abstract methods `load` and `pick`, inherits `loaded` from `Tombola`, overrides `inspect` and adds `__call__`.

Example 11-12. `bingo.py`: `BingoCage` is a concrete subclass of `Tombola`.

```
import random

from tombola import Tombola


class BingoCage(Tombola):  ❶

    def __init__(self, items):
        self._randomizer = random.SystemRandom()  ❷
        self._items = []
        self.load(items)  ❸

    def load(self, items):
        self._items.extend(items)
        self._randomizer.shuffle(self._items)  ❹

    def pick(self):  ❺
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')

    def __call__(self):  ❻
        self.pick()
```

❶ This `BingoCage` class explicitly extends `Tombola`.

❷ Pretend we’ll use this for online gaming. `random.SystemRandom` implements the `random` API on top of the `os.urandom(…)` function which provides random bytes “suitable for cryptographic use” according to the os module docs.

❸ Delegate initial loading to the `.load(…)` method.

❹ Instead of the plain `random.shuffle()` function, we use the `.shuffle()` method of our `SystemRandom` instance.

❺ `pick` is implemented as in Example 5-8.

❻ `__call__` is also from Example 5-8. It's not needed to satisfy the `Tombola` interface, but there's no harm in adding extra methods.

`BingoCage` inherits the expensive `loaded` and the silly `inspect` methods from `Tombola`. Both could be overridden with much faster, one-liners, as in Example 11-13. The point is: we can be lazy and just inherit the sub-optimal concrete methods from an ABC. The methods inherited from `Tombola` are not as fast as they could be for `BingoCage`, but they do provide correct results for any `Tombola` subclass that correctly implements `pick` and `load`.

Example 11-13 shows a very different but equally valid implementation of the `Tombola` interface. Instead of shuffling the "balls" and popping the last, `LotteryBlower` pops from a random position.

Example 11-13. `lotto.py`: `LotteryBlower` is a concrete subclass that overrides the `inspect` and `loaded` methods from `Tombola`.

```
import random

from tombola import Tombola


class LotteryBlower(Tombola):

    def __init__(self, iterable):
        self._balls = list(iterable)  ❶

    def load(self, iterable):
        self._balls.extend(iterable)

    def pick(self):
        try:
            position = random.randrange(len(self._balls))  ❷
        except ValueError:
            raise LookupError('pick from empty BingoCage')
        return self._balls.pop(position)  ❸

    def loaded(self):  ❹
        return bool(self._balls)

    def inspect(self):  ❺
        return tuple(sorted(self._balls))
```

❶ The initializer accepts any iterable: the argument is used to build a list.

❷ The `random.randrange(…)` function raises `ValueError` if the range is empty, so we catch that and throw `LookupError` instead, to be compatible with `Tombola`.

❸ Otherwise the randomly selected item is popped from `self._balls`.

❹ Override `loaded` to avoid calling `inspect` (as `Tombola.loaded` does in Example 11-9). We can make it faster by working with `self._balls` directly — no need to build a whole sorted `tuple`.

❺ Override `inspect` with one-liner.

Example 11-13 illustrates an idiom worth mentioning: in `__init__`, `self._balls` stores `list(iterable)` and not just a reference to `iterable` (i.e. we did not merely assign `iterable` to `self._balls`). As mentioned before[13], this makes our `LotteryBlower` flexible because the `iterable` argument may be any iterable type. At the same time, we make sure to store its items in a `list` so we can `pop` items. And even if we always get lists as the `iterable` argument, `list(iterable)` produces a copy of the argument, which is a good practice considering we will be removing items from it and the client may not be expecting the `list` of items she provided to be changed[14].

We now come to the crucial dynamic feature of goose typing: declaring virtual subclasses with the `register` method.

A virtual subclass of `Tombola`

An essential characteristic of goose typing — and the reason why it deserves a waterfowl name — is the ability to register a class as a *virtual subclass* of an ABC, even if it does not inherit from it. When doing so, we promise that the class faithfully implements the interface defined in the ABC — and Python will believe us without checking. If we lie, we'll be caught by the usual runtime exceptions.

This is done by calling a `register` method on the ABC. The registered class then becomes a virtual subclass of the ABC, and will be recognized as such by functions like `issubclass` and `isinstance`, but it will not inherit any methods or attributes from the ABC.

> Virtual subclasses do not inherit from their registered ABCs, and are not checked for conformance to the ABC interface at any time, not even when they are instantiated. It's up to the subclass to actually implement all the methods needed to avoid runtime errors.

[13] I gave this as an example of duck typing after Martelli's "Waterfowl and ABCs" on page 316.

[14] "Defensive programming with mutable parameters" on page 232 in Chapter 8 was devoted to the aliasing issue we just avoided here.

The `register` method is usually invoked as a plain function (see "Usage of `register` in practice" on page 339), but it can also be used as a decorator. In Example 11-14 we use the decorator syntax and implement `TomboList`, a virtual subclass of `Tombola` depicted in Figure 11-5.

`TomboList` works as advertised, and the doctests that prove it are described in "How the `Tombola` subclasses were tested" on page 336.

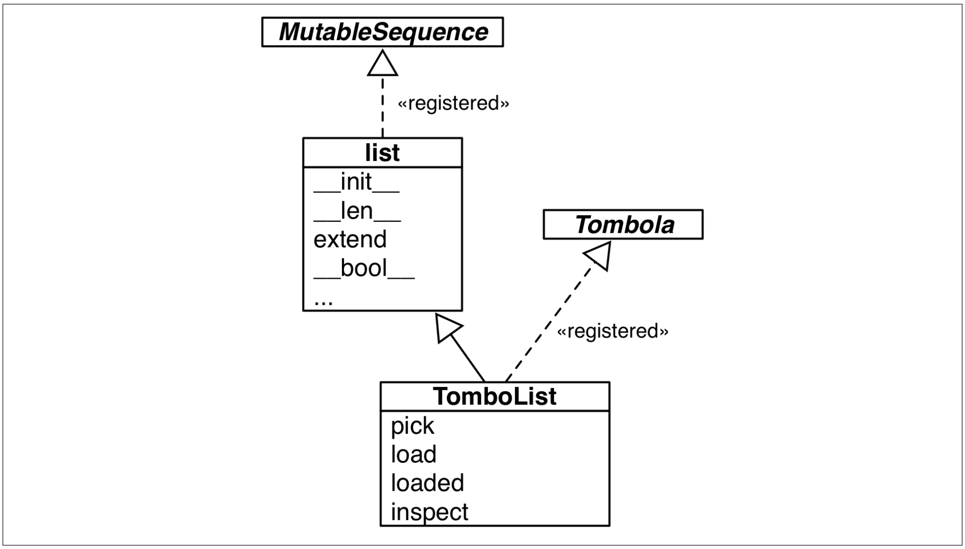


Figure 11-5. UML class diagram for the `TomboList`, a real subclass of `list` and a virtual subclass of `Tombola`.

Example 11-14. `tombolist.py`: class `TomboList` is a virtual subclass of `Tombola`.

```
from random import randrange

from tombola import Tombola

@Tombola.register  # ❶
class TomboList(list):  # ❷

    def pick(self):
        if self:  # ❸
            position = randrange(len(self))
            return self.pop(position)  # ❹
        else:
            raise LookupError('pop from empty TomboList')

    load = list.extend  # ❺
    def loaded(self):
        return bool(self)  # ❻

    def inspect(self):
        return tuple(sorted(self))

# Tombola.register(TomboList)  # ❼
```

❶ Tombolist is registered as a virtual subclass of Tombola.

❷ Tombolist extends list.

❸ Tombolist inherits __bool__ from list, and that returns True if the list is not empty.

❹ Our pick calls self.pop, inherited from list, passing a random item index.

❺ Tombolist.load is the same as list.extend.

❻ loaded delegates to bool`footnote:[The same trick I used with `load doesn't work with loaded, because the list type does not implement __bool__, the method I'd have to bind to loaded. On the other hand, the bool built-in function doesn't need __bool__ to work, it can also use __len__. See Truth Value Testing in the *Built-in Types* documentation.].

❼ If you're using Python 3.3 or earlier, you can't use .register as a class decorator. You must use standard call syntax.

Note that because of the registration, the functions issubclass and isinstance act as if TomboList is a subclass of Tombola:

```
>>> from tombola import Tombola
>>> from tombolist import TomboList
>>> issubclass(TomboList, Tombola)
True
>>> t = TomboList(range(100))
>>> isinstance(t, Tombola)
True
```

However, inheritance is guided by a special class attribute named __mro__ — the Method Resolution Order. It basically lists the class and its superclasses in the order Python uses to search for methods[15]. If you inspect the __mro__ of TomboList, you'll see that it lists only the "real" superclasses: list and object.

```
>>> TomboList.__mro__
(<class 'tombolist.TomboList'>, <class 'list'>, <class 'object'>)
```

[15] There is a whole section explaining the __mro__ class attribute in "Multiple inheritance and method resolution order" on page 353. Right now, this quick explanation will do.

Tombola is not in Tombolist.__mro__, so Tombolist does not inherit any methods from Tombola.

As I coded different classes to implement the same interface, I wanted a way to submit them all to the same suite of doctests. The next section shows how I leveraged the API of regular classes and ABCs to do it.

How the Tombola subclasses were tested

The script I used to test the Tombola examples uses two class attributes that allow introspection of a class hierarchy:

__subclasses__()
: Method that returns a list of the immediate subclasses of the class. The list does not include virtual subclasses.

_abc_registry
: Data attribute — available only in ABCs — that is bound to a WeakSet with weak references to registered virtual subclasses of the abstract class.

To test all Tombola subclasses, I wrote a script to iterate over a list built from Tombola.__subclasses__() and Tombola._abc_registry, and bind each class to the name ConcreteTombola used in the doctests.

A successful run of the test script looks like this:

```
$ python3 tombola_runner.py
BingoCage        23 tests,  0 failed - OK
LotteryBlower    23 tests,  0 failed - OK
TumblingDrum     23 tests,  0 failed - OK
TomboList        23 tests,  0 failed - OK
```

The test script is Example 11-15 and the doctests are in Example 11-16.

Example 11-15. tombola_runner.py: test runner for Tombola subclasses.

```
import doctest

from tombola import Tombola

# modules to test
import bingo, lotto, tombolist, drum  ❶

TEST_FILE = 'tombola_tests.rst'
TEST_MSG = '{0:16} {1.attempted:2} tests, {1.failed:2} failed - {2}'


def main(argv):
    verbose = '-v' in argv
    real_subclasses = Tombola.__subclasses__()  ❷
    virtual_subclasses = list(Tombola._abc_registry)  ❸

    for cls in real_subclasses + virtual_subclasses:  ❹
        test(cls, verbose)


def test(cls, verbose=False):

    res = doctest.testfile(
            TEST_FILE,
            globs={'ConcreteTombola': cls},  ❺
            verbose=verbose,
            optionflags=doctest.REPORT_ONLY_FIRST_FAILURE)
    tag = 'FAIL' if res.failed else 'OK'
    print(TEST_MSG.format(cls.__name__, res, tag))  ❻


if __name__ == '__main__':
    import sys
    main(sys.argv)
```

❶ Import modules containing real or virtual subclasses of `Tombola` for testing.

❷ `__subclasses__()` lists the direct descendants that are alive in memory. That's why we imported the modules to test, even if there is no further mention of them in the source code: to load the classes into memory.

❸ Build a `list` from `_abc_registry` (which is a `WeakSet`) so we can concatenate it with the result of `__subclasses__()`.

❹ Iterate over the subclasses found, passing each to the `test` function.

❺ The `cls` argument — the class to be tested — is bound to the name `ConcreteTombola` in the global namespace provided to run the doctest.

❻ The test result is printed with the name of the class, the number of tests attempted, tests failed and an `'OK'` or `'FAIL'` label.

The doctest file is Example 11-16.

Example 11-16. `tombola_tests.rst`: doctests for `Tombola` subclasses.

```
==============
Tombola tests
==============

Every concrete subclass of Tombola should pass these tests.


Create and load instance from iterable::

    >>> balls = list(range(3))
    >>> globe = ConcreteTombola(balls)
    >>> globe.loaded()
    True
    >>> globe.inspect()
    (0, 1, 2)


Pick and collect balls::

    >>> picks = []
    >>> picks.append(globe.pick())
    >>> picks.append(globe.pick())
    >>> picks.append(globe.pick())


Check state and results::

    >>> globe.loaded()
    False
    >>> sorted(picks) == balls
    True


Reload::

    >>> globe.load(balls)
    >>> globe.loaded()
    True
    >>> picks = [globe.pick() for i in balls]
    >>> globe.loaded()
    False


Check that `LookupError` (or a subclass) is the exception
thrown when the device is empty::

    >>> globe = ConcreteTombola([])
    >>> try:
    ...     globe.pick()
    ... except LookupError as exc:
    ...     print('OK')
    OK


Load and pick 100 balls to verify that they all come out::

    >>> balls = list(range(100))
    >>> globe = ConcreteTombola(balls)
    >>> picks = []
    >>> while globe.inspect():
    ...     picks.append(globe.pick())
    >>> len(picks) == len(balls)
    True
    >>> set(picks) == set(balls)
    True


Check that the order has changed and is not simply reversed::

    >>> picks != balls
    True
    >>> picks[::-1] != balls
    True

Note: the previous 2 tests have a *very* small chance of failing
even if the implementation is OK. The probability of the 100
balls coming out, by chance, in the order they were inspect is
1/100!, or approximately 1.07e-158. It's much easier to win the
Lotto or to become a billionaire working as a programmer.

THE END
```

This concludes our `Tombola` ABC case study. The next section addresses how the `register` ABC function is used in the wild.

Usage of `register` in practice

In Example 11-14 we used `Tombola.register` as a class decorator. Prior to Python 3.3, `register` could not be used like that — it had to be called as plain function after the class definition, as suggested by the comment at the end of Example 11-14.

However, even if `register` can now be used as a decorator, it's more widely deployed as a function to register classes defined elsewhere. For example, in the source code for the `collections.abc` module, the built-in types `tuple`, `str`, `range` and `memoryview` are registered as virtual subclasses of `Sequence` like this:

```
    Sequence.register(tuple)
    Sequence.register(str)
    Sequence.register(range)
    Sequence.register(memoryview)
```

Several other built-in types are registered to ABCs in `_collections_abc.py`. Those registrations happen only when that module is imported, which is OK because you'll have to import it anyway to get the ABCs: you need access to `MutableMapping` to be able to write `isinstance(my_dict, MutableMapping)`.

We'll wrap up this chapter explaining a bit of ABC magic that Alex Martelli performed in "Waterfowl and ABCs" on page 316.

Geese can behave as ducks

In his *Waterfowl and ABCs* essay, Alex shows that a class can be recognized as a virtual subclass of an ABC even without registration. Here is his example again, with an added test using `issubclass`:

```
>>> class Struggle:
...     def __len__(self): return 23
...
>>> from collections import abc
>>> isinstance(Struggle(), abc.Sized)
True
>>> issubclass(Struggle, abc.Sized)
True
```

Class `Struggle` is considered a subclass of `abc.Sized` by the `issubclass` function (and, consequently, by `isinstance` as well) because `abc.Sized` implements a special class method named `__subclasshook__`. See Example 11-17.

Example 11-17. `Sized` definition from the source code of `Lib/_collections_abc.py` (Python 3.4).

```
class Sized(metaclass=ABCMeta):

    __slots__ = ()

    @abstractmethod
    def __len__(self):
        return 0

    @classmethod
    def __subclasshook__(cls, C):
        if cls is Sized:
            if any("__len__" in B.__dict__ for B in C.__mro__):  # ❶
                return True  # ❷
        return NotImplemented  # ❸
```

❶ If there is an attribute named `__len__` in the `__dict__` of any class listed in `C.__mro__` (i.e. `C` and its superclasses)…

❷ …return `True`, signaling that `C` is a virtual subclass of `Sized`.

❸ Otherwise return `NotImplemented` to let the subclass check proceed.

If you are interested in the details of the subclass check, see the source code for the `ABCMeta.__subclasscheck__` method in Lib/abc.py. Beware: it has lots of ifs and two recursive calls.

The `__subclasshook__` adds some duck typing DNA to the whole goose typing proposition. You can have formal interface definitions with ABCs, you can make `isin` `stance` checks everywhere, and still have a completely unrelated class play along just because it implements a certain method (or because it does whatever it takes to convince a `__subclasshook__` to vouch for it). Of course, this only works for ABCs that do provide a `__subclasshook__`.

Is it a good idea to implement `__subclasshook__` in our own ABCs? Probably not. All the implementations of `__subclasshook__` I've seen in the Python source code are in ABCs like `Sized` that declare just one special method, and they simply check for that special method name. Given their "special" status, you can be pretty sure that any method named `__len__` does what you expect. But even in the realm of special methods and fundamental ABCs, it can be risky to make such assumptions. For example, mappings implement `__len__`, `__getitem__` and `__iter__` but they are rightly not considered a subtype of `Sequence`, because you can't retrieve items using an integer offset and they make no guarantees about the ordering of items — except of course for `OrderedDict`, which preserves the insertion order, but does support item retrieval by offset either.

For ABCs that you and I may write, a `__subclasshook__` would be even less dependable. I am not ready to believe that any class `Spam` that implements or inherits `load`, `pick`, `inspect` and `loaded` is guaranteed to behave as a `Tombola`. It's better to let the programmer affirm it by subclassing `Spam` from `Tombola`, or at least registering: `Tombola.register(Spam)`. Of course, your `__subclasshook__` could also check method signatures and other features, but I just don't think its worthwhile.

Chapter summary

The goal of this chapter was to travel from the highly dynamic nature of informal interfaces — called protocols — visit the static interface declarations of ABCs, and conclude with the dynamic side of ABCs: virtual subclasses and dynamic subclass detection with `__subclasshook__`.

We started the journey reviewing the traditional understanding of interfaces in the Python community. For most of the history of Python, we've been mindful of interfaces, but they were informal like the protocols from Smalltalk, and the official docs used language such as "foo protocol", "foo interface", and "foo-like object" interchangeably. Protocol-style interfaces have nothing to do with inheritance; each class stands alone when implementing a protocol. That's how interfaces look like when you embrace duck typing.

With Example 11-3 we observed how deeply Python supports the sequence protocol. If a class implements `__getitem__` and nothing else, Python manages to iterate over it, and the `in` operator just works. We then went back to the old `FrenchDeck` example of Chapter 1 to support shuffling by dynamically adding a method. This illustrated monkey patching and emphasized the dynamic nature of protocols. Again we saw how a partially implemented protocol can be useful: just adding `__setitem__` from the mutable sequence protocol allowed us to leverage a ready-to-use function from the standard library: `random.shuffle`. Being aware of existing protocols let's us make the most of the rich Python Standard library.

Alex Martelli then introduced the term "goose typing[16]" to describe a new style of Python programming. With "goose typing", ABCs are used to make interfaces explicit and classes may claim to implement an interface by subclassing an ABC or by registering with it — without requiring the strong and static link of an inheritance relationship.

The `FrenchDeck2` example made clear the main drawbacks and advantages of explicit ABCs. Inheriting from `abc.MutableSequence` forced us to implement two methods we did not really need: `insert` and `__delitem__`. On the other hand, even a Python newbie can look at a `FrenchDeck2` and see that it's a mutable sequence. And, as bonus, we inherited eleven ready-to-use methods from `abc.MutableSequence` (five indirectly from `abc.Sequence`).

After a panoramic view of existing ABCs from `collections.abc` in Figure 11-3, we wrote an ABC from scratch. Doug Hellmann, creator of the cool PyMOTW.com (Python Module of the Week) explains the motivation:

> By defining an abstract base class, a common API can be established for a set of subclasses. This capability is especially useful in situations where someone less familiar with the source for an application is going to provide plug-in extensions…[17]

Putting the `Tombola` ABC to work, we created three concrete subclasses: two inheriting from `Tombola`, the other a virtual subclass registered with it. All passing the same suite of tests.

To close the chapter, we mentioned how several built-in types are registered to ABCs in the `collections.abc` module so you can ask `isinstance(memoryview, abc.Sequence)` and get `True`, even if `memoryview` does not inherit from `abc.Sequence`. And finally we went over the `__subclasshook__` magic which lets an ABC recognize any unregistered class as a subclass, as long as it passes a test that can be as simple or as complex as you like — the examples in the Standard Library merely check for method names.

In conclusion, I'd like to restate Alex Martelli's admonition that we should refrain from creating our own ABCs, except when we are building user-extensible frameworks — which most of the time we are not. On a daily basis, our contact with ABCs should be subclassing or registering classes with existing ABCs. Less often than subclassing or

[16] Alex coined the expression "goose typing" and this is the first time ever it appears in a book!

[17] PyMOTW, `abc` module page, section Why use Abstract Base Classes?

registering, we might use ABCs for `isinstance` checks. And even more rarely — if ever — we find occasion to write a new ABC from scratch.

After 15 years of Python, the first abstract class I ever wrote that is not a didactic example was the `Board` class of the Pingo project. The drivers that support different single board computers and controllers are subclasses of `Board`, thus sharing the same interface. In reality, although conceived and implemented as an abstract class, the `pingo.Board` class does not subclass `abc.ABC` as I write this[18]. I intend to make `Board` an explicit ABC eventually — but there are more important things to do in the project.

Here is a fitting quote to end this chapter:

> Although ABCs facilitate type checking, it's not something that you should overuse in a program. At its heart, Python is a dynamic language that gives you great flexibility. Trying to enforce type constraints everywhere tends to result in code that is more complicated than it needs to be. You should embrace Python's flexibility[19].
>
> — David Beazley and Brian Jones
> *Python Cookbook 3ed.*

Or, as technical reviewer Leonardo Rochael wrote: "If you feel tempted to create a custom ABC, please first try to solve your problem through regular duck-typing".

Further reading

Beazley & Jones' *Python Cookbook, 3e* (O'Reilly, 2013) has a section about defining an ABC (recipe 8.12). The book was written before Python 3.4, so they don't use the now preferred syntax when declaring ABCs by subclassing from `abc.ABC` instead of using the `metaclass` keyword. Apart from this small detail, the recipe covers the major ABC features very well, ends with the valuable advice quoted at the end of the previous section.

The Python Standard Library by Example (Addison-Wesley, 2011), by Doug Hellmann, has a chapter about the `abc` module. It's also available on the web in Doug's excellent PyMOTW — Python Module of the Week. Both the book and the site focus on Python 2, therefore adjustments must be made if you are using Python 3. And for Python 3.4, remember that the only recommended ABC method decorator is `@abtractmethod` — the others were deprecated. The other quote about ABCs in the chapter summary is from Doug's site and book.

When using ABCs, multiple inheritance is not only common but practically inevitable, because each of the fundamental collection ABCs — `Sequence`, `Mapping` and `Set` —

[18] You'll find that in the Python standard library too: classes that are in fact abstract but nobody ever made them explicitly so.

[19] *Python Cookbook, 3e* (O'Reilly, 2013), *Recipe 8.12. Defining an Interface or Abstract Base Class*, p. 276.

extends multiple ABCs (see Figure 11-3). Therefore, Chapter 12 is an important follow-up to this one.

PEP 3119 — Introducing Abstract Base Classes gives the rationale for ABCs, and PEP 3141 - A Type Hierarchy for Numbers presents the ABCs of the `numbers` module.

For a discussion of the pros and cons of dynamic typing, see Guido van Rossum's interview to Bill Venners in Contracts in Python: A Conversation with Guido van Rossum, Part IV.

The `zope.interface` package provides a way of declaring interfaces, checking whether objects implement them, registering providers and querying for providers of a given interface. The package started as a core piece of Zope 3, but it can and has been used outside of Zope. It is the basis of the flexible component architecture of large scale Python projects like Twisted, Pyramid and Plone. Lennart Regebro has an great introduction to `zope.interface` in A Python component architecture. Baiju M wrote an entire book about it: A Comprehensive Guide to Zope Component Architecture.

Soapbox

Type hints

Probably the biggest news in the Python world in 2014 was that Guido van Rossum gave a green light to the implementation of optional static type checking using function annotations, similar to what the Mypy checker does. This happened in the Python-ideas mailing-list on August 15. The message is Optional static typing — the crossroads. The next month, PEP 484 - Type Hints was published as a draft, authored by Guido.

The idea is to let programmers optionally use annotations to declare parameter and return types in function definitions. The key word here is "optionally". You'd only add such annotations if you want the benefits and constraints that come with them, and you could put them in some functions but not in others.

On the surface this may sound like what Microsoft did with with TypeScript, their JavaScript superset, except that TypeScript goes much further: it adds new language constructs (e.g. modules, classes, explicit interfaces etc.), allows typed variable declarations and actually compiles down to plain JavaScript. As of this writing (September, 2014), the goals of optional static typing in Python are much less ambitious.

To understand the reach of this proposal, there is a key point that Guido makes in the historic August 15, 2014, e-mail:

> I am going to make one additional assumption: the main use cases will be linting, IDEs, and doc generation. These all have one thing in common: it should be possible to run a program even though it fails to type check. Also, adding types to a program should not hinder its performance (nor will it help :-).

So, it seems this is not such a radical move as it seems at first. PEP 482 - Literature Overview for Type Hints is referenced by PEP 484 - Type Hints, and briefly documents type hints in third-party Python tools and in other languages.

Radical or not, type hints are upon us: support for PEP 484 in the form of a `typing` module is likely to land in Python 3.5 already. The way the proposal is worded and implemented makes it clear that no existing code will stop running because of the lack of type hints — or their addition, for that matter.

Finally, PEP 484 clearly states:

> It should also be emphasized that Python will remain a dynamically typed language, and the authors have no desire to ever make type hints mandatory, even by convention.

Is Python weakly typed?

Discussions about language typing disciplines are sometimes confused due to lack of a uniform terminology. Some writers (like Bill Venners in the interview with Guido mentioned in **Further Reading**), say that Python has weak typing, which puts it into the same category of JavaScript and PHP. A better way of talking about typing discipline is to consider two different axes:

Strong versus weak typing

If the language rarely performs implicit conversion of types, it's considered strongly typed; if it often does it, it's weakly typed. Java, C++ and Python are strongly typed. PHP, JavaScript and Perl are weakly typed.

Static versus dynamic typing

If type-checking is performed at compile time, the language is statically typed; it it happens at run-time, it's dynamically typed. Static typing requires type declarations (some modern languages use type inference to avoid some of that). Fortran and Lisp are the two oldest programming languages still alive and they use, respectively, static and dynamic typing.

Strong typing helps catch bugs early. Below are some examples of why weak typing is bad[20].

```
// this is JavaScript (tested with Node.js v0.10.33)
'' == '0'   // false
0 == ''     // true
0 == '0'    // true
'' < 0      // false
'' < '0'    // true
```

Python does not perform automatic coercion between strings and numbers, so the `==` expressions above all result `False` — preserving the the transitivity of `==` — and the `<` comparisons raise `TypeError` in Python 3.

[20] Adapted from Douglas Crockford's *JavaScript: The Good Parts* (O'Reilly, 2008), Appendix B, p. 109

Static typing makes it easier for tools (compilers, IDEs) to analyze code to detect errors and provide other services (optimization, refactoring etc.). Dynamic typing increases opportunities for reuse, reducing line count, and allows interfaces to emerge naturally as protocols, instead of being imposed early on.

To summarize, Python uses dynamic and strong typing. PEP 484 - Type Hints will not change that, but will allow API authors to add optional type annotations so that tools can perform some static type checking.

Monkey patching

Monkey patching has a bad reputation. If abused, it can lead to systems that are hard to understand and maintain. The patch is usually tightly coupled with its target, making it brittle. Another problem is that two libraries that apply monkey-patches may step on each other's toes, with the second library to run destroying patches of the first.

But monkey patching can also be useful, for example, to make a class implement a protocol at runtime. The adapter design pattern solves the same problem by implementing a whole new class.

It's easy to monkey patch Python code, but there are limitations. Unlike Ruby and JavaScript, Python does not let you monkey patch the built-in types. I actually consider this an advantage, since you can be certain that a `str` object will always have those same methods. This limitation reduces the chance that external libraries try to apply conflicting patches.

Interfaces in Java, Go and Ruby

Since C++ 2.0 (1989), abstract classes have been used to specify interfaces that language. The designers of Java opted not to have multiple inheritance of classes, which precluded the use of abstract classes as interface specifications — because often a class needs to implement more than one interface. But they added the `interface` as a language construct, and a class can implement more than one interface — a form of multiple inheritance. Making interface definitions more explicit than ever was a great contribution of Java. With Java 8, an interface can provide method implementations, called Default Methods. With this, Java interfaces became closer to abstract classes in C++ and Python.

The Go language has a completely different approach. First of all, there is no inheritance in Go. You can define interfaces, but you don't need (and you actually can't) explicitly say that a certain type implements an interface. The compiler determines that automatically. So what they have in Go could be called "static duck typing", in the sense that interfaces are checked at compile time but what matters is what types actually implement.

Compared to Python, it's as if, in Go, every ABC implemented the `__subclasshook__` checking function names and signatures, and you never subclassed or registered an ABC. If we wanted Python to look more like Go, we would have to perform type checks on all function arguments. Some of the infrastructure is available (recall "Function annotations" on page 154). Guido has already said he thinks it's OK to use those annotations for type checking — at least in support tools. See "Soapbox" on page 163 in Chapter 5 for more about this.

Rubyists are firm believers in duck typing, and Ruby has no formal way to declare an interface or an abstract class, except to do the same we did in Python prior to 2.6: `raise NotImplementedError` in the body of methods to make them abstract by forcing the user to subclass and implement them.

Meanwhile, I read that Yukihiro "Matz" Matsumoto, creator of Ruby, said in a keynote in September, 2014, that static typing may be in the future of the language. That was at Ruby Kaigi in Japan, one of the most important Ruby conferences every year. As I write this I haven't seen a transcript, but Godfrey Chan posted about it in his blog: Ruby Kaigi 2014: Day 2. From Chan's report, it seems Matz focused on function annotations. There is even mention of Python function annotations.

I wonder if function annotations would be really good without ABCs to add structure to the type system without losing flexibility. So maybe formal interfaces are also in the future of Ruby.

I believe Python ABCs, with the `register` function and `__subclasshook__`, brought formal interfaces to the language without throwing away the advantages of dynamic typing.

Perhaps the geese are poised to overtake the ducks.

Metaphors and idioms in interfaces

A metaphor fosters understanding by making constraints clear. That's the value of the words "stack" and "queue" in describing those fundamental data structures: they make clear how items can be added or removed. On the other hand, Alan Cooper writes in *About Face, 4e* (Wiley, 2014):

> Strict adherence to metaphors ties interfaces unnecessarily tightly to the workings of the physical world.

He's referring to user interfaces, but the admonition applies to APIs as well. But Cooper does grant that when an "truly appropriate" metaphor "falls on our lap", we can use it (he writes "falls on our lap" because it's so hard to find fitting metaphors that you should not spend time actively looking for them). I believe the bingo machine imagery I used in this chapter is appropriate and I stand by it.

About Face is by far the best book about UI design I've read — and I've read a few. Letting go of metaphors as a design paradigm, and replacing it with "idiomatic interfaces" was the most valuable thing I learned from Cooper's work. As mentioned, Cooper does not deal with APIs, but the more I think about his ideas, the more I see they apply to Python. The fundamental protocols of the language are what Cooper calls "idioms". Once we learn what a "sequence" is we can apply that knowledge in different contexts. This is a main theme of *Fluent Python*: highlighting the fundamental idioms of the language, so your code is concise, effective and readable — for a fluent Pythonista.

CHAPTER 12
Inheritance: for good or for worse

> [We] started to push on the inheritance idea as a way to let novices build on frameworks that could only be designed by experts[1].
>
> — Alan Kay
> *The Early History of Smalltalk*

This chapter is about inheritance and subclassing, with emphasis on two particulars that are very specific to Python:

- The pitfalls of subclassing from built-in types.
- Multiple inheritance and the method resolution order.

Many consider multiple inheritance more trouble than it's worth. The lack of it certainly did not hurt Java; it probably fueled its widespread adoption after many were traumatized by the excessive use of multiple inheritance in C++.

However, the amazing success and influence of Java means that a lot of programmers come to Python without having seen multiple inheritance in practice. This is why, instead of toy examples, our coverage of multiple inheritance will be illustrated by two important Python projects: the Tkinter GUI toolkit and the Django Web framework.

We'll start with the issue of subclassing built-ins. The rest of the chapter will cover multiple inheritance with our case studies and discuss good and bad practices when building class hierarchies.

[1] Alan Kay, *The early history of Smalltalk* in SIGPLAN Not. 28, 3 (March 1993), 69-95. Also available online at The Early History of Smalltalk. Thanks to my friend Christiano Anderson who shared this reference as I was writing this chapter.

Subclassing built-in types is tricky

Before Python 2.2 it was not possible to subclass built-in types such as `list` or `dict`. Since then, it can be done but there is a major caveat: the code of the built-ins (written in C) does not call special methods overridden by user-defined classes.

A good short description of the problem is in the documentation for *PyPy*, in *Differences between PyPy and CPython*, section Subclasses of built-in types:

> Officially, CPython has no rule at all for when exactly overridden method of subclasses of built-in types get implicitly called or not. As an approximation, these methods are never called by other built-in methods of the same object. For example, an overridden `__getitem__()` in a subclass of `dict` will not be called by e.g. the built-in `get()` method.

Example 12-1 is illustrates the problem.

Example 12-1. Our `__setitem__` override is ignored by the `__init__` and `__update__` methods of the built-in `dict`.

```
>>> class DoppelDict(dict):
...     def __setitem__(self, key, value):
...         super().__setitem__(key, [value] * 2)  # ❶
...
>>> dd = DoppelDict(one=1)  # ❷
>>> dd
{'one': 1}
>>> dd['two'] = 2  # ❸
>>> dd
{'one': 1, 'two': [2, 2]}
>>> dd.update(three=3)  # ❹
>>> dd
{'three': 3, 'one': 1, 'two': [2, 2]}
```

❶ `DoppelDict.__setitem__` duplicates values when storing (for no good reason, just to have a visible effect). It works by delegating to the superclass.

❷ The `__init__` method inherited from `dict` clearly ignored that `__setitem__` was overridden: the value of `'one'` is not duplicated.

❸ The `[]` operator calls our `__setitem__` and works as expected: `'two'` maps to the duplicated value `[2, 2]`.

❹ The `update` method from `dict` does not use our version of `__setitem__` either: the value of `'three'` was not duplicated.

This built-in behavior is a violation of a basic rule of object oriented programming: the search for methods should always start from the class of the target instance (`self`), even when the call happens inside a method implemented in a superclass. In this sad state of affairs, the `__missing__` method — which we saw in "The `__missing__` method" on page 72 — works as documented only because it's handled as a special case.

The problem is not limited to calls within an instance, i.e. whether `self.get()` calls `self.__getitem__()`, but also happens with overridden methods of other classes that should be called by the built-in methods. Here is an example adapted from the PyPy documentation:

Example 12-2. The `__getitem__` of `AnswerDict` is bypassed by `dict.update`.

```
>>> class AnswerDict(dict):
...     def __getitem__(self, key):  # ❶
...         return 42
...
>>> ad = AnswerDict(a='foo')  # ❷
>>> ad['a']  # ❸
42
>>> d = {}
>>> d.update(ad)  # ❹
>>> d['a']  # ❺
'foo'
>>> d
{'a': 'foo'}
```

❶ `AnswerDict.__getitem__` always returns `42`, no matter what the key.

❷ `ad` is an `AnswerDict` loaded with the key-value pair `('a', 'foo')`.

❸ `ad['a']` returns `42`, as expected.

❹ `d` is an instance of plain `dict`, which we update with `ad`.

❺ The `dict.update` method ignored our `AnswerDict.__getitem__`.

Subclassing built-in types like `dict` or `list` or `str` directly is error-prone because the built-in methods mostly ignore user-defined overrides. Instead of subclassing the built-ins, derive your classes from `UserDict`, `UserList` and `UserString` from the `collections` module, which are designed to be easily extended.

If you subclass `collections.UserDict` instead of `dict`, the issues exposed in Example 12-1 and Example 12-2 are both fixed. See Example 12-3.

Example 12-3. `DoppelDict2` and `AnswerDict2` work as expected because they extend `UserDict` and not `dict`.

```
>>> import collections
>>>
>>> class DoppelDict2(collections.UserDict):
...     def __setitem__(self, key, value):
```

```
...             super().__setitem__(key, [value] * 2)
...
>>> dd = DoppelDict2(one=1)
>>> dd
{'one': [1, 1]}
>>> dd['two'] = 2
>>> dd
{'two': [2, 2], 'one': [1, 1]}
>>> dd.update(three=3)
>>> dd
{'two': [2, 2], 'three': [3, 3], 'one': [1, 1]}
>>>
>>> class AnswerDict2(collections.UserDict):
...     def __getitem__(self, key):
...         return 42
...
>>> ad = AnswerDict2(a='foo')
>>> ad['a']
42
>>> d = {}
>>> d.update(ad)
>>> d['a']
42
>>> d
{'a': 42}
```

As an experiment to measure the extra work required to subclass a built-in, I rewrote the StrKeyDict class from Example 3-8. The original version inherited from collections.UserDict, and implemented just three methods: __missing__, __contains__ and __setitem__. The experimental StrKeyDict subclassed dict directly, and implemented the same three methods with minor tweaks due to the way the data was stored. But in order to make it pass the same suite of tests, I had to implement __init__, get and update because the versions inherited from dict refused to cooperate with with the overridden __missing__, __contains__ and __setitem__. The UserDict subclass from Example 3-8 has 16 lines, while the experimental dict subclass ended up with 37 lines[2].

To summarize: the problem described in this section applies only to method delegation within the C language implementation of the built-in types, and only affects user-defined classes derived directly from those types. If you subclass from a class coded in Python, such as UserDict or MutableMapping, you will not be troubled by this[3].

[2] If you are curious, the experiment is in file strkeydict_dictsub.py in the Fluent Python code repository.

[3] By the way, in this regard PyPy behaves more “correctly” than CPython, at the expense of introducing a minor incompatibility. See Differences between PyPy and CPython for details.

Another matter related to inheritance, particularly of multiple inheritance, is: how does Python decide which attribute to use if superclasses from parallel branches define attributes with the same name? The answer is next.

Multiple inheritance and method resolution order

Any language implementing multiple inheritance needs to deal with potential naming conflicts when unrelated ancestor classes implement a method by the same name. This is called the "diamond problem", and is illustrated in Figure 12-1 and Example 12-4.

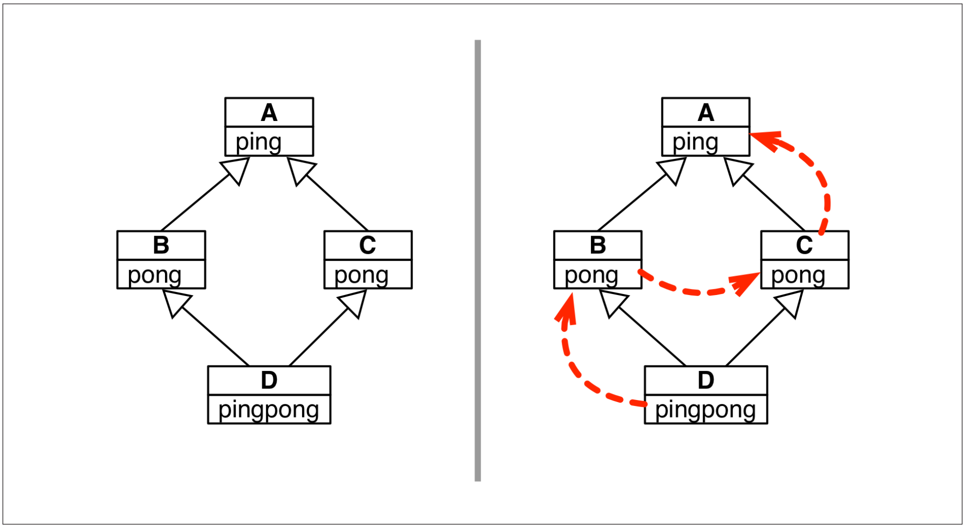


*Figure 12-1. **Left**: UML class diagram illustrating the "diamond problem". **Right**: Dashed arrows depict Python MRO (method resolution order) for Example 12-4.*

Example 12-4. `diamond.py`: classes `A`, `B`, `C` and `D` form the graph in Figure 12-1

```
class A:
    def ping(self):
        print('ping:', self)


class B(A):
    def pong(self):
        print('pong:', self)


class C(A):
    def pong(self):
        print('PONG:', self)
```

```
class D(B, C):

    def ping(self):
        super().ping()
        print('post-ping:', self)

    def pingpong(self):
        self.ping()
        super().ping()
        self.pong()
        super().pong()
        C.pong(self)
```

Note that both classes B and C implement a pong method. The only difference is that C.pong outputs the word PONG in upper case.

If you call d.pong() on an instance of D, which pong method actually runs? In C++ the programmer must qualify method calls with class names to resolve this ambiguity. This can be done in Python as well. Take a look at Example 12-5.

Example 12-5. Two ways of invoking method pong on an instance of class D.

```
>>> from diamond import *
>>> d = D()
>>> d.pong()  # ❶
pong: <diamond.D object at 0x10066c278>
>>> C.pong(d)  # ❷
PONG: <diamond.D object at 0x10066c278>
```

❶ Simply calling d.pong() causes the B version to run.

❷ You can always call a method on a superclass directly, passing the instance as an explicit argument.

The ambiguity of a call like d.pong() is resolved because Python follows a specific order when traversing the inheritance graph. That order is called MRO: Method Resolution Order. Classes have an attribute called __mro__ holding a tuple of references to the superclasses in MRO order, from the current class all the way to the object class. For the D class, this is the __mro__ (see Figure 12-1):

```
>>> D.__mro__
(<class 'diamond.D'>, <class 'diamond.B'>, <class 'diamond.C'>,
<class 'diamond.A'>, <class 'object'>)
```

The recommended way to delegate method calls to superclasses is the super() built-in function, which became easier to use in Python 3, as method pingpong of class D in

Example 12-4 illustrates[4]. However, it's also possible, and sometimes convenient, to bypass the MRO and invoke a method on a superclass directly. For example, the `D.ping` method could be written as:

```
    def ping(self):
        A.ping(self)  # instead of super().ping()
        print('post-ping:', self)
```

Note that when calling an instance method directly on a class, you must pass `self` explicitly, because you are accessing an *unbound method*.

However, it's safest and more future-proof to use `super()`, especially when calling methods on a framework, or any class hierarchies you do not control. Example 12-6 shows that `super()` follows the MRO when invoking a method.

Example 12-6. Using `super()` to call `ping` (source code in Example 12-4)

```
>>> from diamond import D
>>> d = D()
>>> d.ping()  # ❶
ping: <diamond.D object at 0x10cc40630>  # ❷
post-ping: <diamond.D object at 0x10cc40630>  # ❸
```

❶ The `ping` of `D` makes two calls:

❷ The first call is `super().ping()`; the `super` delegates the `ping` call to class `A`; `A.ping` outputs this line.

❸ The second call is `print('post-ping:', self)` which outputs this line.

Now let's see what happens when `pingpong` is called on an instance of `D`.

Example 12-7. The five calls made by `pingpong` (source code in Example 12-4)

```
>>> from diamond import D
>>> d = D()
>>> d.pingpong()
>>> d.pingpong()
ping: <diamond.D object at 0x10bf235c0>  # ❶
post-ping: <diamond.D object at 0x10bf235c0>
ping: <diamond.D object at 0x10bf235c0>  # ❷
pong: <diamond.D object at 0x10bf235c0>  # ❸
pong: <diamond.D object at 0x10bf235c0>  # ❹
PONG: <diamond.D object at 0x10bf235c0>  # ❺
```

❶ Call #1 is`self.ping()` runs the `ping` method of `D`, which outputs this line and the next one.

[4] In Python 2, the first line of `D.pingpong` would be written as `super(D, self).ping()` instead of `super().ping()`.

❷ Call #2 is super.ping() which bypasses the ping in D and finds the ping method in A.

❸ Call #3 is self.pong() which finds the B implementation of pong, according to the __mro__.

❹ Call #4 is super.pong() which finds the same B.pong implementation, also following the __mro__.

❺ Call #5 is C.pong(self) which finds the C.pong implementation, ignoring the __mro__.

The MRO takes into account not only the inheritance graph but also the order in which superclasses are listed in a subclass declaration. In other words, if in diamond.py (Example 12-4) the D class was declared as class D(C, B):, the __mro__ of class D would be different: C would be searched before B.

I often check the __mro__ of classes interactively when I am studying them. Example 12-8 has some examples using familiar classes.

Example 12-8. Inspecting the __mro__ attribute in several classes.

```
>>> bool.__mro__  ❶
(<class 'bool'>, <class 'int'>, <class 'object'>)
>>> def print_mro(cls):  ❷
...     print(', '.join(c.__name__ for c in cls.__mro__))
...
>>> print_mro(bool)
bool, int, object
>>> from frenchdeck2 import FrenchDeck2
>>> print_mro(FrenchDeck2)  ❸
FrenchDeck2, MutableSequence, Sequence, Sized, Iterable, Container, object
>>> import numbers
>>> print_mro(numbers.Integral)  ❹
Integral, Rational, Real, Complex, Number, object
>>> import io  ❺
>>> print_mro(io.BytesIO)
BytesIO, _BufferedIOBase, _IOBase, object
>>> print_mro(io.TextIOWrapper)
TextIOWrapper, _TextIOBase, _IOBase, object
```

❶ bool inherits methods and attributes from int and object.

❷ print_mro produces more compact displays of the MRO.

❸ The ancestors of FrenchDeck2 include several ABCs from the collections.abc module.

❹ These are the numeric ABCs provided by the numbers module.

❺ The `io` module includes ABCs (those with the `...Base` suffix) and concrete classes like `BytesIO` and `TextIOWrapper` which are the types of binary and text file objects returned by `open()`, depending on the mode argument.

The MRO is computed using an algorithm called C3. The canonical paper on the Python MRO explaining C3 is Michele Simionato's The Python 2.3 Method Resolution Order. If you are interested in the subtleties of the MRO, "Further reading" on page 369 has other pointers. But don't fret too much about this, the algorithm is sensible and Simionato wrote:

> […] unless you make strong use of multiple inheritance and you have non-trivial hierarchies, you don't need to understand the C3 algorithm, and you can easily skip this paper.

To wrap-up this discussion of the MRO, Figure 12-2 illustrates part of the complex multiple inheritance graph of the Tkinter GUI toolkit from the Python standard library. To study the picture, start at the `Text` class at the bottom. The `Text` class implements a full featured, multi-line editable text widget. It has rich functionality of its own, but also inherits many methods from other classes. The left side shows a plain UML class diagram. On the right, it's decorated with arrows showing the MRO, as listed here with the help of the the `print_mro` convenience function defined in Example 12-8:

```
>>> import tkinter
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object
```

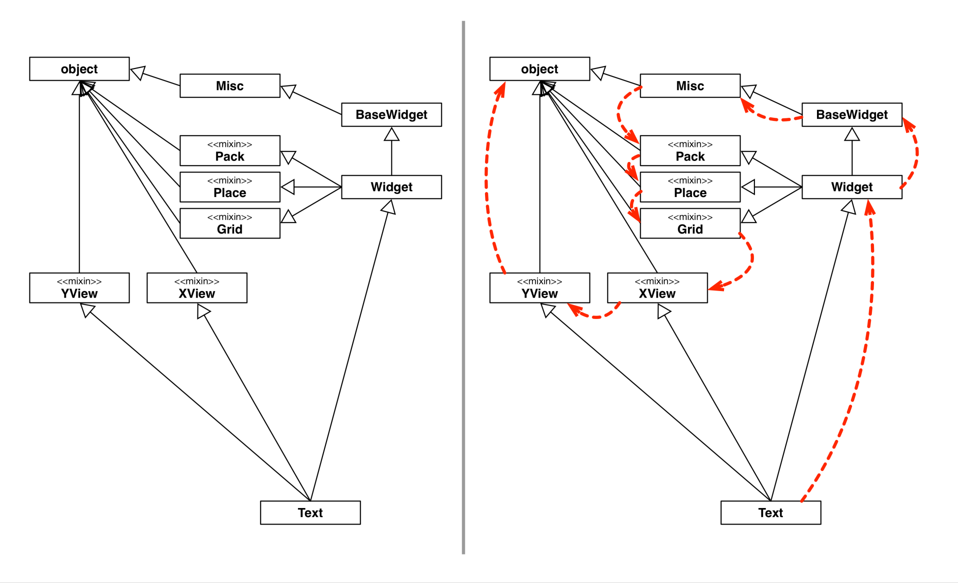


*Figure 12-2. **Left**: UML class diagram of the Tkinter `Text` widget class and its superclasses. **Right**: Dashed arrows depict `Text.__mro__`.*

In the next section we'll discuss the pros and cons of multiple inheritance, with examples from real frameworks that use it.

Multiple inheritance in the real world

It is possible to put multiple inheritance to good use. The Adapter pattern in the *Design Patterns* book uses multiple inheritance, so it can't be completely wrong to do it (the remaining 22 patterns in the book use single inheritance only, so multiple inheritance is clearly not a cure-all).

In the Python standard library, the most visible use of multiple inheritance is the `collections.abc` package. That is not controversial: after all, even Java supports multiple inheritance of interfaces, and ABCs are interface declarations which may optionally provide concrete method implementations[5].

An extreme example of multiple inheritance in the standard library is the Tkinter GUI toolkit (module `tkinter`: Python interface to Tcl/Tk). I used part of the Tkinter widget hierarchy to illustrate the MRO in Figure 12-2, but Figure 12-3 shows all the widget

[5] As previously mentioned, Java 8 allows interfaces to provide method implementations as well. The new feature is called Default Methods in the official Java Tutorial.

classes in the `tkinter` base package (there are more widgets in the `tkinter.ttk` subpackage).

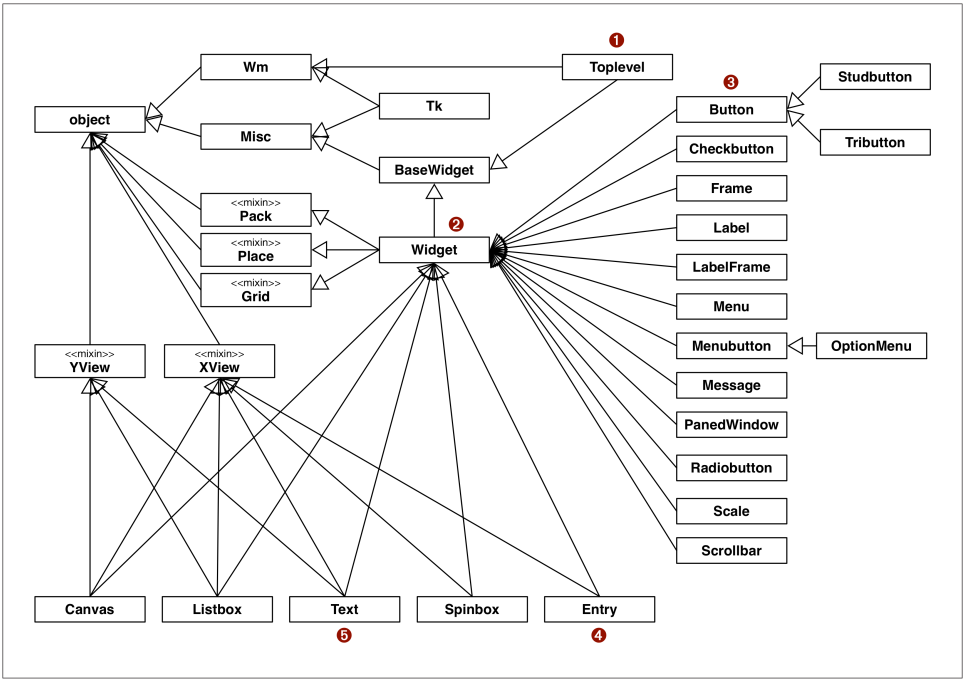


Figure 12-3. Summary UML diagram for the Tkinter GUI class hiearchy. Classes tagged «mixin» are designed to provide concrete methods to other classes via multiple inheritance.

Tkinter is 20 years old as I write this, and is not an example of current best practices. But it shows how multiple inheritance was used when coders did not appreciate its drawbacks. And it will serve as a counter-example when we cover some good practices in the next section.

Consider these classes from Figure 12-3:

❶ `Toplevel`: The class of a top-level window in a Tkinter application.

❷ `Widget`: The superclass of every visible object that can be placed on a window.

❸ `Button`: A plain button widget.

❹ `Entry`: A single-line editable text field.

❺ `Text`: A multi-line editable text field.

Here are the MROs of those classes, displayed by the `print_mro` function from Example 12-8.

```
>>> import tkinter
>>> print_mro(tkinter.Toplevel)
Toplevel, BaseWidget, Misc, Wm, object
>>> print_mro(tkinter.Widget)
Widget, BaseWidget, Misc, Pack, Place, Grid, object
>>> print_mro(tkinter.Button)
Button, Widget, BaseWidget, Misc, Pack, Place, Grid, object
>>> print_mro(tkinter.Entry)
Entry, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, object
>>> print_mro(tkinter.Text)
Text, Widget, BaseWidget, Misc, Pack, Place, Grid, XView, YView, object
```

Things to note about how these classes relate to others:

- `Toplevel` is the only graphical class that does not inherit from `Widget`, because it is the top-level window and does not behave like a widget — for example, it cannot be attached to a window or frame. `Toplevel` inherits from `Wm`, which provides direct access functions of the host window manager, like setting the window title and configuring its borders.
- `Widget` inherits directly from `BaseWidget` and from `Pack`, `Place` and `Grid`. These last three classes are geometry managers: they are responsible for arranging widgets inside a window or frame. Each encapsulates a different layout strategy and widget placement API.
- `Button`, like most widgets, descends only from `Widget`, but indirectly from `Misc`, which provides dozens of methods to every widget.
- `Entry` subclasses `Widget` and `XView`, the class that implements horizontal scrolling.
- `Text` subclasses from `Widget`, `XView` and `YView`, which provides vertical scrolling functionality.

We'll now discuss some good practices of multiple inheritance and see whether Tkinter goes along with them.

Coping with multiple inheritance

> […] we needed a better theory about inheritance entirely (and still do). For example, inheritance and instancing (which is a kind of inheritance) muddles both pragmatics (such as factoring code to save space) and semantics (used for way too many tasks such as: specialization, generalization, speciation, etc.).
>
> — Alan Kay
> *The Early History of Smalltalk*

As Alan Kay wrote, inheritance is used for different reasons, and multiple inheritance adds alternatives and complexity. It's easy to create incomprehensible and brittle designs using multiple inheritance. Since we don't have a comprehensive theory, here are a few tips to avoid spaghetti class graphs.

1. Distinguish interface inheritance from implementation inheritance

When dealing with multiple inheritance it's useful to keep straight the reasons why subclassing is done in the first place. The main reasons are:

- Inheritance of interface: creates a sub-type, implying an "is-a" relationship.
- Inheritance of implementation: avoids code duplication by reuse.

In practice both uses are often simultaneous, but whenever you can make the intent clear, do it. Inheritance for code reuse is an implementation detail, and it can often be replaced by composition and delegation. On the other hand, interface inheritance is the backbone of a framework.

2. Make interfaces explicit with ABCs

In modern Python, if a class is designed to define an interface, it should be an explicit ABC. In Python ≥ 3.4 this means: subclass `abc.ABC` or another ABC (see "ABC syntax details" on page 330 if you need to support older Python versions).

3. Use mixins for code reuse

If a class is designed to provide method implementations for reuse by multiple unrelated subclasses, without implying an "is-a" relationship, it should be an explicit *mixin class*. Conceptually, a mixin does not define a new type, it merely bundles methods for reuse. A mixin should never be instantiated, and concrete classes should not inherit only from a mixin. Each mixin should provide a single specific behavior, implementing few and very closely related methods.

4. Make mixins explicit by naming

There is no formal way in Python to state that a class is a mixin, so it is highly recommended that they are named with a …`Mixin` suffix. Tkinter does not follow this advice, but if it did, `XView`, would be `XViewMixin`, `Pack` would be `PackMixin` and so on with all the classes where I put the «mixin» tag Figure 12-3.

5. An ABC may also be a mixin; the reverse is not true

Since an ABC can implement concrete methods, it works as a mixin as well. An ABC also defines a type, which a mixin does not. And an ABC can be the sole base class of any another class, while a mixin should never be subclassed alone except by another, more specialized mixin — not a common arrangement in real code.

One restriction applies to ABCs and not to mixins: the concrete methods implemented in an ABC should only collaborate with methods of the same ABC and its superclasses. This implies that concrete methods in an ABC are always for convenience, because everything they do an user of the class can also do by calling other methods of the ABC.

6. Don't subclass from more than one concrete class

Concrete classes should have zero or at most one concrete superclass[6]. In other words, all but one of the superclasses of a concrete class should be ABCs or mixins. For example, in the code below, if `Alpha` is a concrete class, then `Beta` and `Gamma` must be ABCs or mixins:

```
class MyConcreteClass(Alpha, Beta, Gamma):
    """This is a concrete class: it can be instantiated."""
    # ... more code ...
```

7. Provide aggregate classes to users

If some combination of ABCs or mixins is particularly useful to client code, provide a class that brings them together in a sensible way. Grady Booch calls this an *aggregate class*.[7]

For example, here is the complete source code for `tkinter.Widget`:

```
class Widget(BaseWidget, Pack, Place, Grid):
    """Internal class.

    Base class for a widget which can be positioned with the
    geometry managers Pack, Place or Grid."""
    pass
```

The body of `Widget` is empty, but the class provides a useful service: it brings together four superclasses so that anyone who needs to create a new widget does not need remember all those mixins, or wonder if they need to be declared in a certain order in a `class` statement. A better example of this is the Django `ListView` class, which we'll discuss shortly, in "A modern example: mixins in Django generic views" on page 364.

[6] In "Waterfowl and ABCs" on page 316, Alex Martelli quotes Scott Meyer's *More Effective C++* which goes even further: "all non-leaf classes should be abstract" i.e. concrete classes should not have concrete superclasses at all.

[7] "A class that is constructed primarily by inheriting from mixins and does not add its own structure or behavior is called an *aggregate class*.", Grady Booch et.al. — *Object Oriented Analysis and Design, 3e* (Addison-Wesley, 2007), p. 109.

8. “Favor object composition over class inheritance.”

This quote comes straight the *Design Patterns* book[8], and is the best advice I can offer here. Once you get comfortable with inheritance, it’s too easy to overuse it. Placing objects in a neat hierarchy appeals to our sense of order; programmers do it just for fun.

However, favoring composition leads to more flexible designs. For example, in the case of the `tkinter.Widget` class, instead of inheriting the methods from all geometry managers, widget instances could hold a reference to a geometry manager, and invoke its methods. After all, a `Widget` should not “be” a geometry manager, but could use the services of one via delegation. Then you could add a new geometry manager without touching the widget class hierarchy and without worrying about name clashes. Even with single inheritance, this principle enhances flexibility, because subclassing is a form of tight coupling, and tall inheritance trees tend to be brittle.

Composition and delegation can replace the use of mixins to make behaviors available to different classes, but cannot replace the use of interface inheritance to define a hierarchy of types.

We will now analyze Tkinter from the point of view of these recommendations.

Tkinter: the good, the bad and the ugly

> Keep in mind that Tkinter has been part of the Standard Library since Python 1.1 was released in 1994. Tkinter is a layer on top of the excellent Tk GUI toolkit of the Tcl language. The Tcl/Tk combo is not originally object oriented, so the Tk API is basically a vast catalog of functions. However, the toolkit is very object oriented in its concepts, if not in its implementation.

Most advice in the previous section is not followed by Tkinter, with #7 being a notable exception. Even then, it’s not a great example, because composition would probably work better for integrating the geometry managers into `Widget`, as discussed in #8.

The docstring of `tkinter.Widget` starts with the words “Internal class.” This suggests that `Widget` should probably be an ABC. Although it has no methods of its own, `Widget` does define an interface. Its message is: “You can count on every Tkinter widget providing basic widget methods (`__init__`, `destroy`, and dozens of Tk API functions), in addition to the methods of all three geometry managers.” We can agree that this is not a great interface definition (it’s just too broad), but it is an interface, and `Widget` “defines” it as the union of the interfaces of its superclasses.

[8] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, *Introduction*, p. 20.

The `Tk` class, which encapsulates the GUI application logic, inherits from `Wm` and `Misc`, neither of which are abstract or mixin (`Wm` is not proper mixin because `TopLevel` subclasses only from it). The name of the `Misc` class is — by itself — a very strong *code smell*. `Misc` has more than 100 methods, and all widgets inherit from it. Why is it necessary that every single widget has methods for clipboard handling, text selection, timer management etc.? You can't really paste into a button or select text from a scrollbar. `Misc` should be split into several specialized mixin classes, and not all widgets should inherit from every one of those mixins.

To be fair, as a Tkinter user you don't need to know or use multiple inheritance at all. It's an implementation detail hidden behind the widget classes that you will instantiate or subclass in your own code. But you will suffer the consequences of excessive multiple inheritance when you type `dir(tkinter.Button)` and try to find the method you need among the 214 attributes listed.

Despite the problems, Tkinter is stable, flexible and not necessarily ugly. The legacy (and default) Tk widgets are not themed to match modern user interfaces, but the `tkinter.ttk` package provides pretty, native-looking widgets, making professional GUI development viable since Python 3.1 (2009). Also, some of the legacy widgets, like `Canvas` and `Text` are incredibly powerful. With just a little coding you can turn a `Canvas` object into a simple drag & drop drawing application. Tkinter and Tcl/Tk are definitely worth a look if you are interested in GUI programming.

However, our theme here is not GUI programming, but the practice of multiple inheritance. A more up-to-date example with explicit mixin classes can be found in Django.

A modern example: mixins in Django generic views

> You don't need to know Django to follow this section. I am just using a small part of the framework as a practical example of multiple inheritance, and I will try to give all the necessary background, assuming you have some experience with server-side Web development in any language or framework.

In Django, a view is a callable object that takes, as argument, an object representing an HTTP request and returns an object representing an HTTP response. The different responses are what interests us in this discussion. They can be as simple as a redirect response, with no content body, or as complex as a catalog page in an online store, rendered from an HTML template and listing multiple merchandise with buttons for buying and links to detail pages.

Originally, Django provided a set of functions, called generic views, that implemented some common use cases. For example, many sites need to show search results that include information from numerous items, with the listing spanning multiple pages, and for each item a link to a page with detailed information about it. In Django, a list view and a detail view are designed work together to solve this problem: a list view renders search results, and a detail view produces pages for individual items.

However, the original generic views were functions, so they were not extensible. If you needed to do something similar but not exactly like a generic list view, you'd have to start from scratch.

In Django 1.3, the concept of class-based views was introduced, along with a set of generic view classes organized as base classes, mixins and ready to use concrete classes. The base classes and mixins are in the `base` module of the `django.views.generic` package, pictured in Figure 12-4. At the top of the diagram we see two classes that take care of very distinct responsibilities: `View` and `TemplateResponseMixin`.

A great resource to study these classes is the Classy Class-Based Views Web site, where you can easily navigate through them, see all methods in each class (inherited, overridden and added methods), view diagrams, browse their documentation and jump to their source code on GitHub.

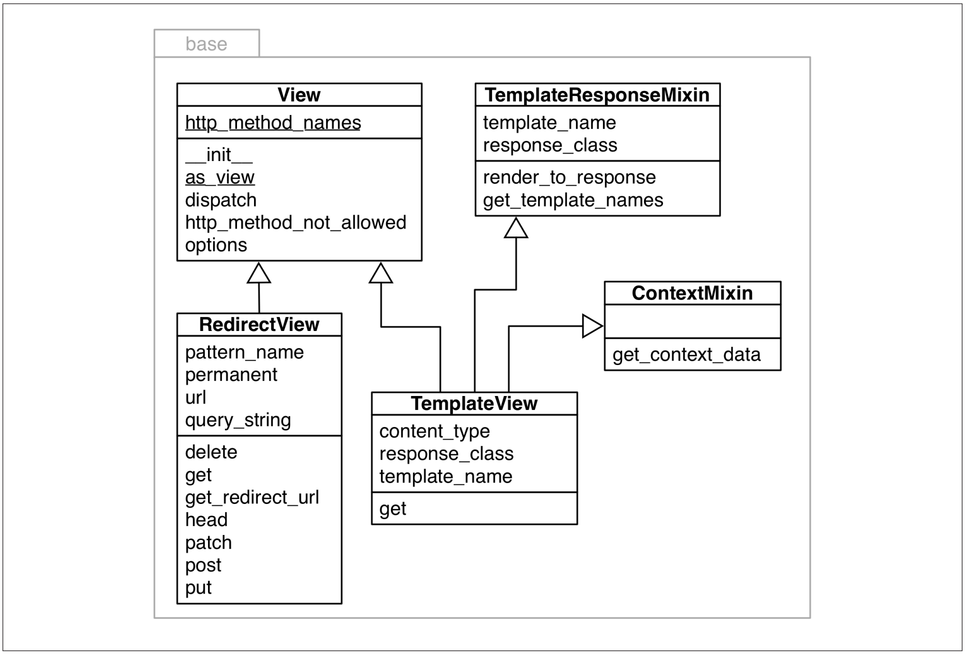


Figure 12-4. UML class diagram for the `django.views.generic.base` module.

View is the base class of all views (it could be an ABC), and it provides core functionality like the dispatch method, which delegates to “handler” methods like get, head, post etc., implemented by concrete subclasses to handle the different HTTP verbs[9]. The RedirectView class inherits only from View, and you can see that it implements get, head, post etc.

Concrete subclasses of View are supposed to implement the handler methods, so why aren’t they part of the View interface? The reason: subclasses are free to implement just the handlers they want to support. A TemplateView is used only to display content, so it only implements get. If an HTTP POST request is sent to a TemplateView, the inherited View.dispatch method checks that there is no post handler, and produces an HTTP 405 Method Not Allowed response[10].

The TemplateResponseMixin provides functionality that is of interest only to views that need to use a template. A RedirectView, for example, has no content body, so it has no need of a template and it does not inherit from this mixin. TemplateResponseMixin provides behaviors to TemplateView and other template-rendering views, such as ListView, DetailView etc., defined in other modules of the django.views.generic package. Figure 12-5 depicts the django.views.generic.list module and part of the base module.

[9] Django programmers know that the as_view class method is the most visible part of the View interface, but it’s not relevant to us here.

[10] If you are into design patterns, you’ll notice that the Django dispatch mechanism is a dynamic variation of the Template Method pattern. It’s dynamic because the View class does not force subclasses to implement all handlers, but dispatch checks at run time if a concrete handler is a available for the specific request.

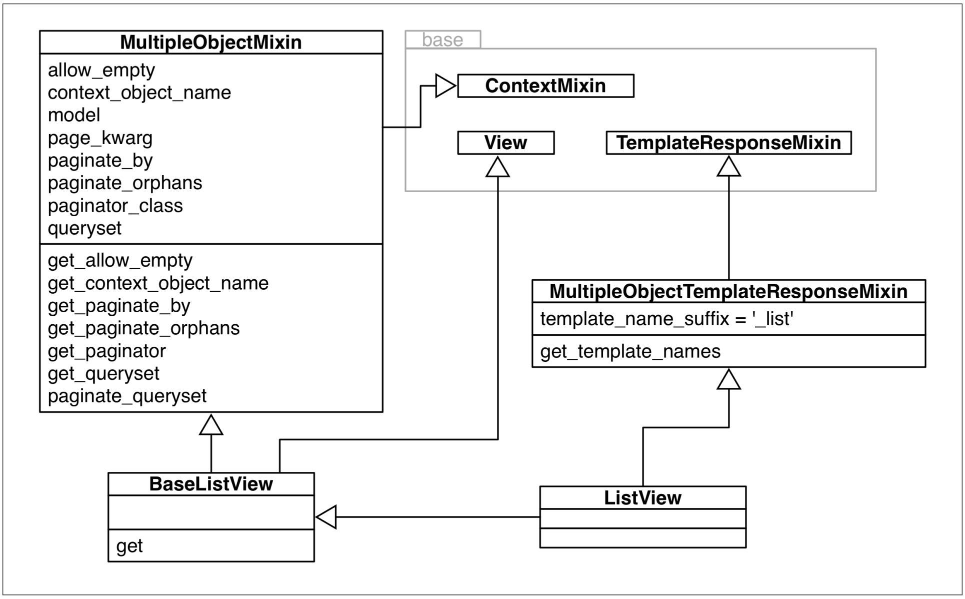


Figure 12-5. UML class diagram for the `django.views.generic.list` module. Here the three classes of the `base` module are collapsed (see Figure 12-4). The `ListView` class has no methods or attributes: it's an aggregate class.

For Django users, the most important class in Figure 12-5 is `ListView` which is an aggregate class, with no code at all (its body is just a docstring). When instantiated, a `ListView` has an `object_list` instance attribute through which the template can iterate to show the page contents, usually the result of a database query returning multiple objects. All the functionality related to generating this iterable of objects comes from the `MultipleObjectMixin`. That mixin also provides the complex pagination logic — to display part of the results in one page and links to more pages.

Suppose you want to create a view that will not render a template, but will produce a list of objects in JSON format. Thats' why the `BaseListView` exists. It provides an easy to use extension point that brings together `View` and `MultipleObjectMixin` functionality, without the overhead of the template machinery.

The Django Class-based views API is a better example of multiple inheritance than Tkinter. In particular, it is easy to make sense of its mixin classes: each has a well defined purpose, and they are all named with the …`Mixin` suffix.

Class-based views were not universally embraced by Django users. Many do use them in a limited way, as black boxes, but when it's necessary to create something new, a lot of Django coders continue writing monolithic view functions that take care of all those responsibilities, instead of trying to reuse the base views and mixins.

It does take some time to learn how to leverage class based views and how to extend them to fulfill specific application needs, but I found that it was worthwhile to study them: they eliminate a lot of boilerplate code, make it easier to reuse solutions and even improve team communication — for example, by defining standard names to templates, and to the variables passed to template contexts. Class-based views are Django views "on rails".

This concludes our tour of multiple inheritance and mixin classes.

Chapter summary

We started our coverage of inheritance explaining the problem with subclassing built-in types: their native methods implemented in C do not call overridden methods in subclasses, except in very few special cases. That's why, when we need a custom `list`, `dict` or `str` type, it's easier to subclass `UserList`, `UserDict` or `UserString` — all defined in the `collections` module, which actually wrap the built-in types and delegate operations to them — three examples of favoring composition over inheritance in the Standard Library. If the desired behavior is very different from what the built-ins offer, it may be easier to subclass the appropriate ABC from `collections.abc` and write your own implementation.

The rest of the chapter was devoted to the double-edged sword of multiple inheritance. First we saw the method resolution order, encoded in the `__mro__` class attribute, addresses the problem of potential naming conflicts in inherited methods. We also saw how the `super()` built-in follows the `__mro__` to call a method on a superclass. We then studied how multiple inheritance is used in the Tkinter GUI toolkit that comes with the Python Standard Library. Tkinter is not an example of current best practices, so we discussed some ways of coping with multiple inheritance, including careful use of mixin classes and avoiding multiple inheritance altogether by using composition instead. After considering how multiple inheritance is abused in Tkinter, we wrapped up by studying the core parts of the Django class-based views hierarchy, which I consider a better example of mixin usage.

Lennart Regebro — a very experienced Pythonista and one of this book's technical reviewers — finds the design of Django's mixin views hierarchy confusing. But he also wrote:

> The dangers and badness of multiple inheritance are greatly overblown. I've actually never head a real big problem with it.

In the end, each of us may have different opinions about how to use multiple inheritance, or whether to use it at all in our own projects. But often we don't have a choice: the frameworks we must use impose their own choices.

Further reading

When using ABCs, multiple inheritance is not only common but practically inevitable, because each of the most fundamental collection ABCs — `Sequence`, `Mapping` and `Set` — extend multiple ABCs. The source code for `collections.abc` (`Lib/_collections_abc.py`) is a good example of multiple inheritance with ABCs — many of which are also mixin classes.

Raymond Hettinger's post Python's super() considered super! explains the workings of `super` and multiple inheritance in Python from a positive perspective. It was written in response to Python's Super is nifty, but you can't use it (a.k.a. Python's Super Considered Harmful) by James Knight.

Despite the titles of those posts, the problem is not really the `super` built-in — which in Python 3 is not as ugly as it was in Python 2. The real issue is multiple inheritance, which is inherently complicated and tricky. Michele Simionato goes beyond criticizing and actually offers a solution in his Setting Multiple Inheritance Straight: he implements traits, a constrained form of mixins that originated in the Self language. Simionato has a long series of illuminating blog posts about multiple inheritance in Python, including The wonders of cooperative inheritance, or using super in Python 3, Mixins considered harmful, part 1 and part 2; and Things to Know About Python Super, part 1, part 2 and part 3. The oldest posts use the Python 2 `super` syntax, but are still relevant.

I read the first edition of Grady Booch's *Object Oriented Analysis and Design, 3e* (Addison-Wesley, 2007), and highly recommend it as a general primer on object oriented thinking, independent of programming language. It is a rare book that covers multiple inheritance without prejudice.

Soapbox

Think about the classes you really need

The vast majority of programmers write applications, not frameworks. Even those who do write frameworks are likely to spend a lot (if not most) of their time writing applications. When we write applications, we normally don't need to code class hierarchies. At most, we write classes that subclass from ABCs or other classes provided by the framework. As application developers, it's very rare that we need to write a class that will act as the superclass of another. The classes we code are almost always leaf classes (i.e. leaves of the inheritance tree).

If, while working as an application developer, you find yourself building multi-level class hierarchies, it's likely that one or more of the following applies:

- You are reinventing the wheel. Go look for a framework or library that provides components you can reuse in your application.

- You are using a badly designed framework. Go look for an alternative.
- You are over-engineering. Remember the *KISS principle*.
- You became bored coding applications and decided to start a new framework. Congratulations and good luck!

It's also possible that all of the above apply to your situation: you became bored and decided to reinvent the wheel by building your own over-engineered and badly designed framework which is forcing you to code class after class to solve trivial problems. Hopefully you are having fun, or at least getting paid for it.

Misbehaving built-ins: bug or feature?

The built-in `dict`, `list` and `str` types are essential building blocks of Python itself, so they must be fast — any performance issues in them would severely impact pretty much everything else. That's why CPython adopted the shortcuts that cause their built-in methods to misbehave by not cooperating with methods overridden by subclasses. A possible way out of this dilemma would be to offer two implementations for each of those types: one "internal", optimized for use by the interpreter and an external, easily extensible one.

But wait, this is what we have: `UserDict`, `UserList`, and `UserString` are not as fast as the built-ins but are easily extensible. The pragmatic approach taken by CPython means we also get to use, in our own applications, the highly optimized implementations that are hard to subclass. Which makes sense, considering that it's not so often that we need a custom mapping, list or string, but we use `dict`, `list` and `str` every day. We just need to be aware of the trade-offs involved.

Inheritance across languages

Alan Kay coined the term "object oriented", and Smalltalk had only single inheritance, although there are forks with various forms of multiple inheritance support, including the modern Squeak and Pharo Smalltalk dialects which support traits — a language construct that fulfills the role of a mixin class, while avoiding some of the issues with multiple inheritance.

The first popular language to implement multiple inheritance was C++, and the feature was abused enough that Java — intended as a C++ replacement — was designed without support for multiple inheritance of implementation, i.e. no mixin classes. That is, until Java 8 introduced default methods which make interfaces very similar to the abstract classes which are used to define interfaces in C++ and in Python. Except that Java interfaces cannot have state — a key distinction. After Java, probably the most widely deployed JVM language is Scala, and it implements traits. Other languages supporting traits are the latest stable versions of PHP and Groovy, and the under-construction languages Rust and Perl 6 — so it's fair to say that traits are trendy as I write this.

Ruby offers an original take on multiple inheritance: it does not support it, but introduces mixins as a language feature. A Ruby class can include a module in its body, so the methods defined in the module become part of the class implementation. This is a "pure" form of mixin, with no inheritance involved, and it's clear that a Ruby mixin has no influence on the type of the class where it's used. This provides the benefits of mixins, while avoiding many of its usual problems.

Two recent languages that are getting a lot of traction severely limit inheritance: Go and Julia. Go has no inheritance at all, but it implements interfaces in a way that resembles a static form of duck typing (see "Soapbox" on page 344 for more about this). Julia avoids the terms "classes" and has only "types". Julia has a type hierarchy but subtypes cannot inherit structure, only behaviors, and only abstract types can be subtyped. In addition, Julia methods are implemented using multiple dispatch — a more advanced form of the mechanism we saw in "Generic functions with single dispatch" on page 202.

CHAPTER 13

Operator overloading: doing it right

> There are some things that I kind of feel torn about, like operator overloading. I left out operator overloading as a fairly personal choice because I had seen too many people abuse it in C++ [1]
>
> — James Gosling
> *creator of Java*

Operator overloading allows user-defined objects to interoperate with infix operators such as `+` and `|` or unary operators like `-` and `~`. More generally, function invocation `()`, attribute access `.` and item access/slicing `[]` are also operators in Python, but this chapter covers unary and infix operators.

In "Emulating numeric types" on page 9 (Chapter 1) we saw some trivial implementations of operators in a bare bones `Vector` class. The `__add__` and `__mul__` methods in Example 1-2 were written to show how special methods support operator overloading, but there are subtle problems in their implementations which we overlooked. Also, in Example 9-2 we noted that the `Vector2d.__eq__` method considers this to be `True`: `Vector(3, 4) == [3, 4]` — which may or not make sense. We will address those matters in this chapter.

In the next sections we will cover:

- How Python supports infix operators with operands of different types.
- Using duck typing or explicit type checks to deal with operands of various types.
- How an infix operator method should signal it cannot handle an operand.
- The special behavior of the rich comparison operators (e.g. `==`, `>`, `<=` etc.).

[1] Source: The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling.

- The default handling of augmented assignment operators, like +=, and how to overload them.

Operator overloading 101

Operator overloading has a bad name in some circles. It is a language feature that can be (and has been) abused, resulting in programmer confusion, bugs and unexpected performance bottlenecks. But if well used, leads to pleasurable APIs and readable code. Python strikes a good balance between flexibility, usability and safety by imposing some limitations:

- We cannot overload operators for the built-in types.
- We cannot create new operators, only overload existing ones.
- A few operators can't be overloaded: `is`, `and`, `or`, `not` (but the bitwise `&`, `|`, `~` can).

In Chapter 10 we already had one infix operator in `Vector`: `==`, supported by the `__eq__` method. In this chapter we'll improve the implementation of `__eq__` to better handle operands of types other than `Vector`. However, the rich comparison operators (`== != > < >= <=`) are special cases in operator overloading, so we'll start by overloading four arithmetic operators in `Vector`: the unary `-` and `+`, followed by the infix `+` and `*`.

Let's start with the easiest topic: unary operators.

Unary operators

Section 6.5. Unary arithmetic and bitwise operations of the *Expressions* chapter in the *Language Reference* lists three unary operators, shown here with their associated special methods:

`-` `__neg__`
: Arithmetic unary negation. If `x` is `-2` then `-x == 2`.

`+` `__pos__`
: Arithmetic unary plus. Usually `x == +x`, but there are a few cases when that's not true. See "When x and +x are not equal" on page 375 if you're curious.

`~` `__invert__`
: Bitwise inverse of an integer, defined as `~x == -(x+1)`. If `x` is `2` then `~x == -3`.

The `Data Model` page of the *Language Reference* also lists the `abs(…)` built-in function as an unary operator. The associated special method is `__abs__`, as we've seen before, starting with "Emulating numeric types" on page 9.

It's easy to support the unary operators. Simply implement the appropriate special method, which will receive just one argument: `self`. Use whatever logic makes sense in your class, but stick to the fundamental rule of operators: always return a new object. In other words, do not modify `self`, but create and return a new instance of a suitable type.

In the case of `-`, `+`, the result will probably be an instance of the same class as `self`; for `+`, returning a copy of `self` is the best approach most of the time. For `abs(…)` the result should be a scalar number. As for `~`, it's difficult to say what would be a sensible result if you're not dealing with bits in an integer, but in an *ORM* it could make sense to return the negation of an SQL `WHERE` clause, for example.

As promised before, we'll implement several new operators on the `Vector` class from Chapter 10. Example 13-1 shows the `__abs__` method we already had in Example 10-16, and the newly added `__neg__` and `__pos__` unary operator method.

Example 13-1. `vector_v6.py`: unary operators - + added to Example 10-16.

```
    def __abs__(self):
        return math.sqrt(sum(x * x for x in self))

    def __neg__(self):
        return Vector(-x for x in self)  ❶

    def __pos__(self):
        return Vector(self)  ❷
```

❶ To compute `-v`, build a new `Vector` with every component of `self` negated.

❷ To compute `+v`, build a new `Vector` with every component of `self`.

Recall that `Vector` instances are iterable, and the `Vector.__init__` takes an iterable argument, so the implementations of `__neg__` and `__pos__` are short and sweet.

We'll not implement `__invert__`, so if the user tries `~v` on a `Vector` instance, Python will raise `TypeError` with a clear message: "bad operand type for unary ~: `'Vector'`".

The following box covers a curiosity that may help you win a bet about unary `+` someday. The next important topic is "Overloading + for vector addition" on page 377.

When x and +x are not equal

Everybody expects that `x == +x`, and that is true almost all the time in Python, but I found two cases in the standard library where `x != +x`.

The first case involves the `decimal.Decimal` class. You can have `x != +x` if `x` is a `Decimal` instance created in an arithmetic context and `+x` is then evaluated in a context with different settings. For example, x is calculated in a context with a certain precision, but the precision of the context is changed and then +x is evaluated. See Example 13-2 for a demonstration.

Example 13-2. A change in the arithmetic context precision may cause x to differ from +x.

```
>>> import decimal
>>> ctx = decimal.getcontext()  ❶
>>> ctx.prec = 40  ❷
>>> one_third = decimal.Decimal('1') / decimal.Decimal('3')  ❸
>>> one_third  ❹
Decimal('0.3333333333333333333333333333333333333333')
>>> one_third == +one_third  ❺
True
>>> ctx.prec = 28  ❻
>>> one_third == +one_third  ❼
False
>>> +one_third  ❽
Decimal('0.3333333333333333333333333333')
```

❶ Get a reference to the current global arithmetic context.

❷ Set the precision of the arithmetic context to 40.

❸ Compute 1/3 using the current precision.

❹ Inspect the result; there are 40 digits after the decimal point.

❺ `one_third == +one_third` is `True`.

❻ Lower precision to 28 — the default for `Decimal` arithmetic in Python 3.4.

❼ Now `one_third == +one_third` is `False`.

❽ Inspect `+one_third`; there are 28 digits after the `'.'` here.

The fact is that each occurrence of the expression `+one_third` produces a new `Decimal` instance from the value of `one_third`, but using the precision of the current arithmetic context.

The second case where `x != +x` you can find in the `collections.Counter` documentation. The `Counter` class implements several arithmetic operators, including infix `+` to add the tallies from two `Counter` instances. However, for practical reasons, `Counter` addition discards from the result any item with a negative or zero count. And the prefix `+` is a shortcut for adding an empty `Counter`, therefore it produces a new `Counter` preserving only the tallies that are greater than zero. See Example 13-3.

Example 13-3. Unary + produces a new `Counter` without zeroed or negative tallies.

```
>>> ct = Counter('abracadabra')
>>> ct
Counter({'a': 5, 'r': 2, 'b': 2, 'd': 1, 'c': 1})
```

```
>>> ct['r'] = -3
>>> ct['d'] = 0
>>> ct
Counter({'a': 5, 'b': 2, 'c': 1, 'd': 0, 'r': -3})
>>> +ct
Counter({'a': 5, 'b': 2, 'c': 1})
```

Now, back to our regularly scheduled programming.

Overloading + for vector addition

The `Vector` class is a sequence type, and section Emulating container types in the Python Data model page says sequences should support the `+` operator for concatenation and `*` for repetition. However, here we will implement `+` and `*` as mathematical vector operations, which are a bit harder but more meaningful for a `Vector` type.

Adding two Euclidean vectors results in a new vector in which the components are the pairwise additions of the components of the addends. To illustrate:

```
>>> v1 = Vector([3, 4, 5])
>>> v2 = Vector([6, 7, 8])
>>> v1 + v2
Vector([9.0, 11.0, 13.0])
>>> v1 + v2 == Vector([3+6, 4+7, 5+8])
True
```

What happens if we try to add two `Vector` instances of different lengths? We could raise an error, but considering practical applications (such as information retrieval), it's better to fill out the shortest `Vector` with zeros. This is the result we want:

```
>>> v1 = Vector([3, 4, 5, 6])
>>> v3 = Vector([1, 2])
>>> v1 + v3
Vector([4.0, 6.0, 5.0, 6.0])
```

Given these basic requirements, the implementation of `__add__` is short and sweet:

Example 13-4. `Vector.__add__` method, take #1

```
# inside the Vector class

def __add__(self, other):
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)  # ❶
    return Vector(a + b for a, b in pairs)  # ❷
```

❶ `pairs` is a generator that will produce tuples `(a, b)` where `a` is from `self`, and `b` is from `other`. If `self` and `other` have different lengths, `fillvalue` is used to supply the missing values for the shortest iterable.

❷ A new `Vector` is built from a generator expression producing one sum for item in `pairs`.

Note how `__add__` returns a new `Vector` instance, and does not affect `self` or `other`.

> Special methods implementing unary or infix operators should never change their operands. Expressions with such operators are expected to produce results by creating new objects. Only augmented assignment operators may change the first operand (`self`), as discussed in "Augmented assignment operators" on page 390.

Example 13-4 allows adding `Vector` to a `Vector2d`, and `Vector` to a tuple or any iterable that produces numbers:

Example 13-5. `Vector.__add__` take #1 supports non-`Vector` objects too.

```
>>> v1 = Vector([3, 4, 5])
>>> v1 + (10, 20, 30)
Vector([13.0, 24.0, 35.0])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> v1 + v2d
Vector([4.0, 6.0, 5.0])
```

Both additions in Example 13-5 work because `__add__` uses `zip_longest(…)`, which can consume any iterable, and the generator expression to build the new `Vector` merely performs `a + b` with the pairs produced by `zip_longest(…)`, so an iterable producing any number items will do.

However, if we swap the operands, the mixed-type additions fail:

Example 13-6. `Vector.__add__` take #1 fails with non-`Vector` left operands.

```
>>> v1 = Vector([3, 4, 5])
>>> (10, 20, 30) + v1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "Vector") to tuple
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> v2d + v1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Vector2d' and 'Vector'
```

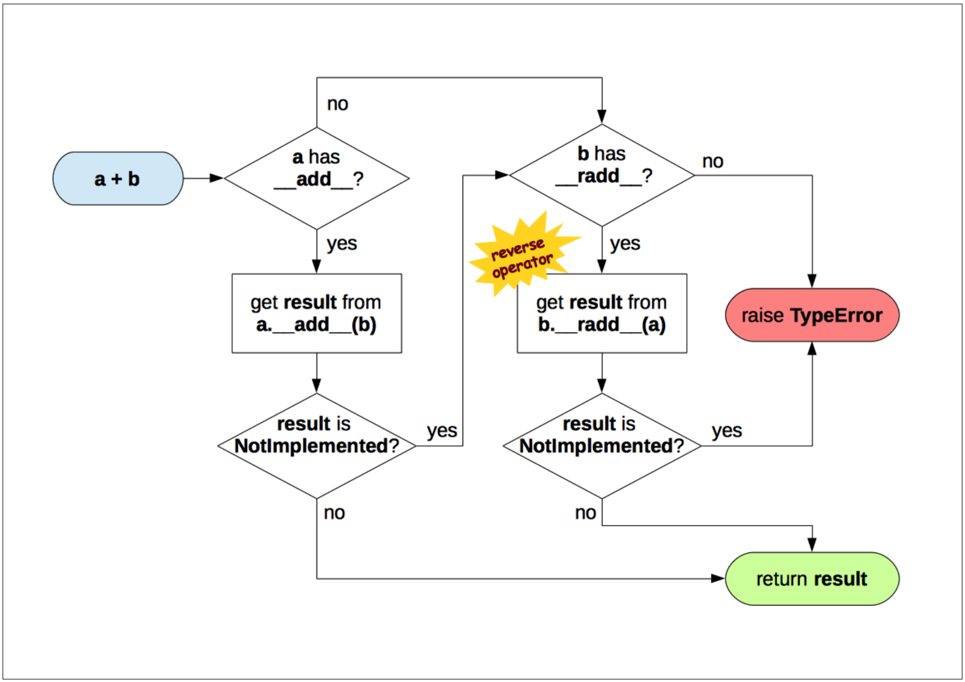

a + b
a has __add__?
no
yes
get result from a.__add__(b)
result is NotImplemented?
yes
no
b has __radd__?
no
yes
reverse operator
get result from b.__radd__(a)
result is NotImplemented?
yes
no
raise TypeError
return result

Figure 13-1. Flowchart for computing `a + b` with `__add__` and `__radd__`.

To support operations involving objects of different types, Python implements a special dispatching mechanism for the infix operator special methods. Given an expression `a + b`, the interpreter will perform these steps (also see Figure 13-1):

1. If `a` has `__add__`, call `a.__add__(b)` and return result unless it's `NotImplemented`.
2. If `a` doesn't have `__add__`, or calling it returns `NotImplemented`, check if `b` has `__radd__`, then call `b.__radd__(a)` and return result unless it's `NotImplemented`.
3. If `b` doesn't have `__radd__`, or calling it returns `NotImplemented`, raise `TypeError` with an `'unsupported operand types'` message.

The `__radd__` method is called the "reflected", or "reversed" version of `__add__`. I prefer to call them "reversed" special methods[2]. Three of this book's technical reviewers — Alex, Anna and Leo — told me they like to think of them as the "right" special methods,

[2] The Python documentation uses both terms. The Data model page uses "reflected", but the Implementing the arithmetic operations section in the `numbers` module docs mention "forward" and "reverse" methods, and I find this terminology better, because "forward" and "reversed" clearly name each of the directions, while "reflected" doesn't have an obvious opposite.

since they are called on the right-hand operand. Whatever “r"-word you prefer, that’s what the “r” prefix stands for in `__radd__`, `__rsub__`, etc.

Therefore, to make the mixed-type additions in Example 13-6 work, we need to implement the `Vector.__radd__` method which Python will invoke as a fall back if the left operand does not implement `__add__` or if it does but returns `NotImplemented` to signal that doesn’t know how to handle the right operand.

> Do not confuse `NotImplemented` with `NotImplementedError`. The first, `NotImplemented` is a special singleton value that an infix operator special method should **`return`** to tell the interpreter it cannot handle a given operand. In contrast, `NotImplementedError` is an exception that stub methods in abstract classes **`raise`** to warn that they must be overwritten by subclasses.

The simplest possible `__radd__` that works is shown in Example 13-7.

Example 13-7. `Vector.__add__` and `__radd__` methods.

```
# inside the Vector class

def __add__(self, other):  # ❶
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
    return Vector(a + b for a, b in pairs)

def __radd__(self, other):  # ❷
    return self + other
```

❶ No changes to `__add__` from Example 13-4; listed here because `__radd__` uses it.

❷ `__radd__` just delegates to `__add__`.

Often, `__radd__` can be as simple as that: just invoke the proper operator, therefore delegating to `__add__` in this case. This applies to any commutative operator; `+` is commutative when dealing with numbers or our vectors, but it’s not commutative when concatenating sequences in Python.

The methods in Example 13-4 work with `Vector` objects, or any iterable with numeric items, such as a `Vector2d`, a `tuple` of integers or an `array` of floats. But if provided with a non-iterable object, `__add__` fails with a message that is not very helpful, as in Example 13-8.

Example 13-8. `Vector.__add__` method needs an iterable operand.

```
>>> v1 + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 328, in __add__
    pairs = itertools.zip_longest(self, other, fillvalue=0.0)
TypeError: zip_longest argument #2 must support iteration
```

Another unhelpful message is given if an operand is iterable but its items cannot be added to the `float` items in the `Vector`. See Example 13-9.

Example 13-9. `Vector.__add__` method needs an iterable with numeric items.

```
>>> v1 + 'ABC'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 329, in __add__
    return Vector(a + b for a, b in pairs)
  File "vector_v6.py", line 243, in __init__
    self._components = array(self.typecode, components)
  File "vector_v6.py", line 329, in <genexpr>
    return Vector(a + b for a, b in pairs)
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

The problems in Example 13-8 and Example 13-9 actually go deeper than obscure error messages: if an operator special method cannot return a valid result because of type incompatibility, it should return `NotImplemented` and not raise `TypeError`. By returning `NotImplemented` you leave the door open for the implementer of the other operand type to perform the operation when Python tries the reversed method call.

In the spirit of duck typing, we will refrain from testing the type of the `other` operand, or the type of its elements. We'll catch the exceptions and return `NotImplemented`. If the interpreter has not yet reversed the operands, it will try that. If the reverse method call returns `NotImplemented`, then Python will raise issue `TypeError` with a standard error message like “unsupported operand type(s) for +: *Vector* and *str*“.

The final implementation of the special methods for `Vector` addition are in Example 13-10.

Example 13-10. `vector_v6.py`: operator `+` methods added to `vector_v5.py` (Example 10-16).

```
    def __add__(self, other):
        try:
            pairs = itertools.zip_longest(self, other, fillvalue=0.0)
            return Vector(a + b for a, b in pairs)
        except TypeError:
            return NotImplemented

    def __radd__(self, other):
        return self + other
```

If an infix operator method raises an exception, it aborts the operator dispatch algorithm. In the particular case of `TypeError`, it is often better to catch it and `return NotImplemented`. This allows the interpreter to try calling the reversed operator method which may correctly handle the computation with the swapped operands, if they are of different types.

At this point we have safely overloaded the `+` operator by writing `__add__` and `__radd__`. We will now tackle another infix operator: `*`.

Overloading * for scalar multiplication

What does `Vector([1, 2, 3]) * x` mean? If `x` is a number, that would be a scalar product, and the result would be a new `Vector` with each component multiplied by `x` — also known as an elementwise multiplication:

```
>>> v1 = Vector([1, 2, 3])
>>> v1 * 10
Vector([10.0, 20.0, 30.0])
>>> 11 * v1
Vector([11.0, 22.0, 33.0])
```

Another kind of product involving `Vector` operands would be the dot product of two vectors — or matrix multiplication, if you take of one vector as a 1 x N matrix and the other as a N x 1 matrix. The current practice in NumPy and similar libraries is not to overload the `*` with these two meanings, but to use `*` only for the scalar product. For example, in NumPy, `numpy.dot()` computes the dot product[3].

Back to our scalar product, again we start with the simplest `__mul__` and `__rmul__` methods that could possibly work:

```
    # inside the Vector class

    def __mul__(self, scalar):
        return Vector(n * scalar for n in self)

    def __rmul__(self, scalar):
        return self * scalar
```

Those methods do work, except when provided with incompatible operands. The `scalar` argument has to be a number that when multiplied by a `float` produces another `float` (because our `Vector` class uses an `array` of floats internally). So a `complex` number

[3] The `@` sign can be used as an infix dot product operator starting with Python 3.5. More about it in "The new `@` infix operator in Python 3.5" on page 385.

will not do, but the scalar can be an `int`, a `bool` (because `bool` is a subclass of `int`) or even a `fractions.Fraction` instance.

We could use the same duck typing technique as we did in Example 13-10 and catch a `TypeError` in `__mul__`, but there is another, more explicit way which makes sense in this situation: *goose typing*. We use `isinstance()` to check the type of `scalar`, but instead of hardcoding some concrete types we check against the `numbers.Real` ABC, which covers all the types we need, and keeps our implementation open to future numeric types that declare themselves actual or *virtual subclasses* of the `numbers.Real` ABC.

> Recall from "ABCs in the standard library." on page 323 that `decimal.Decimal` is not registered as a virtual subclass of `numbers.Real`. Therefore our `Vector` class will not handle `decimal.Decimal` numbers.

Example 13-11. `vector_v7.py`: operator `*` methods added (see full listing in the Fluent Python git repository)

```
from array import array
import reprlib
import math
import functools
import operator
import itertools
import numbers  # ❶

class Vector:
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    # many methods omitted in book listing, see vector_v7.py
    # in https://github.com/fluentpython/example-code ...

    def __mul__(self, scalar):
        if isinstance(scalar, numbers.Real):  # ❷
            return Vector(n * scalar for n in self)
        else:  # ❸
            return NotImplemented

    def __rmul__(self, scalar):
        return self * scalar  # ❹
```

❶ Import the `numbers` module for type checking.

❷ If `scalar` is an instance of a `numbers.Real` subclass, create new `Vector` with multiplied component values.

❸ Otherwise, raise `TypeError` with an explicit message.

❹ In this example, `__rmul__` works fine by just performing `self * scalar`, delegating to the `__mul__` method.

With Example 13-11 we can multiply `Vectors` by scalar values of the usual and not so usual numeric types:

```
>>> v1 = Vector([1.0, 2.0, 3.0])
>>> 14 * v1
Vector([14.0, 28.0, 42.0])
>>> v1 * True
Vector([1.0, 2.0, 3.0])
>>> from fractions import Fraction
>>> v1 * Fraction(1, 3)
Vector([0.3333333333333333, 0.6666666666666666, 1.0])
```

Implementing `+` and `*` we saw the most common patterns for coding infix operators. The techniques we described for `+` and `*` are applicable to all operators listed in Table 13-1 (the in-place operators will be covered in "Augmented assignment operators" on page 390)

Table 13-1. Infix operator method names. The in-place operators are used for augmented assignment. Comparison operators are in Table 13-2.

operator	forward	reverse	in-place	description
`+`	`__add__`	`__radd__`	`__iadd__`	addition or concatenation
`-`	`__sub__`	`__rsub__`	`__isub__`	subtraction
`*`	`__mul__`	`__rmul__`	`__imul__`	multiplication or repetition
`/`	`__truediv__`	`__rtruediv__`	`__itruediv__`	true division
`//`	`__floordiv__`	`__rfloordiv__`	`__ifloordiv__`	floor division
`%`	`__mod__`	`__rmod__`	`__imod__`	modulo
`divmod()`	`__divmod__`	`__rdivmod__`	`__idivmod__`	returns tuple of floor division quotient and modulo
`**`, `pow()`	`__pow__`	`__rpow__`	`__ipow__`	exponentiation[a]
`@`	`__matmul__`	`__rmatmul__`	`__imatmul__`	matrix multiplication[b]
`&`	`__and__`	`__rand__`	`__iand__`	bitwise and
`|`	`__or__`	`__ror__`	`__ior__`	bitwise or
`^`	`__xor__`	`__rxor__`	`__ixor__`	bitwise xor
`<<`	`__lshift__`	`__rlshift__`	`__ilshift__`	bitwise shift left
`>>`	`__rshift__`	`__rrshift__`	`__irshift__`	bitwise shift right

operator	forward	reverse	in-place	description

[a] pow takes an optional third argument modulo: pow(a, b, modulo), also supported by the special methods when invoked directly, e.g. a.__pow__(b, modulo).

[b] new in Python 3.5.

The rich comparison operators are another category of infix operators, using a slightly different set of rules. We cover them in the next main section: “Rich comparison operators” on page 386.

The following optional side bar is about the @ operator introduced in Python 3.5 — not yet released at the time of this writing.

The new @ infix operator in Python 3.5

Python 3.4 does not have an infix operator for the dot product. However, as I write this, Python 3.5 pre-alpha already implements PEP 465 — A dedicated infix operator for matrix multiplication, making the @ sign available for that purpose, e.g. a @ b is the dot product of a and b. The @ operator is supported by the special methods __matmul__, __rmatmul__, and __imatmul__, named for “matrix multiplication”. These methods are not used anywhere in the standard library at this time, but are recognized by the interpreter in Python 3.5 so the NumPy team — and the rest of us — can support the @ operator in user-defined types. The parser was also changed to handle the infix @ (a @ b is a syntax error in Python 3.4).

Just for fun, after compiling Python 3.5 from source I was able to implement and test the @ operator for the Vector dot product.

These are the simple tests I did:

```
>>> va = Vector([1, 2, 3])
>>> vz = Vector([5, 6, 7])
>>> va @ vz == 38.0  # 1*5 + 2*6 + 3*7
True
>>> [10, 20, 30] @ vz
380.0
>>> va @ 3
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for @: 'Vector' and 'int'
```

And here is the code of the relevant special methods:

```
class Vector:
    # many methods omitted in book listing

    def __matmul__(self, other):
        try:
            return sum(a * b for a, b in zip(self, other))
        except TypeError:
            return NotImplemented

    def __rmatmul__(self, other):
        return self @ other
```

The full source is in file `vector_py3_5.py` in the Fluent Python git repository.

Remember to try it with Python 3.5, otherwise you'll get a `SyntaxError`!

Rich comparison operators

The handling of the rich comparison operators `== != > < >= <=` by the Python interpreter is similar to what we just saw, but differs in two important aspects:

1. The same set of methods are used in forward and reverse operator calls. The rules are summarized in Table 13-2. For example, in the case of `==`, both the forward and reverse calls invoke `__eq__`, only swapping arguments; and a forward call to `__gt__` is followed by a reverse call to `__lt__` with the swapped arguments.
2. In the case of `==` and `!=`, if the reverse call fails, Python compares the object ids instead of raising `TypeError`.

Table 13-2. Rich comparison operators: reverse methods invoked when the initial method call returns `NotImplemented`.

group	infix operator	forward method call	reverse method call	fall back
equality	`a == b`	`a.__eq__(b)`	`b.__eq__(a)`	return `id(a) == id(b)`
	`a != b`	`a.__ne__(b)`	`b.__ne__(a)`	return `not (a == b)`
ordering	`a > b`	`a.__gt__(b)`	`b.__lt__(a)`	raise `TypeError`
	`a < b`	`a.__lt__(b)`	`b.__gt__(a)`	raise `TypeError`
	`a >= b`	`a.__ge__(b)`	`b.__le__(a)`	raise `TypeError`
	`a <= b`	`a.__le__(b)`	`b.__ge__(a)`	raise `TypeError`

New behavior in Python 3

The fall back step for all comparison operators changed from Python 2. For `__ne__`, Python 3 now returns the negated result of `__eq__`. For the ordering comparison operators, Python 3 raises `TypeError` with a message like `'unorderable types: int() < tuple()'`. In Python 2 those comparisons produced weird results taking into account object types and ids in some arbitrary way. However, it really makes no sense to compare an `int` to a `tuple`, for example, so raising `TypeError` in such cases is a real improvement in the language.

Given these rules, let’s review and improve the behavior of the `Vector.__eq__` method, which was coded as follows in `vector_v5.py` (Example 10-16):

```
class Vector:
    # many lines omitted

    def __eq__(self, other):
        return (len(self) == len(other) and
                all(a == b for a, b in zip(self, other)))
```

That method produces the following results:

Example 13-12. Comparing a `Vector` to a `Vector`, a `Vector2d` and a `tuple`.

```
>>> va = Vector([1.0, 2.0, 3.0])
>>> vb = Vector(range(1, 4))
>>> va == vb  # ❶
True
>>> vc = Vector([1, 2])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> vc == v2d  # ❷
True
>>> t3 = (1, 2, 3)
>>> va == t3  # ❸
True
```

❶ Two `Vector` instances with equal numeric components compare equal.

❷ A `Vector` and a `Vector2d` are also equal if their components are equal.

❸ A `Vector` is also considered equal to a `tuple` or any iterable with numeric items of equal value.

The last one of the results in Example 13-12 is probably not desirable. I really have no hard rule about this, it depends on the application context. But the Zen of Python says:

> In the face of ambiguity, refuse the temptation to guess.
>
> — Zen of Python

Excessive liberality in the evaluation of operands may lead to surprising results, and programmers hate surprises.

Taking a clue from Python itself, we can see that `[1,2] == (1, 2)` is `False`. Therefore, let’s be conservative and do some type checking. If the second operand is a `Vector` instance (or an instance of a `Vector` subclass), then use the same logic as the current `__eq__`. Otherwise, return `NotImplemented` and let Python handle that. See Example 13-13.

Example 13-13. `vector_v8.py`: improved `__eq__` in the `Vector` class.

```
    def __eq__(self, other):
        if isinstance(other, Vector):  ❶
            return (len(self) == len(other) and
                    all(a == b for a, b in zip(self, other)))
        else:
            return NotImplemented  ❷
```

❶ If the `other` operand is an instance of `Vector` (or of a `Vector` subclass), perform the comparison as before.

❷ Otherwise, return `NotImplemented`.

If you run the tests in Example 13-12 with the new `Vector.__eq__` from Example 13-13, this is what you get now:

Example 13-14. Same comparisons as Example 13-12: last result changed.

```
>>> va = Vector([1.0, 2.0, 3.0])
>>> vb = Vector(range(1, 4))
>>> va == vb  # ❶
True
>>> vc = Vector([1, 2])
>>> from vector2d_v3 import Vector2d
>>> v2d = Vector2d(1, 2)
>>> vc == v2d  # ❷
True
>>> t3 = (1, 2, 3)
>>> va == t3  # ❸
False
```

❶ Same result as before, as expected.

❷ Same result as before, but why? Explanation coming up.

❸ Different result; this is what we wanted. But why does it work? Read on…

Among the three results in Example 13-14 the first one is no news, but the last two were caused by `__eq__` returning `NotImplemented` in Example 13-13. Here is what happens in the example with a `Vector` and a `Vector2d`, step by step:

1. To evaluate `vc == v2d`, Python calls `Vector.__eq__(vc, v2d)`.
2. `Vector.__eq__(vc, v2d)` verifies that `v2d` is not a `Vector` and returns `NotImplemented`.
3. Python gets `NotImplemented` result, so it tries `Vector2d.__eq__(v2d, vc)`.
4. `Vector2d.__eq__(v2d, vc)` turns both operands into tuples an compares them: the result is `True` (the code for `Vector2d.__eq__` is in Example 9-9).

As for the comparison between `Vector` and `tuple` in Example 13-14, the actual steps are:

1. To evaluate `va == t3`, Python calls `Vector.__eq__(va, t3)`.
2. `Vector.__eq__(va, t3)` verifies that `t3` is not a `Vector` and returns `NotImplemented`.
3. Python gets `NotImplemented` result, so it tries `tuple.__eq__(t3, va)`.
4. `tuple.__eq__(t3, va)` has no idea what a `Vector` is, so it returns `NotImplemented`.
5. In the special case of `==`, if the reversed call returns `NotImplemented`, Python compares object ids as a last resort.

How about `!=`? We don't need to implement it because the fall back behavior of the `__ne__` inherited from `object` suits us: when `__eq__` is defined and does not return `NotImplemented`, `__ne__` returns that result negated.

In other words, given the same objects we used in Example 13-14, the results for `!=` are consistent:

```
>>> va != vb
False
>>> vc != v2d
False
>>> va != (1, 2, 3)
True
```

The `__ne__` inherited from `object` works like the following code — except that the original is written in C[4]:

```
    def __ne__(self, other):
        eq_result = self == other
        if eq_result is NotImplemented:
            return NotImplemented
        else:
            return not eq_result
```

[4] The logic for `object.__eq__` and `object.__ne__` is in function `object_richcompare` in Objects/typeobject.c in the CPython source code.

Python 3 documentation bug

As I write this the rich comparison method documentation states: “The truth of x==y does not imply that x!=y is false. Accordingly, when defining __eq__(), one should also define __ne__() so that the operators will behave as expected.” That was true for Python 2, but in Python 3 that’s not good advice, because a useful default __ne__ implementation is inherited from the object class, and it’s rarely necessary to override it. The new behavior is documented in Guido’s What’s New In Python 3.0, section *Operators And Special Methods*. The documentation bug is recorded as issue 4395.

After covering the essentials of infix operator overloading, let’s turn to a different class of operators: the augmented assignment operators.

Augmented assignment operators

Our Vector class already supports the augmented assignment operators += and *=. Here they are in action:

Example 13-15. Augmented assignment works with immutable targets by creating new instances and rebinding.

```
>>> v1 = Vector([1, 2, 3])
>>> v1_alias = v1  # ❶
>>> id(v1)  # ❷
4302860128
>>> v1 += Vector([4, 5, 6])  # ❸
>>> v1  # ❹
Vector([5.0, 7.0, 9.0])
>>> id(v1)  # ❺
4302859904
>>> v1_alias  # ❻
Vector([1.0, 2.0, 3.0])
>>> v1 *= 11  # ❼
>>> v1  # ❽
Vector([55.0, 77.0, 99.0])
>>> id(v1)
4302858336
```

❶ Create alias so we can inspect the Vector([1, 2, 3]) object later.

❷ Remember the id of the initial Vector bound to v1.

❸ Perform augmented addition.

❹ The expected result…

❺ …but a new Vector was created.

❻ Inspect `v1_alias` to confirm the original `Vector` was not altered.

❼ Perform augmented multiplication.

❽ Again, the expected result, but a new `Vector` was created.

If a class does not implement the in-place operators listed in Table 13-1, the augmented assignment operators are just syntactic sugar: `a += b` is evaluated exactly as `a = a + b`. That's the expected behavior for immutable types, and if you have `__add__` then `+=` will work with no additional code.

However, if you do implement an in-place operator method such as `__iadd__`, then that method is called to compute the result of `a += b`. As the name says, those operators are expected to change the left-hand operand in-place, and not create a new object as the result.

The in-place special methods should never be implemented for immutable types like our `Vector` class. This is fairly obvious, but worth stating anyway.

To show the code of an in-place operator, we will extend the `BingoCage` class from Example 11-12 to implement `__add__` and `__iadd__`.

We'll call the subclass `AddableBingoCage`. This is the behavior we want for the `+` operator:

Example 13-16. A new `AddableBingoCage` instance can be created with `+`.

```
>>> vowels = 'AEIOU'
>>> globe = AddableBingoCage(vowels)  ➊
>>> globe.inspect()
('A', 'E', 'I', 'O', 'U')
>>> globe.pick() in vowels  ➋
True
>>> len(globe.inspect())  ➌
4
>>> globe2 = AddableBingoCage('XYZ')  ➍
>>> globe3 = globe + globe2
>>> len(globe3.inspect())  ➎
7
>>> void = globe + [10, 20]  ➏
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for +: 'AddableBingoCage' and 'list'
```

❶ Create a `globe` instance with 5 items (each of the `vowels`).

❷ Pop one of the items, and verify it is one the `vowels`.

❸ Confirm that the `globe` is down to 4 items.

❹ Create a second instance, with 3 items.

❺ Create a third instance by adding the previous two. This instance has 7 items.

❻ Attempting to add an `AddableBingoCage` to a `list` fails with `TypeError`. That error message is produced by the Python interpreter when our `__add__` method returns `NotImplemented`.

Since an `AddableBingoCage` is mutable, this is how it will work when we implement `__iadd__`:

Example 13-17. An existing `AddableBingoCage` can be loaded with `+=` (continuing from Example 13-16.)

```
>>> globe_orig = globe  ❶
>>> len(globe.inspect())  ❷
4
>>> globe += globe2  ❸
>>> len(globe.inspect())
7
>>> globe += ['M', 'N']  ❹
>>> len(globe.inspect())
9
>>> globe is globe_orig  ❺
True
>>> globe += 1  ❻
Traceback (most recent call last):
  ...
TypeError: right operand in += must be 'AddableBingoCage' or an iterable
```

❶ Create an alias so we can check the identity of the object later.

❷ `globe` has 4 items here.

❸ An `AddableBingoCage` instance can receive items from another instance of the same class.

❹ The right-hand operand of `+=` can also be any iterable.

❺ Throughout this example, `globe` has always referred to the `globe_orig` object.

❻ Trying to add a non-iterable to an `AddableBingoCage` fails with a proper error message.

Note that the `+=` operator is more liberal than `+` with regard to the second operand. With `+` we want both operands to be of the same type `AddableBingoCage` because if we accepted different types this might cause confusion as to the type of the result. With the

+= the situation is clearer: the left-hand object is updated in-place, so there's no doubt about the type of the result.

> I validated the contrasting behavior of + and += by observing how the list built-in type works. Writing my_list + x you can only concatenate one list to another list, but if you write my_list += x you can extend the left-hand list with items from any iterable x on the right-hand side. This is consistent with how the list.extend() method works: it accepts any iterable argument.

Now that we are clear on the desired behavior for AddableBingoCage we can look at its implementation in Example 13-18.

Example 13-18. bingoaddable.py: AddableBingoCage extends BingoCage to support + and +=.

```
import itertools  ❶

from tombola import Tombola
from bingo import BingoCage


class AddableBingoCage(BingoCage):  ❷

    def __add__(self, other):
        if isinstance(other, Tombola):  ❸
            return AddableBingoCage(self.inspect() + other.inspect())  ❹
        else:
            return NotImplemented

    def __iadd__(self, other):
        if isinstance(other, Tombola):
            other_iterable = other.inspect()  ❺
        else:
            try:
                other_iterable = iter(other)  ❻
            except TypeError:  ❼
                self_cls = type(self).__name__
                msg = "right operand in += must be {!r} or an iterable"
                raise TypeError(msg.format(self_cls))
        self.load(other_iterable)  ❽
        return self  ❾
```

❶ PEP 8 — Style Guide for Python Code recommends coding imports from the standard library above imports of your own modules.

❷ AddableBingoCage extends BingoCage

❸ Our __add__ will only work with an instance of Tombola as the second operand.

❺ Retrieve items from other, of it is an instance of Tombola.

❻ Otherwise, try obtain an iterator over other`footnote:[The `iter built-in function will be covered in the next chapter. Here I could have used tuple(other), and it would work, but at the cost of building a new tuple when all the .load(…) method needs is to iterate over its argument.].

❹ ❼ If that fails, raise an exception explaining what the user should do. When possible, error messages should explicitly guide the user to the solution.

❽ If we got this far, we can load the other_iterable into self.

❾ Very important: augmented assignment special methods must return self.

We can summarize the whole idea of in-place operators by contrasting the return statements that produce results in __add__ and __iadd__ in Example 13-18:

__add__
: The result is produced by calling the constructor AddableBingoCage to build a new instance.

__iadd__
: The result is produced by returning self, after it has been modified.

To wrap up this example, a final observation on Example 13-18: by design, no __radd__ was coded in AddableBingoCage, because there is no need for it. The forward method __add__ will only deal with right-hand operands of the same type, so if Python is trying to compute a + b where a is an AddableBingoCage and b is not, we return NotImplemented — maybe the class of b can make it work. But if the expression is b + a and b is not an AddableBingoCage, and it returns NotImplemented, then it's better to let Python give up and raise TypeError because we cannot handle b.

> In general, if a forward infix operator method — e.g. __mul__ — is designed to work only with operands of the same type as self, then it's useless to implement the corresponding reverse method — e.g. __rmul__ — because that, by definition, will only be invoked when dealing with an operand of a different type.

This concludes our exploration of operator overloading in Python.

Chapter summary

We started this chapter by reviewing some restrictions Python imposes on operator overloading: no overloading of operators in built-in types, overloading limited to existing operators, except for a few ones (is, and, or, not).

We got down to business with the unary operators, implementing `__neg__` and `__pos__`. Next came the infix operators, starting with `+`, supported by the `__add__` method. We saw that unary and infix operators are supposed produce results by creating new objects, and should never change their operands. To support operations with other types, we but return the `NotImplemented` special value — not an exception — allowing the interpreter to try again by swapping the operands and calling the reverse special method for that operator — e.g. `__radd__`. The algorithm Python uses to handle infix operators is summarized in the flowchart in Figure 13-1.

Mixing operand types means we need to detect when we get an operand we can't handle. In this chapter we did this in two ways: in the duck typing way, we just went ahead and tried the operation, catching a `TypeError` exception if it happened; later, in `__mul__`, we did it with an explicit `isinstance` test. There are pros and cons to these approaches: duck typing is more flexible, but explicit type checking is more predictable. When we did use `isinstance` we were careful to avoid testing with a concrete class, but used the `numbers.Real` ABC: `isinstance(scalar, numbers.Real)`. This is a good compromise between flexibility and safety, because existing or future user-defined types can be declared as actual or virtual subclasses of an ABC, as we saw in Chapter 11.

The next topic we covered were the rich comparison operators. We implemented `==` with `__eq__` and discovered that Python provides a handy implementation of `!=` in the `__ne__` inherited from the `object` base class. The way Python evaluates these operators along with `> < >= <=` is slightly different, with a different logic for choosing the reverse method, and special fall back handling for `==` and `!=`, which never generate errors because Python compares the object ids as a last resort.

In the last section we focused on augmented assignment operators. We saw that Python handles them by default as a combination of plain operator followed by assignment, that is: `a += b` is evaluated exactly as `a = a + b`. That always creates a new object, so it works for mutable or immutable types. For mutable objects, we can implement in-place special methods such as `__iadd__` for `+=`, and alter the value of the left-hand operand. To show this at work we left behind the immutable `Vector` class and worked on implementing a `BingoCage` subclass to support `+=` for adding items to the random pool, similar to the way the `list` built-in supports `+=` as a shortcut for the `list.extend()` method. While doing this, we discussed how `+` tends to be stricter than `+=` regarding the types it accepts. For sequence types, `+` usually requires that both operands are of the same type, while `+=` often accepts any iterable as the right-hand operand.

Further reading

Operator overloading is one area of Python programming where `isinstance` tests are common. In general, libraries should leverage dynamic typing — to be more flexible — by avoiding explicit type tests and just trying operations and then handling the exceptions, opening the door for working with objects regardless of their types, as long as they support the necessary operations. But Python ABCs allow a stricter form of duck typing, dubbed “goose typing” by Alex Martelli, which is often useful when writing code that overloads operators. So, if you skipped Chapter 11, make sure to read it.

The main reference for the operator special methods is the Data model page. It’s the canonical source, but at this time it’s plagued by that glaring bug mentioned in Python 3 documentation bug, advising “when defining `__eq__()`, one should also define `__ne__()`" when in Python 3 the `__ne__` inherited from the `object` class covers the vast majority of real needs, so implementing `__ne__` is rarely necessary in practice. Another relevant reading in the Python documentation is the Implementing the arithmetic operations section in the `numbers` module chapter of the standard library manual.

A related technique is generic functions, supported by the `@singledispatch` decorator in Python 3 (“Generic functions with single dispatch” on page 202). The *Python Cookbook*, 3rd. edition (O’Reilly, 2013), by David Beazley and Brian K. Jones, has a chapter titled *Implementing Multiple Dispatch with Function Annotations* (recipe 9.20) where some advanced metaprogramming — involving a metaclass — is used to implement type-based dispatching with function annotations. The second edition of the *Python Cookbook* by Martelli, Ravenscroft & Ascher has an interesting recipe (2.13, by Erik Max Francis) showing how to overload the `<<` operator to emulate the C++ `iostream` syntax in Python. Both books have other examples with operator overloading, I just picked two notable recipes.

The `functools.total_ordering` function is a class decorator (supported in Python 2.7 and later) that automatically generates methods for all rich comparison operators in any class that defines at least a couple of them. See the functools module docs.

If you are curious about operator method dispatching in languages with dynamic typing, two seminal readings are A Simple Technique for Handling Multiple Polymorphism by Dan Ingalls (member of the original Smalltalk team) and Arithmetic and Double Dispatching in Smalltalk-80 by Kurt J. Hebel and Ralph Johnson (Johnson became famous as one of the authors of the original *Design Patterns* book). Both papers provide deep insight into the power of polymorphism in languages with dynamic typing, like Smalltalk, Python and Ruby. Python does not use double dispatching for handling operators as described in those articles. The Python algorithm using forward and reverse operators is easier for user-defined classes to support than double dispatching, but requires special handling by the interpreter. In contrast, classic double dispatching is a general technique you can use in Python or any OO language beyond the specific context of infix operators, and in fact Ingalls, Hebel and Johnson use very different examples to describe it.

The article The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling from which I quoted the epigraph in this chapter, and two other snippets in **Soapbox** (below), appeared in Java Report, 5(7), July 2000 and C++ Report, 12(7), July/August 2000. It's an awesome reading if you are into programming language design.

Soapbox

Operator overloading: pros and cons

James Gosling, quoted at the top of this chapter, made the conscious decision to leave operator overloading out when he designed Java. In that same interview (The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling) he says:

> Probably about 20 to 30 percent of the population think of operator overloading as the spawn of the devil; somebody has done something with operator overloading that has just really ticked them off, because they've used like + for list insertion and it makes life really, really confusing. A lot of that problem stems from the fact that there are only about half a dozen operators you can sensibly overload, and yet there are thousands or millions of operators that people would like to define — so you have to pick, and often the choices conflict with your sense of intuition.

Guido van Rossum picked the middle way in supporting operator overloading: he did not leave the door open for users creating new arbitrary operators like `<=>` or `:-`), which prevents a Tower of Babel of custom operators, and allows the Python parser to be simple. Python also does not let you overload the operators of the built-in types, another limitation that promotes readability and predictable performance.

Gosling goes on to say:

> Then there's a community of about 10 percent that have actually used operator overloading appropriately and who really care about it, and for whom it's actually really important; this is almost exclusively people who do numerical work, where the notation is very important to appealing to people's intuition, because they come into it with an intuition about what the + means, and the ability to say "a + b" where a and b are complex numbers or matrices or something really does make sense.

The notation side of the issue cannot be underestimated. Here is an illustrative example from the realm of finances. In Python you can compute compound interest using a formula written like this:

```
interest = principal * ((1 + rate) ** periods - 1)
```

That same notation works regardless of the numeric types involved. If you are doing serious financial work, you can make sure that `periods` is an `int`, while `rate`, `interest` and `principal` are exact numbers — instances of the Python `decimal.Decimal` class — and that formula will work exactly as written.

But in Java, if you switch from `float` to `BigDecimal` to get arbitrary precision, you can't use infix operators anymore, as they only work with the primitive types. This is the same formula coded to work with `BigDecimal` numbers in Java:

```
    BigDecimal interest = principal.multiply(BigDecimal.ONE.add(rate)
                              .pow(periods).subtract(BigDecimal.ONE));
```

It’s clear that infix operators make formulas more readable, at least for most of us[5]. And operator overloading is necessary to support non-primitive types with infix operator notation. Having operator overloading in a high-level, easy to use language was probably a key reason for the amazing penetration of Python in scientific computing in recent years.

Of course, there are benefits to disallowing operator overloading in a language. It is arguably a sound decision for lower level systems languages where performance and safety are paramount. The much newer Go language followed the lead of Java in this regard and does not support operator overloading.

But overloaded operators, when used sensibly, do make code easier to read and write. It’s a great feature to have in a modern high level language.

A glimpse at lazy evaluation

If you look closely at the traceback in Example 13-9, you’ll see evidence of the *lazy* evaluation of generator expressions. Below is that same traceback, now with callouts:

Example 13-19. Same as Example 13-9.

```
>>> v1 + 'ABC'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "vector_v6.py", line 329, in __add__
    return Vector(a + b for a, b in pairs)  # ❶
  File "vector_v6.py", line 243, in __init__
    self._components = array(self.typecode, components)  # ❷
  File "vector_v6.py", line 329, in <genexpr>
    return Vector(a + b for a, b in pairs)  # ❸
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

❶ The `Vector` call gets a generator expression as its `components` argument. No problem at this stage.

❷ The `components` genexp is passed to the `array` constructor. Within the `array` constructor, Python tries to iterate over the genexp, causing the evaluation of the first item `a + b`. That’s when the `TypeError` occurs.

❸ The exception propagates to the `Vector` constructor call, where it is reported.

This shows how the generator expression is evaluated at the latest possible moment, and not where it is defined in the source code.

[5] My friend Mario Domenech Goulart, a core developer of the CHICKEN Scheme compiler will probably disagree with this.

In contrast, if the Vector constructor was invoked as Vector([a + b for a, b in pairs]), then the exception would happen right there, as the list comprehension tried to build a list to be passed as the argument to the Vector() call. The body of Vector.__init__ would not be reached at all.

Chapter 14 will cover generator expressions in detail, but I did not want to leave this accidental demonstration of their lazy nature go unnoticed.

PART V

Control flow

CHAPTER 14

Iterables, iterators and generators

> When I see patterns in my programs, I consider it a sign of trouble. The shape of a program should reflect only the problem it needs to solve. Any other regularity in the code is a sign, to me at least, that I'm using abstractions that aren't powerful enough — often that I'm generating by hand the expansions of some macro that I need to write[1].
>
> — Paul Graham
> *Lisp hacker and venture capitalist*

Iteration is fundamental to data processing. And when scanning datasets that don't fit in memory, we need a way to fetch the items *lazily*, that is, one at a time and on demand. This is what the Iterator pattern is about. This chapter shows how the Iterator pattern is built into the Python language so you never need to implement it by hand.

Python does not have macros like Lisp (Paul Graham's favorite language), so abstracting away the Iterator pattern required changing the language: the `yield` keyword was added in Python 2.2 (2001)[2]. The `yield` keyword allows the construction of generators, which work as iterators.

Every generator is an iterator: generators fully implement the iterator interface. But an iterator — as defined in the *GoF book* — retrieves items from a collection, while a generator can produce items "out of thin air". That's why the Fibonacci sequence generator is a common example: an infinite series of numbers cannot be stored in a collection. However, be aware that the Python community treats *iterator* and *generator* as synonyms most of the time.

[1] From Revenge of the Nerds, a blog post.

[2] Python 2.2 users could use `yield` with the directive `from __future__ import generators`; `yield` became available by default in Python 2.3.

Python 3 uses generators in many places. Even the `range()` built-in now returns a generator-like object instead of full-blown lists like before. If you must build a `list` from `range`, you have to be explicit, e.g. `list(range(100))`.

Every collection in Python is *iterable*, and iterators are used internally to support:

- `for` loops;
- collection types construction and extension;
- looping over text files line by line;
- list, dict and set comprehensions;
- tuple unpacking;
- unpacking actual parameters with `*` in function calls.

This chapter covers the following topics:

- How the `iter(…)` built-in function is used internally to handle iterable objects.
- How to implement the classic Iterator pattern in Python.
- How a generator function works in detail, with line by line descriptions.
- How the classic Iterator can be replaced by a generator function or generator expression.
- Leveraging the general purpose generator functions in the standard library.
- Using the new `yield from` statement to combine generators.
- A case study: using generator functions in a database conversion utility designed to work with large data sets.
- Why generators and coroutines look alike but are actually very different and should not be mixed.

We'll get started studying how the `iter(…)` function makes sequences iterable.

Sentence take #1: a sequence of words

We'll start our exploration of iterables by implementing a `Sentence` class: you give its constructor a string with some text, and then you can iterate word by word. The first version will implement the sequence protocol, and it's iterable because all sequences are iterable, as we've seen before, but now we'll see exactly why.

Example 14-1 shows a `Sentence` class that extracts words from a text by index.

Example 14-1. `sentence.py`: A `Sentence` as a sequence of words.

```
import re
import reprlib

RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)  ❶

    def __getitem__(self, index):
        return self.words[index]  ❷

    def __len__(self):  ❸
        return len(self.words)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)  ❹
```

❶ `re.findall` returns a list with all non-overlapping matches of the regular expression, as a list of strings.

❷ `self.words` holds the result of `.findall`, so we simply return the word at the given index.

❸ To complete the sequence protocol, we implement `__len__` — but it is not needed to make an iterable object.

❹ `reprlib.repr` is a utility function to generate abbreviated string representations of data structures that can be very large[3].

By default, `reprlib.repr` limits the generated string to 30 characters. See the following console session to see how `Sentence` is used:

Example 14-2. Testing iteration on a `Sentence` instance.

```
>>> s = Sentence('"The time has come," the Walrus said,')  # ❶
>>> s
Sentence('"The time ha... Walrus said,')  # ❷
>>> for word in s:  # ❸
...     print(word)
The
time
has
come
```

[3] We first used `reprlib` in "`Vector` take #1: `Vector2d` compatible" on page 278.

```
the
Walrus
said
>>> list(s)  # ❹
['The', 'time', 'has', 'come', 'the', 'Walrus', 'said']
```

❶ A sentence is created from a string.

❷ Note the output of `__repr__` using `...` generated by `reprlib.repr`.

❸ `Sentence` instances are iterable, we'll see why in a moment.

❹ Being iterable, `Sentence` objects can be used as input to build lists and other iterable types.

In the next pages, we'll develop other `Sentence` classes that pass the tests in Example 14-2. However, the implementation in Example 14-1 is different from all the others because it's also a sequence, so you can get words by index:

```
    >>> s[0]
    'The'
    >>> s[5]
    'Walrus'
    >>> s[-1]
    'said'
```

Every Python programmer knows that sequences are iterable. Now we'll see precisely why.

Why sequences are iterable: the `iter` function

Whenever the interpreter needs to iterate over an object `x`, it automatically calls `iter(x)`.

The `iter` built-in function:

1. Checks whether the object implements, `__iter__`, and calls that to obtain an iterator;
2. If `__iter__` is not implemented, but `__getitem__` is implemented, Python creates an iterator that attempts to fetch items in order, starting from index `0` (zero);
3. If that fails, Python raises `TypeError`, usually saying `"'C' object is not iterable"`, where `C` is the class of the target object.

That is why any Python sequence is iterable: they all implement `__getitem__`. In fact, the standard sequences also implement `__iter__`, and yours should too, because the special handling of `__getitem__` exists for backward compatibility reasons and may be gone in the future (although it is not deprecated as I write this).

As mentioned in "Python digs sequences" on page 312, this is an extreme form of duck typing: an object is considered iterable not only when it implements the special method `__iter__`, but also when it implements `__getitem__`, as long as `__getitem__` accepts `int` keys starting from `0`.

In the goose-typing approach, the definition for an iterable is simpler but not as flexible: an object is considered iterable if it implements the `__iter__` method. No subclassing or registration is required, because `abc.Iterable` implements the `__subclasshook__`, as seen in "Geese can behave as ducks" on page 340. Here is a demonstration:

```
>>> class Foo:
...     def __iter__(self):
...         pass
...
>>> from collections import abc
>>> issubclass(Foo, abc.Iterable)
True
>>> f = Foo()
>>> isinstance(f, abc.Iterable)
True
```

However, note that our initial `Sentence` class does not pass the `issubclass(Sentence, abc.Iterable)` test, even though it is iterable in practice.

> As of Python 3.4, the most accurate way to check whether an object `x` is iterable is to call `iter(x)` and handle a `TypeError` exception if it isn't. This is more accurate than using `isinstance(x, abc.Iterable)`, because `iter(x)` also considers the legacy `__getitem__` method, while the `Iterable` ABC does not.

Explicitly checking whether an object is iterable may not be worthwhile if right after the check you are going to iterate over the object. After all, when the iteration is attempted on a non-iterable, the exception Python raises is clear enough: `TypeError: 'C' object is not iterable` . If you can do better than just raising `TypeError`, then do so in a `try/except` block instead of doing an explicit check. The explicit check may make sense if you are holding on to the object to iterate over it later; in this case catching the error early may be useful.

The next section makes explicit the relationship between iterables and iterators.

Iterables versus iterators

From the explanation in "Why sequences are iterable: the `iter` function" on page 406 we can extrapolate a definition:

iterable

Any object from which the `iter` built-in function can obtain an iterator. Objects implementing an `__iter__` method returning an *iterator* are iterable. Sequences are always iterable; so as are objects implementing a `__getitem__` method which takes 0-based indexes.

It's important to be clear about the relationship between iterables and iterators: Python obtains iterators from iterables.

Here is a simple `for` loop iterating over a `str`. The `str` `'ABC'` is the iterable here. You don't see it, but there is an iterator behind the curtain:

```
>>> s = 'ABC'
>>> for char in s:
...     print(char)
...
A
B
C
```

If there was no `for` statement and we had to emulate the `for` machinery by hand with a `while` loop, this is what we'd have to write:

```
>>> s = 'ABC'
>>> it = iter(s)  # ❶
>>> while True:
...     try:
...         print(next(it))  # ❷
...     except StopIteration:  # ❸
...         del it  # ❹
...         break  # ❺
...
A
B
C
```

❶ Build an iterator `it` from the iterable.

❷ Repeatedly call `next` on the iterator to obtain the next item.

❸ The iterator raises `StopIteration` when there are no further items.

❹ Release reference to `it` — the iterator object is discarded.

❺ Exit the loop.

`StopIteration` signals that the iterator is exhausted. This exception is handled internally in `for` loops and other iteration contexts like list comprehensions, tuple unpacking etc.

The standard interface for an iterator has two methods:

`__next__`
: Returns the next available item, raising `StopIteration` when there are no more items.

`__iter__`
: Returns `self`; this allows iterators to be used where an iterable is expected, for example, in a `for` loop.

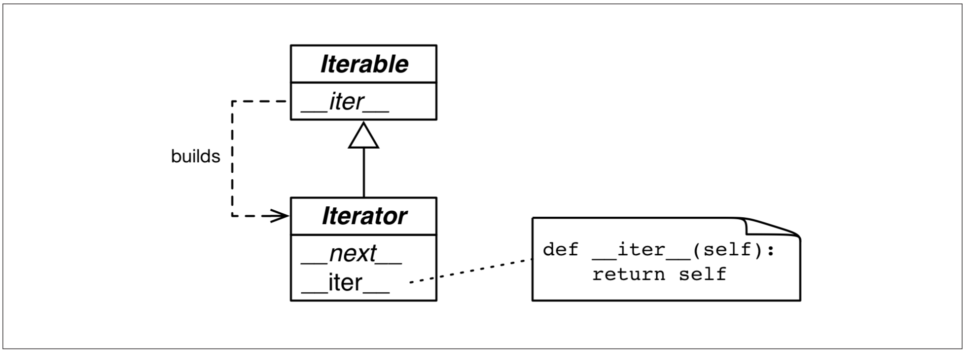


Figure 14-1. The `Iterable` and `Iterator` ABCs. Methods in italic are abstract. A concrete `Iterable.__iter__` should return a new `Iterator` instance. A concrete `Iterator` must implement `__next__`. The `Iterator.__iter__` method just returns the instance itself.

This is formalized in the `collections.abc.Iterator` ABC, which defines the `__next__` abstract method, and subclasses `Iterable`—where the abstract `__iter__` method is defined. See Figure 14-1.

The `Iterator` ABC implements `__iter__` by doing `return self`. This allows an iterator to be used wherever an iterable is required. The source code for `abc.Iterator` is in Example 14-3.

Example 14-3. `abc.Iterator` class; extracted from `Lib/_collections_abc.py`.

```
class Iterator(Iterable):

    __slots__ = ()

    @abstractmethod
    def __next__(self):
        'Return the next item from the iterator. When exhausted, raise StopIteration'
        raise StopIteration

    def __iter__(self):
        return self
    @classmethod
    def __subclasshook__(cls, C):
        if cls is Iterator:
            if (any("__next__" in B.__dict__ for B in C.__mro__) and
                any("__iter__" in B.__dict__ for B in C.__mro__)):
                return True
        return NotImplemented
```

The `Iterator` ABC abstract method is `it.__next__()` in Python 3 and `it.next()` in Python 2. As usual, you should avoid calling special methods directly. Just use the `next(it)`: this built-in function does the right thing in Python 2 and 3.

The Lib/types.py module source code in Python 3.4 has a comment that says:

```
# Iterators in Python aren't a matter of type but of protocol.  A large
# and changing number of builtin types implement *some* flavor of
# iterator.  Don't check the type!  Use hasattr to check for both
# "__iter__" and "__next__" attributes instead.
```

In fact, that's exactly what the `__subclasshook__` method of the `abc.Iterator` ABC does (see Example 14-3).

Taking into account the advice from `Lib/types.py` and the logic implemented in `Lib/_collections_abc.py`, the best way to check if an object `x` is an iterator is to call `isinstance(x, abc.Iterator)`. Thanks to `Iterator.__subclasshook__`, this test works even if the class of `x` is not a real or virtual subclass of `Iterator`.

Back to our `Sentence` class from Example 14-1, you can clearly see how the iterator is built by `iter(…)` and consumed by `next(…)` using the Python console:

```
>>> s3 = Sentence('Pig and Pepper')  # ❶
>>> it = iter(s3)  # ❷
>>> it  # doctest: +ELLIPSIS
<iterator object at 0x...>
>>> next(it)  # ❸
'Pig'
>>> next(it)
'and'
>>> next(it)
'Pepper'
>>> next(it)  # ❹
Traceback (most recent call last):
  ...
StopIteration
>>> list(it)  # ❺
[]
>>> list(iter(s3))  # ❻
['Pig', 'and', 'Pepper']
```

❶ Create a sentence `s3` with 3 words.

❷ Obtain an iterator from `s3`.

❸ `next(it)` fetches the next word.

❹ There are no more words, so the iterator raises a `StopIteration` exception.

❺ Once exhausted, an iterator becomes useless.

❻ To go over the sentence again, a new iterator must be built.

Since the only methods required of an iterator are `__next__` and `__iter__`, there is no way to check whether there are remaining items, other than call `next()` and catch `StopInteration`. Also, it's not possible to "reset" an iterator. If you need to start over, you need to call `iter(…)` on the iterable that built the iterator in the first place. Calling `iter(…)` on the iterator itself won't help, because — as mentioned — `Iterator.__iter__` is implemented by returning `self`, so this will not reset a depleted iterator.

To wrap up this section, here is a definition for *iterator*:

iterator

Any object that implements the `__next__` no-argument method which returns the next item in a series or raises `StopIteration` when there are no more items. Python iterators also implement the `__iter__` method so they are *iterable* as well.

This first version of `Sentence` was iterable thanks to the special treatment the `iter(…)` built-in gives to sequences. Now we'll implement the standard iterable protocol.

Sentence take #2: a classic iterator

The next `Sentence` class is built according to the classic Iterator design pattern following the blueprint in the *GoF book*. Note that this is not idiomatic Python, as the next refactorings will make very clear. But it serves to make explicit the relationship between the iterable collection and the iterator object.

Example 14-4 shows an implementation of a `Sentence` that is iterable because it implements the `__iter__` special method which builds and returns a `SentenceIterator`. This is how the Iterator design pattern is described in the original *Design Patterns* book.

We are doing it this way here just to make clear the crucial distinction between an iterable and an iterator and how they are connected.

Example 14-4. `sentence_iter.py`: `Sentence` implemented using the Iterator pattern.

```
import re
import reprlib
```

```
RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):  ❶
        return SentenceIterator(self.words)  ❷


class SentenceIterator:

    def __init__(self, words):
        self.words = words  ❸
        self.index = 0  ❹

    def __next__(self):
        try:
            word = self.words[self.index]  ❺
        except IndexError:
            raise StopIteration()  ❻
        self.index += 1  ❼
        return word  ❽

    def __iter__(self):  ❾
        return self
```

❶ The `__iter__` method is the only addition to the previous `Sentence` implementation. This version has no `__getitem__`, to make it clear that the class is iterable because it implements `__iter__`.

❷ `__iter__` fulfills the iterable protocol by instantiating and returning an iterator.

❸ `SentenceIterator` holds a reference to the list of words.

❹ `self.index` is used to determine the next word to fetch.

❺ Get the word at `self.index`.

❻ If there is no word at `self.index`, raise `StopIteration`.

❼ Increment `self.index`.

❽ Return the word.

❾ Implement `self.__iter__`.

The code in Example 14-4 passes the tests in Example 14-2.

Note that implementing `__iter__` in `SentenceIterator` is not actually needed for this example to work, but the it's the right thing to do: iterators are supposed to implement both `__next__` and `__iter__`, and doing so makes our iterator pass the `issubclass(SentenceInterator, abc.Iterator)` test. If we had subclassed `SentenceIterator` from `abc.Iterator` we'd inherit the concrete `abc.Iterator.__iter__` method.

That is a lot of work (for us lazy Python programmers, anyway). Note how most code in `SentenceIterator` deals with managing the internal state of the iterator. Soon we'll see how to make it shorter. But first, a brief detour to address an implementation shortcut that may be tempting, but is just wrong.

Making `Sentence` an iterator: bad idea

A common cause of errors in building iterables and iterators is to confuse the two. To be clear: iterables have a `__iter__` method that instantiates a new iterator every time. Iterators implement a `__next__` method that returns individual items, and a `__iter__` method that returns `self`.

Therefore, iterators are also iterable, but iterables are not iterators.

It may be tempting to implement `__next__` in addition to `__iter__` in the `Sentence` class, making each `Sentence` instance at the same time an iterable and iterator over itself. But this is a terrible idea. It's also a common anti-pattern, according to Alex Martelli who has a lot of experience with Python code reviews.

The *Applicability* section[4] of the Iterator design pattern in the *GoF book* says:

> Use the Iterator pattern
>
> - to access an aggregate object's contents without exposing its internal representation.
> - to support multiple traversals of aggregate objects.
> - to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

To "support multiple traversals" it must be possible to obtain multiple independent iterators from the same iterable instance, and each iterator must keep its own internal state, so a proper implementation of the pattern requires each call to `iter(my_iterable)` to create a new, independent, iterator. That is why we need the `SentenceIterator` class in this example.

[4] Gamma et.al., *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 259.

An iterable should never act as an iterator over itself. In other words, iterables must implement `__iter__`, but not `__next__`.

On the other hand, for convenience, iterators should be iterable. An iterator's `__iter__` should just return `self`.

Now that the classic Iterator pattern is properly demonstrated, we can get let it go. The next section presents a more idiomatic implementation of `Sentence`.

Sentence take #3: a generator function

A Pythonic implementation of the same functionality uses a generator function to replace the `SequenceIterator` class. A proper explanation of the generator function comes right after Example 14-5.

Example 14-5. `sentence_gen.py`: `Sentence` implemented using a generator function.

```
import re
import reprlib

RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        for word in self.words:   ❶
            yield word   ❷
        return   ❸

# done! ❹
```

❶ Iterate over `self.word`.

❷ Yield the current `word`.

❸ This `return` is not needed; the function can just “fall-through” and return automatically. Either way, a generator function doesn’t raise `StopIteration`: it simply exits when it’s done producing values[5]

❹ No need for a separate iterator class!

Here again we have a different implementation of `Sentence` that passes the tests in Example 14-2.

Back in the `Sentence` code in Example 14-4, `__iter__` called the `SentenceIterator` constructor to build an iterator and return it. Now the iterator in Example 14-5 is in fact a generator object, built automatically when the `__iter__` method is called, because `__iter__` here is a generator function.

A full explanation of generator functions follows.

How a generator function works

Any Python function that has the `yield` keyword in its body is a generator function: a function which, when called, returns a generator object. In other words, a generator function is a generator factory.

The only syntax distinguishing a plain function from a generator function is the fact that the latter has a `yield` keyword somewhere in its body. Some argued that a new keyword like `gen` should be used for generator functions instead of `def`, but Guido did not agree. His arguments are in PEP 255 — Simple Generators[6].

Here is the simplest function useful to demonstrate the behavior of a generator[7]:

```
>>> def gen_123():  # ❶
...     yield 1  # ❷
...     yield 2
...     yield 3
...
>>> gen_123  # doctest: +ELLIPSIS
<function gen_123 at 0x...>  # ❸
```

[5] When reviewing this code, Alex Martelli suggested the body of this method could simply be `return iter(self.words)`. He is correct, of course: the result of calling `__iter__` would also be an iterator, as it should be. However, I used a `for` loop with `yield` here to introduce the syntax of a generator function, which will be covered in detail in the next section.

[6] Sometimes I add a `gen` prefix or suffix when naming generator functions, but this is not a common practice. And you can’t do that if you’re implementing an iterable, of course: the necessary special method must be named `__iter__`.

[7] Thanks to David Kwast for suggesting this example.

```
>>> gen_123()   # doctest: +ELLIPSIS
<generator object gen_123 at 0x...>  # ❹
>>> for i in gen_123():  # ❺
...     print(i)
1
2
3
>>> g = gen_123()  # ❻
>>> next(g)  # ❼
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)  # ❽
Traceback (most recent call last):
  ...
StopIteration
```

❶ Any Python function that contains the `yield` keyword is a generator function.

❷ Usually the body of a generator function has loop, but not necessarily; here I just repeat `yield` three times.

❸ Looking closely, we see `gen_123` is a function object.

❹ But when invoked, `gen_123()` returns a generator object.

❺ Generators are iterators that produce the values of the expressions passed to `yield`.

❻ For closer inspection, we assign the generator object to `g`.

❼ Since `g` is an iterator, calling `next(g)` fetches the next item produced by `yield`.

❽ When the body of the function completes, the generator object raises a `StopIteration`.

A generator function builds a generator object which wraps the body of the function. When we invoke `next(…)` on the generator object, execution advances to the next `yield` in the function body, and the `next(…)` call evaluates to the value yielded when the function body is suspended. Finally, when the function body returns, the enclosing generator object raises `StopIteration`, in accordance with the `Iterator` protocol.

I find it helpful to be strict when talking about the results obtained from a generator: I say that a generator *yields* or *produces* values. But it’s confusing to say a generator “returns” values. Functions return values. Calling a generator function returns a generator. A generator yields or produces values. A generator doesn’t “return” values in the usual way: the `return` statement in the body of a generator function causes `StopIteration` to be raised by the generator object[8].

The following example makes the interaction between a `for` loop and the body of the function more explicit:

Example 14-6. A generator function which prints messages when it runs.

```
>>> def gen_AB():  # ❶
...     print('start')
...     yield 'A'       # ❷
...     print('continue')
...     yield 'B'       # ❸
...     print('end.')   # ❹
...
>>> for c in gen_AB():  # ❺
...     print('-->', c)  # ❻
...
start     ❼
--> A    ❽
continue ❾
--> B    ❿
end.      ⓫
>>>   ⓬
```

❶ The generator function is defined like any function, but uses `yield`.

❷ The first implicit call to `next()` in the `for` loop at ❺ will print `'start'` and stop at the first `yield`, producing the value `'A'`.

❸ The second implicit call to `next()` in the `for` loop will print `'continue'` and stop at the second `yield`, producing the value `'B'`.

❹ The third call to `next()` will print `'end.'` and fall through the end of the function body, causing the generator object to raise `StopIteration`.

[8] Prior to Python 3.3 it was an error to provide a value with the `return` statement in a generator function. Now that is legal, but the `return` still causes a `StopIteration` exception to be raised. The caller can retrieve the return value from from the exception object. However, this is only relevant when using a generator function as a coroutine, as we’ll see in “Returning a value from a coroutine” on page 477.

❺ To iterate, the `for` machinery does the equivalent of `g = iter(gen_AB())` to get a generator object, and then `next(g)` at each iteration.

❻ The loop block prints `-->` and the value returned by `next(g)`. But this output will be seen only after the output of the `print` calls inside the generator function.

❼ The string `'start'` appears as a result of `print('start')` in the generator function body.

❽ `yield 'A'` in the generator function body produces the value *A* consumed by the `for` loop, which gets assigned to the `c` variable and results in the output `--> A`.

❾ Iteration continues with a second call `next(g)`, advancing the generator function body from `yield 'A'` to `yield 'B'`. The text `continue` is output because of the second `print` in the generator function body.

❿ `yield 'B'` produces the value *B* consumed by the `for` loop, which gets assigned to the `c` loop variable, so the loop prints `--> B`.

⓫ Iteration continues with a third call `next(it)`, advancing to the end of the body of the function. The text `end.` appears in the output because of the third `print` in the generator function body.

⓬ When the generator function body runs to the end, the generator object raises `StopIteration`. The `for` loop machinery catches that exception, and the loop terminates cleanly.

Now hopefully it's clear how `Sentence.__iter__` in Example 14-5 works: `__iter__` is generator function which, when called, builds a generator object which implements the iterator interface, so the `SentenceIterator` class is no longer needed.

This second version of `Sentence` is much shorter than the first, but it's not as lazy as it could be. Nowadays, laziness is considered a good trait, at least in programming languages and APIs. A lazy implementation postpones producing values to the last possible moment. This saves memory and may avoid useless processing as well.

We'll build a lazy `Sentence` class next.

Sentence take #4: a lazy implementation

The `Iterator` interface is designed to be lazy: `next(my_iterator)` produces one item at a time. The opposite of lazy is eager: lazy evaluation and eager evaluation are actual technical terms in programming language theory.

Our `Sentence` implementations so far have not been lazy because the `__init__` eagerly builds a list of all words in the text, binding it to the `self.words` attribute. This will entail processing the entire text, and the list may use as much memory as the text itself (probably more; it depends on how many non-word characters are in the text). Most of this work will be in vain if the user only iterates over the first couple of words.

Whenever you are using Python 3 and start wondering "Is there a lazy way of doing this?", often the answer is "Yes".

The `re.finditer` function is a lazy version of `re.findall` which, instead of a list, returns a generator producing `re.MatchObject` instances on demand. If there are many matches, `re.finditer` saves a lot of memory. Using it, our third version of `Sentence` is now lazy: it only produces the next word when it is needed. The code is in Example 14-7

Example 14-7. `sentence_gen2.py`: `Sentence` implemented using a generator function calling the `re.finditer` generator function.

```
import re
import reprlib

RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
        self.text = text   ❶

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        for match in RE_WORD.finditer(self.text):   ❷
            yield match.group()   ❸
```

❶ No need to have a `words` list.

❷ `finditer` builds an iterator over the matches of `RE_WORD` on `self.text`, yielding `MatchObject` instances.

❸ `match.group()` extracts the actual matched text from the `MatchObject` instance.

Generator functions are an awesome shortcut, but the code can be made even shorter with a generator expression.

Sentence take #5: a generator expression

Simple generator functions like the one in the previous `Sentence` class (Example 14-7) can be replaced by a generator expression.

A generator expression can be understood as a lazy version of a list comprehension: it does not eagerly build a list, but returns a generator that will lazily produce the items on demand. In other words, if a list comprehension is a factory of lists, a generator expression is a factory of generators.

Example 14-8 is a quick demo of a generator expression, comparing it to a list comprehension:

Example 14-8. The `gen_AB` generator function is used by a list comprehension, then by a generator expression.

```
>>> def gen_AB():  # ❶
...     print('start')
...     yield 'A'
...     print('continue')
...     yield 'B'
...     print('end.')
...
>>> res1 = [x*3 for x in gen_AB()]  # ❷
start
continue
end.
>>> for i in res1:  # ❸
...     print('-->', i)
...
--> AAA
--> BBB
>>> res2 = (x*3 for x in gen_AB())  # ❹
>>> res2  # ❺
<generator object <genexpr> at 0x10063c240>
>>> for i in res2:  # ❻
...     print('-->', i)
...
start
--> AAA
continue
--> BBB
end.
```

❶ This is the same `gen_AB` function from Example 14-6.

❷ The list comprehension eagerly iterates over the items yielded by the generator object produced by calling `gen_AB()`: `'A'` and `'B'`. Note the output in the next lines: `start`, `continue`, `end.`

❸ This `for` loop is iterating over the the `res1 list` produced by the list comprehension.

❹ The generator expression returns `res2`. The call to `gen_AB()` is made, but that call returns a generator which is not consumed here.

❺ `res2` is a generator object.

❻ Only when the `for` loop iterates over `res2`, the body of `gen_AB` actually executes. Each iteration of the `for` loop implicitly calls `next(res2)`, advancing `gen_AB` to the next `yield`. Note the output of `gen_AB` with the output of the `print` in the `for` loop.

So, a generator expression produces a generator, and we can use it to further reduce the code in the `Sentence` class. See Example 14-9.

Example 14-9. `sentence_genexp.py`: `Sentence` implemented using a generator expression.

```
import re
import reprlib

RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
        self.text = text

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        return (match.group() for match in RE_WORD.finditer(self.text))
```

The only difference from Example 14-7 is the `__iter__` method which here is not a generator function (it has no `yield`) but uses a generator expression to build a generator and then returns it. The end result is the same: the caller of `__iter__` gets a generator object.

Generator expressions are syntactic sugar: they can always be replaced by generator functions, but sometimes are more convenient. The next section is about generator expression usage.

Generator expressions: when to use them

I used several generator expressions when implementing the `Vector` class in Example 10-16. Each of the methods `__eq__`, `__hash__`, `__abs__`, `angle`, `angles`, `format`, `__add__` and `__mul__` has a generator expression. In all those methods, a list comprehension would also work, at the cost of using more memory to store the intermediate list values.

In Example 14-9 we saw that a generator expression is a syntactic short cut to create a generator without defining and calling a function. On the other hand, generator functions are much more flexible: you can code complex logic with multiple statements, and can even use them as *coroutines* (See chapter XREF).

For the simpler cases, a generator expression will do, and it's easier to read at a glance, as the `Vector` example shows.

My rule of thumb in choosing the syntax to use is simple: if the generator expression spans more than a couple of lines, I prefer to code a generator function for the sake of readability. Also, because generator functions have a name, they can be reused. You can always name a generator expression and use it later by assigning it to a variable, of course, but that is stretching its intended usage as a one-off generator.

Syntax tip

When a generator expression is passed as the single argument to a function or constructor, you don't need to write a set of parenthesis for the function call and another to enclose the generator expression. A single pair will do, like in the `Vector` call from the `__mul__` method in Example 10-16, reproduced below. However, if there are more function arguments after the generator expression you need to enclose it in parenthesis to avoid a `SyntaxError`.

```
def __mul__(self, scalar):
    if isinstance(scalar, numbers.Real):
        return Vector(n * scalar for n in self)
    else:
        return NotImplemented
```

The `Sentence` examples we've seen exemplify the use of generators playing the role of classic iterators: retrieving items from a collection. But generators can also be used to produce values independent of a data source. The next section shows an example of that.

Another example: arithmetic progression generator

The classic Iterator pattern is all about traversal: navigating some data structure. But a standard interface based on a method to fetch the next item in a series is also useful when the items are produced on the fly, instead of retrieved from a collection. For example, the `range` built-in generates a bounded arithmetic progression (AP) of integers, and the `itertools.count` function generates a boundless AP.

We'll cover `itertools.count` in the next section, but what if you need to generate a bounded AP of numbers of any type?

Example 14-10 shows a few console tests of an `ArithmeticProgression` class we will see in a moment. The signature of the constructor in Example 14-10 is `ArithmeticProgression(begin, step[, end])`. The `range()` function is similar to the `ArithmeticProgression` here, but its full signature is `range(start, stop[, step])`. I chose to implement a different signature because for an arithmetic progression the `step` is mandatory but `end` is optional. I also changed the argument names from `start/stop` to `begin/end` to make it very clear that I opted for a different signature. In each test in Example 14-10 I call `list()` on the result to inspect the generated values.

Example 14-10. Demonstration of an `ArithmeticProgression` class.

```
>>> ap = ArithmeticProgression(0, 1, 3)
>>> list(ap)
[0, 1, 2]
>>> ap = ArithmeticProgression(1, .5, 3)
>>> list(ap)
[1.0, 1.5, 2.0, 2.5]
>>> ap = ArithmeticProgression(0, 1/3, 1)
>>> list(ap)
[0.0, 0.3333333333333333, 0.6666666666666666]
>>> from fractions import Fraction
>>> ap = ArithmeticProgression(0, Fraction(1, 3), 1)
>>> list(ap)
[Fraction(0, 1), Fraction(1, 3), Fraction(2, 3)]
>>> from decimal import Decimal
>>> ap = ArithmeticProgression(0, Decimal('.1'), .3)
>>> list(ap)
[Decimal('0.0'), Decimal('0.1'), Decimal('0.2')]
```

Note that type of the numbers in the resulting arithmetic progression follows the type of `begin` or `step`, according to the numeric coercion rules of Python arithmetic. In Example 14-10 you see lists of `int`, `float`, `Fraction` and `Decimal` numbers.

Example 14-11 lists the implementation of the `ArithmeticProgression` class.

Example 14-11. The `ArithmeticProgression` class.

```
class ArithmeticProgression:

    def __init__(self, begin, step, end=None):   ❶
        self.begin = begin
        self.step = step
        self.end = end  # None -> "infinite" series

    def __iter__(self):
        result = type(self.begin + self.step)(self.begin)   ❷
        forever = self.end is None   ❸
        index = 0
        while forever or result < self.end:   ❹
            yield result   ❺
            index += 1
            result = self.begin + self.step * index  ❻
```

❶ `__init__` requires two arguments: `begin` and `step`. `end` is optional, it it's `None`, the series will be unbounded.

❷ This line produces a `result` value equal to `self.begin`, but coerced to the type of the subsequent additions[9].

❸ For readability, the `forever` flag will be `True` if the `self.end` attribute is `None`, resulting in an unbounded series.

❹ This loop runs `forever` or until the result matches or exceeds `self.end`. When this loop exits, so does the function.

❺ The current `result` is produced.

❻ The next potential result is calculated. It may never be yielded, because the `while` loop may terminate.

Note in the last line of Example 14-11 that, instead of simply incrementing the `result` with `self.step` iteratively, I opted to use an `index` variable and calculate each `result` by adding `self.begin` to `self.step` multiplied by `index` to reduce the cumulative effect of errors when working with with floats.

The `ArithmeticProgression` class from Example 14-11 works as intended, and is a clear example of the use of a generator function to implement the `__iter__` special method. However, if the whole point of a class is to build a generator by implementing `__iter__`, the class can be reduced to a generator function. A generator function is, after all, a generator factory.

Example 14-12 shows a generator function that does the same job as `ArithmeticProgression` with less code. The tests in Example 14-10 all pass if you just call `aritprog_gen` instead of `ArithmeticProgression` [10].

Example 14-12. The `aritprog_gen` generator function.

```
def aritprog_gen(begin, step, end=None):
    result = type(begin + step)(begin)
    forever = end is None
    index = 0
```

[9] In Python 2 there was a `coerce()` built-in function but it's gone in Python 3, deemed unnecessary because the numeric coercion rules are implicit in the arithmetic operator methods. So the best way I could think of to coerce the initial value to be of the same type as the rest of the series was to perform the addition and use its type to convert the result. I asked about this in the python-list and got an excellent response from Steven D'Aprano.

[10] The `14-it-generator/` directory in the *Fluent Python* code repository includes doctests and a script, `aritprog_runner.py`, which runs the tests against all variations of the `aritprog*.py` scripts.

```
    while forever or result < end:
        yield result
        index += 1
        result = begin + step * index
```

Example 14-12 is pretty cool, but always remember: there are plenty of ready-to-use generators in the standard library, and the next section will show an even cooler implementation using the `itertools` module.

Arithmetic progression with `itertools`

The `itertools` module in Python 3.4 has 19 generator functions that can be combined in a variety of interesting ways.

For example, the `itertools.count` function returns a generator that produces numbers. Without arguments, it produces a series of integers starting with `0`. But you can provide optional `start` and `step` values to achieve a result very similar to our `aritprog_gen` functions.

```
>>> import itertools
>>> gen = itertools.count(1, .5)
>>> next(gen)
1
>>> next(gen)
1.5
>>> next(gen)
2.0
>>> next(gen)
2.5
```

However, `itertools.count` never stops, so if you call `list(count())`, Python will try to build a `list` larger than available memory and your machine will be very grumpy long before the call fails.

On the other hand, there is the `itertools.takewhile` function: it produces a generator which consumes another generator and stops when a given predicate evaluates to `False`. So we can combine the two and write this:

```
>>> gen = itertools.takewhile(lambda n: n < 3, itertools.count(1, .5))
>>> list(gen)
[1, 1.5, 2.0, 2.5]
```

Leveraging `takewhile` and `count`, Example 14-13 is sweet and short.

Example 14-13. `aritprog_v3.py`: this works like the previous `aritprog_gen` functions.

```
import itertools


def aritprog_gen(begin, step, end=None):
    first = type(begin + step)(begin)
    ap_gen = itertools.count(first, step)
    if end is not None:
        ap_gen = itertools.takewhile(lambda n: n < end, ap_gen)
    return ap_gen
```

Note that `aritprog_gen` is not a generator function in Example 14-13: it has no `yield` in its body. But it returns a generator, so it operates as a generator factory, just as generator function does.

The point of Example 14-13 is: when implementing generators, know what is available in the standard library, otherwise there's a good chance you'll reinvent the wheel. That's why the next section covers several ready-to-use generator functions.

Generator functions in the standard library

The standard library provides many generators, from plain text file objects providing line by line iteration, to the awesome `os.walk` function which yields file names while traversing a directory tree, making recursive file system searches as simple as a `for` loop.

The `os.walk` generator function is impressive, but in this section I want to focus on general purpose functions that take arbitrary iterables as arguments and return generators that produce selected, computed or rearranged items. In the following tables I summarize two dozen of them, from the built-in, `itertools` and `functools` modules. For convenience, I grouped them by high-level functionality, regardless of where they are defined.

> Perhaps you know all the functions mentioned in this section, but some of them are underused, so a quick overview may be good to recall what's already available.

The first group are filtering generator functions: they yield a subset of items produced by the input iterable, without changing the items themselves. We used `itertools.takewhile` previously in this chapter, in "Arithmetic progression with `itertools`" on page 425. Like `takewhile`, most functions listed in Table 14-1 take a `predicate` which is a one-argument boolean function that will be applied to each item in the input to determine whether the item is included in the output.

Table 14-1. Filtering generator functions

module	function	description
`itertools`	`compress(it, selector_it)`	consumes two iterables in parallel; yields items from `it` whenever the corresponding item in `selector_it` is truthy

module	function	description
itertools	dropwhile(predicate, it)	consumes it skipping items while predicate computes truthy, then yields every remaining item (no further checks are made)
(built-in)	filter(predicate, it)	applies predicate to each item of iterable, yielding the item if predicate(item) is truthy; if predicate is None, only truthy items are yielded
itertools	filterfalse(predicate, it)	same as filter, with the predicate logic negated: yields items whenever predicate computes falsy
itertools	islice(it, stop) or islice(it, start, stop, step=1)	yields items from a slice of it, similar to s[:stop] or s[start:stop:step] except it can be any iterable, and the operation is lazy
itertools	takewhile(predicate, it)	yields items while predicate computes truthy, then stops and no further checks are made

The console listing in Example 14-14 shows the use of all functions in Table 14-1.

Example 14-14. Filtering generator functions examples

```
>>> def vowel(c):
...     return c.lower() in 'aeiou'
...
>>> list(filter(vowel, 'Aardvark'))
['A', 'a', 'a']
>>> import itertools
>>> list(itertools.filterfalse(vowel, 'Aardvark'))
['r', 'd', 'v', 'r', 'k']
>>> list(itertools.dropwhile(vowel, 'Aardvark'))
['r', 'd', 'v', 'a', 'r', 'k']
>>> list(itertools.takewhile(vowel, 'Aardvark'))
['A', 'a']
>>> list(itertools.compress('Aardvark', (1,0,1,1,0,1)))
['A', 'r', 'd', 'a']
>>> list(itertools.islice('Aardvark', 4))
['A', 'a', 'r', 'd']
>>> list(itertools.islice('Aardvark', 4, 7))
['v', 'a', 'r']
>>> list(itertools.islice('Aardvark', 1, 7, 2))
['a', 'd', 'a']
```

The next group are the mapping generators: they yield items computed from each individual item in the input iterable — or iterables, in the case of map and starmap [11]. The generators in Table 14-2 yield one result per item in the input iterables. If the input comes from more than one iterable, the output stops as soon as the first input iterable is exhausted.

[11] Here the term “mapping” is unrelated to dictionaries, but has to do with the map built-in.

Table 14-2. Mapping generator functions

module	function	description
`itertools`	`accumulate(it, [func])`	yields accumulated sums; if `func` is provided, yields the result of applying it the first pair of items, then to the first result and next item etc.
(built-in)	`enumerate(iterable, start=0)`	yields 2-tuples of the form `(index, item)`, where `index` is counted from `start`, and `item` is taken from the `iterable`
(built-in)	`map(func, it1, [it2, …, itN])`	applies `func` to each item of `it`, yielding the result; if N iterables are given, `func` must take N arguments and the iterables will be consumed in parallel
`itertools`	`starmap(func, it)`	applies `func` to each item of `it`, yielding the result; the input iterable should yield iterable items `iit`, and `func` is applied as `func(*iit)`

Example 14-15 demonstrates some uses of `itertools.accumulate`.

Example 14-15. `itertools.accumulate` generator function examples.

```
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> import itertools
>>> list(itertools.accumulate(sample))  # ❶
[5, 9, 11, 19, 26, 32, 35, 35, 44, 45]
>>> list(itertools.accumulate(sample, min))  # ❷
[5, 4, 2, 2, 2, 2, 2, 0, 0, 0]
>>> list(itertools.accumulate(sample, max))  # ❸
[5, 5, 5, 8, 8, 8, 8, 8, 9, 9]
>>> import operator
>>> list(itertools.accumulate(sample, operator.mul))  # ❹
[5, 20, 40, 320, 2240, 13440, 40320, 0, 0, 0]
>>> list(itertools.accumulate(range(1, 11), operator.mul))
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]  # ❺
```

❶ Running sum.

❷ Running minimum.

❸ Running maximum.

❹ Running product.

❺ Factorials from `1!` to `10!`.

The remaining functions of Table 14-2 are shown in Example 14-16.

Example 14-16. Mapping generator function examples

```
>>> list(enumerate('albatroz', 1))  # ❶
[(1, 'a'), (2, 'l'), (3, 'b'), (4, 'a'), (5, 't'), (6, 'r'), (7, 'o'), (8, 'z')]
>>> import operator
>>> list(map(operator.mul, range(11), range(11)))  # ❷
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> list(map(operator.mul, range(11), [2, 4, 8]))  # ❸
[0, 4, 16]
>>> list(map(lambda a, b: (a, b), range(11), [2, 4, 8]))  # ❹
```

```
[(0, 2), (1, 4), (2, 8)]
>>> import itertools
>>> list(itertools.starmap(operator.mul, enumerate('albatroz', 1)))  # ❺
['a', 'll', 'bbb', 'aaaa', 'ttttt', 'rrrrrr', 'ooooooo', 'zzzzzzzz']
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> list(itertools.starmap(lambda a, b: b/a,
...     enumerate(itertools.accumulate(sample), 1)))  # ❻
[5.0, 4.5, 3.6666666666666665, 4.75, 5.2, 5.333333333333333,
5.0, 4.375, 4.888888888888889, 4.5]
```

❶ Number the letters in the word, starting from `1`.

❷ Squares of integers from `0` to `10`.

❸ Multiplying numbers from two iterables in parallel: results stop when the shortest iterable ends.

❹ This is what the `zip` built-in function does.

❺ Repeat each letter in the word according to its place in it, starting from `1`.

❻ Running average.

Next is the group of merging generators — all of these yield items from multiple input iterables. The `chain` and `chain.from_iterable` consume the input iterables sequentially (one after the other), while `product`, `zip` and `zip_longest` consume the input iterables in parallel. See Table 14-3.

Table 14-3. Generator functions that merge multiple input iterables.

module	function	description
`itertools`	`chain(it1, …, itN)`	yield all items from `it1`, then from `it2` etc., seamlessly
`itertools`	`chain.from_iterable(it)`	yield all items from each iterable produced by `it`, one after the other, seamlessly; `it` should yield iterable items, for example, a list of iterables
`itertools`	`product(it1, …, itN, repeat=1)`	cartesian product: yields N-tuples made by combining items from each input iterable like nested `for` loops could produce; `repeat` allows the input iterables to be consumed more than once
(built-in)	`zip(it1, …, itN)`	yields N-tuples built from items taken from the iterables in parallel, silently stopping when the first iterable is exhausted
`itertools`	`zip_longest(it1, …, itN, fillvalue=None)`	yields N-tuples built from items taken from the iterables in parallel, stopping only when the last iterable is exhausted, filling the blanks with the `fillvalue`

Example 14-17 shows the use of the `itertools.chain` and `zip` generator functions and their siblings. Recall that the `zip` function is named after the zip fastener or zipper (no relation with compression). Both `zip` and `itertools.zip_longest` were introduced in "The awesome `zip`" on page 295.

Example 14-17. Merging generator function examples

```
>>> list(itertools.chain('ABC', range(2)))  # ❶
['A', 'B', 'C', 0, 1]
>>> list(itertools.chain(enumerate('ABC')))  # ❷
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(itertools.chain.from_iterable(enumerate('ABC')))  # ❸
[0, 'A', 1, 'B', 2, 'C']
>>> list(zip('ABC', range(5)))  # ❹
[('A', 0), ('B', 1), ('C', 2)]
>>> list(zip('ABC', range(5), [10, 20, 30, 40]))  # ❺
[('A', 0, 10), ('B', 1, 20), ('C', 2, 30)]
>>> list(itertools.zip_longest('ABC', range(5)))  # ❻
[('A', 0), ('B', 1), ('C', 2), (None, 3), (None, 4)]
>>> list(itertools.zip_longest('ABC', range(5), fillvalue='?'))  # ❼
[('A', 0), ('B', 1), ('C', 2), ('?', 3), ('?', 4)]
```

❶ `chain` is usually called with two or more iterables.

❷ `chain` does nothing useful when called with a single iterable.

❸ But `chain.from_iterable` takes each item from the iterable, and chains them in sequence, as long as each item is itself iterable.

❹ `zip` is commonly used to merge two iterables into a series of two-tuples.

❺ Any number of iterables can be consumed by `zip` in parallel, but the generator stops as soon as the first iterable ends.

❻ `itertools.zip_longest` works like `zip`, except it consumes all input iterables to the end, padding output tuples with `None` as needed.

❼ The `fillvalue` keyword argument specifies a custom padding value.

The `itertools.product` generator is a lazy way of computing cartesian products, which we built using list comprehensions with more than one `for` clause in "Cartesian products" on page 23. Generator expressions with multiple `for` clauses can also be used to produce cartesian products lazily. Example 14-18 demonstrates `itertools.product`.

Example 14-18. `itertools.product` generator function examples

```
>>> list(itertools.product('ABC', range(2)))  # ❶
[('A', 0), ('A', 1), ('B', 0), ('B', 1), ('C', 0), ('C', 1)]
>>> suits = 'spades hearts diamonds clubs'.split()
>>> list(itertools.product('AK', suits))  # ❷
[('A', 'spades'), ('A', 'hearts'), ('A', 'diamonds'), ('A', 'clubs'),
('K', 'spades'), ('K', 'hearts'), ('K', 'diamonds'), ('K', 'clubs')]
>>> list(itertools.product('ABC'))  # ❸
[('A',), ('B',), ('C',)]
>>> list(itertools.product('ABC', repeat=2))  # ❹
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'),
('B', 'C'), ('C', 'A'), ('C', 'B'), ('C', 'C')]
>>> list(itertools.product(range(2), repeat=3))
```

```
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0),
(1, 0, 1), (1, 1, 0), (1, 1, 1)]
>>> rows = itertools.product('AB', range(2), repeat=2)
>>> for row in rows: print(row)
...
('A', 0, 'A', 0)
('A', 0, 'A', 1)
('A', 0, 'B', 0)
('A', 0, 'B', 1)
('A', 1, 'A', 0)
('A', 1, 'A', 1)
('A', 1, 'B', 0)
('A', 1, 'B', 1)
('B', 0, 'A', 0)
('B', 0, 'A', 1)
('B', 0, 'B', 0)
('B', 0, 'B', 1)
('B', 1, 'A', 0)
('B', 1, 'A', 1)
('B', 1, 'B', 0)
('B', 1, 'B', 1)
```

❶ The cartesian product of a `str` with three characters and a `range` with two integers yields six tuples (because `3 * 2` is `6`).

❷ The product of two card ranks (*AK*), and four suits is a series of eight tuples.

❸ Given a single iterable, `product` yields a series of one-tuples, not very useful.

❹ The `repeat=N` keyword argument tells product to consume each input iterable `N` times.

Some generator functions expand the input by yielding more than one value per input item. They are listed in Table 14-4.

Table 14-4. Generator functions that expand each input item into multiple output items.

module	function	description
`itertools`	`combinations(it, out_len)`	yield combinations of `out_len` items from the items yielded by `it`
`itertools`	`combinations_with_replacement(it, out_len)`	yield combinations of `out_len` items from the items yielded by `it`, including combinations with repeated items.
`itertools`	`count(start=0, step=1)`	yields numbers starting at `start`, incremented by `step`, indefinitely
`itertools`	`cycle(it)`	yields items from `it` storing a copy of each, then yields the entire sequence repeatedly, indefinitely
`itertools`	`permutations(it, out_len=None)`	yield permutations of `out_len` items from the items yielded by `it`; by default, `out_len` is `len(list(it))`

module	function	description
itertools	repeat(item, [times])	yield the given item repeadedly, indefinetly unless a number of times is given

The count and repeat functions from itertools return generators that conjure items out of nothing: neither of them takes an iterable as input. We saw itertools.count in "Arithmetic progression with itertools" on page 425. The cycle generator makes a backup of the input iterable and yields its items repeatedly. Examples with count, repeat and cycle are in Example 14-19.

Example 14-19. count and repeat

```
>>> ct = itertools.count()  # ❶
>>> next(ct)  # ❷
0
>>> next(ct), next(ct), next(ct)  # ❸
(1, 2, 3)
>>> list(itertools.islice(itertools.count(1, .3), 3))  # ❹
[1, 1.3, 1.6]
>>> cy = itertools.cycle('ABC')  # ❺
>>> next(cy)
'A'
>>> list(itertools.islice(cy, 7))  # ❻
['B', 'C', 'A', 'B', 'C', 'A', 'B']
>>> rp = itertools.repeat(7)  # ❼
>>> next(rp), next(rp)
(7, 7)
>>> list(itertools.repeat(8, 4))  # ❽
[8, 8, 8, 8]
>>> list(map(operator.mul, range(11), itertools.repeat(5)))  # ❾
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

❶ Build a count generator ct.

❷ Retrieve the first item from ct.

❸ I can't build a list from ct, because ct never stops, so I fetch the next three items.

❹ I can build a list from a count generator if it is limited by islice or takewhile.

❺ Build a cycle generator from 'ABC' and fetch its first item, 'A'.

❻ A list can only be built if limited by islice; the next seven items are retrieved here.

❼ Build a repeat generator that will yield the number 7 forever.

❽ A repeat generator can be limited by passing the times argument: here the number 8 will be produced 4 times.

❾ A common use of `repeat`: providing a fixed argument in `map`; here it provides the `5` multiplier.

The `combinations`, `combinations_with_replacement` and `permutations` generator functions — together with `product` — are called the *combinatoric generators* in the `itertools` documentation page[12]. There is a close relationship between `itertools.product` and the remaining *combinatoric* functions as well, as Example 14-20 shows.

Example 14-20. Combinatoric generator functions yield multiple values per input item.

```
>>> list(itertools.combinations('ABC', 2))  # ❶
[('A', 'B'), ('A', 'C'), ('B', 'C')]
>>> list(itertools.combinations_with_replacement('ABC', 2))  # ❷
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
>>> list(itertools.permutations('ABC', 2))  # ❸
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
>>> list(itertools.product('ABC', repeat=2))  # ❹
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'), ('B', 'C'),
('C', 'A'), ('C', 'B'), ('C', 'C')]
```

❶ All combinations of `len()==2` from the items in `'ABC'`; item ordering in the generated tuples is irrelevant (they could be sets).

❷ All combinations of `len()==2` from the items in `'ABC'`, including combinations with repeated items.

❸ All permutations of `len()==2` from the items in `'ABC'`; item ordering in the generated tuples is relevant.

❹ Cartesian product from `'ABC'` and `'ABC'` (that's the effect of `repeat=2`).

The last group of generator functions we'll cover in this section are designed to yield all items in the input iterables, but rearranged in some way. Here are two functions that return multiple generators: `itertools.groupby` and `itertools.tee`. The other generator function in this group, the `reversed` built-in, is the only one covered in this section that does not accept any iterable as input, but only sequences. This makes sense: since `reversed` will yield the items from last to first, it only works with a sequence with a known length. But it avoids the cost of making a reversed copy of the sequence by yielding each item as needed.

[12] I put the `itertools.product` function together with the *merging* generators in Table 14-3 because they all consume more than one iterable, while the generators in Table 14-4 all accept at most one input iterable.

Table 14-5. Rearranging generator functions.

module	function	description
`itertools`	`groupby(it, key=None)`	yields 2-tuples of the form `(key, group)`, where `key` is the grouping criterion and `group` is a generator yielding the items in the group
(built-in)	`reversed(seq)`	yields items from `seq` in reverse order, from last to first; `seq` must be a sequence or implement the `__reversed__` special method.
`itertools`	`tee(it, n=2)`	yields a tuple of N generators, each yielding the items of the input iterable independently

Demonstrations for `itertools.groupby` and the `reversed` built-in are in Example 14-21. Note that `itertools.groupby` assumes that the input iterable is sorted by the grouping criterion, or at least that the items are clustered by that criterion — even if not sorted.

Example 14-21. `itertools.groupby`

```
>>> list(itertools.groupby('LLLLAAGGG'))  # ❶
[('L', <itertools._grouper object at 0x102227cc0>),
('A', <itertools._grouper object at 0x102227b38>),
('G', <itertools._grouper object at 0x102227b70>)]
>>> for char, group in itertools.groupby('LLLLAAAGG'):  # ❷
...     print(char, '->', list(group))
...
L -> ['L', 'L', 'L', 'L']
A -> ['A', 'A',]
G -> ['G', 'G', 'G']
>>> animals = ['duck', 'eagle', 'rat', 'giraffe', 'bear',
...            'bat', 'dolphin', 'shark', 'lion']
>>> animals.sort(key=len)  # ❸
>>> animals
['rat', 'bat', 'duck', 'bear', 'lion', 'eagle', 'shark',
'giraffe', 'dolphin']
>>> for length, group in itertools.groupby(animals, len):  # ❹
...     print(length, '->', list(group))
...
3 -> ['rat', 'bat']
4 -> ['duck', 'bear', 'lion']
5 -> ['eagle', 'shark']
7 -> ['giraffe', 'dolphin']
>>> for length, group in itertools.groupby(reversed(animals), len): # ❺
...     print(length, '->', list(group))
...
7 -> ['dolphin', 'giraffe']
5 -> ['shark', 'eagle']
4 -> ['lion', 'bear', 'duck']
3 -> ['bat', 'rat']
>>>
```

❶ `groupby` yields tuples of `(key, group_generator)`.

❷ Handling `groupby` generators involves nested iteration: in this case the outer `for` loop and the inner `list` constructor.

❸ To use `groupby` the input should be sorted; here the words are sorted by length.

❹ Again, the `for` loop over the `key` and `group` pair, to display the `key` and expand the `group` into a `list`.

❺ Here the `reverse` generator is used to iterate over `animals` from right to left.

The last of the generator functions in this group is `iterator.tee`, which has a unique behavior: it yields multiple generators from a single input iterable, each yielding every item from the input. Those generators can be consumed independently, as shown in Example 14-22.

Example 14-22. `itertools.tee` yields multiple generators, each yielding every item of the input generator.

```
>>> list(itertools.tee('ABC'))
[<itertools._tee object at 0x10222abc8>, <itertools._tee object at 0x10222ac08>]
>>> g1, g2 = itertools.tee('ABC')
>>> next(g1)
'A'
>>> next(g2)
'A'
>>> next(g2)
'B'
>>> list(g1)
['B', 'C']
>>> list(g2)
['C']
>>> list(zip(*itertools.tee('ABC')))
[('A', 'A'), ('B', 'B'), ('C', 'C')]
```

Note that several examples in this section used combinations of generator functions. This is a great feature of these functions: since they all take generators as arguments and return generators, they can be combined in many of ways.

While on the subject of combining generators, the `yield from` statement, new in Python 3.3, is a tool for doing just that.

New syntax in Python 3.3: `yield from`

Nested for loops are the traditional solution when a generator function needs to yield values produced from another generator.

For example, here is a homemade implementation of a chaining generator[13]:

```
>>> def chain(*iterables):
...     for it in iterables:
...         for i in it:
...             yield i
...
>>> s = 'ABC'
>>> t = tuple(range(3))
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

The `chain` generator function is delegating to each received iterable in turn. PEP 380 — Syntax for Delegating to a Subgenerator introduced new syntax for doing that, shown in the next console listing:

```
>>> def chain(*iterables):
...     for i in iterables:
...         yield from i
...
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

As you can see, `yield from i` replaces the inner `for` loop completely. The use of `yield from` in this example is correct, and the code reads better, but it seems like mere syntactic sugar. Besides replacing a loop, `yield from` creates a channel connecting the inner generator directly to the client of the outer generator. This channel becomes really important when generators are used as coroutines and not only produce but also consume values from the client code. Chapter 16 dives into coroutines, and has several pages explaining why `yield from` is much more than syntactic sugar.

After this first encounter with `yield from`, we'll go back to our review of iterable-savvy functions in the standard library.

Iterable reducing functions

The functions in Table 14-6 all take an iterable and return a single result. They are known as "reducing", "folding" or "accumulating" functions. Actually, every one of the built-ins listed here can be implemented with `functools.reduce`, but they exist as built-ins because they address some common use cases more easily. Also, in the case of `all` and `any`, there is an important optimization that can't be done with `reduce`: these functions short-circuit, that is, they stop consuming the iterator as soon as the result is determined. See the last test with `any` in Example 14-23.

[13] The `itertools.chain` from the standard library is written in C.

Table 14-6. Built-in functions that read iterables and return single values

module	function	description
(built-in)	`all(it)`	returns `True` if all items in `it` are truthy, otherwise `False`; `all([])` returns `True`
(built-in)	`any(it)`	returns `True` if any item in `it` is truthy, otherwise `False`; `any([])` returns `False`
(built-in)	`max(it, [key=,] [default=])`	returns the maximum value of the items in `it`[a]; `key` is an ordering function, as in `sorted`; `default` is returned if the iterable is empty.
(built-in)	`min(it, [key=,] [default=])`	returns the minimum value of the items in `it`[b]. `key` is an ordering function, as in `sorted`; `default` is returned if the iterable is empty.
`functools`	`reduce(func, it, [initial])`	returns the result of applying `func` to the first pair of items, then to that result and the third item and so on; if given, `initial` forms the initial pair with the first item.
(built-in)	`sum(it, start=0)`	the sum of all items in `it`, with the optional `start` value addedfootnote:[use `math.fsum` for better precision when adding floats

[a] may also be called as `max(arg1, arg2, …, [key=?])`, in which case the maximum among the arguments is returned.

[b] may also be called as `min(arg1, arg2, …, [key=?])`, in which case the minimum among the arguments is returned.

The operation of `all` and `any` is exemplified in Example 14-23

Example 14-23. Results of `all` and `any` for some sequences.

```
>>> all([1, 2, 3])
True
>>> all([1, 0, 3])
False
>>> all([])
True
>>> any([1, 2, 3])
True
>>> any([1, 0, 3])
True
>>> any([0, 0.0])
False
>>> any([])
False
>>> g = (n for n in [0, 0.0, 7, 8])
>>> any(g)
True
>>> next(g)
8
```

A longer explanation about `functools.reduce` appeared in "Vector take #4: hashing and a faster ==" on page 290.

Another built-in that takes an iterable and returns something else is `sorted`. Unlike `reversed`, which is a generator function, `sorted` builds and returns an actual list. After all, every single item of the input iterable must be read so they can be sorted, and the sorting happens in a `list`, therefore `sorted` just returns that `list` after it's done. I mention `sorted` here because it does consume an arbitrary iterable.

Of course, `sorted` and the reducing functions only work with iterables that eventually stop. Otherwise, they will keep on collecting items and never return a result.

We'll now go back to the `iter()` built-in: it has a little known feature that we haven't covered yet.

A closer look at the `iter` function

As we´ve seen, Python calls `iter(x)` when it needs to iterate over an object `x`.

But `iter` has another trick: it can be called with two arguments to create an iterator from a regular function or any callable object. In this usage, the first argument must be a callable to be invoked repeatedly (with no arguments) to yield values, and the second argument is a sentinel: a marker value which, when returned by the callable, causes the iterator to raise `StopIteration` instead of yielding the sentinel.

The following example shows how to use `iter` to roll a 6-sided die until a `1` is rolled:

```
>>> def d6():
...     return randint(1, 6)
...
>>> d6_iter = iter(d6, 1)
>>> d6_iter
<callable_iterator object at 0x00000000029BE6A0>
>>> for roll in d6_iter:
...     print(roll)
...
4
3
6
3
```

Note that the `iter` function here returns a `callable_iterator`. The `for` loop in the example may run for a very long time, but it will never display `1`, because that is the sentinel value. As usual with iterators, the `d6_iter` object in the example becomes useless once exhausted. To start over, you must rebuild the iterator by invoking `iter(…)` again.

A useful example is found in the `iter` built-in function documentation. This snippet reads lines from a file until a blank line is found or the end of file is reached:

```
    with open('mydata.txt') as fp:
        for line in iter(fp.readline, ''):
            process_line(line)
```

To close this chapter, I present a practical example of using generators to handle a large volume of data efficiently.

Case study: generators in a database conversion utility

A few years ago I worked at BIREME, a digital library run by PAHO/WHO (Pan-American Health Organization/World Health Organization) in São Paulo, Brazil. Among the bibliographic data sets created by BIREME are LILACS (Latin American and Caribbean Health Sciences index) and SciELO (Scientific Electronic Library Online), two comprehensive databases indexing the scientific and technical literature produced in the region.

Since the late 1980's the database system used to manage LILACS is CDS/ISIS, a non-relational, document database created by UNESCO and eventually rewritten in C by BIREME to run on GNU/Linux servers. One of my jobs was to research alternatives for a possible migration of LILACS — and eventually the much larger SciELO — to a modern, open-source, document database such as CouchDB or MongoDB.

As part of that research I wrote a Python script, `isis2json.py` that reads a CDS/ISIS file and writes a JSON file suitable for importing to CouchDB or MongoDB. Initially, the script read files in the ISO-2709 format exported by CDS/ISIS. The reading and writing had to be done incrementally because the full data sets were much bigger than main memory. That was easy enough: each iteration of the main `for` loop read one record from the `.iso` file, massaged it and wrote it to the `.json` output.

However, for operational reasons it was deemed necessary that `isis2json.py` supported another CDS/ISIS data format: the binary `.mst` files used in production at BIREME — to avoid the costly export to ISO-2709.

Now I had a problem: the libraries used to read ISO-2709 and `.mst` files had very different APIs. And the JSON writing loop was already complicated because the script accepted a variety of command-line options to restructure each output record. Reading data using two different APIs in the same `for` loop where the JSON was produced would be unwieldy.

The solution was to isolate the reading logic into a pair of generator functions: one for each supported input format. In the end, `isis2json.py` script was split into four functions. You can see the main Python 2 script in Example A-5, but the full source code with dependencies is in fluentpython/isis2json on Github.

Here is a high-level overview of how the script is structured:

`main`
: The `main` function uses `argparse` to read command-line options that configure the structure of the output records. Based on the input filename extension, a suitable generator function is selected to read the data and yield the records, one by one.

`iter_iso_records`
: This generator function reads `.iso` files (assumed to be in the ISO-2709 format). It takes two arguments: the filename and `isis_json_type`, one of the options related to the record structure. Each iteration of its `for` loop reads one record, creates an empty `dict`, populates it with field data, and yields the `dict`.

`iter_mst_records`
: This other generator functions reads `.mst` files[14]. If you look at the source code for `isis2json.py` you'll see that its not as simple as `iter_iso_records` but its interface and overall structure is the same: it takes a filename and an `isis_json_type` argument and enters a `for` loop which builds and yields one `dict` per iteration, representing a single record.

`write_json`
: This function performs the actual writing of the JSON records, one at a time. It takes numerous arguments, but the first one — `input_gen` — is a reference to a generator function: either `iter_iso_records` or `iter_mst_records`. The main `for` loop in `write_json` iterates over the dictionaries yielded by the selected `input_gen` generator, massages it in several ways as determined by the command-line options, and appends the JSON record to the output file.

Leveraging generator functions I was able to decouple the reading logic from the writing logic. Of course, the simplest way to decouple them would be to read all records to memory, then write them to disk. But that was not a viable option because of the size of the data sets. Using generators, the reading and writing is interleaved, so the script can process files of any size.

Now if `isis2json.py` needs to support an additional input format — say, MARCXML, a DTD used by the US Library of Congress to represent ISO-2709 data — it will be easy to add a third generator function to implement the reading logic, without changing anything in the complicated `write_json` function.

This is not rocket science, but it's a real example where generators provided a flexible solution to processing databases as a stream of records, keeping memory usage low

[14] The library used to read the complex `.mst` binary is actually written in Java, so this functionality is only available when `isis2json.py` is executed with the Jython interpreter, version 2.5 or newer. See the README.rst file in the repository for details. The dependencies are imported inside the generator functions that need them, so the script can run even if only one of the external libraries is available.

regardless of the amount of data. Anyone who manages large data sets finds many opportunities for using generators in practice.

The next section addresses an aspect of generators that we'll actually skip for now. Read on to understand why.

Generators as coroutines

About five years after generator functions with the `yield` keyword were introduced in Python 2.2, PEP 342 — Coroutines via Enhanced Generators was implemented in Python 2.5. This proposal added extra methods and functionality to generator objects, most notably the `.send()` method.

Like `.__next__()`, `.send()` causes the generator to advance to the next `yield`, but it also allows the client using the generator to send data into it: whatever argument is passed to `.send()` becomes the value of the corresponding `yield` expression inside the generator function body. In other words, `.send()` allows two-way data exchange between the client code and the generator — in contrast with `.__next__()` which only lets the client receive data from the generator.

This is such a major "enhancement" that it actually changes the nature of generators: when used in this way, they become *coroutines*. David Beazley — probably the most prolific writer and speaker about coroutines in the Python community — warned in a famous PyCon US 2009 tutorial:

> - Generators produce data for iteration
> - Coroutines are consumers of data
> - To keep your brain from exploding, you don't mix the two concepts together
> - Coroutines are not related to iteration
> - Note: There is a use of having yield produce a value in a coroutine, but it's not tied to iteration[15].
>
> — David Beazley
> *A curious course on coroutines and concurrency*

I will follow Dave's advice and close this chapter — which is really about iteration techniques — without touching `send` and the other features that make generators usable as coroutines. Coroutines will be covered in XXXREf.

[15] Slide 33, *Keeping it Straight* in A curious course on coroutines and concurrency

Chapter summary

Iteration is so deeply embedded in the language that I like to say that Python groks iterators[16]. The integration of the Iterator pattern in the semantics of Python is a prime example of how design patterns are not equally applicable in all programming languages. In Python, a classic iterator implemented "by hand" as in Example 14-4 has no practical use, except as a didactic example.

In this chapter we built a few versions of a class to iterate over individual words in text files that may be very long. Thanks to the use of generators, the successive refactorings of the `Sentence` class become shorter and easier to read — when you know how they work.

We then coded a generator of arithmetic progressions and showed how to leverage the `itertools` module to make it simpler. An overview of 24 general-purpose generator functions in the standard library followed.

In two sections we looked at the `iter` built-in function: first to see how it returns an iterator when called as `iter(o)`, then to study how it builds an iterator from any function when called as `iter(func, sentinel)`.

For practical context, I described the implementation of a database conversion utility using generator functions to decouple the reading to the writing logic, enabling efficient handling of large data sets and making it easy to support more than one data input format.

Also mentioned in this chapter were the `yield from` syntax, new in Python 3.3, and coroutines. Both topics ware just introduced here; they get more coverage later in the book.

Further reading

A detailed technical explanation of generators appears in the Yield expressions section of the Python Language Reference. The PEP where generator functions where defined is PEP 255 — Simple Generators.

The `itertools` module documentation is excellent because of all the examples included. Although the functions in that module are implemented in C, the documentation shows how many of them would be written in Python, often by leveraging other functions in the module. The usage examples are also great: for instance, there is a snippet showing how to use the `accumulate` function amortize a loan with interest, given a list of pay-

[16] According to the Jargon file, to *grok* is not merely to learn something, but to absorb it so "it becomes part of you, part of your identity".

ments over time. There is also a Itertools Recipes with additional high-performance functions which use the `itertools` functions as building blocks.

Chapter 4 — Iterators and Generators — of the *Python Cookbook, 3rd. edition* (O'Reilly, 2013), by David Beazley and Brian K. Jones, has 16 recipes covering this subject from many different angles, always focusing on practical applications.

The `yield from` syntax is explained with examples in *What's New In Python 3.3*, section PEP 380: Syntax for Delegating to a Subgenerator. We'll also cover it in detail in "Using `yield from`" on page 479 and "The meaning of `yield from`" on page 485 in Chapter 16.

If you are interested in document databases and would like to learn more about the context of "Case study: generators in a database conversion utility" on page 439, the Code4Lib journal — which covers the intersection between libraries and technology — published my paper From ISIS to CouchDB: Databases and Data Models for Bibliographic Records. One section of the paper describes the `isis2json.py` script. The rest of it explains why and how the semi-structured data model implemented by document databases like CouchDB and MongoDB are more suitable for cooperative bibliographic data collection than the relational model.

Soapbox

Generator function syntax: more sugar would be nice

> Designers need to ensure that controls and displays for different purposes are significantly different from one another.
>
> — Donald Norman
> *The Design of Everyday Things*

Source code plays the role of "controls and displays" in programming languages. I think Python is exceptionally well designed, its source code is often as readable as pseudocode. But nothing is perfect. Guido van Rossum should have followed Donald Norman's advice quoted above and introduced another keyword for defining generator expressions, instead of reusing `def`. The *BDFL Pronouncements* section of PEP 255 — Simple Generators actually argues:

> A "yield" statement buried in the body is not enough warning that the semantics are so different.

But Guido hates introducing new keywords and he did not find that argument convincing, so we are stuck with `def`.

Reusing the function syntax for generators has other bad consequences. In the paper and experimental work *Python, the full monty: A Tested Semantics for the Python Pro-*

gramming Language, Politz[17] et. al. show this trivial example of a generator function (section 4.1 of the paper):

```
    def f(): x=0
        while True:
            x += 1
            yield x
```

The authors then make the point that we can't abstract the process of yielding with a function call:

Example 14-24. "[This] seems to perform a simple abstraction over the process of yielding" (Politz et.al.)

```
def f():
    def do_yield(n):
        yield n
    x = 0
    while True:
        x += 1
        do_yield(x)
```

If we call `f()` in Example 14-24 we get an infinite loop, and not a generator, because the `yield` keyword only makes the immediately enclosing function a generator function. Although generator functions look like functions, we cannot delegate to another generator function with a simple function call. As a point of comparison, the Lua language does not impose this limitation. A Lua coroutine can call other functions and any of them can yield to the original caller.

The new `yield from` syntax was introduced to allow a Python generator or coroutine to delegate to work to another, without requiring the work-around of an inner `for` loop. Example 14-24 can be "fixed" by prefixing the function call with `yield from`:

Example 14-25. This actually performs a simple abstraction over the process of yielding.

```
def f():
    def do_yield(n):
        yield n
    x = 0
    while True:
        x += 1
        yield from do_yield(x)
```

Reusing `def` for declaring generators was a usability mistake, and the problem was compounded in Python 2.5 with coroutines, which are also coded as functions with

[17] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: the full monty. SIGPLAN Not. 48, 10 (October 2013), 217-232.

yield. In the case of coroutines the yield just happens to appear — usually — on the right-hand side of an assignment, because it receives the argument of the .send() call from the client. As David Beazley says:

> Despite some similarities, generators and coroutines are basically two different concepts[18].

I believe coroutines also deserved their own keyword. As we'll see later, coroutines are often used with special decorators which do set them apart from other functions. But generator functions are not decorated as frequently, so we have to scan their bodies for yield to realize they are not functions at all, but a completely different beast.

It can be argued that, since those features were made to work with little additional syntax, extra syntax would be merely "syntactic sugar". I happen to like syntactic sugar when it makes features that are different look different. The lack of syntactic sugar is the main reason why Lisp code is hard to read: every language construct in Lisp looks like a function call.

Semantics of generator versus iterator

There are at least three ways of thinking about the relationship between iterators and generators.

The first is the interface viewpoint. The Python iterator protocol defines two methods: __next__ and __iter__. Generator objects implement both, so from this perspective every generator is an iterator. By this definition, objects created by the enumerate() built-in are iterators:

```
>>> from collections import abc
>>> e = enumerate('ABC')
>>> isinstance(e, abc.Iterator)
True
```

The second is the implementation viewpoint. From this angle, a generator is a Python language construct that can be coded in two ways: as a function with the yield keyword or as a generator expression. The generator objects resulting from calling a generator function or evaluating a generator expression are instances of an internal GeneratorType. From this perspective every generator is also an iterator, because GeneratorType instances implement the iterator interface. But you can write an iterator that is not a generator — by implementing the classic Iterator pattern, as we saw in Example 14-4, or by coding an extension in C. The enumerate objects are not generators from this perspective.

```
>>> import types
>>> e = enumerate('ABC')
>>> isinstance(e, types.GeneratorType)
False
```

[18] Slide 31, A curious course on coroutines and concurrency.

This happens because `types.GeneratorType` is defined as “The type of generator-iterator objects, produced by calling a generator function.”

The third is the conceptual viewpoint. In the classic Iterator design pattern — as defined in the *GoF book* — the iterator traverses a collection and yields items from it. The iterator may be quite complex, for example, it may navigate through a tree-like data structure. But, however much logic is in a classic iterator, it always reads values from an existing data source, and when you call `next(it)`, the iterator is not expected to change the item it gets from the source, it’s supposed to just yield it as is.

In contrast, a generator may produce values without necessarily traversing a collection, like `range` does. And even if attached to a collection, generators are not limited to yielding just the items in it, but may yield some other values derived from them. A clear example of this is the `enumerate` function. By the original definition of the design pattern, the generator returned by `enumerate` is not an iterator because it creates the tuples it yields.

At this conceptual level, the implementation technique is irrelevant. You can write a generator without using a Python generator object. Here is a Fibonacci generator I wrote just to make this point:

Example 14-26. `fibo_by_hand.py`: Fibonacci generator without `GeneratorType` instances

```
class Fibonacci:

    def __iter__(self):
        return FibonacciGenerator()


class FibonacciGenerator:

    def __init__(self):
        self.a = 0
        self.b = 1

    def __next__(self):
        result = self.a
        self.a, self.b = self.b, self.a + self.b
        return result

    def __iter__(self):
        return self
```

Example 14-26 works but is just a silly example. Here is the Pythonic Fibonacci generator:

```
    def fibonacci():
        a, b = 0, 1
        while True:
            yield a
            a, b = b, a + b
```

And of course, you can always use the generator language construct to perform the basic duties of an iterator: traversing a collection and yielding items from it.

In reality, Python programmers are not strict about this distinction: generators are also called iterators, even in the official docs. The canonical definition of an iterator in the Python Glossary is so general it encompasses both iterators and generators:

> Iterator: An object representing a stream of data. […]

The full definition of *iterator* in the Python Glossary is worth reading. On the other hand, the definition of *generator* there treats *iterator* and *generator* as synonyms, and uses the word “generator” to refer both to the generator function and the generator object it builds. So, in the Python community lingo, iterator and generator are fairly close synonyms.

The minimalistic Iterator interface in Python

In the *Implementation* section of the Iterator pattern[19], the *Gang of Four* wrote:

> The minimal interface to Iterator consists of the operations First, Next, IsDone, and CurrentItem.

However, that very sentence has a footnote which reads:

> We can make this interface even smaller by merging Next, IsDone, and CurrentItem into a single operation that advances to the next object and returns it. If the traversal is finished, then this operation returns a special value (0, for instance) that marks the end of the iteration.

This is close to what we have in Python: the single method `__next__` does the job. But instead of using a sentinel which could be overlooked by mistake, the `StopIteration` exception signals the end of the iteration. Simple and correct: that’s the Python way.

[19] Gamma et.al., *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 261.

CHAPTER 15

Context managers and else blocks

> Context managers may end up being almost as important as the subroutine itself. We've only scratched the surface with them. […] Basic has a with statement, there are with statements in lots of languages. But they don't do the same thing, they all do something very shallow, they save you from repeated dotted [attribute] lookups, they don't do setup and tear down. Just because it's the same name don't think it's the same thing. The with statement is a very big deal[1]
>
> — Raymond Hettinger
> *eloquent Python evangelist*

The subjects of this chapter are control flow features that are not so common in other languages, and for this reason tend do be overlooked or underused in Python.

They are:

- The with statement and context managers.
- The else clause in for, while and try statements.

The with statement sets up a temporary context and reliably tears it down, under the control of a context manager object. This prevents errors and reduces boilerplate code, making APIs at the same time safer and easier to use. Python programmers are finding lots of uses for with blocks beyond automatic file closing.

The else clause is completely unrelated to with. But this is **Part V — Control Flow** of the book, and I couldn't find another place for covering else, and I wouldn't have a 1-page chapter about it, so here it is.

Let's go over the smaller topic to get to the real substance of this chapter.

[1] PyCon US 2013 keynote: What Makes Python Awesome; the part about with starts at 23'00" and ends at 26'15".

Do this, then that: `else` blocks beyond `if`

This is no secret, but it is an under-appreciated language feature: the `else` clause can be used not only in `if` statements but also in `for`, `while` and `try` statements.

The semantics of `for/else`, `while/else` and `try/else` are closely related, but very different from `if/else`. Initially the word `else` actually hindered my understanding of these features, but eventually I got used to it.

Here are the rules:

`for`

The `else` block will run only if and when the `for` loop runs to completion; i.e. not if the `for` is aborted with a `break`.

`while`

The `else` block will run only if and when the `while` loop exits because the condition became *falsy*; i.e. not when the `while` is aborted with a `break`.

`try`

The `else` block will only run if no exception is raised in the `try` block. The official docs also state: "Exceptions in the `else` clause are not handled by the preceding `except` clauses."

In all cases, the `else` clause is also skipped if an exception or a `return`, `break` or `continue` statement causes control to jump out of the main block of the compound statement.

I think `else` is a very poor choice for the keyword in all cases except `if`. It implies an excluding alternative, like "Run this loop, otherwise do that", but the semantics for `else` in loops is the opposite: "Run this loop, then do that". This suggests `then` as a better keyword — which would also make sense in the `try` context: "Try this, then do that." However, adding a new keyword is a breaking change to the language, and Guido avoids it like the plague.

Using `else` with these statements often makes the code easier to read and saves the trouble of setting up control flags or adding extra `if` statements.

The use of `else` in loops generally follows the pattern of this snippet:

```
for item in my_list:
    if item.flavor == 'banana':
        break
else:
    raise ValueError('No banana flavor found!')
```

In the case of try/except blocks, else may seem redundant at first. After all, the after_call() below will only run if the dangerous_call() does not raise an exception, correct?

```
try:
    dangerous_call()
    after_call()
except OSError:
    log('OSError...')
```

However, doing so puts the after_call() inside the try block for no good reason. For clarity and correctness, the body of a try block should only have the statements that may generate the expected exceptions. This is much better:

```
try:
    dangerous_call()
except OSError:
    log('OSError...')
else:
    after_call()
```

Now it's clear that the try block is guarding against possible errors in dangerous_call() and not in after_call(). It's also more obvious that after_call() will only execute if no exceptions are raised in the try block.

In Python, try/except is commonly used for control flow, and not just for error handling. There's even an acronym/slogan for that documented in the official Python glossary:

> *EAFP*
>
> Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many try and except statements. The technique contrasts with the *LBYL* style common to many other languages such as C.

The glossary then defines LBYL:

> *LBYL*
>
> Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the *EAFP* approach and is characterized by the presence of many if statements. In a multi-threaded environment, the LBYL approach can risk introducing a race condition between "the looking" and "the leaping". For example, the code, if key in mapping: return mapping[key] can fail if another thread removes key from mapping after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

Given the EAFP style, it makes even more sense to know and use well else blocks in try/except statements.

Now let's address the main topic of this chapter: the powerful with statement.

Context managers and with blocks

Context manager objects exist to control a with statement, just like iterators exist to control a for statement.

The with statement was designed to simplify the try/finally pattern which guarantees that some operation is performed after a block of code, even if the block is aborted because of an exception, a return or sys.exit() call. The code in the finally clause usually releases a critical resource or restores some previous state that was temporarily changed.

The context manager protocol consists of the __enter__ and __exit__ methods. At the start of the with, __enter__ is invoked on the context manager object. The role of the finally clause is played by a call to __exit__ on the context manager object at the end of the with block.

The most common example is making sure a file object is closed. See Example 15-1 for a detailed demonstration of using with to close a file.

Example 15-1. Demonstration of a file object as a context manager.

```
>>> with open('mirror.py') as fp:  # ❶
...     src = fp.read(60)  # ❷
...
>>> len(src)
60
>>> fp  # ❸
<_io.TextIOWrapper name='mirror.py' mode='r' encoding='UTF-8'>
>>> fp.closed, fp.encoding  # ❹
(True, 'UTF-8')
>>> fp.read(60)  # ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

❶ fp is bound to the opened file because the file's __enter__ method returns self.

❷ Read some data from fp.

❸ The fp variable is still available[2].

❹ You can read the attributes of the fp object.

❺ But you can't perform I/O with fp because at the end of the with block, the TextIOWrapper.__exit__ method is called and closes the file.

[2] with blocks don't define a new scope, as functions and modules do.

The first callout in Example 15-1 makes a subtle but crucial point: the context manager object is the result of evaluating the expression after `with`, but the value bound to the target variable (in the `as` clause) is the result of calling `__enter__` on the context manager object.

It just happens that in Example 15-1 ,the `open()` function returns an instance of `TextIOWrapper`, and its `__enter__` method returns `self`. But the `__enter__` method may also return some other object instead of the context manager.

When control flow exits the `with` block in any way, the `__exit__` method is invoked on the context manager object, not on whatever is returned by `__enter__`.

The `as` clause of the `with` statement is optional. In the case of `open` you'll always need it to get a reference to the file, but some context managers return `None` because they have no useful object to give back to the user.

Example 15-2 shows the operation of a perfectly frivolous context manager designed to highlight the distinction between the context manager and the object returned by its `__enter__` method.

Example 15-2. Test driving the `LookingGlass` context manager class.

```
>>> from mirror import LookingGlass
>>> with LookingGlass() as what:  ❶
...      print('Alice, Kitty and Snowdrop')  ❷
...      print(what)
...
pordwonS dna yttiK ,ecilA  ❸
YKCOWREBBAJ
>>> what  ❹
'JABBERWOCKY'
>>> print('Back to normal.')  ❺
Back to normal.
```

❶ The context manager is an instance of `LookingGlass`; Python calls `__enter__` on the context manager and the result is bound to `what`.

❷ Print a `str`, then the value of the target variable `what`

❸ The output of each `print` comes out backwards.

❹ Now the `with` block is over. We can see that the value returned by `__enter__`, held in `what`, is the string `'JABBERWOCKY'`.

❺ Program output is no longer backwards.

Example 15-3 shows the implementation of `LookingGlass`.

Example 15-3. `mirror.py`: code for the `LookingGlass` context manager class.

```
class LookingGlass:

    def __enter__(self):  ❶
        import sys
        self.original_write = sys.stdout.write  ❷
        sys.stdout.write = self.reverse_write  ❸
        return 'JABBERWOCKY'  ❹

    def reverse_write(self, text):  ❺
        self.original_write(text[::-1])

    def __exit__(self, exc_type, exc_value, traceback):  ❻
        import sys  ❼
        sys.stdout.write = self.original_write  ❽
        if exc_type is ZeroDivisionError:  ❾
            print('Please DO NOT divide by zero!')
            return True  ❿
        ⓫
```

❶ Python invokes `__enter__` with no arguments besides `self`.

❷ Hold the original `sys.stdout.write` method in an instance attribute for later use.

❸ Monkey-patch `sys.stdout.write`, replacing it with our own method.

❹ Return the `'JABBERWOCKY'` string just so we have something to put in the target variable `what`.

❺ Our replacement to `sys.stdout.write` reverses the `text` argument and calls the original implementation.

❻ Python calls `__exit__` with `None, None, None` if all went well; if an exception is raised, the three arguments get the exception data, as described below.

❼ It's cheap to import modules again because Python caches them.

❽ Restore the original method to `sys.stdout.write`.

❾ If the exception is not `None` and its type is `ZeroDivisionError`, print a message…

❿ …and return `True` to tell the interpreter that the exception was handled.

⓫ If `__exit__` returns `None` or anything but `True`, any exception raised in the `with` block will be propagated.

When real applications take over standard output they often want to replace `sys.stdout` with another file-like object for a while, then switch back to the original. The `contextlib.redirect_stdout` context manager does exactly that: just pass it the file-like object that will stand in for `sys.stdout`.

The interpreter calls the `__enter__` method with no arguments — beyond the implicit `self`. The three arguments passed to `__exit__` are:

`exc_type`
: The exception class, e.g. `ZeroDivisionError`.

`exc_value`
: The exception instance. Sometimes, parameters passed to the exception constructor — such as the error message — can be found in `exc_value.args`.

`traceback`
: A `traceback` object[3].

For a detailed look at how a context manager works, see Example 15-4 where `LookingGlass` is used outside of a `with` block, so we can manually call its `__enter__` and `__exit__` methods.

Example 15-4. Exercising `LookingGlass` without a `with` block.

```
>>> from mirror import LookingGlass
>>> manager = LookingGlass()  ❶
>>> manager
<mirror.LookingGlass object at 0x2a578ac>
>>> monster = manager.__enter__()  ❷
>>> monster == 'JABBERWOCKY'  ❸
eurT
>>> monster
'YKCOWREBBAJ'
>>> manager
>ca875a2x0 ta tcejbo ssalGgnikooL.rorrim<
>>> manager.__exit__(None, None, None)  ❹
>>> monster
'JABBERWOCKY'
```

❶ Instantiate and inspect the `manager` instance.

❷ Call the context manager `__enter__()` method and store result in `monster`.

❸ Monster is the string `'JABBERWOCKY'`. The `True` identifier appears reversed because all output via `stdout` goes through the `write` method we patched in `__enter__`.

❹ Call `manager.__exit__` to restore previous `stdout.write`.

[3] The three arguments received by `self` are exactly what you get if you call `sys.exc_info()` in the `finally` block of a `try/finally` statement. This makes sense, considering that the `with` statement is meant to replace most uses of `try/finally`, and calling `sys.exc_info()` was often necessary to determine what clean up action would be required.

Context managers are a fairly novel feature and slowly but surely the Python community is finding new, creative uses for them. Some examples from the standard library are:

- Managing transactions in the `sqlite3` module; see Using the connection as a context manager.
- Holding locks, conditions and semaphores in `threading` code; see Using locks, conditions, and semaphores in the with statement.
- Setting up environments for arithmetic operations with `Decimal` objects; see the `decimal.localcontext` documentation.
- Applying temporary patches to objects for testing; see the `unittest.mock.patch` function.

The standard library also includes the `contextlib` utilities, covered next.

The `contextlib` utilities

Before rolling your own context manager classes, take a look at the documentation for contextlib — Utilities for with-statement contexts in the Python standard library. Besides the already mentioned `redirect_stdout`, the `contextlib` module includes classes and other functions that are more widely applicable:

`closing`
: A function to build context managers out of objects that provide a `close()` method but don't implement the `__enter__`/`__exit__` protocol.

`suppress`
: A context manager to temporarily ignore specified exceptions.

`@contextmanager`
: A decorator which lets you build a context manager from a simple generator function, instead of creating a class and implementing the protocol.

`ContextDecorator`
: A base class for defining class-based context managers that can also be used as function decorators, running the entire function within a managed context.

`ExitStack`
: A context manager that lets you enter a variable number of context managers. When the `with` block ends, `ExitStack` calls the stacked context managers `__exit__` methods in LIFO order (last entered, first exited). Use this class when you don't know beforehand how many context managers you need to enter in your `with` block. For example when opening all files from an arbitrary list of files at the same time.

The most widely used of these utilities is surely the `@contextmanager` decorator, so it deserves more attention. That decorator is also intriguing because it shows a use for the `yield` statement unrelated to iteration. This paves the way to the concept of a coroutine, the theme of the next chapter.

Using @contextmanager

The `@contextmanager` decorator reduces the boilerplate of creating a context manager: instead of writing a whole class with `__enter__`/`__exit__` methods, you just implement a generator with a single `yield` that should produce whatever you want the `__enter__` method to return.

In a generator decorated with `@contextmanager`, `yield` is used to split the body of the function in two parts: everything before the `yield` will be executed at the beginning of the `while` block when the interpreter calls `__enter__`; the code after `yield` will run when `__exit__` is called at the end of the block.

Here is an example: Example 15-5 replaces the `LookingGlass` class from Example 15-3 with a generator function.

Example 15-5. `mirror_gen.py`: a context manager implemented with a generator.

```
import contextlib


@contextlib.contextmanager   ❶
def looking_glass():
    import sys
    original_write = sys.stdout.write   ❷

    def reverse_write(text):   ❸
        original_write(text[::-1])

    sys.stdout.write = reverse_write   ❹
    yield 'JABBERWOCKY'   ❺
    sys.stdout.write = original_write   ❻
```

❶ Apply the `contextmanager` decorator.

❷ Preserve original `sys.stdout.write` method.

❸ Define custom `reverse_write` function; `original_write` will be available in the closure.

❹ Replace `sys.stdout.write` with `reverse_write`.

❺ Yield the value that will be bound to the target variable in the `as` clause of the `with` statement. This function pauses at this point while the body of the `with` executes.

❻ When control exits the `with` block in any way, execution continues after the `yield`; here the original `sys.stdout.write` is restored.

Example 15-6 shows the `looking_glass` function in operation.

Example 15-6. Test driving the `looking_glass` context manager function.

```
>>> from mirror_gen import looking_glass
>>> with looking_glass() as what:  ❶
...      print('Alice, Kitty and Snowdrop')
...      print(what)
...
pordwonS dna yttiK ,ecilA
YKCOWREBBAJ
>>> what
'JABBERWOCKY'
```

❶ The only difference from Example 15-2 is the name of the context manager: `looking_glass` instead of `LookingGlass`.

Essentially the `contextlib.contextmanager` decorator wraps the function in a class which implements the `__enter__` and `__exit__` methods[4].

The `__enter__` method of that class:

1. Invokes the generator function and holds on to the generator object — let's call it `gen`.
2. Calls `next(gen)` to make it run to the `yield` keyword.
3. Returns the value yielded by `next(gen)`, so it can be bound to a target variable in the `with/as` form.

When the `with` block terminates, the `__exit__` method:

1. Checks an exception was passed as `exc_type`; if so, `gen.throw(exception)` is invoked, causing the exception to be raised in the `yield` line inside the generator function body.
2. Otherwise, `next(gen)` is called, resuming the execution of the generator function body after the `yield`.

Example 15-5 has a serious flaw: if an exception is raised in the body of the `with` block, the Python interpreter will catch it and raise it again in the `yield` expression inside

[4] The actual class is named `_GeneratorContextManager`. If you want to see exactly how it works, read its source code in `Lib/contextlib.py` in the Python 3.4 distribution.

looking_glass. But there is no error handling there, so the looking_glass function will abort without ever restoring the original sys.stdout.write method, leaving the system in an invalid state.

Example 15-7 adds special handling of the ZeroDivisionError exception, making it functionally equivalent to the class-based Example 15-3.

Example 15-7. mirror_gen_exc.py: generator-based context manager implementing exception handling — same external behavior as Example 15-3.

```
import contextlib


@contextlib.contextmanager
def looking_glass():
    import sys
    original_write = sys.stdout.write

    def reverse_write(text):
        original_write(text[::-1])

    sys.stdout.write = reverse_write
    msg = ''  ❶
    try:
        yield 'JABBERWOCKY'
    except ZeroDivisionError:  ❷
        msg = 'Please DO NOT divide by zero!'
    finally:
        sys.stdout.write = original_write  ❸
        if msg:
            print(msg)  ❹
```

❶ Create a variable for a possible error message; this is the first change in relation to Example 15-5.

❷ Handle ZeroDivisionError by setting an error message.

❸ Undo monkey-patching of sys.stdout.write.

❹ Display error message, if it was set.

Recall that the __exit__ method tells the interpreter that it has handled the exception by returning True; in that case the interpreter suppresses the exception. On the other hand if __exit__ does not explicitly return a value, the interpreter gets the usual None, and propagates the exception. With @contextmanager the default behavior is inverted: the __exit__ method provided by the decorator assumes any exception sent into the generator is handled and should be suppressed[5]. You must explicitly re-raise an exception in the decorated function if you don't want @contextmanager to suppress it[6]

> Having a try/finally (or a with block), around the yield is an unavoidable price of using @contextmanager, as you never know what the users of your context manager are going to do inside their with block[7].

An interesting real-life example of ‘@contextmanager` outside of the standard library is Martijn Pieters' in-place file rewriting context manager. Example 15-8 shows how it's used:

Example 15-8. A context manager for rewriting files in place.

```
import csv

with inplace(csvfilename, 'r', newline='') as (infh, outfh):
    reader = csv.reader(infh)
    writer = csv.writer(outfh)

    for row in reader:
        row += ['new', 'columns']
        writer.writerow(row)
```

The inplace function is a context manager that gives you two handles — infh and outfh in the example — to the same file, allowing your code to read and write to it at the same time. It's easier to use than the standard library's fileinput.input function (which also provides a context manager, by the way).

If you want to study Martijn's inplace source code (listed in the post), find the yield keyword: everything before it deals with setting up the context, which entails creating a backup file, then opening and yielding references to the readable and writable file handles that will be returned by the __enter__ call. The __exit__ processing after the yield closes the file handles and restores the file from the backup if something went wrong.

[5] The exception is sent into the generator using the throw method, covered in "Coroutine termination and exception handling" on page 473.

[6] This convention was adopted because when context managers were created, generators could not return values, only yield. They now can, as explained in "Returning a value from a coroutine" on page 477. As you'll see, returning a value from a generator does involve an exception.

[7] This tip is quoted literally from a comment by Leonardo Rochael, one of the tech reviewers for this book. Nicely said, Leo!

Note that the use of `yield` in a generator used with the `@contextmanager` decorator has nothing to do with iteration. In the examples shown in this section, the generator function is operating more like a coroutine: a procedure that runs up to a point, then suspends to let the client code run until the client wants the coroutine to proceed with its job. Chapter 16 is all about coroutines.

Chapter summary

This chapter started easily enough with discussion of `else` blocks in `for`, `while` and `try` statements. Once you get used to the peculiar meaning of the `else` clause in these statements, I believe `else` can clarify your intentions.

We then covered context managers and the meaning of the `with` statement, quickly moving beyond its common use to automatically close opened files. We implemented a custom context manager: the `LookingGlass` class with the `__enter__`/`__exit__` methods, and saw how to handle exceptions in the `__exit__` method. A key point that Raymond Hettinger made in his PyCon US 2013 keynote is that `with` is not just for resource management, but it's a tool for factoring out common setup and tear down code, or any pair of operations that need to be done before and after another procedure (What makes Python awesome?, slide 21).

Finally we reviewed functions in the `contextlib` standard library module. One of them, the `@contextmanager` decorator, makes it possible implement a context manager using a simple generator with one `yield` — a leaner solution than coding a class with at least two methods. We reimplemented the `LookingGlass` as a `looking_glass` generator function, and discussed how to do exception handling when using `@contextmanager`.

The `@contextmanager` decorator is an elegant and practical tool that brings together three distinctive Python features: a function decorator, a generator and the `with` statement.

Further reading

Chapter 8. Compound statements of *The Python Language Reference* says pretty much everything there is to say about `else` clauses in `if`, `for`, `while` and `try` statements. Regarding Pythonic usage of `try/except`, with or without `else`, Raymond Hettinger has a brilliant answer to the question *Is it a good practice to use try-except-else in Python?* in StackOverflow. Alex Martelli's *Python in a Nutshell, 2 ed.* (O'Reilly, 2006), has a chapter about exceptions with an excellent discussion of the EAFP style, crediting computing pioneer Grace Hopper for coining the phrase "It's easier to ask forgiveness than permission."

The *Python Standard Library*, chapter *4. Built-in Types* has a section devoted to Context Manager Types. The `__enter__/__exit__` special methods are also documented in *Python Language Reference*, chapter *3. Data model*, section 3.3.8. With Statement Context Managers. Context managers were introduced in PEP 343 — The "with" Statement. This PEP is not easy reading as it spends a lot of time covering corner cases and arguing against alternative proposals. That's the nature of PEPs.

Raymond Hettinger highlighted the `with` statement as a "winning language feature" in his PyCon US 2013 keynote. He also showed some interesting applications of context managers in his talk Transforming code into Beautiful, Idiomatic Python at the same conference.

Jeff Preshing' blog post The Python "with" Statement by Example is interesting for the examples using context managers with the `pycairo` graphics library.

Beazley and Jones devised context managers for very different purposes in their *Python Cookbook 3e* (O'Reilly, 2013). Recipe *8.3. Making Objects Support the Context-Management Protocol* implements a `LazyConnection` class whose instances are context managers that open and close network connections automatically in `with` blocks. *9.22. Defining Context Managers the Easy Way* introduces a context manager for timing code, and another for making transactional changes to a `list` object: within the `with` block, a working copy of the `list` instance is made, and all changes are applied to that working copy. Only when the `with` block completes without an exception the working copy replaces the original list. Simple and ingenious.

Soapbox

Factoring out the bread

In his PyCon US 2013 keynote, What Makes Python Awesome, Raymond Hettinger says when he first saw the `with` statement proposal he thought it was "a little bit arcane". I had a similar initial reaction. PEPs are often hard to read, and PEP 343 is typical in that regard.

Then — Hettinger told us — he had an insight: subroutines are the most important invention in the history of computer languages. If you have sequences of operations like A;B;C and P;B;Q you can factor out B in a subroutine. It's like factoring out the filling in a sandwich: using tuna with different breads. But what if you want to factor out the bread, to make sandwiches with wheat bread, using a different filling each time? That's what the `with` statement offers. It's the complement of the subroutine. Hettinger went on to say:

> The `with` statement is a very big deal. I encourage you to go out and take this tip of the iceberg and drill deeper. You can probably do profound things with the `with` statement. The best uses of it have not been discovered yet. I expect that if you make good use of it, it will be copied into other languages and all future languages will have it. You can

> be part of discovering something almost as profound as the invention of the subroutine itself.

Hettinger admits he is overselling the `with` statement. Nevertheless, it is a very useful feature. When he used the sandwich analogy to explain how `with` is the complement to the subroutine, many possibilities opened up in my mind.

If you need to convince anyone that Python is awesome, you should watch Hettinger's keynote. The bit about context managers is from 23'00" to 26'15". But the entire keynote is excellent.

CHAPTER 16
Coroutines

We find two main senses for the verb "to yield" in dictionaries: to produce or to give way. Both senses apply in Python when we use the `yield` keyword in a generator. A line such as `yield item` produces a value that is received by the caller of `next(…)`, and it also gives way, suspending the execution of the generator so that the caller may proceed until it's ready to consume another value by invoking `next()` again. The caller pulls values from the generator.

A coroutine is syntactically like a generator: just a function with the `yield` keyword in its body. However, in a coroutine, `yield` usually appears on the right side of an expression, e.g. `datum = yield`, and it may or may not produce a value — if there is no expression after the `yield` keyword, the generator yields `None`. The coroutine may receive data from the caller, which uses `.send(datum)` instead of `next(…)` to feed the coroutine. Usually, the caller pushes values into the coroutine.

It is even possible that no data goes in or out through the `yield` keyword. Regardless of the flow of data, `yield` is a control flow device that can be used to implement cooperative multi-tasking: each coroutine yields control to a central scheduler so that other coroutines can be activated.

When you start thinking of `yield` primarily in terms of control flow, you have the mindset to understand coroutines.

Python coroutines are the product of a series of enhancements to the humble generator functions we've seen so far in the book. Following the evolution of coroutines in Python helps understand their features in stages of increasing functionality and complexity.

After a brief overview of how generators were enable to act as a coroutine, we jump to the core of the chapter. Then we'll see:

- The behavior and states of a generator operating as a coroutine.

- Priming a coroutine automatically with a decorator.
- How the caller can control a coroutine through the `.close()` and `.throw(…)` methods of the generator object.
- How coroutines can return values upon termination.
- Usage and semantics of the new `yield from` syntax.
- A use case: coroutines for managing concurrent activities in a simulation.

How coroutines evolved from generators

The infrastructure for coroutines appeared in PEP 342 — Coroutines via Enhanced Generators, implemented in Python 2.5 (2006): since then the `yield` keyword can be used in an expression, and the `.send(value)` method was added to the generator API. Using `.send(…)`, the caller of the generator can post data which then becomes the value of the `yield` expression inside the generator function. This allows a generator to be used as a coroutine: a procedure that collaborates with the caller, yielding and receiving values from the caller.

In addition to `.send(…)`, PEP 342 also added `.throw(…)` and `.close()` methods that respectively allow the caller to throw an exception to be handled inside the generator, and to terminate it. These features are covered in the next section and in "Coroutine termination and exception handling" on page 473.

The latest evolutionary step for coroutines came with PEP 380 - Syntax for Delegating to a Subgenerator, implemented in Python 3.3 (2012). PEP 380 made two syntax changes to generator functions, to make them more useful as coroutines:

- A generator can now `return` a value; previously, providing a value to the `return` statement inside a generator raised a `SyntaxError`.
- The `yield from` syntax enables complex generators to be refactored into smaller, nested generators while avoiding a lot of boilerplate code previously required for a generator to delegate to subgenerators.

These latest changes will be addressed in "Returning a value from a coroutine" on page 477 and "Using `yield from`" on page 479.

Let's follow the established tradition of *Fluent Python* and start with some very basic facts and examples, then move into increasingly mind-bending features.

Basic behavior of a generator used as a coroutine

Example 16-1 illustrates the behavior of a coroutine.

Example 16-1. Simplest possible demonstration of coroutine in action.

```
>>> def simple_coroutine():  # ❶
...     print('-> coroutine started')
...     x = yield  # ❷
...     print('-> coroutine received:', x)
...
>>> my_coro = simple_coroutine()
>>> my_coro  # ❸
<generator object simple_coroutine at 0x100c2be10>
>>> next(my_coro)  # ❹
-> coroutine started
>>> my_coro.send(42)  # ❺
-> coroutine received: 42
Traceback (most recent call last):  # ❻
  ...
StopIteration
```

❶ A coroutine is defined as a generator function: with `yield` in its body.

❷ `yield` is used in an expression; when the coroutine is designed just to receive data from the client it yields `None` — this is implicit as there is no expression to the right of the `yield` keyword.

❸ As usual with generators, you call the function to get a generator object back.

❹ The first call is `next(…)` because the generator hasn't started so it's not waiting in a `yield` and we can't send it any data initially.

❺ This call makes the `yield` in the coroutine body evaluate to `42`; now the coroutine resumes and runs until the next `yield` or termination.

❻ In this case, control flows off the end of the coroutine body, which prompts the generator machinery to raise `StopIteration`, as usual.

A coroutine can be in one of four states. You can determine the current state using the `inspect.getgeneratorstate(…)` function which returns one of these strings:

`'GEN_CREATED'`
: Waiting to start execution.

`'GEN_RUNNING'`
: Currently being executed by the interpreter[1]

`'GEN_SUSPENDED'`
: Currently suspended at a `yield` expression.

[1] You'll only see this state in a multi-threaded application — or if the generator object calls `getgeneratorstate` on itself, which is not useful.

`'GEN_CLOSED'`
: Execution has completed.

Since the argument to the `send` method will become the value of the pending `yield` expression, it follows that you can only make a call like `my_coro.send(42)` if the coroutine is currently suspended. But that's not the case if the coroutine has never been activated — when its state is `'GEN_CREATED'`. That's why the first activation of a coroutine is always done with `next(my_coro)` — you can also call `my_coro.send(None)`, the effect is the same.

If you create a coroutine object and immediately try to send it a value that is not `None`, this is what happens:

```
>>> my_coro = simple_coroutine()
>>> my_coro.send(1729)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't send non-None value to a just-started generator
```

Note the error message: it's quite clear.

The initial call `next(my_coro)` is often described as "priming" the coroutine, i.e. advancing it to the first `yield` to make it ready for use as a live coroutine.

To get a better feel for the behavior of a coroutine, an example that yields more than once is useful. See Example 16-2.

Example 16-2. A coroutine that yields twice.

```
>>> def simple_coro2(a):
...     print('-> Started: a =', a)
...     b = yield a
...     print('-> Received: b =', b)
...     c = yield a + b
...     print('-> Received: c =', c)
...
>>> my_coro2 = simple_coro2(14)
>>> from inspect import getgeneratorstate
>>> getgeneratorstate(my_coro2)  ❶
'GEN_CREATED'
>>> next(my_coro2)  ❷
-> Started: a = 14
14
>>> getgeneratorstate(my_coro2)  ❸
'GEN_SUSPENDED'
>>> my_coro2.send(28)  ❹
-> Received: b = 28
42
>>> my_coro2.send(99)  ❺
-> Received: c = 99
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> getgeneratorstate(my_coro2)  ❻
'GEN_CLOSED'
```

❶ `inspect.getgeneratorstate` reports `GEN_CREATED`, i.e. the coroutine has not started.

❷ Advance coroutine to first `yield`, printing `-> Started: a = 14` message then yielding value of `a` and suspending to wait for value to be assigned to `b`.

❸ `getgeneratorstate` reports `GEN_SUSPENDED`, i.e. the coroutine is paused at a `yield` expression.

❹ Send number `28` to suspended coroutine; the `yield` expression evaluates to `28` and that number is bound to `b`. The `-> Received: b = 28` message is displayed, the value of `a + b` is yielded (`42`) and the coroutine is suspended waiting for the value to be assigned to `c`.

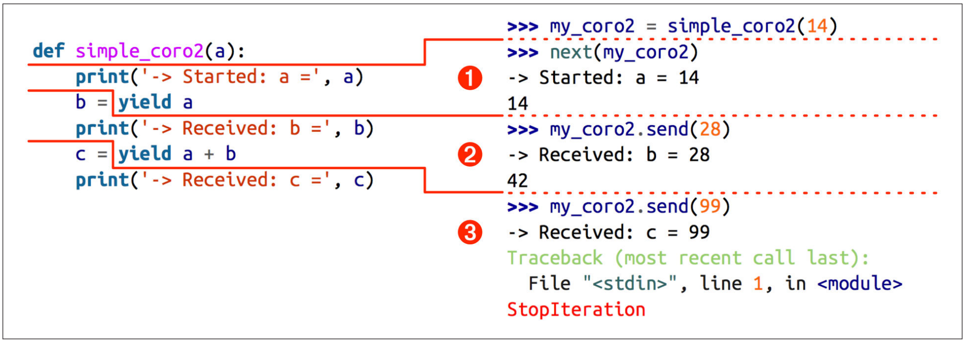
❺ Send number `99` to suspended coroutine; the `yield` expression evaluates to `99` the number is bound to `c`. The `-> Received: c = 99` message is displayed, then the coroutine terminates, causing the generator object to raise `StopIteration`.

❻ `getgeneratorstate` reports `GEN_CLOSED`, i.e. the coroutine execution has completed.

It's crucial to understand that the execution of the coroutine is suspended exactly at the `yield` keyword. As mentioned before, in an assignment statement the code to the right of the = is evaluated before the actual assignment happens. This means that in a line like `b = yield a` the value of `b` will only be set when the coroutine is activated later by the client code. It takes some effort to get used to this fact, but understanding it is essential to make sense of the use of `yield` in asynchronous programming, as we'll see later.

Execution of the `simple_coro2` coroutine can be split in three phases, as shown in in Figure 16-1:

1. `next(my_coro2)` prints first message and runs to `yield a`, yielding number `14`;
2. `my_coro2.send(28)` assigns `28` to `b`, prints second message and runs to `yield a + b`, yielding number `42`.
3. `my_coro2.send(99)` assigns `99` to `c`, prints third message and the coroutine terminates.



```
def simple_coro2(a):
    print('-> Started: a =', a)
    b = yield a
    print('-> Received: b =', b)
    c = yield a + b
    print('-> Received: c =', c)
```

```
>>> my_coro2 = simple_coro2(14)
>>> next(my_coro2)
-> Started: a = 14
14
>>> my_coro2.send(28)
-> Received: b = 28
42
>>> my_coro2.send(99)
-> Received: c = 99
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Figure 16-1. Three phases in the execution of the `simple_coro2` coroutine. Note that each phase ends in a `yield` expression, and the next phase starts in the very same line, when the value of the `yield` expression is assigned to a variable.

Now let's consider a slightly more involved coroutine example.

Example: coroutine to compute a running average

While discussing closures in Chapter 7 we studied objects to compute a running average: Example 7-8 shows a plain class and Example 7-14 presents a higher-order function producing a closure to keep the `total` and `count` variables across invocations. Example 16-3 shows how to do the same with a coroutine[2].

Example 16-3. `coroaverager0.py`: code for a running average coroutine.

```
def averager():
    total = 0.0
    count = 0
    average = None
    while True:  ❶
        term = yield average  ❷
        total += term
        count += 1
        average = total/count
```

❶ This infinite loop means this coroutine will keep on accepting values and producing results as long as the caller sends them. This coroutine will only terminate when the caller calls `.close()` on it, or when it's garbage collected because there are no more references to it.

[2] This example is inspired by a snippet from Jacob Holm in the python-ideas list, message titled Yield-From: Finalization guarantees. Some variations appear later in the thread, and Holm further explains his thinking in message 003912.

❷ The `yield` statement here is used to suspend the coroutine, produce a result to the caller, and — later — to get a value sent by the caller to the coroutine, which resumes its infinite loop.

The advantage of using a coroutine is that `total` and `count` can be simple local variables: no instance attributes or closures are needed to keep the context between calls. Example 16-4 are doctests to show the `averager` coroutine in operation.

Example 16-4. `coroaverager0.py`: doctest for the running average coroutine in Example 16-3.

```
>>> coro_avg = averager()  ❶
>>> next(coro_avg)  ❷
>>> coro_avg.send(10)  ❸
10.0
>>> coro_avg.send(30)
20.0
>>> coro_avg.send(5)
15.0
```

❶ Create the coroutine object.

❷ Prime it by calling `next`.

❸ Now we are in business: each call to `.send(…)` yields the current average.

In the doctest (Example 16-4), the call `next(coro_avg)` makes the coroutine advance to the `yield`, yielding the initial value for average which is `None` so it does not appear on the console. At this point, the coroutine is suspended at the `yield`, waiting for a value to be sent. The line `coro_avg.send(10)` provides that value, causing the coroutine to activate, assigning it to `term`, updating the `total`, `count` and `average` variables and then starting another iteration in the `while` loop, which yields the `average` and waits for another term.

The attentive reader may be anxious to know how the execution of an `averager` instance (e.g. `coro_avg`) may be terminated, since it's body is an infinite loop. We'll cover that real soon in "Coroutine termination and exception handling" on page 473.

But before discussing coroutine termination, let's talk about getting them started. Priming a coroutine before use is a necessary but easy to forget chore. To avoid it, a special decorator can be applied to the coroutine. One such decorator is presented next.

Decorators for coroutine priming

You can't do much with a coroutine without priming it: we must always remember to call `next(my_coro)` before `my_coro.send(x)`. To make coroutine usage more convenient, a priming decorator is sometimes used. The `coroutine` decorator in Example 16-5 is an example[3].

Example 16-5. `coroutil.py`: decorator for priming coroutine.

```
from functools import wraps

def coroutine(func):
    """Decorator: primes `func` by advancing to first `yield`"""
    @wraps(func)
    def primer(*args,**kwargs):   ❶
        gen = func(*args,**kwargs)   ❷
        next(gen)   ❸
        return gen   ❹
    return primer
```

❶ The decorated generator function is replaced by this `primer` function which, when invoked, returns the primed generator.

❷ Call the decorated function to get a generator object.

❸ Prime the generator.

❹ Return it.

Example 16-6 shows the `@coroutine` decorator in use. Contrast with Example 16-3

Example 16-6. `coroaverager1.py`: doctest and code for a running average coroutine using the `@coroutine` decorator from Example 16-5.

```
"""
A coroutine to compute a running average

    >>> coro_avg = averager()   ❶
    >>> from inspect import getgeneratorstate
    >>> getgeneratorstate(coro_avg)   ❷
    'GEN_SUSPENDED'
    >>> coro_avg.send(10)   ❸
    10.0
    >>> coro_avg.send(30)
    20.0
    >>> coro_avg.send(5)
    15.0

"""

from coroutil import coroutine   ❹
```

[3] There are several similar decorators published on the Web. This one is adapted from the ActiveState recipe Pipeline made of coroutines by Chaobin Tang, who in turn credits David Beazley

```
@coroutine  ❺
def averager():  ❻
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield average
        total += term
        count += 1
        average = total/count
```

❶ Call `averager()`, creating a generator object that is primed inside the `primer` function of the `coroutine` decorator.

❷ `getgeneratorstate` reports *GEN_SUSPENDED*, meaning that the coroutine is ready to receive a value.

❸ You can immediately start sending values to `coro_avg`: that's the point of the decorator.

❹ Import the `coroutine` decorator.

❺ Apply it to the `averager` function.

❻ The body of the function is exactly the same as Example 16-3.

Several frameworks provide special decorators designed to work with coroutines. Not all of them actually prime the coroutine — some provide other services, such as hooking it to an event loop. One example from the Tornado asynchronous networking library is the `tornado.gen` decorator.

The `yield from` syntax we'll see in "Using `yield from`" on page 479 automatically primes the coroutine called by it, making it incompatible with decorators such as `@coroutine` from Example 16-5. The `asyncio.coroutine` decorator from the Python 3.4 standard library is designed to work with `yield form` so it does not prime the coroutine. We'll cover it in Chapter 18.

We'll now focus on essential features of coroutines: the methods used to terminate and throw exceptions into them.

Coroutine termination and exception handling

An unhandled exception within a coroutine propagates to the caller of the `next` or `send` which triggered it. Here is an example using the decorated `averager` coroutine from Example 16-6:

Example 16-7. How an unhandled exception kills a coroutine.

```
>>> from coroaverager1 import averager
>>> coro_avg = averager()
>>> coro_avg.send(40)  # ❶
40.0
>>> coro_avg.send(50)
45.0
>>> coro_avg.send('spam')  # ❷
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for +=: 'float' and 'str'
>>> coro_avg.send(60)  # ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

❶ Using the `@coroutine` decorated `averager` we can immediately start sending values.

❷ Sending a non-numeric value causes an exception inside the coroutine.

❸ Because the exception was not handled in the coroutine, it terminated. Any attempt to reactivate it will raise `StopIteration`.

The cause of the error was the sending of a value `'spam'` that could not be added to the `total` variable in the coroutine.

Example 16-7 suggests one way of terminating coroutines: you can use `send` with some sentinel value that tells the coroutine to exit. Constant built-in singletons like `None` and `Ellipsis` are convenient sentinel values. `Ellipsis` has the advantage of being quite unusual in data streams. Another sentinel value I've seen used is `StopIteration`—the class itself, not an instance of it (and not raising it). In other words, using it like: `my_coro.send(StopIteration)`.

Since Python 2.5, generator objects have two methods that allow the client to explicitly send exceptions into the coroutine—`throw` and `close`:

`generator.throw(exc_type[, exc_value[, traceback]])`
: Causes the `yield` expression where the generator was paused to raise the exception given. If the exception is handled by the generator, flow advances to the next `yield`, and the value yielded becomes the value of the `generator.throw` call. If the exception is not handled by the generator, it propagates to the context of the caller.

`generator.close()`
: Causes the `yield` expression where the generator was paused to raise a `GeneratorExit` exception. No error is reported to the caller if the generator does not handle that exception or raises `StopIteration`—usually by running to completion. When receiving a `GeneratorExit` the generator must not yield a value, otherwise a `RuntimeError` is raised. If any other exception is raised by the generator, it propagates to the caller.

The official documentation of the generator object methods is buried deep in the Language Reference, section 6.2.9.1. Generator-iterator methods.

Let's see how `close` and `throw` control a coroutine. Example 16-8 lists the `demo_exc_handling` function used in the following examples.

Example 16-8. `coro_exc_demo.py`: test code for studying exception handling in a coroutine.

```
class DemoException(Exception):
    """An exception type for the demonstration."""

def demo_exc_handling():
    print('-> coroutine started')
    while True:
        try:
            x = yield
        except DemoException:  ❶
            print('*** DemoException handled. Continuing...')
        else:  ❷
            print('-> coroutine received: {!r}'.format(x))
    raise RuntimeError('This line should never run.')  ❸
```

❶ Special handling for `DemoException`.

❷ If no exception, display received value.

❸ This line will never be executed.

The last line in Example 16-8 is unreachable because the infinite loop can only be aborted with an unhandled exception, and that terminates the coroutine immediately.

Normal operation of `demo_exc_handling` is shown in Example 16-9:

Example 16-9. Activating and closing `demo_exc_handling` without an exception.

```
    >>> exc_coro = demo_exc_handling()
    >>> next(exc_coro)
    -> coroutine started
    >>> exc_coro.send(11)
    -> coroutine received: 11
    >>> exc_coro.send(22)
    -> coroutine received: 22
    >>> exc_coro.close()
    >>> from inspect import getgeneratorstate
    >>> getgeneratorstate(exc_coro)
    'GEN_CLOSED'
```

If the DemoException is thrown into the coroutine, it's handled and the demo_exc_handling coroutine continues, as in Example 16-10:

Example 16-10. Throwing DemoException into demo_exc_handling does not break it.

```
>>> exc_coro = demo_exc_handling()
>>> next(exc_coro)
-> coroutine started
>>> exc_coro.send(11)
-> coroutine received: 11
>>> exc_coro.throw(DemoException)
*** DemoException handled. Continuing...
>>> getgeneratorstate(exc_coro)
'GEN_SUSPENDED'
```

On the other hand, if an unhandled exception is thrown into the coroutine, it stops — its state becomes 'GEN_CLOSED'. Example 16-11 demonstrates it:

Example 16-11. Coroutine terminates if it can't handle an exception thrown into it.

```
>>> exc_coro = demo_exc_handling()
>>> next(exc_coro)
-> coroutine started
>>> exc_coro.send(11)
-> coroutine received: 11
>>> exc_coro.throw(ZeroDivisionError)
Traceback (most recent call last):
  ...
ZeroDivisionError
>>> getgeneratorstate(exc_coro)
'GEN_CLOSED'
```

If it's necessary that some cleanup code is run no matter how the coroutine ends, you need to wrap the relevant part of the coroutine body in a try/finally block, as in Example 16-12:

Example 16-12. coro_finally_demo.py: use of try/finally to perform actions on coroutine termination.

```
class DemoException(Exception):
    """An exception type for the demonstration."""


def demo_finally():
    print('-> coroutine started')
    try:
        while True:
            try:
                x = yield
            except DemoException:
                print('*** DemoException handled. Continuing...')
            else:
                print('-> coroutine received: {!r}'.format(x))
    finally:
        print('-> coroutine ending')
```

One of the main reasons why the `yield from` construct was added to Python 3.3 has to do with throwing exceptions into nested coroutines. The other reason was to enable coroutines to return values more conveniently. Read on to see how.

Returning a value from a coroutine

Example 16-13 shows a variation of the `averager` coroutine that returns a result. For didactic reasons, it does not yield the running average with each activation. This is to emphasize that some coroutines do not yield anything interesting, but are designed to return a value at the end, often the result of some accumulation.

The result returned by `averager` in Example 16-13 is a `namedtuple` with the number of terms averaged (`count`) and the `average`. I could have returned just the `average` value, but returning a tuple exposes another interesting piece of data that was accumulated: the `count` of terms.

Example 16-13. `coroaverager2.py`: code for an `averager` coroutine that returns a result.

```
from collections import namedtuple

Result = namedtuple('Result', 'count average')


def averager():
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield
        if term is None:
            break  ❶
        total += term
        count += 1
        average = total/count
    return Result(count, average)  ❷
```

❶ In order to return a value, a coroutine must terminate normally; this is why this version of `averager` has a condition to break out of its accumulating loop.

❷ Return a `namedtuple` with the `count` and `average`. Before Python 3.3 it was a syntax error to return a value in a generator function.

To see how this new `averager` works, we can drive it from the console, as in Example 16-14.

Example 16-14. `coroaverager2.py`: doctest showing the behavior of `averager`.

```
>>> coro_avg = averager()
>>> next(coro_avg)
>>> coro_avg.send(10)  ❶
>>> coro_avg.send(30)
>>> coro_avg.send(6.5)
>>> coro_avg.send(None)  ❷
Traceback (most recent call last):
   ...
StopIteration: Result(count=3, average=15.5)
```

❶ This version does not yield values.

❷ Sending `None` terminates the loop, causing the coroutine to end by returning the result. As usual, the generator object raises `StopIteration`. The `value` attribute of the exception carries the value returned.

Note that the value of the `return` expression is smuggled to the caller as an attribute of the `StopIteration` exception. This is a bit of a hack, but it preserves the existing behavior of generator objects: raising `StopIteration` when exhausted.

Example 16-15 shows how to retrieve the value returned by the coroutine:

Example 16-15. Catching `StopIteration` let's us get the value returned by `averager`.

```
>>> coro_avg = averager()
>>> next(coro_avg)
>>> coro_avg.send(10)
>>> coro_avg.send(30)
>>> coro_avg.send(6.5)
>>> try:
...     coro_avg.send(None)
... except StopIteration as exc:
...     result = exc.value
...
>>> result
Result(count=3, average=15.5)
```

This roundabout way of getting the return value from a coroutine makes more sense when we realize it was defined as part of PEP 380, and the `yield from` construct handles it automatically by catching `StopIteration` internally. This is analogous to the use of `StopIteration` in `for` loops: the exception is handled by the loop machinery in a way that is transparent to the user. In the case of `yield from`, the interpreter not only consumes the `StopIteration`, but its `value` attribute becomes the value of the `yield from` expression itself. Unfortunately we can't test this interactively in the console, because it's a syntax error to use `yield from` — or `yield`, for that matter — outside of a function[4].

The next section has an example where the `averager` coroutine is used with `yield from` to produce a result, as intended in PEP 380. So let's tackle `yield from`.

Using `yield from`

The first thing to know about `yield from` is that it is a completely new language construct. It does so much more than `yield` that the reuse of that keyword is arguably misleading. Similar constructs in other languages are called `await` and that is a much better name because it conveys a crucial point: when a generator `gen` calls `yield from subgen()`, the `subgen` takes over and will yield values to the caller of `gen`; the caller will in effect drive `subgen` directly. Meanwhile `gen` will be blocked, waiting until `subgen` terminantes[5].

We've seen in Chapter 14 that `yield from` can be used as a shortcut to `yield` in a `for` loop. For example, this:

```
>>> def gen():
...     for c in 'AB':
...         yield c
...     for i in range(1, 3):
...         yield i
...
>>> list(gen())
['A', 'B', 1, 2]
```

Can be written as:

```
>>> def gen():
...     yield from 'AB'
...     yield from range(1, 3)
...
>>> list(gen())
['A', 'B', 1, 2]
```

When we first mentioned `yield from` in "New syntax in Python 3.3: `yield from`" on page 435, this code demonstrated a practical use for it:

[4] There is an iPython extension called ipython-yf that enables evaluating `yield from` directly in the iPython console. It's used to test asynchronous code and works with `asyncio`. It was submitted as a patch to Python 3.5 but was not accepted. See Issue #22412: Towards an asyncio-enabled command line in the Python bug tracker.

[5] As I write this, there is an open PEP proposing the addition of `await` and `async` keywords: PEP 492 — Coroutines with async and await syntax

Example 16-16. Chaining iterables with +yield from+[6].

```
>>> def chain(*iterables):
...     for it in iterables:
...         yield from it
...
>>> s = 'ABC'
>>> t = tuple(range(3))
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

A slightly more complicated — but more useful — example of `yield from` is in recipe 4.14, "Flattening a Nested Sequence" in Beazley & Jones's Python Cookbook, 3e (source code available on Github).

The first thing the `yield from x` expression does with the `x` object is to call `iter(x)` to obtain an iterator from it. This means that `x` can be any iterable.

However, if replacing nested `for` loops yielding values was the only contribution of `yield from`, this language addition wouldn't have had a good chance of being accepted. The real nature of `yield from` cannot be demonstrated with simple iterables, it requires the mind-expanding use of nested generators. That's why PEP 380 which introduced `yield from` is titled "Syntax for Delegating to a Subgenerator".

The main feature of `yield from` is to open a bidirectional channel from the outermost caller to the innermost subgenerator, so that values can be sent and yielded back and forth directly from them, and exceptions can be thrown all the way in without adding a lot of exception handling boilerplate code in the intermediate coroutines. This is what enables coroutine delegation in a way that was not possible before.

The use of `yield from` requires a non-trivial arrangement of code. To talk about the required moving parts, PEP 380 uses some terms in a very specific way:

delegating generator
: The generator function that contains the `yield from <iterable>` expression.

subgenerator
: The generator obtained from the `<iterable>` part of the `yield from` expression. This is the "subgenerator" mentioned in the title of PEP 380: "Syntax for Delegating to a Subgenerator".

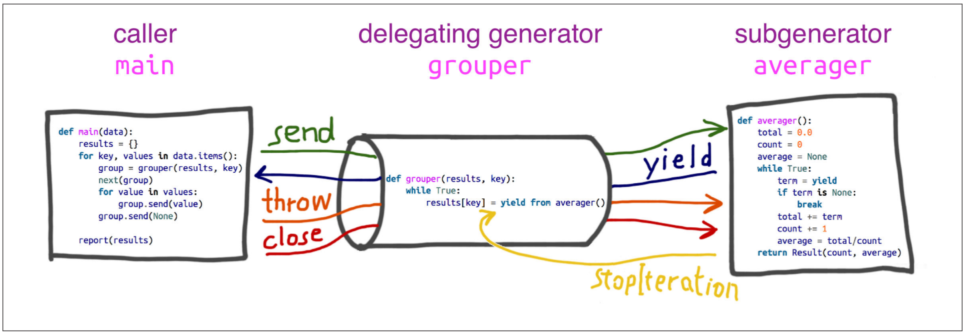
[6] *Example 16-16 is a didactic example only. The `itertools` module already provides an optimized `chain` function written in C.*

caller

PEP 380 uses the term “caller” to refer to the client code that calls the delegating generator. Depending on context, I use “client” instead of “caller”, to distinguish from the delegating generator which is also a “caller” (it calls the subgenerator).

PEP 380 often uses the word “iterator” to refer to the subgenerator. That’s confusing because the delegating generator is also an iterator. So I prefer to use the term subgenerator, in line with the title of the PEP — "Syntax for Delegating to a Subgenerator”. However, the subgenerator can be a simple iterator implementing only `__next__`, and `yield from` can handle that too, although it was created to support generators implementing `__next__`, `send`, `close` and `throw`.

Example 16-17 provides more context to see `yield from` at work, and Figure 16-2 identifies the relevant parts of the example.



```
def main(data):
    results = {}
    for key, values in data.items():
        group = grouper(results, key)
        next(group)
        for value in values:
            group.send(value)
        group.send(None)

    report(results)
```

```
def grouper(results, key):
    while True:
        results[key] = yield from averager()
```

```
def averager():
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield
        if term is None:
            break
        total += term
        count += 1
        average = total/count
    return Result(count, average)
```

Figure 16-2. While the delegating generator is suspended at `yield from`, the caller sends data directly to the subgenerator, which yields data back to the caller. The delegating generator resumes when the subgenerator returns and the interpreter raises `StopIteration` with the returned value attached[7].

The `coroaverager3.py` script reads a `dict` with weights and heights from girls and boys in an imaginary 7th grade class. For example, the key `'boys;m'` maps to the heights of 9 boys, in meters; `'girls;kg'` are the weights of 10 girls in kilograms. The script feeds the data for each group into the `averager` coroutine we’ve seen before, and produces a report like this one:

```
$ python3 coroaverager3.py
 9 boys  averaging 40.42kg
```

[7] *This picture was inspired by a diagram by Paul Sokolovksy.*

```
 9 boys  averaging 1.39m
10 girls averaging 42.04kg
10 girls averaging 1.43m
```

The code in Example 16-17 is certainly not the most straightforward solution to the problem, but it serves to show `yield from` in action. This example is inspired by the one given in What's New In Python 3.3.

Example 16-17. `coroaverager3.py`: using `yield from` to drive `averager` and report statistic.

```
from collections import namedtuple

Result = namedtuple('Result', 'count average')


# the subgenerator
def averager():  ❶
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield  ❷
        if term is None:  ❸
            break
        total += term
        count += 1
        average = total/count
    return Result(count, average)  ❹


# the delegating generator
def grouper(results, key):  ❺
    while True:  ❻
        results[key] = yield from averager()  ❼


# the client code, a.k.a. the caller
def main(data):  ❽
    results = {}
    for key, values in data.items():
        group = grouper(results, key)  ❾
        next(group)  ❿
        for value in values:
            group.send(value)  ⓫
        group.send(None)  # important! ⓬

    # print(results)  # uncomment to debug
    report(results)


# output report
```

```
def report(results):
    for key, result in sorted(results.items()):
        group, unit = key.split(';')
        print('{:2} {:5} averaging {:.2f}{}'.format(
              result.count, group, result.average, unit))


data = {
    'girls;kg':
        [40.9, 38.5, 44.3, 42.2, 45.2, 41.7, 44.5, 38.0, 40.6, 44.5],
    'girls;m':
        [1.6, 1.51, 1.4, 1.3, 1.41, 1.39, 1.33, 1.46, 1.45, 1.43],
    'boys;kg':
        [39.0, 40.8, 43.2, 40.8, 43.1, 38.6, 41.4, 40.6, 36.3],
    'boys;m':
        [1.38, 1.5, 1.32, 1.25, 1.37, 1.48, 1.25, 1.49, 1.46],
}


if __name__ == '__main__':
    main(data)
```

❶ Same `averager` coroutine from Example 16-13. Here it is the subgenerator.

❷ Each value sent by the client code in `main` will be bound to `term` here.

❸ The crucial terminating condition. Without it, a `yield from` calling this coroutine will block forever.

❹ The returned `Result` will be the value of the `yield from` expression in `grouper`.

❺ `grouper` is the delegating generator.

❻ Each iteration in this loop creates a new instance of `averager`; each is a generator object operating as a coroutine.

❼ Whenever `grouper` is sent a value, it's piped into the `averager` instance by the `yield from`. `grouper` will be suspended here as long as the `averager` instance is consuming values sent by the client. When an `averager` instance runs to the end, the value it returns is bound to `results[key]`. The `while` loop then proceeds to create another `averager` instance to consume more values.

❽ `main` is the client code, or "caller" in PEP 380 parlance. This is the function that drives everything.

❾ `group` is a generator object resulting from calling `grouper` with the `results` `dict` to collect the results, and a particular `key`. It will operate as a coroutine.

❿ Prime the coroutine.

⓫ Send each `value` into the `grouper`. That value ends up in the `term = yield` line of `averager`; `grouper` never has a chance to see it.

⓬ Sending `None` into `grouper` causes the current `averager` instance to terminate, and allows `grouper` to run again, which creates another `averager` for the next group of values.

The last callout in Example 16-17 with the comment "`important!`" highlights a crucial line of code: `group.send(None)`, which terminates one `averager` and starts the next. If you comment out that line, the script produces no output. Uncommenting the `print(results)` line near the end of `main` reveals that the `results` dict ends up empty.

> If you want to figure out for yourself why no results are collected, it will be a great way to exercise your understanding of how `yield from` works. The code for *coroaverager3.py* is in the `fluentpython/example-code` repository on Github. The explanation is next.

Here is an overview of how Example 16-17 works, explaining what would happen if we omitted the call `group.send(None)` marked "important!" in `main`:

- Each iteration of the outer `for` loop creates a new `grouper` instance named `group`; this is the delegating generator.
- The call `next(group)` primes the `grouper` delegating generator, which enters its `while True` loop and suspends at the `yield from`, after calling the subgenerator `averager`.
- The inner `for` loop calls `group.send(value)`; this feeds the subgenerator `averager` directly. Meanwhile, the current `group` instance of `grouper` is suspended at the `yield from`.
- When the inner `for` loop ends, the `group` instance is still suspended at the `yield from`, so the assignment to `results[key]` in the body of `grouper` has not happened yet.
- Without the last `group.send(None)` in the outer `for` loop, the `averager` subgenerator never terminates, the delegating generator `group` is never reactivated, and the assignment to `results[key]` never happens.
- When execution loops back to the top of the outer `for` loop, a new `grouper` instance is created and bound to `group`. The previous `grouper` instance is garbage collected (together with its own unfinished `averager` subgenerator instance).

The key takeaway from this experiment is: if a subgenerator never terminates, the delegating generator will be suspended forever at the `yield from`. This will not prevent your program from making progress because the `yield from` (like the simple `yield`) transfers control to the client code i.e. the caller of the delegating generator. But it does mean that some task will be left unfinished.

Example 16-17 demonstrates the simplest arrangement of `yield from`, with only one delegating generator and one subgenerator. Since the delegating generator works as a pipe, you can connect any number of them in a pipeline: one delegating generator uses `yield from` to call a subgenerator, which itself is a delegating generator calling another subgenerator with `yield from` and so on. Eventually this chain must end in a simple generator that uses just `yield`, but it may also end in any iterable object, as in Example 16-16.

Every `yield from` chain must be driven by a client that calls `next(…)` or `.send(…)` on the outermost delegating generator. This call may be implicit, such as a `for` loop.

Now let's go over the formal description of the `yield from` construct, as presented in PEP 380.

The meaning of `yield from`

While developing PEP 380, Greg Ewing — the author — was questioned about the complexity of the proposed semantics. One of his answers was "For humans, almost all the important information is contained in one paragraph near the top." He then quoted part of the draft of PEP 380 which read — at the time:

> "When the iterator is another generator, the effect is the same as if the body of the subgenerator were inlined at the point of the `yield from` expression. Furthermore, the subgenerator is allowed to execute a `return` statement with a value, and that value becomes the value of the `yield from` expression." [8]

Those soothing words are no longer part of the PEP — because they don't cover all the corner cases. But they are OK as a first approximation.

The approved version of PEP 380 explains the behavior of `yield from` in 6 points in the Proposal section. I reproduce them almost exactly here, except that I replaced every occurrence of the ambiguous word "iterator" by "subgenerator" and added a few clarifications.

[8] Message to python-dev: PEP 380 (yield from a subgenerator) comments (Mar 21, 2009).

1. Any values that the subgenerator yields are passed directly to the caller of the delegating generator i.e. the client code.
2. Any values sent to the delegating generator using `send()` are passed directly to the subgenerator. If the sent value is `None`, the subgenerator's `__next__()` method is called. If the sent value is not `None`, the subgenerator's `send()` method is called. If the call raises `StopIteration`, the delegating generator is resumed. Any other exception is propagated to the delegating generator.
3. `return expr` in a generator (or subgenerator) causes `StopIteration(expr)` to be raised upon exit from the generator.
4. The value of the `yield from` expression is the first argument to the `StopIteration` exception raised by the subgenerator when it terminates.

   Example 16-17 illustrated all four points above. The other two features of `yield from` have to do with exceptions and termination:

5. Exceptions other than `GeneratorExit` thrown into the delegating generator are passed to the `throw()` method of the subgenerator. If the call raises `StopIteration`, the delegating generator is resumed. Any other exception is propagated to the delegating generator.
6. If a `GeneratorExit` exception is thrown into the delegating generator, or the `close()` method of the delegating generator is called, then the `close()` method of the subgenerator is called if it has one. If this call results in an exception, it is propagated to the delegating generator. Otherwise, `GeneratorExit` is raised in the delegating generator.

The detailed semantics of `yield from` are subtle, especially the points dealing with exceptions. Greg Ewing did a great job putting them to words in English in PEP 380.

Ewing also documented the behavior of `yield from` using pseudocode (with Python syntax). I personally found it useful to spend some time studying the pseudocode in PEP 380. However, the pseudocode is 40 lines long and not so easy to grasp at first.

A good way to approach that pseudocode is to simplify it to handle only the most basic and common use case of `yield from`.

Consider that `yield from` appears in a delegating generator. The client code drives delegating generator, which drives the subgenerator. So, to simplify the logic involved, let's pretend the client doesn't ever call `.throw(…)` or `.close()` on the delegating generator. Let's also pretend the subgenerator never raises an exception until it terminates, when `StopIteration` is raised by the interpreter.

Example 16-17 is a script where those simplifying assumptions hold. In fact, in much real-life code the delegating generator is expected to run to completion. So let's see how `yield from` works in this happier, simpler world.

Take a look at Example 16-18. The 16 lines of code there are the expansion of this single statement, in the body of the delegating generator:

```
RESULT = yield from EXPR
```

Try to follow the logic in Example 16-18.

Example 16-18. Simplified pseudocode equivalent to the statement `RESULT = yield from EXPR` in the delegating generator. This covers the simplest case: `.throw(...)` and `.close()` are not supported; the only exception handled is `StopIteration`.

```
_i = iter(EXPR)  ❶
try:
    _y = next(_i)  ❷
except StopIteration as _e:
    _r = _e.value  ❸
else:
    while 1:  ❹
        _s = yield _y  ❺
        try:
            _y = _i.send(_s)  ❻
        except StopIteration as _e:  ❼
            _r = _e.value
            break

RESULT = _r  ❽
```

❶ The `EXPR` can be any iterable, because `iter()` is applied to get an iterator `_i` (this is the subgenerator).

❷ The subgenerator is primed; the result is stored to be the first yielded value `_y`.

❸ If `StopIteration` was raised, extract the `value` attribute from the exception and assign it to `_r`: this is the `RESULT` in the simplest case.

❹ While this loop is running, the delegating generator is blocked, operating just as a channel between the caller and the subgenerator.

❺ Yield the current item yielded from the subgenerator; wait for a value `_s` sent by the caller. Note that this is the only `yield` in this listing.

❻ Try to advance the subgenerator, forwarding the `_s` sent by the caller.

❼ If the subgenerator raised `StopIteration`, get the `value`, assign to `_r` and exit the loop, resuming the delegating generator.

❽ `_r` is the `RESULT`: the value of the whole `yield from` expression.

In this simplified pseudocode I preserved the variable names used in the pseudocode published in PEP 380. The variables are:

`_i` *(iterator)*
: The subgenerator.

`_y` *(yielded)*
: A value yielded from the subgnerator.

`_r` *(result)*
: The eventual result, i.e. the value of the `yield from` expression when the subgenerator ends.

`_s` *(sent)*
: A value sent by the caller to the delegating generator, which is forwarded to the subgenerator.

`_e` *(exception)*
: An exception (always an instance of `StopIteration` in this simplified pseudocode).

Besides not handling `.throw(…)` and `.close()`, the simplified pseudocode always uses `.send(…)` to forward `next()` or `.send(…)` calls by the client to the subgenerator. Don't worry about these fine distinctions on a first reading. As mentioned, Example 16-17 would run perfectly well if the `yield from` did only what is shown in the 16 lines of the simplified pseudocode in Example 16-18.

But the reality is more complicated, because of the need to handle `.throw(…)` and `.close()` calls from the client, which must be passed into the subgenerator. Also, the subgenerator may be a plain iterator which does not support `.throw(…)` or `.close()`, so this must be handled by the `yield from` logic. If the subgenerator does implement those methods, inside the subgenerator both methods cause exceptions to be raised, which must be handled by the `yield from` machinery as well. The subgnerator my also throw exceptions of its own, unprovoked by the caller, and this must also be dealt with in the `yield from` implementation. Finally, as an optimization, if the caller calls `next(…)` or `.send(None)`, both are forwarded as a `next(…)` call on the subgenerator; only if the caller sends a non-`None` value, the `.send(…)` method of the subgenerator is used.

Below, for your convenience, is the complete pseudocode of the `yield from` expansion from PEP 380, syntax-highlighted and annotated. Example 16-19 was copied verbatim; only the callout numbers were added by me.

Again, those 40 lines of code are the expansion of this single statement, in the body of the delegating generator:

```
RESULT = yield from EXPR
```

Example 16-19. Pseudocode equivalent to the statement `RESULT = yield from EXPR` in the delegating generator.

```
_i = iter(EXPR)  ❶
try:
    _y = next(_i)  ❷
except StopIteration as _e:
    _r = _e.value  ❸
else:
    while 1:  ❹
        try:
            _s = yield _y  ❺
        except GeneratorExit as _e:  ❻
            try:
                _m = _i.close
            except AttributeError:
                pass
            else:
                _m()
            raise _e
        except BaseException as _e:  ❼
            _x = sys.exc_info()
            try:
                _m = _i.throw
            except AttributeError:
                raise _e
            else:  ❽
                try:
                    _y = _m(*_x)
                except StopIteration as _e:
                    _r = _e.value
                    break
        else:  ❾
            try:  ❿
                if _s is None:  ⓫
                    _y = next(_i)
                else:
                    _y = _i.send(_s)
            except StopIteration as _e:  ⓬
                _r = _e.value
                break

RESULT = _r  ⓭
```

❶ The `EXPR` can be any iterable, because `iter()` is applied to get an iterator `_i` (this is the subgenerator).

❷ The subgenerator is primed; the result is stored to be the first yielded value `_y`.

❸ If `StopIteration` was raised, extract the `value` attribute from the exception and assign it to `_r`: this is the `RESULT` in the simplest case.

4 While this loop is running, the delegating generator is blocked, operating just as a channel between the caller and the subgenerator.

5 Yield the current item yielded from the subgenerator; wait for a value `_s` sent by the caller. This is the only `yield` in this listing.

6 This deals with closing the delegating generator and the subgenerator. Since the subgenerator can be any iterator, it may not have a `close` method.

7 This deals with exceptions thrown in by the caller using `.throw(…)`. Again, the subgenerator may be an iterator with no `throw` method to be called — in which case the exception is raised in the delegating generator.

8 If the subgenerator has a `throw` method, call it with the exception passed from the caller. The subgenerator may handle the exception (and the loop continues); it may raise `StopIteration` (the `_r` result is extracted from it, and the loop ends); or it may raise the same or another exception which is not handled here and propagates to the delegating generator.

9 If no exception was received when yielding…

10 Try to advance the subgenerator…

11 Call `next` on the subgenerator if the last value received from the caller was `None`, otherwise call `send`.

12 If the subgenerator raised `StopIteration`, get the `value`, assign to `_r` and exit the loop, resuming the delegating generator.

13 `_r` is the `RESULT`: the value of the whole `yield from` expression.

Most of the logic is of the `yield from` pseudocode is implemented in six `try/except` blocks nested up to four levels deep, so it's a bit hard to read. The only other control flow keywords used are one `while`, one `if` and one `yield`. Find the `while`, the `yield`, the `next(…)` and the `.send(…)` calls: they will help you get an idea of how the whole structure works.

Right at the top of Example 16-19, one important detail revealed by the pseudocode is that the subgenerator is primed (second callout in Example 16-19)[9]. This means that auto-priming decorators such as that in "Decorators for coroutine priming" on page 471 are incompatible with `yield from`.

In the same message I quoted at the top of this section, Greg Ewing has this to say about the pseudocode expansion of `yield from`:

[9] In a message to python-ideas on Apr. 5, 2009, Nick Coghlan questioned whether the implicit priming done by `yield from` was a good idea.

> You're not meant to learn about it by reading the expansion — that's only there to pin down all the details for language lawyers.

Focusing on the details of the pseudocode expansion may not be helpful — depending on your learning style. Studying real code that uses `yield from` is certainly more profitable than poring over the pseudocode of its implementation. However, almost all the `yield from` examples I've seen are tied to asynchronous programming with the `asyncio` module, so they depend on an active event loop to work. We'll see `yield from` numerous times in Chapter 18. "Further reading" on page 502 has a few links to interesting code using `yield from` without an event loop.

We'll now move on to classic example of coroutine usage: programming simulations. This example does not showcase `yield from`, but it does reveal how coroutines are used to manage concurrent activities on a single thread.

Use case: coroutines for discrete event simulation

> Coroutines are a natural way of expressing many algorithms, such as simulations, games, asynchronous I/O, and other forms of event-driven programming or co-operative multitasking[10].
>
> — Guido van Rossum and Phillip J. Eby
> *PEP 342 -- Coroutines via Enhanced Generators*

In this section I will describe a very simple simulation implemented using just coroutines and standard library objects. Simulation is a classic application of coroutines in the computer science literature. Simula, the first OO language, introduced the concept of coroutines precisely to support simulations.

> The motivation for the following simulation example is not academic. Coroutines are the fundamental building block of the `asyncio` package. A simulation shows how to implement concurrent activities using coroutines instead of threads — and this will greatly help when we tackle `asyncio` with in Chapter 18.

Before going into the example, a word about simulations.

About discrete event simulations

A discrete event simulation (DES) is a type of simulation where a system is modeled as sequence of events. In a DES, the simulation "clock" does not advance by fixed increments, but advances directly to the simulated time of the next modeled event. For example, if we are simulating the operation of a taxi cab from a high-level perspective, one

[10] Opening sentence of the *Motivation* section in PEP 342.

event is picking up a passenger, the next is dropping the passenger off. It doesn't matter if a trip takes 5 or 50 minutes: when the drop off event happens, the clock is updated to the end time of the trip in a single operation. In a DES, we can simulate a year of cab trips in less than a second. This is in contrast with a continuous simulation where the clock advances continuously by a fixed — and usually small — increment.

Intuitively, turn-based games are examples of discrete event simulations: the state of the game only changes when a player moves, and while a player is deciding the next move, the simulation clock is frozen. Real time games, on the other hand, are continuous simulations where the simulation clock is running all the time, the state of the game is updated many times per second and slow players are at a real disadvantage.

Both types of simulations can be written with multiple threads or a single thread using event oriented programming techniques such as callbacks or coroutines driven by an event loop. It's arguably more natural to implement a continuous simulation using threads to account for actions happening in parallel in real time. On the other hand, coroutines offer exactly the right abstraction for writing a DES. SimPy[11] is a DES package for Python that uses one coroutine to represent each process in the simulation.

> In the field of simulation, the term process refers to the activities of an entity in the model, and not to an OS process. A simulation process may be implemented as an OS process, but usually a thread or a coroutine is used for that purpose.

If you are interested in simulations, SimPy is well worth studying. However, in this section I will describe a very simple DES implemented using only standard library features. My goal is to help you develop an intuition about programming concurrent actions with coroutines. Understanding the next section will require careful study, but the reward will come as insights on how libraries such as `asyncio`, Twisted and Tornado can manage many concurrent activities using a single thread of execution.

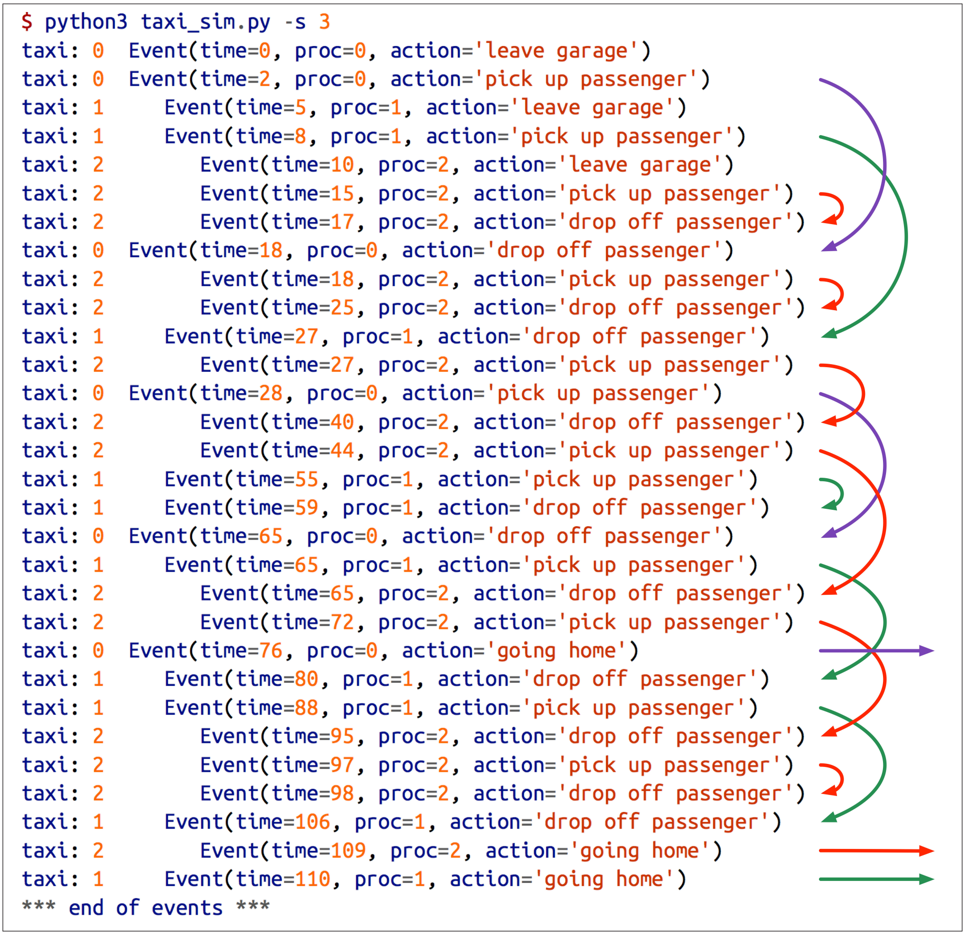
The taxi fleet simulation

In our simulation program, `taxi_sim.py`, a number of taxicabs are created. Each will make a fixed number of trips and then go home. A taxi leaves the garage and starts "prowling" — looking for a passenger. This lasts until a passenger is picked up, and a trip starts. When the passenger is dropped off, the taxi goes back to prowling.

The time elapsed during prowls and trips is generated using an exponential distribution. For a cleaner display, times are in whole minutes, but the simulation would work as well

[11] See the official documentation for Simpy — not to be confused with the well-known but unrelated SymPy, a library for symbolic mathematics.

using `float` intervals[12]. Each change of state in each cab is reported as an event. Figure 16-3 shows a sample run of the program.



```
$ python3 taxi_sim.py -s 3
taxi: 0  Event(time=0, proc=0, action='leave garage')
taxi: 0  Event(time=2, proc=0, action='pick up passenger')
taxi: 1     Event(time=5, proc=1, action='leave garage')
taxi: 1     Event(time=8, proc=1, action='pick up passenger')
taxi: 2        Event(time=10, proc=2, action='leave garage')
taxi: 2        Event(time=15, proc=2, action='pick up passenger')
taxi: 2        Event(time=17, proc=2, action='drop off passenger')
taxi: 0  Event(time=18, proc=0, action='drop off passenger')
taxi: 2        Event(time=18, proc=2, action='pick up passenger')
taxi: 2        Event(time=25, proc=2, action='drop off passenger')
taxi: 1     Event(time=27, proc=1, action='drop off passenger')
taxi: 2        Event(time=27, proc=2, action='pick up passenger')
taxi: 0  Event(time=28, proc=0, action='pick up passenger')
taxi: 2        Event(time=40, proc=2, action='drop off passenger')
taxi: 2        Event(time=44, proc=2, action='pick up passenger')
taxi: 1     Event(time=55, proc=1, action='pick up passenger')
taxi: 1     Event(time=59, proc=1, action='drop off passenger')
taxi: 0  Event(time=65, proc=0, action='drop off passenger')
taxi: 1     Event(time=65, proc=1, action='pick up passenger')
taxi: 2        Event(time=65, proc=2, action='drop off passenger')
taxi: 2        Event(time=72, proc=2, action='pick up passenger')
taxi: 0  Event(time=76, proc=0, action='going home')
taxi: 1     Event(time=80, proc=1, action='drop off passenger')
taxi: 1     Event(time=88, proc=1, action='pick up passenger')
taxi: 2        Event(time=95, proc=2, action='drop off passenger')
taxi: 2        Event(time=97, proc=2, action='pick up passenger')
taxi: 2        Event(time=98, proc=2, action='drop off passenger')
taxi: 1     Event(time=106, proc=1, action='drop off passenger')
taxi: 2        Event(time=109, proc=2, action='going home')
taxi: 1     Event(time=110, proc=1, action='going home')
*** end of events ***
```

Figure 16-3. Sample run of `taxi_sim.py` with three taxis. The `-s 3` argument sets the random generator seed so program runs can be reproduced for debugging and demonstration. Colored arrows highlight taxi trips.

The most important thing to note in Figure 16-3 is the interleaving of the trips by the three taxis. I manually added the arrows to make it easier to see the taxi trips: each arrow starts when a passenger is picked up and ends when the passenger is dropped off. In-

[12] I am not an expert in taxi fleet operations, so don't take my numbers seriously. Exponential distributions are commonly used in DES. You'll see some very short trips. Just pretend it's a rainy day and some passengers are taking cabs just to go around the block — in an ideal city where there are cabs when it rains.

tuitively, this demonstrates how coroutines can be used for managing concurrent activities.

Other things to note about Figure 16-3:

- Each taxi leaves the garage 5 minutes after the other.
- It took 2 minutes for taxi 0 to pick up the first passenger at time=2; 3 minutes for taxi 1 (time=8), and 5 minutes for taxi 2 (time=15).
- The cabbie in taxi 0 only makes 2 trips (purple arrows): the first starts at time=2 and ends at time=18; the second starts at time=28 and ends at time=65 — the longest trip in this simulation run.
- Taxi 1 makes 4 trips (green arrows) then goes home at time=110.
- Taxi 2 makes 6 trips (red arrows) then goes home at time=109. His last trip lasts only one minute, starting at time=97[13].
- While taxi 1 is making her first trip, starting at time=8, taxi 2 leaves the garage at time=10 and completes 2 trips (short red arrows).
- In this sample run all scheduled events completed in the default simulation time of 180 minutes; last event was at time=110.

The simulation may also end with pending events. When that happens, the final message reads like this:

```
*** end of simulation time: 3 events pending ***
```

The full listing of taxi_sim.py is at Example A-6. In this chapter we'll show only the parts that are relevant to our study of coroutines. The really important functions are only two: taxi_process (a coroutine), and the Simulator.run method where the main loop of the simulation is executed.

Example 16-20 shows the code for taxi_process. This coroutine uses two objects defined elsewhere: the compute_delay function, which returns a time interval in minutes, and the Event class, a namedtuple defined like this:

```
Event = collections.namedtuple('Event', 'time proc action')
```

In an Event instance, time is the simulation time when the event will occur, proc is the identifier of the taxi process instance, and action is a string describing the activity.

Let's go over taxi_process play-by-play in Example 16-20:

[13] I was the passenger. I realized I forgot my wallet.

Example 16-20. `taxi_sim.py`: `taxi_process` coroutine which implements the activities of each taxi.

```
def taxi_process(ident, trips, start_time=0):  ❶
    """Yield to simulator issuing event at each state change"""
    time = yield Event(start_time, ident, 'leave garage')  ❷
    for i in range(trips):  ❸
        time = yield Event(time, ident, 'pick up passenger')  ❹
        time = yield Event(time, ident, 'drop off passenger')  ❺

    yield Event(time, ident, 'going home')  ❻
    # end of taxi process  ❼
```

❶ `taxi_process` will be called once per taxi, creating a generator object to represent its operations. `ident` is the number of the taxi (eg. 0, 1, 2 in the sample run); `trips` is the number of trips this taxi will make before going home; `start_time` is when the taxi leaves the garage.

❷ The first `Event` yielded is `'leave garage'`. This suspends the coroutine, and lets the simulation main loop proceed to the next scheduled event. When it's time to reactivate this process, the main loop will `send` the current simulation time, which is assigned to `time`.

❸ This block will be repeated once for each trip.

❹ An `Event` signaling passenger pick up is yielded. The coroutine pauses here. When the time comes to reactivate this coroutine, the main loop will again `send` the current time.

❺ An `Event` signaling passenger drop off is yielded. The coroutine is suspended again, waiting for the main loop to send it the time of when it's reactivated.

❻ The `for` loop ends after the given number of trips, and a final `'going home'` event is yielded. The coroutine will suspend for the last time. When reactivated, it will be sent the time from the simulation main loop, but here I don't assign it to any variable because it will not be used.

❼ When the coroutine falls off the end, the generator object raises `StopIteration`.

You can "drive" a taxi yourself by calling `taxi_process` in the Python console[14]. Example 16-21 shows how:

Example 16-21. Driving the `taxi_process` coroutine.

```
>>> from taxi_sim import taxi_process
>>> taxi = taxi_process(ident=13, trips=2, start_time=0)  ❶
>>> next(taxi)  ❷
```

[14] The verb "drive" is commonly used to describe the operation of a coroutine: the client code drives the coroutine by sending it values. In the Example 16-21 the client code is what you type in the console.

```
Event(time=0, proc=13, action='leave garage')
>>> taxi.send(_.time + 7)  ❸
Event(time=7, proc=13, action='pick up passenger')  ❹
>>> taxi.send(_.time + 23)  ❺
Event(time=30, proc=13, action='drop off passenger')
>>> taxi.send(_.time + 5)  ❻
Event(time=35, proc=13, action='pick up passenger')
>>> taxi.send(_.time + 48)  ❼
Event(time=83, proc=13, action='drop off passenger')
>>> taxi.send(_.time + 1)
Event(time=84, proc=13, action='going home')  ❽
>>> taxi.send(_.time + 10)  ❾
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

❶ Create a generator object to represent a taxi with `ident=13` that will make two trips and start working at `t=0`.

❷ Prime the coroutine; it yields the initial event.

❸ We can now `send` it the current time. In the console, the `_` variable is bound to the last result; here I add `7` to the time, which means the `taxi` will spend 7 minutes searching for the first passenger.

❹ This is yielded by the `for` loop at the start of the first trip.

❺ Sending `_.time + 23` means the trip with the first passenger will last 23 minutes.

❻ Then the `taxi` will prowl for 5 minutes.

❼ The last trip will take 48 minutes.

❽ After two complete trips, the loop ends and the `'going home'` event is yielded.

❾ The next attempt to `send` to the coroutine causes it to fall through the end. When it returns, the interpreter raises `StopIteration`.

Note that in Example 16-21 I am using the console to emulate the simulation main loop. I get `.time` attribute of an `Event` yielded by the `taxi` coroutine, add an arbitrary number and use the sum in the next `taxi.send` call to reactivate it. In the simulation, the taxi coroutines are driven by the main loop in the `Simulator.run` method. The simulation "clock" is held in the `sim_time` variable, and is updated by the time of each event yielded.

To instantiate the `Simulator` class, the `main` function of `taxi_sim.py` builds a `taxis` dictionary like this:

```
    taxis = {i: taxi_process(i, (i + 1) * 2, i * DEPARTURE_INTERVAL)
             for i in range(num_taxis)}
    sim = Simulator(taxis)
```

`DEPARTURE_INTERVAL` is 5; if `num_taxis` is 3 as in the sample run, the lines above will do the same as:

```
taxis = {0: taxi_process(ident=0, trips=2, start_time=0),
         1: taxi_process(ident=1, trips=4, start_time=5),
         2: taxi_process(ident=2, trips=6, start_time=10)}
sim = Simulator(taxis)
```

Therefore, the values of the `taxis` dictionary will be three distinct generator objects with different parameters. For instance, taxi `1` will make `4` trips and begin looking for passengers at `start_time=5`. This `dict` is the only argument required to build a `Simulator` instance.

The `Simulator.__init__` method is shown in Example 16-22. The main data structures of `Simulator` are:

`self.events`

A `PriorityQueue` to hold `Event` instances. A `PriorityQueue` lets you `put` items, then `get` them ordered by `item[0]`; i.e. the `time` attribute in the case of our `Event` `namedtuple` objects.

`self.procs`

A `dict` mapping each process number an active process in the simulation — a generator object representing one taxi. This will be bound to a copy of `taxis dict` shown above.

Example 16-22. `taxi_sim.py`: `Simulator` class initializer.

```
class Simulator:

    def __init__(self, procs_map):
        self.events = queue.PriorityQueue()  ❶
        self.procs = dict(procs_map)  ❷
```

❶ The `PriorityQueue` to hold the scheduled events, ordered by increasing time.

❷ We get the `procs_map` argument as a `dict` (or any mapping), but build a `dict` from it, to have a local copy because when the simulation runs, each taxi that goes home is removed from `self.procs`, and we don't want to change the object passed by the user.

Priority queues are a fundamental building block of discrete event simulations: events are created in any order, placed in the queue, and later retrieved in order according to the scheduled time of each one. For example, the first two events placed in the queue may be:

```
Event(time=14, proc=0, action='pick up passenger')
Event(time=11, proc=1, action='pick up passenger')
```

This means that taxi `0` will take `14` minutes to pick up the first passenger, while taxi `1` — starting at `time=10` — will take `1` minute and pick up a passenger at `time=11`. If those two events are in the queue, the the first event the main loop gets from the priority queue will be `Event(time=11, proc=1, action='pick up passenger')`.

Now let's study the main algorithm of the simulation, the `Simulator.run` method. It's invoked by the `main` function right after the `Simulator` is instantiated, like this:

```
sim = Simulator(taxis)
sim.run(end_time)
```

The listing with callouts for the `Simulator` class is in Example 16-23, but here is an high level view of the algorithm implemented in `Simulator.run`:

1. Loop over processes representing taxis.
   a. Prime the coroutine for each taxi by calling `next()` on it. This will yield the first `Event` for each taxi.
   b. Put each event in the `self.events` queue of the `Simulator`.
2. Run the main loop of the simulation while `sim_time < end_time`.
   a. Check if `self.events` is empty; if so, break from the loop.
   b. Get the `current_event` from `self.events`. This will be the `Event` object with the lowest `time` in the `PriorityQueue`.
   c. Display the `Event`.
   d. Update the simulation time with the `time` attribute of the `current_event`.
   e. Send the `time` to the coroutine identified by the `proc` attribute of the `current_event`. The coroutine will yield the `next_event`.
   f. Schedule `next_event` by adding it to the `self.events` queue.

The complete `Simulator` class is Example 16-23.

Example 16-23. `taxi_sim.py`: `Simulator`, a bare-bones discrete event simulation class. Focus on the `run` method.

```
class Simulator:

    def __init__(self, procs_map):
        self.events = queue.PriorityQueue()
        self.procs = dict(procs_map)

    def run(self, end_time):  ❶
        """Schedule and display events until time is up"""
        # schedule the first event for each cab
        for _, proc in sorted(self.procs.items()):  ❷
            first_event = next(proc)  ❸
            self.events.put(first_event)  ❹

        # main loop of the simulation
        sim_time = 0  ❺
        while sim_time < end_time:  ❻
            if self.events.empty():  ❼
                print('*** end of events ***')
                break

            current_event = self.events.get()  ❽
            sim_time, proc_id, previous_action = current_event  ❾
            print('taxi:', proc_id, proc_id * '   ', current_event)  ❿
            active_proc = self.procs[proc_id]  ⓫
            next_time = sim_time + compute_duration(previous_action)  ⓬
            try:
                next_event = active_proc.send(next_time)  ⓭
            except StopIteration:
                del self.procs[proc_id]  ⓮
            else:
                self.events.put(next_event)  ⓯
        else:  ⓰
            msg = '*** end of simulation time: {} events pending ***'
            print(msg.format(self.events.qsize()))
```

❶ The simulation `end_time` is the only required argument for `run`.

❷ Use `sorted` to retrieve the `self.procs` items ordered by the key; we don't care about the key, so assign it to `_`.

❸ `next(proc)` primes each coroutine by advancing it to the first yield, so it's ready to be sent data. An `Event` is yielded.

❹ Add each event to the `self.events` `PriorityQueue`. The first event for each taxi is `'leave garage'`, as seen in the sample run (Example 16-20).

❺ Zero `sim_time`, the simulation clock.

❻ Main loop of the simulation: run while `sim_time` is less than the `end_time`.

❼ The main loop may also exit if there are no pending events in the queue.

❽ Get `Event` with the smallest `time` in the priority queue; this is the `current_event`.

❾ Unpack the `Event` data. This line updates the simulation clock, `sim_time`, to reflect the time when the event happened[15].

❿ Display the `Event`, identifying the taxi and adding indentation according to the taxi id.

⓫ Retrieve the coroutine for the active taxi from the `self.procs` dictionary.

[15] This is typical of a discrete event simulation: the simulation clock is not incremented by a fixed amount on each loop, but advances according to the duration of each event completed

⓬ Compute the next activation time by adding the sim_time and the result of calling compute_duration(…) with the previous action, e.g. 'pick up passenger', 'drop off passenger' etc.

⓭ Send the time to the taxi coroutine. The coroutine will yield the next_event or raise StopIteration it's finished.

⓮ If StopIteration is raised, delete the coroutine from the self.procs dictionary.

⓯ Otherwise, put the next_event in the queue.

⓰ If the loop exits because the simulation time passed, display the number of events pending (which may be zero by coincidence, sometimes).

Linking back to Chapter 15, note that the Simulator.run method Example 16-23 uses else blocks in two places that are not if statements:

- The main while loop has an else statement to report that the simulation ended because the end_time was reached — and not because there were no more events to process.
- The try statement at the bottom of the while loop tries to get a next_event by sending the next_time to the current taxi process, and if that is successful the else block puts the next_event into the self.events queue.

I believe the code in Simulator.run would be a bit harder to read without those else blocks.

The point of this example was to show a main loop processing events and driving coroutines by sending data to them. This is the basic idea behind asyncio, which we'll study in Chapter 18.

Chapter summary

Guido van Rossum wrote there are three different styles of code you can write using generators:

> There's the traditional "pull" style (iterators), "push" style (like the averaging example), and then there are "tasks" (Have you read Dave Beazley's coroutines tutorial yet?…)[16].

Chapter 14 was devoted to iterators, this chapter introduced coroutines used in "push style" and also as very simple "tasks" — the taxi processes in the simulation example. Chapter 18 will put them to use as asynchronous tasks in concurrent programming.

[16] Message to thread Yield-From: Finalization guarantees in the python-ideas mailing list. The David Beazley tutorial Guido refers to is A Curious Course on Coroutines and Concurrency

The running average example demonstrated a common use for a coroutine: as an accumulator processing items sent to it. We saw how a decorator can be applied to prime a coroutine, making it more convenient to use in some cases. But keep in mind that priming decorators are not compatible with some uses of coroutines. In particular, `yield from subgenerator()` assumes the `subgenerator` is not primed, and primes it automatically.

Accumulator coroutines can yield back partial results with each `send` method call, but they become more useful when they can return values, a feature that was added in Python 3.3 with PEP 380. We saw how the statement `return the_result` in a generator now raises `StopIteration(the_result)`, allowing the caller to retrieve `the_result` from the `value` attribute of the exception. This is a rather cumbersome way to retrieve coroutine results, but it's handled automatically by the `yield from` syntax introduced in PEP 380.

The coverage of `yield from` started with trivial examples using simple iterables, then moved to an example highlighting the three main components of any significant use of `yield from`: the delegating generator (defined by the use of `yield from` in its body), the subgenerator activated by `yield from` and the client code that actually drives the whole setup by sending values to the subgenerator through the pass-through channel established by `yield from` in the delegating generator. This section was wrapped by a look at the formal definition of `yield from` behavior as described in PEP 380 using English and Python-like pseudocode.

We closed the chapter with the discrete event simulation example, showing how generators can be used as an alternative to threads and callbacks to support concurrency. Although simple, the taxi simulation gives a first glimpse at how event-driven frameworks like Tornado and `asyncio` use a main loop to drive coroutines executing concurrent activities with a single thread of execution. In event-oriented programming with coroutines, each concurrent activity is carried out by a coroutine which repeatedly yields control back to the main loop, allowing other coroutines to be activated and move forward. This is a form of cooperative multi-tasking: coroutines voluntarily and explicitly yield control to the central scheduler. In contrast, threads implement preemptive multi-tasking. The scheduler can suspend threads at any time — even half-way through a statement — to give way to other threads.

One final note: this chapter adopted a broad, informal definition of a coroutine: a generator function driven by a client sending it data through `.send(…)` calls or `yield from`. This broad definition is the one used in PEP 342 — Coroutines via Enhanced Generators and in most existing Python books as I write this. The `asyncio` library we'll see in Chapter 18 is built on coroutines, but a stricter definition of coroutine is adopted there: `asyncio` coroutines are (usually) decorated with an `@asyncio.coroutine` decorator, and they are always driven by `yield from`, not by calling `.send(…)` directly on them. Of course, `asyncio` coroutines are driven by `next(…)` and `.send(…)` under the covers, but in user code we only use `yield from` to make them run.

Further reading

David Beazley is the ultimate authority on Python generators and coroutines. The *Python Cookbook 3e* (O'Reilly, 2013) he coauthored with Brian Jones has numerous recipes with coroutines. Beazley's PyCon tutorials on the subject are legendary for their depth and breadth. The first was at PyCon US 2008: Generator Tricks for Systems Programmers. PyCon US 2009 saw the legendary A Curious Course on Coroutines and Concurrency (hard-to-find video links for all 3 parts: part 1, part 2, part 3). His most recent tutorial from PyCon 2014 in Montréal was Generators: The Final Frontier in which he tackles more concurrency examples — so it's really more about topics in Chapter 18 of *Fluent Python*. Dave can't resist making brains explode in his classes, so in the last part of *The Final Frontier*, coroutines replace the classic Visitor pattern in an arithmetic expression evaluator.

Coroutines allow new ways of organizing code, and just as recursion or polymorphism (dynamic dispatch), it takes some time getting used to their possibilities. An interesting example of classic algorithm rewritten with coroutines is in the post Greedy algorithm with coroutines, by James Powell. You may also want to browse Popular recipes tagged "coroutine" in the ActiveState Code recipes database.

Paul Sokolovksy implemented `yield from` in his super lean MicroPython interpreter designed to run on microcontrollers. As he studied the feature, he created a great, detailed diagram to explain how `yield from` works, and shared it in the python-tulip mailing list. Sokolovksy was kind enough to allow me to copy the PDF to this book's site, where it has a more permanent URL: flupy.org/resources/yield-from.pdf.

As I write this, the vast majority of uses of `yield from` to be found are in `asyncio` itself or code that uses it. I spent a lot of time looking for examples of `yield from` that did not depend on `asyncio`. Greg Ewing — who penned PEP 380 and implemented `yield from` in CPython — published a few examples of its use: a `BinaryTree` class, a simple XML parser and a task scheduler.

Brett Slatkin's Effective Python (Addison-Wesley, 2015) has an excellent short chapter titled *Consider Coroutines to Run Many Functions Concurrently* (available online as a sample chapter). That chapter includes the best example of driving generators with `yield from` I've seen: an implementation of John Conway's Game of Life in which coroutines are used to manage the state of each cell as the game runs. The example code for *Effective Python* can be found in a Github repository. I refactored the code for the Game of Life example — separating the functions and classes that implement the game from the testing snippets used in Slatkin's book (original code). I also rewrote the tests as doctests, so you can see the output of the various coroutines and classes without running the script. The refactored example is posted as a Github gist: *http://bit.ly/coro_life*.

Other interesting examples of `yield from` without `asyncio` appear in a message to the Python Tutor list, Comparing two CSV files using Python by Peter Otten, and a Rock-Paper-Scissors game in Ian Ward's Iterables, Iterators and Generators tutorial published as an iPython notebook.

Guido van Rossum sent a long message to the `python-tulip` Google Group titled The difference between `yield` and `yield-from` that is worth reading. Nick Coghlan posted a heavily commented version of the `yield from` expansion in python-dev, Mar 21, 2009 in the same message, he wrote:

> Whether or not different people will find code using `yield from` difficult to understand or not will have more to do with their grasp of the concepts of cooperative multitasking in general more so than the underlying trickery involved in allowing truly nested generators.

PEP 492 — Coroutines with async and await syntax by Yury Selivanov proposes the addition of two keywords to Python: `async` and `await`. The former will be used with other existing keywords to define new language constructs. For example, `async def` will be used to define a coroutine, and `async for` to loop over asynchronous iterables with asynchronous iterators (implementing `__aiter__` and `__anext__`, coroutine versions of `__iter__` and `__next__`). The `await` keyword will do something similar to `yield from`, but will only be allowed inside coroutines defined with `async def` — where the use of `yield` and `yield from` will be forbidden. With new syntax, the PEP establishes a clear separation between the legacy generators that evolved into coroutine-like objects and a new breed of native coroutine objects with better language support thanks to infrastructure like the `async` and `await` keywords and several new special methods. Coroutines are poised to become really important in the future of Python and the language should be adapted to better integrate them.

Experimenting with discrete event simulations is a great way to become comfortable with cooperative multitasking. The Discrete event simulation article in the English language Wikipedia is a good place to start[17]. A short tutorial about writing discrete event simulations by hand (no special libraries) is Ashish Gupta's Writing a Discrete Event Simulation: ten easy lessons. The code is in Java so it's class-based and uses no coroutines, but can easily be ported to Python. Regardless of the code, the tutorial is a good short introduction to the terminology and components of a discrete event simulation. Converting Gupta's examples to Python classes and then to classes leveraging coroutines is a good exercise.

[17] Nowadays even tenured professors agree that the Wikipedia is a good place to start studying pretty much any subject in computer science. Not true about other subjects, but for computer science, Wikipedia rocks.

For a ready-to-use library in Python, using coroutines, there is SimPy. Its online documentation explains:

> SimPy is a process-based discrete-event simulation framework based on standard Python. Its event dispatcher is based on Python's generators and can also be used for asynchronous networking or to implement multi-agent systems (with both simulated and real communication).

Coroutines are not so new in Python but they were pretty much tied to niche application domains before asynchronous programming frameworks started supporting them, starting with Tornado. The addition of `yield from` in Python 3.3 and `asyncio` in Python 3.4 will likely boost the adoption of coroutines — and of Python 3.4 itself. However, Python 3.4 is less than a year old as I write this — so once you watch David Beazley's tutorials and cookbook examples on the subject, there isn't a whole lot of content out there that goes deep into Python coroutine programming. For now.

Soapbox

Raise from lambda

Table 16-1. Number of keywords in programming languages

keywords	language	comment
5	Smalltalk-80	Famous for its minimalist syntax.
25	Go	The language, not the game.
32	C	That's ANSI C. C99 has 37 keywords, C11 has 44.
33	Python	Python 2.7 has 31 keywords; Python 1.5 had 28.
41	Ruby	Keywords may be used as identifiers; e.g. `class` is also a method name.
49	Java	As in C, the names of the primitive types (`char`, `float` etc.) are reserved.
60	JavaScript	Includes all keywords from Java 1.0, many of which are unused.
65	PHP	Since PHP 5.3, 7 keywords were introduced, including `goto`, `trait` and `yield`.
85	C++	According to cppreference.com, C++11 added 10 keywords to the existing 75.
555	COBOL	I did not make this up. See this IBM ILE COBOL manual.
∞	Scheme	Anyone can define new keywords.

In programming languages, keywords establish the basic rules of control flow and expression evaluation.

A keyword in a language is like a piece in a board game. In the language of Chess, the keywords are ♔, ♕, ♖, ♗, ♘ and ♙. In the game of Go, it's ●.

Each player has only one type of piece in Go. But in the semantics of the game, adjacent pieces form larger, solid pieces of many different shapes, with emerging properties. Some arrangements of Go pieces are indestructible. Go is more expressive than Chess. In Go there are 361 possible opening moves, and an estimated `1e+170` legal positions; for Chess, the numbers are 20 opening moves `1e+50` positions.

Adding a new piece to Chess would be a radical change. Adding a new keyword in a programming language is also a radical change. So it makes sense for language designers to be wary of introducing keywords.

Python 3 added `nonlocal`, promoted `None`, `True` and `False` to keyword status, and dropped `print` and `exec`. It's very uncommon for a language to drop keywords as it evolves. Table 16-1 lists some languages, ordered by number of keywords.

Scheme inherited from Lisp a macro facility that allows anyone to create special forms adding new control structures and evaluation rules to the language. The user-defined identifiers of those forms are called "syntactic keywords". The Scheme R5RS standard states "There are no reserved identifiers" (page 45 of the standard), but a typical implementation such as MIT/GNU Scheme comes with 34 syntactic keywords predefined, such as `if`, `lambda` and `define-syntax` — the keyword that lets you conjure new keywords[18].

Python is like Chess, and Scheme is like Go (the game).

Now, back to Python syntax. I think Guido is too conservative with keywords. It's nice to have a small set of them, and adding new keywords potentially breaks a lot of code. But the use of `else` in loops reveals a recurring problem: the overloading of existing keywords when a new one would be a better choice. In the context of `for`, `while` and `try`, a new `then` keyword would be preferable to abusing `else`.

The most serious manifestation of this problem is the overloading of `def`: it's now used to define functions, generators and coroutines — objects that are too different to share the same declaration syntax[19].

The introduction of `yield from` is particularly worrying. Once again, I believe Python users would be best served by a new keyword. Even worse, this starts a new trend: chaining existing keywords to create new syntax, instead of adding sensible, descriptive keywords. I fear one day we may be poring over the meaning of `raise from lambda`.

Breaking news

As I wrap up this book's technical review process, it seems Yury Selivanov's PEP 492 — Coroutines with async and await syntax is on the way to being accepted for implementation in Python 3.5 already! The PEP has the support of Guido van Rossum and Victor Stinner, respectively the author and a leading maintainer of the `asyncio` library which would be the main use case for the new syntax. In response to Selivanov's message to

[18] The Value Of Syntax? is an interesting discussion about extensible syntax and programming language usability. The forum, Lambda the Ultimate, is a watering hole for programming language geeks.

[19] A highly recommended post related to this issue in the context of JavaScript, Python and other languages is What Color is Your Function?, by Bob Nystrom.

python-ideas, Guido even hints at delaying the release of Python 3.5 so the PEP can be implemented.

Of course, this would put to rest most of the complaints I expressed above.

CHAPTER 17

Concurrency with futures

> The people bashing threads are typically system programmers which have in mind use cases that the typical application programmer will never encounter in her life. […] In 99% of the use cases an application programmer is likely to run into, the simple pattern of spawning a bunch of independent threads and collecting the results in a queue is everything one needs to know[1].
>
> — Michele Simionato
> *Python deep thinker*

This chapter focuses on the `concurrent.futures` library introduced in Python 3.2, but also available for Python 2.5 and newer as the `futures` package on PyPI. This library encapsulates the pattern described by Michele Simionato in the quote above, making it almost trivial to use.

Here I also introduce the concept of “futures" — objects representing the asynchronous execution of an operation. This powerful idea is the foundation not only of `concurrent.futures` but also of the `asyncio` package, which we’ll cover in Chapter 18.

We’ll start with a motivating example.

Example: Web downloads in three styles

To handle network I/O efficiently you need concurrency, because it involves high latency — so instead of wasting CPU cycles waiting, it’s better to do something else until a response comes back from the network.

To make this last point with code, I wrote three simple programs to download images of 20 country flags from the Web. The first one, `flags.py`, runs sequentially: it only

[1] From Michele Simionato’s post Threads, processes and concurrency in Python: some thoughts, subtitled “Removing the hype around the multicore (non) revolution and some (hopefully) sensible comment about threads and other forms of concurrency.”

requests the next image when the previous one is downloaded and saved to disk. The other two scripts make concurrent downloads: they request all images practically at the same time, and save the files as they arrive. The `flags_threadpool.py` script uses the `concurrent.futures` package, while `flags_asyncio.py` uses `asyncio`.

Example 17-1 shows the result of running the three scripts, three times each. I also posted a 1'13" video on YouTube so you can watch them running while an OSX Finder window displays the flags as they are saved. The scripts are downloading images from `flupy.org`, which is behind a CDN, so you may see slower results in the first runs. The results in Example 17-1 were obtained after several runs, so the CDN cache was warm.

Example 17-1. Three typical runs of the scripts `flags.py`, `flags_threadpool.py`, `flags_asyncio.py`.

```
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN  ❶
20 flags downloaded in 7.26s  ❷
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 7.20s
$ python3 flags.py
BD BR CD CN DE EG ET FR ID IN IR JP MX NG PH PK RU TR US VN
20 flags downloaded in 7.09s
$ python3 flags_threadpool.py
DE BD CN JP ID EG NG BR RU CD IR MX US PH FR PK VN IN ET TR
20 flags downloaded in 1.37s  ❸
$ python3 flags_threadpool.py
EG BR FR IN BD JP DE RU PK PH CD MX ID US NG TR CN VN ET IR
20 flags downloaded in 1.60s
$ python3 flags_threadpool.py
BD DE EG CN ID RU IN VN ET MX FR CD NG US JP TR PK BR IR PH
20 flags downloaded in 1.22s
$ python3 flags_asyncio.py  ❹
BD BR IN ID TR DE CN US IR PK PH FR RU NG VN ET MX EG JP CD
20 flags downloaded in 1.36s
$ python3 flags_asyncio.py
RU CN BR IN FR BD TR EG VN IR PH CD ET ID NG DE JP PK MX US
20 flags downloaded in 1.27s
$ python3 flags_asyncio.py
RU IN ID DE BR VN PK MX US IR ET EG NG BD FR CN JP PH CD TR  ❺
20 flags downloaded in 1.42s
```

❶ The output for each run starts with the country codes of the flags as they are downloaded, and ends with a message stating the elapsed time.

❷ It took `flags.py` an average 7.18s to download 20 images.

❸ The average for `flags_threadpool.py` was 1.40s.

❹ For `flags_asyncio.py`, 1.35 was the average time.

❺ Note the order of the country codes: the downloads happened in a different order every time with the concurrent scripts.

The difference in performance between the concurrent scripts is not significant, but they are both more than 5 times faster than the sequential script. And this for a fairly small task. If you scale the task to hundreds of downloads, the concurrent scripts can outpace the sequential one by a factor or 20 or more.

While testing concurrent HTTP clients on public Web you may inadvertently launch a DOS (Denial of Service) attack, or be suspected of doing so. In the case of Example 17-1 it's OK to do it because those scripts are hard-coded to make only 20 requests. For testing non-trivial HTTP clients, you should set up your own test server. The `17-futures/countries/README.rst` file in the *Fluent Python* `example-code` Github repository has instructions for setting a local Nginx server.

Now let's study the implementations of two of the scripts tested in Example 17-1: `flags.py` and `flags_threadpool.py`. I will leave the third script, `flags_asyncio.py` for Chapter 18, but I wanted to demonstrate all three together to make a point: regardless of the concurrency strategy you use — threads or `asyncio` — you'll see vastly improved throughput over sequential code in I/O bound applications, if you code it properly.

On to the code.

A sequential download script

Example 17-2 is not very interesting, but we'll reuse most of its code and settings to implement the concurrent scripts, so it deserves some attention.

For clarity, there is no error handling in Example 17-2. We will deal with exceptions later, but here we want to focus on the basic structure of the code, to make it easier to contrast this script with the concurrent ones.

Example 17-2. `flags.py`: sequential download script. Some functions will be reused by the other scripts.

```
import os
import time
import sys

import requests   ❶
```

```
POP20_CC = ('CN IN US ID BR PK NG BD RU JP '
            'MX PH VN ET EG DE IR TR CD FR').split()  ❷

BASE_URL = 'http://flupy.org/data/flags'  ❸

DEST_DIR = 'downloads/'  ❹


def save_flag(img, filename):  ❺
    path = os.path.join(DEST_DIR, filename)
    with open(path, 'wb') as fp:
        fp.write(img)


def get_flag(cc):  ❻
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = requests.get(url)
    return resp.content


def show(text):  ❼
    print(text, end=' ')
    sys.stdout.flush()


def download_many(cc_list):  ❽
    for cc in sorted(cc_list):  ❾
        image = get_flag(cc)
        show(cc)
        save_flag(image, cc.lower() + '.gif')

    return len(cc_list)


def main(download_many):  ❿
    t0 = time.time()
    count = download_many(POP20_CC)
    elapsed = time.time() - t0
    msg = '\n{} flags downloaded in {:.2f}s'
    print(msg.format(count, elapsed))


if __name__ == '__main__':
    main(download_many)  ⓫
```

❶ Import the `requests` library; it's not part of the standard library, so by convention we import it after the standard library modules `os`, `time` and `sys`, and separate it from them with a blank line.

❷ List of the ISO 3166 country codes for the 20 most populous countries in order of decreasing population.

❸ The Web site with the flag images[2].

❹ Local directory where the images are saved.

❺ Simply save the `img` (a byte sequence) to `filename` in the `DEST_DIR`.

❻ Given a country code, build the URL and download the image, returning the binary contents of the response.

❼ Display a string and flush `sys.stdout` so we can see progress in a one-line display; this is needed because Python normally waits for a line break to flush the `stdout` buffer.

❽ `download_many` is the key function to compare with the concurrent implementations.

❾ loop over the list of country codes in alphabetical order, to make it clear that the ordering is preserved in the output; return the number of country codes downloaded.

❿ `main` records and reports the elapsed time after running `download_many`;

⓫ `main` must be called with the function that will make the downloads; we pass the `download_many` function as an argument so that `main` can be used as a library function with other implementations of `download_many` in the next examples.

The `requests` library by Kenneth Reitz is available on PyPI and is more powerful and easier to use than the `urllib.request` module from the Python 3 standard library. In fact, `requests` is considered a model Pythonic API. It is also compatible with Python 2.6 and up, while the `urllib2` from Python 2 was moved and renamed in Python 3, so it's more convenient to use `requests` regardless of the Python version you're targeting.

There's really nothing new to `flags.py`. It serves as a baseline for comparing the other scripts and I used it as a library to avoid redundant code when implementing them. Now let's see a reimplementation using `concurrent.futures`.

Downloading with `concurrent.futures`

The main features of the `concurrent.futures` package are the `ThreadPoolExecutor` and `ProcessPoolExecutor` classes, which implement an interface that allows you to submit callables for execution in different threads or processes, respectively. The classes manage an internal pool of worker threads or processes, and a queue of tasks to be

[2] The images are originally from the CIA World Factbook, a public-domain, US government publication. I copied them to my site to avoid the risk of launching a DOS attack on CIA.gov.

executed. But the interface is very high level and we don't need to know about any of those details for a simple use case like our flag downloads.

Example 17-3 shows the easiest way to implement the downloads concurrently, using the `ThreadPoolExecutor.map` method.

Example 17-3. `flags_threadpool.py`: threaded download script using `futures.ThreadpPoolExecutor`.

```
from concurrent import futures

from flags import save_flag, get_flag, show, main   ❶

MAX_WORKERS = 20   ❷


def download_one(cc):   ❸
    image = get_flag(cc)
    show(cc)
    save_flag(image, cc.lower() + '.gif')
    return cc


def download_many(cc_list):
    workers = min(MAX_WORKERS, len(cc_list))   ❹
    with futures.ThreadPoolExecutor(workers) as executor:   ❺
        res = executor.map(download_one, sorted(cc_list))   ❻

    return len(list(res))   ❼


if __name__ == '__main__':
    main(download_many)   ❽
```

❶ Reuse some functions from the `flags` module (Example 17-2).

❷ Maximum number of threads to be used in the `ThreadPoolExecutor`.

❸ Function to download a single image; this is what each thread will execute.

❹ Set the number of worker threads: use the smaller number between the maximum we want to allow (*MAX_WORKERS*) and the actual items to be processed, so no unnecessary threads are created.

❺ Instantiate the `ThreadPoolExecutor` with that number of worker threads; the `executor.__exit__` method will call `executor.shutdown(wait=True)`, which will block until all threads are done.

❻ The `map` method is similar to the `map` built-in, except that the `download_one` function will be called concurrently from multiple threads; it returns a generator that can be iterated over to retrieve the value returned by each function.

❼ Return the number of results obtained; if any of the threaded calls raised an exception, that exception would be raised here as the implicit next() call tried to retrieve the corresponding return value from the iterator.

❽ Call the main function from the flags module, passing the enhanced version of download_many.

Note that the download_one function from Example 17-3 is essentially the body of the for loop in the download_many function from Example 17-2. This is a common refactoring when writing concurrent code: turning the body of a sequential for loop into a function to be called concurrently.

The library is called concurrency.futures yet there are no futures to be seen in Example 17-3, so you may be wondering where they are. The next section explains.

Where are the futures?

Futures are essential components in the internals of concurrent.futures and of asyncio, but as users of these libraries we sometimes don't see them. Example 17-3 leverages futures behind the scenes, but the code I wrote does not touch them directly. This section is an overview of futures, with an example that shows them in action.

As of Python 3.4 — there are two classes named Future in the standard library: concurrent.futures.Future and asyncio.Future. They serve the same purpose: an instance of either Future class represents a deferred computation that may or may not have completed. This is similar to the the Deferred class in Twisted, the Future class in Tornado, and Promise objects in various JavaScript libraries.

Futures encapsulate pending operations so that they can be put in queues, their state of completion can be queried, and their results (or exceptions) can be retrieved when available.

An important thing to know about futures in general is that you and I should not create them: they are meant to be instantiated exclusively by the concurrency framework, be it concurrent.futures or asyncio. It's easy to understand why: a Future represents something that will eventually happen, and the only way to be sure that something will happen is to schedule its execution. Therefore, concurrent.futures.Future instances are created only as the result of scheduling something for execution with a concurrent.futures.Executor subclass. For example, the Executor.submit() method takes a callable, schedules it to run, and returns a future.

Client code is not supposed to change the state of a future: the concurrency framework changes the state of a future when the computation it represents is done, and we can't control when that happens.

Both types of Future have a .done() method that is non-blocking and returns a boolean that tells you whether the callable linked to that future has executed or not. Instead of asking whether a future is done, client code usually asks to be notified. That's why both Future classes have an .add_done_callback() method: you give it a callable, and the callable will be invoked with the future as the single argument when the future is done.

There is also a .result() method which works the same in both classes when the future is done: it returns the result of the callable, or re-raises whatever exception might have been thrown when the callable was executed. However, when the future is not done the behavior of the result method is very different between the two flavors of Future. In a concurrency.futures.Future instance, invoking f.result() will block caller's thread until the result is ready. An optional timeout argument can be passed, and if the future is not done in the specified time, a TimeoutError exception is raised. In "asyncio.Future: non-blocking by design" on page 547 we'll see that the asyncio.Future.result method does not support timeout, and the preferred way to get the result of futures in that library is to use yield from — which doesn't work with concurrency.futures.Future instances.

Several functions in both libraries return futures; others use them in their implementation in a way that is transparent to the user. An example of the latter is the the Executor.map we saw in Example 17-3: it returns an iterator in which __next__ calls the result method of each future, so what we get are the results of the futures, and not the futures themselves.

To get a practical look at futures, we can rewrite Example 17-3 to use the concurrent.futures.as_completed function, which takes an iterable of futures and returns an iterator that yields futures as they are done.

Using futures.as_completed requires changes to the download_many function only. The higher level executor.map call is replaced by two for loops: one to create and schedule the futures, the other to retrieve their results. While we are at it, we'll add a few print calls to display each future before and after it's done. Example 17-4 shows the code for a new download_many function. The code for download_many increased from 5 to 17 lines, but now we get to inspect the mysterious futures. The remaining functions are the same as in Example 17-3.

Example 17-4. flags_threadpool_ac.py: replacing executor.map with executor.submit and futures.as_completed in the download_many function.

```
def download_many(cc_list):
    cc_list = cc_list[:5]  ❶
    with futures.ThreadPoolExecutor(max_workers=3) as executor:  ❷
        to_do = []
        for cc in sorted(cc_list):  ❸
            future = executor.submit(download_one, cc)  ❹
            to_do.append(future)  ❺
            msg = 'Scheduled for {}: {}'
            print(msg.format(cc, future))  ❻

        results = []
        for future in futures.as_completed(to_do):  ❼
            res = future.result()  ❽
            msg = '{} result: {!r}'
            print(msg.format(future, res))  ❾
            results.append(res)

    return len(results)
```

❶ For this demonstration, use only the top 5 most populous countries.

❷ Hard code `max_workers` to 3 so we can observe pending futures in the output.

❸ Iterate over country codes alphabetically, to make it clear that results arrive out of order.

❹ `executor.submit` schedules the callable to be executed, and returns a `future` representing this pending operation.

❺ Store each `future` so we can later retrieve them with `as_completed`.

❻ Display a message with the country code and the respective `future`.

❼ `as_completed` yields futures as they are completed.

❽ Get the result of this `future`.

❾ Display the `future` and its result.

Note that the `future.result()` call will never block in this example because the `future` is coming out of `as_completed`. Example 17-5 shows the output of one run of Example 17-4.

Example 17-5. Output of `flags_threadpool_ac.py`.

```
$ python3 flags_threadpool_ac.py
Scheduled for BR: <Future at 0x100791518 state=running>  ❶
Scheduled for CN: <Future at 0x100791710 state=running>
Scheduled for ID: <Future at 0x100791a90 state=running>
Scheduled for IN: <Future at 0x101807080 state=pending>  ❷
Scheduled for US: <Future at 0x101807128 state=pending>
CN <Future at 0x100791710 state=finished returned str> result: 'CN'  ❸
BR ID <Future at 0x100791518 state=finished returned str> result: 'BR'  ❹
<Future at 0x100791a90 state=finished returned str> result: 'ID'
IN <Future at 0x101807080 state=finished returned str> result: 'IN'
US <Future at 0x101807128 state=finished returned str> result: 'US'

5 flags downloaded in 0.70s
```

❶ The futures are scheduled in alphabetical order; the `repr()` of a future shows its state: the first 3 are `running`, because there are 3 worker threads.

❷ The last two futures are `pending`, waiting for worker threads.

❸ The first `CN` here is the output of `download_one` in a worker thread; the rest of the line is the output of `download_many`.

❹ Here two threads output codes before `download_many` in the main thread can display the result of the first thread.

> If you run `flags_threadpool_ac.py` several times you'll see the order of the results varying. Increasing the `max_workers` argument to 5 will increase the variation in the order of the results. Decreasing it to one will make this code run sequentially, and the order of the results will always be the order of the `submit` calls.

We saw two variants of the download script using `concurrent.futures`: Example 17-3 with `ThreadPoolExecutor.map` and Example 17-4 with `futures.as_completed`. If you are curious about the code for `flags_asyncio.py`, you may peek at Example 18-5 in Chapter 18.

Strictly speaking, none of the concurrent scripts we tested so far can perform downloads in parallel. The `concurrent.futures` examples are limited by the GIL, and the `flags_asyncio.py` is single-threaded.

At this point you may have questions about the informal benchmarks we just did:

- How can `flags_threadpool.py` perform 5x faster than `flags.py` if Python threads are limited by a GIL (Global Interpreter Lock) that only lets one thread run at any time?
- How can `flags_asyncio.py` perform 5x faster than `flags.py` when both are single threaded?

I will answer the second question in "Running circles around blocking calls" on page 554.

Read on to understand why the GIL is nearly harmless with I/O bound processing.

Blocking I/O and the GIL

The CPython interpreter is not thread-safe internally, so it has a Global Interpreter Lock (GIL) which allows only one thread at a time to execute Python bytecodes. That's why a single Python process usually cannot use multiple CPU cores at the same time[3].

When we write Python code we have no control over the GIL, but a built-in function or an extension written in C can release the GIL while running time consuming tasks. In fact, a Python library coded in C can manage the GIL, launch its own OS threads and take advantage of all available CPU cores. This complicates the code of the library considerably, and most library authors don't do it.

However, all standard library functions that perform blocking I/O release the GIL when waiting for a result from the OS. This means Python programs that are I/O bound can benefit from using threads at the Python level: while one Python thread is waiting for a response from the network, the blocked I/O function releases the GIL so another thread can run.

That's why David Beazley says: "Python threads are great at doing nothing[4]."

Every blocking I/O function in the Python standard library releases the GIL, allowing other threads to run. The `time.sleep()` function also releases the GIL. Therefore, Python threads are perfectly usable in I/O bound applications, despite the GIL.

Now let's take a brief look at a simple way to work around the GIL for CPU-bound jobs using `concurrent.futures`.

Launching processes with `concurrent.futures`

The documentation page for the `concurrent.futures` package is subtitled "Launching parallel tasks". The package does enable truly parallel computations because it supports distributing work among multiple Python processes using the `ProcessPoolExecutor` class — thus bypassing the GIL and leveraging all available CPU cores, if you need to do CPU-bound processing.

Both `ProcessPoolExecutor` and `ThreadPoolExecutor` implement the generic `Executor` interface, so it's very easy to switch from a thread-based to a process-based solution using `concurrent.futures`.

[3] This is a limitation of the CPython interpreter, not of the Python language itself. Jython and IronPython are not limited in this way; but Pypy, the fastest Python interpreter available, also has a GIL.

[4] Slide 106 of Generators: The Final Frontier.

There is no advantage in using a ProcessPoolExecutor for the flags download example or any I/O bound job. It's easy to verify this: just change these lines in Example 17-3:

```
def download_many(cc_list):
    workers = min(MAX_WORKERS, len(cc_list))
    with futures.ThreadPoolExecutor(workers) as executor:
```

To this:

```
def download_many(cc_list):
    with futures.ProcessPoolExecutor() as executor:
```

For simple uses, the only notable difference between the two concrete executor classes is that ThreadPoolExecutor.__init__ requires a max_workers argument setting the number of threads in the pool. That is an optional argument in ProcessPoolExecutor, and most of the time we don't use it — the default is the number of CPUs returned by os.cpu_count(). This makes sense: for CPU bound processing it makes no sense to ask for more workers than CPUs. On the other hand, for I/O bound processing you may use 10, 100 or 1000 threads in a ThreadPoolExecutor; the best number depends on what you're doing and the available memory, and finding the optimal number will require careful testing.

A few tests revealed that the average time to download the 20 flags increased to 1.8s with a ProcessPoolExecutor — compared to 1.4s in the original ThreadPoolExecutor version. The main reason for this is likely to be the limit of 4 concurrent downloads on my 4-core machine, against 20 workers in the thread pool version.

The value of ProcessPoolExecutor is in CPU-intensive jobs. I did some performance tests with a couple of CPU-bound scripts:

arcfour_futures.py
: Encrypt and decrypt a dozen byte arrays with sizes from 149 KB to 384 KB using a pure-Python implementation of the RC4 algorithm (listing: Example A-7).

sha_futures.py
: Compute the SHA-256 hash of a dozen 1 MB byte arrays with the standard library hashlib package, which uses the OpenSSL library (listing: Example A-9).

Neither of these scripts do I/O except to display summary results. They build and process all their data in memory, so I/O does not interfere with their execution time.

Table 17-1 shows the average timings I got after 64 runs of the RC4 example and 48 runs of the SHA example. The timings include the time to actually spawn the worker processes.

Table 17-1. Time and speedup factor for the RC4 and SHA examples with 1 to 4 workers on an Intel Core i7 2.7 GHz quad-core machine, using Python 3.4.

workers	RC4 time	RC4 factor	SHA time	SHA factor
1	11.48s	1.00x	22.66s	1.00x
2	8.65s	1.33x	14.90s	1.52x
3	6.04s	1.90x	11.91s	1.90x
4	5.58s	2.06x	10.89s	2.08x

In summary, for cryptographic algorithms you can expect to double the performance by spawning 4 worker processes with a `ProcessPoolExecutor`, if you have 4 CPU cores.

For the pure-Python RC4 example you can get results 3.8 times faster if you use PyPy and 4 workers, compared with CPython and 4 workers. That's a speedup of 7.8 times in relation to the baseline of 1 worker with CPython in Table 17-1.

> If you are doing CPU-intensive work in Python, you should try PyPy. The `arcfour_futures.py` example ran from 3.8 to 5.1 times faster using PyPy, depending on the number of workers used. I tested with PyPy 2.4.0 which is compatible with Python 3.2.5, so it has `concurrent.futures` in the standard library.

Now let's investigate the behavior of a thread pool with a demonstration program that launches a pool with three workers, running five callables that output timestamped messages.

Experimenting with `Executor.map`

The simplest way to run several callables concurrently is with the `Executor.map` function we first saw in Example 17-3. Example 17-6 is a script to demonstrate how `Executor.map` works in some detail. It's output appears in Example 17-7.

Example 17-6. `demo_executor_map.py`: Simple demonstration of the `map` method of `ThreadPoolExecutor`.

```
from time import sleep, strftime
from concurrent import futures


def display(*args):  ❶
    print(strftime('[%H:%M:%S]'), end=' ')
    print(*args)


def loiter(n):  ❷
```

```
    msg = '{}loiter({}): doing nothing for {}s...'
    display(msg.format('\t'*n, n, n))
    sleep(n)
    msg = '{}loiter({}): done.'
    display(msg.format('\t'*n, n))
    return n * 10  ❸


def main():
    display('Script starting.')
    executor = futures.ThreadPoolExecutor(max_workers=3)  ❹
    results = executor.map(loiter, range(5))  ❺
    display('results:', results)  # ❻.
    display('Waiting for individual results:')
    for i, result in enumerate(results):  ❼
        display('result {}: {}'.format(i, result))


main()
```

❶ This function simply prints whatever arguments it gets, preceded by a timestamp in the format [HH:MM:SS].

❷ loiter does nothing except display a message when it starts, sleep for n seconds, then display a message when it ends; tabs are used to indent the messages according to the value of n.

❸ loiter returns n * 10 so we can see how to collect results.

❹ Create a ThreadPoolExecutor with 3 threads.

❺ Submit five tasks to the executor; because there are only 3 threads, only three of those tasks will start immediately: the calls loiter(0), loiter(1) and loiter(2); this is a non-blocking call.

❻ Immediately display the results of invoking executor.map: it's a generator, as the output in Example 17-7 shows.

❼ The enumerate call in the for loop will implicitly invoke next(results), which in turn will invoke _f.result() on the (internal) _f future representing the first call, loiter(0). The result method will block until the future is done, therefore each iteration in this loop will have to wait for the next result to be ready.

I encourage you to run Example 17-6 and see the display being updated incrementally. While you are at it, play with the max_workers argument for the ThreadPoolExecutor and with the range function that produces the arguments for the executor.map call — or replace it with lists of hand-picked values to create different delays.

Example 17-7 shows a sample run of Example 17-6.

Example 17-7. Sample run of `demo_executor_map.py` from Example 17-6.

```
$ python3 demo_executor_map.py
[15:56:50] Script starting. ❶
[15:56:50] loiter(0): doing nothing for 0s... ❷
[15:56:50] loiter(0): done.
[15:56:50]      loiter(1): doing nothing for 1s... ❸
[15:56:50]              loiter(2): doing nothing for 2s...
[15:56:50] results: <generator object result_iterator at 0x106517168> ❹
[15:56:50]                      loiter(3): doing nothing for 3s... ❺
[15:56:50] Waiting for individual results:
[15:56:50] result 0: 0 ❻
[15:56:51]      loiter(1): done. ❼
[15:56:51]                              loiter(4): doing nothing for 4s...
[15:56:51] result 1: 10 ❽
[15:56:52]              loiter(2): done. ❾
[15:56:52] result 2: 20
[15:56:53]                      loiter(3): done.
[15:56:53] result 3: 30
[15:56:55]                              loiter(4): done. ❿
[15:56:55] result 4: 40
```

❶ This run started at 15:56:50.

❷ The first thread executes `loiter(0)`, so it will sleep for 0s and return even before the second thread has a chance to start, but YMMV[5].

❸ `loiter(1)` and `loiter(2)` start immediately (since the thread pool has 3 workers, it can run 3 functions concurrently).

❹ This shows that the `results` returned by `executor.map` is a generator; nothing so far would block, regardless of the number of tasks and the `max_workers` setting.

❺ Because `loiter(0)` is done, the first worker is now available to start the fourth thread for `loiter(3)`.

❻ This is where execution may block, depending on the parameters given to the `loiter` calls: the `__next__` method of the `results` generator must wait until the first future is complete. In this case, it won't block because the call to `loiter(0)` finished before this loop started. Note that everything up to this point happened within the same second: 15:56:50.

❼ `loiter(1)` is done one second later, at 15:56:51. The thread is freed to start `loiter(4)`.

[5] Your Mileage May Vary: with threads, you never know the exact sequencing of events that should happen practically at the same time; it's possible that, in another machine, you see `loiter(1)` starting before `loiter(0)` finishes, particularly because `sleep` always releases the GIL so Python may switch to another thread even if you sleep for 0s.

❽ The result of `loiter(1)` is shown: 10. Now the `for` loop will block waiting for the result of `loiter(2)`.

❾ The pattern repeats: `loiter(2)` is done, it's result is shown; same with `loiter(3)`.

❿ There is a 2s delay until `loiter(4)` is done, because it started at 15:56:51 and did nothing for 4s.

The `Executor.map` function is easy to use but it has a feature that may or may not be helpful, depending on your needs: it returns the results exactly in the same order as the calls are started: if the first call takes 10s to produce a result, and the others take 1s each, your code will block for 10s as it tries to retrieve the first result of the generator returned by `map`. After that, you'll get the remaining results without blocking because they will be done. That's OK when you must have all the results before proceeding, but often it's preferable to get the results as they are ready, regardless of the order they were submitted. To do that, you need a combination of the `Executor.submit` method and the `futures.as_completed` function, as we saw in Example 17-4. We'll come back to this technique in "Using `futures.as_completed`" on page 529.

> The combination of `executor.submit` and `futures.as_completed` is more flexible than `executor.map` because you can `submit` different callables and arguments, while `executor.map` is designed to run the same callable on the different arguments. In addition, the set of futures you pass to `futures.as_completed` may come from more than one executor — perhaps some were created by a `ThreadPoolExecutor` instance while others are from a `ProcessPoolExecutor`.

In the next section we will resume the flag download examples with new requirements that will force us to iterate over the results of `futures.as_completed` instead of using `executor.map`.

Downloads with progress display and error handling

As mentioned, the scripts in "Example: Web downloads in three styles" on page 507 have no error handling to make them easier to read and to contrast the structure of the three approaches: sequential, threaded and asynchronous.

In order to test the handling of a variety of error conditions, I created the `flags2` examples:

`flags2_common.py`
: This module contains common functions and settings used by all `flags2` examples, including a `main` function which takes care of command-line parsing, timing and

reporting results. This is really support code, not directly relevant to the subject of this chapter, so the source code is in Appendix A, Example A-10.

`flags2_sequential.py`
: A sequential HTTP client with proper error handling and progress bar display. Its `download_one` function is also used by `flags2_threadpool.py`.

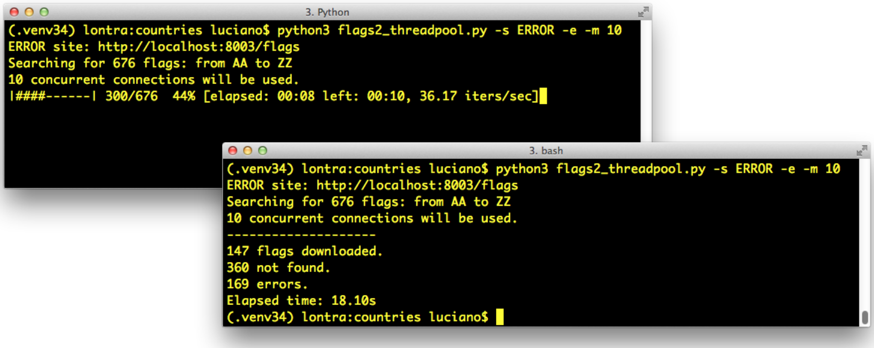
`flags2_threadpool.py`
: Concurrent HTTP client based on `futures.ThreadPoolExecutor` to demonstrate error handling and integration of the progress bar.

`flags2_asyncio.py`
: Same functionality as previous example but implemented with `asyncio` and `aiohttp`. This will be covered in "Enhancing the `asyncio` downloader script" on page 556, in Chapter 18

Be careful when testing concurrent clients

When testing concurrent HTTP clients on public HTTP servers you may generate many requests per second, and that's how DOS attacks (Denial of Service) are made. We don't want to attack anyone, just learn how to build high-performance clients. Carefully throttle your clients when hitting public servers. For high concurrency experiments, setup a local HTTP server for testing. Instructions for doing it are in the `README.rst` file in the `17-futures/countries/` directory of the Fluent Python example code repository.

The most visible feature of the `flags2` examples is that they have an animated, text-mode progress bar implemented with the TQDM package. I posted a 1'48" video on YouTube to show the progress bar and contrast the speed of the three `flags2` scripts. In the video, I start with the sequential download, but I interrupt it after 32s because it was going to take more than 5 minutes to hit on 676 URLs and get 194 flags; I then run the threaded and `asyncio` scripts three times each, and every time they complete the job in 6s or less — i.e. more than 60 times faster. Figure 17-1 shows two screenshots: during and after running `flags2_threadpool.py`.



```
(.venv34) lontra:countries luciano$ python3 flags2_threadpool.py -s ERROR -e -m 10
ERROR site: http://localhost:8003/flags
Searching for 676 flags: from AA to ZZ
10 concurrent connections will be used.
|####------| 300/676  44% [elapsed: 00:08 left: 00:10, 36.17 iters/sec]
```

```
(.venv34) lontra:countries luciano$ python3 flags2_threadpool.py -s ERROR -e -m 10
ERROR site: http://localhost:8003/flags
Searching for 676 flags: from AA to ZZ
10 concurrent connections will be used.
--------------------
147 flags downloaded.
360 not found.
169 errors.
Elapsed time: 18.10s
(.venv34) lontra:countries luciano$
```

Figure 17-1. Top left: `flags2_threadpool.py` running with live progress bar generated by `tqdm`; bottom right: same terminal window after the script is finished.

TQDM is very easy to use, the simplest example appears in an animated `.gif` in the project's README.md. If you type the code below in the Python console after installing the `tqdm` package, you'll see an animated progress bar were the comment is:

```
>>> import time
>>> from tqdm import tqdm
>>> for i in tqdm(range(1000)):
...     time.sleep(.01)
...
>>> # -> progress bar will appear here <-
```

Besides the neat effect, the `tqdm` function is also interesting conceptually: it consumes any iterable and produces an iterator which, while it's consumed, displays the progress bar and estimates the remaining time to complete all iterations. To compute that estimate, `tqdm` needs to get an iterable that has a `len`, or receive as a second argument the expected number of items. Integrating TQDM with our `flags2` examples provide an opportunity to look deeper into how the concurrent scripts actually work, by forcing us to use the `futures.as_completed` and the `asyncio.as_completed` functions so that `tqdm` can display progress as each future is completed.

The other feature of the `flags2` example is a command-line interface. All three scripts accept the same options, and you can see them by running any of the scripts with the `-h` option. Example 17-8 shows the help text.

Example 17-8. Help screen for the scripts in the `flags2` series.

```
$ python3 flags2_threadpool.py -h
usage: flags2_threadpool.py [-h] [-a] [-e] [-l N] [-m CONCURRENT] [-s LABEL]
                            [-v]
                            [CC [CC ...]]
```

```
Download flags for country codes. Default: top 20 countries by population.

positional arguments:
  CC                    country code or 1st letter (eg. B for BA...BZ)

optional arguments:
  -h, --help            show this help message and exit
  -a, --all             get all available flags (AD to ZW)
  -e, --every           get flags for every possible code (AA...ZZ)
  -l N, --limit N       limit to N first codes
  -m CONCURRENT, --max_req CONCURRENT
                        maximum concurrent requests (default=30)
  -s LABEL, --server LABEL
                        Server to hit; one of DELAY, ERROR, LOCAL, REMOTE
                        (default=LOCAL)
  -v, --verbose         output detailed progress info
```

All arguments are optional. The most important arguments are discussed below.

One option you can't ignore is `-s`/`--server`: it lets you choose which HTTP server and base URL will be used in the test. You can pass one of four strings to determine where the script will look for the flags (the strings are case insensitive):

`LOCAL`
: Use `http://localhost:8001/flags`; this is the default. You should configure a local HTTP server to answer at port 8001. I used Nginx for my tests. The `README.rst` file for this chapter's example code explains how to install and configure it.

`REMOTE`
: Use `http://flupy.org/data/flags`; that is a public Web site owned by me, hosted on a shared server. Please do not pound it with too many concurrent requests. The `flupy.org` domain is handled by a free account on the Cloudflare CDN so you may notice that the first downloads are slower, but they get faster when the CDN cache warms up[6].

`DELAY`
: Use `http://localhost:8002/flags`; a proxy delaying HTTP responses should be listening at port 8002. I used a Mozilla Vaurien in front of my local Nginx to introduce delays. The previously mentioned `README.rst` file has instructions for running a Vaurien proxy.

[6] Before configuring Cloudflare, I got HTTP 503 errors — Service Temporarily Unavailable — when testing the scripts with a few dozen concurrent requests on my inexpensive shared host account. Now those errors are gone.

`ERROR`

Use `http://localhost:8003/flags`; a proxy introducing HTTP errors and delaying responses should be installed at port 8003. I used a different Vaurien configuration for this.

> The `LOCAL` option only works if you configure and start a local HTTP server on port 8001. The `DELAY` and `ERROR` options require proxies listening on ports 8002 and 8003. Configuring Nginx and Mozilla Vaurien to enable these options is explained in the `17-futures/countries/README.rst` in the *Fluent Python* ´example-code´ repository on Github.

By default, each `flags2` script will fetch the flags of the 20 most populous countries from the `LOCAL` server (`http://localhost:8001/flags`) using a default number of concurrent connections which varies from script to script. Example 17-9 shows a sample run of the `flags2_sequential.py` script using all defaults.

Example 17-9. Running `flags2_sequential.py` with all defaults: `LOCAL` site, top-20 flags, 1 concurrent connection.

```
$ python3 flags2_sequential.py
LOCAL site: http://localhost:8001/flags
Searching for 20 flags: from BD to VN
1 concurrent connection will be used.
--------------------
20 flags downloaded.
Elapsed time: 0.10s
```

You can select which flags will be downloaded in several ways. Example 17-10 shows how to download all flags with country codes starting with the letters A, B a C.

Example 17-10. Run `flags2_threadpool.py` to fetch all flags with country codes prefixes A, B or C from `DELAY` server.

```
$ python3 flags2_threadpool.py -s DELAY a b c
DELAY site: http://localhost:8002/flags
Searching for 78 flags: from AA to CZ
30 concurrent connections will be used.
--------------------
43 flags downloaded.
35 not found.
Elapsed time: 1.72s
```

Regardless of how the country codes are selected, the number of flags to fetch can be limited with the `-l/--limit` option. Example 17-11 demonstrates how to run exactly 100 requests, combining the `-a` option to get all flags with `-l 100`.

Example 17-11. Run `flags2_asyncio.py` to get 100 flags (`-al 100`) from the `ERROR` server, using 100 concurrent requests (`-m 100`).

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
ERROR site: http://localhost:8003/flags
Searching for 100 flags: from AD to LK
100 concurrent connections will be used.
--------------------
73 flags downloaded.
27 errors.
Elapsed time: 0.64s
```

That's the user interface of the `flags2` examples. Let's see how they are implemented.

Error handling in the `flags2` examples

The common strategy adopted in all three examples to deal with HTTP errors is that 404 errors (Not Found) are handled by the function in charge of downloading a single file (`download_one`). Any other exception propagates to be handled by the `download_many` function.

Again, we'll start by studying the sequential code which is easier to follow — and mostly reused by the thread pool script. Example 17-12 shows the the functions that perform the actual downloads in the `flags2_sequential.py` and `flags2_threadpool.py` scripts.

Example 17-12. `flags2_sequential.py`: Basic functions in charge of downloading. Both are reused in `flags2_threadpool.py`.

```
def get_flag(base_url, cc):
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    resp = requests.get(url)
    if resp.status_code != 200:  ❶
        resp.raise_for_status()
    return resp.content


def download_one(cc, base_url, verbose=False):
    try:
        image = get_flag(base_url, cc)
    except requests.exceptions.HTTPError as exc:  ❷
        res = exc.response
        if res.status_code == 404:
            status = HTTPStatus.not_found  ❸
            msg = 'not found'
        else:  ❹
            raise
    else:
        save_flag(image, cc.lower() + '.gif')
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose:  ❺
        print(cc, msg)

    return Result(status, cc)  ❻
```

❶ `get_flag` does no error handling, it uses the `requests.Response.raise_for_status` to raise an exception for any HTTP code other than 200.

❷ `download_one` catches `requests.exceptions.HTTPError` to handle HTTP code 404 specifically…

❸ …by setting its local `status` to `HTTPStatus.not_found`; `HTTPStatus` is an `Enum` imported from `flags2_common` (Example A-10).

❹ Any other `HTTPError` exception is re-raised; other exceptions will just propagate to the caller.

❺ If the `-v/--verbose` command-line option is set, the country code and status message will be displayed; this how you'll see progress in the verbose mode.

❻ The `Result` `namedtuple` returned by `download_one` will have a `status` field with a value of `HTTPStatus.not_found` or `HTTPStatus.ok`.

Example 17-13 lists the sequential version of the `download_many` function. This code is straightforward, but its worth studying to contrast with the concurrent versions coming up. Focus on how it reports progress, handles errors and tallies downloads.

Example 17-13. `flags2_sequential.py`: The sequential implementation of `download_many`.

```
def download_many(cc_list, base_url, verbose, max_req):
    counter = collections.Counter()  ❶
    cc_iter = sorted(cc_list)  ❷
    if not verbose:
        cc_iter = tqdm.tqdm(cc_iter)  ❸
    for cc in cc_iter:  ❹
        try:
            res = download_one(cc, base_url, verbose)  ❺
        except requests.exceptions.HTTPError as exc:  ❻
            error_msg = 'HTTP error {res.status_code} - {res.reason}'
            error_msg = error_msg.format(res=exc.response)
        except requests.exceptions.ConnectionError as exc:  ❼
            error_msg = 'Connection error'
        else:  ❽
            error_msg = ''
            status = res.status

        if error_msg:
            status = HTTPStatus.error  ❾
        counter[status] += 1  ❿
        if verbose and error_msg:  ⓫
            print('*** Error for {}: {}'.format(cc, error_msg))

    return counter  ⓬
```

❶ This `Counter` will tally the different download outcomes: `HTTPStatus.ok`, `HTTPStatus.not_found` or `HTTPStatus.error`.

❷ `cc_iter` holds the list of the country codes received as arguments, ordered alphabetically.

❸ If not running in verbose mode, `cc_iter` is passed to the `tqdm` function which will return an iterator that yields the items in `cc_iter` while also displaying the animated progress bar.

❹ This `for` loop iterates over `cc_iter` and…

❺ …performs the download by successive calls to `download_one`.

❻ HTTP-related exceptions raised by `get_flag` and not handled by `download_one` are handled here.

❼ Other network-related exceptions are handled here. Any other exception will abort the script, because the `flags2_common.main` function which calls `download_many` has no `try/except`.

❽ If no exception escaped `download_one` then the the `status` is retrieved from the `HTTPStatus` `namedtuple` returned by `download_one`.

❾ If there was an error, set the local `status` accordingly.

❿ Increment the counter by using the value of the `HTTPStatus` `Enum` as key.

⓫ If running in verbose mode, display the error message for the current country code, if any.

⓬ Return the `counter` so that the `main` function can display the numbers in its final report.

We'll now study the refactored thread pool example, `flags2_threadpool.py`.

Using `futures.as_completed`

In order to integrate the TQDM progress bar and handle errors on each request, the `flags2_threadpool.py` script uses `futures.ThreadPoolExecutor` with the `futures.as_completed` function we've already seen. Example 17-14 is the full listing of `flags2_threadpool.py`. Only the `download_many` function is implemented, the other functions are reused from the `flags2_common` and `flags2_sequential` modules.

Example 17-14. `flags2_threadpool.py`: full listing.

```
import collections
from concurrent import futures

import requests
import tqdm  ❶

from flags2_common import main, HTTPStatus  ❷
from flags2_sequential import download_one  ❸

DEFAULT_CONCUR_REQ = 30  ❹
MAX_CONCUR_REQ = 1000  ❺


def download_many(cc_list, base_url, verbose, concur_req):
    counter = collections.Counter()
    with futures.ThreadPoolExecutor(max_workers=concur_req) as executor:  ❻
        to_do_map = {}  ❼
        for cc in sorted(cc_list):  ❽
            future = executor.submit(download_one,
                            cc, base_url, verbose)  ❾
            to_do_map[future] = cc  ❿
        done_iter = futures.as_completed(to_do_map)  ⓫
        if not verbose:
            done_iter = tqdm.tqdm(done_iter, total=len(cc_list))  ⓬
        for future in done_iter:  ⓭
            try:
                res = future.result()  ⓮
            except requests.exceptions.HTTPError as exc:  ⓯
                error_msg = 'HTTP {res.status_code} - {res.reason}'
                error_msg = error_msg.format(res=exc.response)
            except requests.exceptions.ConnectionError as exc:
                error_msg = 'Connection error'
            else:
                error_msg = ''
                status = res.status

            if error_msg:
                status = HTTPStatus.error
            counter[status] += 1
            if verbose and error_msg:
                cc = to_do_map[future]  ⓰
                print('*** Error for {}: {}'.format(cc, error_msg))

    return counter


if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

❶ Import the progress-bar display library.

❷ Import one function and one Enum from the flags2_common module.

❸ Reuse the donwload_one from flags2_sequential (Example 17-12).

❹ If the -m/--max_req command-line option is not given, this will be the maximum number of concurrent requests, implemented as the size of the thread pool; the actual number may be smaller, if the number of flags to download is smaller.

❺ MAX_CONCUR_REQ caps the maximum number of concurrent requests regardless of the number of flags to download or the -m/--max_req command-line option; it's a safety precaution.

❻ Create the executor with max_workers set to concur_req, computed by the main function as the smaller of: MAX_CONCUR_REQ, the length of cc_list and the value of the -m/--max_req command-line option. This avoids creating more threads than necessary.

❼ This dict will map each Future instance — representing one download — with the respective country code for error reporting.

❽ Iterate over the list of country codes in alphabetical order. The order of the results will depend in the timing of the HTTP responses more than anything, but if the size of the thread pool (given by concur_req) is much smaller than len(cc_list) you may notice the downloads batched alphabetically.

❾ Each call to executor.submit schedules the execution of one callable and returns a Future instance. The first argument is the callable, the rest are the arguments it will receive.

❿ Store the future and the country code in the dict.

⓫ futures.as_completed returns an iterator that yields futures as they are done.

⓬ If not in verbose mode, wrap the result of as_completed with the tqdm function to display the progress bar; since done_iter has no len, we must tell tqdm what is the expected number of items as the total= argument, so tqdm can estimate the work remaining.

⓭ Iterate over the futures as they are completed.

⓮ Calling the result method on a future either returns the value returned by the callable, or raises whatever exceptions was caught when the callable was executed. This method may block waiting for a resolution, but not in this example because as_completed only returns futures that are done.

⓯ Handle the potential exceptions; the rest of this function is identical to the the sequential version of download_many (Example 17-13), except for the next callout.

⓰ To provide context for the error message, retrieve the country code from the to_do_map using the current future as key. This was not necessary in the sequential version because we were iterating over the list of country codes, so we had the current cc; here we are iterating over the futures.

Example 17-14 uses an idiom that's very useful with futures.as_completed: building a dict to map each future to other data that may be useful when the future is completed. Here the to_do_map maps each future to the country code assigned to it. This makes it easy to do follow-up processing with the result of the futures, despite the fact that they are produced out of order.

Python threads are well suited for I/O intensive applications, and the concurrent.futures package makes them trivially simple to use for certain use cases. This concludes our basic introduction to concurrent.futures. Let's now discuss alternatives for when ThreadPoolExecutor or ProcessPoolExecutor are not suitable.

Threading and multiprocessing alternatives.

Python has supported threads since its release 0.9.8 (1993); concurrent.futures is just the latest way of using them. In Python 3 the original thread module was deprecated in favor of the higher-level threading module[7]. If futures.ThreadPoolExecutor is not flexible enough for a certain job, you may need to build your own solution out of basic threading components such as Thread, Lock, Semaphore etc. — possibly using the thread-safe queues of the queue module for passing data between threads. Those moving parts are encapsulated by futures.ThreadPoolExecutor.

For CPU-bound work you need to sidestep the GIL by launching multiple processes. The futures.ProcessPoolExecutor is the easiest way to do it. But again, if your use case is complex, you'll need more advanced tools. The multiprocessing package emulates the threading API but delegates jobs to multiple processes. For simple programs, multiprocessing can replace threading with few changes. But multiprocessing also offers facilities to solve the biggest challenge faced by collaborating processes: how to pass around data.

Chapter Summary

We started the chapter by comparing two concurrent HTTP clients with a sequential one, demonstrating a significant performance increase over the sequential script.

[7] The threading module has been available since Python 1.5.1 (1998), yet some insist on using the old thread module. In Python 3 it was renamed to _thread to highlight the fact that it's just a low-level implementation detail, and shouldn't be used in application code.

After studying the first example based on concurrent.futures we took a closer look at future objects, either instances of concurrent.futures.Future or asyncio.Future, emphasizing what these classes have in common (their differences will be emphasized in Chapter 18). We saw how to create futures by calling Executor.submit(…), and iterate over completed futures with concurrent.futures.as_completed(…).

Next we saw why Python threads are well suited for I/O bound applications, despite the GIL: every standard library I/O function written in C releases the GIL, so while a given thread is waiting for I/O, the Python scheduler can switch to another thread. We then discussed the use of multiple processes with the concurrent.futures.ProcessPoolExecutor class, to go around the GIL and use multiple CPU-cores to run cryptographic algorithms, achieving speedups of more than 100% when using 4 workers.

In the following section we took a close look at how the a concurrent.futures.ThreadPoolExcutor works, with a didactic example launching tasks that did nothing for a few seconds, except displaying their status with a timestamp.

Next we went back to the flag downloading examples. Enhancing them with a progress bar and proper error handling prompted further exploration of the future.as_completed generator function showing a common pattern: storing futures in a dict to link further information to them when submitting, so that we can use that information when the future comes out of the as_completed iterator.

We concluded the coverage of concurrency with threads and processes with a brief reminder of the lower-level, but more flexible threading and multiprocessing modules which represent the traditional way of leveraging threads and processes in Python.

Further reading

The concurrent.futures package was contributed by Brian Quinlan, who presented it in a great talk titled The future is soon! at PyCon Australia 2010. Quinlan's talk has no slides, he shows what the library does by typing code directly in the Python console. As a motivating example, the presentation features a short video with XKCD cartoonist/programmer Randall Munroe making an unintended DOS attack on Google Maps to build a colored map of driving times around his city. The formal introduction to the library is PEP 3148 - futures - execute computations asynchronously. In the PEP, Quinlan wrote that the concurrent.futures library was "heavily influenced by the the Java java.util.concurrent package".

Parallel Programming with Python (Packt, 2014), by Jan Palach, covers several tools for concurrent programming, including the concurrent.futures, threading, and multiprocessing modules. It goes beyond the standard library to discuss Celery, a task queue used to distribute work across threads and processes, even on different machines. In the Django community, Celery is probably the most widely used system to offload heavy tasks such as PDF generation to other processes, thus avoiding delays in producing an HTTP response.

In the Beazley & Jones *Python Cookbook*, 3rd. edition (O'Reilly, 2013) there are recipes using `concurrent.futures` starting with 11.12. *Understanding Event-Driven I/O*. Recipe 12.7. *Creating a Thread Pool* shows a simple TCP echo server, and 12.8. *Performing Simple Parallel Programming* offers a very practical example: analyzing a whole directory of `gzip` compressed Apache log files with the help of a `ProcessPoolExecutor`. For more about threads, the entire chapter 12 of Beazley & Jones is great, with special mention to 12.10. *Defining an Actor Task* which demonstrates the Actor model: a proven way of coordinating threads through message passing.

Brett Slatkin's Effective Python (Addison-Wesley, 2015) has a multi-topic chapter about concurrency, including coverage of coroutines, `concurrent.futures` with threads and processes, and the use of locks and queues for thread programming without the `ThreadPoolExecutor`.

High Performance Python, Micha Gorelick and Ian Ozsvald (O'Reilly 2014) and *The Python Standard Library by Example* (Addison-Wesley, 2011), by Doug Hellmann, also cover threads and processes.

For a modern take on concurrency without threads or callbacks, *Seven Concurrency Models in Seven Weeks*, by Paul Butcher (Pragmatic Bookshelf, 2014) is an excellent read. I love its subtitle: "When Threads Unravel". In that book, threads and locks are covered in chapter 1, and the remaining six chapters are devoted to modern alternatives to concurrent programming, as supported by different languages. Python, Ruby and JavaScript are not among them.

If you are intrigued about the GIL, start with the *Python Library and Extension FAQ*: Can't we get rid of the Global Interpreter Lock?. Also worth reading are posts by Guido van Rossum and Jesse Noller (contributor of the `multiprocessing` package): It isn't Easy to Remove the GIL and Python Threads and the Global Interpreter Lock. Finally, David Beazley has a detailed exploration on the inner workings of the GIL: Understanding the Python GIL[8]. In slide #54 of the presentation, Beazley reports some alarming results, including a 20x increase in processing time for a particular benchmark with the new GIL algorithm introduced in Python 3.2. However, Beazley apparently used an empty `while True: pass` to simulate CPU-bound work, and that is not realistic. The issue is not significant with real workloads, according to a comment by Antoine Pitrou — who implemented the new GIL algorithm — in the bug report submitted by Beazley.

[8] Thanks to Lucas Brunialti for sending me a link to this talk.

While the GIL is real problem and is not likely to go away soon, Jesse Noller and Richard Oudkerk contributed a library to make it easier to work around it in CPU-bound applications: the `multiprocessing` package, which emulates the `threading` API across processes, along with supporting infrastructure of locks, queues, pipes, shared memory etc. The package was introduced in PEP 371 — Addition of the multiprocessing package to the standard library. The official documentation for the package is a 93KB `.rst` file — that's about 63 pages — making it one of the longest chapters in the Python standard library. Multiprocessing is the basis for the `concurrent.futures.ProcessPoolExecutor`.

For CPU and data intensive parallel processing, a new option with a lot of momentum in the big data community is the Apache Spark distributed computing engine, offering a friendly Python API and support for Python objects as data, as shown in their examples page.

Two elegant and super easy libraries for parallelizing tasks over processes are `lelo` by João S. O. Bueno and `python-parallelize` by Nat Pryce. The `lelo` package defines a `@parallel` decorator that you can apply to any function to magically make it unblocking: when you call the decorated function, its execution is started in another process. Nat Pryce's `python-parallelize` package provides a `parallelize` generator that you can use to distribute the execution of a `for` loop over multiple CPUs. Both packages use the `multiprocessing` module under the covers.

Soapbox

Thread avoidance

> Concurrency: one of the most difficult topics in computer science (usually best avoided)[9]
>
> — David Beazley
> *Python coach and mad scientist*

I agree with the apparently contradictory quotes by David Beazley, above, and Michele Simionato at the start of this chapter. After attending a concurrency course at the university — in which "concurrent programming" was equated to managing threads and locks — I came to the conclusion that I don't want to manage threads and locks myself, any more than I want to manage memory allocation and deallocation. Those jobs are best carried out by the systems programmers who have the know-how, the inclination and the time to get them right — hopefully.

[9] Slide #9 from A Curious Course on Coroutines and Concurrency, tutorial presented at PyCon 2009.

That's why I think the `concurrent.futures` package is exciting: it treats threads, processes and queues as infrastructure at your service, not something you have to deal with directly. Of course, it's designed with simple jobs in mind, the so-called "embarrassingly parallel" problems. But that's a large slice of the concurrency problems we face when writing applications — as opposed to operating systems or database servers, as Simionato points out in that quote.

For "non-embarrassing" concurrency problems, threads and locks are not the answer either. Threads will never disappear at the OS level, but every programming language I've found exciting in the last several years provides better, higher-level, concurrency abstractions, as the *Seven Concurrency Models* book demonstrates. Go, Elixir and Clojure are among them. Erlang — the implementation language of Elixir — is a prime example of a language designed from the ground up with concurrency in mind. It doesn't excite me for a simple reason: I find its syntax ugly. Python spoiled me that way.

José Valim, well-known as a Ruby on Rails core contributor, designed Elixir with a pleasant, modern syntax. Like Lisp and Clojure, Elixir implements syntactic macros. That's a double-edged sword. Syntactic macros enable powerful DSLs, but the proliferation of sub-languages can lead to incompatible codebases and community fragmentation. Lisp drowned in a flood of macros, with each Lisp shop using their own arcane dialect. Standardizing around Common Lisp resulted in a bloated language. I hope José Valim can inspire the Elixir community to avoid a similar outcome.

Like Elixir, Go is a modern language with fresh ideas. But, in some regards, it's a conservative language, compared to Elixir. Go doesn't have macros, and its syntax is simpler than Python's. Go doesn't support inheritance or operator overloading, and it offers less opportunities for metaprogramming than Python. These limitations are considered features. They lead to more predictable behavior and performance. That's a big plus in the highly-concurrent, mission-critical settings where Go aims to replace C++, Java and Python.

While Elixir and Go are direct competitors in the high concurrency space, their design philosophies appeal to different crowds. Both are likely to thrive. But in the history of programming languages, the conservative ones tend to attract more coders. I'd like to become fluent in Go and Elixir.

About the GIL

The GIL simplifies the implementation of the CPython interpreter and of extensions written in C, so we can thank the GIL for the vast number of extensions in C available for Python — and that is certainly one of the key reasons why Python is so popular today.

For many years I was under the impression that the GIL made Python threads nearly useless beyond toy applications. Only when I discovered that **every** blocking I/O call in the standard library releases the GIL I realized Python threads are excellent for I/O bound systems — the kind of applications customers usually pay me to develop, given my professional experience.

Concurrency in the competition

MRI — the reference implementation of Ruby — also has a GIL, so its threads are under the same limitations as Python's. Meanwhile, JavaScript interpreters don't support user-level threads at all; asynchronous programming with callbacks is their only path to concurrency. I mention this because Ruby and JavaScript are the closest direct competitors to Python as general purpose, dynamic programming languages.

Looking at the concurrency-savvy new crop of languages, Go and Elixir are probably the ones best positioned to eat Python's lunch. But now we have `asyncio`. If hordes of people believe Node.js with raw callbacks is a viable platform for concurrent programming, how hard can it be to win them over to Python when the `asyncio` ecosystem matures? But that's a topic for the next **Soapbox**.

CHAPTER 18

Concurrency with `asyncio`

> Concurrency is about dealing with lots of things at once.
>
> Parallelism is about doing lots of things at once.
>
> Not the same, but related.
>
> One is about structure, one is about execution.
>
> Concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable[1].
>
> — Rob Pike
> *co-inventor of the Go language*

Professor Imre Simon[2] liked to say there are two major sins in science: using different words to mean the same thing and using one word to mean different things. If you do any research on concurrent or parallel programming you will find different definitions for "concurrency" and "parallelism". I will adopt the informal definitions by Rob Pike, quoted above.

For real parallelism you must have multiple cores. A modern laptop has 4 CPU cores but is routinely running more than 100 processes at any given time under normal, casual use. So, in practice, most processing happens concurrently and not in parallel. The computer is constantly dealing with 100+ processes, making sure each has an opportunity to make progress, even if the CPU itself can't do more than 4 things at once. Ten years ago we used machines which were also able to handle 100 processes concurrently, but on a single core. That's why Rob Pike titled that talk "Concurrency is not Parallelism (it's better)."

[1] Slide #5 of the talk Concurrency is not Parallelism (it's better).

[2] Imre Simon (1943-2009), was a pioneer of Computer Science in Brazil who made seminal contributions to Automata Theory and started the field of Tropical Mathematics. He was also an advocate of free software and free culture. I was fortunate to study, work and hang out with him.

This chapter introduces asyncio, a package that implements concurrency with coroutines driven by an event loop. It's one of the largest and most ambitious libraries ever added to Python. Guido van Rossum developed asyncio outside of the Python repository and gave the project a code name of "Tulip" — so you'll see references to that flower when researching this topic online. For example, the main discussion group is still called python-tulip.

Tulip was renamed to asyncio when it was added to the standard library in Python 3.4. It's also compatible with Python 3.3 — you can find it on PyPI under the new official name. Because it uses yield from expressions extensively, asyncio is incompatible with older versions of Python.

> The Trollius project — also named after a flower — is a backport of ayncio to Python 2.6 and newer, replacing yield from with yield and clever callables named From and Return. A yield from … expression becomes yield From(…); and when a coroutine needs to return a result you write raise Return(result) instead of return result. Trollius is led by Victor Stinner, who is also an asyncio core developer, and who kindly agreed to review this chapter as this book was going into production.

In this chapter we'll see:

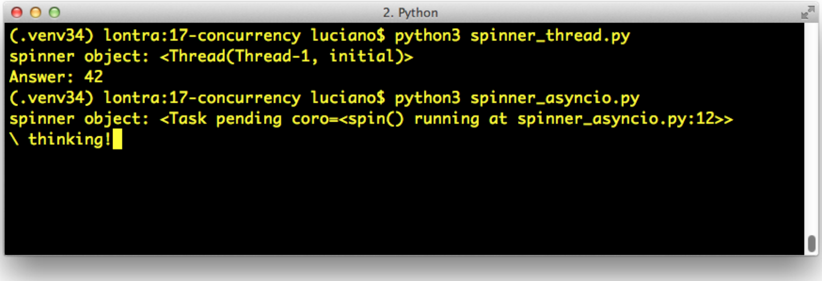
- A comparison between a simple threaded program and the asyncio equivalent, showing the relationship between threads and asynchronous tasks.
- How the asyncio.Future class differs from concurrent.futures.Future.
- Asynchronous versions of the flag download examples from Chapter 17.
- How asynchronous programming manages high concurrency in network applications, without using threads or processes.
- How coroutines are a major improvement over callbacks for asynchronous programming.
- How to avoid blocking the event loop by offloading blocking operations to a thread pool.
- Writing asyncio servers, and how to rethink Web applications for high concurrency.
- Why asyncio is poised to have a big impact in the Python ecosystem.

Let's get started with the simple example contrasting threading and asyncio.

Thread versus coroutine: a comparison

During a discussion about threads and the GIL, Michele Simionato posted a simple but fun example using `multiprocessing` to display an animated spinner made with the ASCII characters `"|/-\"` on the console while some long computation is running.

I adapted Simionato's example to use a thread with the `Threading` module and then a coroutine with `asyncio`, so you can see the two examples side-by-side and understand how to code concurrent behavior without threads.



```
(.venv34) lontra:17-concurrency luciano$ python3 spinner_thread.py
spinner object: <Thread(Thread-1, initial)>
Answer: 42
(.venv34) lontra:17-concurrency luciano$ python3 spinner_asyncio.py
spinner object: <Task pending coro=<spin() running at spinner_asyncio.py:12>>
\ thinking!
```

Figure 18-1. The scripts `spinner_thread.py` and `spinner_asyncio.py` produce similar output: the `repr` of a `spinner` object and the text `Answer: 42`. In the screen shot, `spinner_asyncio.py` is still running, and the spinner message `\ thinking!` is shown; when the script ends, that line will be replaced by the `Answer: 42`.

The output of Example 18-1 and Example 18-2 is animated, so you really should run the scripts to see what happens. If you're on a train, take a look at Figure 18-1 and imagine the \ bar before the word "thinking" is spinning.

Let's go over the `spinner_thread.py` script first (Example 18-1);

Example 18-1. `spinner_thread.py`: Animating a text spinner with a thread.

```
import threading
import itertools
import time
import sys


class Signal:  ❶
    go = True


def spin(msg, signal):  ❷
```

```
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):  ❸
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))  ❹
        time.sleep(.1)
        if not signal.go:  ❺
            break
    write(' ' * len(status) + '\x08' * len(status))  ❻


def slow_function():  ❼
    # pretend waiting a long time for I/O
    time.sleep(3)  ❽
    return 42


def supervisor():  ❾
    signal = Signal()
    spinner = threading.Thread(target=spin,
                               args=('thinking!', signal))
    print('spinner object:', spinner)  ❿
    spinner.start()  ⓫
    result = slow_function()  ⓬
    signal.go = False  ⓭
    spinner.join()  ⓮
    return result


def main():
    result = supervisor()  ⓯
    print('Answer:', result)


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

❶ This class defines a simple mutable object with a go attribute we'll use to control the thread from outside.

❷ This function will run in a separate thread. The `signal` argument is an instance of the `Signal` class just defined.

❸ This is actually an infinite loop because `itertools.cycle` produces items cycling from the given sequence forever.

❹ The trick to do text-mode animation: move the cursor back with backspace characters (`\x08`).

❺ If the go attribute is no longer `True`, exit the loop.

❻ Clear the status line by overwriting with spaces and moving the cursor back to the beginning.

❼ Imagine this is some costly computation.

❽ Calling `sleep` will block the main thread but, crucially, the GIL will be released so the secondary thread will proceed.

❾ This function sets up the secondary thread, displays the thread object, runs the slow computation and kills the thread.

❿ Display the secondary thread object. The output looks like `<Thread(Thread-1, initial)>`.

⓫ Start the secondary thread.

⓬ Run `slow_function`; this blocks the main thread. Meanwhile, the spinner is animated by the secondary thread.

⓭ Change the state of the `signal`; this will terminate the `for` loop inside the `spin` function.

⓮ Wait until the `spinner` thread finishes.

⓯ Run the `supervisor` function.

Note that, by design, there is no API for terminating a thread in Python. You must send it a message to shutdown. Here I used the `signal.go` attribute: when the main thread sets it to false, the `spinner` thread will eventually notice and exit cleanly.

Now let's see how the same behavior can be achieved with an `@asyncio.coroutine` instead of a thread.

As noted in the "Chapter summary" on page 500 (Chapter 16), `asyncio` uses a stricter definition of "coroutine". A coroutine suitable for use with the `asyncio` API must use `yield from` and not `yield` in its body. Also, an `asyncio` coroutine should be driven by a caller invoking it through `yield from` or by passing the coroutine to one of the `asyncio` functions such as `asyncio.async(…)` and others covered in this chapter. Finally, the `@asyncio.coroutine` decorator should be applied to coroutines, as shown in the examples.

Take a look at Example 18-2.

Example 18-2. `spinner_asyncio.py`: Animating a text spinner with a coroutine.

```
import asyncio
import itertools
import sys
```

```
@asyncio.coroutine  ❶
def spin(msg):  ❷
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))
        try:
            yield from asyncio.sleep(.1)  ❸
        except asyncio.CancelledError:  ❹
            break
    write(' ' * len(status) + '\x08' * len(status))


@asyncio.coroutine
def slow_function():  ❺
    # pretend waiting a long time for I/O
    yield from asyncio.sleep(3)  ❻
    return 42


@asyncio.coroutine
def supervisor():  ❼
    spinner = asyncio.async(spin('thinking!'))  ❽
    print('spinner object:', spinner)  ❾
    result = yield from slow_function()  ❿
    spinner.cancel()  ⓫
    return result


def main():
    loop = asyncio.get_event_loop()  ⓬
    result = loop.run_until_complete(supervisor())  ⓭
    loop.close()
    print('Answer:', result)


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

❶ Coroutines intended for use with `asyncio` should be decorated with `@asyncio.coroutine`. This not mandatory, but is highly advisable. See explanation following this listing.

❷ Here we don't need the `signal` argument which was used to shut down the thread in the `spin` function of Example 18-1.

❸ Use `yield from asyncio.sleep(.1)` instead of just `time.sleep(.1)`, to sleep without blocking the event loop.

❹ If `asyncio.CancelledError` is raised after `spin` wakes up, it's because cancellation was requested, so exit the loop.

❺ `slow_function` is now a coroutine, and uses `yield from` to let the event loop proceed while this coroutine pretends to do I/O by sleeping.

❻ The `yield from asyncio.sleep(3)` expression handles the control flow to the main loop, which will resume this coroutine after the sleep delay.

❼ `supervisor` is now a coroutine as well, so it can drive `slow_function` with `yield from`.

❽ `asyncio.async(…)` schedules the `spin` coroutine to run, wrapping it in a `Task` object, which is returned immediately

❾ Display the `Task` object. The output looks like `<Task pending coro=<spin() running at spinner_asyncio.py:12>>`.

❿ Drive the `slow_function()`. When that is done, get the returned value. Meanwhile, the event loop will continue running because `slow_function` ultimately uses `yield from asyncio.sleep(3)` to hand control back to the main loop.

⓫ A `Task` object can be cancelled; this raises `asyncio.CancelledError` at the `yield` line where the coroutine is currently suspended. The coroutine may catch the exception and delay or even refuse to cancel.

⓬ Get a reference to the event loop.

⓭ Drive the `supervisor` coroutine to completion; the return value of the coroutine is the return value of this call.

> Never use `time.sleep(…)` in `asyncio` coroutines unless you want to block the main thread, therefore freezing the event loop and probably the whole application as well. If a coroutine needs to spend some time doing nothing, it should `yield from asyncio.sleep(DELAY)`.

The use of the `@asyncio.coroutine` decorator is not mandatory, but highly recommended: it makes the coroutines stand out among regular functions, and helps with debugging by issuing a warning when a coroutine is garbage collected without being yielded from — which means some operation was left unfinished and is likely a bug. This is not a *priming decorator*.

Note that the line count of `spinner_thread.py` and `spinner_asyncio.py` is nearly the same. The `supervisor` functions are the heart of these examples. Let's compare them in detail. Example 18-3 lists only the `supervisor` from the `Threading` example:

Example 18-3. `spinner_thread.py`: The threaded `supervisor` function.

```
def supervisor():
    signal = Signal()
    spinner = threading.Thread(target=spin,
                               args=('thinking!', signal))
    print('spinner object:', spinner)
    spinner.start()
    result = slow_function()
    signal.go = False
    spinner.join()
    return result
```

For comparison, Example 18-4 shows the `supervisor` coroutine:

Example 18-4. `spinner_asyncio.py`: The asynchronous `supervisor` coroutine.

```
@asyncio.coroutine
def supervisor():
    spinner = asyncio.async(spin('thinking!'))
    print('spinner object:', spinner)
    result = yield from slow_function()
    spinner.cancel()
    return result
```

Here is a summary of the main differences to note between the two `supervisor` implementations:

- An `asyncio.Task` is roughly the equivalent of a `threading.Thread`footnote:[Victor Stinner -- special technical reviewer for this chapter -- points out that a `Task is like a green thread in libraries that implement cooperative multi-tasking, such as `gevent`.].
- A `Task` drives a coroutine, and a `Thread` invokes a callable.
- You don't instantiate `Task` objects yourself, you get them by passing a coroutine to `asyncio.async(…)` or `loop.create_task(…)`.
- When you get a `Task` object, it is already scheduled to run (e.g. by `asyncio.async`); a `Thread` instance must be explicitly told to run by calling its `start` method.
- In the threaded `supervisor`, the `slow_function` is a plain function and is directly invoked by the thread. In the `asyncio` `supervisor`, `slow_function` is a coroutine driven by `yield from`.
- There's no API to terminate a thread from the outside, because a thread could be interrupted at any point, leaving the system in an invalid state. For tasks, there is the `Task.cancel()` instance method which raises `CancelledError` inside the co-

routine. The coroutine can deal with this by catching the exception in the `yield` where it's suspended.

- The `supervisor` coroutine must be executed with `loop.run_until_complete` in the `main` function.

This comparison should help you understand how concurrent jobs are orchestrated with `asyncio`, in contrast to how it's done with the more familiar `Threading` module.

One final point related to threads versus coroutines: if you've done any non-trivial programming with threads, you know how challenging it is to reason about the program because the scheduler can interrupt a thread at any time. You must remember to hold locks to protect the critical sections of your program, to avoid getting interrupted in the middle of a multi-step operation — which could leave data in an invalid state.

With coroutines, everything is protected against interruption by default. You must explicitly yield to let the rest of the program run. Instead of holding locks to synchronize the operations of multiple threads, you have coroutines that are "synchronized" by definition: only one of them is running at any time. And when you want to give up control, you use `yield` or `yield from` to give control back to the scheduler. That's why it is possible to safely cancel a coroutine: by definition, a coroutine can only be cancelled when it's suspended at a `yield` point, so you can perform cleanup by handling the `CancelledError` exception.

We'll no see how the `asyncio.Future` class differs from the `concurrent.futures.Future` class we saw in Chapter 17.

`asyncio.Future`: non-blocking by design

The `asyncio.Future` and the `concurrent.futures.Future` classes have mostly the same interface, but are implemented differently and are not interchangeable. PEP-3156 — Asynchronous IO Support Rebooted: the "asyncio" Module has this to say about this unfortunate situation:

> In the future (pun intended) we may unify `asyncio.Future` and `concurrent.futures.Future`, e.g. by adding an `__iter__` method to the latter that works with `yield from`.

As mentioned in "Where are the futures?" on page 513, futures are created only as the result of scheduling something for execution. In `asyncio`, `BaseEventLoop.create_task(…)` takes a coroutine, schedules it to run, and returns an `asyncio.Task` instance — which is also an instance of `asyncio.Future` because `Task` is a subclass of `Future` designed to wrap a coroutine. This is analogous to how we create `concurrent.futures.Future` instances by invoking `Executor.submit(…)`.

Like its `concurrent.futures.Future` counterpart, the `asyncio.Future` class provides `.done()`, `.add_done_callback(…)` and `.results()` methods, among others. The first two methods work as described in "Where are the futures?" on page 513, but `.result()` is very different.

In `asyncio.Future` the `.result()` method takes no arguments, so you can't specify a timeout. Also, if you call `.result()` and the future is not done, it does not block waiting for the result. Instead, an `asyncio.InvalidStateError` is raised.

However, the usual way to get the result of an `asyncio.Future` is to `yield from` it, as we'll see in Example 18-8.

Using `yield from` with a future automatically takes care of waiting for it to finish, without blocking the event loop — because in `asyncio`, `yield from` is used to give control back to the event loop.

Note that using `yield from` with a future is the coroutine equivalent of the functionality offered by `add_done_callback`: instead of triggering a callback, when the delayed operation is done the event loop sets the result of the future, and the `yield from` expression produces a return value inside our suspended coroutine, allowing it to resume.

In summary, because `asyncio.Future` is designed to work with `yield from`, these methods are often not needed:

- You don't need `my_future.add_done_callback(…)` because you can simply put whatever processing you would do after the future is done in the lines that follow `yield from my_future` in your coroutine. That's the big advantage of having coroutines: functions that can be suspended and resumed.
- You don't need `my_future.result()` because the the value of `yield from` expression on a future is the result, e.g. `result = yield from my_future`.

Of course, there are situations in which `.done()`, `.add_done_callback(…)` and `.results()` are useful. But in normal usage, `asyncio` futures are driven by `yield from`, not by calling those methods.

We'll now consider how `yield from` and the `asyncio` API brings together futures, tasks and coroutines.

Yielding from futures, tasks and coroutines

In `asyncio` there is a close relationship between futures and coroutines because you can get the result of an `asyncio.Future` by yielding from it. This means that `res = yield from foo()` works if `foo` is a coroutine function (therefore it returns a coroutine object when called) or if `foo` is a plain function that returns a `Future` or `Task` instance. This is one of the reasons why coroutines and futures are interchangeable in many parts of the asyncio API.

In order to execute, a coroutine must be scheduled, and then it's wrapped in an asyncio.Task. Given a coroutine, there are two main ways of obtaining a Task:

asyncio.async(coro_or_future, *, loop=None)
: This function unifies coroutines and futures: the first argument can be either one. If it's a Future or Task, it's returned unchanged. If it's a coroutine, async calls loop.create_task(…) on it to create a Task. An optional event loop may be passed as the loop= keyword argument; if omitted, async gets the loop object by calling asyncio.get_event_loop().

BaseEventLoop.create_task(coro)
: This method schedules the coroutine for execution and returns an asyncio.Task object. If called on a custom subclass of BaseEventLoop, the object returned may be an instance of some other Task-compatible class provided by an external library, e.g. Tornado.

BaseEventLoop.create_task(…) is only available in Python 3.4.2 or later. If you're using an older version of Python 3.3 or 3.4, you need to use asyncio.async(…), or install a more recent version of asyncio from PyPI.

Several asyncio functions accept coroutines and wrap them in asyncio.Task objects automatically, using asyncio.async internally. One example is BaseEventLoop.run_until_complete(…).

If you want to experiment with futures and coroutines on the Python console or in small tests, you can use the following snippet[3]:

```
>>> import asyncio
>>> def run_sync(coro_or_future):
...     loop = asyncio.get_event_loop()
...     return loop.run_until_complete(coro_or_future)
...
>>> a = run_sync(some_coroutine())
```

The relationship between coroutines, futures and tasks is documented in section 18.5.3. Tasks and coroutines of the asyncio documentation, where you'll find this note:

> In this documentation, some methods are documented as coroutines, even if they are plain Python functions returning a Future. This is intentional to have a freedom of tweaking the implementation of these functions in the future.

[3] Suggested by Petr Viktorin in a Sep. 11, 2014 message to the python-ideas list.

Having covered these fundamentals, we'll now study the code for the asynchronous flag download script `flags_asyncio.py` demonstrated along with the sequential and tread pool scripts in Example 17-1 (Chapter 17).

Downloading with `asyncio` and `aiohttp`

As of Python 3.4, `asyncio` only supports TCP and UDP directly. For HTTP or any other protocol, we need third-party packages; `aiohttp` is the one everyone seems to be using for `asyncio` HTTP clients and servers at this time.

Example 18-5 is the full listing for the flag downloading script `flags_asyncio.py`. Here is a high level view of how it works:

1. We start the process in `download_many` by feeding the event loop with several coroutine objects produced by calling `download_one`.
2. The `asyncio` event loop activates each coroutine in turn.
3. When a client coroutine such as `get_flag` uses `yield from` to delegate to a library coroutine — such as `aiohttp.request` — control goes back to the event loop, which can execute another previously scheduled coroutine.
4. The event loop uses low-level APIs based on callbacks to get notified when a blocking operation is completed.
5. When that happens, the main loop sends a result to the suspended coroutine.
6. The coroutine then advances to the next `yield`, for example, `yield from resp.read()` in `get_flag`. The event loop takes charge again. Steps 4, 5, 6 repeat until the event loop is terminated.

This is similar to the taxi simulation in "The taxi fleet simulation" on page 492, where a main loop started several taxi processes in turn. As each taxi process yielded, the main loop scheduled the next event for that taxi (to happen in the future), and proceeded to activate the next taxi in the queue. The taxi simulation is much simpler, and you can easily understand it's main loop. But the general flow is the same as in `asyncio`: a single-threaded program where a main loop activates queued coroutines one by one. Each coroutine advances a few steps, then yields control back to the main loop, which then activates the next coroutine in the queue.

Now let's review Example 18-5 play-by-play.

Example 18-5. `flags_asyncio.py`: Asynchronous download script with `asyncio` and `aiohttp`.

```
import asyncio

import aiohttp   ❶
```

```
from flags import BASE_URL, save_flag, show, main  ❷


@asyncio.coroutine  ❸
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)  ❹
    image = yield from resp.read()  ❺
    return image


@asyncio.coroutine
def download_one(cc):  ❻
    image = yield from get_flag(cc)  ❼
    show(cc)
    save_flag(image, cc.lower() + '.gif')
    return cc


def download_many(cc_list):
    loop = asyncio.get_event_loop()  ❽
    to_do = [download_one(cc) for cc in sorted(cc_list)]  ❾
    wait_coro = asyncio.wait(to_do)  ❿
    res, _ = loop.run_until_complete(wait_coro)  ⓫
    loop.close()  ⓬

    return len(res)


if __name__ == '__main__':
    main(download_many)
```

❶ `aiohttp` must be installed — it's not in the standard library.

❷ Reuse some functions from the `flags` module (Example 17-2).

❸ Coroutines should be decorated with `@asyncio.coroutine`.

❹ Blocking operations are implemented as coroutines, and your code delegates to them via `yield from` so they run asynchronously.

❺ Reading the response contents is a separate asynchronous operation.

❻ `download_one` must also be a coroutine, since it uses `yield from`.

❼ The only difference from the sequential implementation of `download_one` are the words `yield from` in this line; the rest of the function body is exactly as before.

❽ Get a reference to the underlying event-loop implementation.

❾ Build a list of generator objects by calling the `download_one` function once for each flag to be retrieved.

➓ Despite its name, `wait` is not a blocking function. It's a coroutine that completes when all the coroutines passed to it are done (that's the default behavior of `wait`; see below).

⓫ Execute the event loop until `wait_coro` is done; this is where the script will block while the event loop runs. We ignore the second item returned by `run_until_complete`. The reason is explained below.

⓬ Shut down the event loop.

> It would be nice if event loop instances were context managers, so we could use a `with` block to make sure the loop is closed. However, the situation is complicated by the fact that client code never creates the event loop directly, but gets a reference to it by calling `asyncio.get_event_loop()`. Sometimes our code does not "own" the event loop, so it would be wrong to close it. For example, when using an external GUI event loop with a package like Quamash, the Qt library is responsible for shutting down the loop when the application quits.

The `asyncio.wait(…)` coroutine accepts an iterable of futures or coroutines; `wait` wraps each coroutine in a `Task`. The end result is that all objects managed by `wait` become instances of `Future`, one way or another. Because it is a coroutine function, calling `wait(…)` returns a coroutine/generator object; this is what the `wait_coro` variable holds. To drive the coroutine, we pass it to `loop.run_until_complete(…)`.

The `loop.run_until_complete` function accepts a future or a coroutine. If it gets a coroutine, `run_until_complete` wraps it into a `Task`, similar to what `wait` does. Coroutines, futures and tasks can all be driven by `yield from`, and this is what `run_until_complete` does with the `wait_coro` object returned by the `wait` call. When `wait_coro` runs to completion, it returns a 2-tuple where the first item is the set of completed futures, and the second is the set of those not completed. In Example 18-5 the second set will always be empty — that's why we explicitly ignore it by assigning to `_`. But `wait` accepts two keyword-only arguments that may cause it to return even if some of the futures are not complete: `timeout` and `return_when`. See the `asyncio.wait` documentation for details.

Note that in Example 18-5 I could not reuse the `get_flag` function from `flags.py` (Example 17-2) because that uses the `requests` library which performs blocking I/O. To leverage `asyncio` we must replace every function that hits the network with an asynchronous version that is invoked with `yield from`, so that control is given back to the event loop. Using `yield from` in `get_flag` means that it must be driven as a coroutine.

That's why I could not reuse the download_one function from flags_threadpool.py (Example 17-3) either. The code in Example 18-5 drives get_flag with yield_from, so download_one is itself also a coroutine. For each request, a download_one coroutine object is created in download_many, and they are all driven by the loop.run_until_complete function, after being wrapped by the asyncio.wait coroutine.

There are a lot of new concepts to grasp in asyncio but the overall logic of Example 18-5 is easy to follow if you employ a trick suggested by Guido van Rossum himself: squint and pretend the yield from keywords are not there. If you do that, you'll notice that the code is as easy to read as plain old sequential code.

For example, imagine that the body of this coroutine…

```
@asyncio.coroutine
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)
    image = yield from resp.read()
    return image
```

…works like the function below, except that it never blocks:

```
def get_flag(cc):
    url = '{}/{cc}/{cc}.gif'.format(BASE_URL, cc=cc.lower())
    resp = aiohttp.request('GET', url)
    image = resp.read()
    return image
```

Using the yield from foo syntax avoids blocking because the current coroutine is suspended (i.e. the delegating generator where the yield from code is), but the control flow goes back to the event loop, which can drive other coroutines. When the foo future or coroutine is done, it returns a result to the suspended coroutine, resuming it.

At the end of the section "Using yield from" on page 479, I stated two facts about every usage of yield from. Here they are, summarized:

1. Every arrangement of coroutines chained with yield from must be ultimately driven by a caller that is not a coroutine, which invokes next(…) or .send(…) on the outermost delegating generator, explicitly or implicitly (e.g. in a for loop).
2. The innermost subgenerator in the chain must be a simple generator that uses just yield — or an iterable object.

When using yield from with the asyncio API, both facts above remain true, with the following specifics:

1. The coroutine chains we write are always driven by passing our outermost delegating generator to an asyncio API call, such as loop.run_until_complete(…).

> In other words, when using asyncio our code doesn't drive a coroutine chain by calling next(…) or .send(…) on it — the asyncio event loop does that. . The coroutine chains we write always end by delegating with yield from to some asyncio coroutine function or coroutine method, e.g. yield from asyncio.sleep(…) in Example 18-2 or coroutines from libraries that implement higher level protocols, such as resp = yield from aiohttp.request('GET', url) in the get_flag coroutine of Example 18-5.
>
> In other words, the innermost subgenerator will be a library function that does the actual I/O, not something we write.

To summarize: as we use asyncio, our asynchronous code consists of coroutines that are delegating generators driven by asyncio itself and that ultimately delegate to asyncio library coroutines — possibly by way of some third-party library such as aiohttp. This arrangement creates pipelines where the asyncio event loop drives — through our coroutines — the library functions that perform the low-level asynchronous I/O.

We are now ready to answer one question raised in Chapter 17:

- How can flags_asyncio.py perform 5x faster than flags.py when both are single threaded?

Running circles around blocking calls

Ryan Dahl, the inventor of Node.js, introduces the philosophy of his project by saying "We're doing I/O completely wrong[4]." He defines a *blocking function* as one that does disk or network I/O, and argues that we can't treat them as we treat non-blocking functions. To explain why, he presents the numbers in the first two columns of Table 18-1.

Table 18-1. Modern computer latency for reading data from different devices. Third column shows proportional times in a scale easier to understand for us slow humans.

device	CPU cycles	Proportional "human" scale
L1 cache	3	3 seconds
L2 cache	14	14 seconds
RAM	250	250 seconds
disk	41,000,000	1.3 years
network	240,000,000	7.6 years

[4] Video: Introduction to Node.js at 4'55".

To make sense of Table 18-1, bear in mind that modern CPUs with GHz clocks run billions of cycles per second. Let's say that a CPU runs exactly 1 billion cycles per second. That CPU can make 333,333,333 L1 cache reads in one second, or 4 (four!) network reads in the same time. The third column of Table 18-1 puts those numbers in perspective by multiplying the second column by a constant factor. So — in an alternate universe — if one read from L1 cache took 3 seconds, then a network read would take 7.6 years!

There are two ways to prevent blocking calls to halt the progress of the entire application:

1. Run each blocking operation in a separate thread.
2. Turn every blocking operation into a non-blocking asynchronous call.

Threads work fine, but the memory overhead for each OS thread — the kind that Python uses — is on the order of megabytes, depending on the OS. We can't afford one thread per connection if we are handling thousands of connections.

Callbacks are the traditional way to implement asynchronous calls with low memory overhead. They are a low-level concept, similar to the oldest and most primitive concurrency mechanism of all: hardware interrupts. Instead of waiting for a response, we register a function to be called when something happens. In this way every call we make can be non-blocking. Ryan Dahl advocates callbacks for their simplicity and low overhead.

Of course we can only make callbacks work because the event loop underlying our asynchronous applications can rely on infrastructure that uses interrupts, threads, polling, background processes etc. to insure that multiple concurrent requests make progress and they eventually get done[5]. When the event loop gets a response, it calls back our code. But the single main thread shared by the event loop and our application code is never blocked — if we don't make mistakes.

When used as coroutines, generators provide an alternative way to do asynchronous programming. From the perspective of the event loop, invoking a callback or calling `.send()` on a suspended coroutine is pretty much the same. There is a memory overhead for each suspended coroutine, but it's orders of magnitude smaller than the overhead for each thread. And they avoid the dreaded "callback hell", which we'll discuss in "From callbacks to futures and coroutines" on page 564.

Now the five-fold performance advantage of `flags_asyncio.py` over `flags.py` should make sense: `flags.py` spends billions of CPU cycles waiting for each download, one after the other. The CPU is actually doing a lot meanwhile, just not running your pro-

[5] In fact, although Node.js does not support user-level threads written in JavaScript, behind the scenes it implements a thread pool in C with the `libeio` library, to provide its callback-based file APIs — because as of 2014 there are no stable and portable asynchronous file handling APIs for most OSes.

gram. In contrast, when `loop_until_complete` is called in the `download_many` function of `flags_asyncio.py`, the event loop drives each `download_one` coroutine to the first `yield from`, and this in turn drives each `get_flag` coroutine to the first `yield from`, calling `aiohttp.request(…)`. None of these calls are blocking, so all requests are started in a fraction of a second.

As the `asyncio` infrastructure gets the first response back, the event loop sends it to the waiting `get_flag` coroutine. As `get_flag` gets a response, it advances to the next `yield from`, which calls `resp.read()` and yields control back to the main loop. Other responses arrive in close succession (since they were made almost at the same time). As each `get_flag` returns, the delegating generator `download_flag` resumes and saves the image file.

> For maximum performance, the `save_flag` operation should be asynchronous, but `asyncio` does not provide an asynchronous filesystem API at this time — as Node does. If that becomes a bottleneck in your application, you can use the `loop.run_in_executor` function to run `save_flag` in a thread pool.]. Example 18-9 will show how.

Since the asynchronous operations are interleaved, the total time needed to download many images concurrently is much less than doing it sequentially. When making 600 HTTP requests with `asyncio` I got all results back more than 70 times faster than with a sequential script.

Now let's go back to the HTTP client example to see how we can display an animated progress bar and perform proper error handling.

Enhancing the `asyncio` downloader script

Recall from "Downloads with progress display and error handling" on page 522 that the `flags2` set of examples share the same command line interface. This includes the `flags2_asyncio.py` we will analyse in this section. For instance, Example 18-6 shows how to get 100 flags (`-al 100`) from the `ERROR` server, using 100 concurrent requests (`-m 100`).

Example 18-6. Running `flags2_asyncio.py`.

```
$ python3 flags2_asyncio.py -s ERROR -al 100 -m 100
ERROR site: http://localhost:8003/flags
Searching for 100 flags: from AD to LK
100 concurrent connections will be used.
--------------------
73 flags downloaded.
```

```
27 errors.
Elapsed time: 0.64s
```

Act responsibly when testing concurrent clients

Even if the overall download time is not different between the threaded and `asyncio` HTTP clients, `asyncio` can send requests faster, so it's even more likely that the server will suspect a DOS attack. To really exercise these concurrent clients at full speed, please setup a local HTTP server for testing, as explained in the `README.rst` inside the `17-futures/countries/` directory of the *Fluent Python* `example-code` repository on Github.

Now let's see how `flags2_asyncio.py` is implemented.

Using `asyncio.as_completed`

In Example 18-5 I passed a list of coroutines to `asyncio.wait`, which — when driven by `loop.run_until.complete` — would return the results of the downloads when all were done. But to update a progress bar we need to get results as they are done. Fortunately there is an `asyncio` equivalent of the `as_completed` generator function we used in the thread pool example with the progress bar (Example 17-14).

Writing a `flags2` example to leverage `asyncio` entails rewriting several functions that the `concurrent.future` version could reuse. That's because there's only one main thread in an `asyncio` program and we can't afford to have blocking calls in that thread, since it's the same thread that runs the event loop. So I had to rewrite `get_flag` to use `yield from` for all network access. Now `get_flag` is a coroutine, so `download_one` must drive it with `yield from`, therefore `download_one` itself becomes a coroutine. Previously, in Example 18-5, `download_one` was driven by `download_many`: the calls to `download_one` were wrapped in an `asyncio.wait` call and passed to `loop.run_until_complete`. Now we need finer control for progress reporting and error handling, so I moved most of the logic from `download_many` into a new `downloader_coro` coroutine, and use `download_many` just to setup the event loop and schedule `downloader_coro`.

Example 18-7 shows the top of the `flags2_asyncio.py` script where the `get_flag` and `download_one` coroutines are defined. Example 18-8 lists the rest of the source, with `downloader_coro` and `download_many`.

Example 18-7. `flags2_asyncio.py`: Top portion of the script; remaining code is in Example 18-8.

```python
import asyncio
import collections

import aiohttp
```

```
from aiohttp import web
import tqdm

from flags2_common import main, HTTPStatus, Result, save_flag

# default set low to avoid errors from remote site, such as
# 503 - Service Temporarily Unavailable
DEFAULT_CONCUR_REQ = 5
MAX_CONCUR_REQ = 1000


class FetchError(Exception):  ❶
    def __init__(self, country_code):
        self.country_code = country_code


@asyncio.coroutine
def get_flag(base_url, cc):  ❷
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    resp = yield from aiohttp.request('GET', url)
    if resp.status == 200:
        image = yield from resp.read()
        return image
    elif resp.status == 404:
        raise web.HTTPNotFound()
    else:
        raise aiohttp.HttpProcessingError(
            code=resp.status, message=resp.reason,
            headers=resp.headers)


@asyncio.coroutine
def download_one(cc, base_url, semaphore, verbose):  ❸
    try:
        with (yield from semaphore):  ❹
            image = yield from get_flag(base_url, cc)  ❺
    except web.HTTPNotFound:  ❻
        status = HTTPStatus.not_found
        msg = 'not found'
    except Exception as exc:
        raise FetchError(cc) from exc  ❼
    else:
        save_flag(image, cc.lower() + '.gif')  ❽
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose and msg:
        print(cc, msg)

    return Result(status, cc)
```

❶ This custom exception will be used to wrap other HTTP or network exceptions and carry the `country_code` for error reporting.

❷ `get_flag` will either return the bytes of the image downloaded, raise `web.HTTPNotFound` if the HTTP response status is 404 or raise an `aiohttp.HttpProcessingError` for other HTTP status codes.

❸ The `semaphore` argument is an instance of `asyncio.Semaphore`, a synchronization device that limits the number of concurrent requests.

❹ A `semaphore` is used as a context manager in a `yield from` expression so that the system as whole is not blocked: only this coroutine is blocked while the semaphore counter is at the maximum allowed number.

❺ When this `with` statement exits, the `semaphore` counter is decreased, unblocking some other coroutine instance that may be waiting for the same `semaphore` object.

❻ If the flag was not found, just set the status for the `Result` accordingly.

❼ Any other exception will be reported as a `FetchError` with the country code and the original exception chained using the `raise X from Y` syntax introduced in PEP 3134 — Exception Chaining and Embedded Tracebacks.

❽ This function call actually saves the flag image to disk.

In Example 18-7 you can see that the code for `get_flag` and `download_one` changed significantly from the sequential version because these functions are now coroutines using `yield from` to make asynchronous calls.

Network client code of the sort we are studying should always use some throttling mechanism to avoid pounding the server with too many concurrent requests — the overall performance of the system may degrade if the server is overloaded. In `flags2_threadpool.py` (Example 17-14) the throttling was done by instantiating the `ThreadPoolExecutor` with the required `max_workers` argument set to `concur_req` in the `download_many` function, so only `concur_req` threads are started in the pool. In `flags2_asyncio.py`, I used an `asyncio.Semaphore` which is created by the `downloader_coro` function (shown next, in Example 18-8) and is passed as the `semaphore` argument to `download_one` in Example 18-7[6].

A `Semaphore` is an object that holds an internal counter that is decreased whenever we call the `.acquire()` coroutine method on it, and increased when we call the `.release()` coroutine method. The initial value of the counter is created when the `Semaphore` is instantiated, as in this line of `downloader_coro`:

[6] Thanks to Guto Maia who noted that `Semaphore` was not explained in the book draft.

```
        semaphore = asyncio.Semaphore(concur_req)
```

Calling .acquire() does not block when the counter is greater than zero, but if the counter is zero, .acquire() will block the calling coroutine until some other coroutine calls .release() on the same Semaphore, thus increasing the counter. In Example 18-7 I don't call .acquire() or .release(), but use the semaphore as a context manager in this block of code inside download_one:

```
            with (yield from semaphore):
                image = yield from get_flag(base_url, cc)
```

That snippet guarantees that no more than concur_req instances of get_flags coroutines will be started at any time.

Now let's take a look at the rest of the script in Example 18-8. Note that most functionality of the old download_many function is now in a coroutine, downloader_coro. This was necessary because we must use yield from to retrieve the results of the futures yielded by asyncio.as_completed, therefore as_completed must be invoked in a coroutine. However, I couldn't simply turn download_many into a coroutine, because I must pass it to the main function from flags2_common in the last line of the script, and that main function is not expecting a coroutine, just a plain function. Therefore I created downloader_coro to run the as_completed loop, and now download_many simply sets up the event loop and schedules downloader_coro by passing it to loop.run_until_complete.

Example 18-8. flags2_asyncio.py: Script continued from Example 18-7.

```
@asyncio.coroutine
def downloader_coro(cc_list, base_url, verbose, concur_req):  ❶
    counter = collections.Counter()
    semaphore = asyncio.Semaphore(concur_req)  ❷
    to_do = [download_one(cc, base_url, semaphore, verbose)
             for cc in sorted(cc_list)]  ❸

    to_do_iter = asyncio.as_completed(to_do)  ❹
    if not verbose:
        to_do_iter = tqdm.tqdm(to_do_iter, total=len(cc_list))  ❺
    for future in to_do_iter:  ❻
        try:
            res = yield from future  ❼
        except FetchError as exc:  ❽
            country_code = exc.country_code  ❾
            try:
                error_msg = exc.__cause__.args[0]  ❿
            except IndexError:
                error_msg = exc.__cause__.__class__.__name__  ⓫
            if verbose and error_msg:
                msg = '*** Error for {}: {}'
                print(msg.format(country_code, error_msg))
            status = HTTPStatus.error
        else:
            status = res.status

        counter[status] += 1  ⓬

    return counter  ⓭


def download_many(cc_list, base_url, verbose, concur_req):
    loop = asyncio.get_event_loop()
    coro = downloader_coro(cc_list, base_url, verbose, concur_req)
    counts = loop.run_until_complete(coro)  ⓮
    loop.close()  ⓯

    return counts


if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

❶ The coroutine receives the same arguments as `download_many`, but it cannot be invoked directly from `main` precisely because it's a coroutine function and not a plain function like `download_many`.

❷ Create an `asyncio.Semaphore` that will allow up to `concur_req` active coroutines among those using this semaphore.

❸ Create a list of coroutine objects, one per call to the `download_one` coroutine.

❹ Get an iterator that will return futures as they are done.

❺ Wrap the iterator in the `tqdm` function to display progress.

❻ Iterate over the completed futures; this loop is very similar to the one in `download_many` in Example 17-14; most changes have to do with exception handling because of differences in the HTTP libraries (`requests` versus `aiohttp`).

❼ The easiest way to retrieve the result of an `asyncio.Future` is using `yield from` instead of calling `future.result()`.

❽ Every exception in `download_one` is wrapped in a `FetchError` with the original exception chained.

❾ Get the country code where the error occurred from the `FetchError` exception.

❿ Try to retrieve the error message from the original exception (`__cause__`).

⓫ If the error message cannot be found in the original exception, use the name of the chained exception class as the error message.

⓬ Tally outcomes.

⓭ Return the counter, as done in the other scripts.

⓮ download_many simply instantiates the coroutine and passes it to the event loop with run_until_complete.

⓯ When all work is done, shut down the event loop and return counts.

In Example 18-8 we could not use the mapping of futures to country codes we saw in Example 17-14 because the futures returned by asyncio.as_completed are not necessarily the same futures we pass into the as_completed call. Internally, the asyncio machinery replaces the future objects we provide with others that will, in the end, produce the same results[7].

Because I could not use the futures as keys to retrieve the country code from a dict in case of failure, I implemented the custom FetchError exception (shown in Example 18-7). FetchError wraps a network exception and hold the country code associated with the it, so the country code can be reported with the error in verbose mode. If there is no error, the country code is available as the result of the yield from future expression at the top of the for loop.

This wraps up the discussion of an asyncio example functionally equivalent to the flags2_threadpool.py we saw earlier. Next, we'll implement enhancements to flags2_asyncio.py that will let us explore asyncio further.

While discussing Example 18-7 I noted that save_flag performs disk I/O and should be executed asynchronously. The following section shows how.

Using an executor to avoid blocking the event loop

In the Python community we tend to overlook the fact that local filesystem access is blocking, rationalizing that it doesn't suffer from the higher latency of network access — which is also dangerously unpredictable. In contrast, Node.js programmers are constantly reminded that all filesystem functions are blocking because their signatures require a callback. Recall from Table 18-1 that blocking for disk I/O wastes millions of CPU cycles, and this may have a significant impact on the performance of the application.

In Example 18-7, the blocking function is save_flag. In the threaded version of the script (Example 17-14), save_flag blocks the thread that's running the download_one function, but that's only one of several worker threads. Behind the scenes, the blocking I/O call releases the GIL, so another thread can proceed. But in flags2_asyncio.py, save_flag blocks the single thread our code shares with the asyncio event loop, therefor

[7] A detailed discussion about this can be found in a thread I started in the python-tulip group, titled Which other futures my come out of asyncio.as_completed?. Guido responds, and gives insight on the implementation of as_completed as well as the close relationship between futures and coroutines in asyncio.

the whole application freezes while the file is being saved. The solution to this problem is the `run_in_executor` method of the event loop object.

Behind the scenes, the `asyncio` event loop has a thread pool executor, and you can send callables to be executed by it with `run_in_executor`. To use this feature in our example, only a few lines need to change in the `download_one` coroutine, as shown in Example 18-9.

Example 18-9. `flags2_asyncio_executor.py`: Using the default thread pool executor to run `save_flag`.

```
@asyncio.coroutine
def download_one(cc, base_url, semaphore, verbose):
    try:
        with (yield from semaphore):
            image = yield from get_flag(base_url, cc)
    except web.HTTPNotFound:
        status = HTTPStatus.not_found
        msg = 'not found'
    except Exception as exc:
        raise FetchError(cc) from exc
    else:
        loop = asyncio.get_event_loop()  ❶
        loop.run_in_executor(None,  ❷
                save_flag, image, cc.lower() + '.gif')  ❸
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose and msg:
        print(cc, msg)

    return Result(status, cc)
```

❶ Get a reference to the event loop object.

❷ The first argument to `run_in_executor` is an executor instance; if `None`, the default thread pool executor of the event loop is used.

❸ The remaining arguments are the callable and its positional arguments.

When I tested Example 18-9 there was no noticeable change in performance for using `run_in_executor` to save the image files because they are not large (13 KB each, on average). But you'll see an effect if you edit the `save_flag` function in `flags2_common.py` to save 10 times as many bytes on each file — just by coding `fp.write(img*10)` instead of `fp.write(img)`. With an average download size of 130 KB, the advantage of using `run_in_executor` becomes clear. If you're downloading megapixel images, the speedup will be significant.

The advantage of coroutines over callbacks becomes evident when we need to coordinate asynchronous requests, and not just make completely independent requests. The next section explains the problem and the solution.

From callbacks to futures and coroutines

Event-oriented programming with coroutines requires some effort to master, so it's good to be clear on how it improves on the classic callback style. This is the theme of this section.

Anyone with some experience in callback-style event oriented programming knows the term "callback hell": the nesting of callbacks when one operation depends on the result of the previous operation. If you have three asynchronous calls that must happen in succession, you need to code callbacks nested three levels deep. Here is an example in JavaScript:

Example 18-10. Callback hell in JavaScript: nested anonymous functions, a.k.a. Pyramid of Doom.

```
api_call1(request1, function (response1) {
    // stage 1
    var request2 = step1(response1);

    api_call2(request2, function (response2) {
        // stage 2
        var request3 = step2(response2);

        api_call3(request3, function (response3) {
            // stage 3
            step3(response3);
        });
    });
});
```

In Example 18-10, `api_call1`, `api_call2` and `api_call3` are library functions your code uses to retrieve results asynchronously — perhaps `api_call1` goes to a database and `api_call2` gets data from a Web service, for example. Each of these take a callback function, which in JavaScript are often anonymous functions (they are named `stage1`, `stage2` and `stage3` in the Python example below). The `step1`, `step2` and `step3` here represent regular functions of your application, that process the responses received by the callbacks.

Here is how callback hell looks like in Python:

Example 18-11. Callback hell in Python: chained callbacks.

```
def stage1(response1):
    request2 = step1(response1)
    api_call2(request2, stage2)
```

```
def stage2(response2):
    request3 = step2(response2)
    api_call3(request3, stage3)


def stage3(response3):
    step3(response3)


api_call1(request1, stage1)
```

Although the code in Example 18-11 is arranged very differently from Example 18-10, they do exactly the same thing, and the JavaScript example could be written using the same arrangement (but the Python code can't be written in the JavaScript style because of the syntactic limitations of `lambda`).

Code organized as Example 18-10 or Example 18-11 is hard to read, but it's even harder to write: each function does part of the job, sets up the next callback and returns, to let the event loop proceed. At this point all local context is lost. When the next callback — e.g. `stage2` — is executed, you don't have the value of `request2` any more. If you need it, you must rely on closures or external data structures to store it between the different stages of the processing.

That's where coroutines really help. Within a coroutine, to perform three asynchronous actions in succession, you `yield` three times to let the event loop continue running. When a result is ready, the coroutine is activated with a `.send()` call. From the perspective of the event loop, that's similar to invoking a callback. But for the users of an coroutine-style asynchronous API, the situation is vastly improved: the entire sequence of three operations is in one function body, like plain old sequential code with local variables to retain the context of the overall task under way. See Example 18-12.

Example 18-12. Coroutines and `yield from` enable asynchronous programming without callbacks.

```
@asyncio.coroutine
def three_stages(request1):
    response1 = yield from api_call1(request1)
    # stage 1
    request2 = step1(response1)
    response2 = yield from api_call2(request2)
    # stage 2
    request3 = step2(response2)
    response3 = yield from api_call3(request3)
    # stage 3
    step3(response3)
```

```
loop.create_task(three_stages(request1))  # must explicitly schedule execution
```

Example 18-12 is much easier to follow the previous JavaScript and Python examples: the three stages of the operation appear one after the other inside the same function. This makes it trivial to use previous results in follow-up processing. It also provides a context for error reporting through exceptions.

Suppose in Example 18-11 the processing of the call `api_call2(request2, stage2)` raises an I/O exception (that's the last line of the `stage1` function). The exception cannot be caught in `stage1` because `api_call2` is an asynchronous call: it returns immediately, before any I/O is performed. In callback-based APIs this is solved by registering two callbacks for each asynchronous call: one for handling the result of successful operations, another for handling errors. Work conditions in callback hell quickly deteriorate when error handling is involved.

In contrast, in Example 18-12 all the asynchronous calls for this three-stage operation are inside the same function, `three_stages`, and if the asynchronous calls `api_call1`, `api_call2` and `api_call3` raise exceptions we can handle them by putting the respective `yield from` lines inside `try/except` blocks.

This is a much better place than callback hell, but I wouldn't call it coroutine heaven because there is a price to pay. Instead of regular functions you must use coroutines and get used to `yield from`, so that's the first obstacle. Once you write `yield from` in a function, it's now a coroutine and you can't simply call it, like we called `api_call1(request1, stage1)` in Example 18-11 to start the callback chain. You must explicitly schedule the execution of the coroutine with the event loop, or activate it using `yield from` in another coroutine that is scheduled for execution. Without the call `loop.create_task(three_stages(request1))` in the last line, nothing would happen in Example 18-12.

The next example puts this theory into practice.

Doing multiple requests for each download

Suppose you want to save each country flag with the name of the country and the country code, instead of just the country code. Now you need to make two HTTP requests per flag: one to get the flag image itself, the other to get the `metadata.json` file in the same directory as the image: that's where the name of the country is recorded.

Articulating multiple requests in the same task is easy in the threaded script: just make one request then the other, blocking the thread twice, and keeping both pieces of data (country code and name) in local variables, ready to use when saving the files. If you need to do the same in an asynchronous script with callbacks you start to smell the sulfur of callback hell: the country code and name will need to be passed around in a closure or held somewhere until you can save the file because each callback runs in a different local context. Coroutines and `yield from` provide relief from that. The solution is not as simple as with threads, but more manageable than chained or nested callbacks.

Example 18-13 shows code from the third variation of the `asyncio` flag downloading script, using the country name to save each flag. The `download_many` and `downloader_coro` are unchanged from `flags2_asyncio.py` (Example 18-7 and Example 18-8). The changes are:

`download_one`
: This coroutine now uses `yield from` to delegate to `get_flag` and the new `get_country` coroutine.

`get_flag`
: Most code from this coroutine was moved to a new `http_get` coroutine so it can also be used by `get_country`.

`get_country`
: This coroutine fetches the `metadata.json` file for the country code, and gets the name of the country from it.

`http_get`
: Common code for getting a file from the Web.

Example 18-13. `flags3_asyncio.py`: More coroutine delegation to perform two requests per flag.

```
@asyncio.coroutine
def http_get(url):
    res = yield from aiohttp.request('GET', url)
    if res.status == 200:
        ctype = res.headers.get('Content-type', '').lower()
        if 'json' in ctype or url.endswith('json'):
            data = yield from res.json()  ❶
        else:
            data = yield from res.read()  ❷
        return data

    elif res.status == 404:
        raise web.HTTPNotFound()
    else:
        raise aiohttp.errors.HttpProcessingError(
            code=res.status, message=res.reason,
            headers=res.headers)


@asyncio.coroutine
def get_country(base_url, cc):
    url = '{}/{cc}/metadata.json'.format(base_url, cc=cc.lower())
    metadata = yield from http_get(url)  ❸
    return metadata['country']


@asyncio.coroutine
def get_flag(base_url, cc):
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    return (yield from http_get(url))  ❹


@asyncio.coroutine
def download_one(cc, base_url, semaphore, verbose):
    try:
        with (yield from semaphore):  ❺
            image = yield from get_flag(base_url, cc)
        with (yield from semaphore):
            country = yield from get_country(base_url, cc)
    except web.HTTPNotFound:
        status = HTTPStatus.not_found
        msg = 'not found'
    except Exception as exc:
        raise FetchError(cc) from exc
    else:
        country = country.replace(' ', '_')
        filename = '{}-{}.gif'.format(country, cc)
        loop = asyncio.get_event_loop()
        loop.run_in_executor(None, save_flag, image, filename)
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose and msg:
        print(cc, msg)

    return Result(status, cc)
```

❶ If the content type has `'json'` in it or the `url` ends with `.json`, use the response `.json()` method to parse it and return a Python data structure — in this case, a `dict`.

❷ Otherwise, use `.read()` to fetch the bytes as they are.

❸ `metadata` will receive a Python `dict` built from the JSON contents.

❹ The outer parenthesis here are required because the Python parser gets confused and produces a syntax error when it sees the keywords `return yield from` lined up like that.

❺ I put the calls to `get_flag` and `get_country` in separate `with` blocks controlled by the `semaphore` because I want to keep it acquired for the shortest possible time.

The `yield from` syntax appears 9 times in Example 18-13. By now you should be getting the hang of how this construct is used to delegate from one coroutine to another without blocking the event loop.

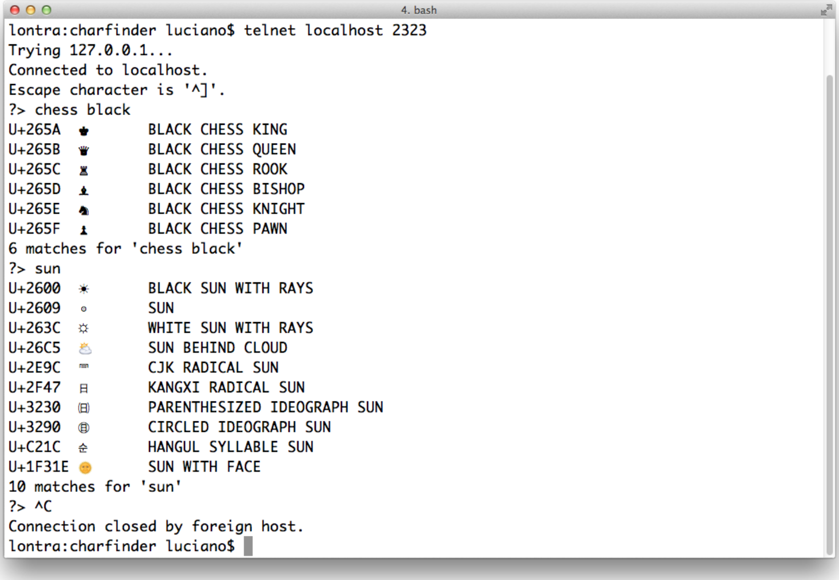
The challenge is to know when you have to use `yield from` and when you can't use it. The answer in principle is easy, you `yield from` coroutines and `asyncio.Future` instances — including tasks. But some APIs are tricky, mixing coroutines and plain functions in seemingly arbitrary ways, like the `StreamWriter` class we'll use in one of the servers in the next section.

Example 18-13 wraps up the `flags2` set of examples. I encourage you to play with them to develop an intuition of how concurrent HTTP clients perform. Use the `-a`, `-e` and `-l` command line options to control the number of downloads, and the `-m` option to set the number of concurrent downloads. Run tests against the `LOCAL`, `REMOTE`, `DELAY` and `ERROR` servers. Discover the optimum number of concurrent downloads to maximize throughput against each server. Tweak the settings of the `vaurien_error_delay.sh` script to add or remove errors and delays.

We'll now go from client scripts to writing servers with `asyncio`.

Writing `asyncio` servers

The classic toy example of a TCP server is an echo server. We'll build slightly more interesting toys: Unicode character finders, first using plain TCP, then using HTTP. These servers will allow clients to query for Unicode characters based on words in their canonical names, using the `unicodedata` module we discussed in "The Unicode database" on page 126. A Telnet session with the TCP character finder server, searching for chess pieces and characters with the word "sun" is shown in Figure 18-2.



```
lontra:charfinder luciano$ telnet localhost 2323
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
?> chess black
U+265A  ♚       BLACK CHESS KING
U+265B  ♛       BLACK CHESS QUEEN
U+265C  ♜       BLACK CHESS ROOK
U+265D  ♝       BLACK CHESS BISHOP
U+265E  ♞       BLACK CHESS KNIGHT
U+265F  ♟       BLACK CHESS PAWN
6 matches for 'chess black'
?> sun
U+2600  ☀       BLACK SUN WITH RAYS
U+2609  ☉       SUN
U+263C  ☼       WHITE SUN WITH RAYS
U+26C5  ⛅       SUN BEHIND CLOUD
U+2E9C  ⺜       CJK RADICAL SUN
U+2F47  ⽇       KANGXI RADICAL SUN
U+3230  ㈰       PARENTHESIZED IDEOGRAPH SUN
U+3290  ㊐       CIRCLED IDEOGRAPH SUN
U+C21C  순       HANGUL SYLLABLE SUN
U+1F31E 🌞      SUN WITH FACE
10 matches for 'sun'
?> ^C
Connection closed by foreign host.
lontra:charfinder luciano$ 
```

Figure 18-2. A Telnet session with the `tcp_charfinder.py` server: querying for "chess black" and "sun".

Now, on to the implementations.

An `asyncio` TCP server

Most of the logic in these examples is in the `charfinder.py` module, which has nothing concurrent about it. You can use `charfinder.py` as a command-line character finder but, more importantly, it was designed to provide content for our `asyncio` servers. The code for `charfinder.py` is in the example code repository for Fluent Python.

The `charfinder` module indexes each word that appears in character names in the Unicode database bundled with Python, and creates an inverted index stored in a `dict`. For example, the inverted index entry for the key `'SUN'` contains a `set` with the 10 Unicode characters that have that word in their names. The inverted index is saved in a local `charfinder_index.pickle` file. If multiple words appear in the query, `charfinder` computes the intersection of the sets retrieved from the index.

We'll now focus on the `tcp_charfinder.py` script that is answering the queries in Figure 18-2. Because I have a lot to say about this code, I split in two parts in the book: Example 18-14 and Example 18-15.

Example 18-14. `tcp_charfinder.py`: A simple TCP server using `asyncio.start_server`. Code for this module continues in Example 18-15.

```
import sys
import asyncio

from charfinder import UnicodeNameIndex  ❶

CRLF = b'\r\n'
PROMPT = b'?> '

index = UnicodeNameIndex()  ❷

@asyncio.coroutine
def handle_queries(reader, writer):  ❸
    while True:  ❹
        writer.write(PROMPT)  # can't yield from!  ❺
        yield from writer.drain()  # must yield from!  ❻
        data = yield from reader.readline()  ❼
        try:
            query = data.decode().strip()
        except UnicodeDecodeError:  ❽
            query = '\x00'
        client = writer.get_extra_info('peername')  ❾
        print('Received from {}: {!r}'.format(client, query))  ❿
        if query:
            if ord(query[:1]) < 32:  ⓫
                break
            lines = list(index.find_description_strs(query)) ⓬
            if lines:
                writer.writelines(line.encode() + CRLF for line in lines)  ⓭
            writer.write(index.status(query, len(lines)).encode() + CRLF)  ⓮

            yield from writer.drain()  ⓯
            print('Sent {} results'.format(len(lines)))  ⓰

    print('Close the client socket')  ⓱
    writer.close()  ⓲
```

❶ `UnicodeNameIndex` is the class that builds the index of names and provides querying methods.

❷ When instantiated, `UnicodeNameIndex` uses `charfinder_index.pickle`, if available, or builds it, so the first run may take a few seconds longer to start[8].

[8] Leonardo Rochael pointed out that building the `UnicodeNameIndex` could be delegated to another thread using `loop.run_with_executor()` in the `main` function of Example 18-15, so the server would be ready to take requests immediately while the index is built. That is true, but querying the index is the only thing this app does, so that would not be a big win. It's an interesting exercise to do as Leo suggests, though. Go ahead and do it, if you like.

❸ This is the coroutine we need to pass to asyncio_startserver; the arguments received are an asyncio.StreamReader and an asyncio.StreamWriter.

❹ This loop handles a session which lasts until any control character is received from the client.

❺ The StreamWriter.write method is not a coroutine, just a plain function; this line sends the ?> prompt.

❻ StreamWriter.drain flushes the writer buffer; it is a coroutine, so it must be called with yield from.

❼ StreamWriter.readline is a coroutine; it returns bytes.

❽ A UnicodeDecodeError may happen when the Telnet client sends control characters; if that happens, we pretend a null character was sent, for simplicity.

❾ This returns the remote address to which the socket is connected.

❿ Log the query to the server console.

⓫ Exit the loop if a control or null character was received.

⓬ This returns a generator that yields strings with the Unicode codepoint, the actual character and its name, e.g. U+0039\t9\tDIGIT NINE; for simplicity, I build a list from it.

⓭ Send the lines converted to bytes using the default UTF-8 encoding, appending a carriage return and a line feed to each; note that the argument is a generator expression.

⓮ Write a status line such as 627 matches for 'digit'.

⓯ Flush the output buffer.

⓰ Log the response to the server console.

⓱ Log the end of the session to the server console.

⓲ Close the StreamWriter.

The handle_queries coroutine has a plural name because it starts an interactive session and handles multiple queries from each client.

Note that all I/O in Example 18-14 is in bytes. We need to decode the strings received from the network, and encode strings sent out. In Python 3, the default encoding is UTF-8, and that's what we are using implicitly.

One caveat is that some of the I/O methods are coroutines and must be driven with yield from, while others are simple functions. For example, StreamWriter.write is a plain function, on the assumption that most of the time it does not block because it writes to a buffer. On the other hand, StreamWriter.drain, which flushes the buffer and performs the actual I/O is a coroutine, as is Streamreader.readline. While I was writing this book, a major improvement to the `asyncio` API docs was the clear labeling of coroutines as such.

Example 18-15 lists the main function for the module started in Example 18-14.

Example 18-15. `tcp_charfinder.py` (continued from Example 18-14): the `main` function sets up and tears down the event loop and the socket server.

```
def main(address='127.0.0.1', port=2323):  ❶
    port = int(port)
    loop = asyncio.get_event_loop()
    server_coro = asyncio.start_server(handle_queries, address, port,
                                loop=loop)  ❷
    server = loop.run_until_complete(server_coro)  ❸

    host = server.sockets[0].getsockname()  ❹
    print('Serving on {}. Hit CTRL-C to stop.'.format(host))  ❺
    try:
        loop.run_forever()  ❻
    except KeyboardInterrupt:  # CTRL+C pressed
        pass

    print('Server shutting down.')
    server.close()  ❼
    loop.run_until_complete(server.wait_closed())  ❽
    loop.close()  ❾


if __name__ == '__main__':
    main(*sys.argv[1:])  ❿
```

❶ The `main` function can be called with no arguments.

❷ When completed, the coroutine object returned by `asyncio.start_server` returns an instance of `asyncio.Server`, a TCP socket server.

❸ Drive `server_coro` to bring up the `server`.

❹ Get address and port of the first socket of the server and…

❺ …display it on the server console. This is the first output generated by this script on the server console.

❻ Run the event loop; this is where `main` will block until killed when `CTRL-C` is pressed on the server console.

❼ Close the server.

❽ `server.wait_closed()` returns a future; use `loop.run_until_complete` to let the future do its job.

❾ Terminate the event loop.

❿ This is a shortcut for handling optional command-line arguments: explode `sys.argv[1:]` and pass it to a `main` function with suitable default arguments.

Note how `run_until_complete` accepts either a coroutine (the result of `start_server`) or a `Future` (the result of `server.wait_closed`). If `run_until_complete` gets a coroutine as argument, it wraps the coroutine in a `Task`.

You may find it easier to understand how control flows in `tcp_charfinder.py` if you take a close look at the output it generates on the server console, listed in Example 18-16.

Example 18-16. `tcp_charfinder.py`: this is the server side of the session depicted in Figure 18-2

```
$ python3 tcp_charfinder.py
Serving on ('127.0.0.1', 2323). Hit CTRL-C to stop.  ❶
Received from ('127.0.0.1', 62910): 'chess black'  ❷
Sent 6 results
Received from ('127.0.0.1', 62910): 'sun'  ❸
Sent 10 results
Received from ('127.0.0.1', 62910): '\x00'   ❹
Close the client socket  ❺
```

❶ This is output by `main`.

❷ First iteration of the `while` loop in `handle_queries`.

❸ Second iteration of the `while` loop.

❹ The user hit `CTRL-C`; the server receives a control character and closes the session.

❺ The client socket is closed but the server is still running, ready to service another client.

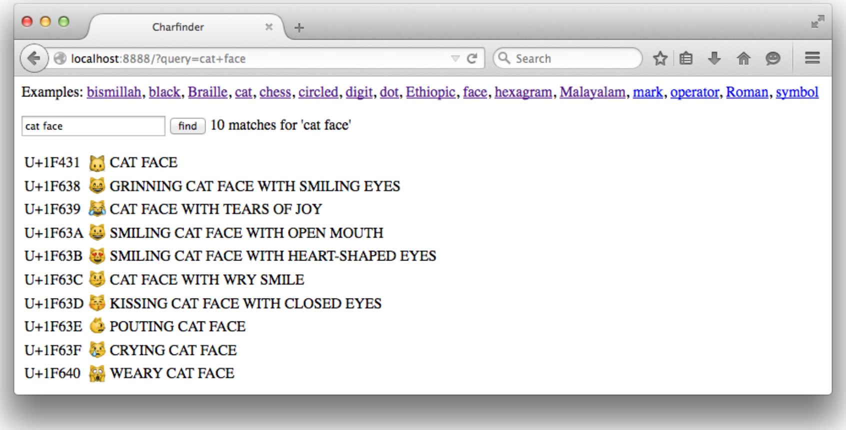
Note how `main` almost immediately displays the `Serving on...` message and blocks in the `loop.run_forever()` call. At that point, control flows into the event loop and stays there, occasionally coming back to the `handle_queries` coroutine, which yields control back to the event loop whenever it needs to wait for the network as it sends or receives data. While the event loop is alive, a new instance of the `handle_queries` coroutine will be started for each client that connects to the server. In this way, multiple clients can be handled concurrently by this simple server. This continues until a `KeyboardInterrupt` occurs or the process is killed by the OS.

The `tcp_charfinder.py` code leverages the high-level `asyncio` Streams API which provides a ready to use server so you only need to implement a handler function, which can be a plain callback or a coroutine. There is also a lower level Transports and Protocols API, inspired by the transport and protocols abstractions in the Twisted framework. Please refer to the the `asyncio` Transports and Protocols documentation for more information, including a TCP echo server implemented with that lower level API.

The next section presents an HTTP character finder server.

An `aiohttp` Web server

The `aiohttp` library we used for the `asyncio` flags examples also supports server-side HTTP, so that's what I used to implement the `http_charfinder.py` script. Figure 18-3 shows the simple Web interface of the server, displaying the result of a search for "cat face" emoji.



Examples: bismillah, black, Braille, cat, chess, circled, digit, dot, Ethiopic, face, hexagram, Malayalam, mark, operator, Roman, symbol

cat face find 10 matches for 'cat face'

```
U+1F431 🐱 CAT FACE
U+1F638 😸 GRINNING CAT FACE WITH SMILING EYES
U+1F639 😹 CAT FACE WITH TEARS OF JOY
U+1F63A 😺 SMILING CAT FACE WITH OPEN MOUTH
U+1F63B 😻 SMILING CAT FACE WITH HEART-SHAPED EYES
U+1F63C 😼 CAT FACE WITH WRY SMILE
U+1F63D 😽 KISSING CAT FACE WITH CLOSED EYES
U+1F63E 😾 POUTING CAT FACE
U+1F63F 😿 CRYING CAT FACE
U+1F640 🙀 WEARY CAT FACE
```

Figure 18-3. Browser window displaying search results for "cat face" on the `http_charfinder.py` server.

> Some browsers are better than others at displaying Unicode. The screen shot in Figure 18-3 was captured with Firefox on OSX, and I got the same result with Safari. But up-to-date Chrome and Opera browsers on the same machine did not display emoji characters like the cat faces. Other search results — e.g. "chess" — looked fine, so it's likely a font issue on Chrome and Opera on OSX.

We'll start by analyzing the most interesting part of `http_charfinder.py`: the bottom half where the event loop and the HTTP server is setup and torn down. See Example 18-17.

Example 18-17. `http_charfinder.py`: The `main` and `init` functions.

```
@asyncio.coroutine
def init(loop, address, port):  ❶
```

```
    app = web.Application(loop=loop)  ❷
    app.router.add_route('GET', '/', home)  ❸
    handler = app.make_handler()  ❹
    server = yield from loop.create_server(handler,
                                           address, port)  ❺
    return server.sockets[0].getsockname()  ❻

def main(address="127.0.0.1", port=8888):
    port = int(port)
    loop = asyncio.get_event_loop()
    host = loop.run_until_complete(init(loop, address, port))  ❼
    print('Serving on {}. Hit CTRL-C to stop.'.format(host))
    try:
        loop.run_forever()  ❽
    except KeyboardInterrupt:  # CTRL+C pressed
        pass
    print('Server shutting down.')
    loop.close()  ❾


if __name__ == '__main__':
    main(*sys.argv[1:])
```

❶ The `init` coroutine yields a server for the event loop to drive.

❷ The `aiohttp.web.Application` class represents a Web application…

❸ …with routes mapping URL patterns to handler functions; here `GET /` is routed to the `home` function (see Example 18-18)

❹ The `app.make_handler` method returns an `aiohttp.web.RequestHandler` instance to handle HTTP requests according to the routes set up in the `app` object.

❺ `create_server` brings up the server, using `handler` as the protocol handler and binding it to `address` and `port`.

❻ Return the address and port of the first server socket.

❼ Run `init` to start the server and get its address and port.

❽ Run the event loop; `main` will block here while the event loop is in control.

❾ Close the event loop.

As you get acquainted with the `asyncio` API, it's interesting to contrast how the servers are set up in Example 18-17 and in the TCP Example 18-15 shown earlier.

In the earlier TCP example, the server was created and scheduled to run in the `main` function with these two lines:

```
    server_coro = asyncio.start_server(handle_queries, address, port,
                                        loop=loop)
    server = loop.run_until_complete(server_coro)
```

In the HTTP example, the `init` function creates the server like this:

```
    server = yield from loop.create_server(handler,
                                            address, port)
```

But `init` itself is a coroutine, and what makes it run is the `main` function, with this line:

```
    host = loop.run_until_complete(init(loop, address, port))
```

Both `asyncio.start_server` and `loop.create_server` are coroutines that return `asyncio.Server` objects. In order to start up a server and return a reference to it, each of these coroutines must be driven to completion. In the TCP example, that was done by calling `loop.run_until_complete(server_coro)`, where `server_coro` was the result of `asyncio.start_server`. In the HTTP example, `create_server` is invoked on a `yield_from` expression inside the `init` coroutine, which is in turn driven by the `main` function when it calls `loop.run_until_complete(init(...))`.

I mention this to emphasize this essential fact we've discussed before: a coroutine only does anything when driven, and to drive an `asyncio.coroutine` you either use `yield from` or pass it to one of several `asyncio` functions that take coroutine or future arguments, such as `run_until_complete`.

Example 18-18 shows the `home` function which is configured to handle the `/` (root) URL in our HTTP server.

Example 18-18. `http_charfinder.py`: The `home` function.

```
def home(request):  ❶
    query = request.GET.get('query', '').strip()  ❷
    print('Query: {!r}'.format(query))  ❸
    if query:  ❹
        descriptions = list(index.find_descriptions(query))
        res = '\n'.join(ROW_TPL.format(**vars(descr))
                        for descr in descriptions)
        msg = index.status(query, len(descriptions))
    else:
        descriptions = []
        res = ''
        msg = 'Enter words describing characters.'

    html = template.format(query=query, result=res,  ❺
                           message=msg)
    print('Sending {} results'.format(len(descriptions)))  ❻
    return web.Response(content_type=CONTENT_TYPE, text=html)  ❼
```

❶ A route handler receives an `aiohttp.web.Request` instance.

❷ Get the query string stripped of leading and trailing blanks.

❸ Log query to server console.

❹ If there was a query, bind `res` to HTML table rows rendered from result of the query to the `index`, and `msg` to a status message.

❺ Render the HTML page.

❻ Log response to server console.

❼ Build `Response` and return it.

Note that `home` is not a coroutine, and does not need to be if there are no `yield from` expressions in it. The `aiohttp` documentation for the `add_route` method states that the handler "is converted to coroutine internally when it is a regular function".

There is a downside to the simplicity of the `home` function in Example 18-18. The fact that it's a plain function and not a coroutine is a symptom of a larger issue: the need to rethink how we code Web applications to achieve high concurrency. Let's consider this matter.

Smarter clients for better concurrency

The `home` function in Example 18-18 looks very much like a view function in Django or Flask. There is nothing asynchronous about its implementation: it gets a request, fetches data from a database and and builds a response by rendering a full HTML page. In this example the "database" is the `UnicodeNameIndex` object which is in memory. But accessing a real database should be done asynchronously, otherwise you're blocking the event loop while waiting for database results. For example, the `aiopg` package provides an asynchronous PostgreSQL driver compatible with `asyncio`; it lets you use `yield from` to send queries and fetch results, so your view function can behave as a proper coroutine.

Besides avoiding blocking calls, highly concurrent systems must split large chunks of work into smaller pieces to stay responsive. The `http_charfinder.py` server illustrates this point: if you search for "cjk" you'll get back 75821 Chinese, Japanese and Korean ideographs[9]. In this case, the `home` function will return a 5.3 MB HTML document, featuring a table with 75821 rows.

On my machine, it takes 2s to fetch the response to the "cjk" query, using the `curl` command-line HTTP client from a local `http_charfinder.py` server. A browser takes even longer to actually layout the page with such a huge table. Of course, most queries return much smaller responses: a query for "braille" returns 256 rows in a 19 KB page

[9] That's what CJK stands for: the ever-expanding set of Chinese, Japanese and Korean characters. Future versions of Python may support more CJK ideographs than Python 3.4 does.

and takes 0.017s on my machine. But if the server spends 2s serving a single "cjk" query, all the other clients will be waiting for at least 2s, and that is not acceptable.

The way to avoid the long response problem is to implement pagination: return results with at most, say, 200 rows, and have the user click or scroll the page to fetch more. If you look up the `charfinder.py` module in the Fluent Python code repository, you'll see that the `UnicodeNameIndex.find_descriptions` method takes optional `start` and `stop` arguments: they are offsets to support pagination. So you could return the first 200 results, then use AJAX or even WebSockets to send the next batch when — and if — the user wants to see it.

Most of the necessary coding for sending results in batches would be on the browser. This explains why Google and all large-scale Internet properties rely on lots of client-side coding to build their services: smart asynchronous clients make better use of server resources.

Although smart clients can help even old-style Django applications, to really serve them well we need frameworks that support asynchronous programming all the way: from the handling of HTTP requests and responses, to the database access. This is specially true if you want to implement real-time services such as games and media streaming with WebSockets[10].

Enhancing `http_charfinder.py` to support progressive download is left as an exercise to the reader. Bonus points if you implement "infinite scroll", like Twitter does. With this challenge I wrap up our coverage of concurrent programming with `asyncio`.

Chapter Summary

This chapter introduced a whole new way of coding concurrency in Python, leveraging `yield from`, coroutines, futures and the `asyncio` event loop. The first simple examples, the spinner scripts, were designed to demonstrate side-by-side the `threading` and the `asyncio` approaches to concurrency.

We then discussed the specifics of `asyncio.Future`, focusing on its support for `yield from`, and its relationship with coroutines and `asyncio.Task`. Next, we analyzed the `asyncio`-based flag download script.

We then reflected on Ryan Dahl's numbers for I/O latency and the effect of blocking calls. To keep a program alive despite the inevitable blocking functions there are two solutions: using threads or asynchronous calls — the latter being implemented as callbacks or coroutines.

[10] I have more to say about this trend in the **Soapbox** section at the end of this chapter.

In practice, asynchronous libraries depend on lower level threads to work — down to kernel level threads — but the user of the library doesn't create threads and doesn't need to be aware of their use in the infrastructure. At the application level, we just make sure none of our code is blocking, and the event loop takes care of the concurrency under the hood. Avoiding the overhead of user level threads is the main reason why asynchronous systems can manage more concurrent connections than multi threaded systems.

Resuming the flag downloading examples, adding a progress bar and proper error handling required significant refactoring, particularly with the switch from `asyncio.wait` to `asyncio.as_completed`, which forced us to move most of the functionality of `download_many` to a new `downloader_coro` coroutine, so we could use `yield from` to get the results from the futures produced by `asyncio.as_completed`, one by one.

We then saw how to delegate blocking jobs — such as saving a file — to a thread pool using the `loop.run_in_executor` method.

This was followed by a discussion of how coroutines solve the main problems of callbacks: loss of context when carrying out multi-step asynchronous tasks, and lack of a proper context for error handling.

The next example — fetching the country names along with the flag images — demonstrated how the combination of coroutines and `yield from` avoids the so-called callback hell. A multi-step procedure making asynchronous calls with `yield from` looks like simple sequential code, if you pay no attention to the `yield from` keywords.

The final examples in the chapter were `asyncio` TCP and HTTP servers that allow searching for Unicode characters by name. Analysis of the HTTP server ended with a discussion on the importance of client-side JavaScript to support higher concurrency on the server side, by enabling the client to make smaller requests on demand, instead of downloading large HTML pages.

Further reading

Nick Coghlan, a Python core developer, made the following comment on the draft of PEP-3156 — Asynchronous IO Support Rebooted: the "asyncio" Module in January, 2013:

> Somewhere early in the PEP, there may need to be a concise description of the two APIs for waiting for an asynchronous `Future`:
>
> 1. `f.add_done_callback(…)`
> 2. `yield from f` in a coroutine (resumes the coroutine when the future completes, with either the result or exception as appropriate)

> At the moment, these are buried in amongst much larger APIs, yet they're key to understanding the way everything above the core event loop layer interacts[11].

Guido van Rossum, the author of PEP-3156, did not heed Coghlan's advice. Starting with PEP-3156, the asyncio documentation is very detailed but not user friendly. The nine .rst files that make up the asyncio package docs total 128KB — that's roughly 71 pages. In the standard library, only the Built-in types chapter is bigger, and it covers the API for the numeric types, sequence types, generators, mappings, sets, bool, context managers etc.

Most pages in the asyncio manual focus on concepts and the API. There are useful diagrams and examples scattered all over it, but one section that is very practical is 18.5.11. Develop with asyncio, which presents essential usage patterns. The asyncio docs need more content explaining how asyncio should be used.

Because it's very new, asyncio lacks coverage in print. Jan Palach's *Parallel Programming with Python* (Packt, 2014) is the only book I found that has a chapter about asyncio, but it's a short chapter.

There are however excellent presentations about asyncio. The best I found is Brett Slatkin's Fan-in and Fan-out: The crucial components of concurrency, subtitled "Why do we need Tulip? (a.k.a. PEP 3156 — asyncio)", which he presented at PyCon 2014 in Montréal (video). In 30 minutes, Slatkin shows a simple web crawler example, highlighting how asyncio is intended to be used. Guido van Rossum is in the audience and mentions that he also wrote a web crawler as a motivating example for asyncio; Guido's code does not depend on aiohttp — it uses only the standard library. Slatkin also wrote the insightful post Python's asyncio is for composition, not raw performance.

Other must-see asyncio talks are by Guido van Rossum himself: the PyCon US 2013 keynote, and talks he gave at LinkedIn and Twitter University. Also recommended are Saúl Ibarra Corretgé's *A deep dive into PEP-3156 and the new asyncio module* (slides, video).

Dino Viehland showed how asyncio can be integrated with the Tkinter event loop in his Using futures for async GUI programming in Python 3.3 talk at PyCon US 2013. Viehland shows how easy it is to implement the essential parts of the asyncio.AbstractEventLoop interface on top of another event loop. His code was written with Tulip, prior to the addition of asyncio to the standard library; I adapted it to work with the Python 3.4 release of asyncio. My updated refactoring is on Gihub.

Victor Stinner — an asyncio core contributor and author of the Trollius backport — regularly updates a list of relevant links: The new Python asyncio module aka "tulip". Other collections of asyncio resources are Asyncio.org and aio-libs on Github, where

[11] Comment on PEP-3156 in a Jan. 20, 2013 message to the python-ideas list.

you'll find asynchronous drivers for PostgreSQL, MySQL and several NoSQL databases. I haven't tested these drivers, but the projects seem very active as I write this.

Web services are going to be an important use case for `asyncio`. Your code will likely depend on the `aiohttp` library led by Andrew Svetlov. You'll also want to setup an environment to test your error handling code, and the Vaurien "chaos TCP proxy" designed by Alexis Métaireau and Tarek Ziadé is invaluable for that. Vaurien was created for the Mozilla Services project and lets you introduce delays and random errors into the TCP traffic between your program and back-end servers such as databases and Web services providers.

Soapbox

The One Loop

For a long time asynchronous programming has been the approach favored by most Pythonistas for network applications, but there was always the dilemma of picking one of the mutually incompatible libraries. Ryan Dahl cites Twisted as a source of inspiration for Node.js, and Tornado championed the use of coroutines for event-oriented programming in Python.

In the JavaScript world there is some debate between advocates of simple callbacks and proponents of various competing higher level abstractions. Early versions the Node.js API used Promises — similar to our Futures — but Ryan Dahl decided to standardize on callbacks only. James Coglan argues this was Node's biggest missed opportunity.

In Python the debate is over: the addition of `asyncio` to the standard library establishes coroutines and futures as the Pythonic way of writing asynchronous code. Furthermore, the `asyncio` package defines standard interfaces for asynchronous futures and the event loop, providing reference implementations for them.

The *Zen of Python* applies perfectly:

> There should be one — and preferably only one — obvious way to do it.
>
> Although that way may not be obvious at first unless you're Dutch.

Maybe it takes a Dutch passport to find `yield from` obvious. It was not obvious at first for this Brazilian, but after a while I got the hang of it.

More importantly, `asyncio` was designed so that its event loop can be replaced by an external package. That's why the `asyncio.get_event_loop` and `set_event_loop` functions exist; they are part of an abstract Event Loop Policy API.

Tornado already has an AsyncIOMainLoop class that implements the `asyncio.AbstractEventLoop` interface, so you can run asynchronous code using both libraries on the same event loop. There is also the intriguing Quamash project that integrates `asyncio` to the Qt event loop for developing GUI applications with PyQt or PySide. These are just two of a growing number of interoperable event-oriented packages made possible by asyncio.

Smarter HTTP clients such as single-page Web applications (like Gmail) or smart phone apps demand quick, lightweight responses and push updates. These needs are better served by asynchronous frameworks instead of traditional Web frameworks like Django which are designed to serve fully rendered HTML pages and lack support for asynchronous database access.

The WebSockets protocol was designed to enable real-time updates for clients that are always connected, from games to streaming applications. This requires highly concurrent asynchronous servers able to keep ongoing interactions with hundreds or thousands of clients. WebSockets is very well supported by the asyncio architecture and at least two libraries already implement it on top of asyncio: Autobahn|Python and WebSockets.

This overall trend — dubbed "the real-time Web" — is a key factor in the demand for Node.js, and the reason why rallying around asyncio is so important for the Python ecosystem. There's still a lot of work to do. For starters, we need an asynchronous HTTP server and client API in the standard library, an asynchronous DBAPI 3.0 and new database drivers built on asyncio.

The biggest advantage Python 3.4 with asyncio has over Node.js is Python itself: a better designed language, with coroutines and yield from to make asynchronous code more maintainable than the primitive callbacks of JavaScript. Our biggest disadvantage is the libraries: Python comes with "batteries included", but our batteries are not designed for asynchronous programming. The rich ecosystem of libraries for Node.js is entirely built around async calls. But Python and Node.js both have a problem that Go and Erlang have solved from the start: we have no transparent way to write code that leverages all available CPU cores.

Standardizing the event loop interface and an asynchronous library was a major coup, and only our BDFL could have pulled it off, given that there were well entrenched, high quality alternatives available. He did it in consultation with the authors of the major Python asynchronous frameworks. The influence of Glyph Lefkowitz, the leader of Twisted, is most evident. Guido's post Deconstructing Deferred to the python-tulip group is a must-read if you want to understand why asyncio.Future is not like the Twisted Deferred class. Making clear his respect for the oldest and largest Python asynchronous framework, Guido also started the meme WWTD — What Would Twisted Do? — when discussing design options in the python-twisted group[12].

Fortunately, Guido van Rossum led the charge so Python is better positioned to face the concurrency challenges of the present. Mastering asyncio takes effort. But if you plan to write concurrent network applications in Python, seek the One Loop.

[12] See Guido's Jan. 29, 2015 message, immediately followed by an answer from Glyph.

One Loop to rule them all, One Loop to find them,
One Loop to bring them all and in liveness bind them.

PART VI

Metaprogramming

CHAPTER 19

Dynamic attributes and properties

> The crucial importance of properties is that their existence makes it perfectly safe and indeed advisable for you to expose public data attributes as part of your class's public interface[1].
>
> — Alex Martelli
> *Python contributor and book author*

Data attributes and methods are collectively known as *attributes* in Python: a method is just an attribute that is *callable*. Besides data attributes and methods, we can also create properties, which can be used to replace a public data attribute with *accessor methods* (i.e. getter/setter), without changing the class interface. This agrees with the *Uniform access principle*:

> All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation[2].

Besides properties, Python provides a rich API for controlling attribute access and implementing dynamic attributes. The interpreter calls special methods such as `__getattr__` and `__setattr__` to evaluate attribute access using dot notation, eg. `obj.attr`. A user-defined class implementing `__getattr__` can implement "virtual attributes" by computing values on the fly whenever somebody tries to read a nonexistent attribute like `obj.no_such_attribute`.

Coding dynamic attributes is the kind of metaprogramming that framework authors do. However, in Python the basic techniques are so straightforward that anyone can put them to work, even for everyday data wrangling tasks. That's how we'll start this chapter.

[1] Alex Martelli, *Python in a Nutshell, 2e.* (O'Reilly, 2006), p. 101

[2] Bertrand Meyer, *Object-Oriented Software Construction*, 2e., p. 57.

Data wrangling with dynamic attributes

In the next few examples we'll leverage dynamic attributes to work with a JSON data feed published by O'Reilly for the OSCON 2014 conference[3].

Example 19-1. Sample records from `osconfeed.json`; some field contents abbreviated.

```
{ "Schedule":
  { "conferences": [{"serial": 115 }],
    "events": [
      { "serial": 34505,
        "name": "Why Schools Don´t Use Open Source to Teach Programming",
        "event_type": "40-minute conference session",
        "time_start": "2014-07-23 11:30:00",
        "time_stop": "2014-07-23 12:10:00",
        "venue_serial": 1462,
        "description": "Aside from the fact that high school programming...",
        "website_url": "http://oscon.com/oscon2014/public/schedule/detail/34505",
        "speakers": [157509],
        "categories": ["Education"] }
    ],
    "speakers": [
      { "serial": 157509,
        "name": "Robert Lefkowitz",
        "photo": null,
        "url": "http://sharewave.com/",
        "position": "CTO",
        "affiliation": "Sharewave",
        "twitter": "sharewaveteam",
        "bio": "Robert ´r0ml´ Lefkowitz is the CTO at Sharewave, a startup..." }
    ],
    "venues": [
      { "serial": 1462,
        "name": "F151",
        "category": "Conference Venues" }
    ]
  }
}
```

Example 19-1 shows 4 out of the 895 records in the JSON feed. As you can see, the entire data set is a single JSON object with the key `"Schedule"`, and its value is another mapping with four keys: `"conferences"`, `"events"`, `"speakers"` and `"venues"`. Each of those four keys is paired with a list of records. In Example 19-1 each list has one record, but in the full dataset those lists have dozens or hundreds of records — except `"conferen`

[3] You can read about this feed and rules for using it at DIY: OSCON schedule. The original 744KB JSON file is still online as I write this. A copy named `osconfeed.json` can be found in the `oscon-schedule/data/` directory in the Fluent Python example code repository on GitHub.

`ces"` which holds just the single record shown. Every item in those four lists has a `"serial"` field which is a unique identifier within the list.

The first script I wrote to deal with the OSCON feed simply downloads the feed, avoiding unnecessary traffic by checking if there is a local copy. This makes sense because OSCON 2014 is history now, so that feed will not be updated.

There is no metaprogramming in Example 19-2, pretty much everything boils down to this expression: `json.load(fp)`, but that's enough to let us explore the dataset. The `osconfeed.load` function will be used in the next several examples.

Example 19-2. `osconfeed.py`: Downloading `osconfeed.json`. Doctests are in Example 19-3.

```
from urllib.request import urlopen
import warnings
import os
import json

URL = 'http://www.oreilly.com/pub/sc/osconfeed'
JSON = 'data/osconfeed.json'


def load():
    if not os.path.exists(JSON):
        msg = 'downloading {} to {}'.format(URL, JSON)
        warnings.warn(msg)  ❶
        with urlopen(URL) as remote, open(JSON, 'wb') as local:  ❷
            local.write(remote.read())

    with open(JSON) as fp:
        return json.load(fp)  ❸
```

❶ Issue a `warning` if a new download will be made.

❷ `with` using 2 context managers (allowed since Python 2.7 and 3.1) to read the remote file and save it.

❸ The `json.load` function parses a JSON file and returns native Python objects. In this feed we have the types: `dict`, `list`, `str` and `int`.

With the code in Example 19-2, we can inspect any field in the data. See Example 19-3.

Example 19-3. `osconfeed.py`: Doctests for Example 19-2.

```
    >>> feed = load()  ❶
    >>> sorted(feed['Schedule'].keys())  ❷
    ['conferences', 'events', 'speakers', 'venues']
    >>> for key, value in sorted(feed['Schedule'].items()):
    ...     print('{:3} {}'.format(len(value), key))  ❸
    ...
```

```
  1 conferences
484 events
357 speakers
 53 venues
>>> feed['Schedule']['speakers'][-1]['name']   ❹
'Carina C. Zona'
>>> feed['Schedule']['speakers'][-1]['serial']   ❺
141590
>>> feed['Schedule']['events'][40]['name']
'There *Will* Be Bugs'
>>> feed['Schedule']['events'][40]['speakers']   ❻
[3471, 5199]
```

❶ `feed` is a `dict` holding nested dicts and lists, with string and integer values.

❷ List the four record collections inside `"Schedule"`.

❸ Display record counts for each collection.

❹ Navigate through the nested dicts and lists to get the name of the last speaker.

❺ Get serial number of that same speaker.

❻ Each event has a `'speakers'` list with 0 or more speaker serial numbers.

Exploring JSON-like data with dynamic attributes

Example 19-2 is simple enough, but the syntax `feed['Schedule']['events'][40]['name']` is cumbersome. In JavaScript you can get the same value by writing `feed.Schedule.events[40].name`. It's easy to implement a `dict`-like class that does the same in Python — there are plenty of implementations on the web[4]. I implemented my own `FrozenJSON`, which is simpler than most recipes because it supports reading only: it's just for exploring the data. However, it's also recursive, dealing automatically with nested mappings and lists.

Example 19-4 is a demonstration of `FrozenJSON` and the source code is in Example 19-5.

Example 19-4. `FrozenJSON` from Example 19-5 allows reading attributes like `name` and calling methods like `.keys()` and `.items()`.

```
>>> from osconfeed import load
>>> raw_feed = load()
>>> feed = FrozenJSON(raw_feed)   ❶
>>> len(feed.Schedule.speakers)   ❷
357
>>> sorted(feed.Schedule.keys())   ❸
['conferences', 'events', 'speakers', 'venues']
>>> for key, value in sorted(feed.Schedule.items()):   ❹
...     print('{:3} {}'.format(len(value), key))
```

[4] An often mentioned one is AttrDict; another, allowing quick creation of nested mappings is addict.

```
...
  1 conferences
484 events
357 speakers
 53 venues
>>> feed.Schedule.speakers[-1].name  ❺
'Carina C. Zona'
>>> talk = feed.Schedule.events[40]
>>> type(talk)  ❻
<class 'explore0.FrozenJSON'>
>>> talk.name
'There *Will* Be Bugs'
>>> talk.speakers  ❼
[3471, 5199]
>>> talk.flavor  ❽
Traceback (most recent call last):
  ...
KeyError: 'flavor'
```

❶ Build a `FrozenJSON` instance from the `raw_feed` made of nested dicts and lists.

❷ `FrozenJSON` allows traversing nested dicts by using attribute notation; here we show the length of the list of speakers.

❸ Methods of the underlying dicts can also be accessed, like `.keys()`, to retrieve the record collection names.

❹ Using `items()` we can retrieve the record collection names and their contents, to display the `len()` of each of them.

❺ A `list`, such as `feed.Schedule.speakers` remains a list, but the items inside are converted to `FrozenJSON` if they are mappings.

❻ Item 40 in the `events` list was a JSON object, now it's a `FrozenJSON` instance.

❼ Event records have a `speakers` list with speaker serial numbers.

❽ Trying to read a missing attribute raises `KeyError`, instead of the usual `AttributeError`.

The keystone of the `FrozenJSON` class is the `__getattr__` method, which we already used in the `Vector` example in "`Vector` take #3: dynamic attribute access" on page 286, to retrieve `Vector` components by letter — `v.x`, `v.y`, `v.z` etc. It's essential to recall that the `__getattr__` special method is only invoked by the interpreter when the usual process fails to retrieve an attribute, i.e. when the named attribute cannot be found in the instance, nor in the class or in its superclasses.

The last line of Example 19-4 exposes a minor issue with the implementation: ideally, trying to read a missing attribute should raise `AttributeError`. I actually did implement the error handling, but it doubled the size of the `__getattr__` method and distracted from the most important logic I wanted to show, so I left it out for didactic reasons.

As you can see in Example 19-5, the FrozenJSON class has only two methods (__init__, __getattr__) and a __data instance attribute, so attempts to retrieve an attribute by any other name will trigger __getattr__. This method will first look if the self.__data dict has an attribute (not a key!) by that name; this allows FrozenJSON instances to handle any dict method such as items, by delegating to self.__data.items(). If self.__data doesn't have an attribute with the given name, __getattr__ uses name as a key to retrieve an item from self.__dict, and passes that item to FrozenJSON.build. This allows navigating through nested structures in the JSON data, as each nested mapping is converted to another FrozenJSON instance by the build class method.

Example 19-5. explore0.py: turn a JSON dataset into a FrozenJSON holding nested FrozenJSON objects, lists and simple types.

```
from collections import abc


class FrozenJSON:
    """A read-only façade for navigating a JSON-like object
       using attribute notation
    """

    def __init__(self, mapping):
        self.__data = dict(mapping)  ❶

    def __getattr__(self, name):  ❷
        if hasattr(self.__data, name):
            return getattr(self.__data, name)  ❸
        else:
            return FrozenJSON.build(self.__data[name])  ❹

    @classmethod
    def build(cls, obj):  ❺
        if isinstance(obj, abc.Mapping):  ❻
            return cls(obj)
        elif isinstance(obj, abc.MutableSequence):  ❼
            return [cls.build(item) for item in obj]
        else:  ❽
            return obj
```

❶ Build a dict from the mapping argument. This serves two purposes: ensures we got a dict (or something that can be converted to one) and makes a copy for safety.

❷ __getattr__ is called only when there's no attribute with that name.

❸ If name matches an attribute of the instance __data, return that. This is how calls to methods like keys are handled.

❹ Otherwise, fetch the item with the key `name` from `self.__data`, and return the result of calling `FrozenJSON.build()` on that[5].

❺ This is an alternate constructor, a common use for the `@classmethod` decorator.

❻ If `obj` is a mapping, build a `FrozenJSON` with it.

❼ If it is a `MutableSequence`, it must be a list[6], so we build a `list` by passing every item in `obj` recursively to `.build()`.

❽ If it's not a `dict` or a `list`, return the item as it is.

Note that no caching or transformation of the original feed is done. As the feed is traversed, the nested data structures are converted again and again into `FrozenJSON`. But that's OK for a data set of this size, and for a script that will only be used to explore or convert the data.

Any script that generates or emulates dynamic attribute names from arbitrary sources must deal with one issue: the keys in the original data may not be suitable attribute names. The next section addresses this.

The invalid attribute name problem

The `FrozenJSON` class has a limitation: there is no special handling for attributes names that are Python keywords. For example, if you build an object like this:

```
>>> grad = FrozenJSON({'name': 'Jim Bo', 'class': 1982})
```

You won't be able to read `grad.class` because `class` is a reserved word in Python:

```
>>> grad.class
  File "<stdin>", line 1
    grad.class
             ^
SyntaxError: invalid syntax
```

You can always do this, of course:

```
>>> getattr(grad, 'class')
1982
```

But the idea of `FrozenJSON` is to provide convenient access to the data, so a better solution is checking whether a key in the mapping given to `FrozenJSON.__init__` is a keyword, and if so, append an _ to it, so the attribute can be read like this:

[5] This line is where a `KeyError` exception may occur, in the expression `self.__data[name]`. It should be handled and an `AttributeError` raised instead, since that's what is expected from `__getattr__`. The diligent reader is invited to code the error handling as an exercise.

[6] The source of the data is JSON, and the only collection types in JSON data are `dict` and `list`.

```
>>> grad.class_
1982
```

This can be achieved by replacing the one-liner `__init__` from Example 19-5 with this version:

Example 19-6. `explore1.py`: append a _ to attribute names that are Python keywords.

```
def __init__(self, mapping):
    self.__data = {}
    for key, value in mapping.items():
        if keyword.iskeyword(key):   ❶
            key += '_'
        self.__data[key] = value
```

❶ The `keyword.iskeyword(…)` function is exactly what we need; to use it, the `keyword` module must be imported, which is not shown in this snippet.

A similar problem may arise if a key in the JSON is not a valid Python identifier:

```
>>> x = FrozenJSON({'2be':'or not'})
>>> x.2be
  File "<stdin>", line 1
    x.2be
      ^
SyntaxError: invalid syntax
```

Such problematic keys are easy to detect in Python 3 because the `str` class provides the `s.isidentifier()` method which tells you whether `s` is a valid Python identifier according to the language grammar. But turning a key that is not a valid identifier into valid attribute name is not trivial. Two simple solutions would be raising an exception or replacing the invalid keys with generic names like `attr_0`, `attr_1` and so on. For the sake of simplicity, I will not worry about this issue.

After giving some thought to the dynamic attribute names, let's turn to another essential feature of `FrozenJSON`: the logic of the `build` class method which is used by `__getattr__` to return a different type of object depending on the value of the attribute being accessed, so that nested structures are converted to `FrozenJSON` instances or lists of `FrozenJSON` instances.

Instead of a class method, the same logic could be implemented as the `__new__` special method, as we'll see next.

Flexible object creation with __new__

Although we often refer to `__init__` as the constructor method, that's because we adopted jargon from other languages. The special method that actually constructs an instance is `__new__`: it's a class method (but gets special treatment, so the `@classme` thod decorator is not used), and it must return an instance. That instance will in turn be passed as the first argument self of __init__. Since __init__ gets an instance when called, and it's actually forbidden from returning anything, __init__ is really an "initializer". The real constructor is __new__ — which we rarely to code because the implementation inherited from object suffices.

The path just described, from __new__ to __init__ is the usual, but not the only one. The __new__ method can also return an instance of a different class, and when that happens, the interpreter does not call __init__.

In other words, the process of building an object in Python can be summarized with this pseudo-code:

```
# pseudo-code for object construction
def object_maker(the_class, some_arg):
    new_object = the_class.__new__(some_arg)
    if isinstance(new_object, the_class):
        the_class.__init__(new_object, some_arg)
    return new_object

# the following statements are roughly equivalent
x = Foo('bar')
x = object_maker(Foo, 'bar')
```

Example 19-7 shows a variation of FrozenJSON where the logic of the former build class method was moved to __new__.

Example 19-7. explore2.py: using __new__ instead of build to construct new objects that may or may not be instances of FrozenJSON.

```
from collections import abc


class FrozenJSON:
    """A read-only façade for navigating a JSON-like object
       using attribute notation
    """

    def __new__(cls, arg):  ❶
        if isinstance(arg, abc.Mapping):
            return super().__new__(cls)  ❷
        elif isinstance(arg, abc.MutableSequence):  ❸
            return [cls(item) for item in arg]
        else:
            return arg

    def __init__(self, mapping):
        self.__data = {}
        for key, value in mapping.items():
            if iskeyword(key):
                key += '_'
            self.__data[key] = value

    def __getattr__(self, name):
        if hasattr(self.__data, name):
            return getattr(self.__data, name)
        else:
            return FrozenJSON(self.__data[name])  ❹
```

❶ As a class method, the first argument `__new__` gets is the class itself, and the remaining arguments are the same that `__init__` gets, except for `self`.

❷ The default behavior is to delegate to the `__new__` of a super class. In this case, we are calling `__new__` from the `object` base class, passing `FrozenJSON` as the only argument.

❸ The remaining lines of `__new__` are exactly as in the old `build` method.

❹ This was where `FrozenJSON.build` was called before; now we just call the `FrozenJSON` constructor.

The `__new__` method gets the class as the first argument because, usually, the created object will be an instance of that class. So, in `FrozenJSON.__new__`, when the expression `super().__new__(cls)` effectively calls `object.__new__(FrozenJSON)`, the instance built by the `object` class is actually an instance of `FrozenJSON` — i.e. the `__class__` attribute of the new instance will hold a reference to `FrozenJSON` — even though the actual construction is performed by `object.__new__`, implemented in C, in the guts of the interpreter.

There is an obvious shortcoming in the way the OSCON JSON feed is structured: the event at index `40`, titled `'There *Will* Be Bugs'` has two speakers, `3471` and `5199`, but finding those speakers is not easy, since those are serial numbers, and the `Schedule.speakers` list is not indexed by them. The `venue` field, present in every `event` record, also holds the a serial number, but finding the corresponding `venue` record requires a linear scan of the `Schedule.venues` list. Our next task is restructuring the data, and then automating the retrieval of linked records.

Restructuring the OSCON feed with `shelve`

The funny name of the standard `shelve` module makes sense when you realize that `pickle` is the name of the Python object serialization format — and of the module that converts objects to/from that format. Since pickle jars are kept in shelves, it makes sense that `shelve` provides `pickle` storage.

The `shelve.open` high-level function returns a `shelve.Shelf` instance — a simple key-value object database backed by the `dbm` module, with these characteristics:

- shelve.Shelf subclasses abc.MutableMapping, so it provides the essential methods we expect of a mapping type
- in addition, shelve.Shelf provides a few other IO management methods, like sync and close; it's also a context manager.
- Keys and values are saved whenever a new value is assigned to a key.
- The keys must be strings.
- The values must be objects that the pickle module can handle.

Please read the documentation for the shelve, dbm and pickle modules for the details and caveats. What matters to us now is that shelve provides a simple, efficient way to reorganize the OSCON schedule data: we will read all records from the JSON file and save them to a shelve.Shelf. Each key will be made from the record type and the serial number, eg. 'event.33950' or 'speaker.3471', and the value will be an instance of a new Record class we are about to introduce.

Example 19-8 shows the doctests for the schedule1.py script using shelve. To try it out interactively, run the script as python -i schedule1.py to get a console prompt with the module loaded. The load_db function does the heavy work: it calls osconfeed.load (from Example 19-2) to read the JSON data and saves each record as a Record instance in the Shelf object passed as db. After that, retrieving a speaker record is as easy as speaker = db['speaker.3471'].

Example 19-8. Trying out the functionality provided by schedule1.py (Example 19-9).

```
>>> import shelve
>>> db = shelve.open(DB_NAME)  ❶
>>> if CONFERENCE not in db:  ❷
...     load_db(db)  ❸
...
>>> speaker = db['speaker.3471']  ❹
>>> type(speaker)  ❺
<class 'schedule1.Record'>
>>> speaker.name, speaker.twitter  ❻
('Anna Martelli Ravenscroft', 'annaraven')
>>> db.close()  ❼
```

❶ shelve.open opens an existing or just-created database file.

❷ A quick way to determine if the database is populated is to look for a known key, in this case 'conference.115' — the key to the single conference record[7]

❸ If the database is empty, call load_db(db) to load it.

[7] I could also do len(db), but that would be costly in a large dbm database.

❹ Fetch a `speaker` record.

❺ It's an instance of the `Record` class defined in Example 19-9.

❻ Each `Record` instance implements a custom set of attributes reflecting the fields of the underlying JSON record.

❼ Always remember to close a `shelve.Shelf`. If possible, use a `with` block to make sure the `Shelf` is closed[8].

The code for `schedule1.py` is in Example 19-9.

Example 19-9. `schedule1.py`: exploring OSCON schedule data saved to a `shelve.Shelf`.

```
import warnings

import osconfeed  ❶

DB_NAME = 'data/schedule1_db'
CONFERENCE = 'conference.115'


class Record:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)  ❷


def load_db(db):
    raw_data = osconfeed.load()  ❸
    warnings.warn('loading ' + DB_NAME)
    for collection, rec_list in raw_data['Schedule'].items():  ❹
        record_type = collection[:-1]  ❺
        for record in rec_list:
            key = '{}.{}'.format(record_type, record['serial'])  ❻
            record['serial'] = key  ❼
            db[key] = Record(**record)  ❽
```

❶ Load the `osconfeed.py` module from Example 19-2.

❷ This is a common shortcut to build an instance with attributes created from keyword arguments (detailed explanation follows).

❸ This may fetch the JSON feed from the Web, if there's no local copy.

❹ Iterate over the collections (eg. `'conferences'`, `'events'` etc.).

❺ `record_type` is set to the collection name without the trailing `'s'`, i.e. `'events'` becomes `'event'`.

[8] A fundamental weakness of doctest is the lack of proper resource setup and guaranteed tear-down. I wrote most tests for `schedule1.py` using `py.test`, and you can see them at Example A-12

❻ Build key from the record_type and the 'serial' field.

❼ Update the 'serial' field with the full key.

❽ Build Record instance and save it to the database under the key.

The Record.__init__ method illustrates a popular Python hack. Recall that the __dict__ of an object is where its attributes are kept — unless __slots__ is declared in the class, as we saw in "Saving space with the __slots__ class attribute" on page 265. So, updating an instance __dict__ with a mapping is a quick way to create a bunch of attributes in that instance[9].

> I am not going to repeat here discussion in "The invalid attribute name problem" on page 593, but depending on the application context, the Record class may need to deal with keys that are not valid attribute names.

The definition of Record in Example 19-9 is so simple that you may be wondering why we did not use it before, instead of the more complicated FrozenJSON. There are a couple reasons. First, FrozenJSON works by recursively converting the nested mappings and lists; Record doesn't need that because our converted dataset doesn't have mappings nested in mappings or lists. The records contain only strings, integers, lists of strings and lists of integers. A second reason is that FrozenJSON provides access to the embedded __data dict attributes — which we used to invoke methods like keys — and now we don't need that functionality either.

> The Python standard library provides at least two classes similar to our Record, where each instance has an arbitrary set of attributes built from keyword arguments to the constructor: multiprocessing.Namespace (documentation, source code), and argparse.Namespace (documentation, source code). I implemented Record to highlight the essence of the idea: __init__ updating the instance __dict__.

After reorganizing the schedule data set as we just did, we can now extend the Record class to provide a useful service: automatically retrieving venue and speaker records referenced in an event record. This is similar to what the Django ORM does when you

[9] By the way, Bunch is the name of the class used by Alex Martelli to share this tip in a recipe from 2001 titled The simple but handy "collector of a bunch of named stuff" class.

access a models.ForeignKey field: instead of the key, you get the linked model object. We'll use properties to do that in the next example.

Linked record retrieval with properties

The goal of this next version is: given an event record retrieved from the shelf, reading its venue or speakers attributes will not return serial numbers but full-fledged record objects. See this partial interaction as an example:

Example 19-10. Extract from the doctests of schedule2.py.

```
>>> DbRecord.set_db(db)  ❶
>>> event = DbRecord.fetch('event.33950')  ❷
>>> event  ❸
<Event 'There *Will* Be Bugs'>
>>> event.venue  ❹
<DbRecord serial='venue.1449'>
>>> event.venue.name  ❺
'Portland 251'
>>> for spkr in event.speakers:  ❻
...     print('{0.serial}: {0.name}'.format(spkr))
...
speaker.3471: Anna Martelli Ravenscroft
speaker.5199: Alex Martelli
```

❶ DbRecord extends Record, adding database support: to operate, DbRecord must be given a reference to a database.

❷ The DbRecord.get class method retrieves records of any type.

❸ Note that event is an instance of the Event class, which extends DbRecord.

❹ Accessing event.venue returns a DbRecord instance.

❺ Now it's easy to find out the name of an event.venue. This automatic dereferencing is the goal of this example.

❻ We can also iterate over the event.speakers list, retrieving DbRecords representing each speaker.

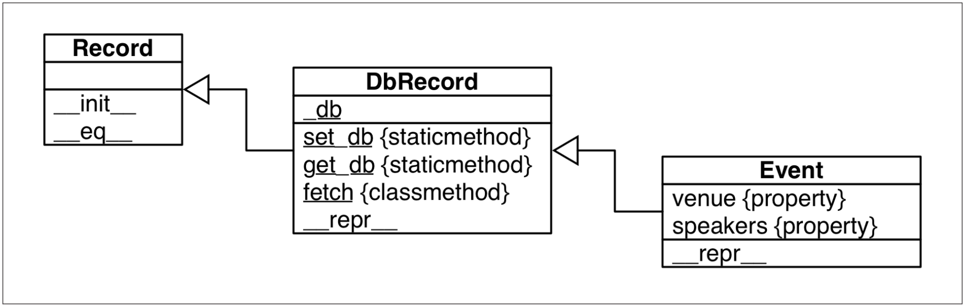


Figure 19-1. UML class diagram for an enhanced `Record` class and two subclasses: `DbRecord` and `Event`.

Figure 19-1 Provides an overview of the classes we'll be studying in this section.

`Record`
: The `__init__` method is the same as in `schedule1.py` (Example 19-9); the `__eq__` method was added to facilitate testing.

`DbRecord`
: Subclass of `Record` adding a `__db` class attribute, `set_db` and `get_db` static methods to set/get that attribute, a `fetch` class method to retrieve records from the database and a `__repr__` instance method to support debugging and testing.

`Event`
: Subclass of `DbRecord` adding `venue` and `speakers` properties to retrieve linked records, and a specialized `__repr__` method.

The `DbRecord.__db` class attribute exists to hold a reference to the opened `shelve.Shelf` database, so it can be used by the `DbRecord.fetch` method and the `Event.venue` and `Event.speakers` properties that depend in it. I coded `__db` as a private class attribute with conventional getter and setter methods because I wanted to protect it from accidental overwriting. I did not use a property to manage `__db` because of a crucial fact: properties are class attributes designed to manage instance attributes[10].

The code for this section is in the `schedule2.py` module in the Fluent Python example code repository. Because the module tops 100 lines, I'll present it in parts[11].

The first statements of `schedule2.py` are shown in Example 19-11.

[10] The StackOverflow topic Class-level read only properties in Python has solutions to read-only attributes in classes, including one by Alex Martelli. The solutions require metaclasses, so you may want to read Chapter 21 before studying them.

[11] The full listing for `schedule2.py` is in Example A-13, together with `py.test` scripts in "Chapter 19: OSCON schedule scripts and tests" on page 710.

Example 19-11. `schedule2.py`: imports, constants and the enhanced `Record` class.

```
import warnings
import inspect  ❶

import osconfeed

DB_NAME = 'data/schedule2_db'  ❷
CONFERENCE = 'conference.115'


class Record:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __eq__(self, other):  ❸
        if isinstance(other, Record):
            return self.__dict__ == other.__dict__
        else:
            return NotImplemented
```

❶ `inspect` will be used in the `load_db` function (Example 19-14).

❷ Since we are storing instances of different classes, we create and use a different database file, `'schedule2_db'`, instead of the `'schedule_db'` of Example 19-9.

❸ An `__eq__` method is always handy for testing.

In Python 2, only "new style" classes support properties. To write a new style class in Python 2 you must subclass directly or indirectly from `object`. `Record` in Example 19-11 is the base class of a hierarchy that will use properties, so in Python 2 its declaration would start with[12]:

```
class Record(object):
    # etc...
```

The next classes defined in `schedule2.py` are a custom exception type and `DbRecord`. See Example 19-12.

Example 19-12. `schedule2.py`: `MissingDatabaseError` and `DbRecord` class.

```
class MissingDatabaseError(RuntimeError):
    """Raised when a database is required but was not set."""  ❶
```

[12] Explicitly subclassing from `object` in Python 3 is not wrong, just redundant because all classes are new-style now. This is one example where breaking with the past made the language cleaner. If the same code must run in Python 2 and Python 3, inheriting from `object` should be explicit.

```
class DbRecord(Record):  ❷

    __db = None  ❸

    @staticmethod  ❹
    def set_db(db):
        DbRecord.__db = db  ❺

    @staticmethod  ❻
    def get_db():
        return DbRecord.__db

    @classmethod  ❼
    def fetch(cls, ident):
        db = cls.get_db()
        try:
            return db[ident]  ❽
        except TypeError:
            if db is None:  ❾
                msg = "database not set; call '{}.set_db(my_db)'"
                raise MissingDatabaseError(msg.format(cls.__name__))
            else:  ❿
                raise

    def __repr__(self):
        if hasattr(self, 'serial'):  ⓫
            cls_name = self.__class__.__name__
            return '<{} serial={!r}>'.format(cls_name, self.serial)
        else:
            return super().__repr__()  ⓬
```

❶ Custom exceptions are usually marker classes, with no body. A docstring explaining the usage of the exception is better than a mere `pass` statement.

❷ `DbRecord` extends `Record`.

❸ The `_db` class attribute will hold a reference to the opened `shelve.Shelf` database.

❹ `set_db` is a `staticmethod` to make it explicit that its effect is always exactly the same, no matter how it's called.

❺ Even if this method is invoked as `Event.set_db(my_db)`, the `_db` attribute will be set in the `DbRecord` class.

❻ `get_db` is also a `staticmethod` because it will always return the object referenced by `DbRecord._db`, no matter how it's invoked.

❼ `fetch` is a class method so that its behavior is easier to customize in subclasses.

❽ This retrieves the record with the `ident` key from the database.

❾ If we get a `TypeError` and `db` is `None`, raise a custom exception explaining that the database must be set.

❿ Otherwise, re-raise the exception because we don't know how to handle it.

⓫ If the record has a `serial` attribute, use it in the string representation.

⓬ Otherwise, default to the inherited `__repr__`.

Now we get to the meat of the example: the `Event` class, listed in Example 19-13.

Example 19-13. `schedule2.py`: The `Event` class.

```
class Event(DbRecord):  ❶

    @property
    def venue(self):
        key = 'venue.{}'.format(self.venue_serial)
        return self.__class__.fetch(key)  ❷

    @property
    def speakers(self):
        if not hasattr(self, '_speaker_objs'):  ❸
            spkr_serials = self.__dict__['speakers']  ❹
            fetch = self.__class__.fetch  ❺
            self._speaker_objs = [fetch('speaker.{}'.format(key))
                                  for key in spkr_serials]  ❻
        return self._speaker_objs  ❼

    def __repr__(self):
        if hasattr(self, 'name'):  ❽
            cls_name = self.__class__.__name__
            return '<{} {!r}>'.format(cls_name, self.name)
        else:
            return super().__repr__()  ❾
```

❶ `Event` extends `DbRecord`.

❷ The `venue` property builds a `key` from the `venue_serial` attribute, and passes it to the `fetch` class method, inherited from `DbRecord` (see explanation below).

❸ The `speakers` property checks if the record has a `_speaker_objs` attribute.

❹ If it doesn't, the `'speakers'` attribute is retrieved directly from the instance `__dict__` to avoid an infinite recursion, because the public name of this property is also `speakers`.

❺ Get a reference to the `fetch` class method (the reason for this will be explained shortly).

❻ `self._speaker_objs` is loaded with a list of `speaker` records, using `fetch`.

❼ That list is returned.

❽ If the record has a `name` attribute, use it in the string representation.

❾ Otherwise, default to the inherited `__repr__`.

In the `venue` property of Example 19-13, the last line returns `self.__class__.fetch(key)`. Why not write that simply as `self.fetch(key)`? The simpler formula works with the specific data set of the OSCON feed because there is no event record with a `'fetch'` key. If even a single event record had a key named `'fetch'`, then within that specific `Event` instance, the reference `self.fetch` would retrieve the value of that field, instead of the `fetch` class method that `Event` inherits from `DbRecord`. This is a subtle bug, and it could easily sneak through testing and blow up only in production when the venue or speaker records linked to that specific `Event` record are retrieved.

> When creating instance attribute names from data, there is always the risk of bugs due to shadowing of class attributes (such as methods) or data loss through accidental overwriting of existing instance attributes. This caveat is probably the main reason why, by default, Python dicts are not like JavaScript objects in the first place.

If the `Record` class behaved more like a mapping, implementing a dynamic `__getitem__` instead of a dynamic `__gettarr__`, there would be no risk of bugs from overwriting or shadowing. A custom mapping is probably the Pythonic way to implement `Record`. But if I took that road, we'd not be reflecting on the tricks and traps of dynamic attribute programming.

The final piece of this example is the revised `load_db` function in Example 19-14.

Example 19-14. `schedule2.py`: The `load_db` function.

```
def load_db(db):
    raw_data = osconfeed.load()
    warnings.warn('loading ' + DB_NAME)
    for collection, rec_list in raw_data['Schedule'].items():
        record_type = collection[:-1]  ❶
        cls_name = record_type.capitalize()  ❷
        cls = globals().get(cls_name, DbRecord)  ❸
        if inspect.isclass(cls) and issubclass(cls, DbRecord):  ❹
            factory = cls  ❺
        else:
            factory = DbRecord  ❻
        for record in rec_list:  ❼
            key = '{}.{}'.format(record_type, record['serial'])
            record['serial'] = key
            db[key] = factory(**record)  ❽
```

❶ So far, no changes from the `load_db` in `schedule1.py` (Example 19-9).

❷ Capitalize the `record_type` to get a potential class name; e.g. `'event'` becomes `'Event'`.

❸ Get an object by that name from the module global scope; get `DbRecord` if there's no such object.

❹ If the object just retrieved is a class, and is a subclass of `DbRecord`…

❺ …bind the `factory` name to it. This means `factory` may be any subclass of `DbRecord`, depending on the `record_type`.

❻ Otherwise, bind the `factory` name to `DbRecord`.

❼ The `for` loop which creates the `key` and saves the records is the same as before, except that…

❽ …the object stored in the database is constructed by `factory`, which may be `DbRecord` or a subclass selected according to the `record_type`.

Note that the only `record_type` which has a custom class is `Event`, but if classes named `Speaker` or `Venue` are coded, `load_db` will automatically use those classes when building and saving records, instead of the default `DbRecord` class.

So far, the examples in this chapter were designed to show a variety of techniques for implementing dynamic attributes using basic tools such as `__getattr__`, `hasattr`, `getattr`, `@property` and `__dict__`.

Properties are frequently used to enforce business rules by changing a public attribute into an attribute managed by a getter and setter without affecting client code, as the next section shows.

Using a property for attribute validation

So far we have only seen the `@property` decorator used to implement read-only properties. In this section we will create a read/write property.

`LineItem` take #1: class for an item in an order

Imagine an app for a store that sells organic food in bulk, where customers can order nuts, dried fruit or cereals by weight. In that system, each order would hold a sequence of line items, and each line item could be represented by a class as in Example 19-15.

Example 19-15. `bulkfood_v1.py`: the simplest `LineItem` class.

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

That's nice and simple. Perhaps too simple. Example 19-16 shows a problem.

Example 19-16. A negative weight results in a negative subtotal.

```
    >>> raisins = LineItem('Golden raisins', 10, 6.95)
    >>> raisins.subtotal()
    69.5
    >>> raisins.weight = -20  # garbage in...
    >>> raisins.subtotal()    # garbage out...
    -139.0
```

This is a toy example, but not as fanciful as you may think. Here is a true story from the early days of Amazon.com:

> We found that customers could order a negative quantity of books! And we would credit their credit card with the price and, I assume, wait around for them to ship the books[13].
>
> — Jeff Bezos
> *founder and CEO of Amazon.com*

How do we fix this? We could change the interface of `LineItem` to use a getter and a setter for the `weight` attribute. That would be the Java way, and it's not wrong.

On the other hand, it's natural to be able set the `weight` of an item by just assigning to it; and perhaps the system is in production with other parts already accessing `item.weight` directly. In this case, the Python way would be to replace the data attribute with a property.

`LineItem` take #2: a validating property

Implementing a property will allow us to use a getter and a setter, but the interface of `LineItem` will not change, i.e. setting the `weight` of a `LineItem` will still be written as `raisins.weight = 12`.

Example 19-17 lists the code for a read/write `weight` property.

Example 19-17. `bulkfood_v2.py`: a `LineItem` with a `weight` property.

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
```

[13] Direct quote by Jeff Bezos in the Wall Street Journal story Birth of a Salesman (October 15, 2011).

```
        self.weight = weight  ❶
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    @property  ❷
    def weight(self):  ❸
        return self.__weight  ❹

    @weight.setter  ❺
    def weight(self, value):
        if value > 0:
            self.__weight = value  ❻
        else:
            raise ValueError('value must be > 0')  ❼
```

❶ Here the property setter is already in use, making sure that no instances with negative `weight` can be created.

❷ `@property` decorates the getter method.

❸ The methods that implement a property all have the name of the public attribute: `weight`.

❹ The actual value is stored in a private attribute `__weight`.

❺ The decorated getter has a `.setter` attribute, which is also a decorator; this ties the getter and setter together.

❻ If the value is greater than zero, we set the private `__weight`.

❼ Otherwise, `ValueError` is raised.

Note how a `LineItem` with an invalid weight cannot be created now:

```
>>> walnuts = LineItem('walnuts', 0, 10.00)
Traceback (most recent call last):
    ...
ValueError: value must be > 0
```

Now we have protected `weight` from users providing negative values. Although buyers usually can't set the price of an item, a clerical error or a bug may create a `LineItem` with a negative `price`. To prevent that, we could also turn `price` into a property, but this would entail some repetition in our code.

Remember the Paul Graham quote from Chapter 14: "When I see patterns in my programs, I consider it a sign of trouble." The cure for repetition is abstraction. There are two ways to abstract away property definitions: using a property factory or a descriptor class. The descriptor class approach is more flexible, and we'll devote the entire Chapter 20 to it. Properties are in fact implemented as descriptor classes themselves. But here we will continue our exploration of properties by implementing a property factory as a function.

But before we can implement a property factory, we need to have a deeper understanding of properties.

A proper look at properties

Although often used as a decorator, the `property` built-in is actually a class. In Python, functions and classes are often interchangeable, because both are callable and there is no `new` operator for object instantiation, so invoking a constructor is no different than invoking a factory function. And both can be used as decorators, as long as they return a new callable which is a suitable replacement of the decorated function.

This is the full signature of the `property` constructor:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

All arguments are optional, and if a function is not provided for one of them, the corresponding operation is not allowed by the resulting property object.

The `property` type was added in Python 2.2, but the `@` decorator syntax appeared only in Python 2.4, so for a few years, properties were defined by passing the accessor functions as the first two arguments (we'll talk about the last two arguments later soon).

The "classic" syntax for defining properties without decorators is illustrated in Example 19-18.

Example 19-18. `bulkfood_v2b.py`: same as Example 19-17 but without using decorators.

```
class LineItem:

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price

    def get_weight(self):  ❶
        return self.__weight

    def set_weight(self, value):  ❷
        if value > 0:
            self.__weight = value
        else:
            raise ValueError('value must be > 0')
    weight = property(get_weight, set_weight)  ❸
```

❶ A plain getter.

❷ A plain setter.

❸ Build the `property` and assign it to a public class attribute.

The classic form is better than the decorator syntax in some situations; the code of the property factory we'll discuss shortly is one example. On the other hand, in a class body with many methods the decorators make it explicit which are the getters and setters, without depending on the convention of using `get` and `set` prefixes in their names.

The presence of a property in a class affects how attributes in instances of that class can be found in a way that may be surprising at first. The next section explains.

Properties override instance attributes

Properties are always class attributes, but they actually manage attribute access in the instances of the class.

In "Overriding class attributes" on page 268 we saw that when an instance and its class both have a data attribute by the same name, the instance attribute overrides, or shadows, the class attribute — at least when read through that instance. Example 19-19 illustrates this point.

Example 19-19. Instance attribute shadows class data attribute

```
>>> class Class:  # ❶
...     data = 'the class data attr'
...     @property
...     def prop(self):
...         return 'the prop value'
...
>>> obj = Class()
>>> vars(obj)  # ❷
{}
>>> obj.data  # ❸
'the class data attr'
>>> obj.data = 'bar' # ❹
>>> vars(obj)  # ❺
{'data': 'bar'}
>>> obj.data  # ❻
'bar'
>>> Class.data  # ❼
'the class data attr'
```

❶ Define `Class` with two class attributes: the `data` data attribute and the `prop` property.

❷ `vars` returns the `__dict__` of `obj`, showing it has no instance attributes.

❸ Reading from `obj.data` retrieves the value of `Class.data`.

❹ Writing to `obj.data` creates an instance attribute.

❺ Inspect the instance to see the instance attribute.

❻ Now reading from `obj.data` retrieves the value of the instance attribute. When read from the `obj` instance, the instance `data` shadows the class `data`.

❼ The `Class.data` attribute is intact.

Now, let's try to override the `prop` attribute on the `obj` instance. Resuming the previous console session we have Example 19-20

Example 19-20. Instance attribute does not shadow class property (continued from Example 19-19)

```
>>> Class.prop  # ❶
<property object at 0x1072b7408>
>>> obj.prop  # ❷
'the prop value'
>>> obj.prop = 'foo'  # ❸
Traceback (most recent call last):
  ...
AttributeError: can't set attribute
>>> obj.__dict__['prop'] = 'foo'  # ❹
>>> vars(obj)  # ❺
{'prop': 'foo', 'attr': 'bar'}
>>> obj.prop  # ❻
'the prop value'
>>> Class.prop = 'baz'  # ❼
>>> obj.prop  # ❽
'foo'
```

❶ Reading `prop` directly from `Class` retrieves the property object itself, without running its getter method.

❷ Reading `obj.prop` executes the property getter.

❸ Trying to set an instance `prop` attribute fails.

❹ Putting `'prop'` directly in the `obj.__dict__` works.

❺ We can see that `obj` now has two instance attributes: `attr` and `prop`.

❻ However, reading `obj.prop` still runs the property getter. The property is not shadowed by an instance attribute.

❼ Overwriting `Class.prop` destroys the property object.

❽ Now `obj.prop` retrieves the instance attribute. `Class.prop` is not a property anymore, so it no longer overrides `obj.prop`.

As a final demonstration, we'll add a new property to `Class`, and see it overriding an instance attribute. Example 19-21 picks up where Example 19-20 left off.

Example 19-21. New class property shadows existing instance attribute (continued from Example 19-20)

```
>>> obj.data  # ❶
'bar'
>>> Class.data  # ❷
'the class data attr'
>>> Class.data = property(lambda self: 'the "data" prop value')  # ❸
>>> obj.data  # ❹
'the "data" prop value'
>>> del Class.data  # ❺
>>> obj.data  # ❻
'bar'
```

❶ `obj.data` retrieves the instance `data` attribute.

❷ `Class.data` retrieves the class `data` attribute.

❸ Overwrite `Class.data` with a new property.

❹ `obj.data` is now shadowed by the `Class.data` property

❺ Delete the property.

❻ `obj.data` now reads the instance `data` attribute again.

The main point of this section is that an expression like `obj.attr` does not search for `attr` starting with `obj`. The search actually starts at `obj.__class__`, and only if there is no property named `attr` in the class, Python looks in the `obj` instance itself. This rule applies not only to properties but to a whole category of descriptors, the *overriding descriptors*. Further treatment of descriptors must wait for Chapter 20, where we'll see that properties are in fact overriding descriptors.

Now back to properties. Every Python code unit — modules, functions, classes, methods — can have a docstring. The next topic is how to attach documentation to properties.

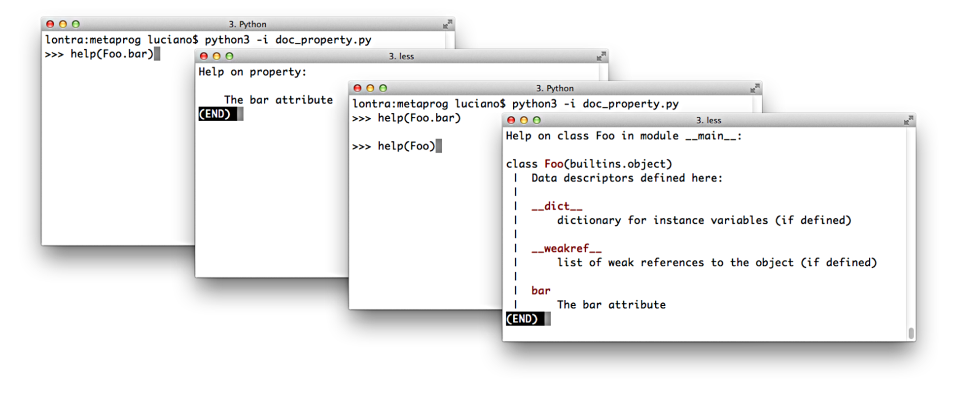
Property documentation

When tools such as the console `help()` function or IDEs need to display the documentation of a property, they extract the information from the `__doc__` attribute of the property.

If used with the classic call syntax, `property` can get the documentation string as the `doc` argument:

```
    weight = property(get_weight, set_weight, doc='weight in kilograms')
```

When `property` is deployed as a decorator, the docstring of the getter method — the one with the `@property` decorator itself — is used as the documentation of the property as a whole. Figure 19-2 shows the help screens generated from the code in Example 19-22.



```
lontra:metaprog luciano$ python3 -i doc_property.py
>>> help(Foo.bar)
```

```
Help on property:

    The bar attribute
(END)
```

```
lontra:metaprog luciano$ python3 -i doc_property.py
>>> help(Foo.bar)

>>> help(Foo)
```

```
Help on class Foo in module __main__:

class Foo(builtins.object)
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  bar
 |      The bar attribute
(END)
```

Figure 19-2. Screenshots of the Python console when issuing the commands `help(Foo.bar)` and `help(Foo)`. Source code in Example 19-22.

Example 19-22. Documentation for a property.

```
class Foo:

    @property
    def bar(self):
        '''The bar attribute'''
        return self.__dict__['bar']

    @bar.setter
    def bar(self, value):
        self.__dict__['bar'] = value
```

Now that we have these property essentials covered, let's go back to the issue of protecting both the `weight` and `price` attributes of `LineItem` so they only accept values greater than `0` — but without implementing two nearly identical pairs of getters/setters by hand.

Coding a property factory

We'll create a `quantity` property factory — so named because the managed attributes represent quantities that can't be negative or zero in the application. Example 19-23 shows the clean look of the `LineItem` class using two instances of `quantity` properties: one for managing the `weight` attribute, the other for `price`.

Example 19-23. `bulkfood_v2prop.py`: The `quantity` property factory in use.

```
class LineItem:
    weight = quantity('weight')  ❶
    price = quantity('price')  ❷

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight  ❸
        self.price = price

    def subtotal(self):
        return self.weight * self.price  ❹
```

❶ Use the factory to define the first custom property, `weight`, as a class attribute.

❷ This second call builds another custom property, `price`.

❸ Here the property is already active, making sure a negative or `0` `weight` is rejected.

❹ The properties are also in use here, retrieving the values stored in the instance.

Recall that properties are class attributes. When building each `quantity` property we need to pass the name of the `LineItem` attribute that will be managed by that specific property. Having to type the word `weight` twice in this line is unfortunate:

```
    weight = quantity('weight')
```

But avoiding that repetition is complicated because the property has no way of knowing which class attribute name will be bound to it. Remember: the right side of an assignment is evaluated first, so when `quantity()` is invoked, the `price` class attribute doesn't even exist.

> Improving the `quantity` property so that the user doesn't need to retype the attribute name is a nontrivial metaprogramming problem. We'll see a workaround in Chapter 20, but real solutions will have to wait until Chapter 21, as they require either a class decorator or a metaclass.

Example 19-24 lists the implementation of the `quantity` property factory[14].

Example 19-24. `bulkfood_v2prop.py`: The `quantity` property factory.

```
def quantity(storage_name):  ❶
```

[14] This code is adapted from recipe *9.21. Avoiding Repetitive Property Methods* from *The Python Cookbook, 3e.* (O'Reilly, 2013), by David Beazley and Brian K. Jones.

```
def qty_getter(instance):  ❷
    return instance.__dict__[storage_name]  ❸

def qty_setter(instance, value):  ❹
    if value > 0:
        instance.__dict__[storage_name] = value  ❺
    else:
        raise ValueError('value must be > 0')

return property(qty_getter, qty_setter)  ❻
```

❶ The `storage_name` argument determines where the data for each property is stored; for the `weight`, the storage name will be `'weight'`.

❷ The first argument of the `qty_getter` could be named `self`, but that would be strange since this is not a class body; `instance` refers to the `LineItem` instance where the attribute will be stored.

❸ `qty_getter` references `storage_name`, so it will be preserved in the closure of this function; the value is retrieved directly from the `instance.__dict__` to bypass the property and avoid an infinite recursion.

❹ `qty_setter` is defined, also taking `instance` as first argument.

❺ The `value` is stored directly in the `instance.__dict__`, again bypassing the property.

❻ Build a custom property object and return it.

The bits of Example 19-24 that deserve careful study revolve around the `storage_name` variable. When you code each property in the traditional way, the name of the attribute where you will store a value is hard coded in the getter and setter methods. But here, the `qty_getter` and `qty_setter` functions are generic, and they depend on the `storage_name` variable to know where to get/set the managed attribute in the instance `__dict__`. Each time the `quantity` factory is called to build a property, the `storage_name` must be set to a unique value.

The functions `qty_getter` and `qty_setter` will be wrapped by the `property` object created in the last line of the factory function. Later when called to perform their duties, these functions will read the `storage_name` from their closures, to determine where to retrieve/store the managed attribute values.

In Example 19-24 I create and inspect a `LineItem` instance, exposing the storage attributes.

Example 19-25. `bulkfood_v2prop.py`: The `quantity` property factory.

```
>>> nutmeg = LineItem('Moluccan nutmeg', 8, 13.95)
>>> nutmeg.weight, nutmeg.price  ❶
(8, 13.95)
```

```
>>> sorted(vars(nutmeg).items())  ❷
[('description', 'Moluccan nutmeg'), ('price', 13.95), ('weight', 8)]
```

❶ Reading the `weight` and `price` through the properties shadowing the namesake instance attributes.

❷ Using `vars` to inspect the `nutmeg` instance: here we see the actual instance attributes used to store the values.

Note how the properties built by our factory leverage the behavior described in "Properties override instance attributes" on page 610: the `weight` property overrides the `weight` instance attribute so that every reverence to `self.weight` or `nutmeg.weight` is handled by the property functions, and the only way to bypass the property logic is to access the instance `__dict__` directly.

The code in Example 19-24 may be a bit tricky, but it's concise: it has the same number of lines as a the decorated getter/setter pair defining just the `weight` property in Example 19-17. The `LineItem` definition in Example 19-23 looks much better without the noise of the getter/setters.

In a real system, that same kind of validation may appear in many fields, across several classes, and the `quantity` factory would be placed in a utility module to be used over and over again. Eventually that simple factory could be refactored into a more extensible descriptor class, with specialized subclasses performing different validations. We'll do that in Chapter 20.

Now let us wrap up the discussion of properties with the issue of attribute deletion.

Handling attribute deletion

Recall from the Python tutorial that object attributes can be deleted using the `del` statement:

```
del my_object.an_attribute
```

In practice, deleting attributes is not something we do every day in Python, and the requirement to handle it with a property is even more unusual. But it is supported, and I can think of a silly example to demonstrate it.

In a property definition, the `@my_propety.deleter` decorator is used to wrap the method in charge of deleting the attribute managed by the property. As promised, Example 19-26 is a silly example showing how to code a property deleter.

Example 19-26. `blackknight.py`: Inspired by the Black Knight character of "Monty Python and the Holy Grail"

```
class BlackKnight:
    def __init__(self):
        self.members = ['an arm', 'another arm',
                        'a leg', 'another leg']
        self.phrases = ["'Tis but a scratch.",
                        "It's just a flesh wound.",
                        "I'm invincible!",
                        "All right, we'll call it a draw."]

    @property
    def member(self):
        print('next member is:')
        return self.members[0]

    @member.deleter
    def member(self):
        text = 'BLACK KNIGHT (loses {})\n-- {}'
        print(text.format(self.members.pop(0), self.phrases.pop(0)))
```

The doctests `blackknight.py` are in Example 19-27.

Example 19-27. `blackknight.py`: Doctests for Example 19-26. The Black Knight never concedes defeat.

```
    >>> knight = BlackKnight()
    >>> knight.member
    next member is:
    'an arm'
    >>> del knight.member
    BLACK KNIGHT (loses an arm)
    -- 'Tis but a scratch.
    >>> del knight.member
    BLACK KNIGHT (loses another arm)
    -- It's just a flesh wound.
    >>> del knight.member
    BLACK KNIGHT (loses a leg)
    -- I'm invincible!
    >>> del knight.member
    BLACK KNIGHT (loses another leg)
    -- All right, we'll call it a draw.
```

Using the classic call syntax instead of decorators, the `fdel` argument is used to set the deleter function. For example, the `member` property would be coded like this in the body of the `BlackKnight` class:

```
        member = property(member_getter, fdel=member_deleter)
```

If you are not using a property, attribute deletion can also be handled by implementing the lower level `__delattr__` special method, presented in "Special methods for attribute handling" on page 619. Coding a silly class with `__delattr__` is left as an exercise to the procrastinating reader.

Properties are a powerful feature, but sometimes simpler or lower-level alternatives are preferable. In the last section of this chapter we'll review some the core APIs that Python offers for dynamic attribute programming.

Essential attributes and functions for attribute handling

Throughout this chapter, and even before in the book, we've used some of the built-in functions and special methods Python provides for dealing with dynamic attributes. This section gives an overview of them in one place, since their documentation is scattered in the official docs.

Special attributes that affect attribute handling

The behavior of many of the functions and special methods listed in the next sections depend on the following three special attributes.

`__class__`
: A reference to the object's class; i.e. `obj.__class__` is the same as `type(obj)`. Python looks for special methods such as `__getattr__` only in an object's class, and not in the instances themselves.

`__dict__`
: A mapping that stores the writable attributes of an object or class. An object that has a `__dict__` can have arbitrary new attributes set at any time. If a class has a `__slots__` attribute, then its instances may not have a `__dict__`. See `__slots__` below.

`__slots__`
: An attribute that may be defined in a class to limit the attributes its instances can have. `__slots__` is a `tuple` of strings naming the allowed attributes[15]. If the `'__dict__'` name is not in `__slots__` then the instances of that class will not have a `__dict__` of their own, and only the named attributes will be allowed in them.

Built-in functions for attribute handling

These five built-in functions perform object attribute reading, writing and introspection.

[15] Alex Martelli points out that, although `__slots__` can be coded as a `list`, it's better to be explicit and always use a `tuple`, because changing the list in the `__slots__` after the class body is processed has no effect, so it would be misleading to use a mutable sequence there.

`dir([object])`

Lists most attributes of the object. The official docs say dir is intended for interactive use so it does not provide a comprehensive list of attributes, but an “interesting” set of names. `dir` can inspect objects implemented with or without a `__dict__`. The `__dict__` attribute itself is not listed by `dir`, but the `__dict__` keys are listed. Several special attributes of classes, such as `__mro__`, `__bases__` and `__name__` are not listed by `dir` either. If the optional `object` argument is not given, `dir` lists the names in the current scope.

`getattr(object, name[, default])`

Gets the attribute identified by the `name` string from the `object`. This may fetch an attribute from the object’s class or from a superclass. If no such attribute exists, `getattr` raises `AttributeError` or returns the `default` value, if given.

`hasattr(object, name)`

Returns `True` if the named attribute exists in the `object`, or can be somehow fetched through it (by inheritance, for example). The documentation explains: “This is implemented by calling getattr(object, name) and seeing whether it raises an AttributeError or not.”

`setattr(object, name, value)`

Assigns the `value` to the named attribute of `object`, if the `object` allows it. This may create a new attribute or overwrite an existing one.

`vars([object])`

Returns the `__dict__` of `object`; `vars` can’t deal with instances of classes that define `__slots__` and don’t have a `__dict__` (contrast with `dir`, which handles such instances). Without an argument, `vars()` does the same as `locals()`: returns a `dict` representing the local scope.

Special methods for attribute handling

When implemented in a user defined class, the special methods listed here handle attribute retrieval, setting, deletion and listing.

Attribute access using either dot notation or the built-in functions `gettattr`, `hasattr` and `setattr` trigger the appropriate special methods listed here. Reading and writing attributes directly in the instance `__dict__` does not trigger these special methods — and that’s the usual way to bypass them if needed.

Section 3.3.9. Special method lookup of the *Data model* chapter warns:

> For custom classes, implicit invocations of special methods are only guaranteed to work correctly if defined on an object’s type, not in the object’s instance dictionary.

In other words, assume that the special methods will be retrieved on the class itself, even when the target of the action is an instance. For this reason, special methods are not shadowed by instance attributes with the same name.

In the examples below, assume there is a class named `Class`, `obj` is an instance of `Class`, and `attr` is an attribute of `obj`.

For every one of these special methods it doesn't matter if the attribute access is done using dot notation or one of the built-in functions listed in "Built-in functions for attribute handling" on page 618. For example, both `obj.attr` and `getattr(obj, 'attr', 42)` trigger `Class.__getattribute__(obj, 'attr')`.

`__delattr__(self, name)`

Always called when there is an attempt to delete an attribute using the `del` statement; e.g. `del obj.attr` triggers `Class.__delattr__(obj, 'attr')`.

`__dir__(self)`

Called when `dir` is invoked on the object, to provide a listing of attributes; e.g. `dir(obj)` triggers `Class.__dir__(obj)`.

`__getattr__(self, name)`

Called only when an attempt to retrieve the named attribute fails, after the `obj`, `Class` and its superclasses are searched. The expressions `obj.no_such_attr`, `getattr(obj, 'no_such_attr')` and `hasattr(obj, 'no_such_attr')` may trigger `Class.__getattr__(obj, 'no_such_attr')`, but only if an attribute by that name cannot be found in `obj` or in `Class` and its superclasses.

`__getattribute__(self, name)`

Always called when there is an attempt to retrieve the named attribute, except when the attribute sought is a special attribute or method. Dot notation and the `getattr` and `hasattr` built-ins trigger this method. `__getattr__` is only invoked after `__gatattribute__`, and only when `__gatattribute__` raises `AttributeError`. To retrieve attributes of the instance `obj` without triggering an infinite recursion, implementations of `__getattribute__` should use `super().__getattribute__(obj, name)`.

`__setattr__(self, name, value)`

Always called when there is an attempt to set the named attribute. Dot notation and the `settattr` built-in trigger this method; e.g. both `obj.attr = 42` and `setattr(obj, 'attr', 42)` trigger `Class.__setattr__(obj, 'attr', 42)`.

> In practice, because they are unconditionally called and affect practically every attribute access, the `__getattribute__` and `__setattr__` special methods are harder to use correctly than `__getattr__` — which only handles non-existing attribute names. Using properties or descriptors is less error prone than defining these special methods.

This concludes our dive into properties, special methods and other techniques for coding dynamic attributes.

Chapter summary

We started our coverage of dynamic attributes by showing practical examples of simple classes to make it easier to deal with a JSON data feed. The first example was the FrozenJSON` class that converted nested dicts and lists into nested `FrozenJSON` instances and lists of them. The `FrozenJSON` code demonstrated the use of the `__getattr__` special method to convert data structures on the fly, whenever their attributes were read. The last version of `FrozenJSON` showcased the use of the `__new__` constructor method to transform a class into a flexible factory of objects, not limited to instances of itself.

We then converted the JSON feed to a `shelve.Shelf` database storing serialized instances of a `Record` class. The first rendition of `Record` was three lines long and introduced the "bunch" idiom: using `self.__dict__.update(**kwargs)` to build arbitrary attributes from keyword arguments passed to `__init__`. The second iteration of this example saw the extension of `Record` with a `DbRecord` class for database integration and an `Event` class implementing automatic retrieval of linked records through properties.

Coverage of properties continued with the `LineItem` class where a property was deployed to protect a `weight` attribute from negative or zero values that make no business sense. After a deeper look at property syntax and semantics, we created a property factory to enforce the same validation on `weight` and `price`, without coding multiple getters and setters. The property factory leveraged subtle concepts --such as closures and the instance attribute overriding by properties — to provide an elegant generic solution using the same number of lines as a single hand-coded property definition.

Finally we had a brief look at handling attribute deletion with properties, followed by an overview of the key special attributes, built-in functions and special methods that support attribute metaprogramming in the core Python language.

Further reading

The official documentation for the attribute handling and introspection built-in functions is Chapter 2. Built-in Functions of *The Python Standard Library*. The related special methods and the `__slots__` special attribute are documented in Section 3.3.2. Customizing attribute access of *Chapter 3. Data model* in *The Python Language Reference*. The semantics of how special methods are invoked bypassing instances is explained in Section 3.3.9. Special method lookup of that same page. Section 4.13. Special Attributes of *Chaper 4. Built-in Types* in the *The Python Standard Library* covers `__class__` and `__dict__` attributes.

The Python Cookbook, 3e. (O'Reilly, 2013), by David Beazley and Brian K. Jones, has several recipes covering the topics of this chapter, but I will highlight three that are outstanding: *8.8. Extending a Property in a Subclass* addresses the thorny issue of overriding the methods inside a property inherited from a superclass; *8.15. Delegating Attribute Access* implements a proxy class showcasing most special methods from "Special methods for attribute handling" on page 619 in this book; and the awesome recipe *9.21. Avoiding Repetitive Property Methods* which was the basis for the property factory function presented in Example 19-24.

Python in a Nutshell, 2nd Edition (O'Reilly, 2003), by Alex Martelli, covers only Python 2.5 but the fundamentals still apply to Python 3 and his treatment is rigorous and objective. Martelli devotes only three pages to properties, but that's because the book follows an axiomatic presentation style: the previous 15 pages or so provide a thorough description of the semantics of Python classes from the ground up, including descriptors which are how properties are actually implemented under the hood. So by the time he gets to properties, he can pack a lot of insights in those three pages — including that which I selected to open this chapter.

Bertrand Meyer, quoted in the *Uniform Access Principle* definition in this chapter opening, wrote the excellent *Object-Oriented Software Construction, 2e.* (Prentice-Hall, 1997). The book is more than 1250 pages long, and I confess I did not read it all, but the first 6 chapters provide one of the best conceptual introductions to OO analysis and design I've seen, chapter 11 introduces Design by Contract (Meyer invented the method and coined the term) and chapter 35 offers his assessments of some key OO languages: Simula, Smalltalk, CLOS (the Lisp OO extension), Objective-C, C++ and Java, with brief comments on some others. Meyer is also the inventor of the pseudo-pseudo-code: only in the last page of the book he reveals that the "notation" he uses throughout as pseudocode is in fact Eiffel.

Soapbox

Meyer's *Uniform Access Principle* (sometimes called UAP by acronym-lovers) is aesthetically appealing. As a programmer using an API I shouldn't have to care whether `coconut.price` simply fetches a data attribute or performs a computation. As a consumer and a citizen I do care: in e-commerce today the value of `coconut.price` often depends on who is asking, so it's certainly not a mere data attribute. In fact, it's common practice that the price is lower if the query comes from outside the store — say, from a price-comparison engine. This effectively punishes loyal customers who like to browse within a particular store. But I digress.

The previous digression does raise a relevant point for programming: although the Uniform Access Principle makes perfect sense in an ideal world, in reality users of an API may need to know whether reading `coconut.price` is potentially too expensive or time consuming. As usual in matters of software engineering, Ward Cunningham's original Wiki hosts insightful arguments about the merits of the Uniform Access Principle.

In object oriented programming languages, application or violations of the Uniform Access Principle usually revolve around the syntax of reading public data attributes versus invoking getter/setter methods.

Smalltalk and Ruby address this issue in a simple and elegant way: they don't support public data attributes at all. Every instance attribute in these languages is private, so every access to them must be through methods. But their syntax makes this painless: in Ruby `coconut.price` invokes the `price` getter; in Smalltalk it's simply `coconut price`.

At the other end of the spectrum, the Java language allows the programmer to choose among four access level modifiers[16]. The general practice does not agree with the syntax established by the Java designers, though. Everybody in Java-land agrees that attributes should be `private`, and you must spell it out every time, since it's not the default. When all attributes are private, all access to them from outside the class must go through accessors. Java IDEs include shortcuts for generating accessor methods automatically. Unfortunately, the IDE is not so helpful when you must read the code six months later. It's up to you to wade through a sea of do-nothing accessors to find those that add value by implementing some business logic.

Alex Martelli speaks for the majority of the Python community when he calls accessors "goofy idioms" and then provides these examples that look very different but do the same thing[17]:

```
someInstance.widgetCounter += 1
# rather than...
someInstance.setWidgetCounter(someInstance.getWidgetCounter() + 1)
```

Sometimes when designing APIs I've wondered whether every method that does not take an argument (besides `self`), returns a value (other than `None`) and is a pure function (i.e. has no side-effects) should be replaced by a read-only property. In this chapter, the `LineItem.subtotal` method (as in Example 19-23) would be a good candidate to become a read-only property. Of course, this excludes methods that are designed to change the object, such as `my_list.clear()`. It would be a terrible idea to turn that in to a property, so that merely acessing `my_list.clear` would delete the contents of the list!

[16] Including the no-name default that the Java Tutorial calls "package-private".

[17] Alex Martelli, *Python in a Nutshell, 2e.* (O'Reilly, 2006), p. 101.

In the Pingo.io GPIO library (mentioned in "The `__missing__` method" on page 72), much of the user level API is based on properties. For example, to read the current value of an analog pin the user writes `pin.value`, and setting a digital pin mode is spelled as `pin.mode = OUT`. Behind the scenes, reading an analog pin value or setting a digital pin mode may involve a lot of code, depending on the specific board driver. We decided to use properties in Pingo because we want the API to be comfortable to use even in interactive environments like iPython Notebook, and we feel `pin.mode = OUT` is easier on the eyes and on the fingers than `pin.set_mode(OUT)`.

Although I find the Smalltalk and Ruby solution cleaner, I think the Python approach makes more sense than the Java one. We are allowed to start simple, coding data members as public attributes, because we know they can always be wrapped by properties (or descriptors, which we'll talk about in the next chapter).

`__new__` is better than `new`

Another example of the Uniform Access Principle (or a variation of it) is the fact that function calls and object instantiation use the same syntax in Python: `my_obj = foo()`, where `foo` may be a class or any other callable.

Other languages influenced by C++ syntax have a `new` operator that makes instantiation look different than a call. Most of the time the user of an API doesn't care whether `foo` is a function or a class. Until recently I was under the impression that `property` was a function. In normal usage it makes no difference.

There are many good reasons for replacing constructors with factories[18] and in *Consider static factory methods instead of constructors* which is Item 1 of the award-winning book *Effective Java* (Addison-Wesley, 2008) by Joshua Bloch.]. A popular motive is limiting the number of instances, by returning previously built ones (as in the Singleton pattern). A related use is caching expensive object construction. Also, sometimes it's convenient to return objects of different types depending on the arguments given.

Coding a constructor is simpler; providing a factory adds flexibility at the expense of more code. In languages that have a `new` operator, the designer of an API must decide in advance whether to stick with a simple constructor or invest in factory. If the initial choice is wrong the correction may be costly — all because `new` is an operator.

Sometimes it may also be convenient to go the other way, and replace a simple function with a class.

In Python, classes and functions are interchangeable in many situations. Not only because there's no `new` operator, but also because there is the `__new__` special method, which can turn a class into a factory producing objects of different kinds (as we saw in

[18] The reasons I am about to mention are given in the Dr. Dobbs Journal article titled Java's new Considered Harmful, by Jonathan Amsterdam

"Flexible object creation with `__new__`" on page 594) or returning pre-built instances instead of creating a new one every time.

This function-class duality would be easier to leverage if PEP 8 — Style Guide for Python Code did not recommend `CamelCase` for class names. On the other hand, dozens of classes in the standard library have lowercase names — e.g. `property`, `str`, `defaultdict` etc. So maybe the use of lowercase class names is a feature, and not a bug. But however we look at it, the inconsistent capitalization of classes in the Python standard library poses a usability problem.

Although calling a function is not different than calling a class, it's good to know which is which because of another thing we can do with a class: subclassing. So I personally use `CamelCase` in every class that I code, and I wish all classes in the Python standard library used the same convention. I am looking at you, `collections.OrderedDict` and `collections.defaultdict`.

CHAPTER 20
Attribute descriptors

> Learning about descriptors not only provides access to a larger toolset, it creates a deeper understanding of how Python works and an appreciation for the elegance of its design[1]
>
> — Raymond Hettinger
> *Python core developer and guru*

Descriptors are a way of reusing the same access logic in multiple attributes. For example, field types in ORMs such as the Django ORM and SQL Alchemy are descriptors, managing the flow of data from the fields in a database record to Python object attributes and vice-versa.

A descriptor is a class which implements a protocol consisting of the `__get__`, `__set__` and `__delete__` methods. The `property` class implements the full descriptor protocol. As usual with protocols, partial implementations are OK. In fact, most descriptors we see in real code implement only `__get__` and `__set__`, and many implement only one of these methods.

Descriptors are a distinguishing feature of Python, deployed not only at the application level but also in the language infrastructure. Besides properties, other Python features that leverage descriptors are methods and the `classmethod` and `staticmethod` decorators. Understanding descriptors is key to Python mastery. This is what this chapter is about.

Descriptor example: attribute validation

As we saw in "Coding a property factory" on page 613, a property factory is a way to avoid repetitive coding of getters and setters by applying functional programming patterns. A property factory is a higher-order function that creates a parametrized set of

[1] Reymond Hettinger, Descriptor HowTo Guide.

accessor functions and builds a custom property instance from them, with closures to hold settings like the storage_name. The object oriented way of solving the same problem is a descriptor class.

We'll continue the series of LineItem examples where we left it, in "Coding a property factory" on page 613, by refactoring the quantity property factory into a Quantity descriptor class.

LineItem take #3: a simple descriptor

A class implementing a __get__, a __set__ or a __delete__ method is a descriptor. You use a descriptor by declaring instances of it as class attributes of another class.

We'll create a Quantity descriptor and the LineItem class will use two instances of Quantity: one for managing the weight attribute, the other for price. A diagram helps, so take a look at Figure 20-1.

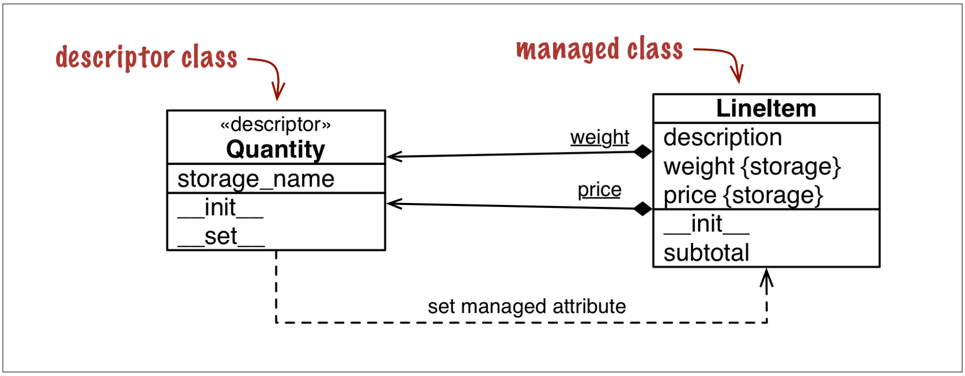


Figure 20-1. UML class diagram for LineItem using a descriptor class named Quantity. Underlined attributes in UML are class attributes. Note that weight and price are instances of Quantity attached to the LineItem class, but LineItem instances also have their own weight and price attributes where those values are stored.

Note that the word weight appears twice in Figure 20-1, because there are really two distinct attributes named weight: one is a class attribute of LineItem, the other is an instance attribute that will exist in each LineItem object. This also applies to price.

From now on, I will use the following definitions:

descriptor class
A class implementing the descriptor protocol. That's Quantity in Figure 20-1.

managed class

The class where the descriptor instances are declared as class attributes — `LineItem` in Figure 20-1.

descriptor instance

Each instance of a descriptor class, declared as a class attribute of the managed class. In Figure 20-1 each descriptor instance is represented by a composition arrow with an underlined name (the underline means class attribute in UML). The black diamonds touch the `LineItem` class, which contains the descriptor instances.

managed instance

One instance of the managed class. In this example, `LineItem` instances will be the managed instances (they are not shown in the class diagram).

storage attribute

An attribute of the managed instance which will hold the value of a managed attribute for that particular instance. In Figure 20-1, the `LineItem` instance attributes `weight` and `price` will be the storage attributes. They are distinct from the descriptor instances, which are always class attributes.

managed attribute

A public attribute in the managed class that will be handled by a descriptor instance, with values stored in storage attributes. In other words, a descriptor instance and a storage attribute provide the infrastructure for a managed attribute.

It's important to realize that `Quantity` instances are class attributes of `LineItem`. This crucial point is highlighted by the mills and gizmos in Figure 20-2.

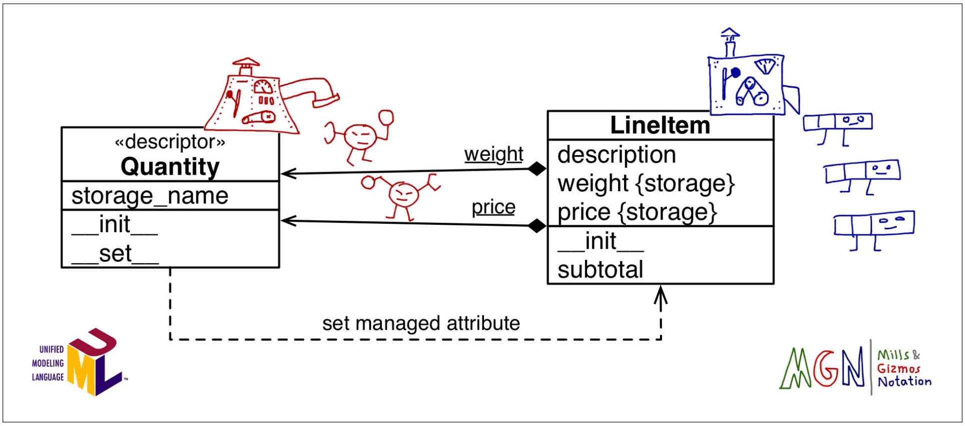


Figure 20-2. UML class diagram annotated with MGN (Mills & Gizmos Notation): classes are mills that produce gizmos — the instances. The `Quantity` mill produces two red gizmos which are attached to the `LineItem` mill: `weight` and `price`. The `LineItem`

mill produces blue gizmos that have their own `weight` and `price` attributes where those values are stored.

Introducing MGN: Mills & Gizmos Notation

After explaining descriptors many times I realized UML is not very good at showing relationships involving classes and instances, like the relationship between a managed class and the descriptor instances[2]. So I invented my own "language", the MGN (Mills & Gizmos Notation), which I use to annotate UML diagrams.

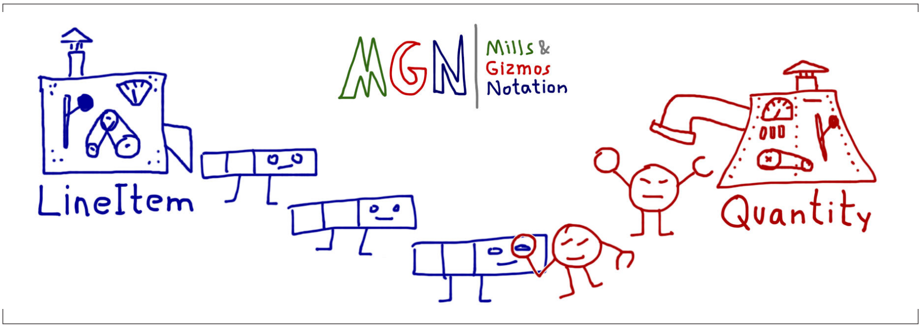


Figure 20-3. MGN sketch showing the `LineItem` class making three instances, and `Quantity` making two. One instance of `Quantity` is retrieving a value stored in a `LineItem` instance.

MGN is designed to make very clear the distinction between classes and instances. See Figure 20-3. In MGN, a class is drawn as a "mill", a complicated machine that produces gizmos. Classes/mills are always machines with levers and dials. The gizmos are the instances, and they look much simpler. A gizmo is the same color as the mill which made it.

For this example, I drew `LineItem` instances as rows in a tabular invoice, with three cells representing the three attributes (`description`, `weight` and `price`). Since `Quantity` instances are descriptors, they have a magnifying glass to `__get__` values and a claw to `__set__` values. When we get to metaclasses, you'll thank me for these doodles.

Enough doodling for now. Here is the code: Example 20-1 shows the `Quantity` descriptor class and a new `LineItem` class using two instances of `Quantity`.

[2] Classes and instances are drawn as rectangles in UML class diagrams. There are visual differences, but instances are rarely shown in class diagrams, so developers may not recognize them as such.

Example 20-1. `bulkfood_v3.py`: `Quantity` descriptors manage attributes in `LineItem`.

```
class Quantity:  ❶

    def __init__(self, storage_name):
        self.storage_name = storage_name  ❷

    def __set__(self, instance, value):  ❸
        if value > 0:
            instance.__dict__[self.storage_name] = value  ❹
        else:
            raise ValueError('value must be > 0')


class LineItem:
    weight = Quantity('weight')  ❺
    price = Quantity('price')  ❻

    def __init__(self, description, weight, price):  ❼
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ Descriptor is a protocol-based feature; no subclassing is needed to implement one.

❷ Each `Quantity` instance will have a `storage_name` attribute: that's the name of the attribute which will hold the value in the managed instances.

❸ `__set__` is called when there is an attempt to assign to the managed attribute. Here `self` is the descriptor instance (i.e. `LineItem.weight` or `LineItem.price`), `instance` is the managed instance (a `LineItem` instance), and `value` is the value being assigned.

❹ Here we must handle the managed instance `__dict__` directly; trying to use the `setattr` built-in would trigger the `__set__` method again, leading to infinite recursion.

❺ The first descriptor instance is bound to the `weight` attribute.

❻ The second descriptor instance is bound to the `price` attribute.

❼ The rest of the class body is as simple and clean as the original code in `bulkfood_v1.py` (Example 19-15).

In Example 20-1, each managed attribute has the same name as its storage attribute, and there is no special getter logic, so `Quantity` doesn't need a `__get__` method.

The code in Example 20-1 works as intended, preventing the sale of truffles for $0[3]:

```
>>> truffle = LineItem('White truffle', 100, 0)
Traceback (most recent call last):
    ...
ValueError: value must be > 0
```

When coding a `__set__` method you must keep in mind what the `self` and `instance` arguments mean: `self` is the descriptor instance, and `instance` is the managed instance. Descriptors managing instance attributes should store values in the managed instances. That's why Python provides the `instance` argument to the descriptor methods.

It may be tempting, but wrong, to store the value of each managed attribute in the descriptor instance itself. In other words, in the `__set__` method, instead of coding:

```
    instance.__dict__[self.storage_name] = value
```

the tempting but bad alternative would be:

```
    self.__dict__[self.storage_name] = value
```

To understand why this would be wrong, think about the meaning of the first two arguments to `__set__`: `self` and `instance`. Here, `self` is the descriptor instance, which is actually a class attribute of the managed class. You may have thousands of `LineItem` instances in memory at one time, but you'll only have two instances of the descriptors: `LineItem.weight` and `LineItem.price`. So anything you store in the descriptor instances themselves is actually part of a `LineItem` class attribute, therefore it is shared among all `LineItem` instances.

A drawback of Example 20-1 is the need to repeat the names of the attributes when the descriptors are instantiated in the managed class body. It would be nice if the `LineItem` class could be declared like this:

```
class LineItem:
    weight = Quantity()
    price = Quantity()

    # remaining methods as before
```

The problem is that — as we saw in Chapter 8 — the right-hand side of an assignment is executed before the variable exists. The expression `Quantity()` is evaluated to create a descriptor instance, and at this time there is no way the code in the `Quantity` class can

[3] White truffles cost thousands of dollars per pound. Disallowing the sale of truffles for $0.01 is left as an exercise for the enterprising reader. I know a person who actually bought a $1,800 encyclopedia of statistics for $18 because of an error in an online store (not Amazon.com).

guess the name of the variable to which the descriptor will be bound (eg. `weight` or `price`).

As it stands, Example 20-1 requires naming each `Quantity` explicitly which is not only inconvenient but dangerous: if a programmer copy and pasting code forgets to edit both names and writes something like `price = Quantity('weight')` the program will misbehave badly, clobbering the value of `weight` whenever the `price` is set.

A not-so-elegant but workable solution to the repeated name problem is presented next. Better solutions require either a class decorator or a metaclass, so I'll leave them for Chapter 21.

`LineItem` take #4: automatic storage attribute names

To avoid retyping the attribute name in the descriptor declarations, we'll generate a unique string for the `storage_name` of each `Quantity` instance.

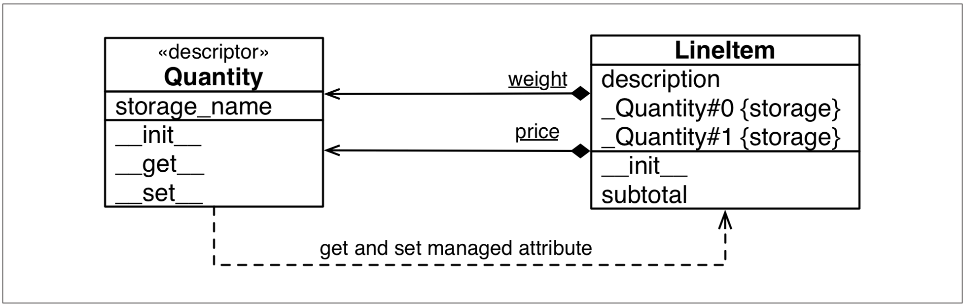


Figure 20-4. UML class diagram for Example 20-2. Now `Quantity` has both `__get__` and `__set__` methods, and `LineItem` instances have storage attributes with generated names: `_Quantity#0` and `_Quantity#1`.

To generate the `storage_name` we start with a `'_Quantity#'` prefix and concatenate an integer: the current value of a `Quantity.__counter` class attribute which we'll increment every time a new `Quantity` descriptor instance is attached to a class. Using the hash character in the prefix guarantees the `storage_name` will not clash with attributes created by the user using dot notation, since `nutmeg._Quantity#0` is not valid Python syntax. But we can always get and set attributes with such "invalid" identifiers using the `getattr` and `setattr` built-in functions, or by poking the instance `__dict__`.

Example 20-2. `bulkfood_v4.py`: Each `Quantity` descriptor gets a unique `storage_name`

```
class Quantity:
    __counter = 0  ❶
    def __init__(self):
        cls = self.__class__  ❷
        prefix = cls.__name__
        index = cls.__counter
        self.storage_name = '_{}#{}'.format(prefix, index)  ❸
        cls.__counter += 1  ❹

    def __get__(self, instance, owner):  ❺
        return getattr(instance, self.storage_name)  ❻

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.storage_name, value)  ❼
        else:
            raise ValueError('value must be > 0')


class LineItem:
    weight = Quantity()  ❽
    price = Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ `__counter` is a class attribute of `Quantity`, counting the number of `Quantity` instances.

❷ `cls` is a reference to the `Quantity` class.

❸ The `storage_name` for each descriptor instance is unique because it's built from the descriptor class name and the current `__counter` value, e.g. `_Quantity#0`.

❹ Increment `__counter`.

❺ We need to implement `__get__` because the name of the managed attribute is not the same as the `storage_name`. The `owner` argument will be explained shortly.

❻ Use the `getattr` built-in function to retrieve the value from the `instance`.

❼ Use the `setattr` built-in to store the value in the `instance`.

❽ Now we don't need to pass the managed attribute name to the `Quantity` constructor. That was the goal for this version.

Here we can use the higher-level `getattr` and `setattr` built-ins to store the value — instead of resorting to `instance.__dict__` — because the managed attribute and the storage attribute have different names so calling gettatr on the storage attribute will not trigger the descriptor, avoiding the infinite recursion discussed in Example 20-1.

If you test bulkfood_v4.py you can see that the weight and price descriptors work as expected, and the storage attributes can also be read directly, which is useful for debugging:

```
>>> from bulkfood_v4 import LineItem
>>> coconuts = LineItem('Brazilian coconut', 20, 17.95)
>>> coconuts.weight, coconuts.price
(20, 17.95)
>>> getattr(raisins, '_Quantity#0'), getattr(raisins, '_Quantity#1')
(20, 17.95)
```

If we wanted to follow the convention Python uses to do name mangling — e.g. _LineItem__quantity0 — we'd have to know the name of the managed class, i.e. LineItem, but the body of a class definition runs before the class itself is built by the interpreter, so we don't have that information when each descriptor instance is created. However, in this case there is no need to include the managed class name to avoid accidental overwriting in subclasses: the descriptor class __counter will be incremented every time a new descriptor is instantiated, guaranteeing that each storage name will be unique across all managed classes.

Note that __get__ receives three arguments. self, instance and owner. The owner argument is a reference to the managed class (e.g. LineItem), and it's handy when the descriptor is used to get attributes from the class. If a managed attribute, such as weight, is retrieved via the class like LineItem.weight, the descriptor __get__ method receives None as the value for the instance argument. This explains the Attribute error in the next console session:

```
>>> from bulkfood_v4 import LineItem
>>> LineItem.weight
Traceback (most recent call last):
  ...
  File ".../descriptors/bulkfood_v4.py", line 54, in __get__
    return getattr(instance, self.storage_name)
AttributeError: 'NoneType' object has no attribute '_Quantity#0'
```

Raising AttributeError is an option when implementing __get__, but if you choose to do so, the message should be fixed to remove the confusing mention of NoneType and _Quantity#0 which are implementation details. A better message would be "'LineItem' class has no such attribute". Ideally the name of the missing attribute should be spelled out, but the descriptor doesn't know the name of the managed attribute in this example, so we can't do better at this point.

On the other hand, to support introspection and other metaprogramming tricks by the user, it's a good practice to make `__get__` return the descriptor instance when the managed attribute is accessed through the class. Example 20-3 is a minor variation of Example 20-2, adding a bit of logic to `Quantity.__get__`.

Example 20-3. `bulkfood_v4b.py` (partial listing): When invoked through the managed class, `__get__` returns a reference to the descriptor itself.

```
class Quantity:
    __counter = 0

    def __init__(self):
        cls = self.__class__
        prefix = cls.__name__
        index = cls.__counter
        self.storage_name = '_{}#{}'.format(prefix, index)
        cls.__counter += 1

    def __get__(self, instance, owner):
        if instance is None:
            return self   ❶
        else:
            return getattr(instance, self.storage_name)   ❷

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.storage_name, value)
        else:
            raise ValueError('value must be > 0')
```

❶ If the call was not through an instance, return the descriptor itself.

❷ Otherwise, return the managed attribute value, as usual.

Trying out Example 20-3, this is what we see:

```
>>> from bulkfood_v4b import LineItem
>>> LineItem.price
<bulkfood_v4b.Quantity object at 0x100721be0>
>>> br_nuts = LineItem('Brazil nuts', 10, 34.95)
>>> br_nuts.price
34.95
```

Looking at Example 20-2 you may think that's a lot of code just for managing a couple of attributes, but it's important to realize that the descriptor logic is now abstracted into a separate code unit: the `Quantity` class. Usually we do not define a descriptor in the same module where it's used, but in a separate utility module designed to be used across the application — even in many applications, if you are developing a framework.

With this in mind, Example 20-4 better represents the typical usage of a descriptor.

Example 20-4. `bulkfood_v4c.py`: `LineItem` definition uncluttered. The `Quantity` descriptor class now resides in the imported `model_v4c` module.

```
import model_v4c as model  ❶


class LineItem:
    weight = model.Quantity()  ❷
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ Import the `model_v4c` module, giving it a friendlier name.

❷ Put `model.Quantity` to use.

Django users will notice that Example 20-4 looks a lot like a model definition. It's no coincidence: Django model fields are descriptors.

As implemented so far, the `Quantity` descriptor works pretty well. Its only real drawback is the use of generated storage names like `_Quantity#0`, making debugging hard for the users. But automatically assigning storage names that resemble the managed attribute names requires a class decorator or a metaclass, topics we'll defer to Chapter 21.

Because descriptors are defined in classes, we can leverage inheritance to reuse some of the code we have for new descriptors. That's what we'll do in the following section.

Property factory versus descriptor class

It's not hard to re-implement the enhanced descriptor class of Example 20-2 by adding a few lines to the property factory shown in Example 19-24. The `__counter` variable presents a difficulty, but we can make it persist across invocations of the factory by defining it as an attribute of factory function object itself, as shown in Example 20-5:

Example 20-5. `bulkfood_v4prop.py`: same functionality as Example 20-2 with a property factory instead of a descriptor class.

```
def quantity():  ❶
    try:
        quantity.counter += 1  ❷
    except AttributeError:
        quantity.counter = 0  ❸

    storage_name = '_{}:{}'.format('quantity', quantity.counter)  ❹

    def qty_getter(instance):  ❺
        return getattr(instance, storage_name)

    def qty_setter(instance, value):
        if value > 0:
            setattr(instance, storage_name, value)
        else:
            raise ValueError('value must be > 0')

    return property(qty_getter, qty_setter)
```

❶ No `storage_name` argument.

❷ We can't rely on class attributes to share the `counter` across invocations, so we define it as an attribute of the `quantity` function itself.

❸ If `quantity.counter` is undefined, set it to `0`.

❹ We also don't have instance attributes, so we create `storage_name` as a local variable and rely on closures to keep them alive for later use by `qty_getter` and `qty_setter`.

❺ The rest of the code is the same as Example 19-24 except that here we can use the `getattr` and `setattr` built-ins instead of fiddling with `instance.__dict__`.

So, which do you prefer? Example 20-2 or Example 20-5?

I prefer the descriptor class approach mainly for two reasons:

1. A descriptor class can be extended by subclassing; reusing code from a factory function without copying & pasting is much harder.
2. It's more straightforward to hold state in class and instance attributes than in function attributes and closures as we had to do in Example 20-5

On the other hand, when I explain Example 20-5 I don't feel the urge to draw mills and gizmos. The property factory code does not depend on strange object relationships evidenced by descriptor methods having arguments named `self` and `instance`.

To summarize, the property factory pattern is simpler in some regards, but the descriptor class approach is more extensible. It's also more widely used.

LineItem take #5: a new descriptor type

The imaginary organic food store hits a snag: somehow a line item instance was created with a blank description and the order could not fulfilled. To prevent that, we'll create a new descriptor NonBlank. As we design NonBlank, we realize it will be very much like the Quantity descriptor, except for the validation logic.

Reflecting on the functionality of Quantity, we note it does two different things: it takes care of the storage attributes in the managed instances, and it validates the value used to set those attribute. This prompts a refactoring, producing two base classes:

AutoStorage
: Descriptor class which manages storage attributes automatically.

Validated
: AutoStorage abstract subclass that overrides the __set__ method, calling a validate method which must be implemented by subclasses.

We'll then rewrite Quantity and implement NonBlank by inheriting from Validated and just coding the validate methods. Figure 20-5 depicts the setup.

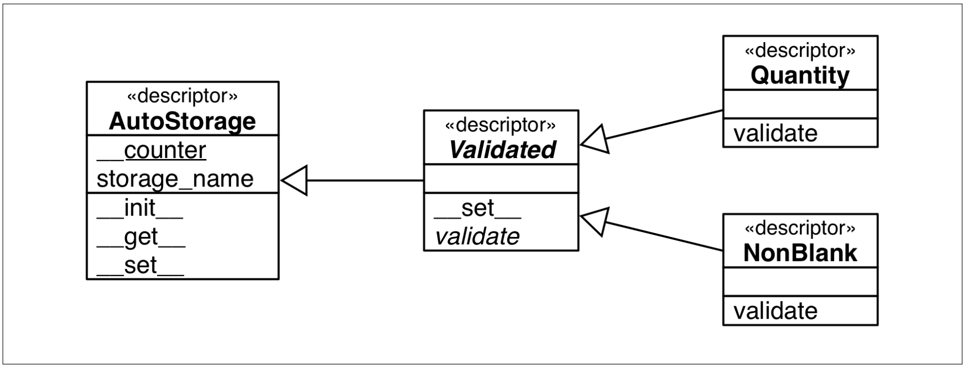


Figure 20-5. A hierarchy of descriptor classes. The AutoStorage base class manages the automatic storage of the attribute, Validated handles validation by delegating to an abstract validate method, Quantity and NonBlank are concrete subclasses of Validated.

The relationship between Validated, Quantity and NonBlank is an application of the Template Method design pattern. In particular, the Validated.__set__ is a clear example of what the Gang of Four describe as a template method:

> A template method defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior[4].

In this case, the abstract operation is validation. Example 20-6 lists the implementation of the classes in Figure 20-5.

Example 20-6. `model_v5.py`: The refactored descriptor classes.

```
import abc


class AutoStorage:  ❶
    __counter = 0

    def __init__(self):
        cls = self.__class__
        prefix = cls.__name__
        index = cls.__counter
        self.storage_name = '_{}#{}'.format(prefix, index)
        cls.__counter += 1

    def __get__(self, instance, owner):
        if instance is None:
            return self
        else:
            return getattr(instance, self.storage_name)

    def __set__(self, instance, value):
        setattr(instance, self.storage_name, value)  ❷


class Validated(abc.ABC, AutoStorage):  ❸

    def __set__(self, instance, value):
        value = self.validate(instance, value)  ❹
        super().__set__(instance, value)  ❺

    @abc.abstractmethod
    def validate(self, instance, value):  ❻
        """return validated value or raise ValueError"""


class Quantity(Validated):  ❼
    """a number greater than zero"""

    def validate(self, instance, value):
        if value <= 0:
            raise ValueError('value must be > 0')
        return value
```

[4] Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 326.

```
class NonBlank(Validated):
    """a string with at least one non-space character"""

    def validate(self, instance, value):
        value = value.strip()
        if len(value) == 0:
            raise ValueError('value cannot be empty or blank')
        return value  ❽
```

❶ `AutoStorage` provides most of the functionality of the former `Quantity` descriptor…

❷ …except validation.

❸ `Validated` is abstract but also inherits from `AutoStorage`.

❹ `__set__` delegates validation to a `validate` method…

❺ …then uses the returned `value` to invoke `__set__` on a superclass, which performs the actual storage.

❻ In this class, `validate` is an abstract method.

❼ `Quantity` and `NonBlank` inherit from `Validated`.

❽ Requiring the concrete `validate` methods to return the validated `value` gives them an opportunity to clean up, convert or normalize the data received. In this case, the `value` is returned stripped of leading and trailing blanks.

Users of `model_v5.py` don't need to know all these details. What matters is that they get to use `Quantity` and `NonBlank` to automate the validation of instance attributes. See the latest `LineItem` class in Example 20-7.

Example 20-7. `bulkfood_v5.py`: `LineItem` using `Quantity` and `NonBlank` descriptors

```
import model_v5 as model  ❶


class LineItem:
    description = model.NonBlank()  ❷
    weight = model.Quantity()
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ Import the `model_v5` module, giving it a friendlier name.

❷ Put `model.NonBlank` to use. The rest of the code is unchanged.

The `LineItem` examples we've seen in this chapter demonstrate a typical use of descriptors to manage data attributes. Such a descriptor is also called an overriding descriptor because its `__set__` method overrides — i.e. interrupts and overrules — the setting of an attribute by the same name in the managed instance. However, there are also non-overriding descriptors. We'll explore this distinction in the detail in the next section.

Overriding versus non-overriding descriptors

Recall that there is an important asymmetry in the way Python handles attributes. Reading an attribute through an instance normally returns the attribute defined in the instance, but if there is no such attribute in the instance, a class attribute will be retrieved. On the other hand, assigning to an attribute in an instance normally creates the attribute in the instance, without affecting the class at all.

This asymmetry also affects descriptors, in effect creating two broad categories of descriptors depending on whether the `__set__` method is defined. Observing the different behaviors requires a few classes, so we are going to use the code in Example 20-8 as our testbed for the next sections.

> Every `__get__` and `__set__` method in Example 20-8 calls `print_args` so their invocations are displayed in a readable way. Understanding `print_args` and the auxiliary functions `cls_name` and `display` is not important, so don't get distracted by them.

Example 20-8. `descriptorkinds.py`: Simple classes for studying descriptor overriding behaviors.

```
### auxiliary functions for display only ###

def cls_name(obj_or_cls):
    cls = type(obj_or_cls)
    if cls is type:
        cls = obj_or_cls
    return cls.__name__.split('.')[-1]

def display(obj):
    cls = type(obj)
    if cls is type:
        return '<class {}>'.format(obj.__name__)
    elif cls in [type(None), int]:
        return repr(obj)
    else:
        return '<{} object>'.format(cls_name(obj))

def print_args(name, *args):
    pseudo_args = ', '.join(display(x) for x in args)
    print('-> {}.__{}__({})'.format(cls_name(args[0]), name, pseudo_args))


### essential classes for this example ###

class Overriding:  ❶
    """a.k.a. data descriptor or enforced descriptor"""

    def __get__(self, instance, owner):
        print_args('get', self, instance, owner)  ❷

    def __set__(self, instance, value):
        print_args('set', self, instance, value)


class OverridingNoGet:  ❸
    """an overriding descriptor without ``__get__``"""

    def __set__(self, instance, value):
        print_args('set', self, instance, value)


class NonOverriding:  ❹
    """a.k.a. non-data or shadowable descriptor"""

    def __get__(self, instance, owner):
        print_args('get', self, instance, owner)


class Managed:  ❺
    over = Overriding()
    over_no_get = OverridingNoGet()
    non_over = NonOverriding()

    def spam(self):  ❻
        print('-> Managed.spam({})'.format(display(self)))
```

❶ A typical overriding descriptor class with `__get__` and `__set__`.

❷ The `print_args` function is called by every descriptor method in this example.

❸ An overriding descriptor without `__get__` method.

❹ No `__set__` method here, so this is a non-overriding descriptor.

❺ The managed class, using one instance of each of the descriptor classes.

❻ The `spam` method is here for comparison, since methods are also descriptors.

In the next sections we will examine the behavior of attribute reads and writes on the `Managed` class and one instance of it, going through each of the different descriptors defined.

Overriding descriptor

A descriptor that implements the `__set__` method it called an *overriding descriptor*, because although it is a class attribute, a descriptor implementing `__set__` will override attempts to assign to instance attributes. This is how Example 20-2 was implemented. Properties are also overriding descriptors: if you don't provide a setter function, the default `__set__` from the `property` class will raise `AttributeError` to signal that the attribute is read-only. Given the code in Example 20-8, experiments with an overriding descriptors can be seen in Example 20-9:

Example 20-9. Behavior of an overriding descriptor: `obj.over` is an instance of `Overriding` (Example 20-8)

```
>>> obj = Managed()  ❶
>>> obj.over  ❷
-> Overriding.__get__(<Overriding object>, <Managed object>, <class Managed>)
>>> Managed.over  ❸
-> Overriding.__get__(<Overriding object>, None, <class Managed>)
>>> obj.over = 7  ❹
-> Overriding.__set__(<Overriding object>, <Managed object>, 7)
>>> obj.over  ❺
-> Overriding.__get__(<Overriding object>, <Managed object>, <class Managed>)
>>> obj.__dict__['over'] = 8  ❻
>>> vars(obj)  ❼
{'over': 8}
>>> obj.over  ❽
-> Overriding.__get__(<Overriding object>, <Managed object>, <class Managed>)
```

❶ create `Managed` object for testing.

❷ `obj.over` triggers the descriptor `__get__` method, passing the managed instance `obj` as the second argument.

❸ `Managed.over` triggers the descriptor `__get__` method, passing `None` as the second argument (`instance`).

❹ Assigning to `obj.over` triggers the descriptor `__set__` method, passing the value `7` as the last argument.

❺ Reading `obj.over` still invokes the descriptor `__get__` method.

❻ Bypassing the descriptor, setting a value directly to the `obj.__dict__`

❼ Verify that the value is in the `obj.__dict__`, under the `over` key.

❽ However, even with an instance attribute named `over`, the `Managed.over` descriptor still overrides attempts to read `obj.over`.

Overriding descriptor without `__get__`

Usually, overriding descriptors implement both `__set__` and `__get__`, but it's also possible to implement only `__set__`, as we saw in Example 20-1. In this case, only writing is handled by the descriptor. Reading the descriptor through an instance will return the descriptor object itself because there is no `__get__` to handle that access. If a namesake instance attribute is created with a new value via direct access to the instance `__dict__`, the `__set__` method will still override further attempts to set that attribute, but reading that attribute will simply return the new value from the instance, instead of returning the descriptor object. In other words, the instance attribute will shadow the descriptor, but only when reading. See Example 20-10.

Example 20-10. Overriding descriptor without `__get__`: `obj.over_no_get` is an instance of `OverridingNoGet` (Example 20-8)

```
>>> obj.over_no_get   ❶
<__main__.OverridingNoGet object at 0x665bcc>
>>> Managed.over_no_get   ❷
<__main__.OverridingNoGet object at 0x665bcc>
>>> obj.over_no_get = 7   ❸
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
>>> obj.over_no_get   ❹
<__main__.OverridingNoGet object at 0x665bcc>
>>> obj.__dict__['over_no_get'] = 9   ❺
>>> obj.over_no_get   ❻
9
>>> obj.over_no_get = 7   ❼
-> OverridingNoGet.__set__(<OverridingNoGet object>, <Managed object>, 7)
>>> obj.over_no_get   ❽
9
```

❶ This overriding descriptor doesn't have a `__get__` method, so reading `obj.over_no_get` retrieves the descriptor instance from the class.

❷ The same thing happens if we retrieve the descriptor instance directly from the managed class.

❸ Trying to set a value to `obj.over_no_get` invokes the `__set__` descriptor method.

❹ Since our `__set__` doesn't make changes, reading `obj.over_no_get` again retrieves the descriptor instance from the managed class.

❺ Going through the instance `__dict__` to set an instance attribute named `over_no_get`.

❻ Now that `over_no_get` instance attribute shadows the descriptor, but only for reading.

❼ Trying to assign a value to `obj.over_no_get` still goes through the descriptor set.

❽ But for reading, that descriptor is shadowed as long as there is a namesake instance attribute.

Non-overriding descriptor

If a descriptor does not implement `__set__`, then it's a non-overriding descriptor. Setting an instance attribute with the same name will shadow the descriptor, rendering it ineffective for handling that attribute in that specific instance. Methods are implemented as non-overriding descriptors. Example 20-11 shows the operation of a non-overriding descriptor.

Example 20-11. Behavior of a non-overriding descriptor: `obj.non_over` is an instance of `NonOverriding` (Example 20-8)

```
>>> obj = Managed()
>>> obj.non_over   ❶
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>, <class Managed>)
>>> obj.non_over = 7   ❷
>>> obj.non_over   ❸
7
>>> Managed.non_over   ❹
-> NonOverriding.__get__(<NonOverriding object>, None, <class Managed>)
>>> del obj.non_over   ❺
>>> obj.non_over   ❻
-> NonOverriding.__get__(<NonOverriding object>, <Managed object>, <class Managed>)
```

❶ `obj.non_over` triggers the descriptor `__get__` method, passing `obj` as the second argument.

❷ `Managed.non_over` is a non-overriding descriptor, so there is no `__set__` to interfere with this assignment.

❸ The `obj` now has an instance attribute named `non_over` which shadows the namesake descriptor attribute in the `Managed` class.

❹ The `Managed.non_over` descriptor is still there, and catches this access via the class.

❺ If the `non_over` instance attribute is deleted…

❻ Then reading `obj.non_over` hits the `__get__` method of the descriptor in the class, but note that the second argument is the managed instance.

Python contributors and authors use different terms when discussing these concepts. Overriding descriptors are also called data descriptors or enforced descriptors. Non-overriding descriptors are also known as non-data descriptors or shadowable descriptors.

In the previous examples we saw several assignments to an instance attribute with the same name as a descriptor, and different results according to the presence of a `__set__` method in the descriptor.

The setting of attributes in the class cannot be controlled by descriptors attached to the same class. In particular, this means that the descriptor attributes themselves can be clobbered by assigning to the class, as the next section explains.

Overwriting a descriptor in the class

Regardless of whether a descriptor is overriding or not, it can be overwritten by assignment to the class. This is a monkey patching technique, but in Example 20-12 the descriptors are replaced by integers, which would effectively break any class that depended on the descriptors for proper operation.

Example 20-12. Any descriptor can be overwritten on the class itself.

```
>>> obj = Managed()   ❶
>>> Managed.over = 1   ❷
>>> Managed.over_no_get = 2
>>> Managed.non_over = 3
>>> obj.over, obj.over_no_get, obj.non_over   ❸
(1, 2, 3)
```

❶ Create a new instance for later testing.

❷ Overwrite the descriptor attributes in the class.

❸ The descriptors are really gone.

Example 20-12 reveals another asymmetry regarding reading and writing attributes: although the reading of a class attribute can be controlled by a descriptor with `__get__` attached to the managed class, the writing of a class attribute cannot be handled by a descriptor with `__set__` attached to the same class.

In order to control the setting of attributes in a class, you have to attach descriptors to the class of the class, in other words, the metaclass. By default, the metaclass of user defined classes is `type`, and you cannot add attributes to `type`. But in Chapter 21 we'll create our own metaclasses.

Let's now focus on how descriptors are used to implement methods in Python.

Methods are descriptors

A function within a class becomes a bound method because all user-defined functions have a `__get__` method, therefore they operate as descriptors when attached to a class. Example 20-13 demonstrates reading the `spam` method from the `Managed` class introduced in Example 20-8.

Example 20-13. A method is a non-overriding descriptor

```
>>> obj = Managed()
>>> obj.spam  ❶
<bound method Managed.spam of <descriptorkinds.Managed object at 0x74c80c>>
>>> Managed.spam  ❷
<function Managed.spam at 0x734734>
>>> obj.spam = 7  ❸
>>> obj.spam
7
```

❶ Reading from `obj.spam` retrieves a bound method object.

❷ But reading from `Managed.spam` retrieves a function.

❸ Assigning a value to `obj.spam` shadows the class attribute, rendering the `spam` method inaccessible from the `obj` instance.

Since functions do not implement `__set__` they are non-overriding descriptors, as the last line of Example 20-13 shows.

The other key takeaway from Example 20-13 is that `obj.spam` and `Managed.spam` retrieve different objects. As usual with descriptors, the `__get__` of a function returns a reference to itself when the access happens through the managed class. But when the access goes through an instance, the `__get__` of the function returns a bound method object: a callable which wraps the function and binds the managed instance (e.g. `obj`) to the first argument of the function (i.e. `self`), like the `functools.partial` function does (as seen in "Freezing arguments with `functools.partial`" on page 159).

For a deeper understanding of this mechanism, take a look at Example 20-14.

Example 20-14. `method_is_descriptor.py`: a `Text` class, derived from `UserString`.

```
import collections


class Text(collections.UserString):

    def __repr__(self):
        return 'Text({!r})'.format(self.data)
    def reverse(self):
        return self[::-1]
```

Now let's investigate the `Text.reverse` method. See Example 20-15.

Example 20-15. Experiments with a method.

```
>>> word = Text('forward')
>>> word  ❶
Text('forward')
>>> word.reverse()  ❷
Text('drawrof')
>>> Text.reverse(Text('backward'))  ❸
Text('drawkcab')
>>> type(Text.reverse), type(word.reverse)  ❹
(<class 'function'>, <class 'method'>)
>>> list(map(Text.reverse, ['repaid', (10, 20, 30), Text('stressed')]))  ❺
['diaper', (30, 20, 10), Text('desserts')]
>>> Text.reverse.__get__(word)  ❻
<bound method Text.reverse of Text('forward')>
>>> Text.reverse.__get__(None, Text)  ❼
<function Text.reverse at 0x101244e18>
>>> word.reverse  ❽
<bound method Text.reverse of Text('forward')>
>>> word.reverse.__self__  ❾
Text('forward')
>>> word.reverse.__func__ is Text.reverse  ❿
True
```

❶ The `repr` of a `Text` instance looks like a `Text` constructor call that would make an equal instance.

❷ The `reverse` method returns the text spelled backwards.

❸ A method called on the class works as a function.

❹ Note the different types: a `function` and a `method`.

❺ `Text.reverse` operates as a function, even working with objects that are not instances of `Text`.

❻ Any function is a non-overriding descriptor. Calling its `__get__` with an instance retrieves a method bound to that instance.

❼ Calling the function's `__get__` with `None` as the `instance` argument retrieves the function itself.

❽ The expression `word.reverse` actually invokes `Text.reverse.__get__(word)`, returning the bound method.

❾ The bound method object has a `__self__` attribute holding a reference to the instance on which the method was called.

⑩ The `__func__` attribute of the bound method is a reference to the original function attached to the managed class.

The bound method object also has a `__call__` method, which handles the actual invocation. This method calls the original function referenced in `__func__`, passing the `__self__` attribute of the method as the first argument. That's how the implicit binding of the conventional `self` argument works.

The way functions are turned into bound methods is a prime example of how descriptors are used as infrastructure in the language.

After this deep dive into how descriptors and methods work, let's go through some practical advice about their use.

Descriptor usage tips

The next short sections address some practical consequences of the descriptor characteristics just described.

1. Use `property` to keep it simple

The `property` built-in actually creates overriding descriptors implementing both `__set__` and `__get__`, even if you do not define a setter method. The default `__set__` of a property raises `AttributeError: can't set attribute`, so a property is the easiest way to create a read-only attribute, avoiding the issue described next.

2. Read-only descriptors require `__set__`

If you use a descriptor class to implement a read-only attribute you must remember to code both `__get__` and `__set__`, otherwise setting a namesake attribute on an instance will shadow the descriptor. The `__set__` method of a read-only attribute should just raise `AttributeError` with a suitable message [5].

3. Validation descriptors can work with `__set__` only

In a descriptor designed only for validation, the `__set__` method should check the `value` argument it gets, and if valid, set it directly in the instance `__dict__` using the descriptor instance name as key. That way, reading the attribute with the same name from the

[5] Python is not consistent in such messages. Trying to change the `c.real` attribute of a `complex` number gets `AttributeError: read-only attribute` but an attempt to change `c.conjugate` (a method of `complex`), results in `AttributeError: 'complex' object attribute 'conjugate' is read-only`.

instance will be as fast as possible, as it will not require a `__get__`. See the code for Example 20-1.

4. Caching can be done efficiently with `__get__` only

If you code just the `__get__` method you have a non-overriding descriptor. These are useful to make some expensive computation and then cache the result by setting an attribute by the same name on the instance. The namesake instance attribute will shadow the descriptor, so subsequent access to that attribute will fetch it directly from the instance `__dict__` and not trigger the descriptor `__get__` anymore.

5. Non-special methods can be shadowed by instance attributes

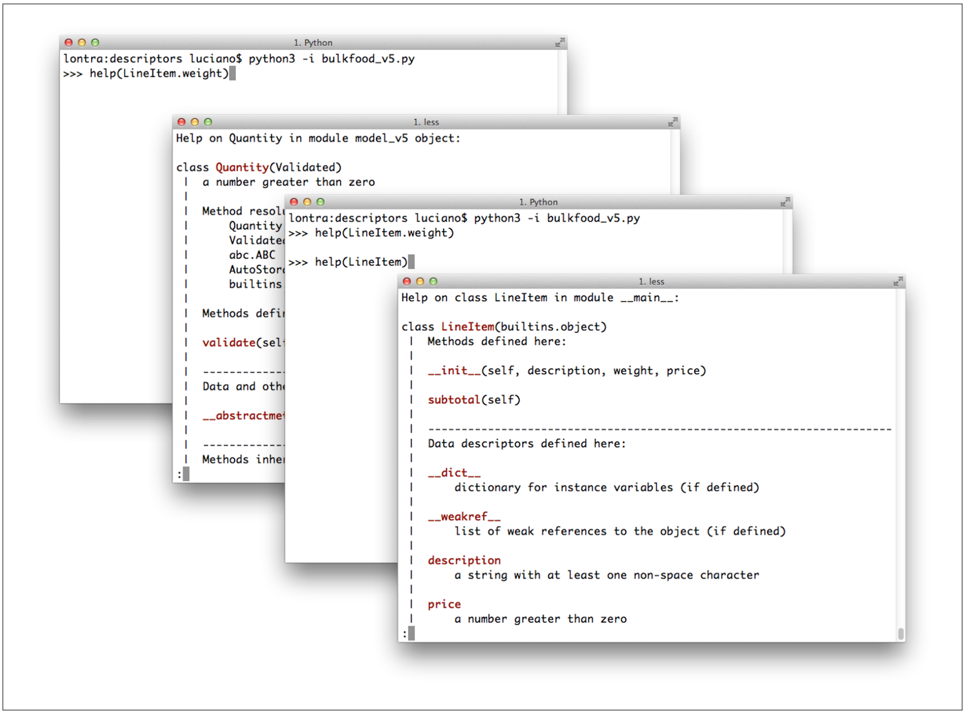
Because functions and methods only implement `__get__`, they do not handle attempts at setting instance attributes with the same name, so a simple assignment like `my_obj.the_method = 7` means that further access to `the_method` through that instance will retrieve the number `7` — without affecting the class or other instances. However, this issue does not interfere with special methods. The interpreter only looks for special methods in the class itself, in other words, `repr(x)` is executed as `x.__class__.__repr__(x)`, so a `__repr__` attribute defined in `x` has no effect of `repr(x)`. For the same reason, the existence of an attribute named `__getattr__` in an instance will not subvert the usual attribute access algorithm.

The fact that non-special methods can be overridden so easily in instances may sound fragile and error-prone, but I personally have never been bitten by this in more than 15 years of Python coding. On the other hand, if you are doing a lot of dynamic attribute creation, where the attribute names come from data you don't control (as we did in the earlier parts of this chapter), then you should be aware of this and perhaps implement some filtering or escaping of the dynamic attribute names to preserve your sanity.

> The `FrozenJSON` class in Example 19-6 is safe from instance attribute shadowing methods because its only methods are special methods and the `build` class method. Class methods are safe as long as they are always accessed through the class, as I did with `FrozenJSON.build` in Example 19-6 — later replaced by `__new__` in Example 19-7. The `Record` class (Example 19-9 and Example 19-11) and subclasses are also safe: they use only special methods, class methods, static methods and properties. Properties are data descriptors, so cannot be overridden by instance attributes.

To close this chapter we'll cover two features we saw with properties that we have not addressed in the context of descriptors: documentation and handling attempts to delete a managed attribute.

Descriptor docstring and overriding deletion



```
lontra:descriptors luciano$ python3 -i bulkfood_v5.py
>>> help(LineItem.weight)
```

```
Help on Quantity in module model_v5 object:

class Quantity(Validated)
 |  a number greater than zero
 |
 |  Method resolu
 |      Quantity
 |      Validated
 |      abc.ABC
 |      AutoStor
 |      builtins
 |
 |  Methods defi
 |
 |  validate(sel
 |
 |  ------------
 |  Data and othe
 |
 |  __abstractme
 |
 |  ------------
 |  Methods inher
:
```

```
lontra:descriptors luciano$ python3 -i bulkfood_v5.py
>>> help(LineItem.weight)

>>> help(LineItem)
```

```
Help on class LineItem in module __main__:

class LineItem(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self, description, weight, price)
 |
 |  subtotal(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  description
 |      a string with at least one non-space character
 |
 |  price
 |      a number greater than zero
:
```

Figure 20-6. Screenshots of the Python console when issuing the commands `help(LineItem.weight)` and `help(LineItem)`.

The docstring of a descriptor class is used to document every instance of the descriptor in the managed class. See Figure 20-6 for the help displays for the `LineItem` class with the `Quantity` and `NonBlank` descriptors from Example 20-6 and Example 20-7.

That is somewhat unsatisfactory. In the case of `LineItem`, it would be good to add, for example, the information that `weight` must be in kilograms. That would be trivial with properties, because each property handles a specific managed attribute. But with descriptors, the same `Quantity` descriptor class is used for `weight` and `price` [6].

The second detail we discussed with properties but have not addressed with descriptors is handling attempts to delete a managed attribute. That can be done by implementing a `__delete__` method alongside or instead of the usual `__get__` and/or `__set__` in the

[6] Customizing the help text for each descriptor instance is surprisingly hard. One solution requires dynamically building a wrapper class for each descriptor instance.

descriptor class. Coding a silly descriptor class with `__delete__` is left as an exercise to the leisurely reader.

Chapter summary

The first example of this chapter was a continuation of the `LineItem` examples from Chapter 19. In Example 20-1 we replaced properties with descriptors. We saw that a descriptor is a class that provides instances which are deployed as attributes in the managed class. Discussing this mechanism required special terminology, introducing terms such as managed instance and storage attribute.

In "`LineItem` take #4: automatic storage attribute names" on page 633 we removed the requirement that `Quantity` descriptors were declared with an explicit `storage_name` which was redundant and error-prone, as that name should always match the attribute name on the left of the assignment in the descriptor instantiation. The solution was to generate unique `storage_names` by combining the descriptor class name with a counter at the class level, e.g. `'_Quantity#1'`.

Next we compared the code size, strengths and weaknesses of a descriptor class with a property factory built on functional programming idioms. The latter works perfectly well and is simpler in some ways, but the former is more flexible and is the standard solution. A key advantage of the descriptor class was exploited in "`LineItem` take #5: a new descriptor type" on page 639: subclassing to share code while building specialized descriptors with some common functionality.

We then looked at the different behavior of descriptors providing or omitting the `__set__` method, making the crucial distinction between overriding and non-overriding descriptors. Through detailed testing we uncovered when descriptors are in control and when they are shadowed, bypassed or overwritten.

Following that we studied a particular category of non-overriding descriptors: methods. Console testing revealed how a function attached to a class becomes a method when accessed through an instance, by leveraging the descriptor protocol.

To conclude the chapter we had "Descriptor usage tips" on page 650 and a brief look at how descriptor deletion and documentation work.

Throughout this chapter we faced a few issues that only class metaprogramming can solve, and we deferred those to Chapter 21.

Further reading

Besides the obligatory reference to the Data model page, Reymond Hettinger's Descriptor HowTo Guide is a valuable resource — part of the HowTo collection in the official Python documentation.

As usual with Python object model subjects, Alex Martelli's *Python in a Nutshell, 2nd Edition* (O'Reilly, 2003) is authoritative and objective, even if somewhat dated: the key mechanisms discussed in this chapter were introduced in Python 2.2, long before the 2.5 version covered by that book. Martelli also has a presentation titled *Python's Object Model* which covers properties and descriptors in depth (slides, video). Highly recommended.

For Python 3 coverage with practical examples, *The Python Cookbook, 3e.* (O'Reilly, 2013), by David Beazley and Brian K. Jones, has many recipes illustrating descriptors, of which I want to highlight *6.12. Reading Nested and Variable-Sized Binary Structures*, *8.10. Using Lazily Computed Properties*, *8.13. Implementing a Data Model or Type System* and *9.9. Defining Decorators As Classes* — the latter of which addresses deep issues with the interaction of function decorators, descriptors and methods, explaining how a function decorator implemented as a class with `__call__` also needs to implement `__get__` if it wants to work decorating methods as well as functions.

Soapbox

The problem with `self`

"Worse is Better" is a design philosophy described by Richard P. Gabriel in The Rise of Worse is Better. The first priority of this philosophy is "Simplicity", which Gabriel states as:

> The design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.

I believe the requirement to explicitly declare `self` as a first argument in methods is an application of "Worse is Better" in Python. The implementation is simple — elegant even — at the expense of the user interface: a method signature like `def zfill(self, width):` doesn't visually match the invocation `pobox.zfill(8)`.

Modula-3 introduced that convention — and the use of the `self` identifier — but there is a difference: in Modula-3 interfaces are declared separately from their implementation, and in the interface declaration the `self` argument is omitted, so from the user's perspective, a method appears in an interface declaration exactly with the same number of explicit arguments it takes.

One improvement is this regard have been the error messages: for a user-defined method with one argument besides `self`, if the user invokes `obj.meth()` Python 2.7 raises `TypeError: meth() takes exactly 2 arguments (1 given)` but in Python 3.4 the message is less confusing, sidestepping the issue of the argument count and naming the missing argument: `meth() missing 1 required positional argument: 'x'`.

Besides the use of self as an explicit argument, the requirement to qualify all access to instance attributes with self is also criticized[7]. I personally don't mind typing the self qualifier: it's good to distinguish local variables from attributes. My issue is with the use of self in the def statement. But I got used to it.

Anyone who is unhappy about the explicit self in Python can feel a lot better by considering the baffling semantics of the implicit this in JavaScript. Guido had some good reasons to make self work as it does, and he wrote about them in Adding Support for User-defined Classes, a post in his blog *The History of Python*.

[7] See for example A. M. Kuchling's famous *Python Warts* post (archived); Kuchling himself is not so bothered by the self qualifier, but he mentions it — probably echoing opinions from comp.lang.python.

CHAPTER 21
Class metaprogramming

> [Metaclasses] are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why)[1].
>
> — Tim Peters
> *inventor of the timsort algorithm and prolific Python contributor*

Class metaprogramming is the art of creating or customizing classes at run time. Classes are first-class objects in Python, so a function can be used to create a new class at any time, without using the `class` keyword. Class decorators are also functions, but capable of inspecting, changing and even replacing the decorated class with another class. Finally, metaclasses are the most advanced tool for class metaprogramming: they let you create whole new categories of classes with special traits, such as the abstract base classes we've already seen.

Metaclasses are powerful but hard to get right. Class decorators solve many of the same problems more simply. In fact, metaclasses are now so hard to justify in real code that my favorite motivating example lost much of its appeal with the introduction of class decorators in Python 3.

Also covered here is the distinction between import time and run time: a crucial prerequisite for effective Python metaprogramming.

This is an exciting topic, and it's easy to get carried away. So I must start this chapter with the following admonition:

[1] Message to `comp.lang.python`, subject: "Acrimony in c.l.p.". This is another part of the same message from Dec. 23, 2002, quoted in the Preface. The TimBot was inspired that day.

If you are not authoring a framework, you should not be writing metaclasses — unless you're doing it for fun or to practice the concepts.

We'll get started reviewing how to create a class at run time.

A class factory

The standard library has a class factory that we've seen several times in this book: `collections.namedtuple`. It's a function that, given a class name and attribute names creates a subclass of `tuple` that allows retrieving items by name and provides a nice `__repr__` for debugging.

Sometimes I've felt the need for a similar factory for mutable objects. Suppose I'm writing a pet shop application and I want to process data for dogs as simple records. It's bad to have to write boilerplate like this:

```
class Dog:
    def __init__(self, name, weight, owner):
        self.name = name
        self.weight = weight
        self.owner = owner
```

Boring… the field names appear three times each. All that boilerplate doesn't even buy us a nice `repr`:

```
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex
<__main__.Dog object at 0x2865bac>
```

Taking a hint from `collections.namedtuple`, let's create a `record_factory` that creates simple classes like `Dog` on the fly. Example 21-1 shows how it should work.

Example 21-1. Testing `record_factory`, a simple class factory.

```
>>> Dog = record_factory('Dog', 'name weight owner')  ❶
>>> rex = Dog('Rex', 30, 'Bob')
>>> rex  ❷
Dog(name='Rex', weight=30, owner='Bob')
>>> name, weight, _ = rex  ❸
>>> name, weight
('Rex', 30)
>>> "{2}'s dog weighs {1}kg".format(*rex)  ❹
"Bob's dog weighs 30kg"
>>> rex.weight = 32  ❺
>>> rex
Dog(name='Rex', weight=32, owner='Bob')
```

```
    >>> Dog.__mro__  ❻
    (<class 'factories.Dog'>, <class 'object'>)
```

❶ Factory signature is similar to that of `namedtuple`: class name, followed by attribute names in a single string, separated by spaces or commas.

❷ Nice `repr`.

❸ Instances are iterable, so they can be conveniently unpacked on assignment…

❹ …or when passing to functions like `format`.

❺ A record instance is mutable.

❻ The newly created class inherits from `object` — no relationship to our factory.

The code for `record_factory` is in Example 21-2[2].

Example 21-2. `record_factory.py`: A simple class factory.

```
def record_factory(cls_name, field_names):
    try:
        field_names = field_names.replace(',', ' ').split()  ❶
    except AttributeError:  # no .replace or .split
        pass  # assume it's already a sequence of identifiers
    field_names = tuple(field_names)  ❷

    def __init__(self, *args, **kwargs):  ❸
        attrs = dict(zip(self.__slots__, args))
        attrs.update(kwargs)
        for name, value in attrs.items():
            setattr(self, name, value)

    def __iter__(self):  ❹
        for name in self.__slots__:
            yield getattr(self, name)

    def __repr__(self):  ❺
        values = ', '.join('{}={!r}'.format(*i) for i
                           in zip(self.__slots__, self))
        return '{}({})'.format(self.__class__.__name__, values)

    cls_attrs = dict(__slots__ = field_names,  ❻
                     __init__  = __init__,
                     __iter__  = __iter__,
                     __repr__  = __repr__)

    return type(cls_name, (object,), cls_attrs)  ❼
```

[2] Thanks to my friend J.S. Bueno for suggesting this solution.

❶ Duck typing in practice: try to split `field_names` by commas or spaces; if that fails, assume it's already an iterable, with one name per item.

❷ Build a tuple of attribute names, this will be the `__slots__` attribute of the new class; this also sets the order of the fields for unpacking and `__repr__`.

❸ This function will become the `__init__` method in the new class. It accepts positional and/or keyword arguments.

❹ Implement an `__iter__`, so the class instances will be iterable; yield the field values in the order given by the `__slots__`.

❺ Produce the nice `repr`, iterating over `__slots__` and `self`.

❻ Assemble dictionary of class attributes.

❼ Build and return the new class, calling the `type` constructor.

We usually think of `type` as a function, because we use it like one, e.g. `type(my_object)` to get the class of the object — same as `my_object.__class__`. However, `type` is a class. It behaves like a class that creates a new class when invoked with three arguments:

```
MyClass = type('MyClass', (MySuperClass, MyMixin),
               {'x': 42, 'x2': lambda self: self.x * 2})
```

The three arguments of `type` are named `name`, `bases` and `dict` — the latter being a mapping of attribute names and attributes for the new class. The code above is functionally equivalent to this:

```
class MyClass(MySuperClass, MyMixin):
    x = 42

    def x2(self):
        return self.x * 2
```

The novelty here is that the instances of `type` are classes, like `MyClass` above, or the `Dog` class in Example 21-1.

In summary, the last line of `record_factory` in Example 21-2 builds a class named by the value of `cls_name`, with `object` as its single immediate superclass and with class attributes named `__slots__`, `__init__`, `__iter__` and `__repr__`, of which the last three are instance methods.

We could have named the `__slots__` class attribute anything else, but then we'd have to implement `__setattr__` to validate the names of attributes being assigned, because for our record-like classes we want the set of attributes to be always the same and in the same order. However, recall that the main feature of `__slots__` is saving memory when you are dealing with millions of instances, and using `__slots__` has some drawbacks, discussed in "Saving space with the `__slots__` class attribute" on page 265.

Invoking `type` with three arguments is a common way of creating a class dynamically. If you peek at the source code for `collections.namedtuple` you'll see a different approach: there is `_class_template`, a source code template as a string, and the `namedtuple` function fills its blanks calling `_class_template.format(…)`. The resulting source code string is then evaluated with the `exec` built-in function.

> It's good practice to avoid `exec` or `eval` for metaprogramming in Python. These functions pose serious security risks if they are fed strings (even fragments) from untrusted sources. Python offers sufficient introspection tools to make `exec` and `eval` unnecessary most of the time. However, the Python core developers chose to use `exec` when implementing `namedtuple`. The chosen approach makes the code generated for the class available in the `._source` attribute.

Instances of classes created by `record_factory` have a limitation: they are not serializable — i.e. can't be used with the `dump/load` functions from the `pickle` module. Solving this problem is beyond the scope of this example which aims to show the `type` class in action in a simple use case. For the full solution, study the source code for `collections.nameduple`; search for the word "pickling".

A class decorator for customizing descriptors

When we left the `LineItem` example in "`LineItem` take #5: a new descriptor type" on page 639 the issue of descriptive storage names was still pending: the values of attributes such as `weight` was stored in an instance attribute named `_Quantity#0`, which made debugging a bit hard. You can retrieve the storage name from a descriptor in Example 20-7 with the following lines:

```
>>> LineItem.weight.storage_name
'_Quantity#0'
```

However it would be better if the storage names actually included the name of the managed attribute, like this:

```
>>> LineItem.weight.storage_name
'_Quantity#weight'
```

Recall from "`LineItem` take #4: automatic storage attribute names" on page 633 that we could not use descriptive storage names because when the descriptor is instantiated it has no way of knowing the name of the managed attribute, i.e. the class attribute to which the descriptor will be bound, such as `weight` in the examples above. But once the whole class is assembled and the descriptors are bound to the class attributes, we can inspect the class and set proper storage names to the descriptors. This could be done in the `__new__` method of the `LineItem` class, so that by the time the descriptors are used in the `__init__` method, the correct storage names are set. The problem of using new for that purpose is wasted effort: the logic of `__new__` will run every time a new `LineItem` instance is created, but the binding of the descriptor to the managed attribute will never change once the `LineItem` class itself is built. So we need to set the storage names when the class is created. That can be done with a class decorator or a metaclass. We'll do it first in the easier way.

A class decorator is very similar to a function decorator: it's a function that gets a class object and returns the same class or a modified one.

In Example 21-3, the `LineItem` class will be evaluated by the interpreter and the resulting class object will be passed to the `model.entity` function. Python will bind the global name `LineItem` to whatever the `model.entity` function returns. In this example, `model.entity` returns the same `LineItem` class with the `storage_name` attribute of each descriptor instance changed.

Example 21-3. `bulkfood_v6.py`: `LineItem` using `Quantity` and `NonBlank` descriptors

```
import model_v6 as model

@model.entity  ❶
class LineItem:
    description = model.NonBlank()
    weight = model.Quantity()
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price

    def subtotal(self):
        return self.weight * self.price
```

❶ The only change in this class is the added decorator.

Example 21-4 shows the implementation of the decorator. Only the new code at the bottom of `model_v6.py` is listed here; the rest of the module is identical to `model_v5.py` (Example 20-6).

Example 21-4. `model_v6.py`: A class decorator.

```
def entity(cls):  ❶
    for key, attr in cls.__dict__.items():  ❷
        if isinstance(attr, Validated):  ❸
            type_name = type(attr).__name__
            attr.storage_name = '_{}#{}'.format(type_name, key)  ❹
    return cls  ❺
```

❶ Decorator gets class as argument.

❷ Iterate over `dict` holding the class attributes.

❸ If the attribute is one of our `Validated` descriptors…

❹ …set the `storage_name` to use the descriptor class name and the managed attribute name, e.g. `_NonBlank#description`.

❺ Return the modified class.

The doctests in `bulkfood_v6.py` prove that the changes are successful. For example, this test shows the names of the storage attributes in a `LineItem` instance:

Example 21-5. `bulkfood_v6.py`: Doctests for new `storage_name` descriptor attributes.

```
    >>> raisins = LineItem('Golden raisins', 10, 6.95)
    >>> dir(raisins)[:3]
    ['_NonBlank#description', '_Quantity#price', '_Quantity#weight']
    >>> LineItem.description.storage_name
    '_NonBlank#description'
    >>> raisins.description
    'Golden raisins'
    >>> getattr(raisins, '_NonBlank#description')
    'Golden raisins'
```

That's not too complicated. Class decorators are a simpler way of doing something that previously required a metaclass: customizing a class the moment it's created.

A significant drawback of class decorators is that they act only on the class where they are directly applied. This means subclasses of the decorated class may or may not inherit the changes made by the decorator, depending on what those changes are. We'll explore the problem and see how it's solved in the next sections.

What happens when: import time versus run time

For successful metaprogramming, you must be aware of when the Python interpreter evaluates each block of code. Python programmers talk about "import time" versus "run time" but the terms are not strictly defined and there is a gray area between them. At import time, the interpreter parses the source code of a `.py` module in one pass from top to bottom, and generates the bytecode to be executed. That's when syntax errors may occur. If there is an up-to-date `.pyc` file available in the local `__pycache__`, those steps are skipped because the bytecode is ready to run.

Although compiling is definitely an import time activity, other things may happen at that time, because almost every statement in Python is executable in the sense that they potentially run user code and change the state of the user program. In particular, the `import` statement is not merely a declaration[3] but it actually runs all the top level code of the imported module when it's imported for the first time in the process — further imports of the same module will use a cache, and only name binding occurs then. That top level code may do anything, including actions typical of "run time", such as connecting to a database[4]. That's why the border between "import time" and "run time" is fuzzy: the `import` statement can trigger all sorts of "run time" behavior.

In the previous paragraph I wrote that importing "runs all the top level code", but "top level code" requires some elaboration. The interpreter executes a `def` statement on the top level of a module when the module is imported, but what does that achieve? The interpreter compiles the function body (if it's the first time that module is imported), and binds the function object to its global name, but it does not execute the body of the function, obviously. In the usual case, this means that the interpreter defines top level functions at import time, but executes their bodies only when — and if — the functions are invoked at run time.

For classes, the story is different: at import time, the interpreter executes the body of every class, even the body of classes nested in other classes. Execution of a class body means that the attributes and methods of the class are defined, and then the class object itself is built. In this sense, the body of classes is "top level code": it runs at import time.

This is all rather subtle and abstract, so here is an exercise to help you see what happens when.

The evaluation time exercises

Students have reported these exercises are helpful to better appreciate how Python evaluates the source code. Do take the time to solve them with paper and pencil before looking at "Solution for scenario #1" on page 666.

Consider a script, `evaltime.py`, which imports a module `evalsupport.py`. Both modules have several `print` calls to output markers in the format `<[N]>`, where `N` is a number. The goal of this pair of exercises is to determine when each of theses calls will be made.

The listings are Example 21-6 and Example 21-7. Grab paper and pencil and — without running the code — write down the markers in the order they will appear in the output, in two scenarios:

[3] Contrast with the `import` statement in Java which is just a declaration to let the compiler know that certain packages are required.

[4] I'm not saying starting a database connection just because a module is imported is a good idea, only pointing out it can be done.

Scenario #1

The module `evaltime.py` is imported interactively in the Python console:

```
>>> import evaltime
```

Scenario #2

The module `evaltime.py` is run from the command shell:

```
$ python3 evaltime.py
```

Example 21-6. `evaltime.py`: Write down the numbered <[N]> markers in the order they will appear in the output.

```
from evalsupport import deco_alpha

print('<[1]> evaltime module start')


class ClassOne():
    print('<[2]> ClassOne body')

    def __init__(self):
        print('<[3]> ClassOne.__init__')

    def __del__(self):
        print('<[4]> ClassOne.__del__')

    def method_x(self):
        print('<[5]> ClassOne.method_x')

    class ClassTwo(object):
        print('<[6]> ClassTwo body')


@deco_alpha
class ClassThree():
    print('<[7]> ClassThree body')

    def method_y(self):
        print('<[8]> ClassThree.method_y')


class ClassFour(ClassThree):
    print('<[9]> ClassFour body')

    def method_y(self):
        print('<[10]> ClassFour.method_y')


if __name__ == '__main__':
    print('<[11]> ClassOne tests', 30 * '.')
    one = ClassOne()
    one.method_x()
    print('<[12]> ClassThree tests', 30 * '.')
    three = ClassThree()
    three.method_y()
    print('<[13]> ClassFour tests', 30 * '.')
    four = ClassFour()
    four.method_y()


print('<[14]> evaltime module end')
```

Example 21-7. `evalsupport.py`: Module imported by `evaltime.py`.

```
print('<[100]> evalsupport module start')

def deco_alpha(cls):
    print('<[200]> deco_alpha')

    def inner_1(self):
        print('<[300]> deco_alpha:inner_1')

    cls.method_y = inner_1
    return cls

# BEGIN META_ALEPH
class MetaAleph(type):
    print('<[400]> MetaAleph body')

    def __init__(cls, name, bases, dic):
        print('<[500]> MetaAleph.__init__')

        def inner_2(self):
            print('<[600]> MetaAleph.__init__:inner_2')

        cls.method_z = inner_2
# END META_ALEPH

print('<[700]> evalsupport module end')
```

Solution for scenario #1

Example 21-8. Scenario #1: importing `evaltime` in the Python console.

```
>>> import evaltime
<[100]> evalsupport module start  ❶
<[400]> MetaAleph body  ❷
<[700]> evalsupport module end
<[1]> evaltime module start
<[2]> ClassOne body  ❸
<[6]> ClassTwo body  ❹
<[7]> ClassThree body
<[200]> deco_alpha  ❺
<[9]> ClassFour body
<[14]> evaltime module end  ❻
```

❶ All top level code in `evalsupport` runs when the module is imported; the `deco_alpha` function is compiled, but its body does not execute.

❷ The body of the `MetaAleph` function does run.

❸ The body of every class is executed…

❹ …including nested classes.

❺ The decorator function runs after the body of the decorated `ClassThree` is evaluated.

❻ In this scenario the `evaltime` is imported, so the `if __name__ == '__main__':` block never runs.

Notes about scenario #1:

1. This scenario is triggered by a simple `import evaltime` statement.
2. The interpreter executes every class body of the imported module and its dependency, `evalsupport`.
3. It makes sense that the interpreter evaluates the body of a decorated class before it invokes the decorator function that is attached on top of it: the decorator must get a class object to process, so the class object must be built first.
4. The only user-defined function or method that runs in this scenario is the `deco_alpha` decorator.

Now let's see what happens in scenario #2.

Solution for scenario #2

Example 21-9. Scenario #2: Running `evaltime.py` from the shell.

```
$ python3 evaltime.py
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime module start
<[2]> ClassOne body
<[6]> ClassTwo body
<[7]> ClassThree body
<[200]> deco_alpha
<[9]> ClassFour body  ❶
<[11]> ClassOne tests ..............................
<[3]> ClassOne.__init__  ❷
<[5]> ClassOne.method_x
<[12]> ClassThree tests ..............................
<[300]> deco_alpha:inner_1  ❸
<[13]> ClassFour tests ..............................
<[10]> ClassFour.method_y
```

```
<[14]> evaltime module end
<[4]> ClassOne.__del__  ❹
```

❶ Same output as Example 21-8 so far.

❷ Standard behavior of a class.

❸ `ClassThree.method_y` was changed by the `deco_alpha` decorator, so the call `three.method_y()` runs the body of the `inner_1` function.

❹ The `ClassOne` instance bound to `one` global variable is garbage-collected only when the program ends.

The main point of scenario #2 is to show that the effects of a class decorator may not affect subclasses. In Example 21-6 `ClassFour` is defined as a subclass of `ClassThree`. The `@deco_alpha` decorator is applied to `ClassThree`, replacing its `method_y`, but that does not affect `ClassFour` at all. Of course, if the `ClassFour.method_y` did invoke the `ClassThree.method_y` with `super(…)`, we would see the effect of the decorator, as the `inner_1` function executed.

In contrast, the next section will show that metaclasses are more effective when we want to customize a whole class hierarchy, and not one class at a time.

Metaclasses 101

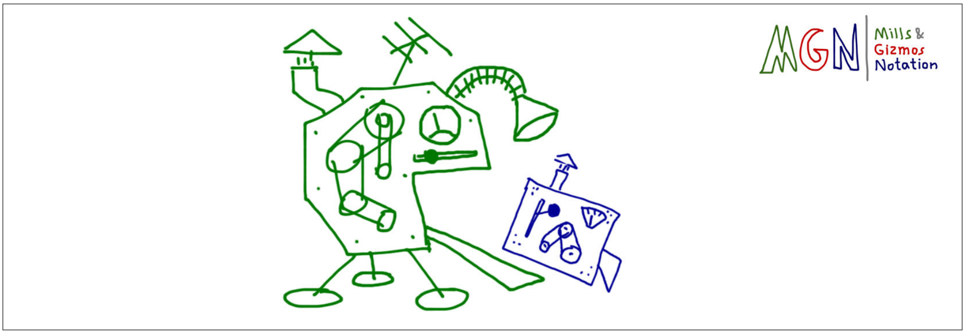


Figure 21-1. A metaclass is a class that builds classes.

A metaclass is a class factory, except that instead of a function, like `record_factory` from Example 21-2, a metaclass is written as a class.

Consider the Python object model: classes are objects, therefore each class must be an instance of some other class. By default, Python classes are instances of `type`. In other words, `type` is the metaclass for most built-in and user-defined classes:

```
>>> 'spam'.__class__
<class 'str'>
>>> str.__class__
<class 'type'>
>>> from bulkfood_v6 import LineItem
>>> LineItem.__class__
<class 'type'>
>>> type.__class__
<class 'type'>
```

To avoid infinite regress, type is an instance of itself, as the last line above shows.

Note that I am not saying that str or LineItem inherit from type. What I am saying is that str and LineItem are instances of type. They all are subclasses of object. Figure 21-2 may help you confront this strange reality.

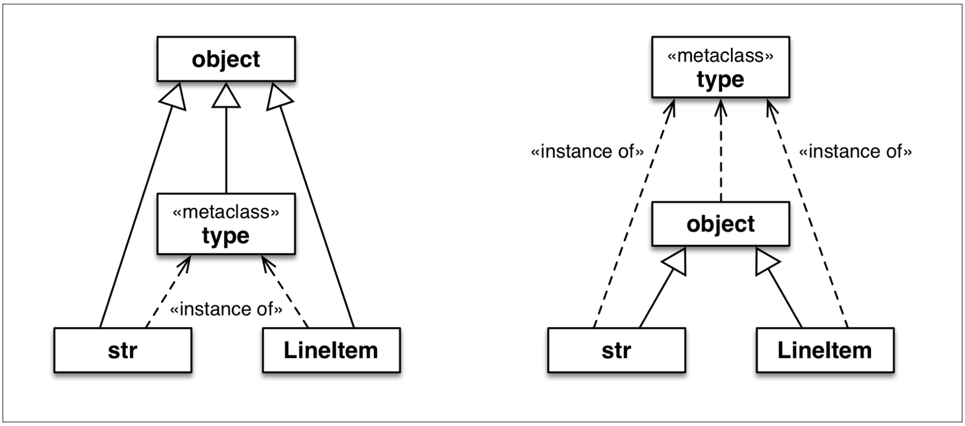


Figure 21-2. Both diagrams are true. The left one emphasizes that str, type and LineItem are subclasses of object. The right one makes it clear that str, object and LineItem are instances of type, because they are all classes.

> The classes object and type have a unique relationship: object is an instance of type, and type is a subclass of object. This relationship is "magic": it cannot be expressed in Python because either class would have to exist before the other could be defined. The fact that type is an instance of itself is also magical.

Besides type, a few other metaclasses exist in the standard library, such as ABCMeta and Enum. The next snippet shows that the class of collections.Iterable is abc.ABCMeta. The class Iterable is abstract, but ABCMeta is not — after all, Iterable is an instance of ABCMeta.

```
>>> import collections
>>> collections.Iterable.__class__
<class 'abc.ABCMeta'>
>>> import abc
>>> abc.ABCMeta.__class__
<class 'type'>
>>> abc.ABCMeta.__mro__
(<class 'abc.ABCMeta'>, <class 'type'>, <class 'object'>)
```

Ultimately, the class of ABCMeta is also type. Every class is an instance of type, directly or indirectly, but only metaclasses are also subclasses of type. That's the most important relationship to understand metaclasses: a metaclass, such as ABCMeta inherits from type the power to construct classes. Figure 21-3 illustrates this crucial relationship.

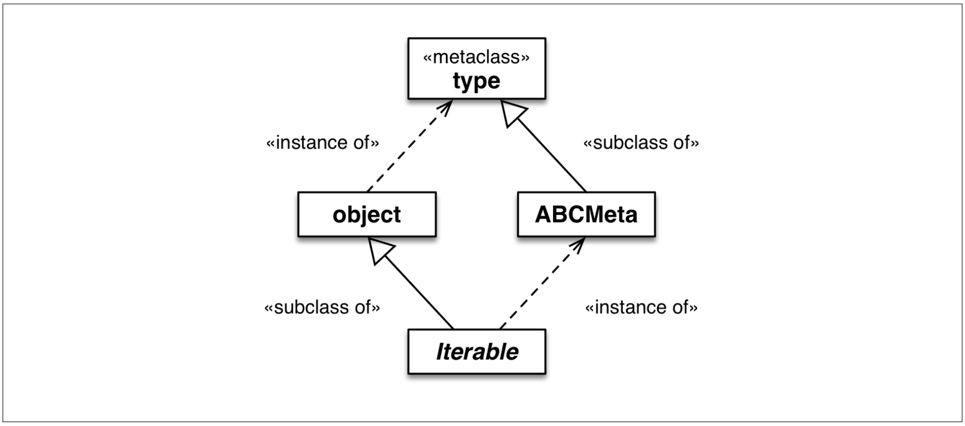


Figure 21-3. Iterable is a subclass of object and an instance of ABCMeta. Both object and ABCMeta are instances of type, but the the key relationship here is that ABCMeta is also a subclass of type, because ABCMeta is a metaclass. In this diagram, Iterable is the only abstract class.

The important takeaway here is that all classes are instances of type, but metaclasses are also subclasses of type, so they act as class factories. In particular, a metaclass can customize its instances by implementing __init__. A metaclass __init__ method can do everything a class decorator can do, but its effects are more profound, as the next exercise demonstrates.

The metaclass evaluation time exercise

This is a variation of "The evaluation time exercises" on page 664. The evalsupport.py module is the same as Example 21-7, but the main script is now evaltime_meta.py, listed in Example 21-10.

Example 21-10. `evaltime_meta.py`: `ClassFive` is an instance of the `MetaAleph` metaclass.

```
from evalsupport import deco_alpha
from evalsupport import MetaAleph

print('<[1]> evaltime_meta module start')


@deco_alpha
class ClassThree():
    print('<[2]> ClassThree body')

    def method_y(self):
        print('<[3]> ClassThree.method_y')


class ClassFour(ClassThree):
    print('<[4]> ClassFour body')

    def method_y(self):
        print('<[5]> ClassFour.method_y')


class ClassFive(metaclass=MetaAleph):
    print('<[6]> ClassFive body')

    def __init__(self):
        print('<[7]> ClassFive.__init__')

    def method_z(self):
        print('<[8]> ClassFive.method_y')


class ClassSix(ClassFive):
    print('<[9]> ClassSix body')

    def method_z(self):
        print('<[10]> ClassSix.method_y')


if __name__ == '__main__':
    print('<[11]> ClassThree tests', 30 * '.')
    three = ClassThree()
    three.method_y()
    print('<[12]> ClassFour tests', 30 * '.')
    four = ClassFour()
    four.method_y()
    print('<[13]> ClassFive tests', 30 * '.')
    five = ClassFive()
    five.method_z()
    print('<[14]> ClassSix tests', 30 * '.')
    six = ClassSix()
    six.method_z()

print('<[15]> evaltime_meta module end')
```

Again, grab pencil and paper and write down the numbered `<[N]>` markers in the order they will appear in the output, considering these two scenarios:

Scenario #3
The module `evaltime_meta.py` is imported interactively in the Python console.

Scenario #4
The module `evaltime_meta.py` is run from the command shell.

Solutions and analysis are next.

Solution for scenario #3

Example 21-11. Scenario #3: Importing `evaltime_meta` in the Python console.

```
>>> import evaltime_meta
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime_meta module start
<[2]> ClassThree body
<[200]> deco_alpha
<[4]> ClassFour body
<[6]> ClassFive body
<[500]> MetaAleph.__init__  ❶
<[9]> ClassSix body
<[500]> MetaAleph.__init__  ❷
<[15]> evaltime_meta module end
```

❶ The key difference from scenario #1 is that the `MetaAleph.__init__` method is invoked to initialize the just-created `ClassFive`.

❷ And `MetaAleph.__init__` also initializes `ClassSix`, which is a subclass of `ClassFive`.

The Python interpreter evaluates the body of `ClassFive` but then, instead of calling `type` to build the actual class body, it calls `MetaAleph`. Looking at the definition of `MetaAleph` in Example 21-12 you'll see that the `__init__` method gets four arguments:

`self`
That's the class object being initialized, e.g. `ClassFive`.

`name`, `bases`, `dic`
The same arguments passed to `type` to build a class.

Example 21-12. `evalsupport.py`: Definition of the metaclass `MetaAleph` from Example 21-7.

```
class MetaAleph(type):
    print('<[400]> MetaAleph body')

    def __init__(cls, name, bases, dic):
        print('<[500]> MetaAleph.__init__')

        def inner_2(self):
            print('<[600]> MetaAleph.__init__:inner_2')

        cls.method_z = inner_2
```

When coding a metaclass, it's conventional to replace `self` by `cls`. For example, in the `__init__` method of the metaclass, using `cls` as the name of the first argument makes it clear that the instance under construction is a class.

The body of `__init__` defines an `inner_2` function, then binds it to `cls.method_z`. The name `cls` in the signature of `MetaAleph.__init__` refers to the class being created, e.g. `ClassFive`. On the other hand, the name `self` in the signature of `inner_2` will eventually refer to an instance of the class we are creating, e.g. an instance of `ClassFive`.

Solution for scenario #4

Example 21-13. Scenario #4: Running `evaltime_meta.py` from the shell.

```
$ python3 evaltime.py
<[100]> evalsupport module start
<[400]> MetaAleph body
<[700]> evalsupport module end
<[1]> evaltime_meta module start
<[2]> ClassThree body
<[200]> deco_alpha
<[4]> ClassFour body
<[6]> ClassFive body
<[500]> MetaAleph.__init__
<[9]> ClassSix body
<[500]> MetaAleph.__init__
<[11]> ClassThree tests ..............................
<[300]> deco_alpha:inner_1  ❶
<[12]> ClassFour tests ..............................
<[5]> ClassFour.method_y  ❷
<[13]> ClassFive tests ..............................
<[7]> ClassFive.__init__
<[600]> MetaAleph.__init__:inner_2  ❸
<[14]> ClassSix tests ..............................
<[7]> ClassFive.__init__
```

```
<[600]> MetaAleph.__init__:inner_2  ❹
<[15]> evaltime_meta module end
```

❶ When the decorator is applied to `ClassThree`, its `method_y` is replaced by the `inner_1` method…

❷ But this has no effect on the undecorated `ClassFour`, even though `ClassFour` is a subclass of `ClassThree`.

❸ The `__init__` method of `MetaAleph` replaces `ClassFive.method_z` with its `inner_2` function.

❹ The same happens with the `ClassFive` subclass, `ClassSix`: its `method_z` is replaced by `inner_2`.

Note that `ClassSix` makes no direct reference to `MetaAleph`, but it is affected by it because it's a subclass of `ClassFive` and therefore it is also an instance of `MetaAleph`, so it's initialized by `MetaAleph.__init__`.

Further class customization can be done by implementing `__new__` in a metaclass. But more often than not, implementing `__init__` is enough.

We can now put all this theory in practice by creating a metaclass to provide a definitive solution to the descriptors with automatic storage attribute names.

A metaclass for customizing descriptors

Back to the `LineItem` examples, it would be nice if the user did not have to be aware of decorators or metaclasses at all, and could just inherit from a class provided by our library, like this:

Example 21-14. `bulkfood_v7.py`: Inheriting from `model.Entity` can work, if a metaclass is behind the scene.

```
import model_v7 as model

class LineItem(model.Entity):  ❶
    description = model.NonBlank()
    weight = model.Quantity()
    price = model.Quantity()

    def __init__(self, description, weight, price):
        self.description = description
        self.weight = weight
        self.price = price
    def subtotal(self):
        return self.weight * self.price
```

❶ `LineItem` is a subclass of `model.Entity`.

Example 21-14 looks pretty harmless. No strange syntax to be seen at all. However, it only works because `model_v7.py` defines a metaclass, and `model.Entity` is an instance of that metaclass.

Example 21-15. `model_v7.py`: The `EntityMeta` metaclass and one instance of it, `Entity`.

```
class EntityMeta(type):
    """Metaclass for business entities with validated fields"""

    def __init__(cls, name, bases, attr_dict):
        super().__init__(name, bases, attr_dict)  ❶
        for key, attr in attr_dict.items():  ❷
            if isinstance(attr, Validated):
                type_name = type(attr).__name__
                attr.storage_name = '_{}#{}'.format(type_name, key)

class Entity(metaclass=EntityMeta):  ❸
    """Business entity with validated fields"""
```

❶ Call `__init__` on the superclass (`type` in this case).

❷ Same logic as the `@entity` decorator in Example 21-4.

❸ This class exists for convenience only: the user of this module can just subclass `Entity` and not worry about `EntityMeta` — or even be aware of its existence.

The code in Example 21-14 passes the tests in Example 21-3. The support module, `model_v7.py` is harder to understand than `model_v6.py`, but the user-level code is simpler: just inherit from `model_v7.entity` and you get custom storage names for your `Validated` fields.

Figure 21-4 is a simplified depiction of what we just implemented. There is a lot going on, but the complexity is hidden inside the `model_v7` module. From the user perspective, `LineItem` is simply a subclass of `Entity`, as coded in Example 21-14. This is the power of abstraction.

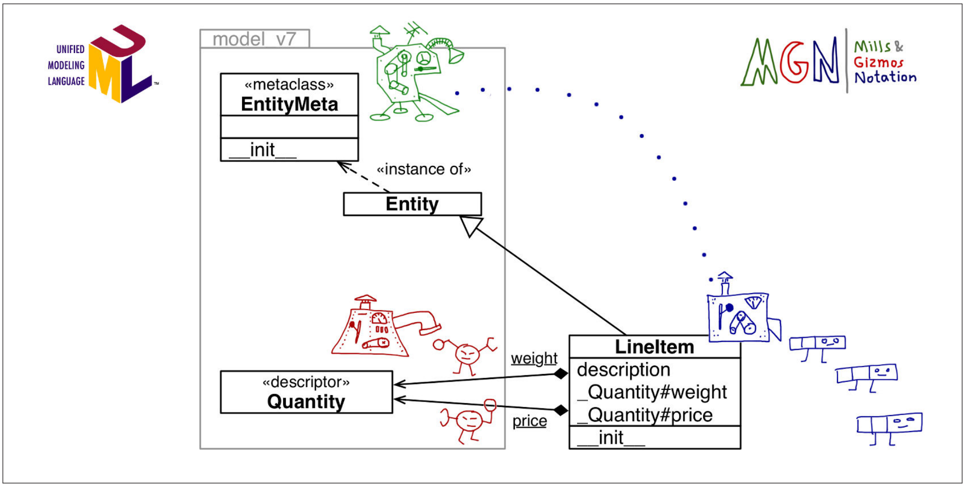


Figure 21-4. UML class diagram annotated with MGN (Mills & Gizmos Notation): the `EntityMeta` meta-mill builds the `LineItem` mill. Configuration of the descriptors (e.g. `weight` and `price`) is done by `EntityMeta.__init__`. Note the package boundary of `model_v7`.

Except for the syntax for linking a class to the metaclass[5], everything written so far about metaclasses applies to versions of Python as early as 2.2, when Python types underwent a major overhaul. The next section covers a feature that is only available in Python 3.

The metaclass `__prepare__` special method

In some applications it's interesting to be able to know the order in which the attributes of a class are defined. For example, a library to read/write CSV files driven by user-defined classes may want to map the order of the fields declared in the class to the order of the columns in the CSV file.

As we've seen, both the `type` constructor and the `__new__` and `__init__` methods of metaclasses receive the body of the class evaluated as a mapping of names to attributes. However, by default, that mapping is a `dict`, which means the order of the attributes as they appear in the class body is lost by the time our metaclass or class decorator can look at them.

The solution to this problem is the `__prepare__` special method, introduced in Python 3. This special method is relevant only in metaclasses, and it must be a class method,

[5] Recall from "ABC syntax details" on page 330 that in Python 2.7 the `__metaclass__` class attribute is used, and the `metaclass=` keyword argument is not supported in the class declaration.

i.e. defined with the `@classmethod` decorator. The `__prepare__` method is invoked by the interpreter before the `__new__` method in the metaclass to create the mapping that will be filled with the attributes from the class body. Besides the metaclass as first argument, `__prepare__` gets the name of the class to be constructed and its tuple of base classes, and it must return a mapping which will be received as the last argument by `__new__` and then `__init__` when the metaclass builds a new class.

This is sounds complicated in theory but in practice every time I've seen `__prepare__` being used it was very simple. Take a look at Example 21-16.

Example 21-16. `model_v8.py`: The `EntityMeta` metaclass uses `__prepare__`, and `Entity` now has a `field_names` class method.

```
class EntityMeta(type):
    """Metaclass for business entities with validated fields"""

    @classmethod
    def __prepare__(cls, name, bases):
        return collections.OrderedDict()  ❶

    def __init__(cls, name, bases, attr_dict):
        super().__init__(name, bases, attr_dict)
        cls._field_names = []  ❷
        for key, attr in attr_dict.items():  ❸
            if isinstance(attr, Validated):
                type_name = type(attr).__name__
                attr.storage_name = '_{}#{}'.format(type_name, key)
                cls._field_names.append(key)  ❹


class Entity(metaclass=EntityMeta):
    """Business entity with validated fields"""

    @classmethod
    def field_names(cls):  ❺
        for name in cls._field_names:
            yield name
```

❶ Return an empty `OrderedDict` instance, where the class attributes will be stored.

❷ Create a `_field_names` attribute in the class under construction.

❸ This line is unchanged from the previous version, but `attr_dict` here is the `OrderedDict` obtained by the interpreter when it called `__prepare__` before calling `__init__`. Therefore, this `for` loop will go over the attributes in the order they were added.

❹ Add the name of each `Validated` field found to `_field_names`.

❺ The `field_names` class method simply yields the names of the fields in the order they were added.

With the simple additions made in Example 21-16, we are now able to iterate over the `Validated` fields of any `Entity` subclass using the `field_names` class method. For example:

Example 21-17. `bulkfood_v8.py`: Doctest showing the use of `field_names`. No changes are needed in the `LineItem` class; `field_names` is inherited from `model.Entity`.

```
>>> for name in LineItem.field_names():
...     print(name)
...
description
weight
price
```

This wraps up our coverage of metaclasses. In the real world, metaclasses are used in frameworks and libraries that help programmers perform, among other tasks:

- attribute validation;
- applying decorators to many methods at once;
- object serialization or data conversion;
- object-relational mapping;
- object-based persistency;
- dynamic translation of class structures from other languages.

We'll now have a brief overview of methods defined in the Python Data Model for all classes.

Classes as objects

Every class has a number of attributes defined in the Python data model, documented in the Special Attributes section of the *Built-in Types* chapter in the *Library Reference*. Three of those attributes we've seen several times in the book already: `__mro__`, `__class__` and `__name__`. Other class attributes are:

`cls.__bases__`
  The tuple of base classes of the class.

`cls.__qualname__`
  A new attribute in Python 3.3 holding the qualified name of a class or function, which is a dotted path from the global scope of the module to the class definition. For example, in Example 21-6 the `__qualname__` the inner class `ClassTwo` is the

string `'ClassOne.ClassTwo'`, while its `__name__` is just `'ClassTwo'`. The specification for this attribute is PEP-3155 — Qualified name for classes and functions.

`cls.__subclasses__()`
: This method returns a list of the immediate subclasses of the class. The implementation uses weak references to avoid circular references between the superclass and its subclasses — which hold a strong reference to the superclasses in their `__bases__` attribute. The method returns the list of subclasses that currently exist in memory.

`cls.mro()`
: The interpreter calls this method when building a class to obtain the tuple of superclasses that is stored in the `__mro__` attribute of the class. A metaclass can override this method to customize the method resolution order of the class under construction.

> None of the attributes mentioned in this section are listed by the `dir(…)` function.

With this, our study of class metaprogramming ends. This is a vast topic and I only scratched the surface. That's why we have **Further Reading** sections in this book.

Chapter summary

Class metaprogramming is about creating or customizing classes dynamically. Classes in Python are first-class objects, so we started the chapter by showing how a class can be created by a function invoking the `type` built-in metaclass.

In the next section we went back to the `LineItem` class with descriptors from Chapter 20 to solve a lingering issue: how to generate names for the storage attributes that reflected the names of the managed attributes, e.g. `_Quantity#price` instead of `_Quantity#1`. The solution was to use a class decorator, essentially a function that gets a just-built class and has the opportunity to inspect it, change it and even replace it with a different class.

We then moved to a discussion of when different parts of the source code of a module actually run. We saw that there is some overlap between the so called "import time" and "run time", but clearly a lot of code runs triggered by the `import` statement. Understanding what runs when is crucial, and there are some subtle rules, so we used the evaluation time exercises to cover this topic.

The following subject was an introduction to metaclasses. We saw that all classes are instances of `type`, directly or indirectly, so that is the "root metaclass" of the language. A variation of the evaluation time exercise was designed to show that a metaclass can customize a hierarchy of classes — in contrast with a class decorator which affects a single class and may have no impact on its descendants.

The first practical application of a metaclass was to solve the issue of the storage attribute names in `LineItem`. The resulting code is a bit trickier than the class decorator solution, but it can be encapsulated in a module so that the user merely subclasses an apparently plain class (`model.Entity`) without being aware that it is an instance of a custom metaclass (`model.EntityMeta`). The end result is reminiscent of the ORM APIs in Django and SQLAlchemy, which use metaclasses in their implementations but don't require the user to know anything about them.

The second metaclass we implemented added a small feature to `model.EntityMeta`: a `__prepare__` method to provide an `OrderedDict` to serve as the mapping from names to attributes. This preserves the order in which those attributes are bound in the body of the class under construction, so that metaclass methods like `__new__` and `__init__` can use that information. In the example, we implemented a `_field_names` class attribute, which made possible an `Entity.field_names()` so users could retrieve the `Validated` descriptors in the same order they appear in the source code.

The last section was a brief overview of attributes and methods available in all Python classes.

Metaclasses are challenging, exciting and — sometimes — abused by programmers trying to be too clever. To wrap up, let's recall Alex Martelli's final advice from his essay "Waterfowl and ABCs" on page 316:

> And, **don't** define custom ABCs (or metaclasses) in production code… if you feel the urge to do so, I'd bet it's likely to be a case of "all problems look like a nail"-syndrome for somebody who just got a shiny new hammer — you (and future maintainers of your code) will be much happier sticking with straightforward and simple code, eschewing such depths.
>
> — Alex Martelli

Wise words from a man who is not only a master of Python metaprogramming but also an accomplished software engineer working on some of the largest mission-critical Python deployments in the world.

Further reading

The essential references for this chapter in the Python documentation are section 3.3.3. Customizing class creation in the *Data model* chapter of the *Language Reference*, the `type` class documentation in the *Built-in Functions* page and the Special Attributes section of the *Built-in Types* chapter in the *Library Reference*. Also, in the *Library Reference* the `types` module documentation covers two functions that are new in Python 3.3 and are designed to help with class metaprogramming: `types.new_class(…)` and `types.prepare_class(…)`.

Class decorators were formalized in PEP 3129 - Class Decorators, written by Collin Winter, with the reference implementation authored by Jack Diederich. The PyCon 2009 talk *Class Decorators: Radically Simple* (video), also by Jack Diederich, is a quick introduction to the feature.

Python in a Nutshell, 2e, by Alex Martelli features outstanding coverage of metaclasses, including a `metaMetaBunch` metaclass that aims to solve the same problem as our simple `record_factory` from Example 21-2 but is much more sophisticated. Martelli does not address class decorators because the feature appeared later than his book.Beazley and Jones provide excellent examples of class decorators and metaclasses in their *Python Cookbook, 3e* (O'Reilly, 2013). Michael Foord wrote an intriguing post titled Metaclasses Made Easy: Eliminating self with Metaclasses. The subtitle says it all.

For metaclasses the main references are PEP 3115 — Metaclasses in Python 3000, in which the `__prepare__` special method was introduced and Unifying types and classes in Python 2.2, authored by Guido van Rossum. The text applies to Python 3 as well, and it covers what were then called the "new-style" class semantics, including descriptors and metaclasses. It's a must-read. One of the references cited by Guido is *Putting Metaclasses to Work: a New Dimension in Object-Oriented Programming*, by Ira R. Forman and Scott H. Danforth (Addison-Wesley, 1998), a book to which he gave 5 stars on Amazon.com, adding the following review:

> **This book contributed to the design for metaclasses in Python 2.2**
>
> Too bad this is out of print; I keep referring to it as the best tutorial I know for the difficult subject of cooperative multiple inheritance, supported by Python via the `super()` function[6].

For Python 3.5 — in alpha as I write this — PEP 487 - Simpler customization of class creation puts forward a new special method, `__init_subclass__` that will allow a regular class — i.e. not a metaclass — to customize the initialization of its subclasses. As with class decorators, `__init_subclass__` will make class metaprogramming more accessible and also make it that much harder to justify the deployment of the nuclear option — metaclasses.

If you are into metaprogramming, you may wish Python had the ultimate metaprogramming feature: syntactic macros, as offered by Elixir and the Lisp family of languages. Be careful what you wish for. I'll just say one word: MacroPy.

[6] Amazon.com catalog page for Putting Metaclasses to Work. You can still buy it used. I bought it and found it a hard read, but I will probably go back to it later.

Soapbox

I will start the last soapbox in the book with a long quote from Brian Harvey and Matthew Wright, two computer science professors from the University of California (Berkeley and Santa Barbara). In their book, *Simply Scheme*, Harvey and Wright wrote:

> There are two schools of thought about teaching computer science. We might caricature the two views this way:
>
> 1. **The conservative view**: Computer programs have become too large and complex to encompass in a human mind. Therefore, the job of computer science education is to teach people how to discipline their work in such a way that 500 mediocre programmers can join together and produce a program that correctly meets its specification.
> 2. **The radical view**: Computer programs have become too large and complex to encompass in a human mind. Therefore, the job of computer science education is to teach people how to expand their minds so that the programs can fit, by learning to think in a vocabulary of larger, more powerful, more flexible ideas than the obvious ones. Each unit of programming thought must have a big payoff in the capabilities of the program[7].
>
> — Brian Harvey and Matthew Wright
> *Preface to Simply Scheme*

Harvey and Wright's exaggerated descriptions are about teaching computer science, but they also apply to programming language design. By now you should have guessed that I subscribe to the "radical" view, and I believe Python was designed in that spirit.

The property idea is a great step forward compared to the accessors-from-the-start approach practically demanded by Java and supported by Java IDEs generating getters/setters with a keyboard shortcut. The main advantage of properties is to let us start our programs simply exposing attributes as public — in the spirit of *KISS* — knowing a public attribute can become a property at any time without much pain. But the descriptor idea goes way beyond that, providing a framework for abstracting away repetitive accessor logic. That framework is so effective that essential Python constructs use it behind the scenes.

Another powerful idea is functions as first class objects, paving the way to higher order functions. As it turns out, the combination of descriptors and higher order functions enable the unification of functions and methods. A function's `__get__` produces a method object on the fly by binding the instance to the `self` argument. This is elegant[8].

[7] Brian Harvey and Matthew Wright, *Simply Scheme* (MIT Press, 1999), p. xvii. Full text available at Berkeley.edu.

[8] *Machine Beauty*, by David Gelernter (Basic Books, 1998), is an intriguing short book about elegance and aesthetics in works of engineering, from bridges to software.

Finally, we have the idea of classes as first class objects. It's an outstanding feat of design that a beginner-friendly language provides powerful abstractions such as class decorators and full-fledged, user defined metaclasses. Best of all: the advanced features are integrated in a way that does not complicate Python's suitability for casual programming — they actually help it, under the covers. The convenience and success of frameworks such as Django, SQLAlchemy owes much to metaclasses, even if many users of these tools aren't aware of them. But they can always learn and create the next great library.

I haven't yet found a language that manages to be easy for beginners, practical for professionals and exciting for hackers in the way that Python is. Thanks, Guido van Rossum and everybody else who makes it so.

Afterword

> Python is a language for consenting adults.
>
> — Alan Runyan
> *co-founder of Plone*

Alan's pithy definition expresses one of the best qualities of Python: it gets out of the way and lets you do what you must. This also means it doesn't give you tools to restrict what others can do with your code and the objects it builds.

Of course, Python is not perfect. Among the top irritants to me is the inconsistent use of `CamelCase`, `snake_case` and `joinedwords` in the standard library. But the language definition and the standard library are only part of an ecosystem. The community of users and contributors is the best part of the Python ecosystem.

Here is one example of the community at its best: one morning while writing about `asyncio` I was frustrated because the API has many functions, dozens of which are coroutines, and you have to call the coroutines with `yield from` but you can't do that with regular functions. This was documented in the `asyncio` pages, but sometimes you had to read a few paragraphs to find out whether a particular function was a coroutine. So I sent a message to `python-tulip` titled Proposal: make coroutines stand out in the asyncio docs. Victor Stinner, an `asyncio` core developer, Andrew Svetlov, main author of `aiohttp`, Ben Darnell, lead developer of Tornado, and Glyph Lefkowitz, inventor of Twisted, joined the conversation. Darnell suggested a solution, Alexander Shorin explained how to implement it in Sphinx, and Stinner added the necessary configuration and markup. Less than 12 hours after I raised the issue, the entire `asyncio` documentation set online was updated with the *coroutine* tags you can see today.

That story did not happen in an exclusive club. Anybody can join the `python-tulip` list, and I had posted only a few times when I wrote the proposal. The story illustrates a community that is really open to new ideas and new members. Guido van Rossum hangs out in `python-tulip` and can regularly be seen answering even simple questions.

Another example of openness: the PSF (Python Software Foundation) has been working to increase diversity in the Python community. Some encouraging results are already in. The 2013-2014 PSF board saw the first women elected directors: Jessica McKellar and Lynn Root. And in the 2015 PyCon North America in Montréal — chaired by Diana Clarke — about 1/3 of the speakers were women. I am unaware of any other major IT conference that has gone so far in the pursuit of gender equality.

If you are a Pythonista but you have not engaged with the community I encourage you to do so. Seek the PUG (Python Users Group) in your area. If there isn't one, create it. Python is everywhere, so you will not be alone. Travel to events if you can. Come to a PythonBrasil conference — we've had international speakers regularly for many years now. Meeting fellow Pythonistas in person beats any online interaction and is known to bring real benefits besides all the knowledge sharing. Like real jobs and real friendships.

I know I could not have written this book without the help of many friends I made over the years in the Python community.

My father Jairo Ramalho used to say "Só erra quem trabalha" — Portuguese for "Only those who work make mistakes" — great advice to avoid being paralyzed by the fear of making errors. I certainly made my share of mistakes while writing this book. The reviewers, editors and Early Release readers caught many of them. Within hours of the first Early Release a reader was reporting typos in the errata page for the book. Other readers contributed more reports, and friends contacted me directly to offer suggestions and corrections. The O'Reilly copyeditors will catch other errors during the production process which will start as soon as I manage to stop writing. I take responsibility and apologize for the errors and sub-optimal prose that remains.

I am very happy to bring this work to conclusion, mistakes and all, and I am very grateful to everybody who helped along the way.

I hope to see you soon at some live event. Please come say hi if you see me around!

Further reading

I will wrap up the book with references regarding what it its to be "Pythonic" — the main question this book tried to address.

Brandon Rhodes is an awesome Python teacher, and his talk A Python Æsthetic: Beauty and Why I Python is beautiful, starting with the use of Unicode U+00C6 (`LATIN CAPITAL LETTER AE`) in the title. Another awesome teacher, Raymond Hettinger, spoke of beauty in Python at PyCon US 2013: Transforming Code into Beautiful, Idiomatic Python.

Ian Lee started a thread on `python-ideas` — Evolution of Style Guides that is worth reading. Lee is the maintainer of the `pep8` package that checks Python source code for

PEP 8 compliance. To check the code in this book I used `flake8` which wraps `pep8`, `pyflakes` and Ned Batchelder's McCabe complexity plugin.

Besides PEP 8, other influential style guides are the Google Python Style Guide an the Pocoo style guide, from the team who brings us Flake, Sphinx, Jinja 2 and other great Python libraries.

The Hitchhiker's Guide to Python! is a collective work about writing Pythonic code. Its most prolific contributor is Kenneth Reitz, a community hero thanks to his beautifully Pythonic `requests` package. David Goodger presented a tutorial at PyCon US 2008 titled Code Like a Pythonista: Idiomatic Python. If printed, the tutorial notes are 30 pages long. Of course, the reStructuredText source is available and can rendered to HTML and S5 slides by docutils. After all, Goodger created both reStructuredText and `docutils` — the foundations of Sphinx, Python's excellent documentation system (which, by the way, is also the official documentation system for MongoDB and many other projects).

Martijn Faassen tackles the question head-on in What is Pythonic? In the `python-list` there is a thread with that same title. Martijn's post is from 2005, and the thread from 2003, but the Pythonic ideal hasn't changed much — neither has the language, for that matter. A great thread with "Pythonic" in the title is Pythonic way to sum n-th list element?, from which I quoted extensively in "Soapbox" on page 304.

PEP 3099 — Things that will Not Change in Python 3000 explains why many things are the way they are, even after the major overhaul that was Python 3. For a long time, Python 3 was nicknamed Python 3000, but it arrived a few centuries sooner — to the dismay of some. PEP 3099 was written by Georg Brandl, compiling many opinions expressed by the *BDFL*, Guido van Rossum. The Python Essays page lists several texts by Guido himself.

APPENDIX A
Support scripts

Here are full listings for some scripts that were too long to fit in the main text. Also included are scripts used to generate some of the tables and data fixtures used in this book.

These scripts are also available at the fluentpython/example-code repository on Github, along with almost every other code snippet that appears in the boook.

Chapter 3: in operator performance test

Example A-1 is the code I used to produce the timings in Table 3-6 using the timeit module. The script mostly deals with setting up the haystack and needles samples and with formatting output.

While coding Example A-1, I found something that really puts dict performance in perspective. If the script is run in "verbose mode" (with the -v command-line option), the timings I get are nearly twice those in Table 3-5. But note that, in this script, "verbose mode" means only four calls to print while setting up the test, and one additional print to show the number of needles found when each test finishes. No output happens within the loop that does the actual search of the needles in the haystack, but these five print calls take about as much time as searching for 1,000 needles.

Example A-1. container_perftest.py: run it with the name of a built-in collection type as a command-line argument, e.g.: container_perftest.py dict.

```
"""
Container ``in`` operator performance test
"""
import sys
import timeit

SETUP = '''
```

```
import array
selected = array.array('d')
with open('selected.arr', 'rb') as fp:
    selected.fromfile(fp, {size})
if {container_type} is dict:
    haystack = dict.fromkeys(selected, 1)
else:
    haystack = {container_type}(selected)
if {verbose}:
    print(type(haystack), end='  ')
    print('haystack: %10d' % len(haystack), end='  ')
needles = array.array('d')
with open('not_selected.arr', 'rb') as fp:
    needles.fromfile(fp, 500)
needles.extend(selected[::{size}//500])
if {verbose}:
    print(' needles: %10d' % len(needles), end='  ')
'''

TEST = '''
found = 0
for n in needles:
    if n in haystack:
        found += 1
if {verbose}:
    print('  found: %10d' % found)
'''

def test(container_type, verbose):
    MAX_EXPONENT = 7
    for n in range(3, MAX_EXPONENT + 1):
        size = 10**n
        setup = SETUP.format(container_type=container_type,
                             size=size, verbose=verbose)
        test = TEST.format(verbose=verbose)
        tt = timeit.repeat(stmt=test, setup=setup, repeat=5, number=1)
        print('|{:{}d}|{:f}'.format(size, MAX_EXPONENT + 1, min(tt)))

if __name__=='__main__':
    if '-v' in sys.argv:
        sys.argv.remove('-v')
        verbose = True
    else:
        verbose = False
    if len(sys.argv) != 2:
        print('Usage: %s <container_type>' % sys.argv[0])
    else:
        test(sys.argv[1], verbose)
```

Example A-2. `container_perftest_datagen.py`: generate files with arrays of unique floating point numbers for use in Example A-1.

```
"""
Generate data for container performance test
"""

import random
import array

MAX_EXPONENT = 7
HAYSTACK_LEN = 10 ** MAX_EXPONENT
NEEDLES_LEN = 10 ** (MAX_EXPONENT - 1)
SAMPLE_LEN = HAYSTACK_LEN + NEEDLES_LEN // 2

needles = array.array('d')

sample = {1/random.random() for i in range(SAMPLE_LEN)}
print('initial sample: %d elements' % len(sample))

# complete sample, in case duplicate random numbers were discarded
while len(sample) < SAMPLE_LEN:
    sample.add(1/random.random())

print('complete sample: %d elements' % len(sample))

sample = array.array('d', sample)
random.shuffle(sample)

not_selected = sample[:NEEDLES_LEN // 2]
print('not selected: %d samples' % len(not_selected))
print('  writing not_selected.arr')
with open('not_selected.arr', 'wb') as fp:
    not_selected.tofile(fp)

selected = sample[NEEDLES_LEN // 2:]
print('selected: %d samples' % len(selected))
print('  writing selected.arr')
with open('selected.arr', 'wb') as fp:
    selected.tofile(fp)
```

Chapter 3: Compare the bit patterns of hashes

This is a simple script to visually show how different are the bit patterns for the hashes of similiar floating point numbers (eg. 1.0001, 1.0002 etc.). Its output appears in Example 3-16.

Example A-3. `hashdiff.py`: display the difference of bit paterns from hash values.

```
import sys
```

```
MAX_BITS = len(format(sys.maxsize, 'b'))
print('%s-bit Python build' % (MAX_BITS + 1))

def hash_diff(o1, o2):
    h1 = '{:>0{}b}'.format(hash(o1), MAX_BITS)
    h2 = '{:>0{}b}'.format(hash(o2), MAX_BITS)
    diff = ''.join('!' if b1 != b2 else ' ' for b1, b2 in zip(h1, h2))
    count = '!= {}'.format(diff.count('!'))
    width = max(len(repr(o1)), len(repr(o2)), 8)
    sep = '-' * (width * 2 + MAX_BITS)
    return '{!r:{width}} {}\n{:{width}} {} {}\n{!r:{width}} {}\n{}'.format(
                o1, h1, ' ' * width, diff, count, o2, h2, sep, width=width)

if __name__ == '__main__':
    print(hash_diff(1, 1.0))
    print(hash_diff(1.0, 1.0001))
    print(hash_diff(1.0001, 1.0002))
    print(hash_diff(1.0002, 1.0003))
```

Chapter 9: RAM usage with and without `__slots__`

The `memtest.py` script was used for a demostration in "Saving space with the `__slots__` class attribute" on page 265: Example 9-12.

The `memtest.py` script takes a module name in the command line and loads it. Assuming the module defines a class named `Vector`, `memtest.py` creates a list with 10 million instances, reporting the memory usage before and after the list is created.

Example A-4. `memtest.py`: create lots of `Vector` instances reporting memory usage.

```
import importlib
import sys
import resource

NUM_VECTORS = 10**7

if len(sys.argv) == 2:
    module_name = sys.argv[1].replace('.py', '')
    module = importlib.import_module(module_name)
else:
    print('Usage: {} <vector-module-to-test>'.format())
    sys.exit(1)

fmt = 'Selected Vector2d type: {.__name__}.{.__name__}'
print(fmt.format(module, module.Vector2d))

mem_init = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
print('Creating {:,} Vector2d instances'.format(NUM_VECTORS))

vectors = [module.Vector2d(3.0, 4.0) for i in range(NUM_VECTORS)]
```

```
mem_final = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
print('Initial RAM usage: {:14,}'.format(mem_init))
print('  Final RAM usage: {:14,}'.format(mem_final))
```

Chatper 14: `isis2json.py` database conversion script

This is the `isis2json.py` script discussed in "Case study: generators in a database conversion utility" on page 439 (Chapter 14). It uses generator functions to lazily convert CDS/ISIS databases to JSON for loading to CouchDB or MongoDB.

Note that this is a Python 2 script, designed to run on CPython or Jython, versions 2.5 to 2.7, but not on Python 3. Under CPython it can read only `.iso` files; with Jython it can also read `.mst` files, using the `Bruma` library available on the fluentpython/isis2json repository in Github. See usage documentation in that repository.

Example A-5. `isis2json.py`: dependencies and documentation available on Github repository fluentpython/isis2json

```
# this script works with Python or Jython (versions >=2.5 and <3)

import sys
import argparse
from uuid import uuid4
import os

try:
    import json
except ImportError:
    if os.name == 'java':  # running Jython
        from com.xhaus.jyson import JysonCodec as json
    else:
        import simplejson as json

SKIP_INACTIVE = True
DEFAULT_QTY = 2**31
ISIS_MFN_KEY = 'mfn'
ISIS_ACTIVE_KEY = 'active'
SUBFIELD_DELIMITER = '^'
INPUT_ENCODING = 'cp1252'


def iter_iso_records(iso_file_name, isis_json_type):   ❶
    from iso2709 import IsoFile
    from subfield import expand

    iso = IsoFile(iso_file_name)
    for record in iso:
        fields = {}
        for field in record.directory:
            field_key = str(int(field.tag))  # remove leading zeroes
            field_occurrences = fields.setdefault(field_key, [])
            content = field.value.decode(INPUT_ENCODING, 'replace')
            if isis_json_type == 1:
                field_occurrences.append(content)
            elif isis_json_type == 2:
                field_occurrences.append(expand(content))
            elif isis_json_type == 3:
                field_occurrences.append(dict(expand(content)))
            else:
                raise NotImplementedError('ISIS-JSON type %s conversion '
                    'not yet implemented for .iso input' % isis_json_type)

        yield fields
    iso.close()


def iter_mst_records(master_file_name, isis_json_type):  ❷
    try:
        from bruma.master import MasterFactory, Record
    except ImportError:
        print('IMPORT ERROR: Jython 2.5 and Bruma.jar '
              'are required to read .mst files')
        raise SystemExit
    mst = MasterFactory.getInstance(master_file_name).open()
    for record in mst:
        fields = {}
        if SKIP_INACTIVE:
            if record.getStatus() != Record.Status.ACTIVE:
                continue
        else:  # save status only there are non-active records
            fields[ISIS_ACTIVE_KEY] = (record.getStatus() ==
                                       Record.Status.ACTIVE)
        fields[ISIS_MFN_KEY] = record.getMfn()
        for field in record.getFields():
            field_key = str(field.getId())
            field_occurrences = fields.setdefault(field_key, [])
            if isis_json_type == 3:
                content = {}
                for subfield in field.getSubfields():
                    subfield_key = subfield.getId()
                    if subfield_key == '*':
                        content['_'] = subfield.getContent()
                    else:
                        subfield_occurrences = content.setdefault(subfield_key, [])
                        subfield_occurrences.append(subfield.getContent())
                field_occurrences.append(content)
            elif isis_json_type == 1:
                content = []
                for subfield in field.getSubfields():
                    subfield_key = subfield.getId()
                    if subfield_key == '*':
                        content.insert(0, subfield.getContent())
                    else:
                        content.append(SUBFIELD_DELIMITER + subfield_key +
                                       subfield.getContent())
                field_occurrences.append(''.join(content))
            else:
                raise NotImplementedError('ISIS-JSON type %s conversion '
                    'not yet implemented for .mst input' % isis_json_type)
        yield fields
    mst.close()


def write_json(input_gen, file_name, output, qty, skip, id_tag,   ❸
               gen_uuid, mongo, mfn, isis_json_type, prefix,
               constant):
    start = skip
    end = start + qty
    if id_tag:
        id_tag = str(id_tag)
        ids = set()
    else:
        id_tag = ''
    for i, record in enumerate(input_gen):
        if i >= end:
            break
        if not mongo:
            if i == 0:
                output.write('[')
            elif i > start:
                output.write(',')
        if start <= i < end:
            if id_tag:
                occurrences = record.get(id_tag, None)
                if occurrences is None:
                    msg = 'id tag #%s not found in record %s'
                    if ISIS_MFN_KEY in record:
                        msg = msg + (' (mfn=%s)' % record[ISIS_MFN_KEY])
                    raise KeyError(msg % (id_tag, i))
                if len(occurrences) > 1:
                    msg = 'multiple id tags #%s found in record %s'
                    if ISIS_MFN_KEY in record:
                        msg = msg + (' (mfn=%s)' % record[ISIS_MFN_KEY])
                    raise TypeError(msg % (id_tag, i))
                else:  # ok, we have one and only one id field
                    if isis_json_type == 1:
                        id = occurrences[0]
                    elif isis_json_type == 2:
                        id = occurrences[0][0][1]
                    elif isis_json_type == 3:
                        id = occurrences[0]['_']
                    if id in ids:
                        msg = 'duplicate id %s in tag #%s, record %s'
                        if ISIS_MFN_KEY in record:
                            msg = msg + (' (mfn=%s)' % record[ISIS_MFN_KEY])
                        raise TypeError(msg % (id, id_tag, i))
                    record['_id'] = id
                    ids.add(id)
            elif gen_uuid:
                record['_id'] = unicode(uuid4())
            elif mfn:
                record['_id'] = record[ISIS_MFN_KEY]
            if prefix:
                # iterate over a fixed sequence of tags
                for tag in tuple(record):
                    if str(tag).isdigit():
                        record[prefix+tag] = record[tag]
                        del record[tag]  # this is why we iterate over a tuple
                        # with the tags, and not directly on the record dict
            if constant:
                constant_key, constant_value = constant.split(':')
                record[constant_key] = constant_value
            output.write(json.dumps(record).encode('utf-8'))
            output.write('\n')
    if not mongo:
        output.write(']\n')


def main():  ❹
    # create the parser
    parser = argparse.ArgumentParser(
        description='Convert an ISIS .mst or .iso file to a JSON array')

    # add the arguments
    parser.add_argument(
        'file_name', metavar='INPUT.(mst|iso)',
        help='.mst or .iso file to read')
    parser.add_argument(
        '-o', '--out', type=argparse.FileType('w'), default=sys.stdout,
        metavar='OUTPUT.json',
        help='the file where the JSON output should be written'
             ' (default: write to stdout)')
    parser.add_argument(
        '-c', '--couch', action='store_true',
        help='output array within a "docs" item in a JSON document'
             ' for bulk insert to CouchDB via POST to db/_bulk_docs')
    parser.add_argument(
        '-m', '--mongo', action='store_true',
        help='output individual records as separate JSON dictionaries,'
             ' one per line for bulk insert to MongoDB via mongoimport utility')
    parser.add_argument(
        '-t', '--type', type=int, metavar='ISIS_JSON_TYPE', default=1,
        help='ISIS-JSON type, sets field structure: 1=string, 2=alist, 3=dict (default=1)')
    parser.add_argument(
        '-q', '--qty', type=int, default=DEFAULT_QTY,
        help='maximum quantity of records to read (default=ALL)')
    parser.add_argument(
        '-s', '--skip', type=int, default=0,
        help='records to skip from start of .mst (default=0)')
    parser.add_argument(
        '-i', '--id', type=int, metavar='TAG_NUMBER', default=0,
        help='generate an "_id" from the given unique TAG field number'
             ' for each record')
    parser.add_argument(
        '-u', '--uuid', action='store_true',
        help='generate an "_id" with a random UUID for each record')
    parser.add_argument(
        '-p', '--prefix', type=str, metavar='PREFIX', default='',
        help='concatenate prefix to every numeric field tag (ex. 99 becomes "v99")')
    parser.add_argument(
        '-n', '--mfn', action='store_true',
        help='generate an "_id" from the MFN of each record'
             ' (available only for .mst input)')
    parser.add_argument(
        '-k', '--constant', type=str, metavar='TAG:VALUE', default='',
        help='Include a constant tag:value in every record (ex. -k type:AS)')

    '''
    # TODO: implement this to export large quantities of records to CouchDB
    parser.add_argument(
        '-r', '--repeat', type=int, default=1,
        help='repeat operation, saving multiple JSON files'
             ' (default=1, use -r 0 to repeat until end of input)')
    '''
    # parse the command line
    args = parser.parse_args()
    if args.file_name.lower().endswith('.mst'):
        input_gen_func = iter_mst_records  ❺
    else:
        if args.mfn:
            print('UNSUPORTED: -n/--mfn option only available for .mst input.')
            raise SystemExit
        input_gen_func = iter_iso_records  ❻
    input_gen = input_gen_func(args.file_name, args.type)  ❼
    if args.couch:
        args.out.write('{ "docs" : ')
    write_json(input_gen, args.file_name, args.out, args.qty,  ❽
               args.skip, args.id, args.uuid, args.mongo, args.mfn,
               args.type, args.prefix, args.constant)
    if args.couch:
        args.out.write('}\n')
    args.out.close()


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

❶ `iter_iso_records` generator function reads `.ISO` file, yields records.

❷ `iter_mst_records` generator function reads `.MST` file, yields records.

❸ `write_json` iterates over `input_gen` generator and outputs the `.JSON` file.

❹ Main function reads command line arguments then…

❺ …selects `iter_iso_records` or…

❻ …`iter_mst_records` depending on input file extension.

❼ A generator object is built from the selected generator function.

❽ `write_json` is called with the generator as the first argument.

Chapter 16: Taxi fleet discrete event simulation

This is the full listing for `taxi_sim.py` discussed in "The taxi fleet simulation" on page 492.

Example A-6. `taxi_sim.py`: The taxi fleed simulator.

```
"""
Taxi simulator
==============

Driving a taxi from the console::

    >>> from taxi_sim import taxi_process
    >>> taxi = taxi_process(ident=13, trips=2, start_time=0)
    >>> next(taxi)
    Event(time=0, proc=13, action='leave garage')
    >>> taxi.send(_.time + 7)
    Event(time=7, proc=13, action='pick up passenger')
    >>> taxi.send(_.time + 23)
    Event(time=30, proc=13, action='drop off passenger')
    >>> taxi.send(_.time + 5)
    Event(time=35, proc=13, action='pick up passenger')
    >>> taxi.send(_.time + 48)
    Event(time=83, proc=13, action='drop off passenger')
    >>> taxi.send(_.time + 1)
    Event(time=84, proc=13, action='going home')
    >>> taxi.send(_.time + 10)
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    StopIteration

Sample run with two cars, random seed 10. This is a valid doctest::

    >>> main(num_taxis=2, seed=10)
    taxi: 0  Event(time=0, proc=0, action='leave garage')
    taxi: 0  Event(time=5, proc=0, action='pick up passenger')
    taxi: 1     Event(time=5, proc=1, action='leave garage')
    taxi: 1     Event(time=10, proc=1, action='pick up passenger')
    taxi: 1     Event(time=15, proc=1, action='drop off passenger')
    taxi: 0  Event(time=17, proc=0, action='drop off passenger')
    taxi: 1     Event(time=24, proc=1, action='pick up passenger')
    taxi: 0  Event(time=26, proc=0, action='pick up passenger')
    taxi: 0  Event(time=30, proc=0, action='drop off passenger')
    taxi: 0  Event(time=34, proc=0, action='going home')
    taxi: 1     Event(time=46, proc=1, action='drop off passenger')
    taxi: 1     Event(time=48, proc=1, action='pick up passenger')
    taxi: 1     Event(time=110, proc=1, action='drop off passenger')
    taxi: 1     Event(time=139, proc=1, action='pick up passenger')
    taxi: 1     Event(time=140, proc=1, action='drop off passenger')
    taxi: 1     Event(time=150, proc=1, action='going home')
    *** end of events ***

See longer sample run at the end of this module.

"""

import random
import collections
import queue
import argparse
import time

DEFAULT_NUMBER_OF_TAXIS = 3
DEFAULT_END_TIME = 180
SEARCH_DURATION = 5
TRIP_DURATION = 20
DEPARTURE_INTERVAL = 5

Event = collections.namedtuple('Event', 'time proc action')


# BEGIN TAXI_PROCESS
def taxi_process(ident, trips, start_time=0):  # ❶
    """Yield to simulator issuing event at each state change"""
    time = yield Event(start_time, ident, 'leave garage')  # ❷
    for i in range(trips):  # ❸
        time = yield Event(time, ident, 'pick up passenger')  # ❹
        time = yield Event(time, ident, 'drop off passenger')  # ❺

    yield Event(time, ident, 'going home')  # ❻
    # end of taxi process # ❼
# END TAXI_PROCESS


# BEGIN TAXI_SIMULATOR
class Simulator:

    def __init__(self, procs_map):
        self.events = queue.PriorityQueue()
        self.procs = dict(procs_map)
    def run(self, end_time):  # ❽
        """Schedule and display events until time is up"""
        # schedule the first event for each cab
        for _, proc in sorted(self.procs.items()):  # ❾
            first_event = next(proc)  # ❿
            self.events.put(first_event)  # ⓫

        # main loop of the simulation
        sim_time = 0  # ⓬
        while sim_time < end_time:  # ⓭
            if self.events.empty():  # ⓮
                print('*** end of events ***')
                break

            current_event = self.events.get()  # ⓯
            sim_time, proc_id, previous_action = current_event  # ⓰
            print('taxi:', proc_id, proc_id * '   ', current_event)  # ⓱
            active_proc = self.procs[proc_id]  # ⓲
            next_time = sim_time + compute_duration(previous_action)  # ⓳
            try:
                next_event = active_proc.send(next_time)  # ⓴
            except StopIteration:
                del self.procs[proc_id]  # ㉑
            else:
                self.events.put(next_event)  # ㉒
        else:  # ㉓
            msg = '*** end of simulation time: {} events pending ***'
            print(msg.format(self.events.qsize()))
# END TAXI_SIMULATOR


def compute_duration(previous_action):
    """Compute action duration using exponential distribution"""
    if previous_action in ['leave garage', 'drop off passenger']:
        # new state is prowling
        interval = SEARCH_DURATION
    elif previous_action == 'pick up passenger':
        # new state is trip
        interval = TRIP_DURATION
    elif previous_action == 'going home':
        interval = 1
    else:
        raise ValueError('Unknown previous_action: %s' % previous_action)
    return int(random.expovariate(1/interval)) + 1


def main(end_time=DEFAULT_END_TIME, num_taxis=DEFAULT_NUMBER_OF_TAXIS,
         seed=None):
    """Initialize random generator, build procs and run simulation"""
    if seed is not None:
        random.seed(seed)  # get reproducible results
    taxis = {i: taxi_process(i, (i+1)*2, i*DEPARTURE_INTERVAL)
             for i in range(num_taxis)}
    sim = Simulator(taxis)
    sim.run(end_time)


if __name__ == '__main__':

    parser = argparse.ArgumentParser(
                        description='Taxi fleet simulator.')
    parser.add_argument('-e', '--end-time', type=int,
                        default=DEFAULT_END_TIME,
                        help='simulation end time; default = %s'
                        % DEFAULT_END_TIME)
    parser.add_argument('-t', '--taxis', type=int,
                        default=DEFAULT_NUMBER_OF_TAXIS,
                        help='number of taxis running; default = %s'
                        % DEFAULT_NUMBER_OF_TAXIS)
    parser.add_argument('-s', '--seed', type=int, default=None,
                        help='random generator seed (for testing)')

    args = parser.parse_args()
    main(args.end_time, args.taxis, args.seed)


"""

Sample run from the command line, seed=3, maximum elapsed time=120::

# BEGIN TAXI_SAMPLE_RUN
$ python3 taxi_sim.py -s 3 -e 120
taxi: 0  Event(time=0, proc=0, action='leave garage')
taxi: 0  Event(time=2, proc=0, action='pick up passenger')
taxi: 1     Event(time=5, proc=1, action='leave garage')
taxi: 1     Event(time=8, proc=1, action='pick up passenger')
taxi: 2        Event(time=10, proc=2, action='leave garage')
taxi: 2        Event(time=15, proc=2, action='pick up passenger')
taxi: 2        Event(time=17, proc=2, action='drop off passenger')
taxi: 0  Event(time=18, proc=0, action='drop off passenger')
taxi: 2        Event(time=18, proc=2, action='pick up passenger')
taxi: 2        Event(time=25, proc=2, action='drop off passenger')
taxi: 1     Event(time=27, proc=1, action='drop off passenger')
taxi: 2        Event(time=27, proc=2, action='pick up passenger')
taxi: 0  Event(time=28, proc=0, action='pick up passenger')
taxi: 2        Event(time=40, proc=2, action='drop off passenger')
taxi: 2        Event(time=44, proc=2, action='pick up passenger')
taxi: 1     Event(time=55, proc=1, action='pick up passenger')
taxi: 1     Event(time=59, proc=1, action='drop off passenger')
taxi: 0  Event(time=65, proc=0, action='drop off passenger')
taxi: 1     Event(time=65, proc=1, action='pick up passenger')
taxi: 2        Event(time=65, proc=2, action='drop off passenger')
```

```
taxi: 2        Event(time=72, proc=2, action='pick up passenger')
taxi: 0  Event(time=76, proc=0, action='going home')
taxi: 1     Event(time=80, proc=1, action='drop off passenger')
taxi: 1     Event(time=88, proc=1, action='pick up passenger')
taxi: 2        Event(time=95, proc=2, action='drop off passenger')
taxi: 2        Event(time=97, proc=2, action='pick up passenger')
taxi: 2        Event(time=98, proc=2, action='drop off passenger')
taxi: 1     Event(time=106, proc=1, action='drop off passenger')
taxi: 2        Event(time=109, proc=2, action='going home')
taxi: 1     Event(time=110, proc=1, action='going home')
*** end of events ***
# END TAXI_SAMPLE_RUN

"""
```

Chapter 17: Cryptographic examples

These scripts were used to show the use of `futures.ProcessPoolExecutor` to run CPU-intensive tasks.

Example A-7 encrypts and decrypts random byte arrays with the RC4 algorithm. It depends on `arcfour.py` module (Example A-8) to run.

Example A-7. `arcfour_futures.py`: `futures.ProcessPoolExecutor` example.

```
import sys
import time
from concurrent import futures
from random import randrange
from arcfour import arcfour

JOBS = 12
SIZE = 2**18

KEY = b"'Twas brillig, and the slithy toves\nDid gyre"
STATUS = '{} workers, elapsed time: {:.2f}s'


def arcfour_test(size, key):
    in_text = bytearray(randrange(256) for i in range(size))
    cypher_text = arcfour(key, in_text)
    out_text = arcfour(key, cypher_text)
    assert in_text == out_text, 'Failed arcfour_test'
    return size


def main(workers=None):
    if workers:
        workers = int(workers)
    t0 = time.time()
    with futures.ProcessPoolExecutor(workers) as executor:
        actual_workers = executor._max_workers
        to_do = []
        for i in range(JOBS, 0, -1):
            size = SIZE + int(SIZE / JOBS * (i - JOBS/2))
            job = executor.submit(arcfour_test, size, KEY)
            to_do.append(job)

        for future in futures.as_completed(to_do):
            res = future.result()
            print('{:.1f} KB'.format(res/2**10))

    print(STATUS.format(actual_workers, time.time() - t0))

if __name__ == '__main__':
    if len(sys.argv) == 2:
        workers = int(sys.argv[1])
    else:
        workers = None
    main(workers)
```

Example A-8 implements the RC4 encryption algorithm in pure Python.

Example A-8. `arcfour.py`: RC4 compatible algorithm

```
"""RC4 compatible algorithm"""

def arcfour(key, in_bytes, loops=20):

    kbox = bytearray(256)  # create key box
    for i, car in enumerate(key):  # copy key and vector
        kbox[i] = car
    j = len(key)
    for i in range(j, 256):  # repeat until full
        kbox[i] = kbox[i-j]

    # [1] initialize sbox
    sbox = bytearray(range(256))

    # repeat sbox mixing loop, as recommened in CipherSaber-2
    # http://ciphersaber.gurus.com/faq.html#cs2
    j = 0
    for k in range(loops):
        for i in range(256):
            j = (j + sbox[i] + kbox[i]) % 256
            sbox[i], sbox[j] = sbox[j], sbox[i]

    # main loop
    i = 0
    j = 0
    out_bytes = bytearray()

    for car in in_bytes:
        i = (i + 1) % 256
        # [2] shuffle sbox
        j = (j + sbox[i]) % 256
        sbox[i], sbox[j] = sbox[j], sbox[i]
        # [3] compute t
        t = (sbox[i] + sbox[j]) % 256
        k = sbox[t]
        car = car ^ k
        out_bytes.append(car)

    return out_bytes


def test():
    from time import time
    clear = bytearray(b'1234567890' * 100000)
    t0 = time()
    cipher = arcfour(b'key', clear)
    print('elapsed time: %.2fs' % (time() - t0))
    result = arcfour(b'key', cipher)
    assert result == clear, '%r != %r' % (result, clear)
    print('elapsed time: %.2fs' % (time() - t0))
    print('OK')


if __name__ == '__main__':
    test()
```

Example A-9 applies the SHA-256 hash algorithm to random byte arrays. It uses `hashlib` from the standard library, which in turn uses the OpenSSL library written in C.

Example A-9. `sha_futures.py`: `futures.ProcessPoolExecutor` example.

```
import sys
import time
import hashlib
from concurrent import futures
from random import randrange

JOBS = 12
SIZE = 2**20
STATUS = '{} workers, elapsed time: {:.2f}s'


def sha(size):
    data = bytearray(randrange(256) for i in range(size))
    algo = hashlib.new('sha256')
    algo.update(data)
    return algo.hexdigest()


def main(workers=None):
    if workers:
        workers = int(workers)
    t0 = time.time()

    with futures.ProcessPoolExecutor(workers) as executor:
        actual_workers = executor._max_workers
        to_do = (executor.submit(sha, SIZE) for i in range(JOBS))
        for future in futures.as_completed(to_do):
            res = future.result()
            print(res)

    print(STATUS.format(actual_workers, time.time() - t0))

if __name__ == '__main__':
    if len(sys.argv) == 2:
        workers = int(sys.argv[1])
    else:
        workers = None
    main(workers)
```

Chapter 17: `flags2` HTTP client examples

All `flags2` examples from "Downloads with progress display and error handling" on page 522 use functions from the `flags2_common.py` module Example A-10:

Example A-10. `flags2_common.py`

```
"""Utilities for second set of flag examples.
"""

import os
import time
import sys
import string
import argparse
from collections import namedtuple
from enum import Enum


Result = namedtuple('Result', 'status data')

HTTPStatus = Enum('Status', 'ok not_found error')

POP20_CC = ('CN IN US ID BR PK NG BD RU JP '
            'MX PH VN ET EG DE IR TR CD FR').split()

DEFAULT_CONCUR_REQ = 1
MAX_CONCUR_REQ = 1

SERVERS = {
    'REMOTE': 'http://flupy.org/data/flags',
    'LOCAL':  'http://localhost:8001/flags',
    'DELAY':  'http://localhost:8002/flags',
    'ERROR':  'http://localhost:8003/flags',
}
DEFAULT_SERVER = 'LOCAL'

DEST_DIR = 'downloads/'
COUNTRY_CODES_FILE = 'country_codes.txt'


def save_flag(img, filename):
    path = os.path.join(DEST_DIR, filename)
    with open(path, 'wb') as fp:
        fp.write(img)


def initial_report(cc_list, actual_req, server_label):
    if len(cc_list) <= 10:
        cc_msg = ', '.join(cc_list)
    else:
        cc_msg = 'from {} to {}'.format(cc_list[0], cc_list[-1])
    print('{} site: {}'.format(server_label, SERVERS[server_label]))
    msg = 'Searching for {} flag{}: {}'
    plural = 's' if len(cc_list) != 1 else ''
    print(msg.format(len(cc_list), plural, cc_msg))
    plural = 's' if actual_req != 1 else ''
    msg = '{} concurrent connection{} will be used.'
    print(msg.format(actual_req, plural))


def final_report(cc_list, counter, start_time):
    elapsed = time.time() - start_time
    print('-' * 20)
    msg = '{} flag{} downloaded.'
    plural = 's' if counter[HTTPStatus.ok] != 1 else ''
    print(msg.format(counter[HTTPStatus.ok], plural))
    if counter[HTTPStatus.not_found]:
        print(counter[HTTPStatus.not_found], 'not found.')
    if counter[HTTPStatus.error]:
        plural = 's' if counter[HTTPStatus.error] != 1 else ''
        print('{} error{}.'.format(counter[HTTPStatus.error], plural))
    print('Elapsed time: {:.2f}s'.format(elapsed))


def expand_cc_args(every_cc, all_cc, cc_args, limit):
    codes = set()
    A_Z = string.ascii_uppercase
    if every_cc:
        codes.update(a+b for a in A_Z for b in A_Z)
    elif all_cc:
        with open(COUNTRY_CODES_FILE) as fp:
            text = fp.read()
        codes.update(text.split())
    else:
        for cc in (c.upper() for c in cc_args):
            if len(cc) == 1 and cc in A_Z:
                codes.update(cc+c for c in A_Z)
            elif len(cc) == 2 and all(c in A_Z for c in cc):
                codes.add(cc)
            else:
                msg = 'each CC argument must be A to Z or AA to ZZ.'
                raise ValueError('*** Usage error: '+msg)
    return sorted(codes)[:limit]


def process_args(default_concur_req):
    server_options = ', '.join(sorted(SERVERS))
    parser = argparse.ArgumentParser(
                description='Download flags for country codes. '
                'Default: top 20 countries by population.')
    parser.add_argument('cc', metavar='CC', nargs='*',
                help='country code or 1st letter (eg. B for BA...BZ)')
    parser.add_argument('-a', '--all', action='store_true',
                help='get all available flags (AD to ZW)')
    parser.add_argument('-e', '--every', action='store_true',
                help='get flags for every possible code (AA...ZZ)')
    parser.add_argument('-l', '--limit', metavar='N', type=int,
                help='limit to N first codes', default=sys.maxsize)
    parser.add_argument('-m', '--max_req', metavar='CONCURRENT', type=int,
                default=default_concur_req,
                help='maximum concurrent requests (default={})'
                      .format(default_concur_req))
    parser.add_argument('-s', '--server', metavar='LABEL',
                default=DEFAULT_SERVER,
                help='Server to hit; one of {} (default={})'
                      .format(server_options, DEFAULT_SERVER))
    parser.add_argument('-v', '--verbose', action='store_true',
                help='output detailed progress info')
    args = parser.parse_args()
    if args.max_req < 1:
        print('*** Usage error: --max_req CONCURRENT must be >= 1')
        parser.print_usage()
        sys.exit(1)
    if args.limit < 1:
        print('*** Usage error: --limit N must be >= 1')
        parser.print_usage()
        sys.exit(1)
    args.server = args.server.upper()
    if args.server not in SERVERS:
        print('*** Usage error: --server LABEL must be one of',
              server_options)
        parser.print_usage()
        sys.exit(1)
    try:
        cc_list = expand_cc_args(args.every, args.all, args.cc, args.limit)
    except ValueError as exc:
        print(exc.args[0])
        parser.print_usage()
        sys.exit(1)

    if not cc_list:
        cc_list = sorted(POP20_CC)
    return args, cc_list


def main(download_many, default_concur_req, max_concur_req):
    args, cc_list = process_args(default_concur_req)
    actual_req = min(args.max_req, max_concur_req, len(cc_list))
    initial_report(cc_list, actual_req, args.server)
    base_url = SERVERS[args.server]
    t0 = time.time()
    counter = download_many(cc_list, base_url, args.verbose, actual_req)
    assert sum(counter.values()) == len(cc_list), \
        'some downloads are unaccounted for'
    final_report(cc_list, counter, t0)
```

The `flags2_sequential.py` script (Example A-11) is the baseline for comparison with the concurrent implementations. The `flags2_threadpool.py` (Example 17-14) also uses the `get_flag` and `download_one` functions from `flags2_sequential.py`.

Example A-11. `flags2_sequential.py`

```
"""Download flags of countries (with error handling).

Sequential version

Sample run::

    $ python3 flags2_sequential.py -s DELAY b
    DELAY site: http://localhost:8002/flags
    Searching for 26 flags: from BA to BZ
    1 concurrent connection will be used.
    --------------------
    17 flags downloaded.
    9 not found.
    Elapsed time: 13.36s

"""

import collections

import requests
import tqdm

from flags2_common import main, save_flag, HTTPStatus, Result
```

```
DEFAULT_CONCUR_REQ = 1
MAX_CONCUR_REQ = 1

# BEGIN FLAGS2_BASIC_HTTP_FUNCTIONS
def get_flag(base_url, cc):
    url = '{}/{cc}/{cc}.gif'.format(base_url, cc=cc.lower())
    resp = requests.get(url)
    if resp.status_code != 200:  # ❶
        resp.raise_for_status()
    return resp.content


def download_one(cc, base_url, verbose=False):
    try:
        image = get_flag(base_url, cc)
    except requests.exceptions.HTTPError as exc:  # ❷
        res = exc.response
        if res.status_code == 404:
            status = HTTPStatus.not_found  # ❸
            msg = 'not found'
        else:  # ❹
            raise
    else:
        save_flag(image, cc.lower() + '.gif')
        status = HTTPStatus.ok
        msg = 'OK'

    if verbose:  # ❺
        print(cc, msg)

    return Result(status, cc)  # ❻
# END FLAGS2_BASIC_HTTP_FUNCTIONS

# BEGIN FLAGS2_DOWNLOAD_MANY_SEQUENTIAL
def download_many(cc_list, base_url, verbose, max_req):
    counter = collections.Counter()  # ❼
    cc_iter = sorted(cc_list)  # ❽
    if not verbose:
        cc_iter = tqdm.tqdm(cc_iter)  # ❾
    for cc in cc_iter:  # ❿
        try:
            res = download_one(cc, base_url, verbose)  # ⓫
        except requests.exceptions.HTTPError as exc:  # ⓬
            error_msg = 'HTTP error {res.status_code} - {res.reason}'
            error_msg = error_msg.format(res=exc.response)
        except requests.exceptions.ConnectionError as exc:  # ⓭
            error_msg = 'Connection error'
        else:  # ⓮
            error_msg = ''
            status = res.status
        if error_msg:
            status = HTTPStatus.error  # ⓯
        counter[status] += 1  # ⓰
        if verbose and error_msg: # ⓱
            print('*** Error for {}: {}'.format(cc, error_msg))

    return counter  # ⓲
# END FLAGS2_DOWNLOAD_MANY_SEQUENTIAL

if __name__ == '__main__':
    main(download_many, DEFAULT_CONCUR_REQ, MAX_CONCUR_REQ)
```

Chapter 19: OSCON schedule scripts and tests

Example A-12 is the test script for `schedule1.py` module (Example 19-9). It uses the `py.test` library and test runner.

Example A-12. `test_schedule1.py`

```
import shelve
import pytest

import schedule1 as schedule


@pytest.yield_fixture
def db():
    with shelve.open(schedule.DB_NAME) as the_db:
        if schedule.CONFERENCE not in the_db:
            schedule.load_db(the_db)
        yield the_db


def test_record_class():
    rec = schedule.Record(spam=99, eggs=12)
    assert rec.spam == 99
    assert rec.eggs == 12


def test_conference_record(db):
    assert schedule.CONFERENCE in db


def test_speaker_record(db):
    speaker = db['speaker.3471']
    assert speaker.name == 'Anna Martelli Ravenscroft'


def test_event_record(db):
    event = db['event.33950']
    assert event.name == 'There *Will* Be Bugs'
```

```
def test_event_venue(db):
    event = db['event.33950']
    assert event.venue_serial == 1449
```

Example A-13 is the full listing of the schedule2.py example presented in “Linked record retrieval with properties” on page 600 in four parts.

Example A-13. schedule2.py

```
"""
schedule2.py: traversing OSCON schedule data

    >>> import shelve
    >>> db = shelve.open(DB_NAME)
    >>> if CONFERENCE not in db: load_db(db)

# BEGIN SCHEDULE2_DEMO

    >>> DbRecord.set_db(db)  # ❶
    >>> event = DbRecord.fetch('event.33950')  # ❷
    >>> event  # ❸
    <Event 'There *Will* Be Bugs'>
    >>> event.venue  # ❹
    <DbRecord serial='venue.1449'>
    >>> event.venue.name  # ❺
    'Portland 251'
    >>> for spkr in event.speakers:  # ❻
    ...     print('{0.serial}: {0.name}'.format(spkr))
    ...
    speaker.3471: Anna Martelli Ravenscroft
    speaker.5199: Alex Martelli

# END SCHEDULE2_DEMO

    >>> db.close()

"""

# BEGIN SCHEDULE2_RECORD
import warnings
import inspect  # ❼

import osconfeed

DB_NAME = 'data/schedule2_db'  # ❽
CONFERENCE = 'conference.115'


class Record:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)
    def __eq__(self, other):  # ❾
        if isinstance(other, Record):
            return self.__dict__ == other.__dict__
        else:
            return NotImplemented
# END SCHEDULE2_RECORD


# BEGIN SCHEDULE2_DBRECORD
class MissingDatabaseError(RuntimeError):
    """Raised when a database is required but was not set."""  # ❿


class DbRecord(Record):  # ⓫

    __db = None  # ⓬

    @staticmethod  # ⓭
    def set_db(db):
        DbRecord.__db = db  # ⓮

    @staticmethod  # ⓯
    def get_db():
        return DbRecord.__db

    @classmethod  # ⓰
    def fetch(cls, ident):
        db = cls.get_db()
        try:
            return db[ident]  # ⓱
        except TypeError:
            if db is None:  # ⓲
                msg = "database not set; call '{}.set_db(my_db)'"
                raise MissingDatabaseError(msg.format(cls.__name__))
            else:  # ⓳
                raise

    def __repr__(self):
        if hasattr(self, 'serial'):  # ⓴
            cls_name = self.__class__.__name__
            return '<{} serial={!r}>'.format(cls_name, self.serial)
        else:
            return super().__repr__()  # ㉑
# END SCHEDULE2_DBRECORD


# BEGIN SCHEDULE2_EVENT
class Event(DbRecord):  # ㉒

    @property
    def venue(self):
        key = 'venue.{}'.format(self.venue_serial)
        return self.__class__.fetch(key)  # ㉓

    @property
    def speakers(self):
        if not hasattr(self, '_speaker_objs'):  # ㉔
            spkr_serials = self.__dict__['speakers']  # ㉕
            fetch = self.__class__.fetch  # ㉖
            self._speaker_objs = [fetch('speaker.{}'.format(key))
                                  for key in spkr_serials]  # ㉗
        return self._speaker_objs  # ㉘

    def __repr__(self):
        if hasattr(self, 'name'):  # ㉙
            cls_name = self.__class__.__name__
            return '<{} {!r}>'.format(cls_name, self.name)
        else:
            return super().__repr__()  # ㉚
# END SCHEDULE2_EVENT


# BEGIN SCHEDULE2_LOAD
def load_db(db):
    raw_data = osconfeed.load()
    warnings.warn('loading ' + DB_NAME)
    for collection, rec_list in raw_data['Schedule'].items():
        record_type = collection[:-1]  # ㉛
        cls_name = record_type.capitalize()  # ㉜
        cls = globals().get(cls_name, DbRecord)  # ㉝
        if inspect.isclass(cls) and issubclass(cls, DbRecord):  # ㉞
            factory = cls  # ㉟
        else:
            factory = DbRecord  # ㊱
        for record in rec_list:  # ㊲
            key = '{}.{}'.format(record_type, record['serial'])
            record['serial'] = key
            db[key] = factory(**record)  # ㊳
# END SCHEDULE2_LOAD
```

Example A-14 was used to test Example A-13 with `py.test`.

Example A-14. test_schedule2.py

```
import shelve
import pytest

import schedule2 as schedule


@pytest.yield_fixture
def db():
    with shelve.open(schedule.DB_NAME) as the_db:
        if schedule.CONFERENCE not in the_db:
            schedule.load_db(the_db)
        yield the_db


def test_record_attr_access():
    rec = schedule.Record(spam=99, eggs=12)
    assert rec.spam == 99
    assert rec.eggs == 12


def test_record_repr():
    rec = schedule.DbRecord(spam=99, eggs=12)
    assert 'DbRecord object at 0x' in repr(rec)
    rec2 = schedule.DbRecord(serial=13)
    assert repr(rec2) == "<DbRecord serial=13>"


def test_conference_record(db):
    assert schedule.CONFERENCE in db


def test_speaker_record(db):
    speaker = db['speaker.3471']
    assert speaker.name == 'Anna Martelli Ravenscroft'


def test_missing_db_exception():
    with pytest.raises(schedule.MissingDatabaseError):
        schedule.DbRecord.fetch('venue.1585')


def test_dbrecord(db):
    schedule.DbRecord.set_db(db)
    venue = schedule.DbRecord.fetch('venue.1585')
    assert venue.name == 'Exhibit Hall B'


def test_event_record(db):
    event = db['event.33950']
    assert repr(event) == "<Event 'There *Will* Be Bugs'>"


def test_event_venue(db):
    schedule.Event.set_db(db)
    event = db['event.33950']
    assert event.venue_serial == 1449
    assert event.venue == db['venue.1449']
    assert event.venue.name == 'Portland 251'


def test_event_speakers(db):
    schedule.Event.set_db(db)
    event = db['event.33950']
    assert len(event.speakers) == 2
    anna_and_alex = [db['speaker.3471'], db['speaker.5199']]
    assert event.speakers == anna_and_alex


def test_event_no_speakers(db):
    schedule.Event.set_db(db)
    event = db['event.36848']
    assert len(event.speakers) == 0
```

Python jargon

Many terms here are not exclusive to Python of course, but particularly in the definitions you may find meanings that are specific to the Python community.

Also see the official Python glossary.

ABC

A programming language created by Leo Geurts, Lambert Meertens and Steven Pemberton. Guido van Rossum worked as a programmer implementing the ABC environment in the 1980s. Block structuring by indentation, built-in tuples and dictionaries, tuple unpacking, the semantics of the `for` loop and uniform handling of all sequence types are some of the distinctive characteristics of Python that came from ABC.

ABC

Abstract Base Class. A class that cannot be instantiated, only subclassed. ABCs are how interfaces are formalized in Python. Instead of inheriting from an ABC, a class may also declare that it fulfills the interface by registering with the ABC to become a *virtual subclass*.

accessor

A method implemented to provide access to a single data attribute. Some authors use *acessor* as a generic term encompassing getter and setter methods, others use it to refer only to getters, referring to setters as mutators.

aliasing

Assigning two or more names to the same object. For example, in `a = []; b = a` the variables `a` and `b` are aliases for the same list object. Aliasing happens naturally all the time in any language where variables store references to objects. To avoid confusion, just forget the idea that variables are boxes that hold objects (an object can't be in two boxes at the same time). It's better to think of them as labels attached to objects (an object can have more than one label).

argument

An expression passed to a function when it is called. In Pythonic parlance, *argument* and *parameter* are almost always synonyms. See *parameter* for more about the distinction and usage of these terms.

attribute

Methods and data attributes (i.e. "fields" in Java terms) are all known as attributes in Python. A method is just an attribute that happens to be a callable object (usually a function, but not necessarily).

BDFL

Benevolent Dictator For Life, alias for Guido van Rossum, creator of the Python language.

binary sequence

Generic term for sequence types with byte elements. The built-in binary sequence types are `byte`, `bytearray` and `memoryview`.

BIF

See *built-in function*.

BOM

Byte Order Mark, a sequence of bytes that may be present at the start of an UTF-16 encoded file. A BOM is the character U+FEFF (`ZERO WIDTH NO-BREAK SPACE`) encoded to produce either `b'\xfe\xff'` on a big-endian CPU, or `b'\xff\xfe'` on a little-endian one. Since there is no U+FFFE character in Unicode, the presence of these bytes unambiguously reveals the byte ordering used in the encoding. Although redundant, a BOM encoded as `b'\xef\xbb\xbf'` may be found in UTF-8 files.

bound method

A method that is accessed through an instance becomes bound to that instance. Any method is actually a descriptor and when accessed, it returns itself wrapped in an object that binds the method to the instance. That object is the bound method. It can be invoked without passing the value of `self`. For example, given the assignment `my_method = my_obj.method`, the bound method can later be called as `my_method()`. Contrast with *unbound method*.

built-in function

A function bundled with the Python interpreter, coded in the underlying implementation language (i.e. C, for CPython; Java, for Jython and so on). The term often refers only to the functions that don't need to be imported, documented in the *The Python Standard Library Reference*, Chapter 2. Built-in Functions. But built-in modules like `sys`, `math`, `re` etc. also contain built-in functions.

byte string

An unfortunate name still used to refer to `bytes` or `bytearray` in Python 3. In Python 2, the `str` type was really a byte string, and the term made sense to distinguish `str` from `unicode` strings. In Python 3 it makes no sense to insist on this term, and I tried to use *byte sequence* whenever I needed to talk in general about… byte sequences.

bytes-like object

A generic sequence of bytes. The most common bytes-like types are `bytes`, `bytearray` and `memoryview` are but other objects supporting the low-level CPython buffer protocol also qualify, if their elements are single bytes.

callable object

An object that can be invoked with the call operator `()`, to return a result or to perform some action. There are seven flavors of callable objects in Python: user-defined functions, built-in functions, built-in methods, instance methods, generator functions, classes and instances of classes that implement the `__call__` special method.

CamelCase

The convention of writing identifiers by joining words with upper cased initials, for example: `ConnectionRefusedError`. PEP-8 recommends class names should be written in CamelCase, but the advice is not followed by the Python standard library. See *snake_case*.

Cheese Shop

Original name of the Python Package Index (PyPI), after the Monty Python skit about a cheese shop where nothing is available. The alias *https://cheeseshop.python.org/* still works, as of this writing. See *PyPI*.

class

A program construct defining a new type, with data attributes and methods specifying possible operations on them. See `type`.

code point

An integer in the range 0 to 0x10FFFF used to identify an entry in the Unicode character database. As of Unicode 7.0, less than 3% of all code points are assigned to characters. In the Python documentation, the term may be spelled as one or two words. For example, in Chapter 2. Built-in Functions of the Python Library Reference, the `chr` function is said to take an integer "codepoint", while its inverse, `ord`, is described as returning a "Unicode code point".

code smell

A coding pattern which suggests there may be something wrong with the design of a program. For example, excessive use of `isinstance` checks against concrete classes is a code smell, as it makes the program harder to extend to deal with new types in the future.

codec

(encoder/decoder) A module with functions to encode and decode, usually from `str` to `bytes` and back, although Python has a few codecs that perform `bytes` to `bytes` and `str` to `str` transformations.

collection

Generic term for data structures made of items that can be accessed individually. Some collections can contain objects of arbitrary types (see *container*) and others only objects of a single atomic type (see *flat sequence*). `list` and `bytes` are both collections, but `list` is a container, and `bytes` is a flat sequence.

considered harmful

Edsger Dijkstra's letter titled "Go To Statement Considered Harmful" established a formula for titles of essays criticizing some computer science technique. The English-language Wikipedia has an *Considered_harmful* article listing several examples, including *"Considered Harmful" Essays Considered Harmful* by Eric A. Meyer.

constructor

Informally, the `__init__` instance method of a class is called its constructor, since its semantics is similar to that of a Java constructor. However, a fitting name for `__init__` is *initializer*, since it does not actually build the instance, but receives it as its `self` argument. The *constructor* term better describes the `__new__` class method, which Python calls before `__init__`, and is responsible for actually creating and instance and returning it. See *initializer*.

container

An object that holds references to other objects. Most collection types in Python are containers, but some are not. Contrast with *flat sequence*, which are collections but not containers.

context manager

An object implementing both the `__enter__` and `__exit__` special methods, for use in a `with` block.

coroutine

A generator used for concurrent programming by receiving values from a scheduler or an event loop via `coro.send(value)`. The term may be used to describe the generator function or the generator object obtained by calling the generator function. See *generator*.

CPython

The standard Python interpreter, implemented in C. This term is only used when discussing implementation-specific behavior, or when talking about the multiple Python interpreters available, such as *PyPy*.

CRUD

Acronym for Create, Read, Update and Delete, the four basic functions in any application that stores records.

decorator

A callable object `A` that returns another callable object `B` and is invoked in code using the syntax `@A` right before the definition of a callable `C`. When reading such code, the

Python interpreter invokes `A(C)` and binds the resulting `B` to the variable previously assigned to `C`, effectively replacing the definition of `C` with `B`. If the target callable `C` is a function, then `A` is a function decorator; if `C` is a class, then `A` is a class decorator.

deep copy
: A copy of an object in which all the objects that are attributes of the object are themselves also copied. Contrast with *shallow copy*.

descriptor
: A class implementing one or more of the `__get__`, `__set__` or `__delete__` special methods becomes a descriptor when one of its instances is used as a class attribute of another class, the *managed class*. Descriptors manage the access and deletion of *managed attributes* in the *managed class*, often storing data in the *managed instances*.

docstring
: Short for documentation string. When the first statement in a module, class or function is a string literal, it is taken to be the *docstring* for the enclosing object, and the interpreter saves it as the `__doc__` attribute of that object. Se also: *doctest*.

doctest
: A module with functions to parse and run examples embedded in the docstrings of Python modules or in plain text files. May also be used from the command line as: `python -m doctest module_with_tests.py`.

DRY
: Don't Repeat Yourself — a software engineering principle stating that "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." It first appeared in the book *The Pragmatic Programmer* by Andy Hunt and Dave Thomas (Addison-Wesley, 1999).

duck typing
: A form of polymorphism where functions operate on any object that implements the appropriate methods, regardless of their classes or explicit interface declarations.

dunder
: Shortcut to pronounce the names of *special methods* and attributes that are written with leading and trailing double-underscores (i.e. `__len__` is read as "dunder len").

dunder method
: See *dunder* and *special methods*.

EAFP
: Acronym standing for the quote "It's easier to ask forgiveness than permission", attributed to computer pioneer Grace Hopper, and quoted by Pythonistas referring to dynamic programming practices like accessing attributes without testing first if they exist, and then catching the exception when that is the case. The docstring for the `hasattr` function actually says that it works "by calling getattr(object, name) and catching AttributeError".

eager
: An iterable object that builds all its items at once. In Python, a *list comprehension* is eager. Contrast with *lazy*.

fail-fast
: A systems design approach recommending that errors should be reported as early as possible. Python adheres to this principle more closely than most dynamic languages. For example, there is no "undefined" value: variables referenced before initialization generate an error, and `my_dict[k]` raises an exception if `k` is missing (in contrast with JavaScript). As another example, parallel assignment via tuple unpacking in Python only works if every item is explicitly handled, while Ruby silently deals with item count mismatches by ignoring unused items on the right side of the `=`, or by assigning `nil` to extra variables on the left side.

falsy
: Any value `x` for which `bool(x)` returns `False`; Python implicitly uses `bool` to evaluate objects in boolean contexts, such as the

expression controlling an `if` or `while` loop. The opposite of *truthy*.

file-like object

Used informally in the official documentation to refer to objects implementing the file protocol, with methods such as `read`, `write`, `close` etc. Common variants are text files containing encoded strings with line-oriented reading and writing, `StringIO` instances which are in-memory text files, and binary files, containing unencoded bytes. The latter may be buffered or unbuffered. ABCs for the standard file types are defined in the `io` module since Python 2.6.

first-class function

Any function that is a first-class object in the language, i.e. can be created at run time, assigned to variables, passed as an argument, and returned as the result of another function. Python functions are first-class functions.

flat sequence

A sequence type that physically stores the values of its items, and not references to other objects. The built-in types `str`, `bytes`, `bytearray`, `memoryview` and `array.array` are flat sequences. Contrast with `list`, `tuple` and `collections.deque` which are container sequences. See *container*.

function

Strictly, an object resulting from evaluation of a **`def`** block or a **`lambda`** expression. Informally, the word *function* is used to describe any callable object, such as methods and even classes sometimes. The official Built-in Functions list includes several built-in classes like `dict`, `range` and `str`. Also see callable object.

genexp

Short for *generator expression*.

generator

An iterator built with a generator function or a generator expression that may produce values without necessarily iterating over a collection; the canonical example is a generator to produce the Fibonacci series which, because it is infinite, would never fit in a collection. The term is sometimes used to describe a *generator function*, besides the object that results from calling it.

generator function

A function that has the `yield` keyword in its body. When invoked, a generator function returns a *generator*.

generator expression

An expression enclosed in parenthesis using the same syntax of a *list comprehension*, but returning a generator instead of a list. A *generator expression* can be understood as a *lazy* version of a *list comprehension*. See *lazy*.

generic function

a group of functions designed to implement the same operation in customized ways for different object types. As of Python 3.4, the `functools.singledispatch` decorator the standard way to create generic functions. This is known as multimethods in other languages.

GoF book

Alias for *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995), authored by the so-called Gang of Four (GoF): Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

hashable

An object is hashable if it has both `__hash__` and `__eq__` methods, with the constraints that the hash value must never change and if `a == b` then `hash(a) == hash(b)` must also be `True`. Most immutable built-in types are hashable, but a tuple is only hashable if every one of its items is also hashable.

higher-order function

A function that takes another function as argument, like `sorted`, `map` and `filter, or a function that returns a function as result, as Python decorators do.

idiom

“A manner of speaking that is natural to native speakers of a language”, according to the Princeton WordNet.

import time

The moment of initial execution of a module when its code is loaded by the Python interpreter, evaluated from top to bottom and compiled into bytecode. This is when classes and functions are defined and become live objects. This is also when decorators are executed.

initializer

A better name for the `__init__` method (instead of *constructor*). Initializing the instance received as `self` is the task of `__init__`. Actual instance construction is done by the `__new__` method. See *constructor*.

iterable

Any object from which the `iter` built-in function can obtain an iterator. An iterable object works as the source of items in for loops, comprehensions and tuple unpacking. Objects implementing an `__iter__` method returning an *iterator* are iterable. Sequences are always iterable; other objects implementing a `__getitem__` method may also be iterable.

iterable unpacking

A modern, more precise synonym for *tuple unpacking*. See also *parallel assignment*.

iterator

Any object that implements the `__next__` no-argument method which returns the next item in a series, or raises `StopIteration` when there are no more items. Python iterators also implement the `__iter__` method so they are also *iterable*. Classic iterators, according to the original design pattern, return items from a collection. A *generator* is also an *iterator*, but it’s more flexible. See *generator*.

KISS principle

The acronym stands for “Keep It Simple, Stupid”. This calls seeking the simplest possible solution, with the fewest moving parts. The phrase was coined by Kelly Johnson, a highly accomplished aerospace engineer who worked in the real Area 51 designing some of the most advanced aircraft of the 20th century.

lazy

An iterable object which produces items on demand. In Python, generators are lazy. Contrast *eager*.

listcomp

Short for *list comprehension*.

list comprehension

An expression enclosed in brackets that uses the `for` and `in` keywords to build a list by processing and filtering the elements from one or more iterables. A list comprehension works eagerly. See *eager*.

liveness

An asynchronous, threaded or distributed system exhibits the liveness property when “something good eventually happens”, i.e. even if some expected computation is not happening right now, it will be completed eventually. If a system deadlocks, it has lost its liveness.

magic method

Same as *special method*.

managed attribute

A public attribute managed by a descriptor object. Although the *managed attribute* is defined in the *managed class*, it operates like an instance attribute, i.e. it usually has a value per instance, held in a *storage attribute*. See *descriptor*.

managed class

A class that uses a descriptor object to manage one of its attributes. See *descriptor*.

managed instance

An instance of a *managed class*. See *managed attribute*` and *descriptor*.

metaclass

A class whose instances are classes. By default, Python classes are instances of `type`, for example, `type(int)` is the class `type`, therefore `type` is a metaclass. User-defined metaclasses can be created by subclassing `type`.

metaprogramming
: The practice of writing programs the use run-time information about themselves to change their behavior. For example, an *ORM* may introspect model class declarations to figure determine how to validate database record fields and convert database types to Python types.

monkey patching
: Dynamically changing a module, class or function at run time, usually to add features or fix bugs. Because it is done in memory and not by changing the source code, a monkey patch only affects the currently running instance of the program. Monkey patches break encapsulation and tend to be tightly coupled to the implementation details of the patched code units, so they are seen as temporary work-arounds and not a recommended technique for code integration.

mixin class
: A class designed to be subclassed together with one or more additional classes in a multiple-inheritance class tree. A mixin class should never be instantiated, and a concrete subclass of a mixin class should also subclass another non-mixin class.

mixin method
: A concrete method implementation provided in an ABC or in a *mixin class*.

mutator
: See *accessor*.

name mangling
: The automatic renaming of private attributes from `__x` to `_MyClass__x`, performed by the Python interpreter at runtime.

non-overriding descriptor
: A *descriptor* that does not implement `__set__` and therefore does not interfere with setting of the *managed attribute* in the *managed instance*. Consequently, if a namesake attribute is set in the *managed instance*, it will shadow the descriptor in that instance. Also called non-data descriptor or shadowable descriptor. Contrast with *overriding descriptor*.

ORM
: Object-Relational Mapper — an API that provides access to database tables and records as Python classes and objects, providing method calls to perform database operations. SQLAlchemy is a popular stand-alone Python ORM; the Django and Web2py frameworks have their own bundled ORMs.

overriding descriptor
: A *descriptor* that implements `__set__` and therefore intercepts and overrides attempts at setting the *managed attribute* in the *managed instance*. Also called data descriptor or enforced descriptor. Contrast with *non-overriding descriptor*.

parallel assignment
: Assigning to several variables from items in an iterable, using syntax like `a, b = [c, d]` — also known as destructuring assignment. This is a common application of *tuple unpacking*.

parameter
: Functions are declared with 0 or more “formal parameters”, which are unbound local variables. When the function is called, the *arguments* or “actual parameters” passed are bound to those variables. In this book I tried to use *argument* to refer to an actual parameter passed to a function, and *parameter* for a formal parameter in the function declaration. However, that is not always feasible because the terms *parameter* and *argument* are use interchangeably all over the Python docs and API. See *argument*.

prime (verb)
: Calling `next(coro)` on a coroutine to advance it to its first `yield` expression so that

it becomes ready to receive values in succeeding `coro.send(value)` calls.

PyPI

The Python Package Index at *https://pypi.python.org/* where more than 49.000 packages are available. A.k.a. the *Cheese shop* (which see). PyPI is pronounced as "pie-P-eye" to avoid confusion with *PyPy*.

PyPy

An alternative implementation of the Python programming language using a toolchain that compiles a subset of Python to machine code, so the interpreter source code is actually written in Python. PyPy also includes a JIT to generate machine code for user programs on the fly — like the Java VM does. As of November, 2014, PyPy is 6.8 times faster than CPython on average, according to published benchmarks. PyPy is pronounced as "pie-pie" to avoid confusion with *PyPI*.

Pythonic

Used to praise idiomatic Python code, which makes good use of language features to be concise, readable and often faster as well. Also said of APIs that enable coding in a way that seems natural to proficient Python programmers. See *idiom*.

refcount

The reference counter that each CPython object keeps internally in order to determine when it can be destroyed by the garbage collector.

referent

The object that is the target of a reference. This term is most often used to discuss *weak references*.

REPL

Read-eval-print-loop, an interactive console, like the standard `python` or alternatives like `ipython`, `bpython` and Python Anywhere.

sequence

Generic name for any iterable data structure with a known size and (eg. `len(s)`) and allowing item access via 0-based integer indexes (eg. `s[0]`). The word *sequence* has been part of the Python jargon from the start, but only with Python 2.6 it was formalized as an abstract class in `collections.abc.Sequence`.

serialization

Converting an object from its in-memory structure to a binary or text oriented format for storage or transmission, in a way that allows the future reconstruction of a clone of the object on the same system or on a different one. The `pickle` module supports serialization of arbitrary Python objects to a binary format.

shallow copy

A copy of an object which shares references to all the objects that are attributes of the original object. Contrast with *deep copy*. Also see *aliasing*.

singleton

An object that is the only existing instance of a class — usually not by accident but because the class is designed to prevent creation of more than one instance. There is also a design pattern named Singleton, which is a recipe for coding such classes. The `None` object is a singleton in Python.

slicing

Producing a subset of a sequence by using the slice notation, e.g. `my_sequence[2:6]`. Slicing usually copies data to produce a new object; in particular, `my_sequence[:]` creates a shallow copy of the entire sequence. But a `memoryview` object can be sliced to produce a new `memoryview` which shares data with the original object.

snake_case

The convention of writing identifiers by joining words with the underscore character _, for example: `run_until_complete`. PEP-8 calls this style "lowercase with words separated by underscores" and recommends it for naming functions, methods, arguments and variables. For packages, PEP-8 recommends concatenating words with no separators. The Python standard library has many examples of `snake_case` identifiers, but also many examples of identifiers with no separation between words, e.g. `getattr`, `classmethod`, `isinstance`, `str.endswith` etc. See *CamelCase*.

special method

A method with a special name such as `__getitem__`, spelled with leading and trailing double underscores. Almost all special methods recognized by Python are described in the Data Model chapter of the Python Language Reference, but a few that are used only in specific contexts are documented in other parts of the documentation. For example the `__missing__` method of mappings is mentioned in the `dict` section of the Built-in Types page in the Standard Library documentation.

storage attribute

An attribute in a *managed instance* used to store the value of an attribute managed by a *descriptor*. See also *managed attribute*.

strong reference

A reference that keeps an object alive in Python. Contrast with *weak reference*.

tuple unpacking

Assigning items from an iterable object to a tuple of variables, for example: `first, second, third == my_list`. This is the usual term used by Pythonistas, but *iterable unpacking* is gaining traction.

truthy

Any value `x` for which `bool(x)` returns `True`; Python implicitly uses `bool` to evaluate objects in boolean contexts, such as the expression controlling an `if` or `while` loop. The opposite of *falsy*.

type

Each specific category of program data, defined by a set of possible values and operations on them. Some Python types are close to machine data types (e.g. `float` and `bytes`) while others are extensions (e.g. `int` is not limited to CPU word size, `str` holds multi-byte Unicode data points) and very high-level abstractions (e.g. `dict`, `deque` etc.). Types may be user defined or built into the interpreter (a "built-in" type). Before the watershed type/class unification in Python 2.2, types and classes were different entities, and user-defined classes could not extend built-in types. Since then, built-in types and new-style classes became compatible, and a class is an instance of `type`. In Python 3 all classes are new-style classes. See *class* and *metaclass*.

unbound method

An instance method accessed directly on a class is not bound to an instance, therefor it's said to be an "unbound method". To succeed, a call to an unbound method must explicitly pass an instance of the class as the first argument. That instance will be assigned to the `self` argument in the method. See *bound method*.

uniform access principle

Bertrand Meyer, creator of the Eiffel Language, wrote: "All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation". Properties and descriptors allow the implementation of the uniform access principle in Python. The lack of a `new` operator, making function calls and object instantiation look the same, is another form of this principle: the caller does not need to know whether the invoked object is a class, a function or any other callable.

user-defined

Almost always in the Python docs the word *user* refers to you and I — programmers who use the Python language — as opposed to the developers who implement a Python interpreter. So the term "user-defined class" means a class written in Python, as opposed to built-in classes written in C, like `str`.

virtual subclass

A class that does not inherit from a superclass but is registered using `TheSuper`

`Class.register(TheSubClass)`. See documentation for `abc.ABCMeta.register`

wart

A misfeature of the language. Andrew Kushling's famous post "Python warts" has been acknowledged by the *BDFL* as influential in the decision to break backwards-compatibility in the design of Python 3, since most of the failings could not be fixed otherwise. Many of Kushling's issues were fixed in Python 3.

weak reference

A special kind of object reference that does not increase the *referent* object reference count. Weak references are created with one of the functions and data structures in the `weakref` module.

YAGNI

"You Ain't Gonna Need It", a slogan to avoid implementing functionality that is not immediately necessary based on assumptions about future needs.

Zen of Python

type `import this` into any Python console since version 2.2.