

*Recipes for Mastering Python 3*

**3rd Edition**



# Python Cookbook™

**O'REILLY®**

*David Beazley & Brian K. Jones*

# Python Cookbook

If you need help writing programs in Python 3, or want to update older Python 2 code, this book is just the ticket. Packed with practical recipes written and tested with Python 3.3, this unique cookbook is for experienced Python programmers who want to focus on modern tools and idioms.

Inside, you'll find complete recipes for more than a dozen topics, covering the core Python language as well as tasks common to a wide variety of application domains. Each recipe contains code samples you can use in your projects right away, along with a discussion about how and why the solution works.

## Topics include:

- Data Structures and Algorithms
- Strings and Text
- Numbers, Dates, and Times
- Iterators and Generators
- Files and I/O
- Data Encoding and Processing
- Functions
- Classes and Objects
- Metaprogramming
- Modules and Packages
- Network and Web Programming
- Concurrency
- Utility Scripting and System Administration
- Testing, Debugging, and Exceptions
- C Extensions

**David Beazley**, an independent software developer, teaches programming courses for developers, scientists, and engineers. He's the author of the *Python Essential Reference* (Addison-Wesley), and has created several open-source Python packages.

**Brian K. Jones** is a system administrator in the Department of Computer Science at Princeton University.

US \$49.99

CAN \$52.99

ISBN: 978-1-449-34037-7



Twitter: @oreillymedia  
facebook.com/oreilly

**O'REILLY®**  
oreilly.com

THIRD EDITION

---

# Python Cookbook

*David Beazley and Brian K. Jones*

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

## **Python Cookbook, Third Edition**

by David Beazley and Brian K. Jones

Copyright © 2013 David Beazley and Brian Jones. All rights reserved.

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://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Editors:** Meghan Blanchette and Rachel Roumeliotis

**Indexer:** WordCo Indexing Services

**Production Editor:** Kristen Borg

**Cover Designer:** Karen Montgomery

**Copyeditor:** Jasmine Kwityn

**Interior Designer:** David Futato

**Proofreader:** BIM Proofreading Services

**Illustrator:** Robert Romano

May 2013: Third Edition

### **Revision History for the Third Edition:**

2013-05-08: First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449340377> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Python Cookbook*, the image of a springhaas, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

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: 978-1-449-34037-7

[LSI]

---

# Table of Contents

<b>Preface.....</b>	<b>xi</b>
<b>1. Data Structures and Algorithms.....</b>	<b>1</b>
1.1. Unpacking a Sequence into Separate Variables	1
1.2. Unpacking Elements from Iterables of Arbitrary Length	3
1.3. Keeping the Last N Items	5
1.4. Finding the Largest or Smallest N Items	7
1.5. Implementing a Priority Queue	8
1.6. Mapping Keys to Multiple Values in a Dictionary	11
1.7. Keeping Dictionaries in Order	12
1.8. Calculating with Dictionaries	13
1.9. Finding Commonalities in Two Dictionaries	15
1.10. Removing Duplicates from a Sequence while Maintaining Order	17
1.11. Naming a Slice	18
1.12. Determining the Most Frequently Occurring Items in a Sequence	20
1.13. Sorting a List of Dictionaries by a Common Key	21
1.14. Sorting Objects Without Native Comparison Support	23
1.15. Grouping Records Together Based on a Field	24
1.16. Filtering Sequence Elements	26
1.17. Extracting a Subset of a Dictionary	28
1.18. Mapping Names to Sequence Elements	29
1.19. Transforming and Reducing Data at the Same Time	32
1.20. Combining Multiple Mappings into a Single Mapping	33
<b>2. Strings and Text.....</b>	<b>37</b>
2.1. Splitting Strings on Any of Multiple Delimiters	37
2.2. Matching Text at the Start or End of a String	38
2.3. Matching Strings Using Shell Wildcard Patterns	40
2.4. Matching and Searching for Text Patterns	42

2.5. Searching and Replacing Text	45
2.6. Searching and Replacing Case-Insensitive Text	46
2.7. Specifying a Regular Expression for the Shortest Match	47
2.8. Writing a Regular Expression for Multiline Patterns	48
2.9. Normalizing Unicode Text to a Standard Representation	50
2.10. Working with Unicode Characters in Regular Expressions	52
2.11. Stripping Unwanted Characters from Strings	53
2.12. Sanitizing and Cleaning Up Text	54
2.13. Aligning Text Strings	57
2.14. Combining and Concatenating Strings	58
2.15. Interpolating Variables in Strings	61
2.16. Reformatting Text to a Fixed Number of Columns	64
2.17. Handling HTML and XML Entities in Text	65
2.18. Tokenizing Text	66
2.19. Writing a Simple Recursive Descent Parser	69
2.20. Performing Text Operations on Byte Strings	78
<b>3. Numbers, Dates, and Times.....</b>	<b>83</b>
3.1. Rounding Numerical Values	83
3.2. Performing Accurate Decimal Calculations	84
3.3. Formatting Numbers for Output	87
3.4. Working with Binary, Octal, and Hexadecimal Integers	89
3.5. Packing and Unpacking Large Integers from Bytes	90
3.6. Performing Complex-Valued Math	92
3.7. Working with Infinity and NaNs	94
3.8. Calculating with Fractions	96
3.9. Calculating with Large Numerical Arrays	97
3.10. Performing Matrix and Linear Algebra Calculations	100
3.11. Picking Things at Random	102
3.12. Converting Days to Seconds, and Other Basic Time Conversions	104
3.13. Determining Last Friday's Date	106
3.14. Finding the Date Range for the Current Month	107
3.15. Converting Strings into Datetimes	109
3.16. Manipulating Dates Involving Time Zones	110
<b>4. Iterators and Generators.....</b>	<b>113</b>
4.1. Manually Consuming an Iterator	113
4.2. Delegating Iteration	114
4.3. Creating New Iteration Patterns with Generators	115
4.4. Implementing the Iterator Protocol	117
4.5. Iterating in Reverse	119
4.6. Defining Generator Functions with Extra State	120

4.7. Taking a Slice of an Iterator	122
4.8. Skipping the First Part of an Iterable	123
4.9. Iterating Over All Possible Combinations or Permutations	125
4.10. Iterating Over the Index-Value Pairs of a Sequence	127
4.11. Iterating Over Multiple Sequences Simultaneously	129
4.12. Iterating on Items in Separate Containers	131
4.13. Creating Data Processing Pipelines	132
4.14. Flattening a Nested Sequence	135
4.15. Iterating in Sorted Order Over Merged Sorted Iterables	136
4.16. Replacing Infinite while Loops with an Iterator	138
<b>5. Files and I/O.....</b>	<b>141</b>
5.1. Reading and Writing Text Data	141
5.2. Printing to a File	144
5.3. Printing with a Different Separator or Line Ending	144
5.4. Reading and Writing Binary Data	145
5.5. Writing to a File That Doesn't Already Exist	147
5.6. Performing I/O Operations on a String	148
5.7. Reading and Writing Compressed Datafiles	149
5.8. Iterating Over Fixed-Sized Records	151
5.9. Reading Binary Data into a Mutable Buffer	152
5.10. Memory Mapping Binary Files	153
5.11. Manipulating Pathnames	156
5.12. Testing for the Existence of a File	157
5.13. Getting a Directory Listing	158
5.14. Bypassing Filename Encoding	160
5.15. Printing Bad Filenames	161
5.16. Adding or Changing the Encoding of an Already Open File	163
5.17. Writing Bytes to a Text File	165
5.18. Wrapping an Existing File Descriptor As a File Object	166
5.19. Making Temporary Files and Directories	167
5.20. Communicating with Serial Ports	170
5.21. Serializing Python Objects	171
<b>6. Data Encoding and Processing.....</b>	<b>175</b>
6.1. Reading and Writing CSV Data	175
6.2. Reading and Writing JSON Data	179
6.3. Parsing Simple XML Data	183
6.4. Parsing Huge XML Files Incrementally	186
6.5. Turning a Dictionary into XML	189
6.6. Parsing, Modifying, and Rewriting XML	191
6.7. Parsing XML Documents with Namespaces	193

6.8. Interacting with a Relational Database	195
6.9. Decoding and Encoding Hexadecimal Digits	197
6.10. Decoding and Encoding Base64	199
6.11. Reading and Writing Binary Arrays of Structures	199
6.12. Reading Nested and Variable-Sized Binary Structures	203
6.13. Summarizing Data and Performing Statistics	214
<b>7. Functions.....</b>	<b>217</b>
7.1. Writing Functions That Accept Any Number of Arguments	217
7.2. Writing Functions That Only Accept Keyword Arguments	219
7.3. Attaching Informational Metadata to Function Arguments	220
7.4. Returning Multiple Values from a Function	221
7.5. Defining Functions with Default Arguments	222
7.6. Defining Anonymous or Inline Functions	224
7.7. Capturing Variables in Anonymous Functions	225
7.8. Making an N-Argument Callable Work As a Callable with Fewer Arguments	227
7.9. Replacing Single Method Classes with Functions	231
7.10. Carrying Extra State with Callback Functions	232
7.11. Inlining Callback Functions	235
7.12. Accessing Variables Defined Inside a Closure	238
<b>8. Classes and Objects.....</b>	<b>243</b>
8.1. Changing the String Representation of Instances	243
8.2. Customizing String Formatting	245
8.3. Making Objects Support the Context-Management Protocol	246
8.4. Saving Memory When Creating a Large Number of Instances	248
8.5. Encapsulating Names in a Class	250
8.6. Creating Managed Attributes	251
8.7. Calling a Method on a Parent Class	256
8.8. Extending a Property in a Subclass	260
8.9. Creating a New Kind of Class or Instance Attribute	264
8.10. Using Lazily Computed Properties	267
8.11. Simplifying the Initialization of Data Structures	270
8.12. Defining an Interface or Abstract Base Class	274
8.13. Implementing a Data Model or Type System	277
8.14. Implementing Custom Containers	283
8.15. Delegating Attribute Access	287
8.16. Defining More Than One Constructor in a Class	291
8.17. Creating an Instance Without Invoking <i>init</i>	293
8.18. Extending Classes with Mixins	294
8.19. Implementing Stateful Objects or State Machines	299



8.20. Calling a Method on an Object Given the Name As a String	305
8.21. Implementing the Visitor Pattern	306
8.22. Implementing the Visitor Pattern Without Recursion	311
8.23. Managing Memory in Cyclic Data Structures	317
8.24. Making Classes Support Comparison Operations	321
8.25. Creating Cached Instances	323
<b>9. Metaprogramming.....</b>	<b>329</b>
9.1. Putting a Wrapper Around a Function	329
9.2. Preserving Function Metadata When Writing Decorators	331
9.3. Unwrapping a Decorator	333
9.4. Defining a Decorator That Takes Arguments	334
9.5. Defining a Decorator with User Adjustable Attributes	336
9.6. Defining a Decorator That Takes an Optional Argument	339
9.7. Enforcing Type Checking on a Function Using a Decorator	341
9.8. Defining Decorators As Part of a Class	345
9.9. Defining Decorators As Classes	347
9.10. Applying Decorators to Class and Static Methods	350
9.11. Writing Decorators That Add Arguments to Wrapped Functions	352
9.12. Using Decorators to Patch Class Definitions	355
9.13. Using a Metaclass to Control Instance Creation	356
9.14. Capturing Class Attribute Definition Order	359
9.15. Defining a Metaclass That Takes Optional Arguments	362
9.16. Enforcing an Argument Signature on *args and **kwargs	364
9.17. Enforcing Coding Conventions in Classes	367
9.18. Defining Classes Programmatically	370
9.19. Initializing Class Members at Definition Time	374
9.20. Implementing Multiple Dispatch with Function Annotations	376
9.21. Avoiding Repetitive Property Methods	382
9.22. Defining Context Managers the Easy Way	384
9.23. Executing Code with Local Side Effects	386
9.24. Parsing and Analyzing Python Source	388
9.25. Disassembling Python Byte Code	392
<b>10. Modules and Packages.....</b>	<b>397</b>
10.1. Making a Hierarchical Package of Modules	397
10.2. Controlling the Import of Everything	398
10.3. Importing Package Submodules Using Relative Names	399
10.4. Splitting a Module into Multiple Files	401
10.5. Making Separate Directories of Code Import Under a Common Namespace	404
10.6. Reloading Modules	406

10.7. Making a Directory or Zip File Runnable As a Main Script	407
10.8. Reading Datafiles Within a Package	408
10.9. Adding Directories to sys.path	409
10.10. Importing Modules Using a Name Given in a String	411
10.11. Loading Modules from a Remote Machine Using Import Hooks	412
10.12. Patching Modules on Import	428
10.13. Installing Packages Just for Yourself	431
10.14. Creating a New Python Environment	432
10.15. Distributing Packages	433
<b>11. Network and Web Programming.....</b>	<b>437</b>
11.1. Interacting with HTTP Services As a Client	437
11.2. Creating a TCP Server	441
11.3. Creating a UDP Server	445
11.4. Generating a Range of IP Addresses from a CIDR Address	447
11.5. Creating a Simple REST-Based Interface	449
11.6. Implementing a Simple Remote Procedure Call with XML-RPC	454
11.7. Communicating Simply Between Interpreters	456
11.8. Implementing Remote Procedure Calls	458
11.9. Authenticating Clients Simply	461
11.10. Adding SSL to Network Services	464
11.11. Passing a Socket File Descriptor Between Processes	470
11.12. Understanding Event-Driven I/O	475
11.13. Sending and Receiving Large Arrays	481
<b>12. Concurrency.....</b>	<b>485</b>
12.1. Starting and Stopping Threads	485
12.2. Determining If a Thread Has Started	488
12.3. Communicating Between Threads	491
12.4. Locking Critical Sections	497
12.5. Locking with Deadlock Avoidance	500
12.6. Storing Thread-Specific State	504
12.7. Creating a Thread Pool	505
12.8. Performing Simple Parallel Programming	509
12.9. Dealing with the GIL (and How to Stop Worrying About It)	513
12.10. Defining an Actor Task	516
12.11. Implementing Publish/Subscribe Messaging	520
12.12. Using Generators As an Alternative to Threads	524
12.13. Polling Multiple Thread Queues	531
12.14. Launching a Daemon Process on Unix	534
<b>13. Utility Scripting and System Administration.....</b>	<b>539</b>

13.1. Accepting Script Input via Redirection, Pipes, or Input Files	539
13.2. Terminating a Program with an Error Message	540
13.3. Parsing Command-Line Options	541
13.4. Prompting for a Password at Runtime	544
13.5. Getting the Terminal Size	545
13.6. Executing an External Command and Getting Its Output	545
13.7. Copying or Moving Files and Directories	547
13.8. Creating and Unpacking Archives	549
13.9. Finding Files by Name	550
13.10. Reading Configuration Files	552
13.11. Adding Logging to Simple Scripts	555
13.12. Adding Logging to Libraries	558
13.13. Making a Stopwatch Timer	559
13.14. Putting Limits on Memory and CPU Usage	561
13.15. Launching a Web Browser	563
<b>14. Testing, Debugging, and Exceptions.....</b>	<b>565</b>
14.1. Testing Output Sent to stdout	565
14.2. Patching Objects in Unit Tests	567
14.3. Testing for Exceptional Conditions in Unit Tests	570
14.4. Logging Test Output to a File	572
14.5. Skipping or Anticipating Test Failures	573
14.6. Handling Multiple Exceptions	574
14.7. Catching All Exceptions	576
14.8. Creating Custom Exceptions	578
14.9. Raising an Exception in Response to Another Exception	580
14.10. Reraising the Last Exception	582
14.11. Issuing Warning Messages	583
14.12. Debugging Basic Program Crashes	585
14.13. Profiling and Timing Your Program	587
14.14. Making Your Programs Run Faster	590
<b>15. C Extensions.....</b>	<b>597</b>
15.1. Accessing C Code Using ctypes	599
15.2. Writing a Simple C Extension Module	605
15.3. Writing an Extension Function That Operates on Arrays	609
15.4. Managing Opaque Pointers in C Extension Modules	612
15.5. Defining and Exporting C APIs from Extension Modules	614
15.6. Calling Python from C	619
15.7. Releasing the GIL in C Extensions	625
15.8. Mixing Threads from C and Python	625
15.9. Wrapping C Code with Swig	627

15.10. Wrapping Existing C Code with Cython	632
15.11. Using Cython to Write High-Performance Array Operations	638
15.12. Turning a Function Pointer into a Callable	643
15.13. Passing NULL-Terminated Strings to C Libraries	644
15.14. Passing Unicode Strings to C Libraries	648
15.15. Converting C Strings to Python	653
15.16. Working with C Strings of Dubious Encoding	654
15.17. Passing Filenames to C Extensions	657
15.18. Passing Open Files to C Extensions	658
15.19. Reading File-Like Objects from C	659
15.20. Consuming an Iterable from C	662
15.21. Diagnosing Segmentation Faults	663
<b>A. Further Reading.....</b>	<b>665</b>
<b>Index.....</b>	<b>667</b>

---

# Preface

Since 2008, the Python world has been watching the slow evolution of Python 3. It was always known that the adoption of Python 3 would likely take a long time. In fact, even at the time of this writing (2013), most working Python programmers continue to use Python 2 in production. A lot has been made about the fact that Python 3 is not backward compatible with past versions. To be sure, backward compatibility is an issue for anyone with an existing code base. However, if you shift your view toward the future, you'll find that Python 3 offers much more than meets the eye.

Just as Python 3 is about the future, this edition of the *Python Cookbook* represents a major change over past editions. First and foremost, this is meant to be a very forward looking book. All of the recipes have been written and tested with Python 3.3 without regard to past Python versions or the “old way” of doing things. In fact, many of the recipes will only work with Python 3.3 and above. Doing so may be a calculated risk, but the ultimate goal is to write a book of recipes based on the most modern tools and idioms possible. It is hoped that the recipes can serve as a guide for people writing new code in Python 3 or those who hope to modernize existing code.

Needless to say, writing a book of recipes in this style presents a certain editorial challenge. An online search for Python recipes returns literally thousands of useful recipes on sites such as ActiveState's [Python recipes](#) or [Stack Overflow](#). However, most of these recipes are steeped in history and the past. Besides being written almost exclusively for Python 2, they often contain workarounds and hacks related to differences between old versions of Python (e.g., version 2.3 versus 2.4). Moreover, they often use outdated techniques that have simply become a built-in feature of Python 3.3. Finding recipes exclusively focused on Python 3 can be a bit more difficult.

Rather than attempting to seek out Python 3-specific recipes, the topics of this book are merely inspired by existing code and techniques. Using these ideas as a springboard, the writing is an original work that has been deliberately written with the most modern Python programming techniques possible. Thus, it can serve as a reference for anyone who wants to write their code in a modern style.

In choosing which recipes to include, there is a certain realization that it is simply impossible to write a book that covers every possible thing that someone might do with Python. Thus, a priority has been given to topics that focus on the core Python language as well as tasks that are common to a wide variety of application domains. In addition, many of the recipes aim to illustrate features that are new to Python 3 and more likely to be unknown to even experienced programmers using older versions. There is also a certain preference to recipes that illustrate a generally applicable programming technique (i.e., programming patterns) as opposed to those that narrowly try to address a very specific practical problem. Although certain third-party packages get coverage, a majority of the recipes focus on the core language and standard library.

## Who This Book Is For

This book is aimed at more experienced Python programmers who are looking to deepen their understanding of the language and modern programming idioms. Much of the material focuses on some of the more advanced techniques used by libraries, frameworks, and applications. Throughout the book, the recipes generally assume that the reader already has the necessary background to understand the topic at hand (e.g., general knowledge of computer science, data structures, complexity, systems programming, concurrency, C programming, etc.). Moreover, the recipes are often just skeletons that aim to provide essential information for getting started, but which require the reader to do more research to fill in the details. As such, it is assumed that the reader knows how to use search engines and Python's excellent online documentation.

Many of the more advanced recipes will reward the reader's patience with a much greater insight into how Python actually works under the covers. You will learn new tricks and techniques that can be applied to your own code.

## Who This Book Is Not For

This is not a book designed for beginners trying to learn Python for the first time. In fact, it already assumes that you know the basics that might be taught in a Python tutorial or more introductory book. This book is also not designed to serve as a quick reference manual (e.g., quickly looking up the functions in a specific module). Instead, the book aims to focus on specific programming topics, show possible solutions, and serve as a springboard for jumping into more advanced material you might find online or in a reference.

# Conventions Used in This Book

The following typographical conventions are used in this book:

## *Italic*

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

## Constant width

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

## Constant width bold

Shows commands or other text that should be typed literally by the user.

## *Constant width italic*

Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

# Online Code Examples

Almost all of the code examples in this book are available online at <http://github.com/dabeaz/python-cookbook>. The authors welcome bug fixes, improvements, and comments.

# Using Code Examples


This book is here to help you get your job done. In general, if this book includes code examples, you may use the code in this book 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: *Python Cookbook*, 3rd edition, by David Beazley and Brian K. Jones (O'Reilly). Copyright 2013 David Beazley and Brian Jones, 978-1-449-34037-7.

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at [permissions@oreilly.com](mailto: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://oreil.ly/python\\_cookbook\\_3e](http://oreil.ly/python_cookbook_3e).



To comment or ask technical questions about this book, send email to [\*bookquestions@oreilly.com\*](mailto:bookquestions@oreilly.com).

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

Find us on Facebook: [\*http://facebook.com/oreilly\*](http://facebook.com/oreilly)

Follow us on Twitter: [\*http://twitter.com/oreillymedia\*](http://twitter.com/oreillymedia)

Watch us on YouTube: [\*http://www.youtube.com/oreillymedia\*](http://www.youtube.com/oreillymedia)

## Acknowledgments

We would like to acknowledge the technical reviewers, Jake Vanderplas, Robert Kern, and Andrea Crotti, for their very helpful comments, as well as the general Python community for their support and encouragement. We would also like to thank the editors of the prior edition, Alex Martelli, Anna Ravenscroft, and David Ascher. Although this edition is newly written, the previous edition provided an initial framework for selecting the topics and recipes of interest. Last, but not least, we would like to thank readers of the early release editions for their comments and suggestions for improvement.

### David Beazley's Acknowledgments

Writing a book is no small task. As such, I would like to thank my wife Paula and my two boys for their patience and support during this project. Much of the material in this book was derived from content I developed teaching Python-related training classes over the last six years. Thus, I'd like to thank all of the students who have taken my courses and ultimately made this book possible. I'd also like to thank Ned Batchelder, Travis Oliphant, Peter Wang, Brian Van de Ven, Hugo Shi, Raymond Hettinger, Michael Foord, and Daniel Klein for traveling to the four corners of the world to teach these courses while I stayed home in Chicago to work on this project. Meghan Blanchette and Rachel Roumeliotis of O'Reilly were also instrumental in seeing this project through to completion despite the drama of several false starts and unforeseen delays. Last, but not least, I'd like to thank the Python community for their continued support and putting up with my flights of diabolical fancy.

David M. Beazley

[\*http://www.dabeaz.com\*](http://www.dabeaz.com)

[\*https://twitter.com/dabeaz\*](https://twitter.com/dabeaz)

## Brian Jones' Acknowledgments

I would like to thank both my coauthor, David Beazley, as well as Meghan Blanchette and Rachel Roumeliotis of O'Reilly, for working with me on this project. I would also like to thank my amazing wife, Natasha, for her patience and encouragement in this project, and her support in all of my ambitions. Most of all, I'd like to thank the Python community at large. Though I have contributed to the support of various open source projects, languages, clubs, and the like, no work has been so gratifying and rewarding as that which has been in the service of the Python community.

Brian K. Jones

*<http://www.protocolostomy.com>*

*<https://twitter.com/bkjones>*

---

# Data Structures and Algorithms

Python provides a variety of useful built-in data structures, such as lists, sets, and dictionaries. For the most part, the use of these structures is straightforward. However, common questions concerning searching, sorting, ordering, and filtering often arise. Thus, the goal of this chapter is to discuss common data structures and algorithms involving data. In addition, treatment is given to the various data structures contained in the collections module.

## 1.1. Unpacking a Sequence into Separate Variables

### Problem

You have an N-element tuple or sequence that you would like to unpack into a collection of N variables.

### Solution

Any sequence (or iterable) can be unpacked into variables using a simple assignment operation. The only requirement is that the number of variables and structure match the sequence. For example:

```
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>>

>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> name, shares, price, date = data
>>> name
```

```

'ACME'
>>> date
(2012, 12, 21)

>>> name, shares, price, (year, mon, day) = data
>>> name
'ACME'
>>> year
2012
>>> mon
12
>>> day
21
>>>

```

If there is a mismatch in the number of elements, you'll get an error. For example:

```

>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>>

```

## Discussion

Unpacking actually works with any object that happens to be iterable, not just tuples or lists. This includes strings, files, iterators, and generators. For example:

```

>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
>>> e
'o'
>>>

```

When unpacking, you may sometimes want to discard certain values. Python has no special syntax for this, but you can often just pick a throwaway variable name for it. For example:

```

>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>>

```

However, make sure that the variable name you pick isn't being used for something else already.

## 1.2. Unpacking Elements from Iterables of Arbitrary Length

### Problem

You need to unpack *N* elements from an iterable, but the iterable may be longer than *N* elements, causing a “too many values to unpack” exception.

### Solution

Python “star expressions” can be used to address this problem. For example, suppose you run a course and decide at the end of the semester that you’re going to drop the first and last homework grades, and only average the rest of them. If there are only four assignments, maybe you simply unpack all four, but what if there are 24? A star expression makes it easy:

```
def drop_first_last(grades):
    first, *middle, last = grades
    return avg(middle)
```

As another use case, suppose you have user records that consist of a name and email address, followed by an arbitrary number of phone numbers. You could unpack the records like this:

```
>>> record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')
>>> name, email, *phone_numbers = user_record
>>> name
'Dave'
>>> email
'dave@example.com'
>>> phone_numbers
['773-555-1212', '847-555-1212']
>>>
```

It’s worth noting that the `phone_numbers` variable will always be a list, regardless of how many phone numbers are unpacked (including none). Thus, any code that uses `phone_numbers` won’t have to account for the possibility that it might not be a list or perform any kind of additional type checking.

The starred variable can also be the first one in the list. For example, say you have a sequence of values representing your company’s sales figures for the last eight quarters. If you want to see how the most recent quarter stacks up to the average of the first seven, you could do something like this:

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```

Here’s a view of the operation from the Python interpreter:

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

## Discussion

Extended iterable unpacking is tailor-made for unpacking iterables of unknown or arbitrary length. Oftentimes, these iterables have some known component or pattern in their construction (e.g. “everything after element 1 is a phone number”), and star unpacking lets the developer leverage those patterns easily instead of performing acrobatics to get at the relevant elements in the iterable.

It is worth noting that the star syntax can be especially useful when iterating over a sequence of tuples of varying length. For example, perhaps a sequence of tagged tuples:

```
records = [
    ('foo', 1, 2),
    ('bar', 'hello'),
    ('foo', 3, 4),
]

def do_foo(x, y):
    print('foo', x, y)

def do_bar(s):
    print('bar', s)

for tag, *args in records:
    if tag == 'foo':
        do_foo(*args)
    elif tag == 'bar':
        do_bar(*args)
```

Star unpacking can also be useful when combined with certain kinds of string processing operations, such as splitting. For example:

```
>>> line = 'nobody::-2::-2:Unprivileged User:/var/empty:/usr/bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

Sometimes you might want to unpack values and throw them away. You can’t just specify a bare `*` when unpacking, but you could use a common throwaway variable name, such as `_` or `ign` (ignored). For example:

```
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *_ , (*_, year) = record
>>> name
'ACME'
>>> year
2012
>>>
```

There is a certain similarity between star unpacking and list-processing features of various functional languages. For example, if you have a list, you can easily split it into head and tail components like this:

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

One could imagine writing functions that perform such splitting in order to carry out some kind of clever recursive algorithm. For example:

```
>>> def sum(items):
...     head, *tail = items
...     return head + sum(tail) if tail else head
...
>>> sum(items)
36
>>>
```

However, be aware that recursion really isn't a strong Python feature due to the inherent recursion limit. Thus, this last example might be nothing more than an academic curiosity in practice.

## 1.3. Keeping the Last N Items

### Problem

You want to keep a limited history of the last few items seen during iteration or during some other kind of processing.

### Solution

Keeping a limited history is a perfect use for a `collections.deque`. For example, the following code performs a simple text match on a sequence of lines and yields the matching line along with the previous N lines of context when found:

```

from collections import deque

def search(lines, pattern, history=5):
    previous_lines = deque(maxlen=history)
    for line in lines:
        if pattern in line:
            yield line, previous_lines
        previous_lines.append(line)

# Example use on a file
if __name__ == '__main__':
    with open('somefile.txt') as f:
        for line, prevlines in search(f, 'python', 5):
            for pline in prevlines:
                print(pline, end='')
            print(line, end='')
            print('- '*20)

```

## Discussion

When writing code to search for items, it is common to use a generator function involving `yield`, as shown in this recipe's solution. This decouples the process of searching from the code that uses the results. If you're new to generators, see [Recipe 4.3](#).

Using `deque(maxlen=N)` creates a fixed-sized queue. When new items are added and the queue is full, the oldest item is automatically removed. For example:

```

>>> q = deque(maxlen=3)
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)

```

Although you could manually perform such operations on a list (e.g., appending, deleting, etc.), the queue solution is far more elegant and runs a lot faster.

More generally, a deque can be used whenever you need a simple queue structure. If you don't give it a maximum size, you get an unbounded queue that lets you append and pop items on either end. For example:

```

>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q

```



```

deque([1, 2, 3])
>>> q.appendleft(4)
>>> q
deque([4, 1, 2, 3])
>>> q.pop()
3
>>> q
deque([4, 1, 2])
>>> q.popleft()
4

```

Adding or popping items from either end of a queue has  $O(1)$  complexity. This is unlike a list where inserting or removing items from the front of the list is  $O(N)$ .

## 1.4. Finding the Largest or Smallest N Items

### Problem

You want to make a list of the largest or smallest  $N$  items in a collection.

### Solution

The `heapq` module has two functions—`nlargest()` and `nsmallest()`—that do exactly what you want. For example:

```

import heapq

nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums)) # Prints [42, 37, 23]
print(heapq.nsmallest(3, nums)) # Prints [-4, 1, 2]

```

Both functions also accept a key parameter that allows them to be used with more complicated data structures. For example:

```

portfolio = [
    {'name': 'IBM', 'shares': 100, 'price': 91.1},
    {'name': 'AAPL', 'shares': 50, 'price': 543.22},
    {'name': 'FB', 'shares': 200, 'price': 21.09},
    {'name': 'HPQ', 'shares': 35, 'price': 31.75},
    {'name': 'YHOO', 'shares': 45, 'price': 16.35},
    {'name': 'ACME', 'shares': 75, 'price': 115.65}
]

cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])

```

### Discussion

If you are looking for the  $N$  smallest or largest items and  $N$  is small compared to the overall size of the collection, these functions provide superior performance. Underneath

the covers, they work by first converting the data into a list where items are ordered as a heap. For example:

```
>>> nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
>>> import heapq
>>> heap = list(nums)
>>> heapq.heapify(heap)
>>> heap
[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
>>>
```

The most important feature of a heap is that `heap[0]` is always the smallest item. Moreover, subsequent items can be easily found using the `heapq.heappop()` method, which pops off the first item and replaces it with the next smallest item (an operation that requires  $O(\log N)$  operations where  $N$  is the size of the heap). For example, to find the three smallest items, you would do this:

```
>>> heapq.heappop(heap)
-4
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
2
```

The `nlargest()` and `nsmallest()` functions are most appropriate if you are trying to find a relatively small number of items. If you are simply trying to find the single smallest or largest item ( $N=1$ ), it is faster to use `min()` and `max()`. Similarly, if  $N$  is about the same size as the collection itself, it is usually faster to sort it first and take a slice (i.e., use `sorted(items)[:N]` or `sorted(items)[-N:]`). It should be noted that the actual implementation of `nlargest()` and `nsmallest()` is adaptive in how it operates and will carry out some of these optimizations on your behalf (e.g., using sorting if  $N$  is close to the same size as the input).

Although it's not necessary to use this recipe, the implementation of a heap is an interesting and worthwhile subject of study. This can usually be found in any decent book on algorithms and data structures. The documentation for the `heapq` module also discusses the underlying implementation details.

## 1.5. Implementing a Priority Queue

### Problem

You want to implement a queue that sorts items by a given priority and always returns the item with the highest priority on each pop operation.

## Solution

The following class uses the `heapq` module to implement a simple priority queue:

```
import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0

    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._queue)[-1]
```

Here is an example of how it might be used:

```
>>> class Item:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
>>>
```

Observe how the first `pop()` operation returned the item with the highest priority. Also observe how the two items with the same priority (foo and grok) were returned in the same order in which they were inserted into the queue.

## Discussion

The core of this recipe concerns the use of the `heapq` module. The functions `heapq.heapush()` and `heapq.heappop()` insert and remove items from a list `_queue` in a way such that the first item in the list has the smallest priority (as discussed in [Recipe 1.4](#)). The `heappop()` method always returns the “smallest” item, so that is the key to making the

queue pop the correct items. Moreover, since the push and pop operations have  $O(\log N)$  complexity where  $N$  is the number of items in the heap, they are fairly efficient even for fairly large values of  $N$ .

In this recipe, the queue consists of tuples of the form `(-priority, index, item)`. The `priority` value is negated to get the queue to sort items from highest priority to lowest priority. This is opposite of the normal heap ordering, which sorts from lowest to highest value.

The role of the `index` variable is to properly order items with the same priority level. By keeping a constantly increasing index, the items will be sorted according to the order in which they were inserted. However, the index also serves an important role in making the comparison operations work for items that have the same priority level.

To elaborate on that, instances of `Item` in the example can't be ordered. For example:

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

If you make `(priority, item)` tuples, they can be compared as long as the priorities are different. However, if two tuples with equal priorities are compared, the comparison fails as before. For example:

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
>>> a < b
True
>>> c = (1, Item('grok'))
>>> a < c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

By introducing the extra index and making `(priority, index, item)` tuples, you avoid this problem entirely since no two tuples will ever have the same value for `index` (and Python never bothers to compare the remaining tuple values once the result of comparison can be determined):

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
>>> a < c
```

```
True
>>>
```

If you want to use this queue for communication between threads, you need to add appropriate locking and signaling. See [Recipe 12.3](#) for an example of how to do this.

The documentation for the `heapq` module has further examples and discussion concerning the theory and implementation of heaps.

## 1.6. Mapping Keys to Multiple Values in a Dictionary

### Problem

You want to make a dictionary that maps keys to more than one value (a so-called “multidict”).

### Solution

A dictionary is a mapping where each key is mapped to a single value. If you want to map keys to multiple values, you need to store the multiple values in another container such as a list or set. For example, you might make dictionaries like this:

```
d = {
    'a' : [1, 2, 3],
    'b' : [4, 5]
}

e = {
    'a' : {1, 2, 3},
    'b' : {4, 5}
}
```

The choice of whether or not to use lists or sets depends on intended use. Use a list if you want to preserve the insertion order of the items. Use a set if you want to eliminate duplicates (and don’t care about the order).

To easily construct such dictionaries, you can use `defaultdict` in the `collections` module. A feature of `defaultdict` is that it automatically initializes the first value so you can simply focus on adding items. For example:

```
from collections import defaultdict

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
...

d = defaultdict(set)
```

```

d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
...

```

One caution with `defaultdict` is that it will automatically create dictionary entries for keys accessed later on (even if they aren't currently found in the dictionary). If you don't want this behavior, you might use `setdefault()` on an ordinary dictionary instead. For example:

```

d = {}      # A regular dictionary
d.setdefault('a', []).append(1)
d.setdefault('a', []).append(2)
d.setdefault('b', []).append(4)
...

```

However, many programmers find `setdefault()` to be a little unnatural—not to mention the fact that it always creates a new instance of the initial value on each invocation (the empty list `[]` in the example).

## Discussion

In principle, constructing a multivalued dictionary is simple. However, initialization of the first value can be messy if you try to do it yourself. For example, you might have code that looks like this:

```

d = {}
for key, value in pairs:
    if key not in d:
        d[key] = []
    d[key].append(value)

```

Using a `defaultdict` simply leads to much cleaner code:

```

d = defaultdict(list)
for key, value in pairs:
    d[key].append(value)

```

This recipe is strongly related to the problem of grouping records together in data processing problems. See [Recipe 1.15](#) for an example.

## 1.7. Keeping Dictionaries in Order

### Problem

You want to create a dictionary, and you also want to control the order of items when iterating or serializing.

## Solution

To control the order of items in a dictionary, you can use an `OrderedDict` from the `collections` module. It exactly preserves the original insertion order of data when iterating. For example:

```
from collections import OrderedDict

d = OrderedDict()
d['foo'] = 1
d['bar'] = 2
d['spam'] = 3
d['grok'] = 4

# Outputs "foo 1", "bar 2", "spam 3", "grok 4"
for key in d:
    print(key, d[key])
```

An `OrderedDict` can be particularly useful when you want to build a mapping that you may want to later serialize or encode into a different format. For example, if you want to precisely control the order of fields appearing in a JSON encoding, first building the data in an `OrderedDict` will do the trick:

```
>>> import json
>>> json.dumps(d)
'{"foo": 1, "bar": 2, "spam": 3, "grok": 4}'
>>>
```

## Discussion

An `OrderedDict` internally maintains a doubly linked list that orders the keys according to insertion order. When a new item is first inserted, it is placed at the end of this list. Subsequent reassignment of an existing key doesn't change the order.

Be aware that the size of an `OrderedDict` is more than twice as large as a normal dictionary due to the extra linked list that's created. Thus, if you are going to build a data structure involving a large number of `OrderedDict` instances (e.g., reading 100,000 lines of a CSV file into a list of `OrderedDict` instances), you would need to study the requirements of your application to determine if the benefits of using an `OrderedDict` outweighed the extra memory overhead.

## 1.8. Calculating with Dictionaries

### Problem

You want to perform various calculations (e.g., minimum value, maximum value, sorting, etc.) on a dictionary of data.

## Solution

Consider a dictionary that maps stock names to prices:

```
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}
```

In order to perform useful calculations on the dictionary contents, it is often useful to invert the keys and values of the dictionary using `zip()`. For example, here is how to find the minimum and maximum price and stock name:

```
min_price = min(zip(prices.values(), prices.keys()))
# min_price is (10.75, 'FB')

max_price = max(zip(prices.values(), prices.keys()))
# max_price is (612.78, 'AAPL')
```

Similarly, to rank the data, use `zip()` with `sorted()`, as in the following:

```
prices_sorted = sorted(zip(prices.values(), prices.keys()))
# prices_sorted is [(10.75, 'FB'), (37.2, 'HPQ'),
#                  (45.23, 'ACME'), (205.55, 'IBM'),
#                  (612.78, 'AAPL')]
```

When doing these calculations, be aware that `zip()` creates an iterator that can only be consumed once. For example, the following code is an error:

```
prices_and_names = zip(prices.values(), prices.keys())
print(min(prices_and_names)) # OK
print(max(prices_and_names)) # ValueError: max() arg is an empty sequence
```

## Discussion

If you try to perform common data reductions on a dictionary, you'll find that they only process the keys, not the values. For example:

```
min(prices) # Returns 'AAPL'
max(prices) # Returns 'IBM'
```

This is probably not what you want because you're actually trying to perform a calculation involving the dictionary values. You might try to fix this using the `values()` method of a dictionary:

```
min(prices.values()) # Returns 10.75
max(prices.values()) # Returns 612.78
```



Unfortunately, this is often not exactly what you want either. For example, you may want to know information about the corresponding keys (e.g., which stock has the lowest price?).

You can get the key corresponding to the min or max value if you supply a key function to `min()` and `max()`. For example:

```
min(prices, key=lambda k: prices[k]) # Returns 'FB'
max(prices, key=lambda k: prices[k]) # Returns 'AAPL'
```

However, to get the minimum value, you'll need to perform an extra lookup step. For example:

```
min_value = prices[min(prices, key=lambda k: prices[k])]
```

The solution involving `zip()` solves the problem by “inverting” the dictionary into a sequence of (value, key) pairs. When performing comparisons on such tuples, the value element is compared first, followed by the key. This gives you exactly the behavior that you want and allows reductions and sorting to be easily performed on the dictionary contents using a single statement.

It should be noted that in calculations involving (value, key) pairs, the key will be used to determine the result in instances where multiple entries happen to have the same value. For instance, in calculations such as `min()` and `max()`, the entry with the smallest or largest key will be returned if there happen to be duplicate values. For example:

```
>>> prices = { 'AAA' : 45.23, 'ZZZ' : 45.23 }
>>> min(zip(prices.values(), prices.keys()))
(45.23, 'AAA')
>>> max(zip(prices.values(), prices.keys()))
(45.23, 'ZZZ')
>>>
```

## 1.9. Finding Commonalities in Two Dictionaries

### Problem

You have two dictionaries and want to find out what they might have in common (same keys, same values, etc.).

### Solution

Consider two dictionaries:

```
a = {
    'x' : 1,
    'y' : 2,
    'z' : 3
}
```

```
b = {
    'w' : 10,
    'x' : 11,
    'y' : 2
}
```

To find out what the two dictionaries have in common, simply perform common set operations using the `keys()` or `items()` methods. For example:

```
# Find keys in common
a.keys() & b.keys()  # { 'x', 'y' }

# Find keys in a that are not in b
a.keys() - b.keys()  # { 'z' }

# Find (key,value) pairs in common
a.items() & b.items() # { ('y', 2) }
```

These kinds of operations can also be used to alter or filter dictionary contents. For example, suppose you want to make a new dictionary with selected keys removed. Here is some sample code using a dictionary comprehension:

```
# Make a new dictionary with certain keys removed
c = {key:a[key] for key in a.keys() - {'z', 'w'}}
# c is {'x': 1, 'y': 2}
```

## Discussion

A dictionary is a mapping between a set of keys and values. The `keys()` method of a dictionary returns a keys-view object that exposes the keys. A little-known feature of keys views is that they also support common set operations such as unions, intersections, and differences. Thus, if you need to perform common set operations with dictionary keys, you can often just use the keys-view objects directly without first converting them into a set.

The `items()` method of a dictionary returns an items-view object consisting of (key, value) pairs. This object supports similar set operations and can be used to perform operations such as finding out which key-value pairs two dictionaries have in common.

Although similar, the `values()` method of a dictionary does not support the set operations described in this recipe. In part, this is due to the fact that unlike keys, the items contained in a values view aren't guaranteed to be unique. This alone makes certain set operations of questionable utility. However, if you must perform such calculations, they can be accomplished by simply converting the values to a set first.

# 1.10. Removing Duplicates from a Sequence while Maintaining Order

## Problem

You want to eliminate the duplicate values in a sequence, but preserve the order of the remaining items.

## Solution

If the values in the sequence are hashable, the problem can be easily solved using a set and a generator. For example:

```
def dedupe(items):
    seen = set()
    for item in items:
        if item not in seen:
            yield item
            seen.add(item)
```

Here is an example of how to use your function:

```
>>> a = [1, 5, 2, 1, 9, 1, 5, 10]
>>> list(dedupe(a))
[1, 5, 2, 9, 10]
>>>
```

This only works if the items in the sequence are hashable. If you are trying to eliminate duplicates in a sequence of unhashable types (such as dicts), you can make a slight change to this recipe, as follows:

```
def dedupe(items, key=None):
    seen = set()
    for item in items:
        val = item if key is None else key(item)
        if val not in seen:
            yield item
            seen.add(val)
```

Here, the purpose of the key argument is to specify a function that converts sequence items into a hashable type for the purposes of duplicate detection. Here's how it works:

```
>>> a = [{'x':1, 'y':2}, {'x':1, 'y':3}, {'x':1, 'y':2}, {'x':2, 'y':4}]
>>> list(dedupe(a, key=lambda d: (d['x'],d['y'])))
[{'x': 1, 'y': 2}, {'x': 1, 'y': 3}, {'x': 2, 'y': 4}]
>>> list(dedupe(a, key=lambda d: d['x']))
[{'x': 1, 'y': 2}, {'x': 2, 'y': 4}]
>>>
```

This latter solution also works nicely if you want to eliminate duplicates based on the value of a single field or attribute or a larger data structure.

## Discussion

If all you want to do is eliminate duplicates, it is often easy enough to make a set. For example:

```
>>> a
[1, 5, 2, 1, 9, 1, 5, 10]
>>> set(a)
{1, 2, 10, 5, 9}
>>>
```

However, this approach doesn't preserve any kind of ordering. So, the resulting data will be scrambled afterward. The solution shown avoids this.

The use of a generator function in this recipe reflects the fact that you might want the function to be extremely general purpose—not necessarily tied directly to list processing. For example, if you want to read a file, eliminating duplicate lines, you could simply do this:

```
with open(somefile, 'r') as f:
    for line in dedupe(f):
        ...
```

The specification of a key function mimics similar functionality in built-in functions such as `sorted()`, `min()`, and `max()`. For instance, see Recipes 1.8 and 1.13.

## 1.11. Naming a Slice

### Problem

Your program has become an unreadable mess of hardcoded slice indices and you want to clean it up.

### Solution

Suppose you have some code that is pulling specific data fields out of a record string with fixed fields (e.g., from a flat file or similar format):

```
##### 0123456789012345678901234567890123456789012345678901234567890'
record = '.....100.....513.25.....'
cost = int(record[20:32]) * float(record[40:48])
```

Instead of doing that, why not name the slices like this?

```
SHARES = slice(20,32)
PRICE = slice(40,48)

cost = int(record[SHARES]) * float(record[PRICE])
```

In the latter version, you avoid having a lot of mysterious hardcoded indices, and what you're doing becomes much clearer.

## Discussion

As a general rule, writing code with a lot of hardcoded index values leads to a readability and maintenance mess. For example, if you come back to the code a year later, you'll look at it and wonder what you were thinking when you wrote it. The solution shown is simply a way of more clearly stating what your code is actually doing.

In general, the built-in `slice()` creates a slice object that can be used anywhere a slice is allowed. For example:

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10,11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del items[a]
>>> items
[0, 1, 4, 5, 6]
```

If you have a `slice` instance `s`, you can get more information about it by looking at its `s.start`, `s.stop`, and `s.step` attributes, respectively. For example:

```
>>> a = slice(10, 50, 2)
>>> a.start
10
>>> a.stop
50
>>> a.step
2
>>>
```

In addition, you can map a slice onto a sequence of a specific size by using its `indices(size)` method. This returns a tuple (`start`, `stop`, `step`) where all values have been suitably limited to fit within bounds (as to avoid `IndexError` exceptions when indexing). For example:

```
>>> s = 'HelloWorld'
>>> a.indices(len(s))
(5, 10, 2)
>>> for i in range(*a.indices(len(s))):
...     print(s[i])
...
W
r
```

```
d
>>>
```

## 1.12. Determining the Most Frequently Occurring Items in a Sequence

### Problem

You have a sequence of items, and you'd like to determine the most frequently occurring items in the sequence.

### Solution

The `collections.Counter` class is designed for just such a problem. It even comes with a handy `most_common()` method that will give you the answer.

To illustrate, let's say you have a list of words and you want to find out which words occur most often. Here's how you would do it:

```
words = [
    'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
    'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around', 'the',
    'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'into',
    'my', 'eyes', "you're", 'under'
]

from collections import Counter
word_counts = Counter(words)
top_three = word_counts.most_common(3)
print(top_three)
# Outputs [('eyes', 8), ('the', 5), ('look', 4)]
```

### Discussion

As input, `Counter` objects can be fed any sequence of hashable input items. Under the covers, a `Counter` is a dictionary that maps the items to the number of occurrences. For example:

```
>>> word_counts['not']
1
>>> word_counts['eyes']
8
>>>
```

If you want to increment the count manually, simply use addition:

```
>>> morewords = ['why', 'are', 'you', 'not', 'looking', 'in', 'my', 'eyes']
>>> for word in morewords:
...     word_counts[word] += 1
```

```
...
>>> word_counts['eyes']
9
>>>
```

Or, alternatively, you could use the `update()` method:

```
>>> word_counts.update(morewords)
>>>
```

A little-known feature of `Counter` instances is that they can be easily combined using various mathematical operations. For example:

```
>>> a = Counter(words)
>>> b = Counter(morewords)
>>> a
Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2,
        'you're': 1, 'don't': 1, 'under': 1, 'not': 1})
>>> b
Counter({'eyes': 1, 'looking': 1, 'are': 1, 'in': 1, 'not': 1, 'you': 1,
        'my': 1, 'why': 1})

>>> # Combine counts
>>> c = a + b
>>> c
Counter({'eyes': 9, 'the': 5, 'look': 4, 'my': 4, 'into': 3, 'not': 2,
        'around': 2, 'you're': 1, 'don't': 1, 'in': 1, 'why': 1,
        'looking': 1, 'are': 1, 'under': 1, 'you': 1})

>>> # Subtract counts
>>> d = a - b
>>> d
Counter({'eyes': 7, 'the': 5, 'look': 4, 'into': 3, 'my': 2, 'around': 2,
        'you're': 1, 'don't': 1, 'under': 1})
>>>
```

Needless to say, `Counter` objects are a tremendously useful tool for almost any kind of problem where you need to tabulate and count data. You should prefer this over manually written solutions involving dictionaries.

## 1.13. Sorting a List of Dictionaries by a Common Key

### Problem

You have a list of dictionaries and you would like to sort the entries according to one or more of the dictionary values.

## Solution

Sorting this type of structure is easy using the `operator` module's `itemgetter` function. Let's say you've queried a database table to get a listing of the members on your website, and you receive the following data structure in return:

```
rows = [
    {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},
    {'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
    {'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
    {'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
]
```

It's fairly easy to output these rows ordered by any of the fields common to all of the dictionaries. For example:

```
from operator import itemgetter

rows_by_fname = sorted(rows, key=itemgetter('fname'))
rows_by_uid = sorted(rows, key=itemgetter('uid'))

print(rows_by_fname)
print(rows_by_uid)
```

The preceding code would output the following:

```
[{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'John', 'uid': 1001, 'lname': 'Cleese'}]

[{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'Big', 'uid': 1004, 'lname': 'Jones'}]
```

The `itemgetter()` function can also accept multiple keys. For example, this code

```
rows_by_lfname = sorted(rows, key=itemgetter('lname', 'fname'))
print(rows_by_lfname)
```

Produces output like this:

```
[{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
 {'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'}]
```

## Discussion

In this example, `rows` is passed to the built-in `sorted()` function, which accepts a key-word argument `key`. This argument is expected to be a callable that accepts a single item



from `rows` as input and returns a value that will be used as the basis for sorting. The `itemgetter()` function creates just such a callable.

The operator `itemgetter()` function takes as arguments the lookup indices used to extract the desired values from the records in `rows`. It can be a dictionary key name, a numeric list element, or any value that can be fed to an object's `__getitem__()` method. If you give multiple indices to `itemgetter()`, the callable it produces will return a tuple with all of the elements in it, and `sorted()` will order the output according to the sorted order of the tuples. This can be useful if you want to simultaneously sort on multiple fields (such as last and first name, as shown in the example).

The functionality of `itemgetter()` is sometimes replaced by `lambda` expressions. For example:

```
rows_by_fname = sorted(rows, key=lambda r: r['fname'])
rows_by_lfname = sorted(rows, key=lambda r: (r['lname'], r['fname']))
```

This solution often works just fine. However, the solution involving `itemgetter()` typically runs a bit faster. Thus, you might prefer it if performance is a concern.

Last, but not least, don't forget that the technique shown in this recipe can be applied to functions such as `min()` and `max()`. For example:

```
>>> min(rows, key=itemgetter('uid'))
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}
>>> max(rows, key=itemgetter('uid'))
{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
>>>
```

## 1.14. Sorting Objects Without Native Comparison Support

### Problem

You want to sort objects of the same class, but they don't natively support comparison operations.

### Solution

The built-in `sorted()` function takes a key argument that can be passed a callable that will return some value in the object that `sorted` will use to compare the objects. For example, if you have a sequence of `User` instances in your application, and you want to sort them by their `user_id` attribute, you would supply a callable that takes a `User` instance as input and returns the `user_id`. For example:

```
>>> class User:
...     def __init__(self, user_id):
...         self.user_id = user_id
```

```

...     def __repr__(self):
...         return 'User({})'.format(self.user_id)
...
>>> users = [User(23), User(3), User(99)]
>>> users
[User(23), User(3), User(99)]
>>> sorted(users, key=lambda u: u.user_id)
[User(3), User(23), User(99)]
>>>

```

Instead of using `lambda`, an alternative approach is to use `operator.attrgetter()`:

```

>>> from operator import attrgetter
>>> sorted(users, key=attrgetter('user_id'))
[User(3), User(23), User(99)]
>>>

```

## Discussion

The choice of whether or not to use `lambda` or `attrgetter()` may be one of personal preference. However, `attrgetter()` is often a tad bit faster and also has the added feature of allowing multiple fields to be extracted simultaneously. This is analogous to the use of `operator.itemgetter()` for dictionaries (see [Recipe 1.13](#)). For example, if `User` instances also had a `first_name` and `last_name` attribute, you could perform a sort like this:

```
by_name = sorted(users, key=attrgetter('last_name', 'first_name'))
```

It is also worth noting that the technique used in this recipe can be applied to functions such as `min()` and `max()`. For example:

```

>>> min(users, key=attrgetter('user_id'))
User(3)
>>> max(users, key=attrgetter('user_id'))
User(99)
>>>

```

## 1.15. Grouping Records Together Based on a Field

### Problem

You have a sequence of dictionaries or instances and you want to iterate over the data in groups based on the value of a particular field, such as `date`.

### Solution

The `itertools.groupby()` function is particularly useful for grouping data together like this. To illustrate, suppose you have the following list of dictionaries:

```
rows = [
    {'address': '5412 N CLARK', 'date': '07/01/2012'},
    {'address': '5148 N CLARK', 'date': '07/04/2012'},
    {'address': '5800 E 58TH', 'date': '07/02/2012'},
    {'address': '2122 N CLARK', 'date': '07/03/2012'},
    {'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'},
    {'address': '1060 W ADDISON', 'date': '07/02/2012'},
    {'address': '4801 N BROADWAY', 'date': '07/01/2012'},
    {'address': '1039 W GRANVILLE', 'date': '07/04/2012'},
]
```

Now suppose you want to iterate over the data in chunks grouped by date. To do it, first sort by the desired field (in this case, date) and then use `itertools.groupby()`:

```
from operator import itemgetter
from itertools import groupby

# Sort by the desired field first
rows.sort(key=itemgetter('date'))

# Iterate in groups
for date, items in groupby(rows, key=itemgetter('date')):
    print(date)
    for i in items:
        print(' ', i)
```

This produces the following output:

```
07/01/2012
    {'date': '07/01/2012', 'address': '5412 N CLARK'}
    {'date': '07/01/2012', 'address': '4801 N BROADWAY'}
07/02/2012
    {'date': '07/02/2012', 'address': '5800 E 58TH'}
    {'date': '07/02/2012', 'address': '5645 N RAVENSWOOD'}
    {'date': '07/02/2012', 'address': '1060 W ADDISON'}
07/03/2012
    {'date': '07/03/2012', 'address': '2122 N CLARK'}
07/04/2012
    {'date': '07/04/2012', 'address': '5148 N CLARK'}
    {'date': '07/04/2012', 'address': '1039 W GRANVILLE'}
```

## Discussion

The `groupby()` function works by scanning a sequence and finding sequential “runs” of identical values (or values returned by the given key function). On each iteration, it returns the value along with an iterator that produces all of the items in a group with the same value.

An important preliminary step is sorting the data according to the field of interest. Since `groupby()` only examines consecutive items, failing to sort first won’t group the records as you want.

If your goal is to simply group the data together by dates into a large data structure that allows random access, you may have better luck using `defaultdict()` to build a `multidict`, as described in [Recipe 1.6](#). For example:

```
from collections import defaultdict
rows_by_date = defaultdict(list)
for row in rows:
    rows_by_date[row['date']].append(row)
```

This allows the records for each date to be accessed easily like this:

```
>>> for r in rows_by_date['07/01/2012']:
...     print(r)
...
{'date': '07/01/2012', 'address': '5412 N CLARK'}
{'date': '07/01/2012', 'address': '4801 N BROADWAY'}
>>>
```

For this latter example, it's not necessary to sort the records first. Thus, if memory is no concern, it may be faster to do this than to first sort the records and iterate using `groupby()`.

## 1.16. Filtering Sequence Elements

### Problem

You have data inside of a sequence, and need to extract values or reduce the sequence using some criteria.

### Solution

The easiest way to filter sequence data is often to use a list comprehension. For example:

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> [n for n in mylist if n > 0]
[1, 4, 10, 2, 3]
>>> [n for n in mylist if n < 0]
[-5, -7, -1]
>>>
```

One potential downside of using a list comprehension is that it might produce a large result if the original input is large. If this is a concern, you can use generator expressions to produce the filtered values iteratively. For example:

```
>>> pos = (n for n in mylist if n > 0)
>>> pos
<generator object <genexpr> at 0x1006a0eb0>
>>> for x in pos:
...     print(x)
...
```

```

1
4
10
2
3
>>>

```

Sometimes, the filtering criteria cannot be easily expressed in a list comprehension or generator expression. For example, suppose that the filtering process involves exception handling or some other complicated detail. For this, put the filtering code into its own function and use the built-in `filter()` function. For example:

```

values = ['1', '2', '-3', '-', '4', 'N/A', '5']

def is_int(val):
    try:
        x = int(val)
        return True
    except ValueError:
        return False

ivals = list(filter(is_int, values))
print(ivals)
# Outputs ['1', '2', '-3', '4', '5']

```

`filter()` creates an iterator, so if you want to create a list of results, make sure you also use `list()` as shown.

## Discussion

List comprehensions and generator expressions are often the easiest and most straightforward ways to filter simple data. They also have the added power to transform the data at the same time. For example:

```

>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> import math
>>> [math.sqrt(n) for n in mylist if n > 0]
[1.0, 2.0, 3.1622776601683795, 1.4142135623730951, 1.7320508075688772]
>>>

```

One variation on filtering involves replacing the values that don't meet the criteria with a new value instead of discarding them. For example, perhaps instead of just finding positive values, you want to also clip bad values to fit within a specified range. This is often easily accomplished by moving the filter criterion into a conditional expression like this:

```

>>> clip_neg = [n if n > 0 else 0 for n in mylist]
>>> clip_neg
[1, 4, 0, 10, 0, 2, 3, 0]
>>> clip_pos = [n if n < 0 else 0 for n in mylist]
>>> clip_pos

```

```
[0, 0, -5, 0, -7, 0, 0, -1]
>>>
```

Another notable filtering tool is `itertools.compress()`, which takes an iterable and an accompanying Boolean selector sequence as input. As output, it gives you all of the items in the iterable where the corresponding element in the selector is `True`. This can be useful if you're trying to apply the results of filtering one sequence to another related sequence. For example, suppose you have the following two columns of data:

```
addresses = [
    '5412 N CLARK',
    '5148 N CLARK',
    '5800 E 58TH',
    '2122 N CLARK',
    '5645 N RAVENSWOOD',
    '1060 W ADDISON',
    '4801 N BROADWAY',
    '1039 W GRANVILLE',
]

counts = [ 0, 3, 10, 4, 1, 7, 6, 1]
```

Now suppose you want to make a list of all addresses where the corresponding count value was greater than 5. Here's how you could do it:

```
>>> from itertools import compress
>>> more5 = [n > 5 for n in counts]
>>> more5
[False, False, True, False, False, True, True, False]
>>> list(compress(addresses, more5))
['5800 E 58TH', '4801 N BROADWAY', '1039 W GRANVILLE']
>>>
```

The key here is to first create a sequence of Booleans that indicates which elements satisfy the desired condition. The `compress()` function then picks out the items corresponding to `True` values.

Like `filter()`, `compress()` normally returns an iterator. Thus, you need to use `list()` to turn the results into a list if desired.

## 1.17. Extracting a Subset of a Dictionary

### Problem

You want to make a dictionary that is a subset of another dictionary.

## Solution

This is easily accomplished using a dictionary comprehension. For example:

```
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}

# Make a dictionary of all prices over 200
p1 = { key:value for key, value in prices.items() if value > 200 }

# Make a dictionary of tech stocks
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:value for key,value in prices.items() if key in tech_names }
```

## Discussion

Much of what can be accomplished with a dictionary comprehension might also be done by creating a sequence of tuples and passing them to the `dict()` function. For example:

```
p1 = dict((key, value) for key, value in prices.items() if value > 200)
```

However, the dictionary comprehension solution is a bit clearer and actually runs quite a bit faster (over twice as fast when tested on the `prices` dictionary used in the example).

Sometimes there are multiple ways of accomplishing the same thing. For instance, the second example could be rewritten as:

```
# Make a dictionary of tech stocks
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:prices[key] for key in prices.keys() & tech_names }
```

However, a timing study reveals that this solution is almost 1.6 times slower than the first solution. If performance matters, it usually pays to spend a bit of time studying it. See [Recipe 14.13](#) for specific information about timing and profiling.

## 1.18. Mapping Names to Sequence Elements

### Problem

You have code that accesses list or tuple elements by position, but this makes the code somewhat difficult to read at times. You'd also like to be less dependent on position in the structure, by accessing the elements by name.

## Solution

`collections.namedtuple()` provides these benefits, while adding minimal overhead over using a normal tuple object. `collections.namedtuple()` is actually a factory method that returns a subclass of the standard Python tuple type. You feed it a type name, and the fields it should have, and it returns a class that you can instantiate, passing in values for the fields you've defined, and so on. For example:

```
>>> from collections import namedtuple
>>> Subscriber = namedtuple('Subscriber', ['addr', 'joined'])
>>> sub = Subscriber('jonesy@example.com', '2012-10-19')
>>> sub
Subscriber(addr='jonesy@example.com', joined='2012-10-19')
>>> sub.addr
'jonesy@example.com'
>>> sub.joined
'2012-10-19'
>>>
```

Although an instance of a `namedtuple` looks like a normal class instance, it is interchangeable with a tuple and supports all of the usual tuple operations such as indexing and unpacking. For example:

```
>>> len(sub)
2
>>> addr, joined = sub
>>> addr
'jonesy@example.com'
>>> joined
'2012-10-19'
>>>
```

A major use case for named tuples is decoupling your code from the position of the elements it manipulates. So, if you get back a large list of tuples from a database call, then manipulate them by accessing the positional elements, your code could break if, say, you added a new column to your table. Not so if you first cast the returned tuples to `namedtuples`.

To illustrate, here is some code using ordinary tuples:

```
def compute_cost(records):
    total = 0.0
    for rec in records:
        total += rec[1] * rec[2]
    return total
```

References to positional elements often make the code a bit less expressive and more dependent on the structure of the records. Here is a version that uses a `namedtuple`:

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price'])
```



```
def compute_cost(records):
    total = 0.0
    for rec in records:
        s = Stock(*rec)
        total += s.shares * s.price
    return total
```

Naturally, you can avoid the explicit conversion to the `Stock` namedtuple if the records sequence in the example already contained such instances.

## Discussion

One possible use of a namedtuple is as a replacement for a dictionary, which requires more space to store. Thus, if you are building large data structures involving dictionaries, use of a namedtuple will be more efficient. However, be aware that unlike a dictionary, a namedtuple is immutable. For example:

```
>>> s = Stock('ACME', 100, 123.45)
>>> s
Stock(name='ACME', shares=100, price=123.45)
>>> s.shares = 75
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

If you need to change any of the attributes, it can be done using the `_replace()` method of a namedtuple instance, which makes an entirely new namedtuple with specified values replaced. For example:

```
>>> s = s._replace(shares=75)
>>> s
Stock(name='ACME', shares=75, price=123.45)
>>>
```

A subtle use of the `_replace()` method is that it can be a convenient way to populate named tuples that have optional or missing fields. To do this, you make a prototype tuple containing the default values and then use `_replace()` to create new instances with values replaced. For example:

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price', 'date', 'time'])

# Create a prototype instance
stock_prototype = Stock('', 0, 0.0, None, None)

# Function to convert a dictionary to a Stock
def dict_to_stock(s):
    return stock_prototype._replace(**s)
```

Here is an example of how this code would work:

```
>>> a = {'name': 'ACME', 'shares': 100, 'price': 123.45}
>>> dict_to_stock(a)
Stock(name='ACME', shares=100, price=123.45, date=None, time=None)
>>> b = {'name': 'ACME', 'shares': 100, 'price': 123.45, 'date': '12/17/2012'}
>>> dict_to_stock(b)
Stock(name='ACME', shares=100, price=123.45, date='12/17/2012', time=None)
>>>
```

Last, but not least, it should be noted that if your goal is to define an efficient data structure where you will be changing various instance attributes, using `namedtuple` is not your best choice. Instead, consider defining a class using `__slots__` instead (see [Recipe 8.4](#)).

## 1.19. Transforming and Reducing Data at the Same Time

### Problem

You need to execute a reduction function (e.g., `sum()`, `min()`, `max()`), but first need to transform or filter the data.

### Solution

A very elegant way to combine a data reduction and a transformation is to use a generator-expression argument. For example, if you want to calculate the sum of squares, do the following:

```
nums = [1, 2, 3, 4, 5]
s = sum(x * x for x in nums)
```

Here are a few other examples:

```
# Determine if any .py files exist in a directory
import os
files = os.listdir('dirname')
if any(name.endswith('.py') for name in files):
    print('There be python!')
else:
    print('Sorry, no python.')
```

```
# Output a tuple as CSV
s = ('ACME', 50, 123.45)
print(','.join(str(x) for x in s))
```

```
# Data reduction across fields of a data structure
portfolio = [
    {'name': 'GOOG', 'shares': 50},
    {'name': 'YHOO', 'shares': 75},
    {'name': 'AOL', 'shares': 20},
```