

1. [Preface](#)
  1. [New for the Second Edition](#)
  2. [Conventions Used in This Book](#)
  3. [Using Code Examples](#)
  4. [Safari® Books Online](#)
  5. [How to Contact Us](#)
  6. [Acknowledgements](#)
    1. [In Memorium: John D. Hunter \(1968-2012\)](#)
    2. [Acknowledgements for the 2nd Edition \(2016\)](#)
    3. [Acknowledgements for the 1st Edition \(2012\)](#)
2. [1. Preliminaries](#)
  1. [What Is This Book About?](#)
    1. [What kinds of data?](#)
  2. [Why Python for Data Analysis?](#)
    1. [Python as Glue](#)
    2. [Solving the “Two-Language” Problem](#)
    3. [Why Not Python?](#)
  3. [Essential Python Libraries](#)
    1. [NumPy](#)
    2. [pandas](#)
    3. [matplotlib](#)
    4. [IPython and Jupyter](#)
    5. [SciPy](#)
    6. [scikit-learn](#)
    7. [statsmodels](#)
  4. [Installation and Setup](#)
    1. [Windows](#)
    2. [Apple \(OS X, macOS\)](#)
    3. [GNU/Linux](#)
    4. [Installing or updating Python packages](#)
    5. [Python 2 and Python 3](#)
    6. [Integrated Development Environments \(IDEs\) and Text Editors](#)
  5. [Community and Conferences](#)
  6. [Navigating This Book](#)
    1. [Code Examples](#)
    2. [Data for Examples](#)

- 3. [Import Conventions](#)
  - 4. [Jargon](#)
- 3. [2. Python Language Basics, IPython, and Jupyter Notebooks](#)
  - 1. [The Python Interpreter](#)
  - 2. [IPython Basics](#)
    - 1. [Running the IPython Shell](#)
    - 2. [Running the Jupyter Notebook](#)
    - 3. [Tab Completion](#)
    - 4. [Introspection](#)
    - 5. [The %run Command](#)
    - 6. [Executing Code from the Clipboard](#)
    - 7. [Terminal Keyboard Shortcuts](#)
    - 8. [Exceptions and Tracebacks](#)
    - 9. [About Magic Commands](#)
    - 10. [Matplotlib Integration](#)
  - 3. [Python Language Basics](#)
    - 1. [Language Semantics](#)
    - 2. [Scalar Types](#)
    - 3. [Control Flow](#)
- 4. [3. Built-in Data Structures, Functions, and Files](#)
  - 1. [Data Structures and Sequences](#)
    - 1. [Tuple](#)
    - 2. [List](#)
    - 3. [Built-in Sequence Functions](#)
    - 4. [Dict](#)
    - 5. [Set](#)
    - 6. [List, Set, and Dict Comprehensions](#)
  - 2. [Functions](#)
    - 1. [Namespaces, Scope, and Local Functions](#)
    - 2. [Returning Multiple Values](#)
    - 3. [Functions Are Objects](#)
    - 4. [Anonymous \(lambda\) Functions](#)
    - 5. [Closures: Functions that Return Functions](#)
    - 6. [Extended Call Syntax with \\*args, \\*\\*kwargs](#)
    - 7. [Currying: Partial Argument Application](#)
    - 8. [Generators](#)
  - 3. [Files and the operating system](#)

1. [Bytes and Unicode with files](#)
5. [\*\*4. NumPy Basics: Arrays and Vectorized Computation\*\*](#)
  1. [The NumPy ndarray: A Multidimensional Array Object](#)
    1. [Creating ndarrays](#)
    2. [Data Types for ndarrays](#)
    3. [Operations between Arrays and Scalars](#)
    4. [Basic Indexing and Slicing](#)
    5. [Boolean Indexing](#)
    6. [Fancy Indexing](#)
    7. [Transposing Arrays and Swapping Axes](#)
  2. [Universal Functions: Fast Element-wise Array Functions](#)
  3. [Loop-free programming with arrays](#)
    1. [Expressing Conditional Logic as Array Operations](#)
    2. [Mathematical and Statistical Methods](#)
    3. [Methods for Boolean Arrays](#)
    4. [Sorting](#)
    5. [Unique and Other Set Logic](#)
  4. [File Input and Output with Arrays](#)
  5. [Linear Algebra](#)
  6. [Pseudorandom Number Generation](#)
  7. [Example: Random Walks](#)
    1. [Simulating Many Random Walks at Once](#)
6. [\*\*5. Getting Started with pandas\*\*](#)
  1. [Introduction to pandas Data Structures](#)
    1. [Series](#)
    2. [DataFrame](#)
    3. [Index Objects](#)
  2. [Essential Functionality](#)
    1. [Reindexing](#)
    2. [Dropping entries from an axis](#)
    3. [Indexing, selection, and filtering](#)
    4. [Arithmetic and data alignment](#)
    5. [Function application and mapping](#)
    6. [Sorting and ranking](#)
    7. [Axis indexes with duplicate values](#)
  3. [Summarizing and Computing Descriptive Statistics](#)
    1. [Correlation and Covariance](#)

- 2. [Unique Values, Value Counts, and Membership](#)
- 4. [Moving ahead](#)
- 7. [\*\*6. Data Loading, Storage, and File Formats\*\*](#)
  - 1. [Reading and Writing Data in Text Format](#)
    - 1. [Reading Text Files in Pieces](#)
    - 2. [Writing Data Out to Text Format](#)
    - 3. [Manually Working with Delimited Formats](#)
    - 4. [JSON Data](#)
    - 5. [XML and HTML: Web Scraping](#)
  - 2. [Binary Data Formats](#)
    - 1. [Using HDF5 Format](#)
    - 2. [Reading Microsoft Excel Files](#)
  - 3. [Interacting with Web APIs](#)
  - 4. [Interacting with Databases](#)
- 8. [\*\*7. Data Cleaning and Preparation\*\*](#)
  - 1. [Handling Missing Data](#)
    - 1. [Filtering Out Missing Data](#)
    - 2. [Filling in Missing Data](#)
  - 2. [Data Transformation](#)
    - 1. [Removing Duplicates](#)
    - 2. [Transforming Data Using a Function or Mapping](#)
    - 3. [Replacing Values](#)
    - 4. [Renaming Axis Indexes](#)
    - 5. [Discretization and Binning](#)
    - 6. [Detecting and Filtering Outliers](#)
    - 7. [Permutation and Random Sampling](#)
    - 8. [Computing Indicator/Dummy Variables](#)
  - 3. [String Manipulation](#)
    - 1. [String Object Methods](#)
    - 2. [Regular expressions](#)
    - 3. [Vectorized string functions in pandas](#)
- 9. [\*\*8. Data Wrangling: Join, Combine, and Reshape\*\*](#)
  - 1. [Hierarchical Indexing](#)
    - 1. [Reordering and Sorting Levels](#)
    - 2. [Summary Statistics by Level](#)
    - 3. [Indexing with a DataFrame's columns](#)
    - 4. [Integer Indexes](#)

- 2. [Combining and Merging Data Sets](#)
  - 1. [Database-style DataFrame Joins](#)
  - 2. [Merging on Index](#)
  - 3. [Concatenating Along an Axis](#)
  - 4. [Combining Data with Overlap](#)
- 3. [Reshaping and Pivoting](#)
  - 1. [Reshaping with Hierarchical Indexing](#)
  - 2. [Pivoting “long” to “wide” Format](#)
- 10. [9. Plotting and Visualization](#)
  - 1. [A Brief matplotlib API Primer](#)
    - 1. [Figures and Subplots](#)
    - 2. [Colors, Markers, and Line Styles](#)
    - 3. [Ticks, Labels, and Legends](#)
    - 4. [Annotations and Drawing on a Subplot](#)
    - 5. [Saving Plots to File](#)
    - 6. [matplotlib Configuration](#)
  - 2. [Plotting with pandas and seaborn](#)
    - 1. [Line Plots](#)
    - 2. [Bar Plots](#)
    - 3. [Histograms and Density Plots](#)
    - 4. [Scatter or Point Plots](#)
    - 5. [Facet grids and categorical data](#)
  - 3. [Other Python Visualization Tools](#)
- 11. [10. Data Aggregation and Group Operations](#)
  - 1. [GroupBy Mechanics](#)
    - 1. [Iterating Over Groups](#)
    - 2. [Selecting a Column or Subset of Columns](#)
    - 3. [Grouping with Dicts and Series](#)
    - 4. [Grouping with Functions](#)
    - 5. [Grouping by Index Levels](#)
  - 2. [Data Aggregation](#)
    - 1. [Column-wise and Multiple Function Application](#)
    - 2. [Returning Aggregated Data without Row Indexes](#)
  - 3. [Apply: General split-apply-combine](#)
    - 1. [Suppressing the group keys](#)
    - 2. [Quantile and Bucket Analysis](#)
    - 3. [Example: Filling Missing Values with Group-specific Values](#)

- 4. [Example: Random Sampling and Permutation](#)
- 5. [Example: Group Weighted Average and Correlation](#)
- 6. [Example: Group-wise Linear Regression](#)
- 4. [Pivot Tables and Cross-Tabulation](#)
  - 1. [Cross-Tabulations: Crosstab](#)
- 12. [11. Time Series](#)
  - 1. [Date and Time Data Types and Tools](#)
    - 1. [Converting between string and datetime](#)
  - 2. [Time Series Basics](#)
    - 1. [Indexing, Selection, Subsetting](#)
    - 2. [Time Series with Duplicate Indices](#)
  - 3. [Date Ranges, Frequencies, and Shifting](#)
    - 1. [Generating Date Ranges](#)
    - 2. [Frequencies and Date Offsets](#)
    - 3. [Shifting \(Leading and Lagging\) Data](#)
  - 4. [Time Zone Handling](#)
    - 1. [Time Zone Localization and Conversion](#)
    - 2. [Operations with Time Zone-aware Timestamp Objects](#)
    - 3. [Operations between Different Time Zones](#)
  - 5. [Periods and Period Arithmetic](#)
    - 1. [Period Frequency Conversion](#)
    - 2. [Quarterly Period Frequencies](#)
    - 3. [Converting Timestamps to Periods \(and Back\)](#)
    - 4. [Creating a PeriodIndex from Arrays](#)
  - 6. [Resampling and Frequency Conversion](#)
    - 1. [Downsampling](#)
    - 2. [Upsampling and Interpolation](#)
    - 3. [Resampling with Periods](#)
  - 7. [Moving Window Functions](#)
    - 1. [Exponentially-weighted functions](#)
    - 2. [Binary Moving Window Functions](#)
    - 3. [User-Defined Moving Window Functions](#)
- 13. [12. Advanced NumPy](#)
  - 1. [ndarray Object Internals](#)
    - 1. [NumPy dtype Hierarchy](#)
  - 2. [Advanced Array Manipulation](#)
    - 1. [Reshaping Arrays](#)

- 2. [C versus Fortran Order](#)
  - 3. [Concatenating and Splitting Arrays](#)
  - 4. [Repeating Elements: Tile and Repeat](#)
  - 5. [Fancy Indexing Equivalents: Take and Put](#)
  - 3. [Broadcasting](#)
    - 1. [Broadcasting Over Other Axes](#)
    - 2. [Setting Array Values by Broadcasting](#)
  - 4. [Advanced ufunc Usage](#)
    - 1. [ufunc Instance Methods](#)
    - 2. [Writing new ufuncs in Python](#)
  - 5. [Structured and Record Arrays](#)
    - 1. [Nested dtypes and Multidimensional Fields](#)
    - 2. [Why Use Structured Arrays?](#)
  - 6. [More About Sorting](#)
    - 1. [Indirect Sorts: argsort and lexsort](#)
    - 2. [Alternate Sort Algorithms](#)
    - 3. [Partially sorting arrays](#)
    - 4. [numpy.searchsorted: Finding elements in a Sorted Array](#)
  - 7. [Writing Fast NumPy Functions with Numba](#)
    - 1. [Creating Custom numpy.ufunc Objects with Numba](#)
  - 8. [Advanced Array Input and Output](#)
    - 1. [Memory-mapped Files](#)
    - 2. [HDF5 and Other Array Storage Options](#)
  - 9. [Performance Tips](#)
    - 1. [The Importance of Contiguous Memory](#)
14. [13. Examples Data Sets](#)
- 1. [1.usa.gov data from bit.ly](#)
    - 1. [Counting Time Zones in Pure Python](#)
    - 2. [Counting Time Zones with pandas](#)
  - 2. [MovieLens 1M Data Set](#)
    - 1. [Measuring rating disagreement](#)
  - 3. [US Baby Names 1880-2010](#)
    - 1. [Analyzing Naming Trends](#)
  - 4. [USDA Food Database](#)
  - 5. [2012 Federal Election Commission Database](#)
    - 1. [Donation Statistics by Occupation and Employer](#)
    - 2. [Bucketing Donation Amounts](#)

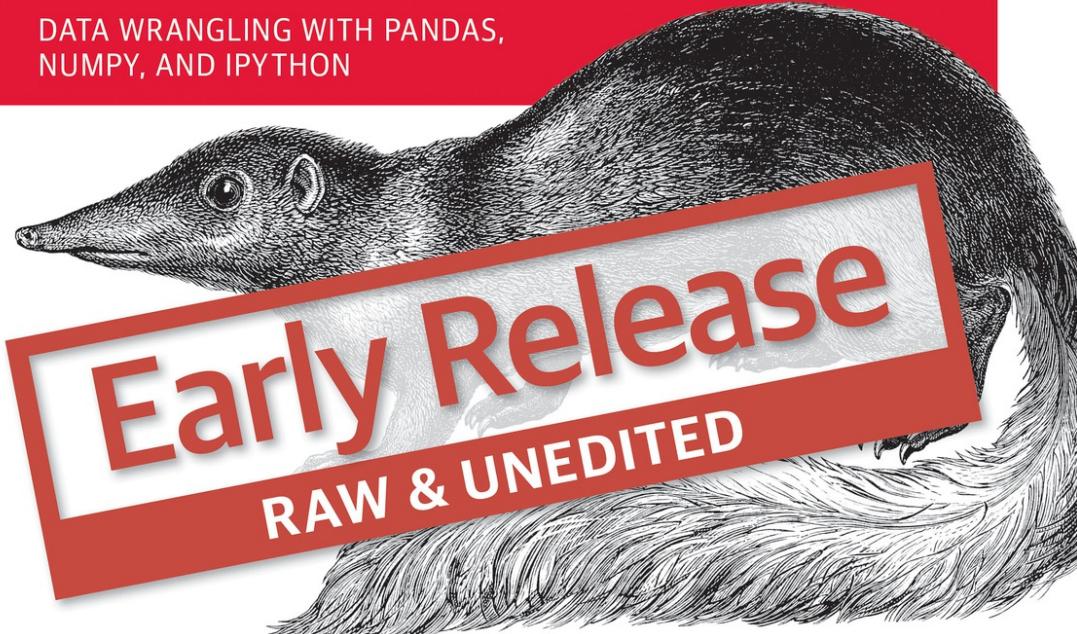
3. [Donation Statistics by State](#)
15. [14. Appendix: Advanced IPython and Jupyter](#)
  1. [Using the Command History](#)
    1. [Searching and Reusing the Command History](#)
    2. [Input and Output Variables](#)
  2. [Interacting with the Operating System](#)
    1. [Shell Commands and Aliases](#)
    2. [Directory Bookmark System](#)
  3. [Software Development Tools](#)
    1. [Interactive Debugger](#)
    2. [Timing Code: %time and %timeit](#)
    3. [Basic Profiling: %prun and %run -p](#)
    4. [Profiling a Function Line-by-Line](#)
  4. [Tips for Productive Code Development Using IPython](#)
    1. [Reloading Module Dependencies](#)
    2. [Code Design Tips](#)
  5. [Advanced IPython Features](#)
    1. [Making Your Own Classes IPython-friendly](#)
    2. [Profiles and Configuration](#)
  6. [Wrapping up](#)

O'REILLY®

2nd Edition

# Python for Data Analysis

DATA WRANGLING WITH PANDAS,  
NUMPY, AND IPYTHON



Wes McKinney

# **Python for Data Analysis**

Second Edition

Wes McKinney

# Python for Data Analysis

by Wes McKinney

Copyright © 2016 William McKinney. 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://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com) .

- Editor: Marie Beaugureau
- Production Editor: FILL IN PRODUCTION EDITOR
- Copyeditor: FILL IN COPYEDITOR
- Proofreader: FILL IN PROOFREADER
- Indexer: FILL IN INDEXER
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest
- June 2017: Second Edition

# Revision History for the Second Edition

- 2016-11-07: First Early Release
- 2017-02-27: Second Early Release
- 2017-05-24: Third Early Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491957660> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. Python for Data Analysis, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-95766-0

[FILL IN]

# Preface

# New for the Second Edition

The first edition of this book was published in 2012, during a time when open source data analysis libraries for Python (such as pandas) were very new and developing rapidly. In this updated and expanded 2nd edition, I have updated the chapters to account both for incompatible changes, deprecations, as well as new features that have occurred in the last 4 years. I added additional content to introduce tools which either did not exist in 2012 or had not matured enough to make the first cut. I have also tried to avoid writing about new or cutting-edge open source projects which may not have had a chance to mature. I would like readers of this edition to find that the content is still almost as relevant in 2020 or 2021 as it is in 2017.

Data files and related material for each chapter are hosted as a git repository on GitHub:

<http://github.com/wesm/pydata-book>

The major updates in this second edition include:

- All code (including the Python tutorial) updated for Python 3.6. The first edition used Python 2.7.
- Updated Python installation instructions for the Anaconda Python Distribution and other needed Python packages.
- Updates for pandas 1.0, released in 2017, after more than 20 major 0.x releases.
- A chapter on “advanced pandas”: advanced usage and best practices.
- A brief introduction to using statsmodels and scikit-learn.

# 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.

## **Tip**

This element signifies a tip or suggestion.

## **Note**

This element signifies a general note.

## **Caution**

This element indicates a warning or caution.

# Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at [https://github.com/oreillymedia/title\\_title](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](mailto:permissions@oreilly.com).

# Safari® Books Online

## Note

Safari Books Online ([www.safaribooksonline.com](http://www.safaribooksonline.com)) 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 [plans and pricing](#) for [enterprise](#), [government](#), and [education](#), and individuals.

Members 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 hundreds [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](mailto: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>

# Acknowledgements

This work is the product of many years of fruitful discussions, collaborations, and assistance with and from many people around the world. I'd like to thank a few of them.

# In Memorium: John D. Hunter (1968-2012)

Our dear friend and colleague John D. Hunter passed away after a battle with colon cancer on August 28, 2012. This was only a short time after I'd completed the final manuscript for this book's first edition.

John's impact and legacy in the Python scientific and data communities would be hard to overstate. In addition to developing matplotlib in the early 2000s (a time when Python was not nearly so popular), he helped shape the culture of a critical generation of open source developers who've become pillars of the Python ecosystem that we now often take for granted.

I was lucky enough to connect with John early in my open source career in January 2010, just after releasing pandas 0.1. His inspiration and mentorship helped me push forward, even in the darkest of times, with my vision for pandas and Python as a first-class data analysis language.

John was very close with Fernando Pérez and Brian Granger, pioneers of IPython, Jupyter, and many other initiatives in the Python community. We had hoped to work on a book together, the four of us, but I ended up being the one with the most free time. I am sure he would be proud of what we've accomplished, as individuals and as a community, over the last four years.

# **Acknowledgements for the 2nd Edition (2016)**

TODO

# Acknowledgements for the 1st Edition (2012)

It would have been difficult for me to write this book without the support of a large number of people.

On the O'Reilly staff, I'm very grateful for my editors Meghan Blanchette and Julie Steele who guided me through the process. Mike Loukides also worked with me in the proposal stages and helped make the book a reality.

I received a wealth of technical review from a large cast of characters. In particular, Martin Blais and Hugh Brown were incredibly helpful in improving the book's examples, clarity, and organization from cover to cover. James Long, Drew Conway, Fernando Pérez, Brian Granger, Thomas Kluyver, Adam Klein, Josh Klein, Chang She, and Stéfan van der Walt each reviewed one or more chapters, providing pointed feedback from many different perspectives.

I got many great ideas for examples and data sets from friends and colleagues in the data community, among them: Mike Dewar, Jeff Hammerbacher, James Johndrow, Kristian Lum, Adam Klein, Hilary Mason, Chang She, and Ashley Williams.

I am of course indebted to the many leaders in the open source scientific Python community who've built the foundation for my development work and gave encouragement while I was writing this book: the IPython core team (Fernando Pérez, Brian Granger, Min Ragan-Kelly, Thomas Kluyver, and others), John Hunter, Skipper Seabold, Travis Oliphant, Peter Wang, Eric Jones, Robert Kern, Josef Perktold, Francesc Alted, Chris Fonnesbeck, and too many others to mention. Several other people provided a great deal of support, ideas, and encouragement along the way: Drew Conway, Sean Taylor, Giuseppe Paleologo, Jared Lander, David Epstein, John Krowas, Joshua Bloom, Den Pilsworth, John Myles-White, and many others I've forgotten.

I'd also like to thank a number of people from my formative years. First, my former AQR colleagues who've cheered me on in my pandas work over the

years: Alex Reyfman, Michael Wong, Tim Sargent, Oktay Kurbanov, Matthew Tschantz, Roni Israelov, Michael Katz, Chris Uga, Prasad Ramanan, Ted Square, and Hoon Kim. Lastly, my academic advisors Haynes Miller (MIT) and Mike West (Duke).

I received significant help from Philip Cloud and Joris Van den Bossche in 2014 to update the book's code examples and fix some other inaccuracies due to changes in pandas.

On the personal side, Casey provided invaluable day-to-day support during the writing process, tolerating my highs and lows as I hacked together the final draft on top of an already overcommitted schedule. Lastly, my parents, Bill and Kim, taught me to always follow my dreams and to never settle for less.

# **Chapter 1. Preliminaries**

# What Is This Book About?

This book is concerned with the nuts and bolts of manipulating, processing, cleaning, and crunching data in Python. My goal is to offer a guide to the parts of the Python programming language and its data-oriented library ecosystem and tools that will equip you to become an effective data analyst. While “data analysis” is in the title of the book, the focus is specifically on Python programming, libraries, and tools as opposed to data analysis methodology. This is the Python programming you need *for* data analysis.

# What kinds of data?

When I say “data”, what am I referring to exactly? The primary focus is on *structured data*, a deliberately vague term that encompasses many different common forms of data, such as

- Tabular or spreadsheet-like data in which each column may be a different type (string, numeric, date, or otherwise). This includes most kinds of data commonly stored in relational databases or tab- or comma-delimited text files
- Multidimensional arrays (matrices)
- Multiple tables of data interrelated by key columns (what would be primary or foreign keys for a SQL user)
- Evenly or unevenly spaced time series

This is by no means a complete list. Even though it may not always be obvious, a large percentage of data sets can be transformed into a structured form that is more suitable for analysis and modeling. If not, it may be possible to extract features from a data set into a structured form. As an example, a collection of news articles could be processed into a word frequency table which could then be used to perform sentiment analysis.

Most users of spreadsheet programs like Microsoft Excel, perhaps the most widely used data analysis tool in the world, will not be strangers to these kinds of data.

# Why Python for Data Analysis?

For many people, the Python programming language has strong appeal. Since its first appearance in 1991, Python has become one of the most popular interpreted programming languages, along with Perl, Ruby, and others. Python and Ruby have become especially popular since 2005 or so for building websites using their numerous web frameworks, like Rails (Ruby) and Django (Python). Such languages are often called *scripting* languages as they can be used to quickly write small programs, or *scripts* to automate other tasks. I don't like the term "scripting language" as it carries a connotation that they cannot be used for building serious software. Among interpreted languages Python, for various historical and cultural reasons, has developed a large and active scientific computing and data analysis community. In the last ten years, Python has gone from a bleeding-edge or "at your own risk" scientific computing language to one of the most important languages for data science, machine learning, and general software development in academia and industry.

For data analysis and interactive computing and data visualization, Python will inevitably draw comparisons with other open source and commercial programming languages and tools in wide use, such as R, MATLAB, SAS, Stata, and others. In recent years, Python's improved library support (such as pandas and scikit-learn) has made it a popular choice for data analysis tasks. Combined with Python's overall strength for general purpose software engineering, it is an excellent option as a primary language for building data applications.

# Python as Glue

Part of Python's success in scientific computing is the ease of integrating C, C++, and FORTRAN code. Most modern computing environments share a similar set of legacy FORTRAN and C libraries for doing linear algebra, optimization, integration, fast fourier transforms, and other such algorithms. The same story has held true for many companies and national labs that have used Python to glue together decades worth of legacy software.

Many programs consist of small portions of code where most of the time is spent, with large amounts of “glue code” that doesn’t run often. In many cases, the execution time of the glue code is insignificant; effort is most fruitfully invested in optimizing the computational bottlenecks, sometimes by moving the code to a lower-level language like C.

In recent years, the Cython project (<http://cython.org>) has become one of the most flexible and accessible ways of both creating fast compiled extensions for Python that may also utilize pure C and C++ code.

# **Solving the “Two-Language” Problem**

In many organizations, it is common to research, prototype, and test new ideas using a more specialized computing language like SAS or R then later port those ideas to be part of a larger production system written in, say, Java, C#, or C++. What people are increasingly finding is that Python is a suitable language not only for doing research and prototyping but also building the production systems, too. Why maintain two development environments when one will suffice? I believe that more and more companies will go down this path as there are often significant organizational benefits to having both researchers and software engineers using the same set of programming tools.

# Why Not Python?

While Python is an excellent environment for building many kinds of analytical applications and general purpose systems, there are a number of uses for which Python may be less suitable.

As Python is an interpreted programming language, in general most Python code will run substantially slower than code written in a compiled language like Java or C++. As *programmer time* is often more valuable than *CPU time*, many are happy to make this tradeoff. However, in an application with very low latency or demanding resource utilization requirements (for example, a high frequency trading system), the time spent programming in a lower-level (but also lower-productivity) language like C++ to achieve the maximum possible performance might be time well spent.

Python can be a challenging language for building highly concurrent, multithreaded applications, particularly applications with many CPU-bound threads. The reason for this is that it has what is known as the *global interpreter lock* (GIL), a mechanism which prevents the interpreter from executing more than one Python instruction at a time. The technical reasons for why the GIL exists are beyond the scope of this book. While it is true that in many big data processing applications, a cluster of computers may be required to process a data set in a reasonable amount of time, there are still situations where a single-process, multithreaded system is desirable.

This is not to say that Python cannot execute truly multithreaded, parallel code. Python C extensions that use native multithreading (in C or C++) can run code in parallel without being impacted by the GIL, so long as they do not need to regularly interact with Python objects.

# **Essential Python Libraries**

For those who are less familiar with the Python data ecosystem and the libraries used throughout the book, I will give a brief overview of some of them.

# NumPy

NumPy (<http://numpy.org>), short for Numerical Python, has long been a cornerstone of numerical computing in Python. It provides the data structures, algorithms, and library glue needed for most scientific applications involving numerical data in Python. NumPy contains, among other things:

- A fast and efficient multidimensional array object *ndarray*
- Functions for performing element-wise computations with arrays or mathematical operations between arrays
- Tools for reading and writing array-based data sets to disk
- Linear algebra operations, Fourier transform, and random number generation
- A mature C API to enable Python extensions and native C or C++ code to access NumPy's data structures and computational facilities.

Beyond the fast array-processing capabilities that NumPy adds to Python, one of its primary uses in data analysis is as a container for data to be passed between algorithms and libraries. For numerical data, NumPy arrays are a much more efficient way of storing and manipulating data than the other built-in Python data structures. Also, libraries written in a lower-level language, such as C or Fortran, can operate on the data stored in a NumPy array without copying data into some other memory representation. As such, most tools for data analysis in Python either assume NumPy arrays as a primary data structure or else target seamless interoperability with NumPy.

# pandas

pandas (<http://pandas.pydata.org>) provides high level data structures and functions designed to make working with structured or tabular data fast, easy, and expressive. Since its emergence in 2010, it has helped enable Python to be a powerful and productive data analysis environment. The primary object in pandas that will be used in this book is the `DataFrame`, a tabular, column-oriented data structure with both row and column labels:

```
>>> frame
      total_bill  tip    sex   smoker  day    time  size
1  16.99       1.01  Female  No     Sun  Dinner  2
2  10.34       1.66   Male   No     Sun  Dinner  3
3  21.01       3.5    Male   No     Sun  Dinner  3
4  23.68       3.31   Male   No     Sun  Dinner  2
5  24.59       3.61  Female  No     Sun  Dinner  4
6  25.29       4.71   Male   No     Sun  Dinner  4
7   8.77        2     Male   No     Sun  Dinner  2
8  26.88       3.12   Male   No     Sun  Dinner  4
9  15.04       1.96   Male   No     Sun  Dinner  2
10 14.78       3.23   Male   No     Sun  Dinner  2
```

pandas blends the high performance array-computing ideas of NumPy with the flexible data manipulation capabilities of spreadsheets and relational databases (such as SQL). It provides sophisticated indexing functionality to make it easy to reshape, slice and dice, perform aggregations, and select subsets of data. Since data manipulation, preparation, and cleaning is such an important skill in data analysis, pandas is one of primary focuses of this book.

As a bit of background, I started building pandas in early 2008 during my tenure at AQR Capital Management, a quantitative investment management firm. At the time, I had a distinct set of requirements that were not well-addressed by any single tool at my disposal:

- Data structures with labeled axes supporting automatic or explicit data alignment. This prevents common errors resulting from misaligned data and working with differently-indexed data coming from different sources.

- Integrated time series functionality.
- The same data structures handle both time series data and non-time series data.
- Arithmetic operations and reductions (like summing across an axis) would pass on the metadata (axis labels).
- Flexible handling of missing data.
- Merge and other relational operations found in popular database databases (SQL-based, for example).

I wanted to be able to do all of these things in one place, preferably in a language well-suited to general purpose software development. Python was a good candidate language for this, but at that time there was not an integrated set of data structures and tools providing this functionality. As a result of having been built initially to solve finance and business analytics problems, pandas features especially deep time series functionality and tools well-suited for working with time-indexed data generated by business processes.

For users of the R language for statistical computing, the DataFrame name will be familiar, as the object was named after the similar R `data.frame` object. Unlike Python, data frames are built-in to the R programming language and its standard library. As a result, many features found in pandas are typically either part of the R core implementation or provided by add-on packages.

The pandas name itself is derived from *panel data*, an econometrics term for multidimensional structured data sets, and a play on the phrase *Python data analysis* itself.

# **matplotlib**

matplotlib (<http://matplotlib.org>) is the most popular Python library for producing plots and other 2D data visualizations. It was originally created by the John D. Hunter (JDH) and is now maintained by a large team of developers. It is well-suited for creating plots suitable for publication. While there are other visualization libraries available to Python programmers, matplotlib is the most widely used and as such has generally good integration with the rest of the ecosystem. I think it is a safe choice as a default visualization tool.

# IPython and Jupyter

The [IPython](#) project began in 2001 as Fernando Pérez's side project to make a better interactive Python interpreter. In the subsequent 15 years it has become one of the most important tools in the modern Python data stack. While it does not provide any computational or data analytical tools by itself, IPython is designed from the ground up to maximize your productivity in both interactive computing and software development. It encourages an *execute-explore* workflow instead of the typical *edit-compile-run* workflow of many other programming languages. It also provides easy access to your operating system's shell and file system. Since much of data analysis coding involves exploration, trial and error, and iteration, IPython can help you get the job done faster.

In 2014, Fernando and the IPython team announced the Jupyter project (<http://jupyter.org>), a broader initiative to design language-agnostic interactive computing tools. The IPython web notebook became the Jupyter notebook, with support now for over 40 programming languages. The IPython system can now be used as a *kernel* (a programming language mode) for using Python with Jupyter.

IPython itself has become a component of the much broader Jupyter (<http://jupyter.org>) open source project, provides a productive environment for interactive and exploratory computing. Its oldest and simplest “mode” is as an enhanced Python shell designed to accelerate the writing, testing, and debugging of Python code. You can also use the IPython system through the Jupyter Notebook, an interactive web-based code “notebook” offering support for dozens of programming languages. The IPython shell and Jupyter notebooks are especially useful for data exploration and visualization.

For me personally, IPython is usually involved with the majority of my Python work, including running, debugging, and testing code.

In the accompanying book materials (see <http://github.com/wesm/pydata-book>), you will find Jupyter notebooks containing all the code examples from

each chapter.

# SciPy

SciPy (<http://scipy.org>) is a collection of packages addressing a number of different standard problem domains in scientific computing. Here is a sampling of the packages included:

- `scipy.integrate`: numerical integration routines and differential equation solvers
- `scipy.linalg`: linear algebra routines and matrix decompositions extending beyond those provided in `numpy.linalg`.
- `scipy.optimize`: function optimizers (minimizers) and root finding algorithms
- `scipy.signal`: signal processing tools
- `scipy.sparse`: sparse matrices and sparse linear system solvers
- `scipy.special`: wrapper around SPECFUN, a Fortran library implementing many common mathematical functions, such as the gamma function
- `scipy.stats`: standard continuous and discrete probability distributions (density functions, samplers, continuous distribution functions), various statistical tests, and more descriptive statistics
- `scipy.weave`: tool for using inline C++ code to accelerate array computations

Together NumPy and SciPy form a reasonably complete and mature computational foundation for many traditional scientific computing applications.

# scikit-learn

Since the project's inception in 2010, scikit-learn (<http://scikit-learn.org>), has become the premier general-purpose machine learning toolkit for Python programmers. In 6 years, it has had over 600 contributors from around the world. It includes submodules for such models as:

- *Classification*: SVM, nearest neighbors, random forest, etc.
- *Regression*: Lasso, Ridge regression, Logistic, etc.
- *Clustering*: k-Means, spectral clustering, etc.
- *Dimensionality reduction*: PCA, feature selection, matrix factorization, etc.
- *Model selection*: grid search, cross-validation, metrics.
- *Preprocessing*: feature extraction, normalization.

Along with pandas, statsmodels, and IPython, scikit-learn has been critical for enabling Python to be a productive data science programming language. While I won't be able to give comprehensive guide to scikit-learn in this book, I will give a brief intro to some of its models and how to use them with the other tools presented in the book.

# statsmodels

statsmodels ((<http://statsmodels.org>)) is a statistical analysis package that was seeded by work from Stanford University statistics professor Jonathan Taylor, who implemented a number of regression analysis models popular in the R programming language. Skipper Seabold and Josef Perktold formally created the new statsmodels project in 2010 and since then have grown the project to a critical mass of engaged users and contributors. Nathaniel Smith developed the Patsy project which provides a formula or model specification framework for statsmodels inspired by R's formula system.

Compared with scikit-learn, statsmodels contains algorithms for classical (primarily frequentist) statistics and econometrics. This includes such submodules as:

- Regression models: linear regression, generalized linear models, robust linear models, linear mixed effects models, etc.
- ANOVA (analysis of variance)
- Time series analysis: AR, ARMA, ARIMA, VAR, and other models.
- Nonparametric methods: kernel density estimation, kernel regression.
- Visualization of statistical model results

# Installation and Setup

Since everyone uses Python for different applications, there is no single solution for setting up Python and required add-on packages. Many readers will not have a complete Python development environment suitable for following along with this book, so here I will give detailed instructions to get set up on each operating system. I recommend using the free Anaconda distribution provided by Continuum Analytics. At the time of this writing, Anaconda is offered in both Python 2.7 and 3.6 forms, though this might change at some point in the future. This book uses Python 3.6, and I encourage you to use Python 3.6 or higher.

# Windows

To get started on Windows, download the Anaconda installer from <http://continuum.io/downloads>, which should be an executable named like `Anaconda3-4.1.0-Windows-x86_64.exe`. Run the installer and accept the default installation location, which will either be `C:\Python35` (if run as an administrator) or `C:\Users\$USERNAME\Anaconda3` if run as a normal user. If you had previously installed Python in this location, you may want to delete it manually first (or using “Add or remove programs” in the System settings).

The installer will also ask you if you wish to make the Anaconda distribution the default Python on your system and whether to add it to your PATH environment variable. I recommend you accept these options. Now, let’s verify that things are configured correctly. Open a command prompt by going to the Start Menu and starting the Command Prompt application, also known as `cmd.exe`. Try starting the Python interpreter by typing `python`. You should see a message that matches the version of Anaconda you installed:

```
C:\Users\wesm>python
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul 5 2016, :
[MSC v.1900 64 bit (AMD64)] on win32
>>>
```

# Apple (OS X, macOS)

Download the OS X Anaconda installer which should be named something like `Anaconda3-4.1.0-MacOSX-x86_64.pkg`. Double-click the `.pkg` file to run the installer. When the installer runs, it automatically appends the Anaconda executable path to your `.bash_profile` file. This is located at `/Users/$USER/.bash_profile`.

To verify everything is working, try launching IPython in the system shell (open the Terminal application to get a command prompt):

```
$ ipython
```

# GNU/Linux

## Note

Some Linux distributions have versions of all the required Python packages in their package managers and can be installed using a tool like `apt`. I detail setup using Anaconda as it's both easily reproducible across distributions and simpler to upgrade packages to their latest versions.

Linux details will vary a bit depending on your Linux flavor, but here I give details for such distributions as Debian, Ubuntu, CentOS, and Fedora. Setup is similar to OS X with the exception of how Anaconda is installed. The installer is a shell script that must be executed in the terminal. Depending on whether you have a 32-bit or 64-bit system, you will either need to install the `x86` (32-bit) or `x86_64` (64-bit) installer. You will then have a file named something similar to `Anaconda3-4.1.0-Linux-x86_64.sh`. To install it, execute this script with `bash`:

```
$ bash Anaconda3-4.1.0-Linux-x86_64.sh
```

After accepting the license, you will be presented with a choice of where to put the Anaconda files. I recommend installing the files in the default location in your home directory, for example `/home/$USER/anaconda` (with your username, naturally).

The Anaconda installer may ask if you wish to prepend its `bin/` directory to your `$PATH` variable. If you have any problems after installation you can do this yourself by modifying your `.bashrc` with something akin to:

```
export PATH=/home/$USER/anaconda/bin:$PATH
```

After doing this you can either start a new terminal process or execute your `.bashrc` again with `source ~/.bashrc`.

If you do not wish for Anaconda to conflict with the system-level Python environment on your system, you may consider adding a bash function to your

profile that does something like:

```
function anaconda {  
    export PATH=/home/wesm/anaconda/bin:$PATH  
}
```

This way, when you open a terminal shell, you can “activate” the environment by executing:

```
$ anaconda
```

# Installing or updating Python packages

At some point while reading, you may wish to install additional Python packages which are not included in the Anaconda distribution. In general, these can be installed by the command

```
conda install package_name
```

If this does not work, you may also be able to install the package using `pip` package management tool:

```
pip install package_name
```

# Python 2 and Python 3

The first version of the Python 3.x line of interpreters was released at the end of 2008. It included a number of changes that made some previously-written Python 2.x code incompatible. Since 17 years had passed since the very first release of Python in 1991, creating a “breaking” release of Python 3 was viewed to be for the greater good given the lessons learned over the preceding nearly 20 years.

In 2012, much of the scientific and data analysis community was still using Python 2.x because many packages had not been made fully Python 3 compatible. Thus, the first edition of this book used Python 2.7. Now, users are free to choose between Python 2.x and 3.x and, in general have full library support with either flavor.

However, Python 2.x will reach its development end of life in 2020 (including critical security patches), and so it is no longer a good idea to start new projects in Python 2.7. As such, this book uses Python 3.6, a widely deployed, well-supported stable release. We have begun to call Python 2.x “Legacy Python” and Python 3.x simply “Python”. I encourage you to do the same.

This book uses Python 3.6 as its basis. Your version of Python may be newer than 3.6, but the code examples should be forward compatible.

# Integrated Development Environments (IDEs) and Text Editors

When asked about my standard development environment, I almost always say “IPython plus a text editor”. I typically write a program and iteratively test and debug each piece of it in IPython or Jupyter notebooks. It is also useful to be able to play around with data interactively and visually verify that a particular set of data manipulations are doing the right thing. Libraries like pandas and NumPy are designed to be easy-to-use in the shell.

When building software, however, some users may prefer to use a more richly-featured IDE rather than a comparatively primitive text editor like Emacs or Vim. Here are some that you can explore:

- PyDev (free), an IDE built on the Eclipse platform
- PyCharm from JetBrains (paid for commercial users, free for open source developers).
- Python Tools for Visual Studio (for Windows users)
- Spyder (free), an IDE currently shipped with Anaconda.
- Komodo IDE (commercial)

Due to the popularity of Python, most text editors, like Atom and Sublime Text 2, have excellent Python support.

# Community and Conferences

Outside of an Internet search, the various scientific and data-related Python mailing lists are generally helpful and responsive to questions. Some ones to take a look at are:

- pydata: a Google Group list for questions related to Python for data analysis and pandas
- pystatsmodels: for statsmodels or pandas-related questions
- Mailing list for scikit-learn and machine learning in Python, generally.
- numpy-discussion: for NumPy-related questions
- scipy-user: for general SciPy or scientific Python questions

I deliberately did not post URLs for these in case they change. They can be easily located via Internet search.

Each year many conferences are held all over the world for Python programmers. If you would like to connect with other Python programmers who share your interests, I encourage you to explore attending one, if possible. Many conferences have financial support available for those who cannot afford admission or travel to the conference.

- PyCon and EuroPython, the two main general Python conferences in North America and Europe, respectively.
- SciPy and EuroSciPy: scientific-computing oriented conferences in North America and Europe, respectively.
- PyData: a worldwide series of regional conferences targeted at data science and data analysis use cases.
- International and regional PyCon conferences: see <http://pycon.org> for a

complete listing.

# Navigating This Book

If you have never programmed in Python before, you will want to spend some time in the first two chapters of the book, where I have placed a condensed tutorial on Python language features and the IPython shell and Jupyter notebooks. These things are prerequisite knowledge for the remainder of the book. If you have Python experience already, you may instead choose to skim or skip these chapters.

Next, I give a short introduction to the key features of NumPy, leaving more advanced NumPy use for another chapter at the end of the book. Then, I introduce pandas and devote the rest of the book to data analysis topics applying pandas, NumPy, and matplotlib (for visualization). I have structured the material in the most incremental way possible, though there is occasionally some minor cross-over between chapters, with occasional concepts used which haven't necessarily been introduced yet.

While readers may have many different end goals for their work, the tasks required generally fall into a number of different broad groups:

Interacting with the outside world

Reading and writing with a variety of file formats and data stores.

Preparation

Cleaning, munging, combining, normalizing, reshaping, slicing and dicing, and transforming data for analysis.

Transformation

Applying mathematical and statistical operations to groups of data sets to derive new data sets. For example, aggregating a large table by group variables.

Modeling and computation

Connecting your data to statistical models, machine learning algorithms, or other computational tools

Presentation

Creating interactive or static graphical visualizations or textual summaries

# Code Examples

Most of the code examples in the book are shown with input and output as it would appear executed in the IPython shell or in Jupyter notebooks.

```
In [5]: code  
Out[5]: output
```

# Data for Examples

Data sets for the examples in each chapter are hosted in a repository on GitHub: [`http://github.com/wesm/pydata-book`](http://github.com/wesm/pydata-book). You can download this data either by using the git version control system on the command line or by downloading a zip file of the repository from the website.

I have made every effort to ensure that it contains everything necessary to reproduce the examples, but I may have made some mistakes or omissions. If so, please send me an e-mail: `wesmckinn@gmail.com`. The best way to report errors in the book is on the Errata page on O'Reilly's website (linked to from the book's main page).

# Import Conventions

The Python community has adopted a number of naming conventions for commonly-used modules:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import sklearn as sk
import statsmodels as sm
```

This means that when you see `np.arange`, this is a reference to the `arange` function in NumPy. This is done as it's considered bad practice in Python software development to import everything (`from numpy import *`) from a large package like NumPy.

# Jargon

I'll use some terms common both to programming and data science that you may not be familiar with. Thus, here are some brief definitions:

## Munge/Munging/Wrangling

Describes the overall process of manipulating unstructured and/or messy data into a structured or clean form. The word has snuck its way into the jargon of many modern day data hackers. Munge rhymes with "grunge".

## Pseudocode

A description of an algorithm or process that takes a code-like form while likely not being actual valid source code.

## Syntactic sugar

Programming syntax which does not add new features, but makes something more convenient or easier to type.

# Chapter 2. Python Language Basics, IPython, and Jupyter Notebooks

Knowledge is a treasure, but practice is the key to it.

Thomas Fuller

When I wrote the 1st edition of this book in 2011 and 2012, there were fewer resources available for learning about doing data analysis in Python. This was partially a chicken-and-egg problem; many libraries that we now take for granted, like pandas, scikit-learn, and statsmodels, were comparatively immature back then. In 2016, there is now a growing literature on data science, data analysis, and machine learning, supplementing the prior works on general purpose scientific computing oriented at computational scientists, physicists, and other research fields. There are also many excellent books about learning the Python programming language itself and becoming an effective software engineer.

As this book is intended as an introductory text in working with data in Python, I feel it is valuable to have a self-contained overview of some of the most important features of Python's built-in data structures and libraries from the perspective of data manipulation. As such, I will only present roughly enough information in the next two chapters to enable you to follow along with the rest of the book.

In my opinion, it is *not* necessary to become proficient at building good software in Python to be able to productively do data analysis. I encourage you to use the IPython shell and Jupyter notebooks to experiment with the code examples and to explore the documentation for the various types, functions, and methods. While I've made best efforts to present the book material in an incremental form, you may occasionally encounter things that have not yet been fully introduced.

Much of this book focuses on table-based analytics and data preparation tools

for working with large data sets. In order to use those tools you must often first do some munging to corral messy data into a more nicely tabular (or *structured*) form. Fortunately, Python is an ideal language for rapidly whipping your data into shape. The greater your facility with Python the language, the easier it will be for you to prepare new data sets for analysis.

Some of the tools in this book are best explored from a live IPython or Jupyter session. Once you learn how to start up IPython and Jupyter, I recommend that you follow along with the examples so you can experiment and try different things. As with any keyboard-driven console-like environment, developing muscle-memory for the common commands is also part of the learning curve.

#### Note

There are many introductory Python that this chapter does not cover, like classes and object oriented programming, which you may find useful in your foray into data analysis in Python.

To deepen your Python language knowledge, I recommend that you supplement this chapter with the official Python tutorial (<http://docs.python.org>) and potentially one of the many excellent books on general purpose Python programming. Some recommendations to get you started include:

- *Python Cookbook, 3rd Edition*, by David Beazley and Brian K. Jones.
- *Fluent Python*, by Luciano Ramalho.
- *Effective Python*, by Brett Slatkin.

# The Python Interpreter

Python is an *interpreted* language. The Python interpreter runs a program by executing one statement at a time. The standard interactive Python interpreter can be invoked on the command line with the `python` command:

```
$ python
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017,
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more info
>>> a = 5
>>> print(a)
5
```

The `>>>` you see is the *prompt* where you'll type expressions. To exit the Python interpreter and return to the command prompt, you can either type `exit()` or press `Ctrl-D`.

Running Python programs is as simple as calling `python` with a `.py` file as its first argument. Suppose we had created `hello_world.py` with these contents:

```
print('Hello world')
```

This can be run from the terminal simply as:

```
$ python hello_world.py
Hello world
```

While some Python programmers execute all of their Python code in this way, many Python programmers doing data analysis or scientific computing make use of IPython, an enhanced Python interpreter, or Jupyter notebooks, a web-based code notebook originally created within the IPython project. I give an introduction to using IPython and Jupyter in this chapter and have included a deeper look at IPython functionality at the end of the book in [Chapter 14](#). By using the `%run` command, IPython executes the code in the specified file in the same process, enabling you to explore the results interactively when it's done.

```
$ ipython
```

```
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017,
Type "copyright", "credits" or "license" for more information.
```

```
IPython 5.1.0 -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra c
```

```
In [1]: %run hello_world.py
Hello world
```

```
In [2]:
```

The default IPython prompt adopts the numbered In [2]: style compared with the standard >>> prompt.

# **IPython Basics**

In this section, we'll get you up and running with the IPython shell and Jupyter notebook, and introduce you to some of the essential concepts.

# Running the IPython Shell

You can launch the IPython shell on the command line just like launching the regular Python interpreter except with the `ipython` command:

```
$ ipython
Python 3.6.0 | packaged by conda-forge | (default, Jan 13 2017,
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?            -> Introduction and overview of IPython's features.
%quickref  -> Quick reference.
help        -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra c

In [1]: a = 5

In [2]: a
Out[2]: 5
```

You can execute arbitrary Python statements by typing them in and pressing `<return>`. When typing just a variable into IPython, it renders a string representation of the object:

```
In [541]: import numpy as np

In [542]: data = {i : np.random.randn() for i in range(7)}

In [543]: data
Out[543]:
{0: 0.6900018528091594,
 1: 1.0015434424937888,
 2: -0.5030873913603446,
 3: -0.6222742250596455,
 4: -0.9211686080130108,
 5: -0.726213492660829,
 6: 0.2228955458351768}
```

Many kinds of Python objects are formatted to be more readable, or *pretty-printed*, which is distinct from normal printing with `print`. If you printed a dict like the above in the standard Python interpreter, it would be much less

readable:

```
>>> from numpy.random import randn
>>> data = {i : randn() for i in range(7)}
>>> print data
{0: -1.5948255432744511, 1: 0.10569006472787983, 2: 1.972367135
3: 0.15455217573074576, 4: -0.24058577449429575, 5: -1.2904897
6: 0.3308507317325902}
```

IPython also provides facilities to execute arbitrary blocks of code (via somewhat glorified copy-and-pasting) and whole Python scripts. You can also use the Jupyter notebook to work with larger blocks of code as we'll see later.

# Running the Jupyter Notebook

One of the major components of the Jupyter project is the *notebook*, a type of interactive document for code, text with or without markup, data visualizations, and other output. The Jupyter notebook interacts with *kernels*, which are implementations of the Jupyter interactive computing protocol in any number of programming languages. Python's Jupyter kernel uses the IPython system for its underlying behavior.

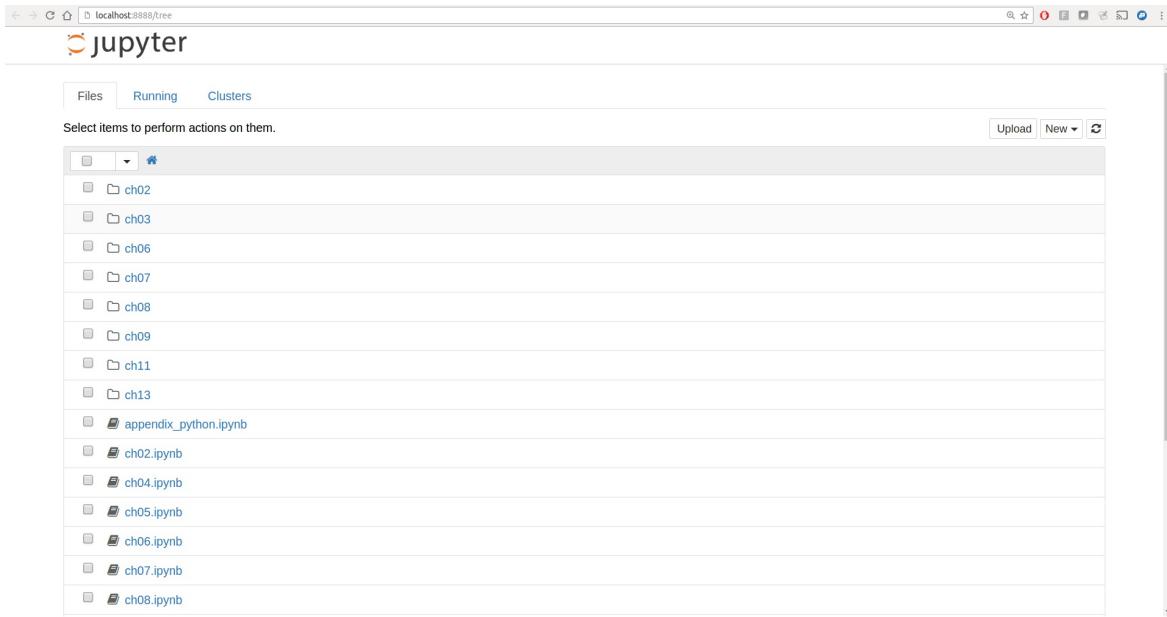
To start up Jupyter, run the command `jupyter notebook` in a terminal:

```
$ jupyter notebook
[I 15:20:52.739 NotebookApp] Serving notebooks from local directory:
/home/wesm/code/pydata-book
[I 15:20:52.739 NotebookApp] 0 active kernels
[I 15:20:52.739 NotebookApp] The Jupyter Notebook is running at
[I 15:20:52.740 NotebookApp] Use Control-C to stop this server
all kernels (twice to skip confirmation).
Created new window in existing browser session.
```

On many platforms, Jupyter will automatically open up in your default web browser (unless you start it with `--no-browser`). Otherwise, you can navigate to the HTTP address printed when you started the notebook, here `http://localhost:8888/`. See [Figure 2-1](#) for what this looks like for me in Google Chrome.

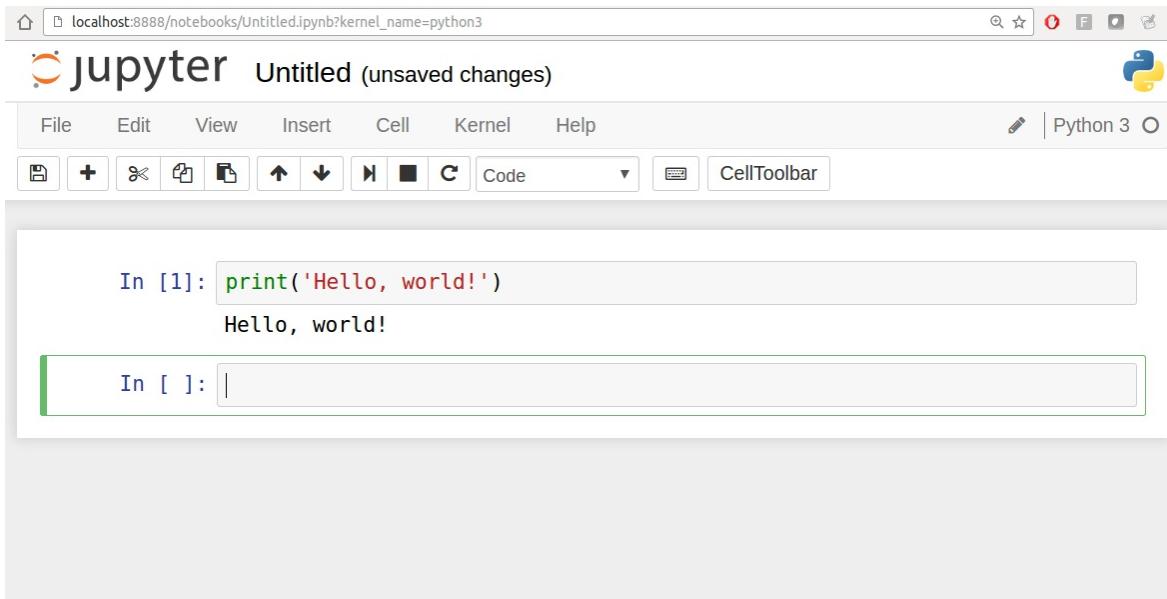
## Note

Many people use Jupyter as a local computing environment, but it can also be deployed on servers and accessed remotely. I won't cover those details here, but encourage you to explore this topic on the Internet if it's relevant to your needs.



**Figure 2-1. Jupyter Notebook Landing Page**

To create a new notebook, click the `New` button and select the `Python 3` option. You should see something like [Figure 2-2](#). If this is your first time, try clicking on the empty code “cell” and entering a line of Python code. Then press `Shift-Enter` to execute it.



**Figure 2-2. Jupyter example view for an existing notebook**

When you save the notebook (see `Save` and `Checkpoint` under the notebook `File` menu), it creates a file with the extension `.ipynb`. This is a self-contained file format that contains all of the content (including any evaluated code output) currently in the notebook. These can be loaded and edited by other Jupyter users. To load an existing notebook, put the file in the same directory (or a folder inside) where you started the notebook process, then double-click the name from the landing page. You can try it out with the notebooks from my `wesm/pydata-book` repository on GitHub. See [Figure 2-3](#).

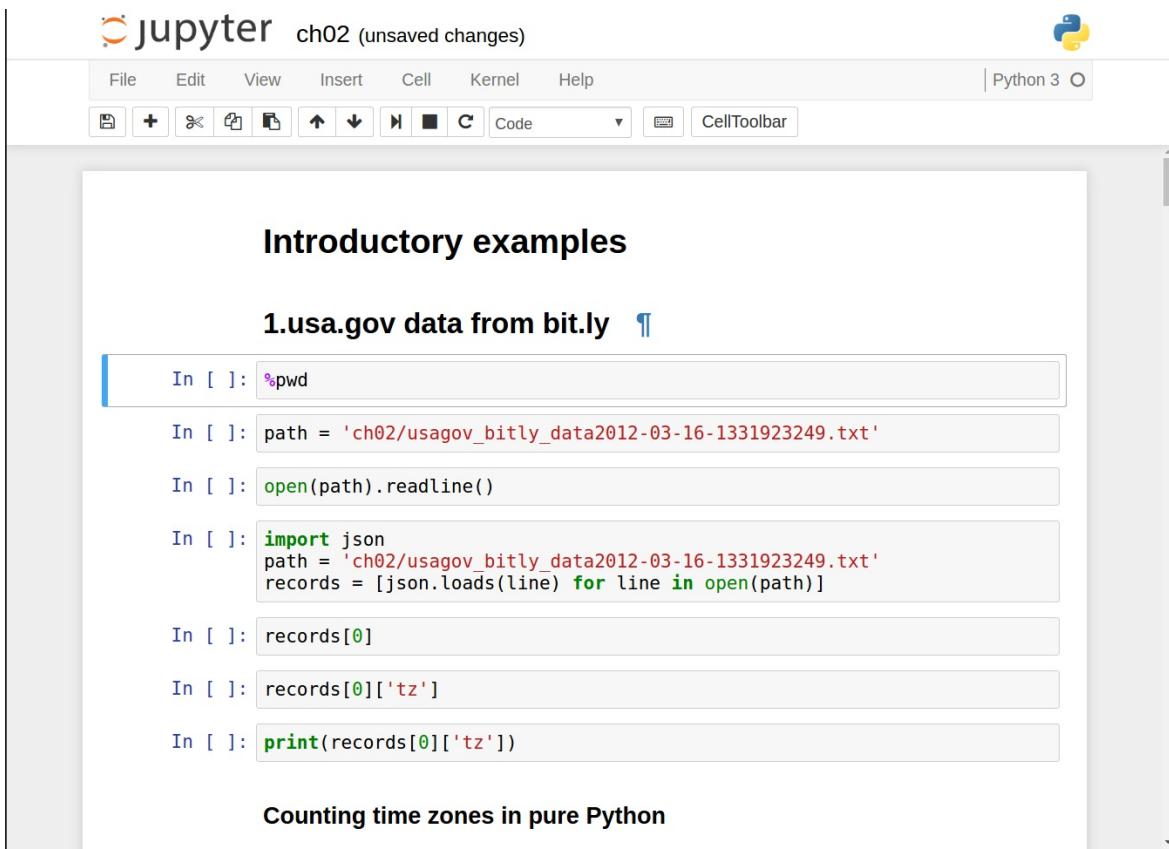


Figure 2-3. Jupyter new notebook view

While the Jupyter notebook can feel like a distinct experience from the IPython shell, nearly all of the commands and tools in this chapter can be used in either environment.

# Tab Completion

On the surface, the IPython shell looks like a cosmetically different version of the standard terminal Python interpreter (invoked with `python`). One of the major improvements over the standard Python shell is *tab completion*, found in many IDEs or other interactive computing analysis environments. While entering expressions in the shell, pressing `<Tab>` will search the namespace for any variables (objects, functions, etc.) matching the characters you have typed so far:

```
In [1]: an_apple = 27  
In [2]: an_example = 42  
In [3]: an<Tab>  
an_apple      and          an_example  any
```

In this example, note that IPython displayed both the two variables I defined as well as the Python keyword `and` and built-in function `any`. Naturally, you can also complete methods and attributes on any object after typing a period:

```
In [3]: b = [1, 2, 3]  
In [4]: b.<Tab>  
b.append  b.count   b.insert  b.reverse  
b.clear    b.extend  b.pop     b.sort  
b.copy     b.index   b.remove
```

The same goes for modules:

```
In [1]: import datetime  
In [2]: datetime.<Tab>  
datetime.date           datetime.MAXYEAR           datetime.timedelta  
datetime.datetime        datetime.MINYEAR           datetime.timezone  
datetime.datetime_CAPI  datetime.time            datetime.tzinfo
```

In the Jupyter notebook and newer versions of IPython (5.0 and higher), the autocompletions show up in a dropdown box rather than as text output.

### Note

Note that IPython by default hides methods and attributes starting with underscores, such as magic methods and internal “private” methods and attributes, in order to avoid cluttering the display (and confusing novice users!). These, too, can be tab-completed but you must first type an underscore to see them. If you prefer to always see such methods in tab completion, you can change this setting in the IPython configuration.

Tab completion works in many contexts outside of searching the interactive namespace and completing object or module attributes. When typing anything that looks like a file path (even in a Python string), pressing <Tab> will complete anything on your computer’s file system matching what you’ve typed:

```
In [7]: datasets/movielens/<Tab>
datasets/movielens/movies.dat      datasets/movielens/README
datasets/movielens/ratings.dat     datasets/movielens/users.dat
```

```
In [3]: path = 'datasets/movielens/<Tab>
datasets/movielens/movies.dat      datasets/movielens/README
datasets/movielens/ratings.dat     datasets/movielens/users.dat
```

Combined with the %run command (see later section), this functionality can save you many keystrokes.

Another area where tab completion saves time is in the completion of function keyword arguments (and including the = sign!). See [Figure 2-4](#).

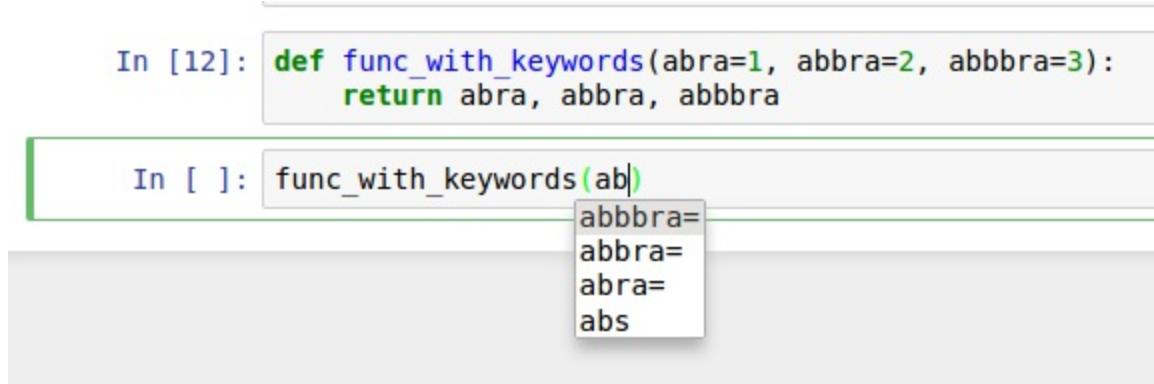


Figure 2-4. Autocomplete function keywords in Jupyter notebook

# Introspection

Using a question mark (?) before or after a variable will display some general information about the object:

```
In [545]: b?
Type:          list
String Form: [1, 2, 3]
Length:        3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

This is referred to as *object introspection*. If the object is a function or instance method, the docstring, if defined, will also be shown. Suppose we'd written the following function:

```
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    -----
    the_sum : type of arguments
    """
    return a + b
```

Then using ? shows us the docstring:

```
In [547]: add_numbers?
Signature: add_numbers(a, b)
Docstring:
Add two numbers together

Returns
-----
the_sum : type of arguments
File:      <ipython-input-9-6a548a216e27>
Type:      function
```

Using ?? will also show the function's source code if possible:

```
In [548]: add_numbers??  
Signature: add_numbers(a, b)  
Source:  
def add_numbers(a, b):  
    """  
        Add two numbers together  
  
    Returns  
    -----  
    the_sum : type of arguments  
    """  
    return a + b  
File:      <ipython-input-9-6a548a216e27>  
Type:      function
```

? has a final usage, which is for searching the IPython namespace in a manner similar to the standard UNIX or Windows command line. A number of characters combined with the wildcard (\*) will show all names matching the wildcard expression. For example, we could get a list of all functions in the top level NumPy namespace containing load:

```
In [549]: np.*load*?  
np.__loader__  
np.load  
np.loads  
np.loadtxt  
np.pkgload
```

# The %run Command

Any file can be run as a Python program inside the environment of your IPython session using the `%run` command. Suppose you had the following simple script stored in `ipython_script_test.py`:

```
def f(x, y, z):
    return (x + y) / z

a = 5
b = 6
c = 7.5

result = f(a, b, c)
```

This can be executed by passing the file name to `%run`:

```
In [550]: %run ipython_script_test.py
```

The script is run in an *empty namespace* (with no imports or other variables defined) so that the behavior should be identical to running the program on the command line using `python script.py`. All of the variables (imports, functions, and globals) defined in the file (up until an exception, if any, is raised) will then be accessible in the IPython shell:

```
In [551]: c
Out[551]: 7.5
```

```
In [552]: result
Out[552]: 1.4666666666666666
```

If a Python script expects command line arguments (to be found in `sys.argv`), these can be passed after the file path as though run on the command line.

## Note

Should you wish to give a script access to variables already defined in the interactive IPython namespace, use `%run -i` instead of plain `%run`.

In the Jupyter notebook, you may also use the related `%load` magic function which imports a script into a code cell:

```
# %load ipython_script_test.py
def f(x, y, z):
    return (x + y) / z

a = 5
b = 6
c = 7.5

result = f(a, b, c)
```

## Interrupting running code

Pressing `<Ctrl-C>` while any code is running, whether a script through `%run` or a long-running command, will cause a `KeyboardInterrupt` to be raised. This will cause nearly all Python programs to stop immediately except in certain exceptional cases.

### Warning

When a piece of Python code has called into some compiled extension modules, pressing `<Ctrl-C>` will not cause the program execution to stop immediately in all cases. In such cases, you will have to either wait until control is returned to the Python interpreter, or, in more dire circumstances, forcibly terminate the Python process.

# Executing Code from the Clipboard

If you are using the Jupyter notebook, you can copy and paste code into any code cell and execute it. It is also possible to run code from the clipboard in the IPython shell. Suppose you had the following code in some other application:

```
x = 5
y = 7
if x > 5:
    x += 1

y = 8
```

The most foolproof methods are the `%paste` and `%cpaste` magic functions. `%paste` takes whatever text is in the clipboard and executes it as a single block in the shell:

```
In [6]: %paste
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
## -- End pasted text --
```

`%cpaste` is similar, except that it gives you a special prompt for pasting code into:

```
In [7]: %cpaste
Pasting code; enter '---' alone on the line to stop or use Ctrl-D.
:x = 5
:y = 7
:if x > 5:
:    x += 1
:
:    y = 8
:--
```

With the `%cpaste` block, you have the freedom to paste as much code as you

like before executing it. You might decide to use `%cpaste` in order to look at the pasted code before executing it. If you accidentally paste the wrong code, you can break out of the `%cpaste` prompt by pressing `<Ctrl-C>`.

# Terminal Keyboard Shortcuts

IPython has many keyboard shortcuts for navigating the prompt (which will be familiar to users of the Emacs text editor or the UNIX bash shell) and interacting with the shell's command history (see later section). [Table 2-1](#) summarizes some of the most commonly used shortcuts. See [Figure 2-5](#) for an illustration of a few of these, such as cursor movement.

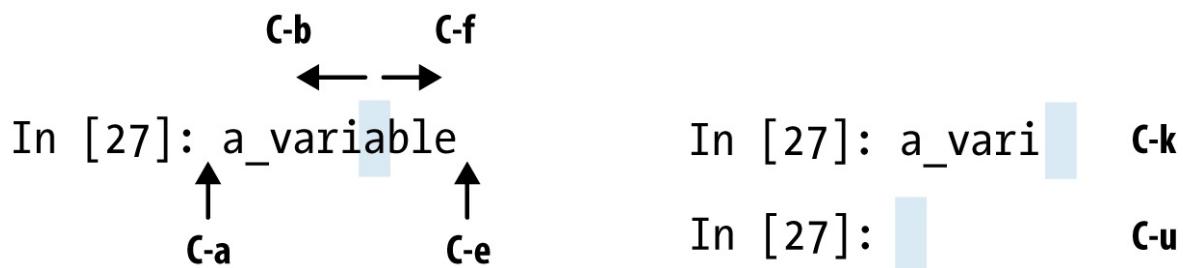


Figure 2-5. Illustration of some keyboard shortcuts in the IPython shell

Table 2-1. Standard IPython Keyboard Shortcuts

| Command              | Description  |
|----------------------|--|
| Ctrl-p or up-arrow   | Search backward in command history for commands starting with currently-entered text |
| Ctrl-n or down-arrow | Search forward in command history for commands starting with currently-entered text  |
| Ctrl-r               | Readline-style reverse history search (partial matching)                             |
| Ctrl-Shift-v         | Paste text from clipboard  |
| Ctrl-c               | Interrupt currently-executing code   |
| Ctrl-a               | Move cursor to beginning of line   |
| Ctrl-e               | Move cursor to end of line   |
| Ctrl-k               | Delete text from cursor until end of line  |
| Ctrl-u               | Discard all text on current line   |
| Ctrl-f               | Move cursor forward one character  |
| Ctrl-b               | Move cursor back one character   |

`Ctrl-l`

Clear screen

Note that Jupyter notebooks have a largely separate set of keyboard shortcuts for navigation and editing.

# Exceptions and Tracebacks

If an exception is raised while %run-ing a script or executing any statement, IPython will by default print a full call stack trace (traceback) with a few lines of context around the position at each point in the stack.

```
In [553]: %run examples/ipython_bug.py
-----
AssertionError                                     Traceback (most recent call last)
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>
      13     throws_an_exception()
      14
-> 15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in calling_things()
      11 def calling_things():
      12     works_fine()
-> 13     throws_an_exception()
      14
      15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in throws_an_exception()
      7     a = 5
      8     b = 6
->  9     assert(a + b == 10)
     10
     11 def calling_things():

AssertionError:
```

Having additional context by itself is a big advantage over the standard Python interpreter (which does not provide any additional context). The amount of context shown can be controlled using the %xmode magic command, from plain (same as the standard Python interpreter) to verbose (which inlines function argument values and more). As you will see later in the chapter, you can step *into the stack* (using the %debug or %pdb magics) after an error has occurred for interactive post-mortem debugging.

# About Magic Commands

IPython's special commands (which are not built into Python itself) are known as "magic" commands. These are designed to facilitate common tasks and enable you to easily control the behavior of the IPython system. A magic command is any command prefixed by the percent symbol %. For example, you can check the execution time of any Python statement, such as a matrix multiplication, using the %timeit magic function (which will be discussed in more detail later):

```
In [554]: a = np.random.randn(100, 100)

In [555]: %timeit np.dot(a, a)
10000 loops, best of 3: 20.9 µs per loop
```

Magic commands can be viewed as command line programs to be run within the IPython system. Many of them have additional "command line" options, which can all be viewed (as you might expect) using ?:

```
In [40]: %debug?
Docstring:
::

%debug [--breakpoint FILE:LINE] [statement [statement ...]]
```

Activate the interactive debugger.

This magic command support two ways of activating debugger. One is to activate debugger before executing code. This way, I can set a break point, to step through the code from the point. You can use this mode by giving statements to execute and option a breakpoint.

The other one is to activate debugger in post-mortem mode. You activate this mode simply running %debug without any argument. If an exception has just occurred, this lets you inspect its stack frames interactively. Note that this will always work only on traceback that occurred, so you must call this quickly after an exception that you wish to inspect has fired, because if another occurs, it clobbers the previous one.

If you want IPython to automatically do this on every exception the %pdb magic for more details.

positional arguments:

|           |   |
|-----------|---|
| statement | Code to run in debugger. You can omit the magic mode. |
|-----------|---|

optional arguments:

|  |                                  |
|--|----------------------------------|
| --breakpoint <FILE:LINE>, -b <FILE:LINE> | Set break point at LINE in FILE. |
|--|----------------------------------|

Magic functions can be used by default without the percent sign, as long as no variable is defined with the same name as the magic function in question. This feature is called *automagic* and can be enabled or disabled using %automagic.

Some magic functions behave like Python functions and their output can be assigned to a variable:

```
In [1]: %pwd  
Out[1]: '/home/wesm/code/pydata-book'
```

```
In [2]: foo = %pwd
```

```
In [3]: foo  
Out[3]: '/home/wesm/code/pydata-book'
```

Since IPython's documentation is accessible from within the system, I encourage you to explore all of the special commands available by typing %quickref or %magic. I will highlight a few more of the most critical ones for being productive in interactive computing and Python development in IPython.

Table 2-2. Some Frequently-used IPython Magic Commands

| Command   | Description  |
|-----------|--|
| %quickref | Display the IPython Quick Reference Card                                     |
| %magic    | Display detailed documentation for all of the available magic commands       |
| %debug    | Enter the interactive debugger at the bottom of the last exception traceback |
| %hist     | Print command input (and optionally output) history                          |
| %pdb      | Automatically enter debugger after any exception                             |

|                             |   |
|-----------------------------|---|
| %paste                      | Execute pre-formatted Python code from clipboard  |
| %cpaste                     | Open a special prompt for manually pasting Python code to be executed   |
| %reset                      | Delete all variables / names defined in interactive namespace   |
| %page<br><i>OBJECT</i>      | Pretty print the object and display it through a pager  |
| %run<br><i>script.py</i>    | Run a Python script inside IPython  |
| %prun<br><i>statement</i>   | Execute <i>statement</i> with <code>cProfile</code> and report the profiler output  |
| %time<br><i>statement</i>   | Report the execution time of single statement   |
| %timeit<br><i>statement</i> | Run a statement multiple times to compute an ensemble average execution time. Useful for timing code with very short execution time |
| %who,<br>%who_ls,<br>%whos  | Display variables defined in interactive namespace, with varying levels of information / verbosity                                  |
| %xdel<br><i>variable</i>    | Delete a variable and attempt to clear any references to the object in the IPython internals  |

## Jupyter Magic Commands

There are a number of magic commands, often loaded by some third party extension, intended for use in the Jupyter Notebook. These are prefixed by two percentage signs `%%` and frequently modify the behavior of the whole code cell. As an example, the Cython project (<http://cython.org>), a popular tool for creating faster compiled extension code, can be used in Jupyter without having to set up a Cython extension by hand:

```
%%cython

cimport numpy as np
np.import_array()

def sum_arr(np.ndarray[np.float64] arr):
    cdef int i
    cdef double total = 0
    for i in range(len(arr)):
```

```
total += arr[i]

return total
```

After you execute, the new function `sum_arr` is now available in the notebook:

```
In [2]: arr = np.random.randn(1000000)
```

```
In [3]: sum(arr)
Out[3]: 340.78566319852672
```

# Matplotlib Integration

One reason for IPython's popularity in analytical computing is that it integrates well with data visualization and other user interface libraries like matplotlib. Don't worry if you have never used matplotlib before; it will be discussed in more detail later in this book. The `%matplotlib` magic function configures its integration with the IPython shell or Jupyter notebook. This is important as otherwise plots you create will either not appear (notebook) or take control of the session until closed (shell).

In the IPython shell, running `%matplotlib` sets up the integration so you can create multiple plot windows without interfering with the console session:

```
In [4]: %matplotlib  
Using matplotlib backend: Qt4Agg
```

In Jupyter, the command is a little different:

```
In [4]: %matplotlib inline
```

See [Figure 2-6](#)

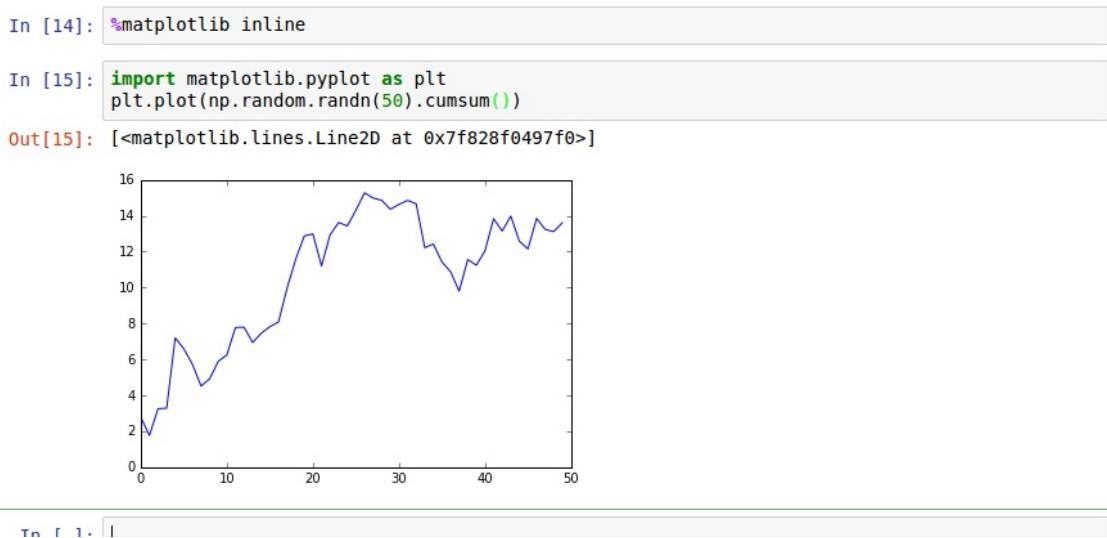


Figure 2-6. Jupyter inline matplotlib plotting

# Python Language Basics

In this section, I will give you an overview of essential Python programming concepts and language mechanics. In the next chapter, I will go into more detail about Python's data structures, functions, and some other built-in tools.

# Language Semantics

The Python language design is distinguished by its emphasis on readability, simplicity, and explicitness. Some people go so far as to liken it to “executable pseudocode”.

## Indentation, not braces

Python uses whitespace (tabs or spaces) to structure code instead of using braces as in many other languages like R, C++, Java, and Perl. Take the for loop in the above quicksort algorithm:

```
for x in array:  
    if x < pivot:  
        less.append(x)  
    else:  
        greater.append(x)
```

A colon denotes the start of an indented code block after which all of the code must be indented by the same amount until the end of the block. In another language, you might instead have something like:

```
for x in array {  
    if x < pivot {  
        less.append(x)  
    } else {  
        greater.append(x)  
    }  
}
```

While occasionally a divisive topic among programmers, one major reason that whitespace matters is that it results in most Python code looking cosmetically similar. This can mean less cognitive dissonance when you read a piece of code that you didn’t write yourself (or wrote in a hurry a year ago!). In a language without significant whitespace, you might stumble on some differently formatted code like:

```
for x in array
```

```
{  
    if x < pivot  
    {  
        less.append(x)  
    }  
    else  
    {  
        greater.append(x)  
    }  
}
```

Love it or hate it, significant whitespace is a fact of life for Python programmers, and in my experience it helps make Python code a lot more readable than other languages I've used. While it may seem foreign at first, I suspect that it will grow on you after a while.

#### Note

I strongly recommend that you use *4 spaces* to as your default indentation and that your editor replace tabs with 4 spaces. Many text editors have a setting that will replace tab stops with spaces automatically (do this!). Some people use tabs or a different number of spaces, with 2 spaces not being terribly uncommon. 4 spaces is by and large the standard adopted by the vast majority of Python programmers, so I recommend doing that in the absence of a compelling reason otherwise.

As you can see by now, Python statements also do not need to be terminated by semicolons. Semicolons can be used, however, to separate multiple statements on a single line:

```
a = 5; b = 6; c = 7
```

Putting multiple statements on one line is generally discouraged in Python as it often makes code less readable.

## Everything is an object

An important characteristic of the Python language is the consistency of its *object model*. Every number, string, data structure, function, class, module, and

so on exists in the Python interpreter in its own “box” which is referred to as a *Python object*. Each object has an associated *type* (for example, *string* or *function*) and internal data. In practice this makes the language very flexible, as even functions can be treated like any other object.

## Comments

Any text preceded by the hash mark (pound sign) # is ignored by the Python interpreter. This is often used to add comments to code. At times you may also want to exclude certain blocks of code without deleting them. An easy solution is to *comment out* the code:

```
results = []
for line in file_handle:
    # keep the empty lines for now
    # if len(line) == 0:
    #     continue
    results.append(line.replace('foo', 'bar'))
```

Comments can also occur *after* a line of executed code. While some programmers prefer comments to be placed in the line preceding a particular line of code, this can be useful at times.

```
print("Reached this line") # Simple status report
```

## Function and object method calls

Functions are called using parentheses and passing zero or more arguments, optionally assigning the returned value to a variable:

```
result = f(x, y, z)
g()
```

Almost every object in Python has attached functions, known as *methods*, that have access to the object’s internal contents. They can be called using the syntax:

```
obj.some_method(x, y, z)
```

Functions can take both *positional* and *keyword* arguments:

```
result = f(a, b, c, d=5, e='foo')
```

More on this later.

## Variables and pass-by-reference

When assigning a variable (or *name*) in Python, you are creating a *reference* to the object on the right hand side of the equals sign. In practical terms, consider a list of integers:

```
In [5]: a = [1, 2, 3]
```

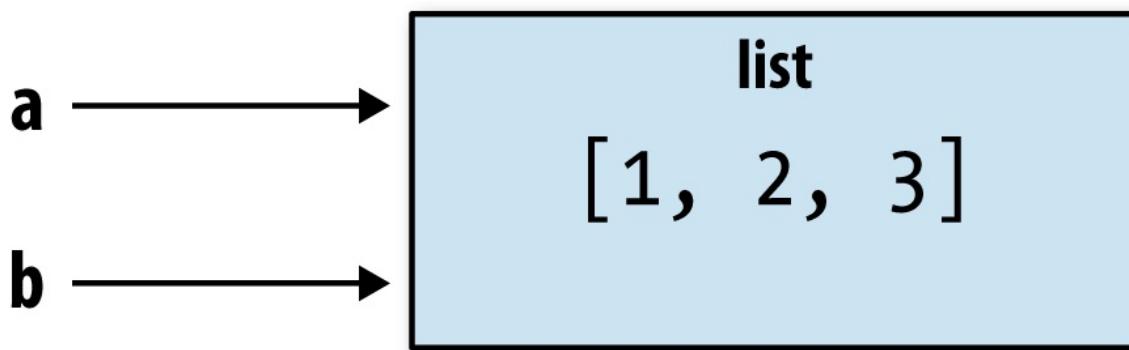
Suppose we assign `a` to a new variable `b`:

```
In [6]: b = a
```

In some languages, this assignment would cause the data `[1, 2, 3]` to be copied. In Python, `a` and `b` actually now refer to the same object, the original list `[1, 2, 3]` (see [Figure 2-7](#) for a mockup). You can prove this to yourself by appending an element to `a` and then examining `b`:

```
In [7]: a.append(4)
```

```
In [8]: b  
Out[8]: [1, 2, 3, 4]
```



**Figure 2-7.** Two references for the same object

Understanding the semantics of references in Python and when, how, and why data is copied is especially critical when working with larger data sets in Python.

#### Note

Assignment is also referred to as *binding*, as we are binding a name to an object. Variable names that have been assigned may occasionally be referred to as bound variables.

When you pass objects as arguments to a function, you are only passing references; no copying occurs. Thus, Python is said to always *pass-by-reference*, whereas some other languages (like C++) support various argument passing conventions, including as pass-by-value (creating copies) and pass-by-reference. This means that a function can alter the internals of its arguments. Suppose we had the following function:

```
def append_element(some_list, element):
    some_list.append(element)
```

Then given what's been said, this should not come as a surprise:

```
In [2]: data = [1, 2, 3]
In [3]: append_element(data, 4)
In [4]: data
Out[4]: [1, 2, 3, 4]
```

## Dynamic references, strong types

In contrast with many compiled languages, such as Java and C++, object *references* in Python have no type associated with them. There is no problem with the following:

```
In [9]: a = 5
In [10]: type(a)
Out[10]: int
```

```
In [11]: a = 'foo'
```

```
In [12]: type(a)
```

```
Out[12]: str
```

Variables are names for objects within a particular namespace; the type information is stored in the object itself. Some observers might hastily conclude that Python is not a “typed language”. This is not true; consider this example:

```
In [13]: '5' + 5
```

```
-----  
TypeError                                         Traceback (most recent call last)  
<ipython-input-13-f9dbf5f0b234> in <module>()  
----> 1 '5' + 5  
TypeError: must be str, not int
```

In some languages, such as Visual Basic, the string '5' might get implicitly converted (or *casted*) to an integer, thus yielding 10. Yet in other languages, such as JavaScript, the integer 5 might be casted to a string, yielding the concatenated string '55'. In this regard Python is considered a *strongly-typed* language, which means that every object has a specific type (or *class*), and implicit conversions will occur only in certain obvious circumstances, such as the following:

```
In [14]: a = 4.5
```

```
In [15]: b = 2
```

```
# String formatting, to be visited later  
In [16]: print('a is {0}, b is {1}'.format(type(a), type(b)))  
a is <class 'float'>, b is <class 'int'>
```

```
In [17]: a / b
```

```
Out[17]: 2.25
```

Knowing the type of an object is important, and it's useful to be able to write functions that can handle many different kinds of input. You can check that an object is an instance of a particular type using the `isinstance` function:

```
In [18]: a = 5
```

```
In [19]: isinstance(a, int)
```

```
Out[19]: True
```

`isinstance` can accept a tuple of types if you want to check that an object's type is among those present in the tuple:

```
In [20]: a = 5; b = 4.5
```

```
In [21]: isinstance(a, (int, float))  
Out[21]: True
```

```
In [22]: isinstance(b, (int, float))  
Out[22]: True
```

## Attributes and methods

Objects in Python typically have both attributes, other Python objects stored “inside” the object, and methods, functions associated with an object which can have access to the object’s internal data. Both of them are accessed via the syntax `obj.attribute_name`:

```
In [1]: a = 'foo'
```

```
In [2]: a.<Press Tab>  
a.capitalize a.format      a.isupper     a.rindex      a.strip  
a.center     a.index       a.join        a.rjust       a.swapcase  
a.count      a.isalnum    a.ljust       a.rpartition a.title  
a.decode     a.isalpha    a.lower       a.rsplit      a.translate  
a.encode     a.isdigit    a.lstrip      a.rstrip      a.upper  
a.endswith   a.islower   a.partition  a.split       a.zfill  
a.expandtabs a.isspace  a.replace    a.splitlines a.startswith
```

Attributes and methods can also be accessed by name using the `getattr` function:

```
>>> getattr(a, 'split')  
<function split>
```

In other languages, accessing objects by name is often referred to as “reflection”. While we will not extensively use the functions `getattr` and related functions `hasattr` and `setattr` in this book, they can be used very effectively to write generic, reusable code.

## “Duck” typing

Often you may not care about the type of an object but rather only whether it has certain methods or behavior. For example, you can verify that an object is iterable if it implemented the *iterator protocol*. For many objects, this means it has a `__iter__` “magic method”, though an alternative and better way to check is to try using the `iter` function:

```
def isiterable(obj):
    try:
        iter(obj)
        return True
    except TypeError: # not iterable
        return False
```

This function would return `True` for strings as well as most Python collection types:

```
In [24]: isiterable('a string')
Out[24]: True

In [25]: isiterable([1, 2, 3])
Out[25]: True

In [26]: isiterable(5)
Out[26]: False
```

A place where I use this functionality all the time is to write functions that can accept multiple kinds of input. A common case is writing a function that can accept any kind of sequence (list, tuple, ndarray) or even an iterator. You can first check if the object is a list (or a NumPy array) and, if it is not, convert it to be one:

```
if not isinstance(x, list) and isiterable(x):
    x = list(x)
```

## Imports

In Python a *module* is simply a `.py` file containing function and variable definitions along with such things imported from other `.py` files. Suppose that

we had the following module:

```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

def g(a, b):
    return a + b
```

If we wanted to access the variables and functions defined in `some_module.py`, from another file in the same directory we could do:

```
import some_module
result = some_module.f(5)
pi = some_module.PI
```

Or equivalently:

```
from some_module import f, g, PI
result = g(5, PI)
```

By using the `as` keyword you can give imports different variable names:

```
import some_module as sm
from some_module import PI as pi, g as gf

r1 = sm.f(pi)
r2 = gf(6, pi)
```

## Binary operators and comparisons

Most of the binary math operations and comparisons are as you might expect:

```
In [27]: 5 - 7
Out[27]: -2
```

```
In [28]: 12 + 21.5
Out[28]: 33.5
```

```
In [29]: 5 <= 2
Out[29]: False
```

See [Table 2-3](#) for all of the available binary operators.

To check if two references refer to the same object, use the `is` keyword. `is not` is also perfectly valid if you want to check that two objects are not the same:

```
In [30]: a = [1, 2, 3]
```

```
In [31]: b = a
```

```
# Note, the list function always creates a new list  
In [32]: c = list(a)
```

```
In [33]: a is b  
Out[33]: True
```

```
In [34]: a is not c  
Out[34]: True
```

Note this is not the same thing as comparing with `==`, because in this case we have:

```
In [35]: a == c  
Out[35]: True
```

A very common use of `is` and `is not` is to check if a variable is `None`, since there is only one instance of `None`:

```
In [36]: a = None
```

```
In [37]: a is None  
Out[37]: True
```

Table 2-3. Binary operators

| Operation           | Description   |
|---------------------|---|
| <code>a + b</code>  | Add <code>a</code> and <code>b</code>   |
| <code>a - b</code>  | Subtract <code>b</code> from <code>a</code>                                       |
| <code>a * b</code>  | Multiply <code>a</code> by <code>b</code>   |
| <code>a / b</code>  | Divide <code>a</code> by <code>b</code>   |
| <code>a // b</code> | Floor-divide <code>a</code> by <code>b</code> , dropping any fractional remainder |
| <code>a ** b</code> | Raise <code>a</code> to the <code>b</code> power                                  |

|  |  |
|--|--|
| <code>a &amp; b</code>                             | True if both <code>a</code> and <code>b</code> are True. For integers, take the bitwise AND.                               |
| <code>a   b</code>                                 | True if either <code>a</code> or <code>b</code> is True. For integers, take the bitwise OR.                                |
| <code>a ^ b</code>                                 | For booleans, True if <code>a</code> or <code>b</code> is True, but not both. For integers, take the bitwise EXCLUSIVE-OR. |
| <code>a == b</code>                                | True if <code>a</code> equals <code>b</code>   |
| <code>a != b</code>                                | True if <code>a</code> is not equal to <code>b</code>  |
| <code>a &lt;= b</code> ,<br><code>a &lt; b</code>  | True if <code>a</code> is less than (less than or equal) to <code>b</code>   |
| <code>a &gt; b</code> , <code>a<br/>&gt;= b</code> | True if <code>a</code> is greater than (greater than or equal) to <code>b</code>   |
| <code>a is b</code>                                | True if <code>a</code> and <code>b</code> reference same Python object   |
| <code>a is not b</code>                            | True if <code>a</code> and <code>b</code> reference different Python objects   |

## Strictness versus laziness

When using any programming language, it's important to understand *when* expressions are evaluated. Consider the simple expression:

```
a = b = c = 5
d = a + b * c
```

In Python, once these statements are evaluated, the calculation is immediately (or *strictly*) carried out, setting the value of `d` to 30. In another programming paradigm, such as in a pure functional programming language like Haskell, the value of `d` might not be computed until it is actually used elsewhere. The idea of deferring computations in this way is commonly known as *lazy evaluation*. Python, on the other hand, is a very *strict* (or eager) language. Nearly all of the time, computations and expressions are evaluated immediately. Even in the above simple expression, the result of `b * c` is computed as a separate step before adding it to `a`.

There are Python techniques, especially using iterators and generators, which can be used to achieve laziness. When performing very expensive computations which are only necessary some of the time, this can be an important technique in data-intensive applications.

## Mutable and immutable objects

Most objects in Python are mutable, such as lists, dicts, NumPy arrays, or most user-defined types (classes). This means that the object or values that they contain can be modified.

```
In [38]: a_list = ['foo', 2, [4, 5]]
```

```
In [39]: a_list[2] = (3, 4)
```

```
In [40]: a_list
Out[40]: ['foo', 2, (3, 4)]
```

Others, like strings and tuples, are immutable:

```
In [41]: a_tuple = (3, 5, (4, 5))
```

```
In [42]: a_tuple[1] = 'four'
```

```
-----
TypeError                                                 Traceback (most recent call last)
<ipython-input-42-b7966a9ae0f1> in <module>()
----> 1 a_tuple[1] = 'four'
TypeError: 'tuple' object does not support item assignment
```

Remember that just because you *can* mutate an object does not mean that you always *should*. Such actions are known as *side effects*. For example, when writing a function, any side effects should be explicitly communicated to the user in the function's documentation or comments. If possible, I recommend trying to avoid side effects and *favor immutability*, even though there may be mutable objects involved.

# Scalar Types

Python has a small set of built-in types for handling numerical data, strings, boolean (`True` or `False`) values, and dates and time. See [Table 2-4](#) for a list of the main scalar types. Date and time handling will be discussed separately as these are provided by the `datetime` module in the standard library.

Table 2-4. Standard Python Scalar Types

| Type               | Description   |
|--------------------|---|
| <code>None</code>  | The Python “null” value (only one instance of the <code>None</code> object exists)      |
| <code>str</code>   | String type. Holds Unicode (UTF-8 encoded) strings.                                     |
| <code>bytes</code> | Raw ASCII bytes (or Unicode encoded as bytes)   |
| <code>float</code> | Double-precision (64-bit) floating point number. Note there is no separate double type. |
| <code>bool</code>  | A <code>True</code> or <code>False</code> value   |
| <code>int</code>   | Arbitrary precision signed integer.   |

## Numeric types

The primary Python types for numbers are `int` and `float`. An `int` can store arbitrarily large numbers:

```
In [43]: ival = 17239871
```

```
In [44]: ival ** 6
Out[44]: 26254519291092456596965462913230729701102721
```

Floating point numbers are represented with the Python `float` type. Under the hood each one is a double-precision (64-bit) value. They can also be expressed using scientific notation:

```
In [45]: fval = 7.243
```

```
In [46]: fval2 = 6.78e-5
```

Integer division not resulting in a whole number will always yield a floating point number:

```
In [47]: 3 / 2
Out[47]: 1.5
```

To get C-style integer division (which drops the fractional part if the result is not a whole number), use the floor division operator `//`:

```
In [48]: 3 // 2
Out[48]: 1
```

## Strings

Many people use Python for its powerful and flexible built-in string processing capabilities. You can write *string literal* using either single quotes '`'` or double quotes `"`:

```
a = 'one way of writing a string'
b = "another way"
```

For multiline strings with line breaks, you can use triple quotes, either `'''` or `"""`:

```
c = """
This is a longer string that
spans multiple lines
"""
```

It may surprise you that this string `c` actually contains 4 lines of text; the line break after `"""` and on the last line are included in the string:

```
In [50]: len(c.split('\n')) # split on new line
Out[50]: 4
```

Python strings are immutable; you cannot modify a string without creating a new string:

```
In [51]: a = 'this is a string'
```

```
In [52]: a[10] = 'f'
```

```
-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-52-5ca625d1e504> in <module>()  
----> 1 a[10] = 'f'  
TypeError: 'str' object does not support item assignment  
  
In [53]: b = a.replace('string', 'longer string')  
  
In [54]: b  
Out[54]: 'this is a longer string'
```

Many Python objects can be converted to a string using the `str` function:

```
In [55]: a = 5.6  
  
In [56]: s = str(a)  
  
In [57]: s  
Out[57]: '5.6'
```

Strings are a sequence of Unicode characters and therefore can be treated like other sequences, such as lists and tuples:

```
In [58]: s = 'python'  
  
In [59]: list(s)  
Out[59]: ['p', 'y', 't', 'h', 'o', 'n']  
  
In [60]: s[:3]  
Out[60]: 'pyt'
```

The backslash character `\` is an *escape character*, meaning that it is used to specify special characters like newline `\n` or unicode characters. To write a string literal with backslashes, you need to escape them:

```
In [61]: s = '12\\34'  
  
In [62]: print(s)  
12\\34
```

If you have a string with a lot of backslashes and no special characters, you might find this a bit annoying. Fortunately you can preface the leading quote of the string with `r` which means that the characters should be interpreted as is:

```
In [63]: s = r'this\has\nospecial\characters'
```

```
In [64]: s
```

```
Out[64]: 'this\\has\\\\no\\\\special\\\\characters'
```

Adding two strings together concatenates them and produces a new string:

```
In [65]: a = 'this is the first half '
```

```
In [66]: b = 'and this is the second half'
```

```
In [67]: a + b
```

```
Out[67]: 'this is the first half and this is the second half'
```

String templating or formatting is another important topic. The number of ways to do so has expanded with the advent of Python 3, here I will briefly describe the mechanics of one of the main interfaces. String objects have a `format` method which can be used to substitute formatted arguments into the string, producing a new string.

```
In [68]: template = '{0:.2f} {1:s} are worth US${2:d}'
```

In this string,

- `{0:.2f}` means to format the first argument as a floating point number with 2 decimal places.
- `{1:s}` means to format the 2nd argument as a string.
- `{2:d}` means to format the 3rd argument as an exact integer.

To substitute arguments for these format parameters, we pass a sequence of arguments to the `format` method

```
In [69]: template.format(4.5560, 'Argentine Pesos', 1)  
Out[69]: '4.56 Argentine Pesos are worth US$1'
```

String formatting is a deep topic; there are multiple methods and numerous options and tweaks available to control how values are formatted in the resulting string. To learn more, I recommend you seek out more information on the official Python documentation.

I discuss general string processing as it relates to data analysis in more detail in [Chapter 8](#).

# Bytes and Unicode

In modern Python (that is, Python 3.0 and up), Unicode has become the first-class string type to enable more consistent handling of ASCII and non-ASCII text. In older versions of Python, strings were all bytes without any explicit Unicode encoding. You could convert to Unicode assuming you knew the character encoding. Let's look at an example:

```
In [70]: val = "español"
```

```
In [71]: val  
Out[71]: 'español'
```

We can convert this Unicode string to its UTF-8 bytes representation using the `encode` method:

```
In [72]: val_utf8 = val.encode('utf-8')

In [73]: val_utf8
Out[73]: b'espac\xc3\xb1ol'

In [74]: type(val_utf8)
Out[74]: bytes
```

Assuming you know the Unicode encoding of a bytes object, you can go back using the `decode` method:

```
In [75]: val_utf8.decode('utf-8')
Out[75]: 'español'
```

While it's become preferred to use UTF-8 for any encoding, for historical reasons you may encounter data in any number of different encodings:

```
In [76]: val.encode('latin1')
Out[76]: b'esp\u00e1\xf1ol'
```

```
In [77]: val.encode('utf-16')
Out[77]: b'\xff\xfee\x00s\x00p\x00a\x00\xf1\x00o\x001\x00'
```

```
In [78]: val.encode('utf-16le')
Out[78]: b'e\x00s\x00p\x00a\x00\xf1\x00o\x001\x00'
```

It is most common to encounter `bytes` objects in the context of working with files, where implicitly decoding all data to Unicode strings may not be desired.

Note that you can define your own byte literals by prefixing a string with `b`:

```
In [79]: bytes_val = b'this is bytes'
```

```
In [80]: bytes_val
Out[80]: b'this is bytes'
```

```
In [81]: decoded = bytes_val.decode('utf8')
```

```
In [82]: decoded # this is str (Unicode) now
Out[82]: 'this is bytes'
```

## Booleans

The two boolean values in Python are written as `True` and `False`. Comparisons and other conditional expressions evaluate to either `True` or `False`. Boolean values are combined with the `and` and `or` keywords:

```
In [83]: True and True
Out[83]: True
```

```
In [84]: False or True
Out[84]: True
```

Almost all built-in Python types and any class defining the `__nonzero__` magic method have a `True` or `False` interpretation in an `if` statement:

```
In [85]: a = [1, 2, 3]
....: if a:
....:     print('I found something!')
....:
I found something!
```

```
In [86]: b = []
....: if not b:
....:     print('Empty!')
....:
```

Empty!

Most objects in Python have a notion of true- or falseness. For example, empty sequences (lists, dicts, tuples, etc.) are treated as `False` if used in control flow (as above with the empty list `b`). You can see exactly what boolean value an object coerces to by invoking `bool` on it:

```
In [87]: bool([]), bool([1, 2, 3])
Out[87]: (False, True)

In [88]: bool('Hello world!'), bool('')
Out[88]: (True, False)

In [89]: bool(0), bool(1)
Out[89]: (False, True)
```

## Type casting

The `str`, `bool`, `int` and `float` types are also functions which can be used to cast values to those types:

```
In [90]: s = '3.14159'
```

```
In [91]: fval = float(s)
```

```
In [92]: type(fval)
Out[92]: float
```

```
In [93]: int(fval)
Out[93]: 3
```

```
In [94]: bool(fval)
Out[94]: True
```

```
In [95]: bool(0)
Out[95]: False
```

## None

`None` is the Python null value type. If a function does not explicitly return a value, it implicitly returns `None`.

```
In [96]: a = None  
  
In [97]: a is None  
Out[97]: True  
  
In [98]: b = 5  
  
In [99]: b is not None  
Out[99]: True
```

`None` is also a common default value for optional function arguments:

```
def add_and_maybe_multiply(a, b, c=None):  
    result = a + b  
  
    if c is not None:  
        result = result * c  
  
    return result
```

While a technical point, it's worth bearing in mind that `None` is not only a reserved keyword but also a unique instance of `NoneType`.

```
In [100]: type(None)  
Out[100]: NoneType
```

## Dates and times

The built-in Python `datetime` module provides `datetime`, `date`, and `time` types. The `datetime` type as you may imagine combines the information stored in `date` and `time` and is the most commonly used:

```
In [101]: from datetime import datetime, date, time  
  
In [102]: dt = datetime(2011, 10, 29, 20, 30, 21)  
  
In [103]: dt.day  
Out[103]: 29  
  
In [104]: dt.minute  
Out[104]: 30
```

Given a `datetime` instance, you can extract the equivalent `date` and `time`

objects by calling methods on the `datetime` of the same name:

```
In [105]: dt.date()  
Out[105]: datetime.date(2011, 10, 29)
```

```
In [106]: dt.time()  
Out[106]: datetime.time(20, 30, 21)
```

The `strftime` method formats a `datetime` as a string:

```
In [107]: dt.strftime('%m/%d/%Y %H:%M')  
Out[107]: '10/29/2011 20:30'
```

Strings can be converted (parsed) into `datetime` objects using the `strptime` function:

```
In [108]: datetime.strptime('20091031', '%Y%m%d')  
Out[108]: datetime.datetime(2009, 10, 31, 0, 0)
```

See [Table 2-5](#) for a full list of format specifications.

When aggregating or otherwise grouping time series data, it will occasionally be useful to replace time fields of a series of `datetimes`, for example replacing the minute and second fields with zero:

```
In [109]: dt.replace(minute=0, second=0)  
Out[109]: datetime.datetime(2011, 10, 29, 20, 0)
```

Since `datetime.datetime` is an immutable type, methods like these always produce new objects.

The difference of two `datetime` objects produces a `datetime.timedelta` type:

```
In [110]: dt2 = datetime(2011, 11, 15, 22, 30)
```

```
In [111]: delta = dt2 - dt
```

```
In [112]: delta  
Out[112]: datetime.timedelta(17, 7179)
```

```
In [113]: type(delta)  
Out[113]: datetime.timedelta
```

Adding a `timedelta` to a `datetime` produces a new shifted `datetime`:

```
In [114]: dt
Out[114]: datetime.datetime(2011, 10, 29, 20, 30, 21)
```

```
In [115]: dt + delta
Out[115]: datetime.datetime(2011, 11, 15, 22, 30)
```

Table 2-5. Datetime format specification (ISO C89 compatible)

| Type | Description  |
|------|--|
| %Y   | 4-digit year   |
| %y   | 2-digit year   |
| %m   | 2-digit month [01, 12]   |
| %d   | 2-digit day [01, 31]   |
| %H   | Hour (24-hour clock) [00, 23]  |
| %I   | Hour (12-hour clock) [01, 12]  |
| %M   | 2-digit minute [00, 59]  |
| %S   | Second [00, 61] (seconds 60, 61 account for leap seconds)  |
| %w   | Weekday as integer [0 (Sunday), 6]   |
| %U   | Week number of the year [00, 53]. Sunday is considered the first day of the week, and days before the first Sunday of the year are “week 0”. |
| %W   | Week number of the year [00, 53]. Monday is considered the first day of the week, and days before the first Monday of the year are “week 0”. |
| %z   | UTC time zone offset as +HHMM or -HHMM, empty if time zone naive   |
| %F   | Shortcut for %Y-%m-%d, for example 2012-4-18   |
| %D   | Shortcut for %m/%d/%y, for example 04/18/12  |

# Control Flow

Python has several built-in key words for conditional logic, loops, and other standard *control flow* concepts found in other programming languages.

## if, elif, and else

The `if` statement is one of the most well-known control flow statement types. It checks a condition which, if `True`, evaluates the code in the block that follows:

```
if x < 0:  
    print('It's negative')
```

An `if` statement can be optionally followed by one or more `elif` blocks and a catch-all `else` block if all of the conditions are `False`:

```
if x < 0:  
    print('It's negative')  
elif x == 0:  
    print('Equal to zero')  
elif 0 < x < 5:  
    print('Positive but smaller than 5')  
else:  
    print('Positive and larger than or equal to 5')
```

If any of the conditions is `True`, no further `elif` or `else` blocks will be reached. With a compound condition using `and` or `or`, conditions are evaluated left-to-right and will short circuit:

```
In [116]: a = 5; b = 7  
  
In [117]: c = 8; d = 4  
  
In [118]: if a < b or c > d:  
.....:     print('Made it')  
Made it
```

In this example, the comparison `c > d` never gets evaluated because the first comparison was `True`.

It is also possible to chain comparisons:

```
In [119]: 4 > 3 > 2 > 1
Out[119]: True
```

## for loops

for loops are for iterating over a collection (like a list or tuple) or an iterator. The standard syntax for a for loop is:

```
for value in collection:
    # do something with value
```

A for loop can be advanced to the next iteration, skipping the remainder of the block, using the continue keyword. Consider this code which sums up integers in a list and skips None values:

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
    if value is None:
        continue
    total += value
```

A for loop can be exited altogether using the break keyword. This code sums elements of the list until a 5 is reached:

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
    if value == 5:
        break
    total_until_5 += value
```

The break keyword only terminates the innermost for loop, any outer for loops will continue to run:

```
In [120]: for i in range(4):
.....:     for j in range(4):
.....:         if j > i:
.....:             break
.....:         print((i, j))
```

```
....:  
(0, 0)  
(1, 0)  
(1, 1)  
(2, 0)  
(2, 1)  
(2, 2)  
(3, 0)  
(3, 1)  
(3, 2)  
(3, 3)
```

As we will see in more detail, if the elements in the collection or iterator are sequences (tuples or lists, say), they can be conveniently *unpacked* into variables in the `for` loop statement:

```
for a, b, c in iterator:  
    # do something
```

## while loops

A `while` loop specifies a condition and a block of code that is to be executed until the condition evaluates to `False` or the loop is explicitly ended with `break`:

```
x = 256  
total = 0  
while x > 0:  
    if total > 500:  
        break  
    total += x  
    x = x // 2
```

## pass

`pass` is the “no-op” statement in Python. It can be used in blocks where no action is to be taken (or as a placeholder for code not yet implemented); it is only required because Python uses whitespace to delimit blocks:

```
if x < 0:  
    print('negative!')
```

```
elif x == 0:  
    # TODO: put something smart here  
    pass  
else:  
    print('positive!')
```

## Exception handling

Handling Python errors or *exceptions* gracefully is an important part of building robust programs. In data analysis applications, many functions only work on certain kinds of input. As an example, Python's `float` function is capable of casting a string to a floating point number, but fails with `ValueError` on improper inputs:

```
In [121]: float('1.2345')  
Out[121]: 1.2345
```

```
In [122]: float('something')  
-----  
ValueError Traceback (most recent call last)  
<ipython-input-122-439904410854> in <module>()  
----> 1 float('something')  
ValueError: could not convert string to float: 'something'
```

Suppose we wanted a version of `float` that fails gracefully, returning the input argument. We can do this by writing a function that encloses the call to `float` in a `try/except` block:

```
def attempt_float(x):  
    try:  
        return float(x)  
    except:  
        return x
```

The code in the `except` part of the block will only be executed if `float(x)` raises an exception:

```
In [124]: attempt_float('1.2345')  
Out[124]: 1.2345
```

```
In [125]: attempt_float('something')  
Out[125]: 'something'
```

You might notice that `float` can raise exceptions other than `ValueError`:

```
In [126]: float((1, 2))
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-126-842079ebb635> in <module>()
----> 1 float((1, 2))
TypeError: float() argument must be a string or a number, not 'tuple'
```

You might want to only suppress `ValueError`, since a `TypeError` (the input was not a string or numeric value) might indicate a legitimate bug in your program. To do that, write the exception type after `except`:

```
def attempt_float(x):
    try:
        return float(x)
    except ValueError:
        return x
```

We have then:

```
In [128]: attempt_float((1, 2))
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-128-9bdfd730cead> in <module>()
----> 1 attempt_float((1, 2))
<ipython-input-127-3e06b8379b6b> in attempt_float(x)
      1 def attempt_float(x):
      2     try:
----> 3         return float(x)
      4     except ValueError:
      5         return x
TypeError: float() argument must be a string or a number, not 'tuple'
```

You can catch multiple exception types by writing a tuple of exception types instead (the parentheses are required):

```
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError):
        return x
```

In some cases, you may not want to suppress an exception, but you want some

code to be executed regardless of whether the code in the `try` block succeeds or not. To do this, use `finally`:

```
f = open(path, 'w')

try:
    write_to_file(f)
finally:
    f.close()
```

Here, the file handle `f` will *always* get closed. Similarly, you can have code that executes only if the `try:` block succeeds using `else`:

```
f = open(path, 'w')

try:
    write_to_file(f)
except:
    print('Failed')
else:
    print('Succeeded')
finally:
    f.close()
```

## range

The `range` function returns an iterator that yields a sequence of evenly-spaced integers:

```
In [130]: range(10)
Out[130]: range(0, 10)
```

```
In [131]: list(range(10))
Out[131]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Both a start, end, and step can be given:

```
In [132]: list(range(0, 20, 2))
Out[132]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

As you can see, `range` produces integers up to but not including the endpoint. A common use of `range` is for iterating through sequences by index:

```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
    val = seq[i]
```

While you can use functions like `list` to store all the integers generated by `range` in some other data structure, often the default iterator form will be what you want. This snippet sums all numbers from 0 to 99999 that are multiples of 3 or 5:

```
sum = 0
for i in range(100000):
    # % is the modulo operator
    if i % 3 == 0 or i % 5 == 0:
        sum += i
```

While the range generated can be arbitrarily large, the memory use at any given time may be very small.

## Ternary Expressions

A *ternary expression* in Python allows you combine an `if-else` block which produces a value into a single line or expression. The syntax for this in Python is

```
value = true-expr if condition else
false-expr
```

Here, `true-expr` and `false-expr` can be any Python expressions. It has the identical effect as the more verbose

```
if condition:
    value = true-expr
else:
    value = false-expr
```

This is a more concrete example:

```
In [133]: x = 5
```

```
In [134]: 'Non-negative' if x >= 0 else 'Negative'
Out[134]: 'Non-negative'
```

As with `if-else` blocks, only one of the expressions will be executed. Thus, the “if” and “else” sides of the ternary expression could contain costly computations, but only the true branch is ever evaluated.

While it may be tempting to always use ternary expressions to condense your code, realize that you may sacrifice readability if the condition as well and the true and false expressions are very complex.

Closely related to ternary expressions is using the `or` keyword to choose one expression or variable over another:

```
In [135]: x, y, z = 5, None, 7
```

```
In [136]: y or x
Out[136]: 5
```

```
In [137]: z or x
Out[137]: 7
```

Here, if the left value is “false-y” (`bool(x)` returns `False`) then the right value is returned.

# **Chapter 3. Built-in Data Structures, Functions, and Files**

This chapter discusses capabilities built-in to the Python language which will be used ubiquitously throughout the book. While add-on libraries like pandas and NumPy add advanced computational functionality for larger datasets, they are designed to be used together with Python's built-in data manipulation tools.

I first start with Python's workhorse data structures: tuples, lists, dicts, and sets. Then, I discuss creating your own reusable Python functions. Finally, I explain the mechanics of Python file objects and interacting with your local hard drive.

# Data Structures and Sequences

Python's data structures are simple, but powerful. Mastering their use is a critical part of becoming a proficient Python programmer.

# Tuple

A tuple is a one-dimensional, fixed-length, *immutable* sequence of Python objects. The easiest way to create one is with a comma-separated sequence of values:

```
In [1]: tup = 4, 5, 6
```

```
In [2]: tup  
Out[2]: (4, 5, 6)
```

When defining tuples in more complicated expressions, it's often necessary to enclose the values in parentheses, as in this example of creating a tuple of tuples:

```
In [3]: nested_tup = (4, 5, 6), (7, 8)
```

```
In [4]: nested_tup  
Out[4]: ((4, 5, 6), (7, 8))
```

Any sequence or iterator can be converted to a tuple by invoking `tuple`:

```
In [5]: tuple([4, 0, 2])  
Out[5]: (4, 0, 2)
```

```
In [6]: tup = tuple('string')
```

```
In [7]: tup  
Out[7]: ('s', 't', 'r', 'i', 'n', 'g')
```

Elements can be accessed with square brackets `[]` as with most other sequence types. Like C, C++, Java, and many other languages, sequences are 0-indexed in Python:

```
In [8]: tup[0]  
Out[8]: 's'
```

While the objects stored in a tuple may be mutable themselves, once created it's not possible to modify which object is stored in each slot:

```
In [9]: tup = tuple(['foo', [1, 2], True])  
In [10]: tup[2] = False  
-----  
TypeError Traceback (most recent call last)  
<ipython-input-10-c7308343b841> in <module>()  
----> 1 tup[2] = False  
TypeError: 'tuple' object does not support item assignment  
  
# however  
In [11]: tup[1].append(3)  
  
In [12]: tup  
Out[12]: ('foo', [1, 2, 3], True)
```

Tuples can be concatenated using the `+` operator to produce longer tuples:

```
In [13]: (4, None, 'foo') + (6, 0) + ('bar',)  
Out[13]: (4, None, 'foo', 6, 0, 'bar')
```

Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple.

```
In [14]: ('foo', 'bar') * 4  
Out[14]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Note that the objects themselves are not copied, only the references to them.

## Unpacking tuples

If you try to *assign* to a tuple-like expression of variables, Python will attempt to *unpack* the value on the right-hand side of the equals sign:

```
In [15]: tup = (4, 5, 6)  
In [16]: a, b, c = tup  
In [17]: b  
Out[17]: 5
```

Even sequences with nested tuples can be unpacked:

```
In [18]: tup = 4, 5, (6, 7)
```

```
In [19]: a, b, (c, d) = tup
```

```
In [20]: d  
Out[20]: 7
```

Using this functionality it's easy to swap variable names, a task which in many languages might look like:

```
tmp = a  
a = b  
b = tmp  
  
b, a = a, b
```

One of the most common uses of variable unpacking when iterating over sequences of tuples or lists:

```
In [21]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
In [22]: for a, b, c in seq:  
....:     print('a={0}, b={1}, c={2}'.format(a, b, c))  
a=1, b=2, c=3  
a=4, b=5, c=6  
a=7, b=8, c=9
```

Another common use is for returning multiple values from a function. I'll cover this in more detail later..

The Python language recently acquired some more advanced tuple unpacking to help with situations where you may want to “pluck” a few elements from the beginning of a tuple. This uses the special syntax `*rest`, which is also used in function signatures to capture an arbitrarily long list of positional arguments.

```
In [23]: values = 1, 2, 3, 4, 5
```

```
In [24]: a, b, *rest = values
```

```
In [25]: a, b  
Out[25]: (1, 2)
```

```
In [26]: rest  
Out[26]: [3, 4, 5]
```

This `rest` bit is often something you want to discard. As a matter of convention, many Python programmers will use the underscore `_` for unwanted variables:

```
In [27]: a, b, *_ = values
```

## Tuple methods

Since the size and contents of a tuple cannot be modified, it is very light on instance methods. One useful one (also available on lists) is `count`, which counts the number of occurrences of a value:

```
In [28]: a = (1, 2, 2, 2, 3, 4, 2)
```

```
In [29]: a.count(2)
Out[29]: 4
```

# List

In contrast with tuples, lists are variable-length and their contents can be modified in place. They can be defined using square brackets [] or using the `list` type function:

```
In [30]: a_list = [2, 3, 7, None]  
In [31]: tup = ('foo', 'bar', 'baz')  
In [32]: b_list = list(tup)  
In [33]: b_list  
Out[33]: ['foo', 'bar', 'baz']  
In [34]: b_list[1] = 'peekaboo'  
In [35]: b_list  
Out[35]: ['foo', 'peekaboo', 'baz']
```

Lists and tuples are semantically similar as one-dimensional sequences of objects and thus can be used interchangeably in many functions.

The `list` function is frequently used in data processing as a way to materialize an iterator or generator expression:

```
In [36]: gen = range(10)  
In [37]: gen  
Out[37]: range(0, 10)  
In [38]: list(gen)  
Out[38]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

## Adding and removing elements

Elements can be appended to the end of the list with the `append` method:

```
In [39]: b_list.append('dwarf')
```

```
In [40]: b_list  
Out[40]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

Using `insert` you can insert an element at a specific location in the list:

```
In [41]: b_list.insert(1, 'red')  
  
In [42]: b_list  
Out[42]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

#### Warning

`insert` is computationally expensive compared with `append` as references to subsequent elements have to be shifted internally to make room for the new element. If you need to insert elements at both the beginning and end of a sequence, you may explore `collections.deque`, a double-ended queue, for this purpose.

The inverse operation to `insert` is `pop`, which removes and returns an element at a particular index:

```
In [43]: b_list.pop(2)  
Out[43]: 'peekaboo'  
  
In [44]: b_list  
Out[44]: ['foo', 'red', 'baz', 'dwarf']
```

Elements can be removed by value using `remove`, which locates the first such value and removes it from the last:

```
In [45]: b_list.append('foo')  
  
In [46]: b_list.remove('foo')  
  
In [47]: b_list  
Out[47]: ['red', 'baz', 'dwarf', 'foo']
```

If performance is not a concern, by using `append` and `remove`, a Python list can be used as a perfectly suitable “multi-set” data structure.

You can check if a list contains a value using the `in` keyword:

```
In [48]: 'dwarf' in b_list  
Out[48]: True
```

Note that checking whether a list contains a value is a lot slower than dicts and sets (to be introduced shortly) as Python makes a linear scan across the values of the list, whereas the others (based on hash tables) can make the check in constant time.

## Concatenating and combining lists

Similar to tuples, adding two lists together with + concatenates them:

```
In [49]: [4, None, 'foo'] + [7, 8, (2, 3)]  
Out[49]: [4, None, 'foo', 7, 8, (2, 3)]
```

If you have a list already defined, you can append multiple elements to it using the `extend` method:

```
In [50]: x = [4, None, 'foo']  
  
In [51]: x.extend([7, 8, (2, 3)])  
  
In [52]: x  
Out[52]: [4, None, 'foo', 7, 8, (2, 3)]
```

Note that list concatenation by addition is a comparatively expensive operation since a new list must be created and the objects copied over. Using `extend` to append elements to an existing list, especially if you are building up a large list, is usually preferable. Thus,

```
everything = []  
for chunk in list_of_lists:  
    everything.extend(chunk)
```

is faster than the concatenative alternative

```
everything = []  
for chunk in list_of_lists:  
    everything = everything + chunk
```

## Sorting

A list can be sorted in-place (without creating a new object) by calling its `sort` function:

```
In [53]: a = [7, 2, 5, 1, 3]
```

```
In [54]: a.sort()
```

```
In [55]: a  
Out[55]: [1, 2, 3, 5, 7]
```

`sort` has a few options that will occasionally come in handy. One is the ability to pass a secondary *sort key*, i.e. a function that produces a value to use to sort the objects. For example, we could sort a collection of strings by their lengths:

```
In [56]: b = ['saw', 'small', 'He', 'foxes', 'six']
```

```
In [57]: b.sort(key=len)
```

```
In [58]: b  
Out[58]: ['He', 'saw', 'six', 'small', 'foxes']
```

## Binary search and maintaining a sorted list

The built-in `bisect` module implements binary-search and insertion into a sorted list. `bisect.bisect` finds the location where an element should be inserted to keep it sorted, while `bisect.insort` actually inserts the element into that location:

```
In [59]: import bisect
```

```
In [60]: c = [1, 2, 2, 2, 3, 4, 7]
```

```
In [61]: bisect.bisect(c, 2)  
Out[61]: 4
```

```
In [62]: bisect.bisect(c, 5)  
Out[62]: 6
```

```
In [63]: bisect.insort(c, 6)
```

```
In [64]: c  
Out[64]: [1, 2, 2, 2, 3, 4, 6, 7]
```

### Caution

The `bisect` module functions do not check whether the list is sorted as doing so would be computationally expensive. Thus, using them with an unsorted list will succeed without error but may lead to incorrect results.

## Slicing

You can select sections of list-like types (arrays, tuples, NumPy arrays, pandas Series) by using slice notation, which in its basic form consists of `start:stop` passed to the indexing operator `[]`:

```
In [65]: seq = [7, 2, 3, 7, 5, 6, 0, 1]
```

```
In [66]: seq[1:5]
Out[66]: [2, 3, 7, 5]
```

Slices can also be assigned to with a sequence:

```
In [67]: seq[3:4] = [6, 3]
```

```
In [68]: seq
Out[68]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

While element at the `start` index is included, the `stop` index is *not included*, so that the number of elements in the result is `stop - start`.

Either the `start` or `stop` can be omitted in which case they default to the start of the sequence and the end of the sequence, respectively:

```
In [69]: seq[:5]
Out[69]: [7, 2, 3, 6, 3]
```

```
In [70]: seq[3:]
Out[70]: [6, 3, 5, 6, 0, 1]
```

Negative indices slice the sequence relative to the end:

```
In [71]: seq[-4:]
Out[71]: [5, 6, 0, 1]
```

```
In [72]: seq[-6:-2]
Out[72]: [6, 3, 5, 6]
```

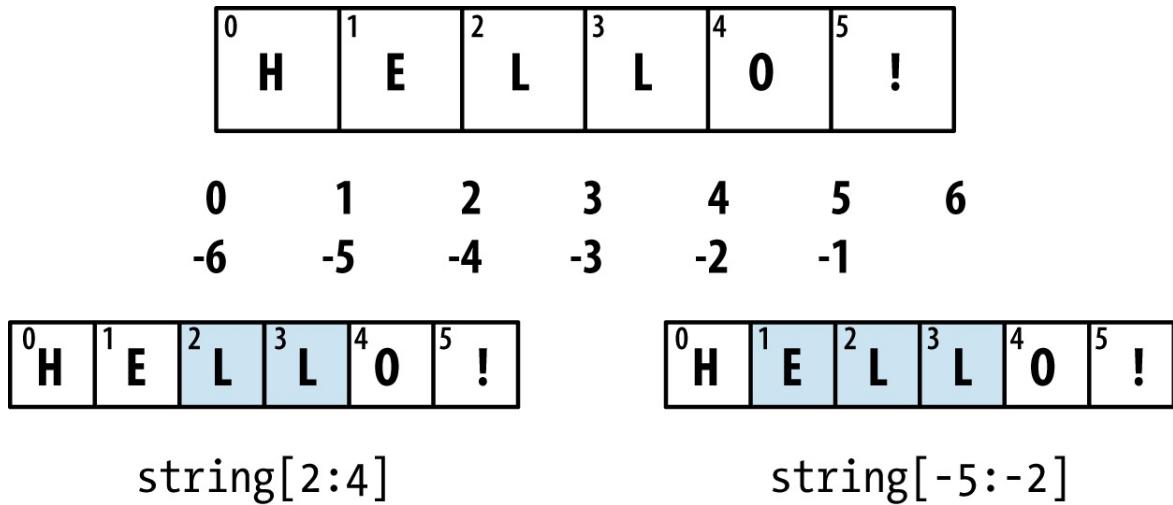
Slicing semantics takes a bit of getting used to, especially if you're coming from R or MATLAB. See [Figure 3-1](#) for a helpful illustrating of slicing with positive and negative integers.

A step can also be used after a second colon to, say, take every other element:

```
In [73]: seq[::-2]
Out[73]: [7, 3, 3, 6, 1]
```

A clever use of this is to pass `-1` which has the useful effect of reversing a list or tuple:

```
In [74]: seq[::-1]
Out[74]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```



**Figure 3-1. Illustration of Python slicing conventions**

# Built-in Sequence Functions

Python has a handful of useful sequence functions that you should familiarize yourself with and use at any opportunity.

## enumerate

It's common when iterating over a sequence to want to keep track of the index of the current item. A do-it-yourself approach would look like:

```
i = 0
for value in collection:
    # do something with value
    i += 1
```

Since this is so common, Python has a built-in function `enumerate` which returns a sequence of `(i, value)` tuples:

```
for i, value in enumerate(collection):
    # do something with value
```

When indexing data, a useful pattern that uses `enumerate` is computing a dict mapping the values of a sequence (which are assumed to be unique) to their locations in the sequence:

```
In [75]: some_list = ['foo', 'bar', 'baz']

In [76]: mapping = dict((v, i) for i, v in enumerate(some_list))

In [77]: mapping
Out[77]: {'bar': 1, 'baz': 2, 'foo': 0}
```

## sorted

The `sorted` function returns a new sorted list from the elements of any sequence:

```
In [78]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[78]: [0, 1, 2, 3, 6, 7]
```

```
In [79]: sorted('horse race')
Out[79]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

A common pattern for getting a sorted list of the unique elements in a sequence is to combine `sorted` with `set`:

```
In [80]: sorted(set('this is just some string'))
Out[80]: [' ', 'e', 'g', 'h', 'i', 'j', 'm', 'n', 'o', 'r', 's']
```

## zip

`zip` “pairs” up the elements of a number of lists, tuples, or other sequences, to create a list of tuples:

```
In [81]: seq1 = ['foo', 'bar', 'baz']
```

```
In [82]: seq2 = ['one', 'two', 'three']
```

```
In [83]: zipped = zip(seq1, seq2)
```

```
In [84]: list(zipped)
Out[84]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

`zip` can take an arbitrary number of sequences, and the number of elements it produces is determined by the *shortest* sequence:

```
In [85]: seq3 = [False, True]
```

```
In [86]: list(zip(seq1, seq2, seq3))
Out[86]: [('foo', 'one', False), ('bar', 'two', True)]
```

A very common use of `zip` is for simultaneously iterating over multiple sequences, possibly also combined with `enumerate`:

```
In [87]: for i, (a, b) in enumerate(zip(seq1, seq2)):
....:     print('{0}: {1}, {2}'.format(i, a, b))
....:
0: foo, one
1: bar, two
2: baz, three
```

Given a “zipped” sequence, `zip` can be applied in a clever way to “unzip” the sequence. Another way to think about this is converting a list of *rows* into a list of *columns*. The syntax, which looks a bit magical, is:

```
In [88]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
....:                  ('Schilling', 'Curt'))]

In [89]: first_names, last_names = zip(*pitchers)

In [90]: first_names
Out[90]: ('Nolan', 'Roger', 'Schilling')

In [91]: last_names
Out[91]: ('Ryan', 'Clemens', 'Curt')
```

We’ll look in more detail at the use of `*` in a function call. It is equivalent to the following:

```
zip(seq[0], seq[1], ..., seq[len(seq) - 1])
```

## reversed

`reversed` iterates over the elements of a sequence in reverse order:

```
In [92]: list(reversed(range(10)))
Out[92]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Keep in mind that `reversed` is an iterator, so it does not iterate until materialized (e.g. with `list` or a `for` loop).

# Dict

`dict` is likely the most important built-in Python data structure. A more common name for it is *hash map* or *associative array*. It is a flexibly-sized collection of *key-value* pairs, where *key* and *value* are Python objects. One way to create one is by using curly braces `{ }` and using colons to separate keys and values:

```
In [93]: empty_dict = {}

In [94]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}

In [95]: d1
Out[95]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

Elements can be accessed and inserted or set using the same syntax as accessing elements of a list or tuple:

```
In [96]: d1[7] = 'an integer'

In [97]: d1
Out[97]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}

In [98]: d1['b']
Out[98]: [1, 2, 3, 4]
```

You can check if a dict contains a key using the same syntax as with checking whether a list or tuple contains a value:

```
In [99]: 'b' in d1
Out[99]: True
```

Values can be deleted either using the `del` keyword or the `pop` method (which simultaneously returns the value and deletes the key):

```
In [100]: d1[5] = 'some value'

In [101]: d1['dummy'] = 'another value'

In [102]: del d1[5]
```

```
In [103]: ret = d1.pop('dummy')
```

```
In [104]: ret
```

```
Out[104]: 'another value'
```

The `keys` and `values` method give you iterators of the dict's keys and values, respectively. While the key-value pairs are not in any particular order, these functions output the keys and values in the same order:

```
In [105]: list(d1.keys())
Out[105]: ['a', 'b', 7]
```

```
In [106]: list(d1.values())
```

```
Out[106]: ['some value', [1, 2, 3, 4], 'an integer']
```

One dict can be merged into another using the `update` method:

```
In [107]: d1.update({'b' : 'foo', 'c' : 12})
```

```
In [108]: d1
```

```
Out[108]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

## Creating dicts from sequences

It's common to occasionally end up with two sequences that you want to pair up element-wise in a dict. As a first cut, you might write code like this:

```
mapping = {}
for key, value in zip(key_list, value_list):
    mapping[key] = value
```

Since a dict is essentially a collection of 2-tuples, it should be no shock that the `dict` type function accepts a list of 2-tuples:

```
In [109]: mapping = dict(zip(range(5), reversed(range(5))))
```

```
In [110]: mapping
```

```
Out[110]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

In a later section we'll talk about *dict comprehensions*, another elegant way to construct dicts.

## Default values

It's very common to have logic like:

```
if key in some_dict:  
    value = some_dict[key]  
else:  
    value = default_value
```

Thus, the dict methods `get` and `pop` can take a default value to be returned, so that the above `if-else` block can be written simply as:

```
value = some_dict.get(key, default_value)
```

`get` by default will return `None` if the key is not present, while `pop` will raise an exception. With *setting* values, a common case is for the values in a dict to be other collections, like lists. For example, you could imagine categorizing a list of words by their first letters as a dict of lists:

```
In [111]: words = ['apple', 'bat', 'bar', 'atom', 'book']  
  
In [112]: by_letter = {}  
  
In [113]: for word in words:  
....:     letter = word[0]  
....:     if letter not in by_letter:  
....:         by_letter[letter] = [word]  
....:     else:  
....:         by_letter[letter].append(word)  
....:  
  
In [114]: by_letter  
Out[114]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

The `setdefault` dict method is for precisely this purpose. The `if-else` block above can be rewritten as:

```
by_letter.setdefault(letter, []).append(word)
```

The built-in `collections` module has a useful class, `defaultdict`, which makes this even easier. One is created by passing a type or function for generating the default value for each slot in the dict:

```
from collections import defaultdict
by_letter = defaultdict(list)
for word in words:
    by_letter[word[0]].append(word)
```

The initializer to `defaultdict` only needs to be a callable object (e.g. any function), not necessarily a type. Thus, if you wanted the default value to be 4 you could pass a function returning 4

```
counts = defaultdict(lambda: 4)
```

## Valid dict key types

While the values of a dict can be any Python object, the keys generally have to be immutable objects like scalar types (`int`, `float`, `string`) or tuples (all the objects in the tuple need to be immutable, too). The technical term here is *hashability*. You can check whether an object is hashable (can be used as a key in a dict) with the `hash` function:

```
In [115]: hash('string')
Out[115]: -5220963831342811611

In [116]: hash((1, 2, (2, 3)))
Out[116]: 1097636502276347782

In [117]: hash((1, 2, [2, 3])) # fails because lists are mutable
-----
TypeError                                     Traceback (most recent call last)
<ipython-input-117-800cd14ba8be> in <module>()
----> 1 hash((1, 2, [2, 3])) # fails because lists are mutable
TypeError: unhashable type: 'list'
```

To use a list as a key, one option is to convert it to a tuple, which can be hashed as long as its elements also can:

```
In [118]: d = {}

In [119]: d[tuple([1, 2, 3])] = 5

In [120]: d
Out[120]: {(1, 2, 3): 5}
```

# Set

A set is an unordered collection of unique elements. You can think of them like dicts, but keys only, no values. A set can be created in two ways: via the `set` function or using a *set literal* with curly braces:

```
In [121]: set([2, 2, 2, 1, 3, 3])  
Out[121]: {1, 2, 3}
```

```
In [122]: {2, 2, 2, 1, 3, 3}  
Out[122]: {1, 2, 3}
```

Sets support mathematical *set operations* like union, intersection, difference, and symmetric difference. See [Table 3-1](#) for a list of commonly used set methods.

```
In [123]: a = {1, 2, 3, 4, 5}
```

```
In [124]: b = {3, 4, 5, 6, 7, 8}
```

```
In [125]: a | b # union (or)  
Out[125]: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
In [126]: a & b # intersection (and)  
Out[126]: {3, 4, 5}
```

```
In [127]: a - b # difference  
Out[127]: {1, 2}
```

```
In [128]: a ^ b # symmetric difference (xor)  
Out[128]: {1, 2, 6, 7, 8}
```

Each of the logical set operations have in place counterparts, which enable you to replace the contents of the set on the left side of the operation with the result. For very large sets, this will be more efficient.

```
In [129]: c = a.copy()
```

```
In [130]: c |= b
```

```
In [131]: c
```

```
Out[131]: {1, 2, 3, 4, 5, 6, 7, 8}

In [132]: d = a.copy()

In [133]: d &= b

In [134]: d
Out[134]: {3, 4, 5}
```

Like dicts, set elements generally must be immutable. To have a list-like elements, you must convert it to a tuple:

```
In [135]: my_data = [1, 2, 3, 4]

In [136]: my_set = {tuple(my_data) }

In [137]: my_set
Out[137]: {(1, 2, 3, 4)}
```

You can also check if a set is a subset of (is contained in) or a superset of (contains all elements of) another set:

```
In [138]: a_set = {1, 2, 3, 4, 5}

In [139]: {1, 2, 3}.issubset(a_set)
Out[139]: True

In [140]: a_set.issuperset({1, 2, 3})
Out[140]: True
```

Sets are equal if and only if their contents are equal:

```
In [141]: {1, 2, 3} == {3, 2, 1}
Out[141]: True
```

Table 3-1. Python Set Operations

| Function  | Alternate Syntax | Description  |
|-----------|------------------|--|
| a.add(x)  | N/A              | Add element x to the set a   |
| a.clear() | N/A              | Reset the set a to an empty state, discarding all of its elements. |

|   |                         |  |
|---|-------------------------|--|
| <code>a.remove(x)</code>                      | N/A                     | Remove element <code>x</code> from the set <code>a</code>  |
| <code>a.pop()</code>                          | N/A                     | Remove an arbitrary element from the set <code>a</code> , raising <code>KeyError</code> if the set is empty.     |
| <code>a.union(b)</code>                       | <code>a   b</code>      | All of the unique elements in <code>a</code> and <code>b</code> .  |
| <code>a.update(b)</code>                      | <code>a  = b</code>     | Set the contents of <code>a</code> to be the union of the elements in <code>a</code> and <code>b</code> .        |
| <code>a.intersection(b)</code>                | <code>a &amp; b</code>  | All of the elements in <i>both</i> <code>a</code> and <code>b</code> .   |
| <code>a.intersection_update(b)</code>         | <code>a &amp;= b</code> | Set the contents of <code>a</code> to be the intersection of the elements in <code>a</code> and <code>b</code> . |
| <code>a.difference(b)</code>                  | <code>a - b</code>      | The elements in <code>a</code> that are not in <code>b</code> .  |
| <code>a.difference_update(b)</code>           | <code>a -= b</code>     | Set <code>a</code> to the elements in <code>a</code> that are not in <code>b</code> .                            |
| <code>a.symmetric_difference(b)</code>        | <code>a ^ b</code>      | All of the elements in either <code>a</code> or <code>b</code> but <i>not both</i> .                             |
| <code>a.symmetric_difference_update(b)</code> | <code>a ^= b</code>     | Set <code>a</code> to contain the elements in either <code>a</code> or <code>b</code> but <i>not both</i> .      |
| <code>a.issubset(b)</code>                    | N/A                     | True if the elements of <code>a</code> are all contained in <code>b</code> .                                     |
| <code>a.issuperset(b)</code>                  | N/A                     | True if the elements of <code>b</code> are all contained in <code>a</code> .                                     |
| <code>a.isdisjoint(b)</code>                  | N/A                     | True if <code>a</code> and <code>b</code> have no elements in common.  |

# List, Set, and Dict Comprehensions

*List comprehensions* are one of the most-loved Python language features. They allow you to concisely form a new list by filtering the elements of a collection and transforming the elements passing the filter in one concise expression. They take the basic form:

```
[expr for val in collection if condition]
```

This is equivalent to the following `for` loop:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

The filter condition can be omitted, leaving only the expression. For example, given a list of strings, we could filter out strings with length 2 or less and also convert them to uppercase like this:

```
In [142]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
In [143]: [x.upper() for x in strings if len(x) > 2]
Out[143]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Set and dict comprehensions are a natural extension, producing sets and dicts in a idiomatically similar way instead of lists. A dict comprehension looks like this:

```
dict_comp = {key-expr : value-expr for value in collection
             if condition}
```

A set comprehension looks like the equivalent list comprehension except with curly braces instead of square brackets:

```
set_comp = {expr for value in collection if condition}
```

Like list comprehensions, set and dict comprehensions are mostly conveniences, but they similarly can make code both easier to write and read.

Consider the list of strings above. Suppose we wanted a set containing just the lengths of the strings contained in the collection; this could be easily computed using a set comprehension:

```
In [144]: unique_lengths = {len(x) for x in strings}
```

```
In [145]: unique_lengths
Out[145]: {1, 2, 3, 4, 6}
```

This could also be expressed more functionally using the `map` function, introduced shortly:

```
In [146]: set(map(len, strings))
Out[146]: {1, 2, 3, 4, 6}
```

As a simple dict comprehension example, we could create a lookup map of these strings to their locations in the list:

```
In [147]: loc_mapping = {val : index for index, val in enumerate(strings)}
In [148]: loc_mapping
Out[148]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'pyt': 5}
```

Note that this dict could be equivalently constructed by:

```
loc_mapping = dict((val, idx) for idx, val in enumerate(strings))
```

The dict comprehension version is shorter and cleaner in my opinion.

## Nested list comprehensions

Suppose we have a list of lists containing some boy and girl names:

```
In [149]: all_data = [['Tom', 'Billy', 'Jefferson', 'Andrew', 'John'],
.....:                  ['Susie', 'Casey', 'Jill', 'Ana', 'Eva', 'Sarah']]
```

You might have gotten these names from a couple of files and decided to keep the boy and girl names separate. Now, suppose we wanted to get a single list containing all names with two or more e's in them. We could certainly do this with a simple `for` loop:

```
names_of_interest = []
for names in all_data:
    enough_es = [name for name in names if name.count('e') >= 2]
    names_of_interest.extend(enough_es)
```

You can actually wrap this whole operation up in a single *nested list comprehension*, which will look like:

```
In [150]: result = [name for names in all_data for name in names
.....:           if name.count('e') >= 2]
```

```
In [151]: result
Out[151]: ['Jefferson', 'Wesley', 'Steven', 'Jennifer', 'Stepha
```

At first, nested list comprehensions are a bit hard to wrap your head around. The `for` parts of the list comprehension are arranged according to the order of nesting, and any filter condition is put at the end as before. Here is another example where we “flatten” a list of tuples of integers into a simple list of integers:

```
In [152]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
In [153]: flattened = [x for tup in some_tuples for x in tup]
```

```
In [154]: flattened
Out[154]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Keep in mind that the order of the `for` expressions would be the same if you wrote a nested `for` loop instead of a list comprehension:

```
flattened = []

for tup in some_tuples:
    for x in tup:
        flattened.append(x)
```

You can have arbitrarily many levels of nesting, though if you have more than two or three levels of nesting you should probably start to question whether this makes sense from a code readability standpoint. It’s important to distinguish the above syntax from a list comprehension inside a list comprehension, which is also perfectly valid:

```
In [229]: [[x for x in tup] for tup in some_tuples]
```

# Functions

Functions are the primary and most important method of code organization and reuse in Python. There may not be such a thing as having too many functions. In fact, I would argue that most programmers doing data analysis don't write enough functions! As you have in prior examples, functions are declared using the `def` keyword and returned from using the `return` keyword:

```
def my_function(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

There is no issue with having multiple `return` statements. If the end of a function is reached without encountering a `return` statement, `None` is returned automatically.

Each function can have some number of *positional* arguments and some number of *keyword* arguments. Keyword arguments are most commonly used to specify default values or optional arguments. In the above function, `x` and `y` are positional arguments while `z` is a keyword argument. This means that it can be called in either of these equivalent ways:

```
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
```

The main restriction on function arguments is that the keyword arguments *must* follow the positional arguments (if any). You can specify keyword arguments in any order; this frees you from having to remember which order the function arguments were specified in and only what their names are.

# Namespaces, Scope, and Local Functions

Functions can access variables in two different scopes: *global* and *local*. An alternate and more descriptive name describing a variable scope in Python is a *namespace*. Any variables that are assigned within a function by default are assigned to the local namespace. The local namespace is created when the function is called and immediately populated by the function's arguments. After the function is finished, the local namespace is destroyed (with some exceptions, see section on closures below). Consider the following function:

```
def func():
    a = []
    for i in range(5):
        a.append(i)
```

Upon calling `func()`, the empty list `a` is created, 5 elements are appended, then `a` is destroyed when the function exits. Suppose instead we had declared `a`

```
a = []
def func():
    for i in range(5):
        a.append(i)
```

Assigning global variables within a function is possible, but those variables must be declared as global using the `global` keyword:

```
In [155]: a = None
In [156]: def bind_a_variable():
.....:     global a
.....:     a = []
.....: bind_a_variable()
.....:
In [157]: print(a)
[]
```

**Caution**

I generally discourage people from using the `global` keyword frequently. Typically global variables are used to store some kind of state in a system. If you find yourself using a lot of them, it's probably a sign that some object-oriented programming (using classes) is in order.

Functions can be declared anywhere, and there is no problem with having *local* functions that are dynamically created when a function is called:

```
def outer_function(x, y, z):
    def inner_function(a, b, c):
        pass
    pass
```

In the above code, the `inner_function` will not exist until `outer_function` is called. As soon as `outer_function` is done executing, the `inner_function` is destroyed.

Nested inner functions can access the local namespace of the enclosing function, but they cannot bind new variables in it. I'll talk a bit more about this in the section on closures.

Strictly speaking, all functions are local to some scope, that scope may just be the module level scope.

# Returning Multiple Values

When I first programmed in Python after having programmed in Java and C++, one of my favorite features was the ability to return multiple values from a function. Here's a simple example:

```
def f():
    a = 5
    b = 6
    c = 7
    return a, b, c

a, b, c = f()
```

In data analysis and other scientific applications, you will likely find yourself doing this very often as many functions may have multiple outputs, whether those are data structures or other auxiliary data computed inside the function. If you think about tuple packing and unpacking from earlier in this chapter, you may realize that what's happening here is that the function is actually just returning *one* object, namely a tuple, which is then being unpacked into the result variables. In the above example, we could have done instead:

```
return_value = f()
```

In this case, `return_value` would be a 3-tuple with the three returned variables. A potentially attractive alternative to returning multiple values like above might be to return a dict instead:

```
def f():
    a = 5
    b = 6
    c = 7
    return {'a' : a, 'b' : b, 'c' : c}
```

# Functions Are Objects

Since Python functions are objects, many constructs can be easily expressed that are difficult to do in other languages. Suppose we were doing some data cleaning and needed to apply a bunch of transformations to the following list of strings:

```
states = ['    Alabama ', 'Georgia!', 'Georgia', 'georgia', 'Florid
          'south    carolina##', 'West virginia?']
```

Anyone who has ever worked with user-submitted survey data can expect messy results like these. Lots of things need to happen to make this list of strings uniform and ready for analysis: whitespace stripping, removing punctuation symbols, and proper capitalization. As a first pass, we might write some code like:

```
import re  # Regular expression module

def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub('![#?]', '', value) # remove punctuation
        value = value.title()
        result.append(value)
    return result
```

The result looks like this:

```
In [15]: clean_strings(states)
Out[15]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

An alternate approach that you may find useful is to make a list of the

operations you want to apply to a particular set of strings:

```
def remove_punctuation(value):
    return re.sub('![#?]', '', value)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result
```

Then we have

```
In [22]: clean_strings(states, clean_ops)
Out[22]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

A more *functional* pattern like this enables you to easily modify how the strings are transformed at a very high level. The `clean_strings` function is also now more reusable!

You can naturally use functions as arguments to other functions like the built-in `map` function, which applies a function to a collection of some kind:

```
In [23]: map(remove_punctuation, states)
Out[23]:
['Alabama',
 'Georgia',
 'Georgia',
 'georgia',
 'FlOrIda',
 'south carolina',
 'West virginia']
```

# Anonymous (`lambda`) Functions

Python has support for so-called *anonymous* or *lambda* functions, which are really just simple functions consisting of a single statement, the result of which is the return value. They are defined using the `lambda` keyword, which has no meaning other than “we are declaring an anonymous function.”

```
def short_function(x):
    return x * 2

equiv_anon = lambda x: x * 2
```

I usually refer to these as lambda functions in the rest of the book. They are especially convenient in data analysis because, as you’ll see, there are many cases where data transformation functions will take functions as arguments. It’s often less typing (and clearer) to pass a lambda function as opposed to writing a full-out function declaration or even assigning the lambda function to a local variable. For example, consider this silly example:

```
def apply_to_list(some_list, f):
    return [f(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

You could also have written `[x * 2 for x in ints]`, but here we were able to succinctly pass a custom operator to the `apply_to_list` function.

As another example, suppose you wanted to sort a collection of strings by the number of distinct letters in each string:

```
In [158]: strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

Here we could pass a lambda function to the list’s `sort` method:

```
In [159]: strings.sort(key=lambda x: len(set(list(x))))
```

```
In [160]: strings
Out[160]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

**Note**

One reason lambda functions are called anonymous functions is that the function object itself is never given an explicit `__name__` attribute like functions declared with the `def` keyword are given.

# Closures: Functions that Return Functions

Closures are nothing to fear. They can actually be a very useful and powerful tool in the right circumstance! In a nutshell, a closure is any *dynamically-generated* function returned by another function. The key property is that the returned function has access to the variables in the local namespace where it was created. Here is a very simple example:

```
def make_closure(a):
    def closure():
        print('I know the secret: {}'.format(a))
    return closure

closure = make_closure(5)
```

The difference between a closure and a regular Python function is that the closure continues to have access to the namespace (the function) where it was created, even though that function is done executing. So in the above case, the returned closure will always print `I know the secret: 5` whenever you call it. While it's common to create closures whose internal state (in this example, only the value of `a`) is static, you can just as easily have a mutable object like a dict, set, or list that can be modified. For example, here's a function that returns a function that keeps track of arguments it has been called with:

```
def make_watcher():
    have_seen = {}

    def has_been_seen(x):
        if x in have_seen:
            return True
        else:
            have_seen[x] = True
            return False

    return has_been_seen
```

Using this on a sequence of integers I obtain:

```
In [162]: watcher = make_watcher()
```

```
In [163]: vals = [5, 6, 1, 5, 1, 6, 3, 5]

In [164]: [watcher(x) for x in vals]
Out[164]: [False, False, False, True, True, True, False, True]
```

However, one technical limitation to keep in mind is that while you can mutate any internal state objects (like adding key-value pairs to a dict), you cannot create new local variables in the enclosing function scope. One way to work around this is to modify a dict or list rather than assigning to a variable name:

```
def make_counter():
    count = [0]
    def counter():
        # increment and return the current count
        count[0] += 1
        return count[0]
    return counter

counter = make_counter()
```

#### Note

Python has a special `nonlocal` keyword which enables you to overcome this limitation.

```
In [165]: def make_counter():
....:     count = 0
....:     def counter():
....:         nonlocal count
....:         # increment and return the current count
....:         count += 1
....:         return count
....:     return counter
....:
....: counter = make_counter()
....:

In [166]: counter()
Out[166]: 1

In [167]: counter()
Out[167]: 2
```

You might be wondering why this is useful. In practice, you can write very

general functions with lots of options, then fabricate simpler, more specialized functions. Here's an example of creating a string formatting function:

```
def format_and_pad(template, space):
    def formatter(x):
        return template.format(x).rjust(space)

    return formatter
```

You could then create a floating point formatter that always returns a length-15 string like so:

```
In [169]: fmt = format_and_pad('{0:.4f}', 15)
In [170]: fmt(1.756)
Out[170]: '      1.7560'
```

If you learn more about object-oriented programming in Python, you might observe that these patterns also could be implemented (albeit more verbosely) using classes.

# Extended Call Syntax with \*args, \*\*kwargs

The way that function arguments work under the hood in Python is actually very simple. When you write `func(a, b, c, d=some, e=value)`, the positional and keyword arguments are actually packed up into a tuple and dict, respectively. So the internal function receives a tuple `args` and dict `kwargs` and internally does the equivalent of:

```
a, b, c = args
d = kwargs.get('d', d_default_value)
e = kwargs.get('e', e_default_value)
```

This all happens nicely behind the scenes. Of course, it also does some error checking and allows you to specify some of the positional arguments as keywords also (even if they aren't keyword in the function declaration!).

```
def say_hello_then_call_f(f, *args, **kwargs):
    print 'args is', args
    print 'kwargs is', kwargs
    print("Hello! Now I'm going to call %s" % f)
    return f(*args, **kwargs)

def g(x, y, z=1):
    return (x + y) / z
```

Then if we call `g` with `say_hello_then_call_f` we get:

```
In [8]: say_hello_then_call_f(g, 1, 2, z=5.)
args is (1, 2)
kwargs is {'z': 5.0}
Hello! Now I'm going to call <function g at 0x2dd5cf8>
Out[8]: 0.6
```

There are some other advanced capabilities with function arguments that you can investigate on your own, such as *keyword-only arguments*.

# Currying: Partial Argument Application

*Currying* is computer science jargon (named after the mathematician Haskell Curry) which means deriving new functions from existing ones by *partial argument application*. For example, suppose we had a trivial function that adds two numbers together:

```
def add_numbers(x, y):  
    return x + y
```

Using this function, we could derive a new function of one variable, `add_five`, that adds 5 to its argument:

```
add_five = lambda y: add_numbers(5, y)
```

The second argument to `add_numbers` is said to be *curried*. There's nothing very fancy here as we really only have defined a new function that calls an existing function. The built-in `functools` module can simplify this process using the `partial` function:

```
from functools import partial  
add_five = partial(add_numbers, 5)
```

# Generators

Having a consistent way to iterate over sequences, like objects in a list or lines in a file, is an important Python feature. This is accomplished by means of the *iterator protocol*, a generic way to make objects iterable. For example, iterating over a dict yields the dict keys:

```
In [171]: some_dict = {'a': 1, 'b': 2, 'c': 3}

In [172]: for key in some_dict:
    ....:     print(key)
a
b
c
```

When you write `for key in some_dict`, the Python interpreter first attempts to create an iterator out of `some_dict`:

```
In [173]: dict_iterator = iter(some_dict)

In [174]: dict_iterator
Out[174]: <dict_keyiterator at 0x7ff191fb32c8>
```

Any iterator is any object that will yield objects to the Python interpreter when used in a context like a for loop. Most methods expecting a list or list-like object will also accept any iterable object. This includes built-in methods such as `min`, `max`, and `sum`, and type constructors like `list` and `tuple`:

```
In [175]: list(dict_iterator)
Out[175]: ['a', 'b', 'c']
```

A *generator* is a concise way to construct a new iterable object. Whereas normal functions execute and return a single value, generators return a sequence of values lazily, pausing after each one until the next one is requested. To create a generator, use the `yield` keyword instead of `return` in a function:

```
def squares(n=10):
    print('Generating squares from 1 to {}'.format(n ** 2))
```

```
for i in range(1, n + 1):
    yield i ** 2
```

When you actually call the generator, no code is immediately executed:

```
In [177]: gen = squares()
```

```
In [178]: gen
```

```
Out[178]: <generator object squares at 0x7ff191fdd620>
```

It is not until you request elements from the generator that it begins executing its code:

```
In [179]: for x in gen:
.....:     print(x, end=' ')
Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

As a less trivial example, suppose we wished to find all unique ways to make change for \$1 (100 cents) using an arbitrary set of coins. You can probably think of various ways to implement this and how to store the unique combinations as you come up with them. One way is to write a generator that yields lists of coins (represented as integers):

```
def make_change(amount, coins=[1, 5, 10, 25], hand=None):
    hand = [] if hand is None else hand
    if amount == 0:
        yield hand
    for coin in coins:
        # ensures we don't give too much change, and combinations
        if coin > amount or (len(hand) > 0 and hand[-1] < coin)
            continue

        for result in make_change(amount - coin, coins=coins,
                                  hand=hand + [coin]):
            yield result
```

The details of the algorithm are not that important (can you think of a shorter way?). Then we can write:

```
In [181]: for way in make_change(100, coins=[10, 25, 50]):
.....:     print(way)
[10, 10, 10, 10, 10, 10, 10, 10, 10]
[25, 25, 10, 10, 10, 10]
```

```
[25, 25, 25, 25]
[50, 10, 10, 10, 10, 10]
[50, 25, 25]
[50, 50]

In [182]: len(list(make_change(100)))    # number of combinations
Out[182]: 242
```

## Generator expresssions

Another even more concise way to make a generator is by using a *generator expression*. This is a generator analogue to list, dict and set comprehensions; to create one, enclose what would otherwise be a list comprehension with parenthesis instead of brackets:

```
In [183]: gen = (x ** 2 for x in range(100))

In [184]: gen
Out[184]: <generator object <genexpr> at 0x7ff191f79620>
```

This is completely equivalent to the following more verbose generator:

```
def _make_gen():
    for x in range(100):
        yield x ** 2
gen = _make_gen()
```

Generator expressions can be used inside any Python function that will accept a generator:

```
In [185]: sum(x ** 2 for x in range(100))
Out[185]: 328350

In [186]: dict((i, i **2) for i in range(5))
Out[186]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

## itertools module

The standard library `itertools` module has a collection of generators for many common data algorithms. For example, `groupby` takes any sequence and a function; this groups consecutive elements in the sequence by return value of

the function. Here's an example:

```
In [187]: import itertools

In [188]: first_letter = lambda x: x[0]

In [189]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']

In [190]: for letter, names in itertools.groupby(names, first_letter):
....:     print(letter, list(names)) # names is a generator object
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

See [Table 3-2](#) for a list of a few other `itertools` functions I've frequently found helpful. You may like to check out the official Python documentation for more on this useful built-in utility module.

Table 3-2. Some useful `itertools` functions

| Function                                   | Description  |
|--|--|
| <code>combinations(iterable, k)</code>     | Generates a sequence of all possible $k$ -tuples of elements in the iterable, ignoring order and without replacement. See also the companion function <code>combinations_with_replacement</code> . |
| <code>permutations(iterable, k)</code>     | Generates a sequence of all possible $k$ -tuples of elements in the iterable, respecting order.  |
| <code>groupby(iterable[, keyfunc])</code>  | Generates <code>(key, sub-iterator)</code> for each unique key   |
| <code>product(*iterables, repeat=1)</code> | Generates the Cartesian product of the input iterables as tuples, similar to a nested <code>for</code> loop.   |

# Files and the operating system

Most of this book uses high-level tools like `pandas.read_csv` to read data files from disk into Python data structures. However, it's important to understand the basics of how to work with files in Python. Fortunately, it's very simple, which is part of why Python is so popular for text and file munging.

To open a file for reading or writing, use the built-in `open` function with either a relative or absolute file path:

```
In [192]: path = 'examples/segismundo.txt'  
In [193]: f = open(path)
```

By default, the file is opened in read-only mode '`r`'. We can then treat the file handle `f` like a list and iterate over the lines like so

```
for line in f:  
    pass
```

The lines come out of the file with the end-of-line (EOL) markers intact, so you'll often see code to get an EOL-free list of lines in a file like

```
In [194]: lines = [x.rstrip() for x in open(path)]
```

```
In [195]: lines  
Out[195]:  
['Sueña el rico en su riqueza,',  
 'que más cuidados le ofrece;',  
 '',  
 'sueña el pobre que padece',  
 'su miseria y su pobreza;',  
 '',  
 'sueña el que a medrar empieza,',  
 'sueña el que afana y pretende,',  
 'sueña el que agravia y ofende,',  
 '',  
 'y en el mundo, en conclusión,',  
 'todos sueñan lo que son,',
```

```
'aunque ninguno lo entiende.',  
[ ]
```

If we had typed `f = open(path, 'w')`, a *new file* at `examples/segismundo.txt` would have been created (be careful!), overwriting any one in its place. There is also the '`x`' file mode which creates a writeable file, but fails if the file path already exists. See below for a list of all valid file read/write modes.

For readable files, some of the most commonly used methods are `read`, `seek`, and `tell`. `read` returns a certain number of characters from the file. What constitutes a “character” is determined by the file’s encoding (e.g. UTF-8) or simply raw bytes if the file is opened in binary mode:

```
In [196]: f.read(10)  
Out[196]: 'Sueña el r'  
  
In [197]: f2 = open(path, 'rb') # Binary mode  
  
In [198]: f2.read(10)  
Out[198]: b'Sue\xc3\xb1a el '
```

The `read` method advances the file handle’s position by the number of bytes read. `tell` gives you the current position:

```
In [199]: f.tell()  
Out[199]: 11  
  
In [200]: f2.tell()  
Out[200]: 10
```

Even though we read 10 characters from the file, the position is 11 because it took that many bytes to decode 10 characters using the default UTF-8 encoding.

Lastly, `seek` changes the file position to the indicates byte in the file:

```
In [201]: f.seek(3)  
Out[201]: 3  
  
In [202]: f.read(1)  
Out[202]: '\n'
```

Table 3-3. Python file modes

| Mode | Description   |
|------|---|
| r    | Read-only mode  |
| w    | Write-only mode. Creates a new file (erasing the data for any file with the same name)  |
| x    | Write-only mode. Creates a new file, but fails if the file path already exists.   |
| a    | Append to existing file (create it if it does not exist)  |
| r+   | Read and write  |
| b    | Add to mode for binary files, that is 'rb' or 'wb'  |
| t    | Text mode for files (automatically decoding bytes to unicode). This is the default if not specified. Add t to other modes to use this, that is 'rt' or 'xt' |

To write text to a file, you can use either the file's `write` or `writelines` methods. For example, we could create a version of `prof_mod.py` with no blank lines like so:

```
In [203]: with open('tmp.txt', 'w') as handle:
.....      handle.writelines(x for x in open(path) if len(x))

In [204]: open('tmp.txt').readlines()
Out[204]:
['Sueña el rico en su riqueza,\n',
 'que más cuidados le ofrece;\n',
 'sueña el pobre que padece\n',
 'su miseria y su pobreza;\n',
 'sueña el que a medrar empieza,\n',
 'sueña el que afana y pretende,\n',
 'sueña el que agravia y ofende,\n',
 'y en el mundo, en conclusión,\n',
 'todos sueñan lo que son,\n',
 'aunque ninguno lo entiende.\n']
```

See [Table 3-4](#) for many of the most commonly-used file methods.

Table 3-4. Important Python file methods or attributes

| Method                    | Description  |
|---------------------------|--|
| <code>read([size])</code> | Return data from file as a string, with optional <code>size</code> argument indicating the number of bytes to read |

|                                  |  |
|----------------------------------|--|
| <code>readlines([size])</code>   | Return list of lines in the file, with optional <code>size</code> argument |
| <code>readlines([size])</code>   | Return list of lines (as strings) in the file                              |
| <code>write(str)</code>          | Write passed string to file.   |
| <code>writelines(strings)</code> | Write passed sequence of strings to the file.                              |
| <code>close()</code>             | Close the handle   |
| <code>flush()</code>             | Flush the internal I/O buffer to disk                                      |
| <code>seek(pos)</code>           | Move to indicated file position (integer).                                 |
| <code>tell()</code>              | Return current file position as integer.                                   |
| <code>closed</code>              | True if the file is closed.  |

# Bytes and Unicode with files

The default behavior for Python files (whether readable or writeable) is *text mode*, which means that you intend to work with Python strings (i.e. Unicode). This contrasts with *binary mode*, which you can obtain by appending `b` onto the file mode. Let's look at the file (which contains non-ASCII characters with UTF-8 encoding) from the previous section:

```
In [207]: with open(path) as f:  
.....:     chars = f.read(10)  
  
In [208]: chars  
Out[208]: 'Sueña el r'
```

UTF-8 is a variable-length unicode encoding, so when I requested some number of characters from the file, Python reads enough bytes (which could be as few as 10 or as many as 40 bytes) from the file to decode that many characters. If I open the file in '`rb`' mode instead, `read` requests exact numbers of bytes:

```
In [209]: with open(path, 'rb') as f:  
.....:     data = f.read(10)  
  
In [210]: data  
Out[210]: b'Sue\xc3\xb1a el '
```

Depending on the text encoding, you may be able to decode the bytes to a `str` object yourself, but only if each of the encoded Unicode characters are fully formed:

```
In [211]: data.decode('utf8')  
Out[211]: 'Sueña el '  
  
In [212]: data[:4].decode('utf8')  
-----  
UnicodeDecodeError                                 Traceback (most recent  
<ipython-input-212-300e0af10bb7> in <module>()  
----> 1 data[:4].decode('utf8')  
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in pos
```

Text mode, combined with the `encoding` option of `open`, provides a convenient way to convert from one Unicode encoding to another:

```
In [213]: sink_path = 'sink.txt'

In [214]: with open(path) as source:
.....:     with open(sink_path, 'xt', encoding='iso-8859-1')
.....:         sink.write(source.read())

In [215]: open(sink_path, encoding='iso-8859-1').read(10)
Out[215]: 'Sueña el r'
```

Beware using `seek` when opening files in any mode other than binary. If the file position falls in the middle of the bytes defining a unicode character, then subsequent reads will result in an error.

```
In [217]: f = open('examples/segismundo.txt')

In [218]: f.read(5)
Out[218]: 'Sueña'

In [219]: f.seek(4)
Out[219]: 4

In [220]: f.read(1)
-----
UnicodeDecodeError                                 Traceback (most recent call last)
<ipython-input-220-7841103e33f5> in <module>()
----> 1 f.read(1)
/home/wesm/conda-envs/book-env/lib/python3.6/codecs.py in decode(self, input, errors)
    319         # decode input (taking the buffer into account)
    320         data = self.buffer + input
--> 321         (result, consumed) = self._buffer_decode(data,
    322             # keep undecoded input until the next call
    323             self.buffer = data[consumed:])
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in position 4: invalid start byte
```

While this may seem like a somewhat esoteric topic, if you find yourself regularly doing data analysis on non-ASCII text data, mastering Python's Unicode functionality will prove valuable. See Python's online documentation for much more.

# Chapter 4. NumPy Basics: Arrays and Vectorized Computation

NumPy, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python. Most computational packages providing scientific functionality use NumPy's array objects as the *lingua franca* for data exchange.

Here are some of the things you'll find in NumPy:

- `ndarray`, an efficient multidimensional array providing fast array-oriented arithmetic operations and flexible *broadcasting* capabilities.
- Mathematical functions for fast operations on entire arrays of data without having to write loops.
- Tools for reading / writing array data to disk and working with memory-mapped files.
- Linear algebra, random number generation, and Fourier transform capabilities.
- A C API for connecting NumPy with libraries written in C, C++, or FORTRAN.

Because NumPy provides an easy-to-use C API, it is straightforward to pass data to external libraries written in a low-level language and also for external libraries to return data to Python as NumPy arrays. This feature has made Python a language of choice for wrapping legacy C / C++ / Fortran codebases and giving them a dynamic and easy-to-use interface.

While NumPy by itself does not provide modeling or scientific functionality, having an understanding of NumPy arrays and array-oriented computing will help you use tools with array-oriented semantics, like pandas, much more effectively. If you're new to Python and looking to get your hands dirty working

with data using pandas, feel free to give this chapter a skim. For more on advanced NumPy features like broadcasting, see [Chapter 12](#).

For most data analysis applications, the main areas of functionality I'll focus on are:

- Fast vectorized array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Data alignment and relational data manipulations for merging and joining together heterogeneous data sets
- Expressing conditional logic as array expressions instead of loops with `if-elif-else` branches
- Group-wise data manipulations (aggregation, transformation, function application). Much more on this in [Chapter 5](#)

While NumPy provides a computational foundation for general numerical data processing, many readers will want to use pandas as the basis for most kinds of statistics or analytics, especially on tabular data. pandas also provides some more domain-specific functionality like time series manipulation, which is not present in NumPy.

#### Note

Array-oriented computing in Python traces its roots back to 1995, when Jim Hugunin created the Numeric library. Over the next 10 years, many scientific programming communities began doing array programming in Python, but the library ecosystem had become fragmented in the early 2000s. In 2005, Travis Oliphant was able to forge the NumPy project from the then Numeric and Numarray projects to bring together the community around a single array computing framework.

One of the reasons NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data. There are a number of reasons for this:

- NumPy internally stores data in a contiguous block of memory, independent of other built-in Python objects. NumPy's library of algorithms written in the C language can operate on this memory without any type checking or other overhead. NumPy arrays also use much less memory than built-in Python sequences.
- NumPy operations perform complex computations on entire arrays without the need for Python for loops.

To give you an idea of the performance difference, consider a NumPy array of 1 million integers, and the equivalent Python list:

```
In [5]: import numpy as np  
  
In [6]: my_arr = np.arange(1000000)  
  
In [7]: my_list = list(range(1000000))
```

Now let's multiple each sequence by 2:

```
In [8]: %time for _ in range(10): my_arr2 = my_arr * 2  
CPU times: user 12 ms, sys: 0 ns, total: 12 ms  
Wall time: 11.1 ms  
  
In [9]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]  
CPU times: user 408 ms, sys: 64 ms, total: 472 ms  
Wall time: 475 ms
```

It's expected for NumPy-based algorithms to be 10 to 100 times faster (or more) than their pure Python counterparts and use significantly less memory.

# The NumPy ndarray: A Multidimensional Array Object

One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large data sets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements:

```
In [10]: import numpy as np  
  
# Generate some random data  
In [11]: data = np.random.randn(2, 3)  
  
In [12]: data  
Out[12]:  
array([-0.2047,  0.4789, -0.5194],  
      [-0.5557,  1.9658,  1.3934])  
  
In [13]: data * 10  
Out[13]:  
array([-2.0471,  4.7894, -5.1944],  
      [-5.5573, 19.6578, 13.9341])  
  
In [14]: data + data  
Out[14]:  
array([-0.4094,  0.9579, -1.0389],  
      [-1.1115,  3.9316,  2.7868])
```

## Note

In this chapter and throughout the book, I use the standard NumPy convention of always using `import numpy as np`. You are, of course, welcome to put `from numpy import *` in your code to avoid having to write `np.`, but I advise you against making a habit of this. The `numpy` is very large and contains a number of functions whose names conflict with built-in Python functions (like `min` and `max`).

An ndarray is a generic multidimensional container for homogeneous data; that

is, all of the elements must be the same type. Every array has a `shape`, a tuple indicating the size of each dimension, and a `dtype`, an object describing the *data type* of the array:

```
In [15]: data.shape  
Out[15]: (2, 3)
```

```
In [16]: data.dtype  
Out[16]: dtype('float64')
```

This chapter will introduce you to the basics of using NumPy arrays, and should be sufficient for following along with the rest of the book. While it's not necessary to have a deep understanding of NumPy for many data analytical applications, becoming proficient in array-oriented programming and thinking is a key step along the way to becoming a scientific Python guru.

#### Note

Whenever you see “array”, “NumPy array”, or “ndarray” in the text, with few exceptions they all refer to the same thing: the `ndarray` object.

# Creating ndarrays

The easiest way to create an array is to use the `array` function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data. For example, a list is a good candidate for conversion:

```
In [17]: data1 = [6, 7.5, 8, 0, 1]
In [18]: arr1 = np.array(data1)
In [19]: arr1
Out[19]: array([ 6.,  7.5,  8.,  0.,  1.])
```

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

```
In [20]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
In [21]: arr2 = np.array(data2)
In [22]: arr2
Out[22]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
In [23]: arr2.ndim
Out[23]: 2
In [24]: arr2.shape
Out[24]: (2, 4)
```

Unless explicitly specified (more on this later), `np.array` tries to infer a good data type for the array that it creates. The data type is stored in a special `dtype` metadata object; for example, in the above two examples we have:

```
In [25]: arr1.dtype
Out[25]: dtype('float64')
In [26]: arr2.dtype
Out[26]: dtype('int64')
```

In addition to `np.array`, there are a number of other functions for creating new arrays. As examples, `zeros` and `ones` create arrays of 0's or 1's, respectively, with a given length or shape. `empty` creates an array without initializing its values to any particular value. To create a higher dimensional array with these methods, pass a tuple for the shape:

```
In [27]: np.zeros(10)
Out[27]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [28]: np.zeros((3, 6))
Out[28]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]))

In [29]: np.empty((2, 3, 2))
Out[29]:
array([[[ 0.,  0.],
         [ 0.,  0.],
         [ 0.,  0.]],
        [[ 0.,  0.],
         [ 0.,  0.],
         [ 0.,  0.]]])
```

#### Caution

It's not safe to assume that `np.empty` will return an array of all zeros. In many cases, as previously shown, it will return uninitialized "garbage" values.

`arange` is an array-valued version of the built-in Python `range` function:

```
In [30]: np.arange(15)
Out[30]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
```

See [Table 4-1](#) for a short list of standard array creation functions. Since NumPy is focused on numerical computing, the data type, if not specified, will in many cases be `float64` (floating point).

Table 4-1. Array creation functions

| Function              | Description  |
|-----------------------|--|
| <code>np.array</code> | Convert input data (list, tuple, array, or other sequence type) to |

|  |  |
|--|--|
| <code>array</code>                             | an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default.   |
| <code>asarray</code>                           | Convert input to ndarray, but do not copy if the input is already an ndarray   |
| <code>arange</code>                            | Like the built-in <code>range</code> but returns an ndarray instead of a list.   |
| <code>ones,</code><br><code>ones_like</code>   | Produce an array of all 1's with the given shape and dtype.<br><code>ones_like</code> takes another array and produces a ones array of the same shape and dtype.                                       |
| <code>zeros,</code><br><code>zeros_like</code> | Like <code>ones</code> and <code>ones_like</code> but producing arrays of 0's instead  |
| <code>empty,</code><br><code>empty_like</code> | Create new arrays by allocating new memory, but do not populate with any values like <code>ones</code> and <code>zeros</code>  |
| <code>full,</code><br><code>full_like</code>   | Produce an array of the given shape and dtype with all values set to the indicated “fill value”. <code>full_like</code> takes another array and produces a a filled array of the same shape and dtype. |
| <code>eye,</code><br><code>identity</code>     | Create a square N x N identity matrix (1's on the diagonal and 0's elsewhere)  |

# Data Types for ndarrays

The *data type* or `dtype` is a special object containing the information (or *metadata*, data about data) the ndarray needs to interpret a chunk of memory as a particular type of data:

```
In [31]: arr1 = np.array([1, 2, 3], dtype=np.float64)  
In [32]: arr2 = np.array([1, 2, 3], dtype=np.int32)  
  
In [33]: arr1.dtype  
Out[33]: dtype('float64')  
  
In [34]: arr2.dtype  
Out[34]: dtype('int32')
```

Dtypes are a source of NumPy's flexibility for interacting with data coming from other systems. In most cases they provide a mapping directly onto an underlying disk or memory representation, which makes it easy to read and write binary streams of data to disk and also to connect to code written in a low-level language like C or Fortran. The numerical dtypes are named the same way: a type name, like `float` or `int`, followed by a number indicating the number of bits per element. A standard double-precision floating point value (what's used under the hood in Python's `float` object) takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as `float64`. See [Table 4-2](#) for a full listing of NumPy's supported data types.

## Note

Don't worry about memorizing the NumPy dtypes, especially if you're a new user. It's often only necessary to care about the general *kind* of data you're dealing with, whether floating point, complex, integer, boolean, string, or general Python object. When you need more control over how data are stored in memory and on disk, especially large data sets, it is good to know that you have control over the storage type.

Table 4-2. NumPy data types

| Type                                    | Type Code          | Description  |
|---|--------------------|--|
| int8, uint8                             | i1,<br>u1          | Signed and unsigned 8-bit (1 byte) integer types   |
| int16, uint16                           | i2,<br>u2          | Signed and unsigned 16-bit integer types   |
| int32, uint32                           | i4,<br>u4          | Signed and unsigned 32-bit integer types   |
| int64, uint64                           | i8,<br>u8          | Signed and unsigned 32-bit integer types   |
| float16                                 | f2                 | Half-precision floating point  |
| float32                                 | f4 or<br>f         | Standard single-precision floating point.<br>Compatible with C float   |
| float64                                 | f8 or<br>d         | Standard double-precision floating point.<br>Compatible with C double and Python <code>float</code> object                                       |
| float128                                | f16<br>or g        | Extended-precision floating point  |
| complex64,<br>complex128,<br>complex256 | c8,<br>c16,<br>c32 | Complex numbers represented by two 32, 64, or 128 floats, respectively   |
| bool                                    | ?                  | Boolean type storing <code>True</code> and <code>False</code> values   |
| object                                  | O                  | Python object type, a value can be any Python object   |
| string_                                 | S                  | Fixed-length ASCII string type (1 byte per character). For example, to create a string dtype with length 10, use ' <code>s10</code> '.           |
| unicode_                                | U                  | Fixed-length unicode type (number of bytes platform specific). Same specification semantics as <code>string_</code> (e.g. ' <code>u10</code> '). |

You can explicitly convert or *cast* an array from one dtype to another using ndarray's `astype` method:

```
In [35]: arr = np.array([1, 2, 3, 4, 5])
```

```
In [36]: arr.dtype
Out[36]: dtype('int64')
```

```
In [37]: float_arr = arr.astype(np.float64)
```

```
In [38]: float_arr.dtype  
Out[38]: dtype('float64')
```

In this example, integers were cast to floating point. If I cast some floating point numbers to be of integer dtype, the decimal part will be truncated:

```
In [39]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
```

```
In [40]: arr  
Out[40]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])
```

```
In [41]: arr.astype(np.int32)  
Out[41]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

Should you have an array of strings representing numbers, you can use `astype` to convert them to numeric form:

```
In [42]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=object)
```

```
In [43]: numeric_strings.astype(float)  
Out[43]: array([ 1.25, -9.6, 42. ])
```

If casting were to fail for some reason (like a string that cannot be converted to `float64`), a `ValueError` will be raised. See that I was a bit lazy and wrote `float` instead of `np.float64`; NumPy aliases the Python types to its own equivalent data dtypes.

You can also use another array's `dtype` attribute:

```
In [44]: int_array = np.arange(10)
```

```
In [45]: calibers = np.array(.22, .270, .357, .380, .44, .50),
```

```
In [46]: int_array.astype(calibers.dtype)  
Out[46]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

There are shorthand type code strings you can also use to refer to a `dtype`:

```
In [47]: empty_uint32 = np.empty(8, dtype='u4')
```

```
In [48]: empty_uint32  
Out[48]:  
array([ 0, 1075314688, 0, 1075707904,  
       1075838976, 0, 1072693248], dtype=uint32)
```

**Note**

Calling `astype` *always* creates a new array (a copy of the data), even if the new dtype is the same as the old dtype.

**Caution**

It's worth keeping in mind that floating point numbers, such as those in `float64` and `float32` arrays, are only capable of approximating fractional quantities. In complex computations, you may accrue some *floating point error*, making comparisons only valid up to a certain number of decimal places.

# Operations between Arrays and Scalars

Arrays are important because they enable you to express batch operations on data without writing any `for` loops. NumPy users call this *vectorization*. Any arithmetic operations between equal-size arrays applies the operation elementwise:

```
In [49]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])
```

```
In [50]: arr
Out[50]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

```
In [51]: arr * arr
Out[51]:
array([[ 1.,   4.,   9.],
       [16.,  25.,  36.]])
```

```
In [52]: arr - arr
Out[52]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Arithmetic operations with scalars are as you would expect, propagating the value to each element:

```
In [53]: 1 / arr
Out[53]:
array([[ 1.      ,  0.5     ,  0.3333],
       [ 0.25    ,  0.2     ,  0.1667]])
```

```
In [54]: arr ** 0.5
Out[54]:
array([[ 1.        ,  1.4142   ,  1.7321  ],
       [ 2.        ,  2.2361   ,  2.4495  ]])
```

Operations between differently sized arrays is called *broadcasting* and will be discussed in more detail in [Chapter 12](#). Having a deep understanding of broadcasting is not necessary for most of this book.

# Basic Indexing and Slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
In [55]: arr = np.arange(10)

In [56]: arr
Out[56]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [57]: arr[5]
Out[57]: 5

In [58]: arr[5:8]
Out[58]: array([5, 6, 7])

In [59]: arr[5:8] = 12

In [60]: arr
Out[60]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

As you can see, if you assign a scalar value to a slice, as in `arr[5:8] = 12`, the value is propagated (or *broadcasted* henceforth) to the entire selection. An important first distinction from Python's built-in lists is that array slices are *views* on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array:

```
In [61]: arr_slice = arr[5:8]

In [62]: arr_slice[1] = 12345

In [63]: arr
Out[63]: array([ 0, 1, 2, 3, 4, 12, 12345])

In [64]: arr_slice[:] = 64

In [65]: arr
Out[65]: array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])
```

If you are new to NumPy, you might be surprised by this, especially if you have

used other array programming languages which copy data more eagerly. As NumPy has been designed to be able to work with very large arrays, you could imagine performance and memory problems if NumPy insisted on always copying data.

#### Caution

If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array; for example `arr[5:8].copy()`.

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
In [66]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
In [67]: arr2d[2]  
Out[67]: array([7, 8, 9])
```

Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements. So these are equivalent:

```
In [68]: arr2d[0][2]  
Out[68]: 3  
  
In [69]: arr2d[0, 2]  
Out[69]: 3
```

See [Figure 4-1](#) for an illustration of indexing on a 2D array.

|        |      | axis 1 |      |      |      |
|--------|------|--------|------|------|------|
|        |      | 0      | 1    | 2    |      |
| axis 0 |      | 0      | 0, 0 | 0, 1 | 0, 2 |
|        |      | 1      | 1, 0 | 1, 1 | 1, 2 |
| 2      | 2, 0 | 2, 1   | 2, 2 |      |      |

Figure 4-1. Indexing elements in a NumPy array

In multidimensional arrays, if you omit later indices, the returned object will be a lower-dimensional ndarray consisting of all the data along the higher dimensions. So in the  $2 \times 2 \times 3$  array `arr3d`

```
In [70]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]])
```

```
In [71]: arr3d
Out[71]:
array([[[ 1,  2,  3],
       [ 4,  5,  6]],
      [[ 7,  8,  9],
       [10, 11, 12]])
```

`arr3d[0]` is a  $2 \times 3$  array:

```
In [72]: arr3d[0]
Out[72]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Both scalar values and arrays can be assigned to `arr3d[0]`:

```
In [73]: old_values = arr3d[0].copy()
```

```
In [74]: arr3d[0] = 42
```

```
In [75]: arr3d
Out[75]:
array([[42, 42, 42],
       [42, 42, 42]],
      [[ 7,  8,  9],
       [10, 11, 12]])
```

```
In [76]: arr3d[0] = old_values
```

```
In [77]: arr3d
Out[77]:
array([[ 1,  2,  3],
       [ 4,  5,  6]],
      [[ 7,  8,  9],
       [10, 11, 12]])
```

Similarly, `arr3d[1, 0]` gives you all of the values whose indices start with `(1, 0)`, forming a 1-dimensional array:

```
In [78]: arr3d[1, 0]
Out[78]: array([7, 8, 9])
```

Note that in all of these cases where subsections of the array have been selected, the returned arrays are views.

## Indexing with slices

Like one-dimensional objects such as Python lists, ndarrays can be sliced using the familiar syntax:

```
In [79]: arr[1:6]
Out[79]: array([ 1,  2,  3,  4, 64])
```

Higher dimensional objects give you more options as you can slice one or more axes and also mix integers. Consider the 2D array above, `arr2d`. Slicing this array is a bit different:

```
In [80]: arr2d
Out[80]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
In [81]: arr2d[:2]
Out[81]:
array([[1, 2, 3],
       [4, 5, 6]])
```

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. You can pass multiple slices just like you can pass multiple indexes:

```
In [82]: arr2d[:2, 1:]
Out[82]:
array([[2, 3],
       [5, 6]])
```

When slicing like this, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower dimensional slices:

```
In [83]: arr2d[1, :2]
Out[83]: array([4, 5])
```

```
In [84]: arr2d[2, :1]
Out[84]: array([7])
```

See [Figure 4-2](#) for an illustration. Note that a colon by itself means to take the entire axis, so you can slice only higher dimensional axes by doing:

```
In [85]: arr2d[:, :1]
Out[85]:
array([[1],
       [4],
```

```
[7])
```

Of course, assigning to a slice expression assigns to the whole selection:

```
In [86]: arr2d[:2, 1:] = 0
```

# Boolean Indexing

Let's consider an example where we have some data in an array and an array of names with duplicates. I'm going to use here the `randn` function in `numpy.random` to generate some random normally distributed data:

```
In [87]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will',  
In [88]: data = np.random.randn(7, 4)  
  
In [89]: names  
Out[89]:  
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],  
      dtype='<U4')  
  
In [90]: data  
Out[90]:  
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],  
      [ 1.0072, -1.2962,  0.275 ,  0.2289],  
      [ 1.3529,  0.8864, -2.0016, -0.3718],  
      [ 1.669 , -0.4386, -0.5397,  0.477 ],  
      [ 3.2489, -1.0212, -0.5771,  0.1241],  
      [ 0.3026,  0.5238,  0.0009,  1.3438],  
      [-0.7135, -0.8312, -2.3702, -1.8608]])
```

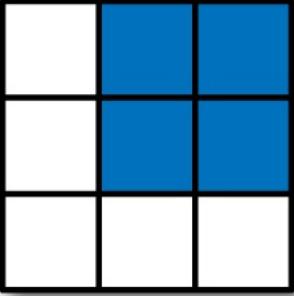
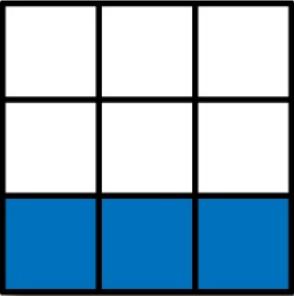
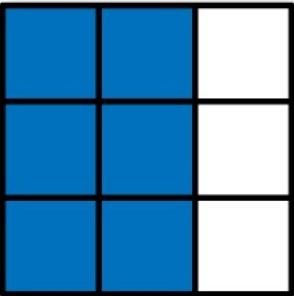
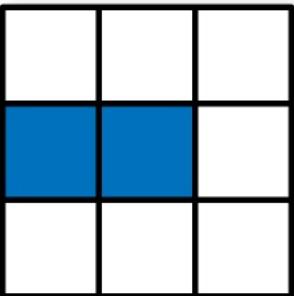
|   | Expression   | Shape                  |
|---|--|------------------------|
|    | <code>arr[:2, 1:]</code>   | (2, 2)                 |
|    | <code>arr[2]</code><br><code>arr[2, :]</code><br><code>arr[2:, :]</code> | (3,)<br>(3,)<br>(1, 3) |
|  | <code>arr[:, :2]</code>  | (3, 2)                 |
|  | <code>arr[1, :2]</code><br><code>arr[1:2, :2]</code>                     | (2,)<br>(1, 2)         |

Figure 4-2. Two-dimensional array slicing

Suppose each name corresponds to a row in the `data` array and we wanted to

select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as `==`) with arrays are also vectorized. Thus, comparing `names` with the string 'Bob' yields a boolean array:

```
In [91]: names == 'Bob'  
Out[91]: array([ True, False, False,  True, False, False, False])
```

This boolean array can be passed when indexing the array:

```
In [92]: data[names == 'Bob']  
Out[92]:  
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],  
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

```
In [93]: data[names == 'Bob', 2:]  
Out[93]:  
array([[ 0.769 ,  1.2464],  
       [-0.5397,  0.477 ]])  
  
In [94]: data[names == 'Bob', 3]  
Out[94]: array([ 1.2464,  0.477 ])
```

To select everything but 'Bob', you can either use `!=` or negate the condition using `~`:

```
In [95]: names != 'Bob'  
Out[95]: array([False,  True,  True, False,  True,  True,  True])  
  
In [96]: data[~(names == 'Bob')]  
Out[96]:  
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],  
       [ 1.3529,  0.8864, -2.0016, -0.3718],  
       [ 3.2489, -1.0212, -0.5771,  0.1241],  
       [ 0.3026,  0.5238,  0.0009,  1.3438],  
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like `&` (and) and `|` (or):

```
In [97]: mask = (names == 'Bob') | (names == 'Will')

In [98]: mask
Out[98]: array([ True, False,  True,  True,  True, False, False])

In [99]: data[mask]
Out[99]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241]])
```

Selecting data from an array by boolean indexing *always* creates a copy of the data, even if the returned array is unchanged.

#### Caution

The Python keywords `and` and `or` do not work with boolean arrays. Use `&` (and) and `|` (or) instead.

Setting values with boolean arrays works in a common-sense way. To set all of the negative values in `data` to 0 we need only do:

```
In [100]: data[data < 0] = 0

In [101]: data
Out[101]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072,  0.        ,  0.275 ,  0.2289],
       [ 1.3529,  0.8864,  0.        ,  0.        ],
       [ 1.669 ,  0.        ,  0.        ,  0.477 ],
       [ 3.2489,  0.        ,  0.        ,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [ 0.        ,  0.        ,  0.        ,  0.        ]])
```

Setting whole rows or columns using a 1D boolean array is also easy:

```
In [102]: data[names != 'Joe'] = 7

In [103]: data
Out[103]:
array([[ 7.        ,  7.        ,  7.        ,  7.        ],
       [ 1.0072,  0.        ,  0.275 ,  0.2289],
       [ 7.        ,  7.        ,  7.        ,  7.        ],
```

```
[ 7.      ,  7.      ,  7.      ,  7.      ],  
[ 7.      ,  7.      ,  7.      ,  7.      ],  
[ 0.3026,  0.5238,  0.0009,  1.3438],  
[ 0.      ,  0.      ,  0.      ,  0.      ]])
```

# Fancy Indexing

*Fancy indexing* is a term adopted by NumPy to describe indexing using integer arrays. Suppose we had a  $8 \times 4$  array:

```
In [104]: arr = np.empty((8, 4))
```

```
In [105]: for i in range(8):  
....:     arr[i] = i
```

```
In [106]: arr  
Out[106]:  
array([[ 0.,  0.,  0.,  0.],  
       [ 1.,  1.,  1.,  1.],  
       [ 2.,  2.,  2.,  2.],  
       [ 3.,  3.,  3.,  3.],  
       [ 4.,  4.,  4.,  4.],  
       [ 5.,  5.,  5.,  5.],  
       [ 6.,  6.,  6.,  6.],  
       [ 7.,  7.,  7.,  7.]])
```

To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

```
In [107]: arr[[4, 3, 0, 6]]  
Out[107]:  
array([[ 4.,  4.,  4.,  4.],  
       [ 3.,  3.,  3.,  3.],  
       [ 0.,  0.,  0.,  0.],  
       [ 6.,  6.,  6.,  6.]])
```

Hopefully this code did what you expected! Using negative indices select rows from the end:

```
In [108]: arr[[-3, -5, -7]]  
Out[108]:  
array([[ 5.,  5.,  5.,  5.],  
       [ 3.,  3.,  3.,  3.],  
       [ 1.,  1.,  1.,  1.]])
```

Passing multiple index arrays does something slightly different; it selects a 1D

array of elements corresponding to each tuple of indices:

```
# more on reshape in Chapter 12
In [109]: arr = np.arange(32).reshape((8, 4))

In [110]: arr
Out[110]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])

In [111]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[111]: array([ 4, 23, 29, 10])
```

Take a moment to understand what happened: the elements  $(1, 0)$ ,  $(5, 3)$ ,  $(7, 1)$ , and  $(2, 2)$  were selected. The behavior of fancy indexing in this case is a bit different from what some users might have expected (myself included), which is the rectangular region formed by selecting a subset of the matrix's rows and columns. Here is one way to get that:

```
In [112]: arr[[1, 5, 7, 2]][[:, [0, 3, 1, 2]]
Out[112]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Another way is to use the `np.ix_` function, which converts two 1D integer arrays to an indexer that selects the square region:

```
In [113]: arr[np.ix_([1, 5, 7, 2], [0, 3, 1, 2])]
Out[113]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array.

# Transposing Arrays and Swapping Axes

Transposing is a special form of reshaping which similarly returns a view on the underlying data without copying anything. Arrays have the `transpose` method and also the special `T` attribute:

```
In [114]: arr = np.arange(15).reshape((3, 5))

In [115]: arr
Out[115]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [116]: arr.T
Out[116]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

When doing matrix computations, you will do this very often, like for example computing the inner matrix product  $x^T x$  using `np.dot`:

```
In [117]: arr = np.random.randn(6, 3)

In [118]: np.dot(arr.T, arr)
Out[118]:
array([[ 9.2291,  0.9394,  4.948 ],
       [ 0.9394,  3.7662, -1.3622],
       [ 4.948 , -1.3622,  4.3437]])
```

For higher dimensional arrays, `transpose` will accept a tuple of axis numbers to permute the axes (for extra mind bending):

```
In [119]: arr = np.arange(16).reshape((2, 2, 4))

In [120]: arr
Out[120]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
```

```
[[ 8,  9, 10, 11],
 [12, 13, 14, 15]])

In [121]: arr.transpose((1, 0, 2))
Out[121]:
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

Simple transposing with `.T` is a special case of swapping axes. `ndarray` has the method `swapaxes` which takes a pair of axis numbers:

```
In [122]: arr
Out[122]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [123]: arr.swapaxes(1, 2)
Out[123]:
array([[ 0,  4],
       [ 1,  5],
       [ 2,  6],
       [ 3,  7]],
      [[ 8, 12],
       [ 9, 13],
       [10, 14],
       [11, 15]]])
```

`swapaxes` similarly returns a view on the data without making a copy.

# Universal Functions: Fast Elementwise Array Functions

A universal function, or *ufunc*, is a function that performs elementwise operations on data in ndarrays. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results.

Many ufuncs are simple elementwise transformations, like `sqrt` or `exp`:

```
In [124]: arr = np.arange(10)

In [125]: np.sqrt(arr)
Out[125]:
array([ 0.        ,  1.        ,  1.4142   ,  1.7321   ,
       2.        ,  2.2361   ,  2.6458   ,
       2.8284   ,  3.        ])

In [126]: np.exp(arr)
Out[126]:
array([ 1.        ,  2.7183   ,  7.3891   ,  20.0855  ,
       54.5982  ,
       148.4132 ,  403.4288 ,  1096.6332 ,  2980.958 ,
       8103.08 ])
```

These are referred to as *unary* ufuncs. Others, such as `add` or `maximum`, take 2 arrays (thus, *binary* ufuncs) and return a single array as the result:

```
In [127]: x = np.random.randn(8)

In [128]: y = np.random.randn(8)

In [129]: x
Out[129]:
array([-0.0119,  1.0048,  1.3272, -0.9193, -1.5491,  0.0222,
       -0.6605])

In [130]: y
Out[130]:
array([ 0.8626, -0.01  ,  0.05  ,  0.6702,  0.853 , -0.9559, -( -2.3042])

In [131]: np.maximum(x, y) # element-wise maximum
```

```
Out[131]:  
array([ 0.8626,  1.0048,  1.3272,  0.6702,  0.853 ,  0.0222,  (  
-0.6605)])
```

While not common, a ufunc can return multiple arrays. `modf` is one example, a vectorized version of the built-in Python `divmod`: it returns the fractional and integral parts of a floating point array:

```
In [132]: arr = np.random.randn(7) * 5  
  
In [133]: np.modf(arr)  
Out[133]:  
(array([-0.2623, -0.0915, -0.663 ,  0.3731,  0.6182,  0.45 ,  
       array([-3., -6., -6.,  5.,  3.,  3.,  5.])))
```

See [Table 4-3](#) and [Table 4-4](#) for a listing of available ufuncs.

Table 4-3. Unary ufuncs

| Function  | Description   |
|---|---|
| <code>abs</code> , <code>fabs</code>  | Compute the absolute value element-wise for integer, floating point, or complex values. Use <code>fabs</code> as a faster alternative for non-complex-valued data |
| <code>sqrt</code>   | Compute the square root of each element. Equivalent to <code>arr ** 0.5</code>  |
| <code>square</code>   | Compute the square of each element. Equivalent to <code>arr ** 2</code>   |
| <code>exp</code>  | Compute the exponent $e^x$ of each element  |
| <code>log</code> , <code>log10</code> , <code>log2</code> ,<br><code>log1p</code> | Natural logarithm (base $e$ ), log base 10, log base 2, and <code>log(1 + x)</code> , respectively  |
| <code>sign</code>   | Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)  |
| <code>ceil</code>   | Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element  |
| <code>floor</code>  | Compute the floor of each element, i.e. the largest integer less than or equal to each element  |
| <code>rint</code>   | Round elements to the nearest integer, preserving the <code>dtype</code>  |
|   | Return fractional and integral parts of array as  |

|  |  |
|--|--|
| <code>modf</code>  | separate array   |
| <code>isnan</code>   | Return boolean array indicating whether each value is <code>NaN</code> (Not a Number)  |
| <code>isfinite</code> , <code>isinf</code>   | Return boolean array indicating whether each element is finite ( <code>non-inf</code> , <code>non-NaN</code> ) or infinite, respectively |
| <code>cos</code> , <code>cosh</code> , <code>sin</code> ,<br><code>sinh</code> , <code>tan</code> , <code>tanh</code>                      | Regular and hyperbolic trigonometric functions   |
| <code>arccos</code> , <code>arccosh</code> ,<br><code>arcsin</code> , <code>arcsinh</code> ,<br><code>arctan</code> , <code>arctanh</code> | Inverse trigonometric functions  |
| <code>logical_not</code>   | Compute truth value of <code>not x</code> element-wise.<br>Equivalent to <code>-arr</code> .   |

Table 4-4. Binary universal functions

| Function  | Description  |
|---|--|
| <code>add</code>  | Add corresponding elements in arrays   |
| <code>subtract</code>   | Subtract elements in second array from first array   |
| <code>multiply</code>   | Multiply array elements  |
| <code>divide</code> , <code>floor_divide</code>   | Divide or floor divide (truncating the remainder)  |
| <code>power</code>  | Raise elements in first array to powers indicated in second array  |
| <code>maximum</code> , <code>fmax</code>  | Element-wise maximum. <code>fmax</code> ignores <code>NaN</code>   |
| <code>minimum</code> , <code>fmin</code>  | Element-wise minimum. <code>fmin</code> ignores <code>NaN</code>   |
| <code>mod</code>  | Element-wise modulus (remainder of division)   |
| <code>copysign</code>   | Copy sign of values in second argument to values in first argument   |
| <code>greater</code> , <code>greater_equal</code> ,<br><code>less</code> , <code>less_equal</code> , <code>equal</code> ,<br><code>not_equal</code> | Perform element-wise comparison, yielding boolean array. Equivalent to infix operators <code>&gt;</code> , <code>&gt;=</code> , <code>&lt;</code> , <code>&lt;=</code> , <code>==</code> , <code>!=</code> |
| <code>logical_and</code> ,<br><code>logical_or</code> , <code>logical_xor</code>  | Compute element-wise truth value of logical operation. Equivalent to infix operators <code>&amp;</code> , <code> </code> , <code>^</code>  |

# Loop-free programming with arrays

Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops. This practice of replacing explicit loops with array expressions is commonly referred to as *vectorization*. In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations. Later, in [Chapter 12](#), I will explain *broadcasting*, a powerful method for vectorizing computations.

As a simple example, suppose we wished to evaluate the function `sqrt(x^2 + y^2)` across a regular grid of values. The `np.meshgrid` function takes two 1D arrays and produces two 2D matrices corresponding to all pairs of `(x, y)` in the two arrays:

```
In [134]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points

In [135]: xs, ys = np.meshgrid(points, points)

In [136]: ys
Out[136]:
array([[-5. , -5. , -5. , ..., -5. , -5. , -5. ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

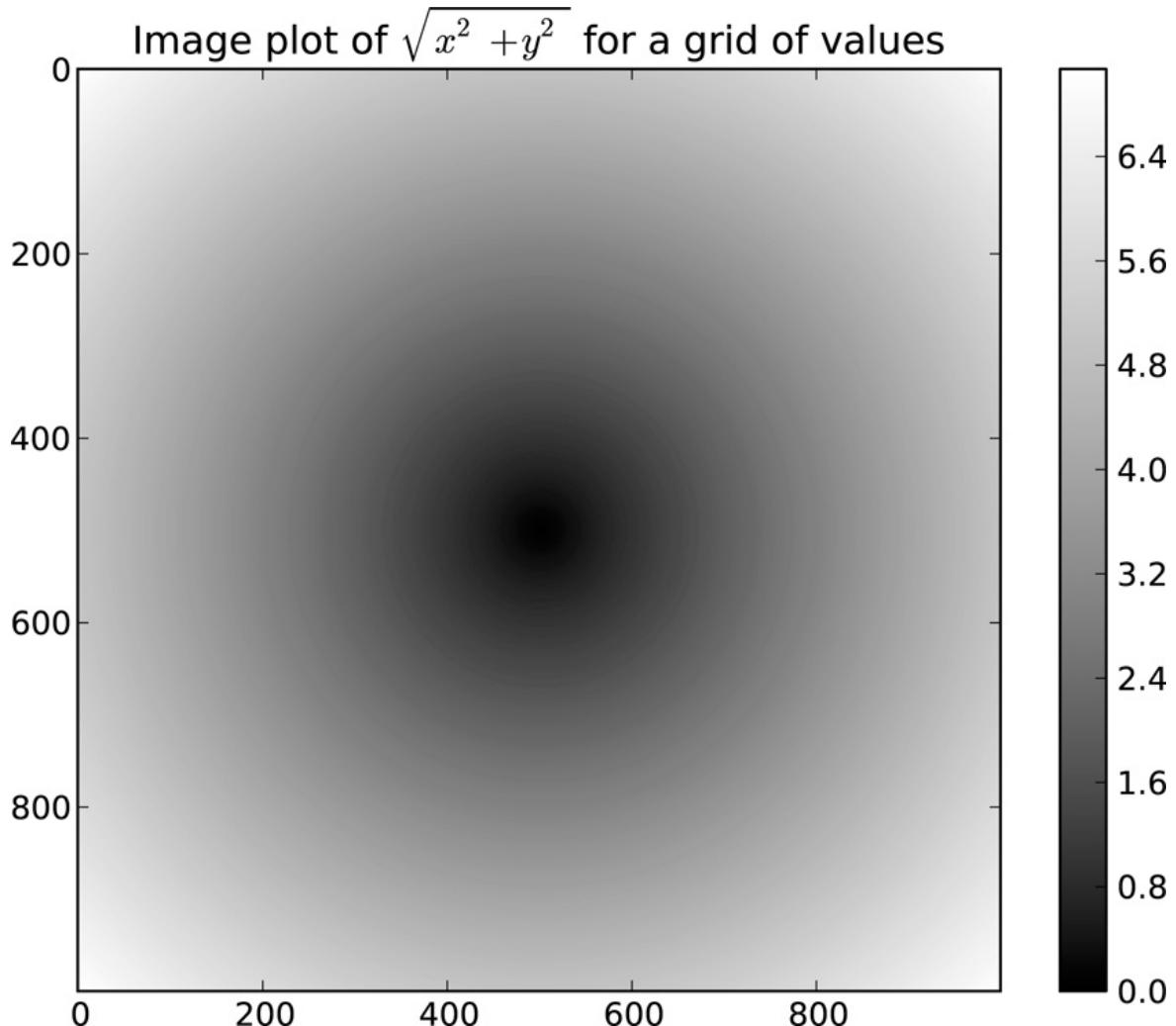
Now, evaluating the function is a simple matter of writing the same expression you would write with two points:

```
[ 7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569,
[ 7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499,
...,
[ 7.0499, 7.0428, 7.0357, ..., 7.0286, 7.0357, 7.0428,
[ 7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499,
[ 7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569,
```

```
In [140]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[140]: <matplotlib.colorbar.Colorbar at 0x7f7d89cdc2b0>
```

```
In [141]: plt.title("Image plot of  $\sqrt{x^2 + y^2}$  for a grid")
Out[141]: <matplotlib.text.Text at 0x7f7d89d40588>
```

See [Figure 4-3](#). Here I used the matplotlib function `imshow` to create an image plot from a 2D array of function values.



**Figure 4-3.** Plot of function evaluated on grid

# Expressing Conditional Logic as Array Operations

The `numpy.where` function is a vectorized version of the ternary expression `x if condition else y`. Suppose we had a boolean array and two arrays of values:

```
In [142]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
```

```
In [143]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
```

```
In [144]: cond = np.array([True, False, True, True, False])
```

Suppose we wanted to take a value from `xarr` whenever the corresponding value in `cond` is `True` otherwise take the value from `yarr`. A list comprehension doing this might look like:

```
In [145]: result = [(x if c else y)
....:             for x, y, c in zip(xarr, yarr, cond)]
```

```
In [146]: result
Out[146]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999998]
```

This has multiple problems. First, it will not be very fast for large arrays (because all the work is being done in pure Python). Secondly, it will not work with multidimensional arrays. With `np.where` you can write this very concisely:

```
In [147]: result = np.where(cond, xarr, yarr)
```

```
In [148]: result
Out[148]: array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```

The second and third arguments to `np.where` don't need to be arrays; one or both of them can be scalars. A typical use of `where` in data analysis is to produce a new array of values based on another array. Suppose you had a matrix of randomly generated data and you wanted to replace all positive values with 2 and all negative values with -2. This is very easy to do with

```

np.where:

In [149]: arr = np.random.randn(4, 4)

In [150]: arr
Out[150]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 0.2229,  0.0513, -1.1577,  0.8167],
       [ 0.4336,  1.0107,  1.8249, -0.9975],
       [ 0.8506, -0.1316,  0.9124,  0.1882]]))

In [151]: np.where(arr > 0, 2, -2)
Out[151]:
array([-2, -2, -2, -2],
      [ 2,  2, -2,  2],
      [ 2,  2,  2, -2],
      [ 2, -2,  2,  2]])

In [152]: np.where(arr > 0, 2, arr) # set only positive values
Out[152]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 2.,     2.,     -1.1577,  2.     ],
       [ 2.,     2.,     2.,     -0.9975],
       [ 2.,     -0.1316,  2.,     2.     ]])

```

The arrays passed to where can be more than just equal sizes array or scalars.

With some cleverness you can use where to express more complicated logic; consider this example where I have two boolean arrays, `cond1` and `cond2`, and wish to assign a different value for each of the 4 possible pairs of boolean values:

```

result = []
for i in range(n):
    if cond1[i] and cond2[i]:
        result.append(0)
    elif cond1[i]:
        result.append(1)
    elif cond2[i]:
        result.append(2)
    else:
        result.append(3)

```

While perhaps not immediately obvious, this `for` loop can be converted into a nested `where` expression:

```
np.where(cond1 & cond2, 0,  
        np.where(cond1, 1,  
                 np.where(cond2, 2, 3)))
```

# Mathematical and Statistical Methods

A set of mathematical functions which compute statistics about an entire array or about the data along an axis are accessible as array methods. Aggregations (often called *reductions*) like `sum`, `mean`, and standard deviation `std` can either be used by calling the array instance method or using the top level NumPy function:

```
In [153]: arr = np.random.randn(5, 4) # normally-distributed random numbers
In [154]: arr.mean()
Out[154]: 0.19607051119998253
In [155]: np.mean(arr)
Out[155]: 0.19607051119998253
In [156]: arr.sum()
Out[156]: 3.9214102239996507
```

Functions like `mean` and `sum` take an optional `axis` argument which computes the statistic over the given axis, resulting in an array with one fewer dimension:

```
In [157]: arr.mean(axis=1)
Out[157]: array([ 1.022 ,  0.1875, -0.502 , -0.0881,  0.3611])
In [158]: arr.sum(0)
Out[158]: array([ 3.1693, -2.6345,  2.2381,  1.1486])
```

Other methods like `cumsum` and `cumprod` do not aggregate, instead producing an array of the intermediate results:

```
In [159]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
In [160]: arr.cumsum()
Out[160]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

In multidimensional arrays, accumulation functions like `cumsum` return an array of the same size, but with the partial aggregates computed along the indicated axis according to each lower-dimensional slice:

```
In [161]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [162]: arr.cumsum(0)
Out[162]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])

In [163]: arr.cumprod(1)
Out[163]:
array([[ 0,  0,  0],
       [ 3, 12, 60],
       [ 6, 42, 336]])
```

See [Table 4-5](#) for a full listing. We'll see many examples of these methods in action in later chapters.

Table 4-5. Basic array statistical methods

| Method         | Description   |
|----------------|---|
| sum            | Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.   |
| mean           | Arithmetic mean. Zero-length arrays have <code>NaN</code> mean.   |
| std, var       | Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator <code>n</code> ). |
| min, max       | Minimum and maximum.  |
| argmin, argmax | Indices of minimum and maximum elements, respectively.  |
| cumsum         | Cumulative sum of elements starting from 0  |
| cumprod        | Cumulative product of elements starting from 1  |

# Methods for Boolean Arrays

Boolean values are coerced to 1 (`True`) and 0 (`False`) in the above methods. Thus, `sum` is often used as a means of counting `True` values in a boolean array:

```
In [164]: arr = np.random.randn(100)
```

```
In [165]: (arr > 0).sum() # Number of positive values
Out[165]: 42
```

There are two additional methods, `any` and `all`, useful especially for boolean arrays. `any` tests whether one or more values in an array is `True`, while `all` checks if every value is `True`:

```
In [166]: bools = np.array([False, False, True, False])
```

```
In [167]: bools.any()
Out[167]: True
```

```
In [168]: bools.all()
Out[168]: False
```

These methods also work with non-boolean arrays, where non-zero elements evaluate to `True`.

# Sorting

Like Python's built-in list type, NumPy arrays can be sorted in-place using the `sort` method:

```
In [169]: arr = np.random.randn(6)

In [170]: arr
Out[170]: array([ 0.6095, -0.4938,  1.24   , -0.1357,  1.43   ,
                  ...])

In [171]: arr.sort()

In [172]: arr
Out[172]: array([-0.8469, -0.4938, -0.1357,  0.6095,  1.24   ,
                  ...])
```

Multidimensional arrays can have each 1D section of values sorted in-place along an axis by passing the axis number to `sort`:

```
In [173]: arr = np.random.randn(5, 3)

In [174]: arr
Out[174]:
array([[ 0.6033,  1.2636, -0.2555],
       [-0.4457,  0.4684, -0.9616],
       [-1.8245,  0.6254,  1.0229],
       [ 1.1074,  0.0909, -0.3501],
       [ 0.218 , -0.8948, -1.7415]])

In [175]: arr.sort(1)

In [176]: arr
Out[176]:
array([[ -0.2555,   0.6033,   1.2636],
       [ -0.9616,  -0.4457,   0.4684],
       [ -1.8245,   0.6254,   1.0229],
       [ -0.3501,   0.0909,   1.1074],
       [ -1.7415,  -0.8948,   0.218 ]])
```

The top level method `np.sort` returns a sorted copy of an array instead of modifying the array in place. A quick-and-dirty way to compute the quantiles of an array is to sort it and select the value at a particular rank:

```
In [177]: large_arr = np.random.randn(1000)

In [178]: large_arr.sort()

In [179]: large_arr[int(0.05 * len(large_arr))] # 5% quantile
Out[179]: -1.5311513550102103
```

For more details on using NumPy's sorting methods, and more advanced techniques like indirect sorts, see [Chapter 12](#). Several other kinds of data manipulations related to sorting (for example, sorting a table of data by one or more columns) are also to be found in pandas.

# Unique and Other Set Logic

NumPy has some basic set operations for one-dimensional ndarrays. A commonly used one is `np.unique`, which returns the sorted unique values in an array:

```
In [180]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will']

In [181]: np.unique(names)
Out[181]:
array(['Bob', 'Joe', 'Will'],
      dtype='<U4')

In [182]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [183]: np.unique(ints)
Out[183]: array([1, 2, 3, 4])
```

Contrast `np.unique` with the pure Python alternative:

```
In [184]: sorted(set(names))
Out[184]: ['Bob', 'Joe', 'Will']
```

Another function, `np.in1d`, tests membership of the values in one array in another, returning a boolean array:

```
In [185]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [186]: np.in1d(values, [2, 3, 6])
Out[186]: array([ True, False, False,  True,  True, False,  True])
```

See [Table 4-6](#) for a listing of set functions in NumPy.

Table 4-6. Array set operations

| Method                         | Description  |
|--------------------------------|--|
| <code>unique(x)</code>         | Compute the sorted, unique elements in <code>x</code>                    |
| <code>intersect1d(x, y)</code> | Compute the sorted, common elements in <code>x</code> and <code>y</code> |
| <code>union1d(x, y)</code>     | Compute the sorted union of elements                                     |

|                              |  |
|------------------------------|--|
| <code>in1d(x, y)</code>      | Compute a boolean array indicating whether each element of <code>x</code> is contained in <code>y</code> |
| <code>setdiff1d(x, y)</code> | Set difference, elements in <code>x</code> that are not in <code>y</code>                                |
| <code>setxor1d(x, y)</code>  | Set symmetric differences; elements that are in either of the arrays, but not both                       |

# File Input and Output with Arrays

NumPy is able to save and load data to and from disk either in text or binary format. In this section I only discuss NumPy's built-in binary format, since most users will prefer pandas and other tools for loading text or tabular data (see [Chapter 6](#) for much more).

`np.save` and `np.load` are the two workhorse functions for efficiently saving and loading array data on disk. Arrays are saved by default in an uncompressed raw binary format with file extension `.npy`.

```
In [187]: arr = np.arange(10)  
In [188]: np.save('some_array', arr)
```

If the file path does not already end in `.npy`, the extension will be appended. The array on disk can then be loaded using `np.load`:

```
In [189]: np.load('some_array.npy')  
Out[189]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

You save multiple arrays in a zip archive using `np.savez` and passing the arrays as keyword arguments:

```
In [190]: np.savez('array_archive.npz', a=arr, b=arr)
```

When loading an `.npz` file, you get back a dict-like object which loads the individual arrays lazily:

```
In [191]: arch = np.load('array_archive.npz')  
In [192]: arch['b']  
Out[192]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

If your data compresses well, you may wish to use `numpy.savez_compressed` instead:

```
In [193]: np.savez('arrays_compressed.npz', a=arr, b=arr)
```

# Linear Algebra

Linear algebra, like matrix multiplication, decompositions, determinants, and other square matrix math, is an important part of any array library. Unlike some languages like MATLAB, multiplying two two-dimensional arrays with `*` is an element-wise product instead of a matrix dot product. As such, there is a function `dot`, both an array method, and a function in the `numpy` namespace, for matrix multiplication:

```
In [197]: x = np.array([[1., 2., 3.], [4., 5., 6.]])  
In [198]: y = np.array([[6., 23.], [-1, 7], [8, 9]])  
  
In [199]: x  
Out[199]:  
array([[ 1.,  2.,  3.],  
       [ 4.,  5.,  6.]])  
  
In [200]: y  
Out[200]:  
array([[ 6., 23.],  
      [-1.,  7.],  
      [ 8.,  9.]])  
  
In [201]: x.dot(y) # equivalently np.dot(x, y)  
Out[201]:  
array([[ 28., 64.],  
       [ 67., 181.]])
```

A matrix product between a 2D array and a suitably sized 1D array results in a 1D array:

```
In [202]: np.dot(x, np.ones(3))  
Out[202]: array([ 6., 15.])
```

The `@` symbol (as of Python 3.5) also works as an infix operator that performs matrix multiplication:

```
In [203]: x @ np.ones(3)  
Out[203]: array([ 6., 15.])
```

`numpy.linalg` has a standard set of matrix decompositions and things like inverse and determinant. These are implemented under the hood using the same industry-standard Fortran libraries used in other languages like MATLAB and R, such as like BLAS, LAPACK, or possibly (depending on your NumPy build) the proprietary Intel MKL (Math Kernel Library):

```
In [204]: from numpy.linalg import inv, qr
In [205]: X = np.random.randn(5, 5)
In [206]: mat = X.T.dot(X)

In [207]: inv(mat)
Out[207]:
array([[ 933.1189,   871.8258, -1417.6902, -1460.4005,  1782.1,
       [ 871.8258,   815.3929, -1325.9965, -1365.9242,  1666.1,
      [-1417.6902, -1325.9965,  2158.4424,  2222.0191, -2711.6,
      [-1460.4005, -1365.9242,  2222.0191,  2289.0575, -2793.4,
      [ 1782.1391,  1666.9347, -2711.6822, -2793.422 ,  3409.4

In [208]: mat.dot(inv(mat))
Out[208]:
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1., -0., -0., -0.],
       [-0., -0.,  1.,  0., -0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0., -0., -0.,  1.]))

In [209]: q, r = qr(mat)

In [210]: r
Out[210]:
array([[-1.6914,   4.38   ,   0.1757,   0.4075,  -0.7838],
       [ 0.        , -2.6436,   0.1939,  -3.072 ,  -1.0702],
       [ 0.        ,   0.      , -0.8138,   1.5414,   0.6155],
       [ 0.        ,   0.      ,   0.      , -2.6445,  -2.1669],
       [ 0.        ,   0.      ,   0.      ,   0.      ,   0.0002]])
```

See [Table 4-7](#) for a list of some of the most commonly-used linear algebra functions.

Table 4-7. Commonly-used `numpy.linalg` functions

| Function             | Description  |
|----------------------|--|
| <code>np.diag</code> | Return the diagonal (or off-diagonal) elements of a square matrix as |

|       |   |
|-------|---|
| diag  | a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal |
| dot   | Matrix multiplication   |
| trace | Compute the sum of the diagonal elements  |
| det   | Compute the matrix determinant  |
| eig   | Compute the eigenvalues and eigenvectors of a square matrix                           |
| inv   | Compute the inverse of a square matrix  |
| pinv  | Compute the Moore-Penrose pseudo-inverse inverse of a matrix                          |
| qr    | Compute the QR decomposition  |
| svd   | Compute the singular value decomposition (SVD)  |
| solve | Solve the linear system $Ax = b$ for $x$ , where $A$ is a square matrix               |
| lstsq | Compute the least-squares solution to $Ax = b$  |

# Pseudorandom Number Generation

The `numpy.random` module supplements the built-in Python `random` with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions. For example, you can get a 4 by 4 array of samples from the standard normal distribution using `normal`:

```
In [211]: samples = np.random.normal(size=(4, 4))
```

```
In [212]: samples
Out[212]:
array([[ 0.5732,  0.1933,  0.4429,  1.2796],
       [ 0.575 ,  0.4339, -0.7658, -1.237 ],
       [-0.5367,  1.8545, -0.92 , -0.1082],
       [ 0.1525,  0.9435, -1.0953, -0.144 ]])
```

Python's built-in `random` module, by contrast, only samples one value at a time. As you can see from this benchmark, `numpy.random` is well over an order of magnitude faster for generating very large samples:

```
In [213]: from random import normalvariate
```

```
In [214]: N = 1000000
```

```
In [215]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
710 ms +- 17.5 ms per loop (mean +- std. dev. of 7 runs, 1 loop)
```

```
In [216]: %timeit np.random.normal(size=N)
41.2 ms +- 351 us per loop (mean +- std. dev. of 7 runs, 10 loops)
```

We say that these are *pseudorandom* numbers because they are generated by an algorithm with deterministic behavior based on the *seed* of the random number generator. You can change NumPy's random number generation seed using `np.random.seed`:

```
In [217]: np.random.seed(1234)
```

See [Table 4-8](#) for a partial list of functions available in `numpy.random`. I'll give some examples of leveraging these functions' ability to generate large

arrays of samples all at once in the next section.

Table 4-8. Partial list of numpy.random functions

| Function    | Description  |
|-------------|--|
| seed        | Seed the random number generator   |
| permutation | Return a random permutation of a sequence, or return a permuted range                                |
| shuffle     | Randomly permute a sequence in place   |
| rand        | Draw samples from a uniform distribution   |
| randint     | Draw random integers from a given low-to-high range  |
| randn       | Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface) |
| binomial    | Draw samples from a binomial distribution  |
| normal      | Draw samples from a normal (Gaussian) distribution   |
| beta        | Draw samples from a beta distribution  |
| chisquare   | Draw samples from a chi-square distribution  |
| gamma       | Draw samples from a gamma distribution   |
| uniform     | Draw samples from a uniform [0, 1) distribution  |

# Example: Random Walks

An illustrative application of utilizing array operations is in the simulation of random walks. Let's first consider a simple random walk starting at 0 with steps of 1 and -1 occurring with equal probability. A pure Python way to implement a single random walk with 1,000 steps using the built-in `random` module:

```
import random
position = 0
walk = [position]
steps = 1000
for i in xrange(steps):
    step = 1 if random.randint(0, 1) else -1
    position += step
    walk.append(position)
```

See [Figure 4-4](#) for an example plot of the first 100 values on one of these random walks.



Figure 4-4. A simple random walk

You might make the observation that `walk` is simply the cumulative sum of the random steps and could be evaluated as an array expression. Thus, I use the `np.random` module to draw 1,000 coin flips at once, set these to 1 and -1, and compute the cumulative sum:

```
In [219]: nsteps = 1000  
In [220]: draws = np.random.randint(0, 2, size=nsteps)  
In [221]: steps = np.where(draws > 0, 1, -1)  
In [222]: walk = steps.cumsum()
```

From this we can begin to extract statistics like the minimum and maximum value along the walk's trajectory:

```
In [223]: walk.min()  
Out[223]: -3  
  
In [224]: walk.max()  
Out[224]: 31
```

A more complicated statistic is the *first crossing time*, the step at which the random walk reaches a particular value. Here we might want to know how long it took the random walk to get at least 10 steps away from the origin 0 in either direction. `np.abs(walk) >= 10` gives us a boolean array indicating where the walk has reached or exceeded 10, but we want the index of the *first* 10 or -10. Turns out this can be computed using `argmax`, which returns the first index of the maximum value in the boolean array (`True` is the maximum value):

```
In [225]: (np.abs(walk) >= 10).argmax()  
Out[225]: 37
```

Note that using `argmax` here is not always efficient because it always makes a full scan of the array. In this special case once a `True` is observed we know it to be the maximum value.

# Simulating Many Random Walks at Once

If your goal was to simulate many random walks, say 5,000 of them, you can generate all of the random walks with minor modifications to the above code. The `numpy.random` functions if passed a 2-tuple will generate a 2D array of draws, and we can compute the cumulative sum across the rows to compute all 5,000 random walks in one shot:

```
In [226]: nwalks = 5000  
  
In [227]: nsteps = 1000  
  
In [228]: draws = np.random.randint(0, 2, size=(nwalks, nsteps))  
  
In [229]: steps = np.where(draws > 0, 1, -1)  
  
In [230]: walks = steps.cumsum(1)  
  
In [231]: walks  
Out[231]:  
array([[ 1,  0,  1, ...,  8,  7,  8],  
       [ 1,  0, -1, ..., 34, 33, 32],  
       [ 1,  0, -1, ...,  4,  5,  4],  
       ...,  
       [ 1,  2,  1, ..., 24, 25, 26],  
       [ 1,  2,  3, ..., 14, 13, 14],  
       [-1, -2, -3, ..., -24, -23, -22]])
```

Now, we can compute the maximum and minimum values obtained over all of the walks:

```
In [232]: walks.max()  
Out[232]: 138  
  
In [233]: walks.min()  
Out[233]: -133
```

Out of these walks, let's compute the minimum crossing time to 30 or -30. This is slightly tricky because not all 5,000 of them reach 30. We can check this using the `any` method:

```
In [234]: hits30 = (np.abs(walks) >= 30).any(1)

In [235]: hits30
Out[235]: array([False,  True, False, ..., False,  True, False])

In [236]: hits30.sum() # Number that hit 30 or -30
Out[236]: 3410
```

We can use this boolean array to select out the rows of `walks` that actually cross the absolute 30 level and call `argmax` across axis 1 to get the crossing times:

```
In [237]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax()
In [238]: crossing_times.mean()
Out[238]: 498.88973607038122
```

Feel free to experiment with other distributions for the steps other than equal sized coin flips. You need only use a different random number generation function, like `normal` to generate normally distributed steps with some mean and standard deviation:

# Chapter 5. Getting Started with pandas

pandas will be a major tool of interest throughout much of the rest of the book. It contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python. pandas is often used in tandem with numerical computing tools like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and data visualization libraries like matplotlib. pandas adopts significant parts of NumPy's idiomatic style of array-based computing, especially array-based functions and a preference for data processing without for-loops.

Since becoming an open source project in 2010, pandas has matured into a quite large library that's applicable in a broad set of real world use cases. The developer community has grown to over 600 distinct contributors, who've been helping us build the project as they've used it to solve their day-to-day data problems.

Throughout the rest of the book, I use the following import convention for pandas:

```
In [1]: import pandas as pd
```

Thus, whenever you see `pd.` in code, it's referring to pandas. You may also find it easier to import Series and DataFrame into the local namespace since they are so frequently used:

```
In [2]: from pandas import Series, DataFrame
```

# Introduction to pandas Data Structures

To get started with pandas, you will need to get comfortable with its two workhorse data structures: *Series* and *DataFrame*. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

# Series

A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its *index*. The simplest Series is formed from only an array of data:

```
In [11]: obj = pd.Series([4, 7, -5, 3])  
  
In [12]: obj  
Out[12]:  
0    4  
1    7  
2   -5  
3    3  
dtype: int64
```

The string representation of a Series displayed interactively shows the index on the left and the values on the right. Since we did not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created. You can get the array representation and index object of the Series via its values and index attributes, respectively:

```
In [13]: obj.values  
Out[13]: array([ 4,  7, -5,  3])  
  
In [14]: obj.index # like range(4)  
Out[14]: RangeIndex(start=0, stop=4, step=1)
```

Often it will be desirable to create a Series with an index identifying each data point with a label:

```
In [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a',  
In [16]: obj2  
Out[16]:  
d    4  
b    7  
a   -5  
c    3  
dtype: int64
```

```
In [17]: obj2.index  
Out[17]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Compared with NumPy arrays, you can use values in the index when selecting single values or a set of values:

```
In [18]: obj2['a']  
Out[18]: -5
```

```
In [19]: obj2['d'] = 6
```

```
In [20]: obj2[['c', 'a', 'd']]  
Out[20]:  
c    3  
a   -5  
d    6  
dtype: int64
```

Using NumPy functions or NumPy-like operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

```
In [21]: obj2[obj2 > 0]  
Out[21]:  
d    6  
b    7  
c    3  
dtype: int64
```

```
In [22]: obj2 * 2  
Out[22]:  
d    12  
b    14  
a   -10  
c     6  
dtype: int64
```

```
In [23]: np.exp(obj2)  
Out[23]:  
d      403.428793  
b     1096.633158  
a      0.006738  
c     20.085537  
dtype: float64
```

Another way to think about a Series is as a fixed-length, ordered dict, as it is a mapping of index values to data values. It can be used in many contexts where you might use a dict:

```
In [24]: 'b' in obj2  
Out[24]: True
```

```
In [25]: 'e' in obj2  
Out[25]: False
```

Should you have data contained in a Python dict, you can create a Series from it by passing the dict:

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000  
In [27]: obj3 = pd.Series(sdata)  
  
In [28]: obj3  
Out[28]:  
Ohio      35000  
Oregon    16000  
Texas     71000  
Utah      5000  
dtype: int64
```

When only passing a dict, the index in the resulting Series will have the dict's keys in sorted order. You can override this by passing the dict keys in the order you want them to appear in the resulting Series:

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']  
In [30]: obj4 = pd.Series(sdata, index=states)  
  
In [31]: obj4  
Out[31]:  
California      NaN  
Ohio          35000.0  
Oregon         16000.0  
Texas          71000.0  
dtype: float64
```

Here, 3 values found in `sdata` were placed in the appropriate locations, but since no value for '`California`' was found, it appears as `NaN` (not a number) which is considered in pandas to mark missing or `NA` values. I will use the

terms “missing” or “NA” to refer to missing data. The `isnull` and `notnull` functions in pandas should be used to detect missing data:

```
In [32]: pd.isnull(obj4)
Out[32]:
California      True
Ohio            False
Oregon          False
Texas           False
dtype: bool
```

```
In [33]: pd.notnull(obj4)
Out[33]:
California     False
Ohio            True
Oregon          True
Texas           True
dtype: bool
```

Series also has these as instance methods:

```
In [34]: obj4.isnull()
Out[34]:
California      True
Ohio            False
Oregon          False
Texas           False
dtype: bool
```

I discuss working with missing data in more detail in [Chapter 7](#).

A useful Series feature for many applications is that it automatically aligns differently-indexed data in arithmetic operations:

```
In [35]: obj3
Out[35]:
Ohio      35000
Oregon    16000
Texas     71000
Utah      5000
dtype: int64
```

```
In [36]: obj4
Out[36]:
California      NaN
```

```
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
In [37]: obj3 + obj4
Out[37]:
California      NaN
Ohio            70000.0
Oregon          32000.0
Texas           142000.0
Utah            NaN
dtype: float64
```

Data alignment features are addressed as a separate topic. If you have experience with databases, you can think about this as being similar to a join operation.

Both the Series object itself and its index have a `name` attribute, which integrates with other key areas of pandas functionality:

```
In [38]: obj4.name = 'population'

In [39]: obj4.index.name = 'state'

In [40]: obj4
Out[40]:
state
California      NaN
Ohio            35000.0
Oregon          16000.0
Texas           71000.0
Name: population, dtype: float64
```

A Series's index can be altered in place by assignment:

```
In [41]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [42]: obj
Out[42]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

# DataFrame

A DataFrame represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series all sharing the same index. Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays. The exact details of DataFrame's internals are outside the scope of this book.

## Note

While a DataFrame is physically two-dimensional, you can use it to represent higher-dimensional data in a tabular format using hierarchical indexing, a subject of a later chapter and an ingredient in some of the more advanced data-handling features in pandas.

There are many ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = pd.DataFrame(data)
```

The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order:

```
In [44]: frame
Out[44]:
   pop    state  year
0  1.5      Ohio  2000
1  1.7      Ohio  2001
2  3.6      Ohio  2002
3  2.4    Nevada  2001
4  2.9    Nevada  2002
```

For large DataFrames, the `head` method is useful to get see the first 5 rows:

```
In [45]: frame.head()
Out[45]:
   pop    state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
```

If you specify a sequence of columns, the DataFrame's columns will be arranged in that order:

```
In [46]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[46]:
   year    state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
```

If you pass a column that isn't contained in the dict, it will appear with missing values in the result:

```
In [47]: frame2 = pd.DataFrame(data, columns=['year', 'state',
                                             ....,
                                             index=['one', 'two', 'three', 'four', 'five'])
In [48]: frame2
Out[48]:
   year    state  pop  debt
one   2000    Ohio  1.5   NaN
two   2001    Ohio  1.7   NaN
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4   NaN
five   2002  Nevada  2.9   NaN

In [49]: frame2.columns
Out[49]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

```
In [50]: frame2['state']
Out[50]:
one      Ohio
```

```
two        Ohio
three      Ohio
four       Nevada
five       Nevada
Name: state, dtype: object
```

```
In [51]: frame2.year
Out[51]:
one    2000
two    2001
three   2002
four    2001
five    2002
Name: year, dtype: int64
```

#### Note

Attribute-like access (e.g. `frame2.year`) and tab completion of column names in IPython is provided as a convenience; it's more precise to write `frame2['year']` instead.

Note that the returned Series have the same index as the DataFrame, and their `name` attribute has been appropriately set.

Rows can also be retrieved by position or name by a couple of methods, such as the `loc` indexing field (much more on this later):

```
In [52]: frame2.loc['three']
Out[52]:
year    2002
state   Ohio
pop     3.6
debt    NaN
Name: three, dtype: object
```

Columns can be modified by assignment. For example, the empty 'debt' column could be assigned a scalar value or an array of values:

```
In [53]: frame2['debt'] = 16.5
In [54]: frame2
Out[54]:
      year  state  pop  debt
```

```
one    2000    Ohio  1.5  16.5
two    2001    Ohio  1.7  16.5
three  2002    Ohio  3.6  16.5
four   2001    Nevada 2.4  16.5
five   2002    Nevada 2.9  16.5
```

```
In [55]: frame2['debt'] = np.arange(5.)
```

```
In [56]: frame2
```

```
Out[56]:
```

```
      year    state  pop  debt
one    2000    Ohio  1.5  0.0
two    2001    Ohio  1.7  1.0
three  2002    Ohio  3.6  2.0
four   2001    Nevada 2.4  3.0
five   2002    Nevada 2.9  4.0
```

When assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If you assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any holes:

```
In [57]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
```

```
In [58]: frame2['debt'] = val
```

```
In [59]: frame2
```

```
Out[59]:
```

```
      year    state  pop  debt
one    2000    Ohio  1.5  NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6  NaN
four   2001    Nevada 2.4  -1.5
five   2002    Nevada 2.9  -1.7
```

Assigning a column that doesn't exist will create a new column. The `del` keyword will delete columns as with a dict:

```
In [60]: frame2['eastern'] = frame2.state == 'Ohio'
```

```
In [61]: frame2
```

```
Out[61]:
```

```
      year    state  pop  debt  eastern
one    2000    Ohio  1.5  NaN    True
two    2001    Ohio  1.7  -1.2   True
three  2002    Ohio  3.6  NaN    True
four   2001    Nevada 2.4  -1.5  False
```

```
five    2002  Nevada  2.9  -1.7    False
In [62]: del frame2['eastern']

In [63]: frame2.columns
Out[63]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

### Caution

The column returned when indexing a DataFrame is a *view* on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame. The column can be explicitly copied using the Series's `copy` method.

Another common form of data is a nested dict of dicts format:

```
In [64]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
....:             'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

If passed to DataFrame, it will interpret the outer dict keys as the columns and the inner keys as the row indices:

```
In [65]: frame3 = pd.DataFrame(pop)

In [66]: frame3
Out[66]:
      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
2002     2.9   3.6
```

You can transpose the DataFrame (swap rows and columns) with similar syntax to a NumPy array:

```
In [67]: frame3.T
Out[67]:
      2000  2001  2002
Nevada  NaN   2.4   2.9
Ohio    1.5   1.7   3.6
```

The keys in the inner dicts are combined and sorted to form the index in the result. This isn't true if an explicit index is specified:

```
In [68]: pd.DataFrame(pop, index=[2001, 2002, 2003])
Out[68]:
      Nevada  Ohio
2001      2.4   1.7
2002      2.9   3.6
2003      NaN   NaN
```

Dicts of Series are treated much in the same way:

```
In [69]: pdata = {'Ohio': frame3['Ohio'][:-1],
....:             'Nevada': frame3['Nevada'][:2]}

In [70]: pd.DataFrame(pdata)
Out[70]:
      Nevada  Ohio
2000      NaN   1.5
2001      2.4   1.7
```

For a complete list of things you can pass the DataFrame constructor, see [Table 5-1](#).

If a DataFrame's index and columns have their name attributes set, these will also be displayed:

```
In [71]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [72]: frame3
Out[72]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6
```

Like Series, the values attribute returns the data contained in the DataFrame as a 2D ndarray:

```
In [73]: frame3.values
Out[73]:
array([[ nan,  1.5],
       [ 2.4,  1.7],
       [ 2.9,  3.6]])
```

If the DataFrame's columns are different dtypes, the dtype of the values array

will be chosen to accommodate all of the columns:

```
In [74]: frame2.values  
Out[74]:  
array([[2000, 'Ohio', 1.5, nan],  
       [2001, 'Ohio', 1.7, -1.2],  
       [2002, 'Ohio', 3.6, nan],  
       [2001, 'Nevada', 2.4, -1.5],  
       [2002, 'Nevada', 2.9, -1.7]], dtype=object)
```

Table 5-1. Possible data inputs to DataFrame constructor

| Type                                | Notes   |
|-------------------------------------|---|
| 2D ndarray                          | A matrix of data, passing optional row and column labels  |
| dict of arrays,<br>lists, or tuples | Each sequence becomes a column in the DataFrame. All sequences must be the same length.   |
| NumPy<br>structured/record<br>array | Treated as the “dict of arrays” case  |
| dict of Series                      | Each value becomes a column. Indexes from each Series are unioned together to form the result’s row index if no explicit index is passed. |
| dict of dicts                       | Each inner dict becomes a column. Keys are unioned to form the row index as in the “dict of Series” case.                                 |
| list of dicts or<br>Series          | Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame’s column labels                       |
| List of lists or<br>tuples          | Treated as the “2D ndarray” case  |

|                   |   |
|-------------------|---|
| Another DataFrame | The DataFrame's indexes are used unless different ones are passed                         |
| NumPy MaskedArray | Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result |

# Index Objects

pandas's Index objects are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or other sequence of labels used when constructing a Series or DataFrame is internally converted to an Index:

```
In [75]: obj = pd.Series(range(3), index=['a', 'b', 'c'])

In [76]: index = obj.index

In [77]: index
Out[77]: Index(['a', 'b', 'c'], dtype='object')

In [78]: index[1:]
Out[78]: Index(['b', 'c'], dtype='object')
```

Index objects are immutable and thus can't be modified by the user:

```
index[1] = 'd' # TypeError
```

Immutability makes it safer to share Index objects among data structures:

```
In [79]: index = pd.Index(np.arange(3))

In [80]: obj2 = pd.Series([1.5, -2.5, 0], index=index)

In [81]: obj2.index is index
Out[81]: True
```

[Table 5-2](#) has a list of built-in Index classes in the library. With some development effort, Index can even be subclassed to implement specialized axis indexing functionality.

## Note

Some users will not often take advantage of the capabilities provided by pandas's indexes, but, even if you don't, it's important to understand how they work as some operations will yield results containing indexed data.

Table 5-2. Main Index objects in pandas

| Class            | Description   |
|------------------|---|
| Index            | The most general Index object, representing axis labels in a NumPy array of Python objects.   |
| Int64Index       | Specialized Index for integer values.   |
| Float64Index     | Specialized Index for floating point values.  |
| MultiIndex       | “Hierarchical” index object representing multiple levels of indexing on a single axis. Can be thought of as similar to an array of tuples.  |
| RangeIndex       | An integer index for the special case of a regularly spaced sequence, similar to the Python <code>range(start, stop, step)</code> function. |
| CategoricalIndex | An index of values with <code>category</code>   |
| DatetimeIndex    | Stores nanosecond timestamps (represented using NumPy’s <code>datetime64</code> dtype).   |
| PeriodIndex      | Specialized Index for Period data (timespans).  |

In addition to being array-like, an Index also behaves like a fixed-size set:

```
In [82]: frame3
Out[82]:
state    Nevada    Ohio
year
2000      NaN     1.5
2001      2.4     1.7
2002      2.9     3.6

In [83]: 'Ohio' in frame3.columns
Out[83]: True

In [84]: 2003 in frame3.index
Out[84]: False
```

Each Index has a number of methods and properties for set logic and answering other common questions about the data it contains. Some useful ones are summarized in [Table 5-3](#).

Table 5-3. Some Index methods and properties

| <b>Method</b> | <b>Description</b>   |
|---------------|--|
| append        | Concatenate with additional Index objects, producing a new Index                           |
| diff          | Compute set difference as an Index   |
| intersection  | Compute set intersection   |
| union         | Compute set union  |
| isin          | Compute boolean array indicating whether each value is contained in the passed collection  |
| delete        | Compute new Index with element at index $i$ deleted  |
| drop          | Compute new index by deleting passed values  |
| insert        | Compute new Index by inserting element at index $i$  |
| is_monotonic  | Returns <code>True</code> if each element is greater than or equal to the previous element |
| is_unique     | Returns <code>True</code> if the Index has no duplicate values                             |
| unique        | Compute the array of unique values in the Index  |

# Essential Functionality

In this section, I'll walk you through the fundamental mechanics of interacting with the data contained in a Series or DataFrame. Upcoming chapters will delve more deeply into data analysis and manipulation topics using pandas. This book is not intended to serve as exhaustive documentation for the pandas library; I instead focus on the most important features, leaving the less common (that is, more esoteric) things for you to explore on your own.

# Reindexing

An important method on pandas objects is `reindex`, which means to create a new object with the data *conformed* to a new index. Consider a simple example from above:

```
In [85]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b'])

In [86]: obj
Out[86]:
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

Calling `reindex` on this Series rearranges the data according to the new index, introducing missing values if any index values were not already present:

```
In [87]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [88]: obj2
Out[88]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

For ordered data like time series, it may be desirable to do some interpolation or filling of values when reindexing. The `method` option allows us to do this, using a method such as `ffill` which forward fills the values:

```
In [89]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index=range(3))

In [90]: obj3.reindex(range(6), method='ffill')
Out[90]:
0      blue
1      blue
2    purple
3    purple
4    purple
5    purple
```

```
3    purple
4    yellow
5    yellow
dtype: object
```

[Table 5-4](#) lists available `method` options. At this time, interpolation more sophisticated than forward- and backfilling would need to be applied after the fact.

Table 5-4. `reindex` method (interpolation) options

| Argument                                    | Description                     |
|---|---------------------------------|
| <code>ffill</code> or <code>pad</code>      | Fill (or carry) values forward  |
| <code>bfill</code> or <code>backfill</code> | Fill (or carry) values backward |

With `DataFrame`, `reindex` can alter either the (row) index, columns, or both. When passed only a sequence, the rows are reindexed in the result:

```
In [91]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)), index=range(3), columns=['Ohio', 'Texas', 'California'])
Out[91]:
```

```
   Ohio    Texas    California
0      0        1            2
1      3        4            5
2      6        7            8
```

```
In [92]: frame
Out[92]:
```

```
   Ohio    Texas    California
0      0        1            2
1      3        4            5
2      6        7            8
```

```
In [93]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [94]: frame2
Out[94]:
```

|   | Ohio | Texas | California |
|---|------|-------|------------|
| a | 0.0  | 1.0   | 2.0        |
| b | NaN  | NaN   | NaN        |
| c | 3.0  | 4.0   | 5.0        |
| d | 6.0  | 7.0   | 8.0        |

The columns can be reindexed using the `columns` keyword:

```
In [95]: states = ['Texas', 'Utah', 'California']

In [96]: frame.reindex(columns=states)
Out[96]:
```

|   | Texas | Utah | California |
|---|-------|------|------------|
| a | 1     | NaN  | 2          |
| c | 4     | NaN  | 5          |
| d | 7     | NaN  | 8          |

Both can be reindexed in one shot, though interpolation will only apply row-wise (axis 0).

See [Table 5-5](#) for more about the arguments to `reindex`.

As you'll see soon, reindexing can be done more succinctly by label-indexing with `loc`:

```
In [97]: frame.loc[['a', 'b', 'c', 'd'], states]
Out[97]:
   Texas  Utah  California
a     1.0    NaN      2.0
b     NaN    NaN      NaN
c     4.0    NaN      5.0
d     7.0    NaN      8.0
```

Table 5-5. `reindex` function arguments

| Argument                | Description   |
|-------------------------|---|
| <code>index</code>      | New sequence to use as index. Can be <code>Index</code> instance or any other sequence-like Python data structure. An <code>Index</code> will be used exactly as is without any copying |
| <code>method</code>     | Interpolation (fill) method, see <a href="#">Table 5-4</a> for options.   |
| <code>fill_value</code> | Substitute value to use when introducing missing data by reindexing   |
| <code>limit</code>      | When forward- or backfilling, maximum size gap (in number of elements) to fill  |
| <code>tolerance</code>  | When forward- or backfilling, maximum size gap (in absolute numeric distance) to fill for inexact matches   |
| <code>level</code>      | Match simple <code>Index</code> on level of <code>MultiIndex</code> , otherwise select subset of  |
| <code>copy</code>       | If <code>True</code> , always copy underlying data even if new index is equivalent to old index. If <code>False</code> , do not copy the data when the indexes are equivalent.          |

# Dropping entries from an axis

Dropping one or more entries from an axis is easy if you already have an index array or list without those entries. As that can require a bit of munging and set logic, the `drop` method will return a new object with the indicated value or values deleted from an axis:

```
In [98]: obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [99]: new_obj = obj.drop('c')
```

```
In [100]: new_obj
```

```
Out[100]:
```

```
a    0.0  
b    1.0  
d    3.0  
e    4.0  
dtype: float64
```

```
In [101]: obj.drop(['d', 'c'])
```

```
Out[101]:
```

```
a    0.0  
b    1.0  
e    4.0  
dtype: float64
```

With DataFrame, index values can be deleted from either axis:

```
In [102]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),  
.....:                               index=['Ohio', 'Colorado', 'Utah',  
.....:                               columns=['one', 'two', 'three', 'four'])
```

```
In [103]: data.drop(['Colorado', 'Ohio'])
```

```
Out[103]:
```

|          | one | two | three | four |
|----------|-----|-----|-------|------|
| Utah     | 8   | 9   | 10    | 11   |
| New York | 12  | 13  | 14    | 15   |

```
In [104]: data.drop('two', axis=1)
```

```
Out[104]:
```

|          | one | three | four |
|----------|-----|-------|------|
| Ohio     | 0   | 2     | 3    |
| Colorado | 4   | 6     | 7    |

```
Utah      8      10      11
New York  12      14      15
```

```
In [105]: data.drop(['two', 'four'], axis=1)
Out[105]:
          one   three
Ohio      0      2
Colorado  4      6
Utah      8      10
New York  12      14
```

Many functions, like `drop`, which modify the size or shape of a Series or DataFrame, can manipulate an object *in place* without returning a new object:

```
In [106]: obj.drop('c', inplace=True)
```

```
In [107]: obj
Out[107]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

# Indexing, selection, and filtering

Series indexing (`obj[...]`) works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers. Here are some examples of this:

```
In [108]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c',  
In [109]: obj['b']  
Out[109]: 1.0  
  
In [110]: obj[1]  
Out[110]: 1.0  
  
In [111]: obj[2:4]  
Out[111]:  
c    2.0  
d    3.0  
dtype: float64  
  
In [112]: obj[['b', 'a', 'd']]  
Out[112]:  
b    1.0  
a    0.0  
d    3.0  
dtype: float64  
  
In [113]: obj[[1, 3]]  
Out[113]:  
b    1.0  
d    3.0  
dtype: float64  
  
In [114]: obj[obj < 2]  
Out[114]:  
a    0.0  
b    1.0  
dtype: float64
```

Slicing with labels behaves differently than normal Python slicing in that the endpoint is inclusive:

```
In [115]: obj['b':'c']
```

```
Out[115]:  
b    1.0  
c    2.0  
dtype: float64
```

*Setting* using these methods modifies the corresponding section of the Series:

```
In [116]: obj['b':'c'] = 5
```

```
In [117]: obj  
Out[117]:  
a    0.0  
b    5.0  
c    5.0  
d    3.0  
dtype: float64
```

As we'll see soon, there are more precise ways to select or index a Series.

Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

```
In [118]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),  
.....:                               index=['Ohio', 'Colorado', 'Utah',  
.....:                               columns=['one', 'two', 'three',
```

```
In [119]: data  
Out[119]:  
      one  two  three  four  
Ohio      0    1     2    3  
Colorado  4    5     6    7  
Utah     8    9    10   11  
New York 12   13    14   15
```

```
In [120]: data['two']  
Out[120]:  
Ohio      1  
Colorado  5  
Utah     9  
New York 13  
Name: two, dtype: int64
```

```
In [121]: data[['three', 'one']]  
Out[121]:  
      three  one  
Ohio      2    0
```

```
Colorado      6      4
Utah         10     8
New York    14     12
```

Indexing like this has a few special cases. First selecting rows by slicing or a boolean array:

```
In [122]: data[:2]
Out[122]:
          one  two  three  four
Ohio      0    1    2    3
Colorado  4    5    6    7

In [123]: data[data['three'] > 5]
Out[123]:
          one  two  three  four
Colorado  4    5    6    7
Utah     8    9   10   11
New York 12   13   14   15
```

This might seem inconsistent to some readers, but this syntax arose out of practicality. Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

```
In [124]: data < 5
Out[124]:
          one  two  three  four
Ohio     True  True  True  True
Colorado  True False False False
Utah     False False False False
New York False False False False

In [125]: data[data < 5] = 0

In [126]: data
Out[126]:
          one  two  three  four
Ohio      0    0    0    0
Colorado  0    5    6    7
Utah     8    9   10   11
New York 12   13   14   15
```

This makes DataFrame syntactically more like a two-dimensional NumPy array in this particular case.

For DataFrame label-indexing on the rows, I introduce the special indexing operators `loc` and `iloc`. They enable you to select a subset of the rows and columns from a DataFrame with NumPy-like notation using either axis labels (`loc`) or integers (`iloc`).

```
In [127]: data.loc['Colorado', ['two', 'three']]  
Out[127]:  
two      5  
three     6  
Name: Colorado, dtype: int64  
  
In [128]: data.iloc[[1, 2], [3, 0, 1]]  
Out[128]:  
        four   one   two  
Colorado    7     0     5  
Utah       11    8     9  
  
In [129]: data.iloc[2]  
Out[129]:  
one      8  
two      9  
three    10  
four     11  
Name: Utah, dtype: int64  
  
In [130]: data.loc[:'Utah', 'two']  
Out[130]:  
Ohio      0  
Colorado  5  
Utah      9  
Name: two, dtype: int64  
  
In [131]: data.iloc[:, :3][data.three > 5]  
Out[131]:  
        one   two   three  
Colorado  0     5     6  
Utah     8     9     10  
New York 12    13    14
```

So there are many ways to select and rearrange the data contained in a pandas object. For DataFrame, there is a short summary of many of them in [Table 5-6](#). You have a number of additional options when working with hierarchical indexes as you'll later see.

#### Note

When designing pandas, I felt that having to type `frame[:, col]` to select a column was too verbose (and error-prone), since column selection is one of the most common operations. I originally made the design trade-off to push all of the fancy indexing behavior (both labels and integers) into the `ix` operator. In practice, this led to many edge cases in data with integer axis labels, so the pandas team decided to create the `loc` and `iloc` operators to deal with strictly label-based and integer-based indexing, respectively.

Table 5-6. Indexing options with DataFrame

| Type  | Notes  |
|---|--|
| <code>df[val]</code>                                      | Select single column or sequence of columns from the DataFrame. Special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion). |
| <code>df.loc[val]</code>                                  | Selects single row or subset of rows from the DataFrame by label.  |
| <code>df.loc[:, val]</code>                               | Selects single column of subset of columns by label.   |
| <code>df.loc[val1, val2]</code>                           | Select both rows and columns by label.   |
| <code>df.iloc[where]</code>                               | Selects single row or subset of rows from the DataFrame by label.  |
| <code>df.iloc[:, where]</code>                            | Selects single column of subset of columns by integer position.  |
| <code>df.iloc[where_i, where_j]</code>                    | Select both rows and columns by integer position.  |
| <code>df.at[label_i, label_j]</code>                      | Select a single scalar value by row and column label.  |
| <code>df.iat[i, j]</code>                                 | Select a single scalar value by row and column position (integers).  |
| <code>reindex</code> method                               | Select either rows or columns by labels.   |
| <code>xs</code> method                                    | Select single row or column as a Series by label.  |
| <code>get_value,</code><br><code>set_value</code> methods | Select single value by row and column label.   |

# Arithmetic and data alignment

An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes. When adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs. For users with database experience, this is similar to an automatic outer join on the index labels. Let's look at an example:

```
In [132]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c'])  
In [133]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'd', 'e', 'f'])  
  
In [134]: s1  
Out[134]:  
a    7.3  
c   -2.5  
d    3.4  
e    1.5  
dtype: float64  
  
In [135]: s2  
Out[135]:  
a   -2.1  
c    3.6  
e   -1.5  
f    4.0  
g    3.1  
dtype: float64
```

Adding these together yields:

```
In [136]: s1 + s2  
Out[136]:  
a    5.2  
c    1.1  
d    NaN  
e    0.0  
f    NaN  
g    NaN  
dtype: float64
```

The internal data alignment introduces NA values in the label locations that

don't overlap. Missing values will then propagate in further arithmetic computations.

In the case of DataFrame, alignment is performed on both the rows and the columns:

```
In [137]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), co:  
.....: index=['Ohio', 'Texas', 'Colorado']  
  
In [138]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), co:  
.....: index=['Utah', 'Ohio', 'Texas', 'O  
  
In [139]: df1  
Out[139]:  
          b    c    d  
Ohio      0.0  1.0  2.0  
Texas     3.0  4.0  5.0  
Colorado   6.0  7.0  8.0  
  
In [140]: df2  
Out[140]:  
          b    d    e  
Utah      0.0  1.0  2.0  
Ohio      3.0  4.0  5.0  
Texas     6.0  7.0  8.0  
Oregon    9.0 10.0 11.0
```

Adding these together returns a DataFrame whose index and columns are the unions of the ones in each DataFrame:

```
In [141]: df1 + df2  
Out[141]:  
          b    c    d    e  
Colorado  NaN  NaN  NaN  NaN  
Ohio      3.0  NaN  6.0  NaN  
Oregon    NaN  NaN  NaN  NaN  
Texas     9.0  NaN 12.0  NaN  
Utah      NaN  NaN  NaN  NaN
```

## Arithmetic methods with fill values

In arithmetic operations between differently-indexed objects, you might want to fill with a special value, like 0, when an axis label is found in one object but

not the other:

```
In [142]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)), columns=['a', 'b', 'c', 'd'])
In [143]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)), columns=['a', 'b', 'c', 'd', 'e'])
In [144]: df1
Out[144]:
   a    b    c    d
0  0.0  1.0  2.0  3.0
1  4.0  5.0  6.0  7.0
2  8.0  9.0  10.0 11.0

In [145]: df2
Out[145]:
   a    b    c    d    e
0  0.0  1.0  2.0  3.0  4.0
1  5.0  6.0  7.0  8.0  9.0
2 10.0 11.0 12.0 13.0 14.0
3 15.0 16.0 17.0 18.0 19.0
```

Adding these together results in NA values in the locations that don't overlap:

```
In [146]: df1 + df2
Out[146]:
   a    b    c    d    e
0  0.0  2.0  4.0  6.0  NaN
1  9.0 11.0 13.0 15.0  NaN
2 18.0 20.0 22.0 24.0  NaN
3  NaN  NaN  NaN  NaN  NaN
```

Using the `add` method on `df1`, I pass `df2` and an argument to `fill_value`:

```
In [147]: df1.add(df2, fill_value=0)
Out[147]:
   a    b    c    d    e
0  0.0  2.0  4.0  6.0  4.0
1  9.0 11.0 13.0 15.0  9.0
2 18.0 20.0 22.0 24.0 14.0
3 15.0 16.0 17.0 18.0 19.0
```

Relatedly, when reindexing a Series or DataFrame, you can also specify a different fill value:

```
In [148]: df1.reindex(columns=df2.columns, fill_value=0)
```

```
Out[148]:
```

|   | a   | b   | c    | d    | e |
|---|-----|-----|------|------|---|
| 0 | 0.0 | 1.0 | 2.0  | 3.0  | 0 |
| 1 | 4.0 | 5.0 | 6.0  | 7.0  | 0 |
| 2 | 8.0 | 9.0 | 10.0 | 11.0 | 0 |

Table 5-7. Flexible arithmetic methods

| Method              | Description                     |
|---------------------|---------------------------------|
| add, radd           | Methods for addition (+)        |
| sub, rsub           | Methods for subtraction (-)     |
| div, rdiv           | Methods for division (/)        |
| floordiv, rfloordiv | Methods for floor division (//) |
| mul, rmul           | Methods for multiplication (*)  |
| pow, rpow           | Methods for exponentiation (**) |

## Operations between DataFrame and Series

As with NumPy arrays of different dimensions, arithmetic between DataFrame and Series is also defined. First, as a motivating example, consider the difference between a 2D array and one of its rows:

```
In [149]: arr = np.arange(12.).reshape((3, 4))

In [150]: arr
Out[150]:
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

```
In [151]: arr[0]
Out[151]: array([ 0.,  1.,  2.,  3.])

In [152]: arr - arr[0]
Out[152]:
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])
```

This is referred to as *broadcasting* and is explained in more detail in [Chapter 12](#). Operations between a DataFrame and a Series are similar:

```
In [153]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
.....:                               index=['Utah', 'Ohio', 'Texas',
In [154]: series = frame.iloc[0]

In [155]: frame
Out[155]:
      b      d      e
Utah  0.0  1.0  2.0
Ohio  3.0  4.0  5.0
Texas 6.0  7.0  8.0
Oregon 9.0 10.0 11.0

In [156]: series
Out[156]:
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

By default, arithmetic between DataFrame and Series matches the index of the Series on the DataFrame's columns, broadcasting down the rows:

```
In [157]: frame - series
Out[157]:
      b      d      e
Utah  0.0  0.0  0.0
Ohio  3.0  3.0  3.0
Texas 6.0  6.0  6.0
Oregon 9.0  9.0  9.0
```

If an index value is not found in either the DataFrame's columns or the Series's index, the objects will be reindexed to form the union:

```
In [158]: series2 = pd.Series(range(3), index=['b', 'e', 'f'])

In [159]: frame + series2
Out[159]:
      b      d      e      f
Utah  0.0  NaN  3.0  NaN
Ohio  3.0  NaN  6.0  NaN
Texas 6.0  NaN  9.0  NaN
Oregon 9.0  NaN 12.0  NaN
```

If you want to instead broadcast over the columns, matching on the rows, you

have to use one of the arithmetic methods. For example:

```
In [160]: series3 = frame['d']
```

```
In [161]: frame
Out[161]:
      b      d      e
Utah    0.0    1.0    2.0
Ohio     3.0    4.0    5.0
Texas    6.0    7.0    8.0
Oregon   9.0   10.0   11.0
```

```
In [162]: series3
```

```
Out[162]:
Utah      1.0
Ohio      4.0
Texas     7.0
Oregon   10.0
Name: d, dtype: float64
```

```
In [163]: frame.sub(series3, axis=0)
```

```
Out[163]:
      b      d      e
Utah  -1.0    0.0    1.0
Ohio  -1.0    0.0    1.0
Texas -1.0    0.0    1.0
Oregon -1.0   0.0    1.0
```

The axis number that you pass is the *axis to match on*. In this case we mean to match on the DataFrame's row index (`axis=0`) and broadcast across.

# Function application and mapping

NumPy ufuncs (element-wise array methods) also work with pandas objects:

```
In [164]: frame = pd.DataFrame(np.random.randn(4, 3), columns=[  
.....: index=['Utah', 'Ohio', 'Texas',  
  
In [165]: frame  
Out[165]:  
          b           d           e  
Utah    -0.204708  0.478943 -0.519439  
Ohio     -0.555730  1.965781  1.393406  
Texas    0.092908  0.281746  0.769023  
Oregon   1.246435  1.007189 -1.296221  
  
In [166]: np.abs(frame)  
Out[166]:  
          b           d           e  
Utah    0.204708  0.478943  0.519439  
Ohio     0.555730  1.965781  1.393406  
Texas    0.092908  0.281746  0.769023  
Oregon   1.246435  1.007189  1.296221
```

Another frequent operation is applying a function on 1D arrays to each column or row. DataFrame's `apply` method does exactly this:

```
In [167]: f = lambda x: x.max() - x.min()  
  
In [168]: frame.apply(f)  
Out[168]:  
b    1.802165  
d    1.684034  
e    2.689627  
dtype: float64  
  
In [169]: frame.apply(f, axis=1)  
Out[169]:  
Utah      0.998382  
Ohio      2.521511  
Texas     0.676115  
Oregon    2.542656  
dtype: float64
```

Many of the most common array statistics (like `sum` and `mean`) are `DataFrame` methods, so using `apply` is not necessary.

The function passed to `apply` need not return a scalar value, it can also return a `Series` with multiple values:

```
In [170]: def f(x):
....:     return pd.Series([x.min(), x.max()], index=['min', 'max'])

In [171]: frame.apply(f)
Out[171]:
      b          d          e
min -0.555730  0.281746 -1.296221
max  1.246435  1.965781  1.393406
```

Element-wise Python functions can be used, too. Suppose you wanted to compute a formatted string from each floating point value in `frame`. You can do this with `applymap`:

```
In [172]: format = lambda x: '%.2f' % x

In [173]: frame.applymap(format)
Out[173]:
      b          d          e
Utah   -0.20    0.48   -0.52
Ohio    -0.56    1.97    1.39
Texas    0.09    0.28    0.77
Oregon   1.25    1.01   -1.30
```

The reason for the name `applymap` is that `Series` has a `map` method for applying an element-wise function:

```
In [174]: frame['e'].map(format)
Out[174]:
Utah      -0.52
Ohio       1.39
Texas      0.77
Oregon     -1.30
Name: e, dtype: object
```

# Sorting and ranking

Sorting a data set by some criterion is another important built-in operation. To sort lexicographically by row or column index, use the `sort_index` method, which returns a new, sorted object:

```
In [175]: obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])

In [176]: obj.sort_index()
Out[176]:
a    1
b    2
c    3
d    0
dtype: int64
```

With a DataFrame, you can sort by index on either axis:

```
In [177]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)), index=['one', 'three'],
                               columns=['d', 'a', 'b', 'c'])

In [178]: frame.sort_index()
Out[178]:
      d  a  b  c
one  4  5  6  7
three 0  1  2  3

In [179]: frame.sort_index(axis=1)
Out[179]:
      a  b  c  d
three 1  2  3  0
one   5  6  7  4
```

The data is sorted in ascending order by default, but can be sorted in descending order, too:

```
In [180]: frame.sort_index(axis=1, ascending=False)
Out[180]:
      d  c  b  a
three 0  3  2  1
one   4  7  6  5
```

To sort a Series by its values, use its `sort_values` method:

```
In [181]: obj = pd.Series([4, 7, -3, 2])
```

```
In [182]: obj.sort_values()
```

```
Out[182]:
```

```
2    -3  
3     2  
0     4  
1     7  
dtype: int64
```

Any missing values are sorted to the end of the Series by default:

```
In [183]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
```

```
In [184]: obj.sort_values()
```

```
Out[184]:
```

```
4    -3.0  
5     2.0  
0     4.0  
2     7.0  
1     NaN  
3     NaN  
dtype: float64
```

When sorting a DataFrame, you can use the data in one or more columns as the sort keys. To do so, pass one or more column names to the `by` option of `sort_values`:

```
In [185]: frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1,
```

```
In [186]: frame
```

```
Out[186]:
```

```
   a   b  
0  0   4  
1  1   7  
2  0  -3  
3  1   2
```

```
In [187]: frame.sort_values(by='b')
```

```
Out[187]:
```

```
   a   b  
2  0  -3  
3  1   2  
0  0   4
```

```
1 1 7
```

To sort by multiple columns, pass a list of names:

```
In [188]: frame.sort_values(by=['a', 'b'])
Out[188]:
   a   b
2  0 -3
0  0  4
3  1  2
1  1  7
```

*Ranking* is closely related to sorting, assigning ranks from one through the number of valid data points in an array. The `rank` methods for Series and DataFrame are the place to look; by default `rank` breaks ties by assigning each group the mean rank:

```
In [189]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
In [190]: obj.rank()
Out[190]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

Ranks can also be assigned according to the order they're observed in the data:

```
In [191]: obj.rank(method='first')
Out[191]:
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

Naturally, you can rank in descending order, too:

```
# Assign tie values the maximum rank in the group
In [192]: obj.rank(ascending=False, method='max')
Out[192]:
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

See [Table 5-8](#) for a list of tie-breaking methods available. DataFrame can compute ranks over the rows or the columns:

```
In [193]: frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0,
.....:                   'c': [-2, 5, 8, -2.5]})

In [194]: frame
Out[194]:
   a    b    c
0  0  4.3 -2.0
1  1  7.0  5.0
2  0 -3.0  8.0
3  1  2.0 -2.5

In [195]: frame.rank(axis=1)
Out[195]:
   a    b    c
0  2.0  3.0  1.0
1  1.0  3.0  2.0
2  2.0  1.0  3.0
3  2.0  3.0  1.0
```

**Table 5-8.** Tie-breaking methods with rank

| Method    | Description  |
|-----------|--|
| 'average' | Default: assign the average rank to each entry in the equal group.   |
| 'min'     | Use the minimum rank for the whole group.  |
| 'max'     | Use the maximum rank for the whole group.  |
| 'first'   | Assign ranks in the order the values appear in the data.   |
| 'dense'   | Like <code>method='min'</code> , but ranks always increase by 1 in between groups rather than the number of equal elements in a group. |

# Axis indexes with duplicate values

Up until now all of the examples I've shown you have had unique axis labels (index values). While many pandas functions (like `reindex`) require that the labels be unique, it's not mandatory. Let's consider a small Series with duplicate indices:

```
In [196]: obj = pd.Series(range(5), index=['a', 'a', 'b', 'b',  
In [197]: obj  
Out[197]:  
a    0  
a    1  
b    2  
b    3  
c    4  
dtype: int64
```

The index's `is_unique` property can tell you whether its values are unique or not:

```
In [198]: obj.index.is_unique  
Out[198]: False
```

Data selection is one of the main things that behaves differently with duplicates. Indexing a value with multiple entries returns a Series while single entries return a scalar value:

```
In [199]: obj['a']  
Out[199]:  
a    0  
a    1  
dtype: int64
```

```
In [200]: obj['c']  
Out[200]: 4
```

This can make your code more complicated as the output type from indexing can vary based on whether a label is repeated or not.

The same logic extends to indexing rows in a DataFrame:

```
In [201]: df = pd.DataFrame(np.random.randn(4, 3), index=['a',
```

```
In [202]: df
```

```
Out[202]:
```

|   | 0        | 1         | 2         |
|---|----------|-----------|-----------|
| a | 0.274992 | 0.228913  | 1.352917  |
| a | 0.886429 | -2.001637 | -0.371843 |
| b | 1.669025 | -0.438570 | -0.539741 |
| b | 0.476985 | 3.248944  | -1.021228 |

```
In [203]: df.loc['b']
```

```
Out[203]:
```

|   | 0        | 1         | 2         |
|---|----------|-----------|-----------|
| b | 1.669025 | -0.438570 | -0.539741 |
| b | 0.476985 | 3.248944  | -1.021228 |

# Summarizing and Computing Descriptive Statistics

pandas objects are equipped with a set of common mathematical and statistical methods. Most of these fall into the category of *reductions* or *summary statistics*, methods that extract a single value (like the sum or mean) from a Series or a Series of values from the rows or columns of a DataFrame. Compared with the similar methods found on NumPy arrays, they have built-in handling for missing data. Consider a small DataFrame:

```
In [204]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
.....:                   [np.nan, np.nan], [0.75, -1.3]],
.....:                   index=['a', 'b', 'c', 'd'],
.....:                   columns=['one', 'two'])

In [205]: df
Out[205]:
   one  two
a  1.40  NaN
b  7.10 -4.5
c    NaN  NaN
d  0.75 -1.3
```

Calling DataFrame's `sum` method returns a Series containing column sums:

```
In [206]: df.sum()
Out[206]:
one    9.25
two   -5.80
dtype: float64
```

Passing `axis=1` sums over the rows instead:

```
In [207]: df.sum(axis=1)
Out[207]:
a    1.40
b    2.60
c    NaN
d   -0.55
```

```
dtype: float64
```

NA values are excluded unless the entire slice (row or column in this case) is NA. This can be disabled using the `skipna` option:

```
In [208]: df.mean(axis=1, skipna=False)
Out[208]:
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

See [Table 5-9](#) for a list of common options for each reduction method.

Table 5-9. Options for reduction methods

| Method              | Description   |
|---------------------|---|
| <code>axis</code>   | Axis to reduce over. 0 for DataFrame's rows and 1 for columns.              |
| <code>skipna</code> | Exclude missing values, <code>True</code> by default.                       |
| <code>level</code>  | Reduce grouped by level if the axis is hierarchically-indexed (MultiIndex). |

Some methods, like `idxmin` and `idxmax`, return indirect statistics like the index value where the minimum or maximum values are attained:

```
In [209]: df.idxmax()
Out[209]:
one    b
two    d
dtype: object
```

Other methods are *accumulations*:

```
In [210]: df.cumsum()
Out[210]:
       one  two
a  1.40  NaN
b  8.50 -4.5
c  NaN   NaN
d  9.25 -5.8
```

Another type of method is neither a reduction nor an accumulation. `describe`

is one such example, producing multiple summary statistics in one shot:

```
In [211]: df.describe()
Out[211]:
          one      two
count    3.000000  2.000000
mean    3.083333 -2.900000
std     3.493685  2.262742
min     0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

On non-numeric data, `describe` produces alternate summary statistics:

```
In [212]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)
In [213]: obj.describe()
Out[213]:
count    16
unique     3
top       a
freq      8
dtype: object
```

See [Table 5-10](#) for a full list of summary statistics and related methods.

Table 5-10. Descriptive and summary statistics

| Method         | Description   |
|----------------|---|
| count          | Number of non-NA values   |
| describe       | Compute set of summary statistics for Series or each DataFrame column                       |
| min, max       | Compute minimum and maximum values  |
| argmin, argmax | Compute index locations (integers) at which minimum or maximum value obtained, respectively |
| idxmin, idxmax | Compute index values at which minimum or maximum value obtained, respectively               |
| quantile       | Compute sample quantile ranging from 0 to 1   |
| sum            | Sum of values   |

|                   |  |
|-------------------|--|
| mean              | Mean of values   |
| median            | Arithmetic median (50% quantile) of values                 |
| mad               | Mean absolute deviation from mean value                    |
| prod              | Product of all values                                      |
| var               | Sample variance of values                                  |
| std               | Sample standard deviation of values                        |
| skew              | Sample skewness (3rd moment) of values                     |
| kurt              | Sample kurtosis (4th moment) of values                     |
| cumsum            | Cumulative sum of values                                   |
| cummin,<br>cummax | Cumulative minimum or maximum of values, respectively      |
| cumprod           | Cumulative product of values                               |
| diff              | Compute 1st arithmetic difference (useful for time series) |
| pct_change        | Compute percent changes                                    |

# Correlation and Covariance

Some summary statistics, like correlation and covariance, are computed from pairs of arguments. Let's consider some DataFrames of stock prices and volumes obtained from Yahoo! Finance using the add-on `pandas.read_csv` package:

```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker)
            for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}

price = pd.DataFrame({ticker: data['Adj Close']
                      for ticker, data in all_data.items()})
volume = pd.DataFrame({ticker: data['Volume']
                      for ticker, data in all_data.items()})
```

I now compute percent changes of the prices:

```
In [216]: returns = price.pct_change()
```

```
In [217]: returns.tail()
```

```
Out[217]:
```

| Date       | AAPL      | GOOG      | IBM       | MSFT      |
|------------|-----------|-----------|-----------|-----------|
| 2016-10-17 | -0.000680 | 0.001837  | 0.002072  | -0.003483 |
| 2016-10-18 | -0.000681 | 0.019616  | -0.026168 | 0.007690  |
| 2016-10-19 | -0.002979 | 0.007846  | 0.003583  | -0.002255 |
| 2016-10-20 | -0.000512 | -0.005652 | 0.001719  | -0.004867 |
| 2016-10-21 | -0.003930 | 0.003011  | -0.012474 | 0.042096  |

The `corr` method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series. Relatedly, `cov` computes the covariance:

```
In [218]: returns.MSFT.corr(returns.IBM)
Out[218]: 0.49976361144151144
```

```
In [219]: returns.MSFT.cov(returns.IBM)
Out[219]: 8.8706554797035462e-05
```

DataFrame's `corr` and `cov` methods, on the other hand, return a full correlation

or covariance matrix as a DataFrame, respectively:

```
In [220]: returns.corr()
Out[220]:
          AAPL      GOOG      IBM      MSFT
AAPL    1.000000   0.407919   0.386817   0.389695
GOOG    0.407919   1.000000   0.405099   0.465919
IBM     0.386817   0.405099   1.000000   0.499764
MSFT    0.389695   0.465919   0.499764   1.000000
```

```
In [221]: returns.cov()
Out[221]:
          AAPL      GOOG      IBM      MSFT
AAPL    0.000277   0.000107   0.000078   0.000095
GOOG    0.000107   0.000251   0.000078   0.000108
IBM     0.000078   0.000078   0.000146   0.000089
MSFT    0.000095   0.000108   0.000089   0.000215
```

Using DataFrame's `corrwith` method, you can compute pairwise correlations between a DataFrame's columns or rows with another Series or DataFrame. Passing a Series returns a Series with the correlation value computed for each column:

```
In [222]: returns.corrwith(returns.IBM)
Out[222]:
AAPL      0.386817
GOOG      0.405099
IBM       1.000000
MSFT      0.499764
dtype: float64
```

Passing a DataFrame computes the correlations of matching column names. Here I compute correlations of percent changes with volume:

```
In [223]: returns.corrwith(volume)
Out[223]:
AAPL     -0.075565
GOOG     -0.007067
IBM      -0.204849
MSFT     -0.092950
dtype: float64
```

Passing `axis=1` does things row-wise instead. In all cases, the data points are aligned by label before computing the correlation.

# Unique Values, Value Counts, and Membership

Another class of related methods extracts information about the values contained in a one-dimensional Series. To illustrate these, consider this example:

```
In [224]: obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c'])
```

The first function is `unique`, which gives you an array of the unique values in a Series:

```
In [225]: uniques = obj.unique()
```

```
In [226]: uniques
Out[226]: array(['c', 'a', 'd', 'b'], dtype=object)
```

The unique values are not necessarily returned in sorted order, but could be sorted after the fact if needed (`uniques.sort()`). Relatedly, `value_counts` computes a Series containing value frequencies:

```
In [227]: obj.value_counts()
Out[227]:
c    3
a    3
b    2
d    1
dtype: int64
```

The Series is sorted by value in descending order as a convenience. `value_counts` is also available as a top-level pandas method that can be used with any array or sequence:

```
In [228]: pd.value_counts(obj.values, sort=False)
Out[228]:
b    2
a    3
d    1
c    3
```

```
dtype: int64
```

`isin` performs a vectorized set membership check and can be useful in filtering a data set down to a subset of values in a Series or column in a DataFrame:

```
In [229]: mask = obj.isin(['b', 'c'])
```

```
In [230]: mask
```

```
Out[230]:
```

```
0      True
1     False
2     False
3     False
4     False
5      True
6      True
7      True
8      True
dtype: bool
```

```
In [231]: obj[mask]
```

```
Out[231]:
```

```
0    c
5    b
6    b
7    c
8    c
dtype: object
```

Related to `isin` is the `Index.get_indexer` method, which gives you an index array from an array of possibly non-distinct values into another array of distinct values:

```
In [232]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])
```

```
In [233]: unique_vals = pd.Series(['c', 'b', 'a'])
```

```
In [234]: pd.Index(unique_vals).get_indexer(to_match)
Out[234]: array([0, 2, 1, 1, 0, 2])
```

See [Table 5-11](#) for a reference on these methods.

Table 5-11. Unique, value counts, and set membership methods

| Method       | Description  |
|--------------|--|
| isin         | Compute boolean array indicating whether each Series value is contained in the passed sequence of values.                                      |
| match        | Compute integer indices for each value in an array into another array of distinct values. Helpful for data alignment and join-type operations. |
| unique       | Compute array of unique values in a Series, returned in the order observed.  |
| value_counts | Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order.                        |

In some cases, you may want to compute a histogram on multiple related columns in a DataFrame. Here's an example:

```
In [235]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
.....: 'Qu2': [2, 3, 1, 2, 3],
.....: 'Qu3': [1, 5, 2, 4, 4]})

In [236]: data
Out[236]:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

Passing `pandas.value_counts` to this DataFrame's `apply` function gives:

```
In [237]: result = data.apply(pd.value_counts).fillna(0)

In [238]: result
Out[238]:
   Qu1  Qu2  Qu3
1  1.0  1.0  1.0
2  0.0  2.0  1.0
3  2.0  2.0  0.0
4  2.0  0.0  2.0
5  0.0  0.0  1.0
```

# Moving ahead

In the next chapter, we'll discuss tools for reading (or *loading*) and writing datasets with pandas. After that, we'll dig deeper into data cleaning, wrangling, analysis, and visualization tools using pandas.

# Chapter 6. Data Loading, Storage, and File Formats

Accessing data is a necessary first step for using most of the tools in this book. I'm going to be focused on data input and output using pandas, though there are numerous tools in other libraries to help with reading and writing data in various formats.

Input and output typically falls into a few main categories: reading text files and other more efficient on-disk formats, loading data from databases, and interacting with network sources like web APIs.

# Reading and Writing Data in Text Format

Python is a popular language for text and file munging in part due to its simple syntax for interacting with files, lightweight built-in data structures, and convenient language features like tuple packing and unpacking.

pandas features a number of functions for reading tabular data as a DataFrame object. [Table 6-1](#) has a summary of some of them, though `read_csv` and `read_table` are likely the ones you'll use the most.

Table 6-1. Parsing functions in pandas

| Function                    | Description   |
|-----------------------------|---|
| <code>read_csv</code>       | Load delimited data from a file, URL, or file-like object.<br>Use comma as default delimiter                          |
| <code>read_table</code>     | Load delimited data from a file, URL, or file-like object.<br>Use tab ('\\t') as default delimiter                    |
| <code>read_fwf</code>       | Read data in fixed-width column format (that is, no delimiters)   |
| <code>read_clipboard</code> | Version of <code>read_table</code> that reads data from the clipboard.<br>Useful for converting tables from web pages |
| <code>read_excel</code>     | Read tabular data from an Excel XLS or XLSX file  |
| <code>read_hdf</code>       | Read pandas data from an HDF5 file  |
| <code>read_html</code>      | Read all tables found in the given HTML document  |
| <code>read_json</code>      | Read data from a JSON (JavaScript Object Notation) string representation  |
| <code>read_msgpack</code>   | Read pandas data encoded using the MessagePack binary format  |
| <code>read_pickle</code>    | Read an arbitrary object stored in Python pickle format   |
| <code>read_sas</code>       | Read a SAS dataset stored in one of the SAS system's custom storage formats   |

|                           |   |
|---------------------------|---|
| <code>read_sql</code>     | Read the results of a SQL query (using SQLAlchemy) as a pandas DataFrame. |
| <code>read_stata</code>   | Read a dataset from Stata file format                                     |
| <code>read_feather</code> | Read the Feather binary file format                                       |

I'll give an overview of the mechanics of these functions, which are meant to convert text data into a DataFrame. The optional arguments for these functions may fall into a few categories:

- Indexing: can treat one or more columns as the returned DataFrame, and whether to get column names from the file, the user, or not at all.
- Type inference and data conversion: this includes the user-defined value conversions and custom list of missing value markers.
- Datetime parsing: includes combining capability, including combining date and time information spread over multiple columns into a single column in the result.
- Iterating: support for iterating over chunks of very large files.
- Unclean data issues: skipping rows or a footer, comments, or other minor things like numeric data with thousands separated by commas.

Because of how messy data in the real world can be, some of the data loading functions (especially `read_csv`) have grown very complex in their options over time. It's normal to feel overwhelmed by the number of different parameters (`read_csv` has over 50 as of this writing). The online pandas documentation has many examples about how each of them work, so if you find that you're struggling to read a particular file, you may be able to find a similar enough example to help you find the right parameters.

*Type inference* is one of the more important features of these functions; that means you don't necessarily have to specify which columns are numeric, integer, boolean, or string. Handling dates and other custom types requires a bit more effort, though. Let's start with a small comma-separated (CSV) text file:

```
In [8]: !cat examples/ex1.csv
```

```
a,b,c,d,message  
1,2,3,4,hello  
5,6,7,8,world  
9,10,11,12,foo
```

Since this is comma-delimited, we can use `read_csv` to read it into a `DataFrame`:

```
In [9]: df = pd.read_csv('examples/ex1.csv')
```

```
In [10]: df  
Out[10]:  
      a    b    c    d message  
0    1    2    3    4    hello  
1    5    6    7    8    world  
2    9   10   11   12      foo
```

We could also have used `read_table` and specifying the delimiter:

```
In [11]: pd.read_table('examples/ex1.csv', sep=',')  
Out[11]:  
      a    b    c    d message  
0    1    2    3    4    hello  
1    5    6    7    8    world  
2    9   10   11   12      foo
```

#### Note

Here I used the Unix `cat` shell command to print the raw contents of the file to the screen. If you're on Windows, you can use `type` instead of `cat` to achieve the same effect.

A file will not always have a header row. Consider this file:

```
In [12]: !cat examples/ex2.csv  
1,2,3,4,hello  
5,6,7,8,world  
9,10,11,12,foo
```

To read this file, you have a couple of options. You can allow pandas to assign default column names, or you can specify names yourself:

```
In [13]: pd.read_csv('examples/ex2.csv', header=None)
```

```
Out[13]:  
      0    1    2    3    4  
0    1    2    3    4    hello  
1    5    6    7    8    world  
2    9   10   11   12     foo  
  
In [14]: pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c',  
Out[14]:  
      a    b    c    d  message  
0    1    2    3    4    hello  
1    5    6    7    8    world  
2    9   10   11   12     foo
```

Suppose you wanted the `message` column to be the index of the returned DataFrame. You can either indicate you want the column at index 4 or named '`message`' using the `index_col` argument:

```
In [15]: names = ['a', 'b', 'c', 'd', 'message']  
  
In [16]: pd.read_csv('examples/ex2.csv', names=names, index_col='message')  
Out[16]:  
      a    b    c    d  
message  
hello      1    2    3    4  
world      5    6    7    8  
foo        9   10   11   12
```

In the event that you want to form a hierarchical index from multiple columns, pass a list of column numbers or names:

```
In [17]: !cat examples/csv_mindex.csv  
key1,key2,value1,value2  
one,a,1,2  
one,b,3,4  
one,c,5,6  
one,d,7,8  
two,a,9,10  
two,b,11,12  
two,c,13,14  
two,d,15,16  
  
In [18]: parsed = pd.read_csv('examples/csv_mindex.csv', index_col=[0,1])  
  
In [19]: parsed  
Out[19]:  
      value1  value2
```

```

key1  key2
one   a      1      2
      b      3      4
      c      5      6
      d      7      8
two   a      9     10
      b     11     12
      c     13     14
      d     15     16

```

In some cases, a table might not have a fixed delimiter, using whitespace or some other pattern to separate fields. In these cases, you can pass a regular expression as a delimiter for `read_table`. Consider a text file that looks like this:

```

In [20]: list(open('examples/ex3.txt'))
Out[20]:
['          A          B          C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb  0.927272  0.302904 -0.032399\n',
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382  1.100491']

```

While you could do some munging by hand, in this case fields are separated by a variable amount of whitespace. This can be expressed by the regular expression `\s+`, so we have then:

```

In [21]: result = pd.read_table('examples/ex3.txt', sep='\s+')
In [22]: result
Out[22]:
          A          B          C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491

```

Because there was one fewer column name than the number of data rows, `read_table` infers that the first column should be the DataFrame's index in this special case.

The parser functions have many additional arguments to help you handle the wide variety of exception file formats that occur (see a partial listing in

[Table 6-2](#)). For example, you can skip the first, third, and fourth rows of a file with `skiprows`:

```
In [23]: !cat examples/ex4.csv
# hey!
a,b,c,d,message
# just wanted to make things more difficult for you
# who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
In [24]: pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
Out[24]:
   a    b    c    d  message
0  1    2    3    4    hello
1  5    6    7    8    world
2  9   10   11   12      foo
```

Handling missing values is an important and frequently nuanced part of the file parsing process. Missing data is usually either not present (empty string) or marked by some *sentinel* value. By default, pandas uses a set of commonly occurring sentinels, such as `NA`, `-1.#IND`, and `NULL`:

```
In [25]: !cat examples/ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,,6,8,world
three,9,10,11,12,foo
In [26]: result = pd.read_csv('examples/ex5.csv')

In [27]: result
Out[27]:
   something    a    b      c    d  message
0        one    1    2    3.0    4      NaN
1       two    5    6     NaN    8    world
2      three    9   10   11.0   12      foo

In [28]: pd.isnull(result)
Out[28]:
   something      a      b      c      d  message
0      False    False  False  False  False    True
1      False    False  False   True  False   False
2      False    False  False  False  False   False
```

The `na_values` option can take either a list or set of strings to consider

missing values:

```
In [29]: result = pd.read_csv('examples/ex5.csv', na_values=['\n', '\r'])

In [30]: result
Out[30]:
   something    a    b      c    d  message
0       one     1     2    3.0    4      NaN
1      two     5     6    NaN    8  world
2    three     9    10   11.0   12      foo
```

Different NA sentinels can be specified for each column in a dict:

```
In [31]: sentinels = {'message': ['foo', 'NA'], 'something': ['\n', '\r']}

In [32]: pd.read_csv('examples/ex5.csv', na_values=sentinels)
Out[32]:
   something    a    b      c    d  message
0       one     1     2    3.0    4      NaN
1      NaN     5     6    NaN    8  world
2    three     9    10   11.0   12      NaN
```

Table 6-2. Some `read_csv` / `read_table` function arguments

| Argument                                      | Description   |
|---|---|
| <code>path</code>                             | String indicating filesystem location, URL, or file-like object   |
| <code>sep</code> or<br><code>delimiter</code> | Character sequence or regular expression to use to split fields in each row   |
| <code>header</code>                           | Row number to use as column names. Defaults to 0 (first row), but should be <code>None</code> if there is no header row               |
| <code>index_col</code>                        | Column numbers or names to use as the row index in the result. Can be a single name/number or a list of them for a hierarchical index |
| <code>names</code>                            | List of column names for result, combine with <code>header=None</code>  |
| <code>skiprows</code>                         | Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip  |
| <code>na_values</code>                        | Sequence of values to replace with NA   |
| <code>comment</code>                          | Character or characters to split comments off the end of lines  |
|   | Attempt to parse data to datetime; False by default. If True, will attempt to parse all columns. Otherwise can specify a              |

|                            |  |
|----------------------------|--|
| <code>parse_dates</code>   | list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (for example if date/time split across two columns) |
| <code>keep_date_col</code> | If joining columns to parse date, keep the joined columns.<br>Default False  |
| <code>converters</code>    | Dict containing column number or name mapping to functions. For example <code>{'foo': f}</code> would apply the function <code>f</code> to all values in the <code>'foo'</code> column     |
| <code>dayfirst</code>      | When parsing potentially ambiguous dates, treat as international format (e.g. <code>7/6/2012 -&gt; June 7, 2012</code> ). Default False  |
| <code>date_parser</code>   | Function to use to parse dates   |
| <code>nrows</code>         | Number of rows to read from beginning of file  |
| <code>iterator</code>      | Return a TextParser object for reading file piecemeal  |
| <code>chunksize</code>     | For iteration, size of file chunks   |
| <code>skip_footer</code>   | Number of lines to ignore at end of file   |
| <code>verbose</code>       | Print various parser output information, like the number of missing values placed in non-numeric columns   |
| <code>encoding</code>      | Text encoding for unicode. For example <code>'utf-8'</code> for UTF-8 encoded text   |
| <code>squeeze</code>       | If the parsed data only contains one column return a Series  |
| <code>thousands</code>     | Separator for thousands, e.g. <code>,</code> <code>.</code> or <code>.</code>  |

# Reading Text Files in Pieces

When processing very large files or figuring out the right set of arguments to correctly process a large file, you may only want to read in a small piece of a file or iterate through smaller chunks of the file.

```
In [33]: result = pd.read_csv('examples/ex6.csv')

In [34]: pd.options.display.max_rows = 10 # Make display more

In [35]: result
Out[35]:
      one      two      three      four key
0   0.467976 -0.038649 -0.295344 -1.824726 L
1  -0.358893  1.404453  0.704965 -0.200638 B
2  -0.501840  0.659254 -0.421691 -0.057688 G
3   0.204886  1.074134  1.388361 -0.982404 R
4   0.354628 -0.133116  0.283763 -0.837063 Q
...
9995  2.311896 -0.417070 -1.409599 -0.515821 L
9996 -0.479893 -0.650419  0.745152 -0.646038 E
9997  0.523331  0.787112  0.486066  1.093156 K
9998 -0.362559  0.598894 -1.843201  0.887292 G
9999 -0.096376 -1.012999 -0.657431 -0.573315 O
[10000 rows x 5 columns]
```

If you want to only read out a small number of rows (avoiding reading the entire file), specify that with `nrows`:

```
In [36]: pd.read_csv('examples/ex6.csv', nrows=5)
Out[36]:
      one      two      three      four key
0   0.467976 -0.038649 -0.295344 -1.824726 L
1  -0.358893  1.404453  0.704965 -0.200638 B
2  -0.501840  0.659254 -0.421691 -0.057688 G
3   0.204886  1.074134  1.388361 -0.982404 R
4   0.354628 -0.133116  0.283763 -0.837063 Q
```

To read out a file in pieces, specify a `chunksize` as a number of rows:

```
In [37]: chunker = pd.read_csv('examples/ex6.csv', chunksize=10)
```

```
In [38]: chunker
Out[38]: <pandas.io.parsers.TextFileReader at 0x7f0fccea01d0>
```

The `TextParser` object returned by `read_csv` allows you to iterate over the parts of the file according to the `chunksize`. For example, we can iterate over `ex6.csv`, aggregating the value counts in the 'key' column like so:

```
chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

tot = pd.Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)
```

We have then:

```
In [40]: tot[:10]
Out[40]:
E    368.0
X    364.0
L    346.0
O    343.0
Q    340.0
M    338.0
J    337.0
F    335.0
K    334.0
H    330.0
dtype: float64
```

`TextParser` is also equipped with a `get_chunk` method which enables you to read pieces of an arbitrary size.

# Writing Data Out to Text Format

Data can also be exported to delimited format. Let's consider one of the CSV files read above:

```
In [41]: data = pd.read_csv('examples/ex5.csv')
```

```
In [42]: data
Out[42]:
   something    a    b      c    d  message
0        one    1    2    3.0    4      NaN
1       two    5    6    NaN    8  world
2     three    9   10   11.0   12      foo
```

Using DataFrame's `to_csv` method, we can write the data out to a comma-separated file:

```
In [43]: data.to_csv('examples/out.csv')
```

```
In [44]: !cat examples/out.csv
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

Other delimiters can be used, of course (writing to `sys.stdout` so it prints the text result to the console):

```
In [45]: import sys
```

```
In [46]: data.to_csv(sys.stdout, sep='|')
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6||8|world
2|three|9|10|11.0|12|foo
```

Missing values appear as empty strings in the output. You might want to denote them by some other sentinel value:

```
In [47]: data.to_csv(sys.stdout, na_rep='NULL')
,something,a,b,c,d,message
```

```
0,one,1,2,3.0,4,NULL  
1,two,5,6,NULL,8,world  
2,three,9,10,11.0,12,foo
```

With no other options specified, both the row and column labels are written.  
Both of these can be disabled:

```
In [48]: data.to_csv(sys.stdout, index=False, header=False)  
one,1,2,3.0,4,  
two,5,6,,8,world  
three,9,10,11.0,12,foo
```

You can also write only a subset of the columns, and in an order of your choosing:

```
In [49]: data.to_csv(sys.stdout, index=False, columns=['a', 'b'  
a,b,c  
1,2,3.0  
5,6,  
9,10,11.0
```

Series also has a `to_csv` method:

```
In [50]: dates = pd.date_range('1/1/2000', periods=7)
```

```
In [51]: ts = pd.Series(np.arange(7), index=dates)
```

```
In [52]: ts.to_csv('examples/tseries.csv')
```

```
In [53]: !cat examples/tseries.csv  
2000-01-01,0  
2000-01-02,1  
2000-01-03,2  
2000-01-04,3  
2000-01-05,4  
2000-01-06,5  
2000-01-07,6
```

# Manually Working with Delimited Formats

Most forms of tabular data can be loaded from disk using functions like `pandas.read_table`. In some cases, however, some manual processing may be necessary. It's not uncommon to receive a file with one or more malformed lines that trip up `read_table`. To illustrate the basic tools, consider a small CSV file:

```
In [54]: !cat examples/ex7.csv
"a", "b", "c"
"1", "2", "3"
"1", "2", "3", "4"
```

For any file with a single-character delimiter, you can use Python's built-in `csv` module. To use it, pass any open file or file-like object to `csv.reader`:

```
import csv
f = open('examples/ex7.csv')

reader = csv.reader(f)
```

Iterating through the reader like a file yields tuples of values in each line with any quote characters removed:

```
In [56]: for line in reader:
    ....:     print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3', '4']
```

From there, it's up to you to do the wrangling necessary to put the data in the form that you need it. For example:

```
In [57]: lines = list(csv.reader(open('examples/ex7.csv')))

In [58]: header, values = lines[0], lines[1:]

In [59]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [60]: data_dict
```

```
Out[60]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

CSV files come in many different flavors. Defining a new format with a different delimiter, string quoting convention, or line terminator is done by defining a simple subclass of `csv.Dialect`:

```
class my_dialect(csv.Dialect):
    lineterminator = '\n'
    delimiter = ';'
    quotechar = "'"
    quoting = csv.QUOTE_MINIMAL

reader = csv.reader(f, dialect=my_dialect)
```

Individual CSV dialect parameters can also be given as keywords to `csv.reader` without having to define a subclass:

```
reader = csv.reader(f, delimiter='|')
```

The possible options (attributes of `csv.Dialect`) and what they do can be found in [Table 6-3](#).

Table 6-3. CSV dialect options

| Argument                      | Description   |
|-------------------------------|---|
| <code>delimiter</code>        | One-character string to separate fields. Defaults to <code>' , '</code> .   |
| <code>lineterminator</code>   | Line terminator for writing, defaults to <code>'\r\n'</code> . Reader ignores this and recognizes cross-platform line terminators.  |
| <code>quotechar</code>        | Quote character for fields with special characters (like a delimiter). Default is <code>' ''</code> .   |
| <code>quoting</code>          | Quoting convention. Options include <code>csv.QUOTE_ALL</code> (quote all fields), <code>csv.QUOTE_MINIMAL</code> (only fields with special characters like the delimiter), <code>csv.QUOTE_NONNUMERIC</code> , and <code>csv.QUOTE_NONE</code> (no quoting). See Python's documentation for full details. Defaults to <code>QUOTE_MINIMAL</code> . |
| <code>skipinitialspace</code> | Ignore whitespace after each delimiter. Default <code>False</code> .  |
| <code>doublequote</code>      | How to handle quoting character inside a field. If <code>True</code> , it is doubled. See online documentation for full detail and  |

behavior.

`escapechar`

String to escape the delimiter if `quoting` is set to `csv.QUOTE_NONE`. Disabled by default

**Note**

For files with more complicated or fixed multicharacter delimiters, you will not be able to use the `csv` module. In those cases, you'll have to do the line splitting and other cleanup using string's `split` method or the regular expression method `re.split`.

To *write* delimited files manually, you can use `csv.writer`. It accepts an open, writable file object and the same dialect and format options as `csv.reader`:

```
with open('mydata.csv', 'w') as f:  
    writer = csv.writer(f, dialect=my_dialect)  
    writer.writerow(('one', 'two', 'three'))  
    writer.writerow((1, 2, 3))  
    writer.writerow((4, 5, 6))  
    writer.writerow((7, 8, 9))
```

# JSON Data

JSON (short for JavaScript Object Notation) has become one of the standard formats for sending data by HTTP request between web browsers and other applications. It is a much more free-form data format than a tabular text form like CSV. Here is an example:

```
obj = """
{"name": "Wes",
"places_lived": ["United States", "Spain", "Germany"],
"pet": null,
"siblings": [{"name": "Scott", "age": 29, "pets": ["Zeus", "Zi
    {"name": "Katie", "age": 38,
     "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""

```

JSON is very nearly valid Python code with the exception of its null value `null` and some other nuances (such as disallowing trailing commas at the end of lists). The basic types are objects (dicts), arrays (lists), strings, numbers, booleans, and nulls. All of the keys in an object must be strings. There are several Python libraries for reading and writing JSON data. I'll use `json` here as it is built into the Python standard library. To convert a JSON string to Python form, use `json.loads`:

```
In [62]: import json

In [63]: result = json.loads(obj)

In [64]: result
Out[64]:
{'name': 'Wes',
'pet': None,
'places_lived': ['United States', 'Spain', 'Germany'],
'siblings': [{'age': 29, 'name': 'Scott', 'pets': ['Zeus', 'Zi
{'age': 38, 'name': 'Katie', 'pets': ['Sixes', 'Stache', 'Cis
```

`json.dumps` on the other hand converts a Python object back to JSON:

```
In [65]: asjson = json.dumps(result)
```

How you convert a JSON object or list of objects to a DataFrame or some other data structure for analysis will be up to you. Conveniently, you can pass a list of JSON objects to the DataFrame constructor and select a subset of the data fields:

```
In [66]: siblings = pd.DataFrame(result['siblings'], columns=['name', 'age'])
In [67]: siblings
Out[67]:
   name  age
0  Scott   29
1  Katie   38
```

The `pandas.read_json` can automatically convert JSON datasets in specific arrangements into a Series or DataFrame. For example:

```
In [68]: !cat example.json
cat: example.json: No such file or directory
```

The default options for `pandas.read_json` assume that each object in the JSON array is a row in the table:

```
In [69]: data = pd.read_json('examples/example.json')
In [70]: data
Out[70]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

For an extended example of reading and manipulating JSON data (including nested records), see the USDA Food Database example in the next chapter.

If you need to export data from pandas to JSON, one way is to use the `to_json` methods on Series and DataFrame:

```
In [71]: print(data.to_json())
{"a":{"0":1,"1":4,"2":7}, "b":{"0":2,"1":5,"2":8}, "c":{"0":3,"1":6,"2":9}}
In [72]: print(data.to_json(orient='records'))
[{"a":1,"b":2,"c":3}, {"a":4,"b":5,"c":6}, {"a":7,"b":8,"c":9}]
```

# XML and HTML: Web Scraping

Python has many libraries for reading and writing data in the ubiquitous HTML and XML formats. Examples include lxml (<http://lxml.de>), BeautifulSoup, and html5lib. While lxml is generally much faster than other libraries, the other libraries can better handle malformed HTML or XML files.

pandas has a built-in function, `read_html`, which uses libraries like lxml and BeautifulSoup to automatically parse tables out of HTML files as DataFrame objects. To show how this works, I downloaded an HTML file (used in the pandas documentation) from the United States FDIC government agency showing bank failures<sup>1</sup>. First, you must install some additional libraries used by `read_html`:

```
conda install lxml  
pip install beautifulsoup4 html5lib
```

The `pandas.read_html` function has a number of options, but by default it searches for and attempts to parse all tabular data contained within `<table>` tags. The result is a list of DataFrame objects:

```
In [73]: tables = pd.read_html('examples/fdic_failed_bank_list.html')  
  
In [74]: len(tables)  
Out[74]: 1  
  
In [75]: failures = tables[0]  
  
In [76]: failures.head()  
Out[76]:  
          Bank Name           City  ST CERT \\  
0      Allied Bank       Mulberry  AR   91  
1  The Woodbury Banking Company  Woodbury  GA 11297  
2    First CornerStone Bank  King of Prussia  PA 35312  
3      Trust Company Bank      Memphis  TN  9956  
4    North Milwaukee State Bank      Milwaukee  WI 20364  
          Acquiring Institution      Closing Date  
0            Today's Bank  September 23, 2016  Nov  
1            United Bank     August 19, 2016  Nov  
2  First-Citizens Bank & Trust Company      May 6, 2016  Sep
```

3                   The Bank of Fayette County  
4   First-Citizens Bank & Trust Company

April 29, 2016 Sep  
March 11, 2016

As you will learn in later chapters, from here we could proceed to do some data cleaning and analysis, like computing the number of bank failures by year:

```
In [77]: close_timestamps = pd.to_datetime(failures['Closing Date'])

In [78]: close_timestamps.dt.year.value_counts()
Out[78]:
2010      157
2009      140
2011       92
2012       51
2008       25
...
2004        4
2001        4
2007        3
2003        3
2000        2
Name: Closing Date, Length: 15, dtype: int64
```

In the next section, I will show you how to extract

## Parsing XML with lxml.objectify

XML (extensible markup language) is another common structured data format supporting hierarchical, nested data with metadata. The files that generate the book you are reading actually form a series of large XML documents.

Above, I showed the `pandas.read_html` function which uses either `lxml` or Beautiful Soup under the hood to parse data from HTML. XML and HTML are structurally similar, but XML is more general. Here, I will show an example of how to use `lxml` to parse data from a more general XML format.

The New York Metropolitan Transportation Authority (MTA) publishes a number of data series about its bus and train services

(<http://www.mta.info/developers/download.html>). Here we'll look at the performance data which is contained in a set of XML files. Each train or bus service has a different file (like `Performance_MNR.xml` for the Metro-North

Railroad) containing monthly data as a series of XML records that look like this:

```
<INDICATOR>
  <INDICATOR_SEQ>373889</INDICATOR_SEQ>
  <PARENT_SEQ></PARENT_SEQ>
  <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
  <INDICATOR_NAME>Escalator Availability</INDICATOR_NAME>
  <DESCRIPTION>Percent of the time that escalators are operating systemwide. The availability rate is based on physical observations in the morning of regular business days only. This is a new indicator that began reporting in 2009.</DESCRIPTION>
  <PERIOD_YEAR>2011</PERIOD_YEAR>
  <PERIOD_MONTH>12</PERIOD_MONTH>
  <CATEGORY>Service Indicators</CATEGORY>
  <FREQUENCY>M</FREQUENCY>
  <DESIRED_CHANGE>U</DESIRED_CHANGE>
  <INDICATOR_UNIT>%</INDICATOR_UNIT>
  <DECIMAL_PLACES>1</DECIMAL_PLACES>
  <YTD_TARGET>97.00</YTD_TARGET>
  <YTD_ACTUAL></YTD_ACTUAL>
  <MONTHLY_TARGET>97.00</MONTHLY_TARGET>
  <MONTHLY_ACTUAL></MONTHLY_ACTUAL>
</INDICATOR>
```

Using `lxml.objectify`, we parse the file and get a reference to the root node of the XML file with `getroot`:

```
from lxml import objectify

path = 'examples/mta_perf/Performance_MNR.xml'
parsed = objectify.parse(open(path))
root = parsed.getroot()
```

`root.INDICATOR` return a generator yielding each `<INDICATOR>` XML element. For each record, we can populate a dict of tag names (like `YTD_ACTUAL`) to data values (excluding a few tags):

```
data = []

skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ',
               'DESIRED_CHANGE', 'DECIMAL_PLACES']

for elt in root.INDICATOR:
    el_data = {}
```

```
for child in elt.getchildren():
    if child.tag in skip_fields:
        continue
    el_data[child.tag] = child.pyval
data.append(el_data)
```

Lastly, convert this list of dicts into a DataFrame:

```
In [81]: perf = pd.DataFrame(data)
```

```
In [82]: perf.head()
Out[82]:
Empty DataFrame
Columns: []
Index: []
```

XML data can get much more complicated than this example. Each tag can have metadata, too. Consider an HTML link tag which is also valid XML:

```
from io import StringIO
tag = '<a href="http://www.google.com">Google</a>'
root = objectify.parse(StringIO(tag)).getroot()
```

You can now access any of the fields (like `href`) in the tag or the link text:

```
In [84]: root
Out[84]: <Element a at 0x7f0fc44d9e08>
```

```
In [85]: root.get('href')
Out[85]: 'http://www.google.com'
```

```
In [86]: root.text
Out[86]: 'Google'
```

# Binary Data Formats

One of the easiest ways to store data (also known as *serialization*) efficiently in binary format is using Python’s built-in `pickle` serialization. `pandas` objects all have a `to_pickle` method which writes the data to disk in pickle format:

```
In [87]: frame = pd.read_csv('examples/ex1.csv')

In [88]: frame
Out[88]:
   a    b    c    d  message
0  1    2    3    4    hello
1  5    6    7    8    world
2  9   10   11   12      foo

In [89]: frame.to_pickle('examples/frame_pickle')
```

Any “pickled” object stored in a file can be read by using the built-in `pickle` directly, or even more conveniently using `pandas.read_pickle`:

```
In [90]: pd.read_pickle('examples/frame_pickle')
Out[90]:
   a    b    c    d  message
0  1    2    3    4    hello
1  5    6    7    8    world
2  9   10   11   12      foo
```

## Caution

`pickle` is only recommended as a short-term storage format. The problem is that it is hard to guarantee that the format will be stable over time; an object pickled today may not unpickle with a later version of a library. We have tried to maintain backwards compatibility when possible, but at some point in the future it may be necessary to “break” the pickle format.

`pandas` has built-in support for two more binary data formats: HDF5 and MessagePack. I will give some HDF5 examples in the next section, but I encourage you to explore different file formats to see how fast they are and

how well they work for your analysis. Some other storage formats for pandas or NumPy data include:

- *bcolz* (<http://bcolz.blosc.org/>): a compressable column-oriented binary format based on the Blosc compression library.
- *Feather* (<http://github.com/wesm/feather>): a cross-language column-oriented file format I designed with Hadley Wickham (<http://hadley.nz/>) from the R programming community.

# Using HDF5 Format

HDF5 is a well-regarded file format intended for storing large quantities of scientific array data. It is available as a C library, and it has interfaces available in many other languages like Java, Julia, MATLAB, and Python. The “HDF” in HDF5 stands for *hierarchical data format*. Each HDF5 file can store multiple datasets and supporting metadata. Compared with simpler formats, HDF5 supports on-the-fly compression with a variety of compression modes, enabling data with repeated patterns to be stored more efficiently. For very large datasets that don’t fit into memory, HDF5 can be a good choice for working with large datasets that don’t fit into memory, as you can efficiently read and write small sections of much larger arrays.

While it’s possible to directly access HDF5 files using either the PyTables or h5py libraries, pandas provides a high level interface that simplifies storing Series and DataFrame object. The `HDFStore` class works like a dict and handles the low-level details:

```
In [92]: frame = pd.DataFrame({'a': np.random.randn(100)})  
In [93]: store = pd.HDFStore('mydata.h5')  
In [94]: store['obj1'] = frame  
In [95]: store['obj1_col'] = frame['a']  
  
In [96]: store  
Out[96]:  
<class 'pandas.io.pytables.HDFStore'>  
File path: mydata.h5  
/obj1           frame      (shape->[100, 1])  
/obj1_col       series     (shape->[100])  
/obj2           frame_table (typ->appendable, nrows->100, r  
/obj3           frame_table (typ->appendable, nrows->100, r
```

Objects contained in the HDF5 file can then be retrieved using the same dict-like API:

```
In [97]: store['obj1']
```

```
Out[97]:  
      a  
0   -0.204708  
1    0.478943  
2   -0.519439  
3   -0.555730  
4    1.965781  
..   ...  
95   0.795253  
96   0.118110  
97  -0.748532  
98   0.584970  
99   0.152677  
[100 rows x 1 columns]
```

HDFStore supports two storage schemas, 'fixed' and 'table'. The latter is generally slower, but it supports query operations using a special syntax

```
In [98]: store.put('obj2', frame, format='table')  
  
In [99]: store.select('obj2', where=['index >= 10 and index %lt  
File "<unknown>", line 1  
    (index >=10 and index %lt ;==15 )  
    ^  
SyntaxError: invalid syntax
```

The pandas.read\_hdf function gives you a shortcut to these tools:

```
In [100]: frame.to_hdf('mydata.h5', 'obj3', format='table')  
  
In [101]: pd.read_hdf('mydata.h5', 'obj3', where=['index < 5'])  
Out[101]:  
      a  
0   -0.204708  
1    0.478943  
2   -0.519439  
3   -0.555730  
4    1.965781
```

If you work with huge quantities of data, I would encourage you to explore PyTables and h5py to see how they can suit your needs. Since many data analysis problems are IO-bound (rather than CPU-bound), using a tool like HDF5 can massively accelerate your applications.

**Caution**

HDF5 is *not* a database. It is best suited for write-once, read-many datasets. While data can be added to a file at any time, if multiple writers do so simultaneously, the file can become corrupted.

# Reading Microsoft Excel Files

pandas also supports reading tabular data stored in Excel 2003 (and higher) files using either the `ExcelFile` class or `pandas.read_excel` function. Internally these tools use the add-on packages `xlrd` and `openpyxl` to read XLS and XLSX files, respectively. You may need to install these manually with `pip` or `conda`.

To use `ExcelFile`, create an instance by passing a path to an `xls` or `xlsx` file:

```
In [104]: xlsx = pd.ExcelFile('examples/ex1.xlsx')
```

Data stored in a sheet can then be read into `DataFrame` using `parse`:

```
In [105]: pd.read_excel(xlsx, 'Sheet1')
Out[105]:
      a    b    c    d  message
0     1    2    3    4    hello
1     5    6    7    8   world
2     9   10   11   12      foo
```

If you are reading multiple sheets in a file, then it is faster to create the `ExcelFile`, but you can also pass simply the file name to `pandas.read_excel`:

```
In [106]: frame = pd.read_excel('examples/ex1.xlsx', 'Sheet1')

In [107]: frame
Out[107]:
      a    b    c    d  message
0     1    2    3    4    hello
1     5    6    7    8   world
2     9   10   11   12      foo
```

To write pandas data to Excel format, you must first create an `ExcelWriter`, then write data to it using pandas objects' `to_excel` method:

```
In [108]: writer = pd.ExcelWriter('examples/ex2.xlsx')
```

```
In [109]: frame.to_excel(writer, 'Sheet1')
```

```
In [110]: writer.save()
```

# Interacting with Web APIs

Many websites have public APIs providing data feeds via JSON or some other format. There are a number of ways to access these APIs from Python; one easy-to-use method that I recommend is the `requests` package (<http://docs.python-requests.org>). To find the last 30 GitHub issues for pandas on GitHub, we can make a GET HTTP request using the add-on `requests` library::

```
In [112]: import requests  
  
In [113]: url = 'https://api.github.com/repos/pandas-dev/pandas'  
  
In [114]: resp = requests.get(url)  
  
In [115]: resp  
Out[115]: <Response [200]>
```

The Response object's `json` method will return a dictionary containing JSON parsed into native Python objects.

```
In [116]: data = resp.json()  
  
In [117]: data[0]['title']  
Out[117]: 'DataFrame apply results in a DataFrame when lambda :
```

Each element in `data` is a dictionary containing all of the data found on a GitHub issue page (except for the comments). We can pass `data` directly to `DataFrame` and extract fields of interest:

```
In [118]: issues = pd.DataFrame(data, columns=['number', 'title'])  
  
In [119]: issues  
Out[119]:  
   number                      title  
0    16353  DataFrame apply results in a DataFrame when la...  
1    16351  ENH: Improve error message for header argument...  
2    16350              DOC: dev docs authentication failing  
3    16349          Using .assign() with MultiIndexed Columns  
4    16346            PERF: improve MultiIndex get_loc performance
```

```
..          ...
25    16306           some warnings not being silenced
26    16304 Index order of joined frame does not respect t...
27    16303           DEPS: Drop Python 3.4 support
28    16301           ENH: Support fspath protocol
29    16300 Wrong error message in HDFStore.append when ap...
                           labels state
0                               []  open
1                               []  open
2  [{id': 134699, 'url': 'https://api.github.com...}  open
3                               []  open
4  [{id': 8935311, 'url': 'https://api.github.co...}  open
..
25  [{id': 42670965, 'url': 'https://api.github.c...}  open
26  [{id': 13098779, 'url': 'https://api.github.c...}  open
27  [{id': 527603109, 'url': 'https://api.github....}  open
28  [{id': 76865106, 'url': 'https://api.github.c...}  open
29  [{id': 29648920, 'url': 'https://api.github.c...}  open
[30 rows x 4 columns]
```

With a bit of elbow grease, you can create some higher-level interfaces to common web APIs that return DataFrame objects for easy analysis.

# Interacting with Databases

In a business setting, most data may not be stored in text or Excel files. SQL-based relational databases (such as SQL Server, PostgreSQL, and MySQL) are in wide use, and many alternative databases have become quite popular. The choice of database is usually dependent on the performance, data integrity, and scalability needs of an application.

Loading data from SQL into a DataFrame is fairly straightforward, and pandas has some functions to simplify the process. As an example, I'll create a SQLite database using Python's built-in `sqlite3` driver:

```
import sqlite3

query = """
CREATE TABLE test
(a VARCHAR(20), b VARCHAR(20),
 c REAL,           d INTEGER
);"""

con = sqlite3.connect('mydata.sqlite')
con.execute(query)
con.commit()
```

Then, insert a few rows of data:

```
data = [('Atlanta', 'Georgia', 1.25, 6),
        ('Tallahassee', 'Florida', 2.6, 3),
        ('Sacramento', 'California', 1.7, 5)]
stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

con.executemany(stmt, data)
con.commit()
```

Most Python SQL drivers (PyODBC, psycopg2, MySQLdb, pymssql, etc.) return a list of tuples when selecting data from a table:

```
In [122]: cursor = con.execute('select * from test')
```

```
In [123]: rows = cursor.fetchall()
```

```
In [124]: rows
Out[124]:
[('Atlanta', 'Georgia', 1.25, 6),
 ('Tallahassee', 'Florida', 2.6, 3),
 ('Sacramento', 'California', 1.7, 5)]
```

You can pass the list of tuples to the DataFrame constructor, but you also need the column names, contained in the cursor's `description` attribute:

```
In [125]: cursor.description
Out[125]:
([('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))]

In [126]: pd.DataFrame(rows, columns=[x[0] for x in cursor.des
Out[126]:
      a          b    c   d
0  Atlanta    Georgia  1.25  6
1 Tallahassee  Florida  2.60  3
2 Sacramento California  1.70  5
```

This is quite a bit of munging that you'd rather not repeat each time you query the database. The SQLAlchemy project (<http://www.sqlalchemy.org/>) is a popular Python SQL toolkit that abstracts away many of the common differences between SQL databases. pandas has a `read_sql` function that enables you to read data easily from a general SQLAlchemy connection. Here, we'll connect to the same SQLite database with SQLAlchemy and read data from the table created above:

```
In [127]: import sqlalchemy as sqla

In [128]: db = sqla.create_engine('sqlite:///mydata.sqlite')

In [129]: pd.read_sql('select * from test', db)
Out[129]:
      a          b    c   d
0  Atlanta    Georgia  1.25  6
1 Tallahassee  Florida  2.60  3
2 Sacramento California  1.70  5
```

<sup>1</sup> <https://www.fdic.gov/bank/individual/failed/banklist.html>.

# Chapter 7. Data Cleaning and Preparation

A significant amount of time in data analysis and modeling is spent on data preparation: loading, cleaning, transforming, and rearranging. Such tasks are often reported to take up 80% or more of your coding time. Sometimes the way that data is stored in files or databases is not in the right format for a particular task. Many people choose to do ad hoc processing of data from one form to another using a general purpose programming, like Python, Perl, R, or Java, or UNIX text processing tools like sed or awk. Fortunately, pandas along with the built-in Python language features provide you with a high-level, flexible, and fast set of tools to enable you to manipulate data into the right form.

If you identify a type of data manipulation that isn't anywhere in this book or elsewhere in the pandas library, feel free to share your use case on one of the Python mailing lists or on pandas's GitHub site. Indeed, much of the design and implementation of pandas has been driven by the needs of real world applications.

In this chapter I discuss tools for missing data, duplicate data, string manipulation, and some other analytical data transformations. In the next chapter, I focus on combining and rearranging datasets in various ways.

# Handling Missing Data

Missing data occurs commonly in many data analysis applications. One of the goals of pandas is to make working with missing data as painless as possible. For example, all of the descriptive statistics on pandas objects exclude missing data by default.

The way that missing data is represented in pandas object is somewhat imperfect, but it is functional for a lot of usres. For numeric data, pandas uses the floating point value `NaN` (Not a Number) to represent missing data. We call this a *sentinel value* that can be easily detected:

```
In [10]: string_data = pd.Series(['aardvark', 'artichoke', np.nan])  
  
In [11]: string_data  
Out[11]:  
0    aardvark  
1    artichoke  
2        NaN  
3    avocado  
dtype: object  
  
In [12]: string_data.isnull()  
Out[12]:  
0    False  
1    False  
2     True  
3    False  
dtype: bool
```

In pandas, we've adopted a convention used in the R programming language by referring to missing data as “NA”, which stands for *not available*. In statistics applications, NA data may either be data that does not exist or that exists but was not observed (through problems with data collection, for example). When cleaning up data for analysis, it is often important to do analysis on the missing data itself to identify data collection problems or potential biases in the data caused by missing data.

The built-in Python `None` value is also treated as NA in object arrays:

```
In [13]: string_data[0] = None  
  
In [14]: string_data.isnull()  
Out[14]:  
0      True  
1     False  
2      True  
3     False  
dtype: bool
```

There is work ongoing in the pandas project to improve the internal details of how missing data is handled, but the user API functions, like `pandas.isnull`, abstract away many of the annoying details.

Table 7-1. NA handling methods

| Argument             | Description   |
|----------------------|---|
| <code>dropna</code>  | Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate. |
| <code>fillna</code>  | Fill in missing data with some value or using an interpolation method such as ' <code>ffill</code> ' or ' <code>bfill</code> '.             |
| <code>isnull</code>  | Return like-type object containing boolean values indicating which values are missing / NA.   |
| <code>notnull</code> | Negation of <code>isnull</code> .   |

# Filtering Out Missing Data

There are a few ways to filter out missing data. While doing it by hand using `pandas.isnull` and boolean indexing is always an option, the `dropna` can be helpful. On a Series, it returns the Series with only the non-null data and index values:

```
In [15]: from numpy import nan as NA  
  
In [16]: data = pd.Series([1, NA, 3.5, NA, 7])  
  
In [17]: data.dropna()  
Out[17]:  
0    1.0  
2    3.5  
4    7.0  
dtype: float64
```

This is equivalent to:

```
In [18]: data[data.notnull()]  
Out[18]:  
0    1.0  
2    3.5  
4    7.0  
dtype: float64
```

With DataFrame objects, things are a bit more complex. You may want to drop rows or columns which are all NA or only those containing any NAs. `dropna` by default drops any row containing a missing value:

```
In [19]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],  
...:                   [NA, NA, NA], [NA, 6.5, 3.]])  
  
In [20]: cleaned = data.dropna()  
  
In [21]: data  
Out[21]:  
      0    1    2  
0  1.0  6.5  3.0  
1  1.0  NaN  NaN
```

```
2    NaN    NaN    NaN  
3    NaN    6.5    3.0
```

```
In [22]: cleaned  
Out[22]:  
      0    1    2  
0  1.0  6.5  3.0
```

Passing `how='all'` will only drop rows that are all NA:

```
In [23]: data.dropna(how='all')  
Out[23]:  
      0    1    2  
0  1.0  6.5  3.0  
1  1.0  NaN  NaN  
3  NaN  6.5  3.0
```

To drop columns in the same way, pass `axis=1`:

```
In [24]: data[4] = NA
```

```
In [25]: data  
Out[25]:  
      0    1    2    4  
0  1.0  6.5  3.0  NaN  
1  1.0  NaN  NaN  NaN  
2  NaN  NaN  NaN  NaN  
3  NaN  6.5  3.0  NaN
```

```
In [26]: data.dropna(axis=1, how='all')  
Out[26]:  
      0    1    2  
0  1.0  6.5  3.0  
1  1.0  NaN  NaN  
2  NaN  NaN  NaN  
3  NaN  6.5  3.0
```

A related way to filter out DataFrame rows tends to concern time series data. Suppose you want to keep only rows containing a certain number of observations. You can indicate this with the `thresh` argument:

```
In [27]: df = pd.DataFrame(np.random.randn(7, 3))  
In [28]: df.iloc[:4, 1] = NA; df.iloc[:2, 2] = NA
```

```
In [29]: df
```

```
Out[29]:
```

|   | 0         | 1         | 2         |
|---|-----------|-----------|-----------|
| 0 | -0.204708 | NaN       | NaN       |
| 1 | -0.555730 | NaN       | NaN       |
| 2 | 0.092908  | NaN       | 0.769023  |
| 3 | 1.246435  | NaN       | -1.296221 |
| 4 | 0.274992  | 0.228913  | 1.352917  |
| 5 | 0.886429  | -2.001637 | -0.371843 |
| 6 | 1.669025  | -0.438570 | -0.539741 |

```
In [30]: df.dropna(thresh=3)
```

```
Out[30]:
```

|   | 0        | 1         | 2         |
|---|----------|-----------|-----------|
| 4 | 0.274992 | 0.228913  | 1.352917  |
| 5 | 0.886429 | -2.001637 | -0.371843 |
| 6 | 1.669025 | -0.438570 | -0.539741 |

# Filling in Missing Data

Rather than filtering out missing data (and potentially discarding other data along with it), you may want to fill in the “holes” in any number of ways. For most purposes, the `fillna` method is the workhorse function to use. Calling `fillna` with a constant replaces missing values with that value:

```
In [31]: df.fillna(0)
Out[31]:
      0          1          2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

Calling `fillna` with a dict you can use a different fill value for each column:

```
In [32]: df.fillna({1: 0.5, 3: -1, 2: 0})
Out[32]:
      0          1          2
0 -0.204708  0.500000  0.000000
1 -0.555730  0.500000  0.000000
2  0.092908  0.500000  0.769023
3  1.246435  0.500000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

`fillna` returns a new object, but you can modify the existing object in place:

```
# always returns a reference to the filled object
In [33]: _ = df.fillna(0, inplace=True)

In [34]: df
Out[34]:
      0          1          2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
```

```
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

The same interpolation methods available for reindexing can be used with `fillna`:

```
In [35]: df = pd.DataFrame(np.random.randn(6, 3))
```

```
In [36]: df.iloc[2:, 1] = NA; df.iloc[4:, 2] = NA
```

```
In [37]: df
```

```
Out[37]:
```

|   | 0         | 1        | 2         |
|---|-----------|----------|-----------|
| 0 | 0.476985  | 3.248944 | -1.021228 |
| 1 | -0.577087 | 0.124121 | 0.302614  |
| 2 | 0.523772  | NaN      | 1.343810  |
| 3 | -0.713544 | NaN      | -2.370232 |
| 4 | -1.860761 | NaN      | NaN       |
| 5 | -1.265934 | NaN      | NaN       |

```
In [38]: df.fillna(method='ffill')
```

```
Out[38]:
```

|   | 0         | 1        | 2         |
|---|-----------|----------|-----------|
| 0 | 0.476985  | 3.248944 | -1.021228 |
| 1 | -0.577087 | 0.124121 | 0.302614  |
| 2 | 0.523772  | 0.124121 | 1.343810  |
| 3 | -0.713544 | 0.124121 | -2.370232 |
| 4 | -1.860761 | 0.124121 | -2.370232 |
| 5 | -1.265934 | 0.124121 | -2.370232 |

```
In [39]: df.fillna(method='ffill', limit=2)
```

```
Out[39]:
```

|   | 0         | 1        | 2         |
|---|-----------|----------|-----------|
| 0 | 0.476985  | 3.248944 | -1.021228 |
| 1 | -0.577087 | 0.124121 | 0.302614  |
| 2 | 0.523772  | 0.124121 | 1.343810  |
| 3 | -0.713544 | 0.124121 | -2.370232 |
| 4 | -1.860761 | NaN      | -2.370232 |
| 5 | -1.265934 | NaN      | -2.370232 |

With `fillna` you can do lots of other things with a little creativity. For example, you might pass the mean or median value of a Series:

```
In [40]: data = pd.Series([1., NA, 3.5, NA, 7])
```

```
In [41]: data.fillna(data.mean())
Out[41]:
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

See [Table 7-2](#) for a reference on `fillna`.

Table 7-2. `fillna` function arguments

| Argument             | Description   |
|----------------------|---|
| <code>value</code>   | Scalar value or dict-like object to use to fill missing values                              |
| <code>method</code>  | Interpolation, by default ' <code>ffill</code> ' if function called with no other arguments |
| <code>axis</code>    | Axis to fill on, default <code>axis=0</code>  |
| <code>inplace</code> | Modify the calling object without producing a copy  |
| <code>limit</code>   | For forward and backward filling, maximum number of consecutive periods to fill             |

# Data Transformation

So far in this chapter we've been concerned with rearranging data. Filtering, cleaning, and other tranformations are another class of important operations.

# Removing Duplicates

Duplicate rows may be found in a DataFrame for any number of reasons. Here is an example:

```
In [42]: data = pd.DataFrame({'k1': ['one'] * 3 + ['two'] * 4,
.....
      'k2': [1, 1, 2, 3, 3, 4, 4]})
```

```
In [43]: data
Out[43]:
   k1    k2
0  one    1
1  one    1
2  one    2
3  two    3
4  two    3
5  two    4
6  two    4
```

The DataFrame method `duplicated` returns a boolean Series indicating whether each row is a duplicate or not:

```
In [44]: data.duplicated()
Out[44]:
0    False
1     True
2    False
3    False
4     True
5    False
6     True
dtype: bool
```

Relatedly, `drop_duplicates` returns a DataFrame where the `duplicated` array is `False`:

```
In [45]: data.drop_duplicates()
Out[45]:
   k1    k2
0  one    1
2  one    2
3  two    3
```

```
5    two    4
```

Both of these methods by default consider all of the columns; alternatively you can specify any subset of them to detect duplicates. Suppose we had an additional column of values and wanted to filter duplicates only based on the 'k1' column:

```
In [46]: data['v1'] = range(7)
```

```
In [47]: data.drop_duplicates(['k1'])
```

```
Out[47]:
```

|   | k1  | k2 | v1 |
|---|-----|----|----|
| 0 | one | 1  | 0  |
| 3 | two | 3  | 3  |

duplicated and drop\_duplicates by default keep the first observed value combination. Passing keep='last' will return the last one:

```
In [48]: data.drop_duplicates(['k1', 'k2'], keep='last')
```

```
Out[48]:
```

|   | k1  | k2 | v1 |
|---|-----|----|----|
| 1 | one | 1  | 1  |
| 2 | one | 2  | 2  |
| 4 | two | 3  | 4  |
| 6 | two | 4  | 6  |

# Transforming Data Using a Function or Mapping

For many data sets, you may wish to perform some transformation based on the values in an array, Series, or column in a DataFrame. Consider the following hypothetical data collected about some kinds of meat:

```
In [49]: data = pd.DataFrame({'food': ['bacon', 'pulled pork',
....:                               'corned beef', 'Bacon',
....:                               'nova lox'],
....: 'ounces': [4, 3, 12, 6, 7.5, 8, :]
```

```
In [50]: data
Out[50]:
```

|   | food        | ounces |
|---|-------------|--------|
| 0 | bacon       | 4.0    |
| 1 | pulled pork | 3.0    |
| 2 | bacon       | 12.0   |
| 3 | Pastrami    | 6.0    |
| 4 | corned beef | 7.5    |
| 5 | Bacon       | 8.0    |
| 6 | pastrami    | 3.0    |
| 7 | honey ham   | 5.0    |
| 8 | nova lox    | 6.0    |

Suppose you wanted to add a column indicating the type of animal that each food came from. Let's write down a mapping of each distinct meat type to the kind of animal:

```
meat_to_animal = {
    'bacon': 'pig',
    'pulled pork': 'pig',
    'pastrami': 'cow',
    'corned beef': 'cow',
    'honey ham': 'pig',
    'nova lox': 'salmon'
}
```

The `map` method on a Series accepts a function or dict-like object containing a mapping, but here we have a small problem in that some of the meats above

are capitalized and others are not. Thus, we first need to convert each value to lower case:

```
In [52]: data['animal'] = data['food'].str.lower().map(meat_to_
```

```
In [53]: data
```

```
Out[53]:
```

|   | food        | ounces | animal |
|---|-------------|--------|--------|
| 0 | bacon       | 4.0    | pig    |
| 1 | pulled pork | 3.0    | pig    |
| 2 | bacon       | 12.0   | pig    |
| 3 | Pastrami    | 6.0    | cow    |
| 4 | corned beef | 7.5    | cow    |
| 5 | Bacon       | 8.0    | pig    |
| 6 | pastrami    | 3.0    | cow    |
| 7 | honey ham   | 5.0    | pig    |
| 8 | nova lox    | 6.0    | salmon |

We could also have passed a function that does all the work:

```
In [54]: data['food'].map(lambda x: meat_to_animal[x.lower()])
```

```
Out[54]:
```

|   |        |
|---|--------|
| 0 | pig    |
| 1 | pig    |
| 2 | pig    |
| 3 | cow    |
| 4 | cow    |
| 5 | pig    |
| 6 | cow    |
| 7 | pig    |
| 8 | salmon |

```
Name: food, dtype: object
```

Using `map` is a convenient way to perform element-wise transformations and other data cleaning-related operations.

# Replacing Values

Filling in missing data with the `fillna` method is a special case of more general value replacement. While `map`, as you've seen above, can be used to modify a subset of values in an object, `replace` provides a simpler and more flexible way to do so. Let's consider this Series:

```
In [55]: data = pd.Series([1., -999., 2., -999., -1000., 3.])  
  
In [56]: data  
Out[56]:  
0      1.0  
1     -999.0  
2      2.0  
3     -999.0  
4    -1000.0  
5      3.0  
dtype: float64
```

The `-999` values might be sentinel values for missing data. To replace these with NA values that pandas understands, we can use `replace`, producing a new Series:

```
In [57]: data.replace(-999, np.nan)  
Out[57]:  
0      1.0  
1      NaN  
2      2.0  
3      NaN  
4    -1000.0  
5      3.0  
dtype: float64
```

If you want to replace multiple values at once, you instead pass a list then the substitute value:

```
In [58]: data.replace([-999, -1000], np.nan)  
Out[58]:  
0      1.0  
1      NaN  
2      2.0
```

```
3      NaN  
4      NaN  
5      3.0  
dtype: float64
```

To use a different replacement for each value, pass a list of substitutes:

```
In [59]: data.replace([-999, -1000], [np.nan, 0])  
Out[59]:  
0    1.0  
1    NaN  
2    2.0  
3    NaN  
4    0.0  
5    3.0  
dtype: float64
```

The argument passed can also be a dict:

```
In [60]: data.replace({-999: np.nan, -1000: 0})  
Out[60]:  
0    1.0  
1    NaN  
2    2.0  
3    NaN  
4    0.0  
5    3.0  
dtype: float64
```

# Renaming Axis Indexes

Like values in a Series, axis labels can be similarly transformed by a function or mapping of some form to produce new, differently labeled objects. The axes can also be modified in place without creating a new data structure. Here's a simple example:

```
In [61]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),  
....:                      index=['Ohio', 'Colorado', 'New York'],  
....:                      columns=['one', 'two', 'three', 'four'])
```

Like a Series, the axis indexes have a `map` method:

```
In [62]: transform = lambda x: x[:4].upper()
```

```
In [63]: data.index.map(transform)  
Out[63]: Index(['OHIO', 'COLO', 'NEW YORK'], dtype='object')
```

You can assign to `index`, modifying the DataFrame in place:

```
In [64]: data.index = data.index.map(transform)
```

```
In [65]: data  
Out[65]:
```

|      | one | two | three | four |
|------|-----|-----|-------|------|
| OHIO | 0   | 1   | 2     | 3    |
| COLO | 4   | 5   | 6     | 7    |
| NEW  | 8   | 9   | 10    | 11   |

If you want to create a transformed version of a data set without modifying the original, a useful method is `rename`:

```
In [66]: data.rename(index=str.title, columns=str.upper)  
Out[66]:
```

|      | ONE | TWO | THREE | FOUR |
|------|-----|-----|-------|------|
| Ohio | 0   | 1   | 2     | 3    |
| Colo | 4   | 5   | 6     | 7    |
| New  | 8   | 9   | 10    | 11   |

Notably, `rename` can be used in conjunction with a dict-like object providing new values for a subset of the axis labels:

```
In [67]: data.rename(index={'OHIO': 'INDIANA'},  
....:  
      columns={'three': 'peekaboo'})  
Out[67]:  
      one  two  peekaboo  four  
INDIANA    0    1          2    3  
COLO       4    5          6    7  
NEW        8    9          10   11
```

`rename` saves having to copy the DataFrame manually and assign to its `index` and `columns` attributes. Should you wish to modify a data set in place, pass `inplace=True`:

```
In [68]: data.rename(index={'OHIO': 'INDIANA'}, inplace=True)  
  
In [69]: data  
Out[69]:  
      one  two  three  four  
INDIANA    0    1      2    3  
COLO       4    5      6    7  
NEW        8    9      10   11
```

# Discretization and Binning

Continuous data is often discretized or otherwise separated into “bins” for analysis. Suppose you have data about a group of people in a study, and you want to group them into discrete age buckets:

```
In [70]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Let’s divide these into bins of 18 to 25, 26 to 35, 36 to 60, and finally 61 and older. To do so, you have to use `cut`, a function in pandas:

```
In [71]: bins = [18, 25, 35, 60, 100]
```

```
In [72]: cats = pd.cut(ages, bins)
```

```
In [73]: cats
Out[73]:
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60]
```

The object pandas returns is a special `Categorical` object. You can treat it like an array of strings indicating the bin name; internally it contains a `categories` array indicating the distinct category names along with a labeling for the `ages` data in the `codes` attribute:

```
In [74]: cats.codes
Out[74]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
```

```
In [75]: cats.categories
Out[75]:
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100])
              closed='right',
              dtype='interval[int64]')
```

```
In [76]: pd.value_counts(cats)
```

```
Out[76]:
(18, 25]      5
(35, 60]      3
(25, 35]      3
(60, 100]     1
```

```
dtype: int64
```

Consistent with mathematical notation for intervals, a parenthesis means that the side is *open* while the square bracket means it is *closed* (inclusive). Which side is closed can be changed by passing `right=False`:

```
In [77]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)
Out[77]:
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36),
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61)
```

You can also pass your own bin names by passing a list or array to the `labels` option:

```
In [78]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
In [79]: pd.cut(ages, bins, labels=group_names)
Out[79]:
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior,
Length: 12
Categories (4, object): [MiddleAged < Senior < YoungAdult < Youth]
```

If you pass `cut` a integer number of bins instead of explicit bin edges, it will compute equal-length bins based on the minimum and maximum values in the data. Consider the case of some uniformly distributed data chopped into fourths:

```
In [80]: data = np.random.rand(20)

In [81]: pd.cut(data, 4, precision=2)
Out[81]:
[(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.97], (0.34, 0.55),
Length: 20
Categories (4, interval[float64]): [(0.12, 0.34] < (0.34, 0.55]
```

A closely related function, `qcut`, bins the data based on sample quantiles. Depending on the distribution of the data, using `cut` will not usually result in each bin having the same number of data points. Since `qcut` uses sample quantiles instead, by definition you will obtain roughly equal-size bins:

```
In [82]: data = np.random.randn(1000) # Normally distributed
```

```
In [83]: cats = pd.qcut(data, 4) # Cut into quartiles

In [84]: cats
Out[84]:
[(-0.0265, 0.62], (0.62, 3.928], (-0.68, -0.0265], (0.62, 3.928]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.68] < (-0.68, -(0.62, 3.928]]

In [85]: pd.value_counts(cats)
Out[85]:
(0.62, 3.928]      250
(-0.0265, 0.62]    250
(-0.68, -0.0265]   250
(-2.95, -0.68]     250
dtype: int64
```

Similar to `cut` you can pass your own quantiles (numbers between 0 and 1, inclusive):

```
In [86]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
Out[86]:
[(-0.0265, 1.286], (-0.0265, 1.286], (-1.187, -0.0265], (-0.0265, 1.286]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -1.187] < (-1.187, (1.286, 3.928]]
```

We'll return to `cut` and `qcut` later in the chapter on aggregation and group operations, as these discretization functions are especially useful for quantile and group analysis.

# Detecting and Filtering Outliers

Filtering or transforming outliers is largely a matter of applying array operations. Consider a DataFrame with some normally distributed data:

```
In [87]: data = pd.DataFrame(np.random.randn(1000, 4))
```

```
In [88]: data.describe()
```

```
Out[88]:
```

|       | 0           | 1           | 2           | 3           |
|-------|-------------|-------------|-------------|-------------|
| count | 1000.000000 | 1000.000000 | 1000.000000 | 1000.000000 |
| mean  | 0.049091    | 0.026112    | -0.002544   | -0.051827   |
| std   | 0.996947    | 1.007458    | 0.995232    | 0.998311    |
| min   | -3.645860   | -3.184377   | -3.745356   | -3.428254   |
| 25%   | -0.599807   | -0.612162   | -0.687373   | -0.747478   |
| 50%   | 0.047101    | -0.013609   | -0.022158   | -0.088274   |
| 75%   | 0.756646    | 0.695298    | 0.699046    | 0.623331    |
| max   | 2.653656    | 3.525865    | 2.735527    | 3.366626    |

Suppose you wanted to find values in one of the columns exceeding three in magnitude:

```
In [89]: col = data[3]
```

```
In [90]: col[np.abs(col) > 3]
```

```
Out[90]:
```

```
258    -3.428254  
635     3.366626  
Name: 3, dtype: float64
```

To select all rows having a value exceeding 3 or -3, you can use the `any` method on a boolean DataFrame:

```
In [91]: data[(np.abs(data) > 3).any(1)]
```

```
Out[91]:
```

|     | 0         | 1         | 2         | 3         |
|-----|-----------|-----------|-----------|-----------|
| 41  | 0.457246  | -0.025907 | -3.399312 | -0.974657 |
| 60  | 1.951312  | 3.260383  | 0.963301  | 1.201206  |
| 136 | 0.508391  | -0.196713 | -3.745356 | -1.520113 |
| 235 | -0.242459 | -3.056990 | 1.918403  | -0.578828 |
| 258 | 0.682841  | 0.326045  | 0.425384  | -3.428254 |
| 322 | 1.179227  | -3.184377 | 1.369891  | -1.074833 |

```
544 -3.548824  1.553205 -2.186301  1.277104
635 -0.578093  0.193299  1.397822  3.366626
782 -0.207434  3.525865  0.283070  0.544635
803 -3.645860  0.255475 -0.549574 -1.907459
```

Values can be set based on these criteria. Here is code to cap values outside the interval -3 to 3:

```
In [92]: data[np.abs(data) > 3] = np.sign(data) * 3
```

```
In [93]: data.describe()
```

```
Out[93]:
```

|       | 0           | 1           | 2           | 3           |
|-------|-------------|-------------|-------------|-------------|
| count | 1000.000000 | 1000.000000 | 1000.000000 | 1000.000000 |
| mean  | 0.050286    | 0.025567    | -0.001399   | -0.051765   |
| std   | 0.992920    | 1.004214    | 0.991414    | 0.995761    |
| min   | -3.000000   | -3.000000   | -3.000000   | -3.000000   |
| 25%   | -0.599807   | -0.612162   | -0.687373   | -0.747478   |
| 50%   | 0.047101    | -0.013609   | -0.022158   | -0.088274   |
| 75%   | 0.756646    | 0.695298    | 0.699046    | 0.623331    |
| max   | 2.653656    | 3.000000    | 2.735527    | 3.000000    |

The ufunc `np.sign` returns an array of 1 and -1 depending on the sign of the values.

# Permutation and Random Sampling

Permuting (randomly reordering) a Series or the rows in a DataFrame is easy to do using the `numpy.random.permutation` function. Calling `permutation` with the length of the axis you want to permute produces an array of integers indicating the new ordering:

```
In [94]: df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))  
In [95]: sampler = np.random.permutation(5)  
In [96]: sampler  
Out[96]: array([3, 1, 4, 2, 0])
```

That array can then be used in ix-based indexing or the `take` function:

```
In [97]: df  
Out[97]:  
   0   1   2   3  
0   0   1   2   3  
1   4   5   6   7  
2   8   9  10  11  
3  12  13  14  15  
4  16  17  18  19  
  
In [98]: df.take(sampler)  
Out[98]:  
   0   1   2   3  
3  12  13  14  15  
1   4   5   6   7  
4  16  17  18  19  
2   8   9  10  11  
0   0   1   2   3
```

To select a random subset without replacement, one way is to slice off the first `k` elements of the array returned by `permutation`, where `k` is the desired subset size. There are much more efficient sampling-without-replacement algorithms, but this is an easy strategy that uses readily available tools:

```
In [99]: df.take(np.random.permutation(len(df))[:3])  
Out[99]:
```

```
    0    1    2    3  
3  12   13   14   15  
4  16   17   18   19  
2    8    9   10   11
```

To generate a sample *with* replacement, the fastest way is to use `np.random.randint` to draw random integers:

```
In [100]: bag = np.array([5, 7, -1, 6, 4])  
  
In [101]: sampler = np.random.randint(0, len(bag), size=10)  
  
In [102]: sampler  
Out[102]: array([4, 1, 4, 2, 0, 3, 1, 4, 0, 4])  
  
In [103]: draws = bag.take(sampler)  
  
In [104]: draws  
Out[104]: array([ 4,  7,  4, -1,  5,  6,  7,  4,  5,  4])
```

# Computing Indicator/Dummy Variables

Another type of transformation for statistical modeling or machine learning applications is converting a categorical variable into a “dummy” or “indicator” matrix. If a column in a DataFrame has  $k$  distinct values, you would derive a matrix or DataFrame containing  $k$  columns containing all 1’s and 0’s. pandas has a `get_dummies` function for doing this, though devising one yourself is not difficult. Let’s return to an earlier example DataFrame:

```
In [105]: df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a',
.....:                               'data1': range(6))}

In [106]: pd.get_dummies(df['key'])
Out[106]:
   a   b   c
0  0   1   0
1  0   1   0
2  1   0   0
3  0   0   1
4  1   0   0
5  0   1   0
```

In some cases, you may want to add a prefix to the columns in the indicator DataFrame, which can then be merged with the other data. `get_dummies` has a `prefix` argument for doing this:

```
In [107]: dummies = pd.get_dummies(df['key'], prefix='key')

In [108]: df_with_dummy = df[['data1']].join(dummies)

In [109]: df_with_dummy
Out[109]:
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

If a row in a DataFrame belongs to multiple categories, things are a bit more

complicated. Let's return to the MovieLens 1M dataset from earlier in the book:

```
In [110]: mnames = ['movie_id', 'title', 'genres']

In [111]: movies = pd.read_table('datasets/movielens/movies.dat'
.....:           names=mnames)

In [112]: movies[:10]
Out[112]:
   movie_id          title
0         1    Toy Story (1995)  Animation|Chi...
1         2        Jumanji (1995) Adventure|Chi...
2         3  Grumpier Old Men (1995)
3         4      Waiting to Exhale (1995)
4         5 Father of the Bride Part II (1995)
5         6             Heat (1995)            Action
6         7        Sabrina (1995)
7         8      Tom and Huck (1995)        Advent...
8         9      Sudden Death (1995)
9        10       GoldenEye (1995) Action|Adve...
```

Adding indicator variables for each genre requires a little bit of wrangling. First, we extract the list of unique genres in the dataset:

```
all_genres = []
for x in movies.genres:
    all_genres.extend(x.split('|'))
genres = pd.unique(all_genres)
```

Now we have:

```
In [114]: genres
Out[114]:
array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
       'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
       'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Romantic',
       'Western'], dtype=object)
```

One way to construct the indicator DataFrame is to start with a DataFrame of all zeros:

```
In [115]: zero_matrix = np.zeros((len(movies), len(genres)))

In [116]: dummies = pd.DataFrame(zero_matrix, columns=genres)
```

Now, iterate through each movie and set entries in each row of `dummies` to 1. To do this, we use the `dummies.columns` to compute the column indices for each genre:

```
In [117]: gen = movies.genres[0]

In [118]: gen.split('|')
Out[118]: ['Animation', 'Children's', 'Comedy']

In [119]: dummies.columns.get_indexer(gen.split('|'))
Out[119]: array([0, 1, 2])
```

Then, we can use `.iloc` to set values based on these indices:

```
for i, gen in enumerate(movies.genres):
    indices = dummies.columns.get_indexer(gen.split('|'))
    dummies.iloc[i, indices] = 1
```

Then, as above, you can combine this with `movies`:

```
In [121]: movies_windic = movies.join(dummies.add_prefix('Genre_'))

In [122]: movies_windic.iloc[0]
Out[122]:
movie_id
title           Toy Story (1995)
genres          Animation|Children's|Comedy
Genre_Animation           1
Genre_Children's           1
Genre_Comedy              1
Genre_Adventure            0
Genre_Fantasy              0
Genre_Romance              0
Genre_Drama                0
...
Genre_Crime               0
Genre_Thriller             0
Genre_Horror               0
Genre_Sci-Fi               0
Genre_Documentary          0
Genre_War                  0
Genre_Musical              0
Genre_Mystery              0
Genre_Film-Noir             0
Genre_Western               0
Name: 0, Length: 21, dtype: object
```

## Note

For much larger data, this method of constructing indicator variables with multiple membership is not especially speedy. It would be better to write a lower-level function that writes directly to a NumPy array, then wrapping the result in a DataFrame.

A useful recipe for statistical applications is to combine `get_dummies` with a discretization function like `cut`:

```
In [123]: np.random.seed(12345)    # Set the random seed for determinism

In [124]: values = np.random.rand(10)

In [125]: values
Out[125]:
array([ 0.9296,  0.3164,  0.1839,  0.2046,  0.5677,  0.5955,  0.6532,
       0.7489,  0.6536])

In [126]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [127]: pd.get_dummies(pd.cut(values, bins))
Out[127]:
  (0.0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1.0]
0            0            0            0            0            1
1            0            1            0            0            0
2            1            0            0            0            0
3            0            1            0            0            0
4            0            0            1            0            0
5            0            0            1            0            0
6            0            0            0            0            1
7            0            0            0            1            0
8            0            0            0            1            0
9            0            0            0            1            0
```

# String Manipulation

Python has long been a popular raw data manipulation language in part due to its ease-of-use for string and text processing. Most text operations are made simple with the string object's built-in methods. For more complex pattern matching and text manipulations, regular expressions may be needed. pandas adds to the mix by enabling you to apply string and regular expressions concisely on whole arrays of data, additionally handling the annoyance of missing data.

# String Object Methods

In many string munging and scripting applications, built-in string methods are sufficient. As an example, a comma-separated string can be broken into pieces with `split`:

```
In [128]: val = 'a,b, guido'  
In [129]: val.split(',')  
Out[129]: ['a', 'b', ' guido']
```

`split` is often combined with `strip` to trim whitespace (including line breaks):

```
In [130]: pieces = [x.strip() for x in val.split(',')]  
In [131]: pieces  
Out[131]: ['a', 'b', 'guido']
```

These substrings could be concatenated together with a two-colon delimiter using addition:

```
In [132]: first, second, third = pieces  
In [133]: first + '::' + second + '::' + third  
Out[133]: 'a::b::guido'
```

But, this isn't a practical generic method. A faster and more Pythonic way is to pass a list or tuple to the `join` method on the string `:::`:

```
In [134]: '::'.join(pieces)  
Out[134]: 'a::b::guido'
```

Other methods are concerned with locating substrings. Using Python's `in` keyword is the best way to detect a substring, though `index` and `find` can also be used:

```
In [135]: 'guido' in val  
Out[135]: True
```

```
In [136]: val.index(',')
Out[136]: 1
```

```
In [137]: val.find(':')
Out[137]: -1
```

Note the difference between `find` and `index` is that `index` raises an exception if the string isn't found (versus returning `-1`):

```
In [138]: val.index(':')
-----
ValueError                                Traceback (most recent call last)
<ipython-input-138-280f8b2856ce> in <module>()
----> 1 val.index(':')
ValueError: substring not found
```

Relatedly, `count` returns the number of occurrences of a particular substring:

```
In [139]: val.count(',')
Out[139]: 2
```

`replace` will substitute occurrences of one pattern for another. This is commonly used to delete patterns, too, by passing an empty string:

```
In [140]: val.replace(',', ' :: ')
Out[140]: 'a::b:: guido'
```

```
In [141]: val.replace(',', '')
Out[141]: 'ab guido'
```

Regular expressions can also be used with many of these operations as you'll see below.

Table 7-3. Python built-in string methods

| Argument                | Description  |
|-------------------------|--|
| <code>count</code>      | Return the number of non-overlapping occurrences of substring in the string. |
| <code>endswith</code>   | Returns True if string ends with suffix.                                     |
| <code>startswith</code> | Returns True if string starts with prefix.                                   |
| <code>join</code>       | Use string as delimiter for concatenating a sequence of other strings.       |

|                             |  |
|-----------------------------|--|
| index                       | Return position of first character in substring if found in the string. Raises <code>ValueError</code> if not found.   |
| find                        | Return position of first character of <i>first</i> occurrence of substring in the string. Like <code>index</code> , but returns -1 if not found.             |
| rfind                       | Return position of first character of <i>last</i> occurrence of substring in the string. Returns -1 if not found.  |
| replace                     | Replace occurrences of string with another string.   |
| strip,<br>rstrip,<br>lstrip | Trim whitespace, including newlines; equivalent to <code>x.strip()</code> (and <code>rstrip</code> , <code>lstrip</code> , respectively) for each element.   |
| split                       | Break string into list of substrings using passed delimiter.   |
| lower                       | Convert alphabet characters to lowercase   |
| upper                       | Convert alphabet characters to uppercase   |
| casefold                    | Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form                                 |
| ljust,<br>rjust             | Left justify or right justify, respectively. Pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width. |

# Regular expressions

*Regular expressions* provide a flexible way to search or match (often more complex) string patterns in text. A single expression, commonly called a *regex*, is a string formed according to the regular expression language. Python's built-in `re` module is responsible for applying regular expressions to strings; I'll give a number of examples of its use here.

## Note

The art of writing regular expressions could be a chapter of its own and thus is outside the book's scope. There are many excellent tutorials and references available on the internet and in other books.

The `re` module functions fall into three categories: pattern matching, substitution, and splitting. Naturally these are all related; a regex describes a pattern to locate in the text, which can then be used for many purposes. Let's look at a simple example: suppose I wanted to split a string with a variable number of whitespace characters (tabs, spaces, and newlines). The regex describing one or more whitespace characters is `\s+`:

```
In [142]: import re  
  
In [143]: text = "foo      bar\tbaz  \tqux"  
  
In [144]: re.split('\s+', text)  
Out[144]: ['foo', 'bar', 'baz', 'qux']
```

When you call `re.split('\s+', text)`, the regular expression is first *compiled*, then its `split` method is called on the passed text. You can compile the regex yourself with `re.compile`, forming a reusable regex object:

```
In [145]: regex = re.compile('\s+')  
  
In [146]: regex.split(text)  
Out[146]: ['foo', 'bar', 'baz', 'qux']
```

If, instead, you wanted to get a list of all patterns matching the regex, you can

use the `.findall` method:

```
In [147]: regex.findall(text)
Out[147]: ['', '\t', '\t']
```

**Note**

To avoid unwanted escaping with `\` in a regular expression, use *raw* string literals like `r'C:\x'` instead of the equivalent `'C:\\\x'`.

Creating a regex object with `re.compile` is highly recommended if you intend to apply the same expression to many strings; doing so will save CPU cycles.

`match` and `search` are closely related to `.findall`. While `.findall` returns all matches in a string, `search` returns only the first match. More rigidly, `match` *only* matches at the beginning of the string. As a less trivial example, let's consider a block of text and a regular expression capable of identifying most email addresses:

```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""

pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

# re.IGNORECASE makes the regex case-insensitive
regex = re.compile(pattern, flags=re.IGNORECASE)
```

Using `.findall` on the text produces a list of the e-mail addresses:

```
In [149]: regex.findall(text)
Out[149]: ['dave@google.com', 'steve@gmail.com', 'rob@gmail.cor
```

`search` returns a special match object for the first email address in the text. For the above regex, the match object can only tell us the start and end position of the pattern in the string:

```
In [150]: m = regex.search(text)

In [151]: m
Out[151]: <sre.SRE_Match object; span=(5, 20), match='dave@goo
```

```
In [152]: text[m.start():m.end()]
Out[152]: 'dave@google.com'
```

`regex.match` returns `None`, as it only will match if the pattern occurs at the start of the string:

```
In [153]: print(regex.match(text))
None
```

Relatedly, `sub` will return a new string with occurrences of the pattern replaced by the a new string:

```
In [154]: print(regex.sub('REDACTED', text))
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

Suppose you wanted to find email addresses and simultaneously segment each address into its 3 components: username, domain name, and domain suffix. To do this, put parentheses around the parts of the pattern to segment:

```
In [155]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,3})'
In [156]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

A match object produced by this modified regex returns a tuple of the pattern components with its `groups` method:

```
In [157]: m = regex.match('wesm@bright.net')
In [158]: m.groups()
Out[158]: ('wesm', 'bright', 'net')
```

`findall` returns a list of tuples when the pattern has groups:

```
In [159]: regex.findall(text)
Out[159]:
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

`sub` also has access to groups in each match using special symbols like `\1` and `\2`. The symbol `\1` corresponds to the first matched group, `\2` the second, and so forth.

```
In [160]: print(regex.sub(r'Username: \1, Domain: \2, Suffix: `'
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

There is much more to regular expressions in Python, most of which is outside the book's scope. To give you a flavor, one variation on the above email regex gives names to the match groups:

```
regex = re.compile(r"""
    (?P<username>[A-Z0-9._%+-]+)
    @@
    (?P<domain>[A-Z0-9.-]+)
    \.
    (?P<suffix>[A-Z]{2,4})""", flags=re.IGNORECASE|re.VERBOSE)
```

The match object produced by such a regex can produce a handy dict with the specified group names:

```
In [162]: m = regex.match('wesm@bright.net')
In [163]: m.groupdict()
Out[163]: {'domain': 'bright', 'suffix': 'net', 'username': 'wesm'}
```

Table 7-4. Regular expression methods

| Argument              | Description  |
|-----------------------|--|
| <code>findall</code>  | Return all non-overlapping matching patterns in a string as a list.  |
| <code>finditer</code> | Like <code>findall</code> , but returns an iterator  |
| <code>match</code>    | Match pattern at start of string and optionally segment pattern components into groups. If the pattern matches, returns a match object, otherwise <code>None</code> .      |
| <code>search</code>   | Scan string for match to pattern; returning a match object if so. Unlike <code>match</code> , the match can be anywhere in the string as opposed to only at the beginning. |
| <code>split</code>    | Break string into pieces at each occurrence of pattern.  |

`sub,`  
`subn`

Replace all (`sub`) or first `n` occurrences (`subn`) of pattern in string with replacement expression. Use symbols `\1`, `\2`, ... to refer to match group elements in the replacement string.

# Vectorized string functions in pandas

Cleaning up a messy data set for analysis often requires a lot of string munging and regularization. To complicate matters, a column containing strings will sometimes have missing data:

```
In [164]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gr
.....:           'Rob': 'rob@gmail.com', 'Wes': np.nan}

In [165]: data = pd.Series(data)

In [166]: data
Out[166]:
Dave      dave@google.com
Rob       rob@gmail.com
Steve    steve@gmail.com
Wes          NaN
dtype: object

In [167]: data.isnull()
Out[167]:
Dave     False
Rob     False
Steve   False
Wes      True
dtype: bool
```

String and regular expression methods can be applied (passing a `lambda` or other function) to each value using `data.map`, but it will fail on the NA (null) values. To cope with this, Series has array-oriented methods for string operations that skip NA values. These are accessed through Series's `str` attribute; for example, we could check whether each email address has '`gmail`' in it with `str.contains`:

```
In [168]: data.str.contains('gmail')
Out[168]:
Dave     False
Rob      True
Steve    True
Wes      NaN
dtype: object
```

Regular expressions can be used, too, along with any `re` options like `IGNORECASE`:

```
In [169]: pattern
Out[169]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.\.([A-Z]{2,4})'

In [170]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[170]:
Dave      [(dave, google, com)]
Rob      [(rob, gmail, com)]
Steve     [(steve, gmail, com)]
Wes          NaN
dtype: object
```

There are a couple of ways to do vectorized element retrieval. Either use `str.get` or `index` into the `str` attribute:

```
In [171]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [172]: matches
Out[172]:
Dave      True
Rob      True
Steve     True
Wes          NaN
dtype: object

In [173]: matches.str.get(1)
Out[173]:
Dave      NaN
Rob      NaN
Steve     NaN
Wes          NaN
dtype: float64

In [174]: matches.str[0]
Out[174]:
Dave      NaN
Rob      NaN
Steve     NaN
Wes          NaN
dtype: float64
```

You can similarly slice strings using this syntax:

```
In [175]: data.str[:5]
```

```
Out[175] :
Dave      dave@  

Rob       rob@g  

Steve     steve  

Wes        NaN  

dtype: object
```

Table 7-5. Vectorized string methods

| Method          | Description   |
|-----------------|---|
| cat             | Concatenate strings element-wise with optional delimiter  |
| contains        | Return boolean array if each string contains pattern/regex  |
| count           | Count occurrences of pattern  |
| endswith        | Equivalent to <code>x.endswith(pattern)</code> for each element.  |
| startswith      | Equivalent to <code>x.startswith(pattern)</code> for each element.  |
| findall         | Compute list of all occurrences of pattern/regex for each string  |
| get             | Index into each element (retrieve i-th element)   |
| isalnum         | Equivalent to built-in <code>str.isalnum</code>   |
| isalpha         | Equivalent to built-in <code>str.isalpha</code>   |
| isdecimal       | Equivalent to built-in <code>str.isdecimal</code>   |
| isdigit         | Equivalent to built-in <code>str.isdigit</code>   |
| islower         | Equivalent to built-in <code>str.islower</code>   |
| isnumeric       | Equivalent to built-in <code>str.isnumeric</code>   |
| isupper         | Equivalent to built-in <code>str.isupper</code>   |
| join            | Join strings in each element of the Series with passed separator  |
| len             | Compute length of each string   |
| lower,<br>upper | Convert cases; equivalent to <code>x.lower()</code> or <code>x.upper()</code> for each element.                 |
| match           | Use <code>re.match</code> with the passed regular expression on each element, returning matched groups as list. |
| pad             | Add whitespace to left, right, or both sides of strings   |
| center          | Equivalent to <code>pad(side='both')</code>   |
| repeat          | Duplicate values; for example <code>s.str.repeat(3)</code> equivalent to <code>x * 3</code> for each string.    |
| replace         | Replace occurrences of pattern/regex with some other string   |

|                     |   |
|---------------------|---|
| <code>slice</code>  | Slice each string in the Series.                    |
| <code>split</code>  | Split strings on delimiter or regular expression    |
| <code>strip</code>  | Trim whitespace from both sides, including newlines |
| <code>rstrip</code> | Trim whitespace on right side                       |
| <code>lstrip</code> | Trim whitespace on left side                        |

# Chapter 8. Data Wrangling: Join, Combine, and Reshape

In many applications, data may be spread across many files or databases or be arranged in a form that is not easy to analyze. This chapter focuses on tools to help combine, join, and rearrange data.

First, I introduce the concept of *hierarchical indexing* in pandas, which is used extensively in some of these operations. I then dig into the particular data manipulations. You can see various applied usages of these tools in [Chapter 13](#).

# Hierarchical Indexing

*Hierarchical indexing* is an important feature of pandas enabling you to have multiple (two or more) index *levels* on an axis. Somewhat abstractly, it provides a way for you to work with higher dimensional data in a lower dimensional form. Let's start with a simple example; create a Series with a list of lists or arrays as the index:

```
In [9]: data = pd.Series(np.random.randn(9),
...:                      index=[[['a', 'a', 'a', 'b', 'b', 'c',
...:                      [1, 2, 3, 1, 3, 1, 2, 2, 3]])
```

```
In [10]: data
Out[10]:
a    1    -0.204708
      2     0.478943
      3    -0.519439
b    1    -0.555730
      3     1.965781
c    1     1.393406
      2     0.092908
d    2     0.281746
      3     0.769023
dtype: float64
```

What you're seeing is a prettified view of a Series with a `MultiIndex` as its index. The “gaps” in the index display mean “use the label directly above”:

```
In [11]: data.index
Out[11]:
MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 2,
```

With a hierarchically-indexed object, so-called *partial* indexing is possible, enabling you to concisely select subsets of the data:

```
In [12]: data['b']
Out[12]:
1    -0.555730
3     1.965781
dtype: float64
```

```
In [13]: data['b':'c']
Out[13]:
b    1    -0.555730
      3    1.965781
c    1    1.393406
      2    0.092908
dtype: float64

In [14]: data.loc[['b', 'd']]
Out[14]:
b    1    -0.555730
      3    1.965781
d    2    0.281746
      3    0.769023
dtype: float64
```

Selection is even possible in some cases from an “inner” level:

```
In [15]: data.loc[:, 2]
Out[15]:
a    0.478943
c    0.092908
d    0.281746
dtype: float64
```

Hierarchical indexing plays an important role in reshaping data and group-based operations like forming a pivot table. For example, this data could be rearranged into a DataFrame using its `unstack` method:

```
In [16]: data.unstack()
Out[16]:
          1         2         3
a -0.204708  0.478943 -0.519439
b -0.555730      NaN  1.965781
c  1.393406  0.092908      NaN
d      NaN  0.281746  0.769023
```

The inverse operation of `unstack` is `stack`:

```
In [17]: data.unstack().stack()
Out[17]:
a    1    -0.204708
      2    0.478943
      3    -0.519439
```

```
b    1    -0.555730
     3    1.965781
c    1    1.393406
     2    0.092908
d    2    0.281746
     3    0.769023
dtype: float64
```

stack and unstack will be explored in more detail in [Chapter 8](#).

With a DataFrame, either axis can have a hierarchical index:

```
In [18]: frame = pd.DataFrame(np.arange(12).reshape((4, 3)),
....:                           index=[['a', 'a', 'b', 'b'], [1,
....:                           columns=[['Ohio', 'Ohio', 'Colorado',
....:                                     ['Green', 'Red', 'Green']

In [19]: frame
Out[19]:
      Ohio      Colorado
      Green  Red      Green
a 1      0      1      2
   2      3      4      5
b 1      6      7      8
   2      9     10     11
```

The hierarchical levels can have names (as strings or any Python objects). If so, these will show up in the console output (don't confuse the index names with the axis labels!):

```
In [20]: frame.index.names = ['key1', 'key2']

In [21]: frame.columns.names = ['state', 'color']

In [22]: frame
Out[22]:
      state      Ohio      Colorado
      color      Green  Red      Green
key1 key2
a 1      0      1      2
   2      3      4      5
b 1      6      7      8
   2      9     10     11
```

With partial column indexing you can similarly select groups of columns:

```
In [23]: frame['Ohio']
Out[23]:
color      Green   Red
key1  key2
a      1       0     1
      2       3     4
b      1       6     7
      2       9    10
```

A `MultiIndex` can be created by itself and then reused; the columns in the above `DataFrame` with level names could be created like this:

```
MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green',
names=['state', 'color'])
```

# Reordering and Sorting Levels

At times you will need to rearrange the order of the levels on an axis or sort the data by the values in one specific level. The `swaplevel` takes two level numbers or names and returns a new object with the levels interchanged (but the data is otherwise unaltered):

```
In [24]: frame.swaplevel('key1', 'key2')
Out[24]:
state      Ohio      Colorado
color      Green   Red      Green
key2 key1
1      a          0      1          2
2      a          3      4          5
1      b          6      7          8
2      b          9     10         11
```

`sort_index`, on the other hand, can the data (stably) using only the values in a single level. When swapping levels, it's not uncommon to also use `sort_index` so that the result is lexicographically sorted:

```
In [25]: frame.sort_index(level=1)
Out[25]:
state      Ohio      Colorado
color      Green   Red      Green
key1 key2
a      1          0      1          2
b      1          6      7          8
a      2          3      4          5
b      2          9     10         11
```

```
In [26]: frame.swaplevel(0, 1).sort_index(level=0)
Out[26]:
state      Ohio      Colorado
color      Green   Red      Green
key2 key1
1      a          0      1          2
      b          6      7          8
2      a          3      4          5
      b          9     10         11
```

**Note**

Data selection performance is much better on hierarchically indexed objects if the index is lexicographically sorted starting with the outermost level, that is, the result of calling `sort_index(level=0)` or `sort_index()`.

# Summary Statistics by Level

Many descriptive and summary statistics on DataFrame and Series have a `level` option in which you can specify the level you want to sum by on a particular axis. Consider the above DataFrame; we can sum by level on either the rows or columns like so:

```
In [27]: frame.sum(level='key2')
Out[27]:
state    Ohio      Colorado
color   Green   Red      Green
key2
1          6      8      10
2         12     14      16
```

```
In [28]: frame.sum(level='color', axis=1)
Out[28]:
color      Green   Red
key1 key2
a       1      2      1
        2      8      4
b       1     14      7
        2     20     10
```

Under the hood, this utilizes pandas's `groupby` machinery which will be discussed in more detail later in the book.

# Indexing with a DataFrame's columns

It's not unusual to want to use one or more columns from a DataFrame as the row index; alternatively, you may wish to move the row index into the DataFrame's columns. Here's an example DataFrame:

```
In [29]: frame = pd.DataFrame({'a': range(7), 'b': range(7, 0,
.....:                               'c': ['one', 'one', 'one', 'two'
.....:                               'd': [0, 1, 2, 0, 1, 2, 3]})}

In [30]: frame
Out[30]:
   a   b     c   d
0  0   7  one   0
1  1   6  one   1
2  2   5  one   2
3  3   4  two   0
4  4   3  two   1
5  5   2  two   2
6  6   1  two   3
```

DataFrame's `set_index` function will create a new DataFrame using one or more of its columns as the index:

```
In [31]: frame2 = frame.set_index(['c', 'd'])

In [32]: frame2
Out[32]:
      a   b
c   d
one 0   7
    1   6
    2   5
two 0   4
    1   3
    2   2
    3   1
```

By default the columns are removed from the DataFrame, though you can leave them in:

```
In [33]: frame.set_index(['c', 'd'], drop=False)
```

```
Out[33]:
```

|     | a | b | c | d   |   |
|-----|---|---|---|-----|---|
| c   | d |   |   |     |   |
| one | 0 | 0 | 7 | one | 0 |
|     | 1 | 1 | 6 | one | 1 |
|     | 2 | 2 | 5 | one | 2 |
| two | 0 | 3 | 4 | two | 0 |
|     | 1 | 4 | 3 | two | 1 |
|     | 2 | 5 | 2 | two | 2 |
|     | 3 | 6 | 1 | two | 3 |

`reset_index`, on the other hand, does the opposite of `set_index`; the hierarchical index levels are moved into the columns:

```
In [34]: frame2.reset_index()
```

```
Out[34]:
```

|   | c   | d | a | b |
|---|-----|---|---|---|
| 0 | one | 0 | 0 | 7 |
| 1 | one | 1 | 1 | 6 |
| 2 | one | 2 | 2 | 5 |
| 3 | two | 0 | 3 | 4 |
| 4 | two | 1 | 4 | 3 |
| 5 | two | 2 | 5 | 2 |
| 6 | two | 3 | 6 | 1 |

# Integer Indexes

Working with pandas objects indexed by integers is something that often trips up new users due to some differences with indexing semantics on built-in Python data structures like lists and tuples. For example, you would not expect the following code to generate an error:

```
ser = pd.Series(np.arange(3.))
ser[-1]
```

In this case, pandas could “fall back” on integer indexing, but it’s difficult to do this in general without introducing subtle bugs. Here we have an index containing 0, 1, 2, but inferring what the user wants (label-based indexing or position-based) is difficult:

```
In [36]: ser
Out[36]:
0    0.0
1    1.0
2    2.0
dtype: float64
```

On the other hand, with a non-integer index, there is no potential for ambiguity:

```
In [37]: ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])

In [38]: ser2[-1]
Out[38]: 2.0
```

To keep things consistent, if you have an axis index containing integers, data selection will always be label-oriented. For more precise handling, use `loc` (for labels) or `iloc` (for integers)

```
In [39]: ser[:1]
Out[39]:
0    0.0
dtype: float64

In [40]: ser.loc[:1]
Out[40]:
```

```
0      0.0  
1      1.0  
dtype: float64
```

```
In [41]: ser.iloc[:1]  
Out[41]:  
0      0.0  
dtype: float64
```

# Combining and Merging Data Sets

Data contained in pandas objects can be combined together in a number of built-in ways:

- `pandas.merge` connects rows in DataFrames based on one or more keys. This will be familiar to users of SQL or other relational databases, as it implements database *join* operations.
- `pandas.concat` concatenates or “stacks” together objects along an axis.
- `combine_first` instance method enables splicing together overlapping data to fill in missing values in one object with values from another.

I will address each of these and give a number of examples. They'll be utilized in examples throughout the rest of the book.

# Database-style DataFrame Joins

*Merge* or *join* operations combine data sets by linking rows using one or more *keys*. These operations are central to relational databases (e.g. SQL-based). The `merge` function in pandas is the main entry point for using these algorithms on your data.

Let's start with a simple example:

```
In [42]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a',
....:                           'data1': range(7)})

In [43]: df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
....:                           'data2': range(3)})

In [44]: df1
Out[44]:
   data1  key
0      0    b
1      1    b
2      2    a
3      3    c
4      4    a
5      5    a
6      6    b

In [45]: df2
Out[45]:
   data2  key
0      0    a
1      1    b
2      2    d
```

This is an example of a *many-to-one* join; the data in `df1` has multiple rows labeled `a` and `b`, whereas `df2` has only one row for each value in the `key` column. Calling `merge` with these objects we obtain:

```
In [46]: pd.merge(df1, df2)
Out[46]:
   data1  key  data2
0      0    b      1
1      1    b      1
```

```
2      6    b      1
3      2    a      0
4      4    a      0
5      5    a      0
```

Note that I didn't specify which column to join on. If not specified, `merge` uses the overlapping column names as the keys. It's a good practice to specify explicitly, though:

```
In [47]: pd.merge(df1, df2, on='key')
Out[47]:
   data1  key  data2
0      0    b      1
1      1    b      1
2      6    b      1
3      2    a      0
4      4    a      0
5      5    a      0
```

If the column names are different in each object, you can specify them separately:

```
In [48]: df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a',
.....:                 'data1': range(5)})
In [49]: df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
.....:                 'data2': range(3)})
In [50]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[50]:
   data1  lkey  data2  rkey
0      0    b      1    b
1      1    b      1    b
2      6    b      1    b
3      2    a      0    a
4      4    a      0    a
5      5    a      0    a
```

You may notice that the '`c`' and '`d`' values and associated data are missing from the result. By default `merge` does an '`inner`' join; the keys in the result are the intersection, or the common set found in both tables. Other possible options are '`left`', '`right`', and '`outer`'. The outer join takes the union of the keys, combining the effect of applying both left and right joins:

```
In [51]: pd.merge(df1, df2, how='outer')
Out[51]:
   data1  key  data2
0      0.0    b    1.0
1      1.0    b    1.0
2      6.0    b    1.0
3      2.0    a    0.0
4      4.0    a    0.0
5      5.0    a    0.0
6      3.0    c    NaN
7      NaN    d    2.0
```

See [Table 8-1](#) for a summary of the options for `how`:

*Many-to-many* merges have well-defined though not necessarily intuitive behavior. Here's an example:

```
In [52]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a',
....:                           'data1': range(6)})

In [53]: df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
....:                           'data2': range(5)})

In [54]: df1
Out[54]:
   data1  key
0      0    b
1      1    b
2      2    a
3      3    c
4      4    a
5      5    b

In [55]: df2
Out[55]:
   data2  key
0      0    a
1      1    b
2      2    a
3      3    b
4      4    d

In [56]: pd.merge(df1, df2, on='key', how='left')
Out[56]:
   data1  key  data2
0      0    b    1.0
```

```
1      0   b    3.0
2      1   b    1.0
3      1   b    3.0
4      2   a    0.0
5      2   a    2.0
6      3   c    NaN
7      4   a    0.0
8      4   a    2.0
9      5   b    1.0
10     5   b    3.0
```

Many-to-many joins form the Cartesian product of the rows. Since there were 3 'b' rows in the left DataFrame and 2 in the right one, there are 6 'b' rows in the result. The join method only affects the distinct key values appearing in the result:

```
In [57]: pd.merge(df1, df2, how='inner')
Out[57]:
   data1  key  data2
0      0   b      1
1      0   b      3
2      1   b      1
3      1   b      3
4      5   b      1
5      5   b      3
6      2   a      0
7      2   a      2
8      4   a      0
9      4   a      2
```

To merge with multiple keys, pass a list of column names:

```
In [58]: left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],
...:                           'key2': ['one', 'two', 'one'],
...:                           'lval': [1, 2, 3]})

In [59]: right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 't'],
...:                            'key2': ['one', 'one', 'one', 'two'],
...:                            'rval': [4, 5, 6, 7]})

In [60]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
Out[60]:
  key1  key2  lval  rval
0  foo   one   1.0   4.0
1  foo   one   1.0   5.0
2  foo   two   2.0   NaN
```

```
3    bar    one    3.0    6.0
4    bar    two    NaN    7.0
```

To determine which key combinations will appear in the result depending on the choice of merge method, think of the multiple keys as forming an array of tuples to be used as a single join key (even though it's not actually implemented that way).

#### Caution

When joining columns-on-columns, the indexes on the passed DataFrame objects are discarded.

A last issue to consider in merge operations is the treatment of overlapping column names. While you can address the overlap manually (see the later section on renaming axis labels), `merge` has a `suffixes` option for specifying strings to append to overlapping names in the left and right DataFrame objects:

```
In [61]: pd.merge(left, right, on='key1')
Out[61]:
   key1  key2_x  lval  key2_y  rval
0    foo      one     1      one     4
1    foo      one     1      one     5
2    foo      two     2      one     4
3    foo      two     2      one     5
4    bar      one     3      one     6
5    bar      one     3      two     7

In [62]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
Out[62]:
   key1  key2_left  lval  key2_right  rval
0    foo      one     1      one     4
1    foo      one     1      one     5
2    foo      two     2      one     4
3    foo      two     2      one     5
4    bar      one     3      one     6
5    bar      one     3      two     7
```

See [Table 8-2](#) for an argument reference on `merge`. Joining using the DataFrame's row index is the subject of the next section.

Table 8-1. Different join types with `how` argument

| Option   | Behavior  |
|----------|---|
| 'inner'  | Use only the key combinations observed in both tables.    |
| 'left'   | Use all key combinations found in the left table.         |
| 'right'  | Use all key combinations found in the right table.        |
| 'output' | Use all key combinations observed in both tables together |

Table 8-2. merge function arguments

| Argument                 | Description   |
|--------------------------|---|
| left                     | DataFrame to be merged on the left side   |
| right                    | DataFrame to be merged on the right side  |
| how                      | One of 'inner', 'outer', 'left' or 'right'. 'inner' by default  |
| on                       | Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names in <code>left</code> and <code>right</code> as the join keys |
| <code>left_on</code>     | Columns in <code>left</code> DataFrame to use as join keys  |
| <code>right_on</code>    | Analogous to <code>left_on</code> for <code>left</code> DataFrame   |
| <code>left_index</code>  | Use row index in <code>left</code> as its join key (or keys, if a MultiIndex)   |
| <code>right_index</code> | Analogous to <code>left_index</code>  |
| sort                     | Sort merged data lexicographically by join keys; <code>True</code> by default. Disable to get better performance in some cases on large datasets  |
| suffixes                 | Tuple of string values to append to column names in case of overlap; defaults to ('_x', '_y'). For example, if 'data' in both DataFrame objects, would appear as 'data_x' and 'data_y' in result                            |
| copy                     | If <code>False</code> , avoid copying data into resulting data structure in some exceptional cases. By default always copies  |

# Merging on Index

In some cases, the merge key or keys in a DataFrame will be found in its index. In this case, you can pass `left_index=True` or `right_index=True` (or both) to indicate that the index should be used as the merge key:

```
In [63]: left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b',
.....                           'value': range(6)})  
  
In [64]: right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=  
  
In [65]: left1  
Out[65]:  
   key  value  
0    a      0  
1    b      1  
2    a      2  
3    a      3  
4    b      4  
5    c      5  
  
In [66]: right1  
Out[66]:  
   group_val  
a        3.5  
b        7.0  
  
In [67]: pd.merge(left1, right1, left_on='key', right_index=True)  
Out[67]:  
   key  value  group_val  
0    a      0      3.5  
2    a      2      3.5  
3    a      3      3.5  
1    b      1      7.0  
4    b      4      7.0
```

Since the default merge method is to intersect the join keys, you can instead form the union of them with an outer join:

```
In [68]: pd.merge(left1, right1, left_on='key', right_index=True)  
Out[68]:  
   key  value  group_val  
0    a      0      3.5
```

```

2    a      2      3.5
3    a      3      3.5
1    b      1      7.0
4    b      4      7.0
5    c      5      NaN

```

With hierarchically-indexed data, things are more complicated, as joining on index is implicitly a multiple-key merge:

```

In [69]: lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio',
.....:                               'key2': [2000, 2001, 2002, 2001,
.....:                               'data': np.arange(5.)}])

In [70]: righth = pd.DataFrame(np.arange(12).reshape((6, 2)),
.....:                           index=[['Nevada', 'Nevada', 'Oh:
.....:                           [2001, 2000, 2000, 2000,
.....:                           columns=['event1', 'event2'])

In [71]: lefth
Out[71]:
   data   key1  key2
0  0.0   Ohio  2000
1  1.0   Ohio  2001
2  2.0   Ohio  2002
3  3.0  Nevada 2001
4  4.0  Nevada 2002

In [72]: righth
Out[72]:
        event1  event2
Nevada 2001      0      1
          2000      2      3
Ohio   2000      4      5
          2000      6      7
          2001      8      9
          2002     10     11

```

In this case, you have to indicate multiple columns to merge on as a list (note the handling of duplicate index values with `how='outer'`):

```

In [73]: pd.merge(lefth, righth, left_on=['key1', 'key2'], rigl
Out[73]:
   data   key1  key2  event1  event2
0  0.0   Ohio  2000      4      5
0  0.0   Ohio  2000      6      7
1  1.0   Ohio  2001      8      9

```

```
2    2.0      Ohio  2002       10       11
3    3.0    Nevada  2001        0        1

In [74]: pd.merge(left, right, left_on=['key1', 'key2'],
....:           right_index=True, how='outer')
Out[74]:
   data    key1  key2  event1  event2
0  0.0    Ohio  2000     4.0     5.0
0  0.0    Ohio  2000     6.0     7.0
1  1.0    Ohio  2001     8.0     9.0
2  2.0    Ohio  2002    10.0    11.0
3  3.0  Nevada  2001     0.0     1.0
4  4.0  Nevada  2002     NaN     NaN
4  NaN  Nevada  2000     2.0     3.0
```

Using the indexes of both sides of the merge is also possible:

```
In [75]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]], :
....:                   columns=['Ohio', 'Nevada'])

In [76]: right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.],
....:                   index=['b', 'c', 'd', 'e'], colu

In [77]: left2
Out[77]:
   Ohio  Nevada
a    1.0     2.0
c    3.0     4.0
e    5.0     6.0

In [78]: right2
Out[78]:
   Missouri  Alabama
b        7.0     8.0
c        9.0    10.0
d       11.0    12.0
e       13.0    14.0

In [79]: pd.merge(left2, right2, how='outer', left_index=True,
Out[79]:
   Ohio  Nevada  Missouri  Alabama
a    1.0     2.0      NaN     NaN
b    NaN     NaN      7.0     8.0
c    3.0     4.0      9.0    10.0
d    NaN     NaN     11.0    12.0
e    5.0     6.0     13.0    14.0
```

DataFrame has a convenient `join` instance for merging by index. It can also be used to combine together many DataFrame objects having the same or similar indexes but non-overlapping columns. In the prior example, we could have written:

```
In [80]: left2.join(right2, how='outer')
Out[80]:
   Ohio    Nevada    Missouri    Alabama
a    1.0      2.0        NaN        NaN
b    NaN      NaN        7.0       8.0
c    3.0      4.0        9.0      10.0
d    NaN      NaN       11.0      12.0
e    5.0      6.0       13.0      14.0
```

In part for legacy reasons (much earlier versions of pandas), DataFrame's `join` method performs a left join on the join keys, exactly preserving the left frame's row index. It also supports joining the index of the passed DataFrame on one of the columns of the calling DataFrame:

```
In [81]: left1.join(right1, on='key')
Out[81]:
   key  value  group_val
0   a      0      3.5
1   b      1      7.0
2   a      2      3.5
3   a      3      3.5
4   b      4      7.0
5   c      5      NaN
```

Lastly, for simple index-on-index merges, you can pass a list of DataFrames to `join` as an alternative to using the more general `concat` function described below:

```
In [82]: another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.],
....:                               index=['a', 'c', 'e', 'f'], col
In [83]: another
Out[83]:
   New York  Oregon
a      7.0     8.0
c      9.0    10.0
e     11.0    12.0
f     16.0    17.0
```

```
In [84]: left2.join([right2, another])
Out[84]:
      Ohio    Nevada    Missouri    Alabama    New York    Oregon
a      1.0      2.0        NaN        NaN      7.0      8.0
c      3.0      4.0      9.0      10.0      9.0     10.0
e      5.0      6.0     13.0     14.0     11.0     12.0
```

```
In [85]: left2.join([right2, another], how='outer')
Out[85]:
      Ohio    Nevada    Missouri    Alabama    New York    Oregon
a      1.0      2.0        NaN        NaN      7.0      8.0
b      NaN      NaN        7.0      8.0      NaN      NaN
c      3.0      4.0      9.0      10.0      9.0     10.0
d      NaN      NaN     11.0     12.0      NaN      NaN
e      5.0      6.0     13.0     14.0     11.0     12.0
f      NaN      NaN        NaN        NaN     16.0     17.0
```

# Concatenating Along an Axis

Another kind of data combination operation is referred to interchangeably as concatenation, binding, or stacking. NumPy's `concatenate` function can do this with NumPy arrays:

```
In [86]: arr = np.arange(12).reshape((3, 4))

In [87]: arr
Out[87]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [88]: np.concatenate([arr, arr], axis=1)
Out[88]:
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

In the context of pandas objects such as Series and DataFrame, having labeled axes enable you to further generalize array concatenation. In particular, you have a number of additional things to think about:

- If the objects are indexed differently on the other axes, should we combine the distinct elements in these axes or use only the shared values (the intersection)?
- Do the concatenated chunks of data need to be identifiable in the resulting object?
- Does the “concatenation axis” contain data that needs to be preserved? In many cases, the default integer labels in a DataFrame are best discarded during concatenation.

The `concat` function in pandas provides a consistent way to address each of these concerns. I'll give a number of examples to illustrate how it works. Suppose we have three Series with no index overlap:

```
In [89]: s1 = pd.Series([0, 1], index=['a', 'b'])

In [90]: s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])

In [91]: s3 = pd.Series([5, 6], index=['f', 'g'])
```

Calling `concat` with these objects in a list glues together the values and indexes:

```
In [92]: pd.concat([s1, s2, s3])
Out[92]:
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

By default `concat` works along `axis=0`, producing another Series. If you pass `axis=1`, the result will instead be a DataFrame (`axis=1` is the columns):

```
In [93]: pd.concat([s1, s2, s3], axis=1)
Out[93]:
   0    1    2
a  0.0  NaN  NaN
b  1.0  NaN  NaN
c  NaN  2.0  NaN
d  NaN  3.0  NaN
e  NaN  4.0  NaN
f  NaN  NaN  5.0
g  NaN  NaN  6.0
```

In this case there is no overlap on the other axis, which as you can see is the sorted union (the '`outer`' join) of the indexes. You can instead intersect them by passing `join='inner'`:

```
In [94]: s4 = pd.concat([s1 * 5, s3])

In [95]: s4
Out[95]:
a    0
b    5
f    5
```

```
g      6
dtype: int64

In [96]: pd.concat([s1, s4], axis=1)
Out[96]:
   0   1
a  0.0  0
b  1.0  5
f  NaN  5
g  NaN  6

In [97]: pd.concat([s1, s4], axis=1, join='inner')
Out[97]:
   0   1
a  0   0
b  1   5
```

You can even specify the axes to be used on the other axes with `join_axes`:

```
In [98]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b'],
Out[98]:
   0   1
a  0.0  0.0
c  NaN  NaN
b  1.0  5.0
e  NaN  NaN
```

A potential issue is that the concatenated pieces are not identifiable in the result. Suppose instead you wanted to create a hierarchical index on the concatenation axis. To do this, use the `keys` argument:

```
In [99]: result = pd.concat([s1, s1, s3], keys=['one', 'two',
In [100]: result
Out[100]:
one    a    0
       b    1
two    a    0
       b    1
three   f    5
       g    6
dtype: int64

# Much more on the unstack function later
In [101]: result.unstack()
Out[101]:
```

```
      a      b      f      g
one   0.0   1.0    NaN    NaN
two   0.0   1.0    NaN    NaN
three  NaN   NaN    5.0    6.0
```

In the case of combining Series along `axis=1`, the keys become the DataFrame column headers:

```
In [102]: pd.concat([s1, s2, s3], axis=1, keys=['one', 'two',
Out[102]:
      one    two    three
a  0.0    NaN    NaN
b  1.0    NaN    NaN
c  NaN    2.0    NaN
d  NaN    3.0    NaN
e  NaN    4.0    NaN
f  NaN    NaN    5.0
g  NaN    NaN    6.0
```

The same logic extends to DataFrame objects:

```
In [103]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=
.....:                               columns=['one', 'two'])
In [104]: df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=
.....:                               columns=['three', 'four'])
In [105]: df1
Out[105]:
      one    two
a      0      1
b      2      3
c      4      5
In [106]: df2
Out[106]:
      three    four
a        5      6
c        7      8
In [107]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
Out[107]:
      level1      level2
      one    two    three    four
a        0      1      5.0    6.0
b        2      3      NaN    NaN
```

```
c      4      5      7.0    8.0
```

If you pass a dict of objects instead of a list, the dict's keys will be used for the `keys` option:

```
In [108]: pd.concat({'level1': df1, 'level2': df2}, axis=1)
Out[108]:
   level1      level2
   one  two  three  four
a      0     1     5.0   6.0
b      2     3     NaN   NaN
c      4     5     7.0   8.0
```

There are additional arguments governing how the hierarchical index is created (see [Table 8-3](#)). For example, we can name the created axis levels with the `names` argument:

```
In [109]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
.....:                  names=['upper', 'lower'])
Out[109]:
   upper  level1      level2
   lower      one  two  three  four
a          0     1     5.0   6.0
b          2     3     NaN   NaN
c          4     5     7.0   8.0
```

A last consideration concerns DataFrames in which the row index does not contain any relevant data:

```
In [110]: df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])
In [111]: df2 = pd.DataFrame(np.random.randn(2, 3), columns=['k', 'l', 'm'])

In [112]: df1
Out[112]:
   a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741

In [113]: df2
Out[113]:
   b         d         a
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
```

In this case, you can pass `ignore_index=True`:

```
In [114]: pd.concat([df1, df2], ignore_index=True)
Out[114]:
      a          b          c          d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741
3 -1.021228  0.476985        NaN  3.248944
4  0.302614 -0.577087        NaN  0.124121
```

Table 8-3. concat function arguments

| Argument                      | Description  |
|-------------------------------|--|
| <code>objs</code>             | List or dict of pandas objects to be concatenated. The only required argument  |
| <code>axis</code>             | Axis to concatenate along; defaults to 0   |
| <code>join</code>             | One of 'inner', 'outer', defaulting to 'outer'; whether to intersection (inner) or union (outer) together indexes along the other axes   |
| <code>join_axes</code>        | Specific indexes to use for the other n-1 axes instead of performing union/intersection logic  |
| <code>keys</code>             | Values to associate with objects being concatenated, forming a hierarchical index along the concatenation axis. Can either be a list or array of arbitrary values, an array of tuples, or a list of arrays (if multiple level arrays passed in <code>levels</code> ) |
| <code>levels</code>           | Specific indexes to use as hierarchical index level or levels if <code>keys</code> passed  |
| <code>names</code>            | Names for created hierarchical levels if <code>keys</code> and / or <code>levels</code> passed   |
| <code>verify_integrity</code> | Check new axis in concatenated object for duplicates and raise exception if so. By default ( <code>False</code> ) allows duplicates  |
| <code>ignore_index</code>     | Do not preserve indexes along concatenation <code>axis</code> , instead producing a new <code>range(total_length)</code> index   |

# Combining Data with Overlap

Another data combination situation can't be expressed as either a merge or concatenation operation. You may have two datasets whose indexes overlap in full or part. As a motivating example, consider NumPy's `where` function, which performs the array-oriented equivalent of an if-else expression:

```
In [115]: a = pd.Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan]
.....:                 index=['f', 'e', 'd', 'c', 'b', 'a'])

In [116]: b = pd.Series(np.arange(len(a), dtype=np.float64),
.....:                 index=['f', 'e', 'd', 'c', 'b', 'a'])

In [117]: b[-1] = np.nan

In [118]: a
Out[118]:
f      NaN
e      2.5
d      NaN
c      3.5
b      4.5
a      NaN
dtype: float64

In [119]: b
Out[119]:
f      0.0
e      1.0
d      2.0
c      3.0
b      4.0
a      NaN
dtype: float64

In [120]: np.where(pd.isnull(a), b, a)
Out[120]: array([ 0.,  2.5,  2.,  3.5,  4.5,  nan])
```

Series has a `combine_first` method, which performs the equivalent of this operation along with pandas's usual data alignment logic:

```
In [121]: b[:-2].combine_first(a[2:])
```

```
Out[121]:  
a      NaN  
b      4.5  
c      3.0  
d      2.0  
e      1.0  
f      0.0  
dtype: float64
```

With `DataFrames`, `combine_first` does the same thing column by column, so you can think of it as “patching” missing data in the calling object with data from the object you pass:

```
In [122]: df1 = pd.DataFrame({'a': [1., np.nan, 5., np.nan],  
.....:                      'b': [np.nan, 2., np.nan, 6.],  
.....:                      'c': range(2, 18, 4)})  
  
In [123]: df2 = pd.DataFrame({'a': [5., 4., np.nan, 3., 7.],  
.....:                      'b': [np.nan, 3., 4., 6., 8.]})  
  
In [124]: df1  
Out[124]:  
       a    b    c  
0  1.0  NaN   2  
1  NaN  2.0   6  
2  5.0  NaN  10  
3  NaN  6.0  14  
  
In [125]: df2  
Out[125]:  
       a    b  
0  5.0  NaN  
1  4.0  3.0  
2  NaN  4.0  
3  3.0  6.0  
4  7.0  8.0  
  
In [126]: df1.combine_first(df2)  
Out[126]:  
       a    b    c  
0  1.0  NaN  2.0  
1  4.0  2.0  6.0  
2  5.0  4.0 10.0  
3  3.0  6.0 14.0  
4  7.0  8.0  NaN
```

# Reshaping and Pivoting

There are a number of basic operations for rearranging tabular data. These are alternately referred to as *reshape* or *pivot* operations.

# Reshaping with Hierarchical Indexing

Hierarchical indexing provides a consistent way to rearrange data in a DataFrame. There are two primary actions:

- `stack`: this “rotates” or pivots from the columns in the data to the rows
- `unstack`: this pivots from the rows into the columns

I'll illustrate these operations through a series of examples. Consider a small DataFrame with string arrays as row and column indexes:

```
In [127]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),  
.....: index=pd.Index(['Ohio', 'Colorado', 'Colorado', 'Ohio'],  
.....: columns=pd.Index(['one', 'two', 'three'],  
  
In [128]: data  
Out[128]:  
number      one    two    three  
state  
Ohio         0      1      2  
Colorado     3      4      5
```

Using the `stack` method on this data pivots the columns into the rows, producing a Series:

```
In [129]: result = data.stack()  
  
In [130]: result  
Out[130]:  
state      number  
Ohio      one      0  
          two      1  
          three     2  
Colorado   one      3  
          two      4  
          three     5  
dtype: int64
```

From a hierarchically-indexed Series, you can rearrange the data back into a DataFrame with `unstack`:

```
In [131]: result.unstack()  
Out[131]:  
number      one    two   three  
state  
Ohio        0      1      2  
Colorado    3      4      5
```

By default the innermost level is unstacked (same with `stack`). You can unstack a different level by passing a level number or name:

```
In [132]: result.unstack(0)  
Out[132]:  
state    Ohio  Colorado  
number  
one        0      3  
two        1      4  
three     2      5
```

```
In [133]: result.unstack('state')  
Out[133]:  
state    Ohio  Colorado  
number  
one        0      3  
two        1      4  
three     2      5
```

Unstacking might introduce missing data if all of the values in the level aren't found in each of the subgroups:

```
In [134]: s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])  
In [135]: s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])  
In [136]: data2 = pd.concat([s1, s2], keys=['one', 'two'])  
  
In [137]: data2  
Out[137]:  
one    a    0  
      b    1  
      c    2  
      d    3  
two    c    4  
      d    5  
      e    6  
dtype: int64
```

```
In [138]: data2.unstack()
Out[138]:
      a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

Stacking filters out missing data by default, so the operation is more easily invertible:

```
In [139]: data2.unstack().stack()
Out[139]:
one   a    0.0
      b    1.0
      c    2.0
      d    3.0
two   c    4.0
      d    5.0
      e    6.0
dtype: float64

In [140]: data2.unstack().stack(dropna=False)
Out[140]:
one   a    0.0
      b    1.0
      c    2.0
      d    3.0
      e    NaN
two   a    NaN
      b    NaN
      c    4.0
      d    5.0
      e    6.0
dtype: float64
```

When unstacking in a DataFrame, the level unstacked becomes the lowest level in the result:

```
In [141]: df = pd.DataFrame({'left': result, 'right': result +
.....:                                         columns=pd.Index(['left', 'right'],
In [142]: df
Out[142]:
      side      left  right
state  number
Ohio    one        0      5
       two        1      6
```

```
      three      2      7
Colorado one      3      8
          two      4      9
          three     5     10
```

```
In [143]: df.unstack('state')
Out[143]:
side   left           right
state  Ohio Colorado  Ohio Colorado
number
one      0        3        5        8
two      1        4        6        9
three     2        5        7       10
```

```
In [144]: df.unstack('state').stack('side')
Out[144]:
state           Colorado  Ohio
number side
one    left        3        0
          right       8        5
two    left        4        1
          right       9        6
three  left        5        2
          right      10       7
```

# Pivoting “long” to “wide” Format

A common way to store multiple time series in databases and CSV is in so-called *long* or *stacked* format. First, let’s load some example data and do a small amount of time series wrangling and other data cleaning:

```
data = pd.read_csv('examples/macrodata.csv')
periods = pd.PeriodIndex(year=data.year, quarter=data.quarter,
data = pd.DataFrame(data.to_records(),
                     columns=pd.Index(['realgdp', 'infl', 'unemp'],
                     index=periods.to_timestamp('D', 'end')))

ldata = data.stack().reset_index().rename(columns={0: 'value'})
```

Now, `ldata` looks like:

```
In [146]: ldata[:10]
Out[146]:
      date      item    value
0 1959-03-31  realgdp  2710.349
1 1959-03-31      infl   0.000
2 1959-03-31     unemp  5.800
3 1959-06-30  realgdp  2778.801
4 1959-06-30      infl   2.340
5 1959-06-30     unemp  5.100
6 1959-09-30  realgdp  2775.488
7 1959-09-30      infl   2.740
8 1959-09-30     unemp  5.300
9 1959-12-31  realgdp  2785.204
```

This is the so-called *long* format for multiple time series, or other observational data with two or more keys (here, our keys are `date` and `item`). Each row in the table represents a single observation.

Data is frequently stored this way in relational databases like MySQL as a fixed schema (column names and data types) allows the number of distinct values in the `item` column to change as data is added to the table. In the above example `date` and `item` would usually be the primary keys (in relational database parlance), offering both relational integrity and easier joins. In some cases, the data may be more difficult to work with in this format; you might

prefer to have a DataFrame containing one column per distinct `item` value indexed by timestamps in the `date` column. DataFrame's `pivot` method performs exactly this transformation:

```
In [147]: pivoted = ldata.pivot('date', 'item', 'value')

In [148]: pivoted
Out[148]:
item      infl    realgdp   unemp
date
1959-03-31  0.00  2710.349   5.8
1959-06-30  2.34  2778.801   5.1
1959-09-30  2.74  2775.488   5.3
1959-12-31  0.27  2785.204   5.6
1960-03-31  2.31  2847.699   5.2
1960-06-30  0.14  2834.390   5.2
1960-09-30  2.70  2839.022   5.6
1960-12-31  1.21  2802.616   6.3
1961-03-31 -0.40  2819.264   6.8
1961-06-30  1.47  2872.005   7.0
...
...
2007-06-30  2.75  13203.977   4.5
2007-09-30  3.45  13321.109   4.7
2007-12-31  6.38  13391.249   4.8
2008-03-31  2.82  13366.865   4.9
2008-06-30  8.53  13415.266   5.4
2008-09-30 -3.16  13324.600   6.0
2008-12-31 -8.79  13141.920   6.9
2009-03-31  0.94  12925.410   8.1
2009-06-30  3.37  12901.504   9.2
2009-09-30  3.56  12990.341   9.6
[203 rows x 3 columns]
```

The first two values passed are the columns to be used respectively as the row and column index, then finally an optional value column to fill the DataFrame. Suppose you had two value columns that you wanted to reshape simultaneously:

```
In [149]: ldata['value2'] = np.random.randn(len(ldata))

In [150]: ldata[:10]
Out[150]:
       date     item    value  value2
0 1959-03-31  realgdp  2710.349  0.523772
1 1959-03-31      infl    0.000  0.000940
```

```

2 1959-03-31      unemp      5.800  1.343810
3 1959-06-30    realgdp  2778.801 -0.713544
4 1959-06-30      infl      2.340 -0.831154
5 1959-06-30      unemp      5.100 -2.370232
6 1959-09-30    realgdp  2775.488 -1.860761
7 1959-09-30      infl      2.740 -0.860757
8 1959-09-30      unemp      5.300  0.560145
9 1959-12-31    realgdp  2785.204 -1.265934

```

By omitting the last argument, you obtain a DataFrame with hierarchical columns:

```
In [151]: pivoted = ldata.pivot('date', 'item')
```

```
In [152]: pivoted[:5]
```

```
Out[152]:
```

|            | value |          |       | value2    |           |           |
|------------|-------|----------|-------|-----------|-----------|-----------|
| item       | infl  | realgdp  | unemp | infl      | realgdp   | unemp     |
| date       |       |          |       |           |           |           |
| 1959-03-31 | 0.00  | 2710.349 | 5.8   | 0.000940  | 0.523772  | 1.343810  |
| 1959-06-30 | 2.34  | 2778.801 | 5.1   | -0.831154 | -0.713544 | -2.370232 |
| 1959-09-30 | 2.74  | 2775.488 | 5.3   | -0.860757 | -1.860761 | 0.560145  |
| 1959-12-31 | 0.27  | 2785.204 | 5.6   | 0.119827  | -1.265934 | -1.063512 |
| 1960-03-31 | 2.31  | 2847.699 | 5.2   | -2.359419 | 0.332883  | -0.199543 |

```
In [153]: pivoted['value'][:5]
```

```
Out[153]:
```

| item       | infl | realgdp  | unemp |
|------------|------|----------|-------|
| date       |      |          |       |
| 1959-03-31 | 0.00 | 2710.349 | 5.8   |
| 1959-06-30 | 2.34 | 2778.801 | 5.1   |
| 1959-09-30 | 2.74 | 2775.488 | 5.3   |
| 1959-12-31 | 0.27 | 2785.204 | 5.6   |
| 1960-03-31 | 2.31 | 2847.699 | 5.2   |

Note that `pivot` is equivalent to creating a hierarchical index using `set_index` followed by a call to `unstack`:

```
In [154]: unstacked = ldata.set_index(['date', 'item']).unstack()
```

```
In [155]: unstacked[:7]
```

```
Out[155]:
```

|            | value |          |       | value2   |          |          |
|------------|-------|----------|-------|----------|----------|----------|
| item       | infl  | realgdp  | unemp | infl     | realgdp  | unemp    |
| date       |       |          |       |          |          |          |
| 1959-03-31 | 0.00  | 2710.349 | 5.8   | 0.000940 | 0.523772 | 1.343810 |

|            |      |          |     |           |           |           |
|------------|------|----------|-----|-----------|-----------|-----------|
| 1959-06-30 | 2.34 | 2778.801 | 5.1 | -0.831154 | -0.713544 | -2.370232 |
| 1959-09-30 | 2.74 | 2775.488 | 5.3 | -0.860757 | -1.860761 | 0.560145  |
| 1959-12-31 | 0.27 | 2785.204 | 5.6 | 0.119827  | -1.265934 | -1.063512 |
| 1960-03-31 | 2.31 | 2847.699 | 5.2 | -2.359419 | 0.332883  | -0.199543 |
| 1960-06-30 | 0.14 | 2834.390 | 5.2 | -0.970736 | -1.541996 | -1.307030 |
| 1960-09-30 | 2.70 | 2839.022 | 5.6 | 0.377984  | 0.286350  | -0.753887 |

# Chapter 9. Plotting and Visualization

Making informative visualizations (sometimes called *plots*) is one of the most important tasks in data analysis. It may be a part of the exploratory process; for example, helping identify outliers, needed data transformations, or coming up with ideas for models. For others, building an interactive visualization for the web may be the end goal. Python has many add-on libraries for making static or dynamic visualizations, but I'll be mainly focused on matplotlib (<http://matplotlib.sourceforge.net>) and libraries that build on top of matplotlib.

matplotlib is a desktop plotting package designed for creating (mostly two-dimensional) publication-quality plots. The project was started by John Hunter in 2002 to enable a MATLAB-like plotting interface in Python. The matplotlib and IPython communities have collaborated to simplify interactive plotting from the IPython shell (and now, Jupyter notebook). matplotlib supports various GUI backends on all operating systems and additionally can export graphics to all of the common vector and raster graphics formats: PDF, SVG, JPG, PNG, BMP, GIF, etc. I have used it to produce almost all of the graphics outside of diagrams in this book.

matplotlib has a number of add-on toolkits, such as `mplot3d` for 3D plots and `basemap` for mapping and projections.

The simplest way to follow the code examples in the chapter is to use inline plotting in the Jupyter notebook (Execute `%matplotlib inline`, as described in [Chapter 2](#).

# A Brief matplotlib API Primer

With matplotlib, we use the following import convention:

```
In [10]: import matplotlib.pyplot as plt
```

After running `%matplotlib inline` in Jupyter (or simply `%matplotlib` in IPython), try creating a simple plot:

```
import numpy as np  
plt.plot(np.arange(10))
```

If everything is set up right, a plot should appear with a line plot.

While libraries like seaborn and pandas's built-in plotting functions will deal with many of the mundane details of making plots, should you wish to customize them beyond the function options provided you will need to learn a bit about the matplotlib API.

## Note

There is not enough room in the book to give a comprehensive treatment to the breadth and depth of functionality in matplotlib. It should be enough to teach you the ropes to get up and running. The matplotlib gallery and documentation are the best resource for learning advanced features.

# Figures and Subplots

Plots in matplotlib reside within a `Figure` object. You can create a new figure with `plt.figure`:

```
In [11]: fig = plt.figure()
```

In IPython, an empty plot window will appear, but in Jupyter nothing will be shown until we use a few more commands. `plt.figure` has a number of options, notably `figsize` will guarantee the figure has a certain size and aspect ratio if saved to disk.

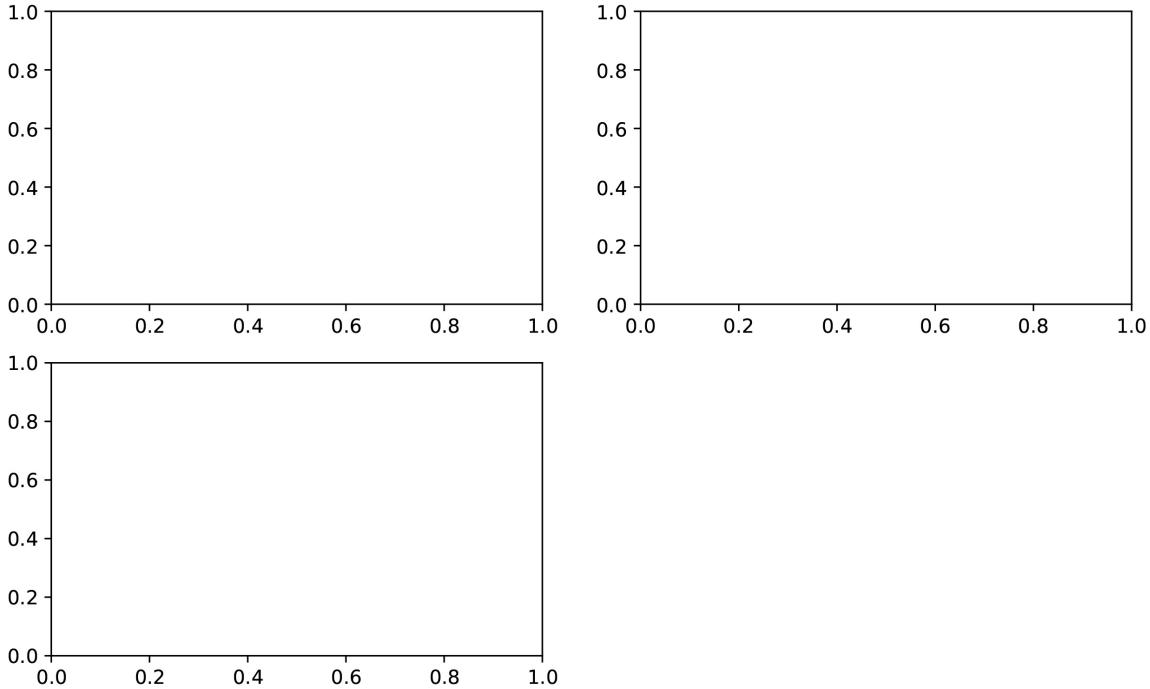
You can't make a plot with a blank figure. You have to create one or more subplots using `add_subplot`:

```
In [12]: ax1 = fig.add_subplot(2, 2, 1)
```

This means that the figure should be  $2 \times 2$ , and we're selecting the first of 4 subplots (numbered from 1). If you create the next two subplots, you'll end up with a figure that looks like [Figure 9-1](#).

```
In [13]: ax2 = fig.add_subplot(2, 2, 2)
```

```
In [14]: ax3 = fig.add_subplot(2, 2, 3)
```



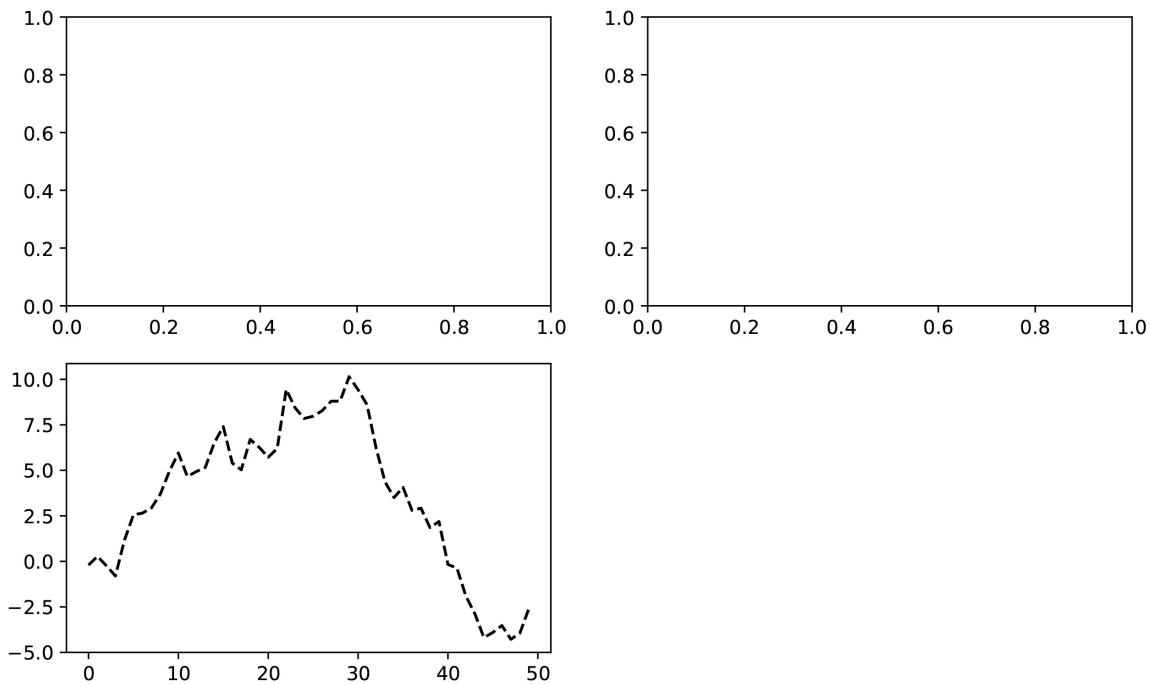
**Figure 9-1.** An empty matplotlib Figure with 3 subplots

One nuance of using Jupyter notebooks is that plots are reset after each cell is evaluated. So here we must run all of these commands in the same cell:

```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
```

When you issue a plotting command like `plt.plot([1.5, 3.5, -2, 1.6])`, matplotlib draws on the last figure and subplot used (creating one if necessary), thus hiding the figure and subplot creation. Thus, if we add the following command, you'll get something like [Figure 9-2](#):

```
In [15]: plt.plot(np.random.randn(50).cumsum(), 'k--')
```

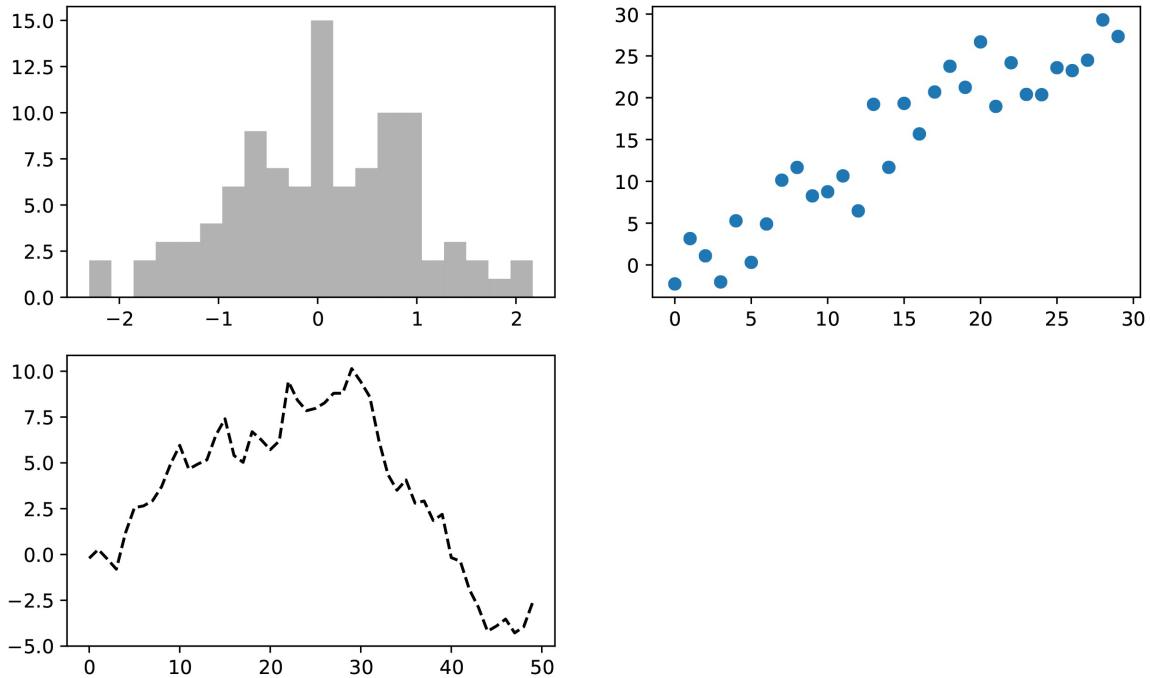


**Figure 9-2. Figure after single plot**

The '`k--`' is a *style* option instructing matplotlib to plot a black dashed line. The objects returned by `fig.add_subplot` above are `AxesSubplot` objects, on which you can directly plot on the other empty subplots by calling each one's instance methods, see [Figure 9-3](#):

```
In [16]: _ = ax1.hist(np.random.randn(100), bins=20, color='k',
```

```
In [17]: ax2.scatter(np.arange(30), np.arange(30) + 3 * np.rand
```



**Figure 9-3. Figure after additional plots**

You can find a comprehensive catalogue of plot types in the matplotlib documentation.

Since creating a figure with a grid of subplots is such a common task, there is a convenience method, `plt.subplots`, that creates a new figure and returns a NumPy array containing the created subplot objects:

```
In [19]: fig, axes = plt.subplots(2, 3)

In [20]: axes
Out[20]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f64c00000>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f64c00000>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f64c00000>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7f64c00000>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f64c00000>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x7f64c00000>]])
```

This is very useful as the `axes` array can be easily indexed like a two-dimensional array; for example, `axes[0, 1]`. You can also indicate that subplots should have the same X or Y axis using `sharex` and `sharey`,

respectively. This is especially useful when comparing data on the same scale; otherwise, matplotlib auto-scales plot limits independently. See [Table 9-1](#) for more on this method.

Table 9-1. pyplot.subplots options

| Argument                | Description   |
|-------------------------|---|
| nrows                   | Number of rows of subplots  |
| ncols                   | Number of columns of subplots   |
| sharex                  | All subplots should use the same X-axis ticks (adjusting the <code>xlim</code> will affect all subplots)                                |
| sharey                  | All subplots should use the same Y-axis ticks (adjusting the <code>ylim</code> will affect all subplots)                                |
| <code>subplot_kw</code> | Dict of keywords passed to <code>add_subplot</code> call used to create each subplot.   |
| <code>**fig_kw</code>   | Additional keywords to <code>subplots</code> are used when creating the figure, such as <code>plt.subplots(2, 2, figsize=(8, 6))</code> |

## Adjusting the spacing around subplots

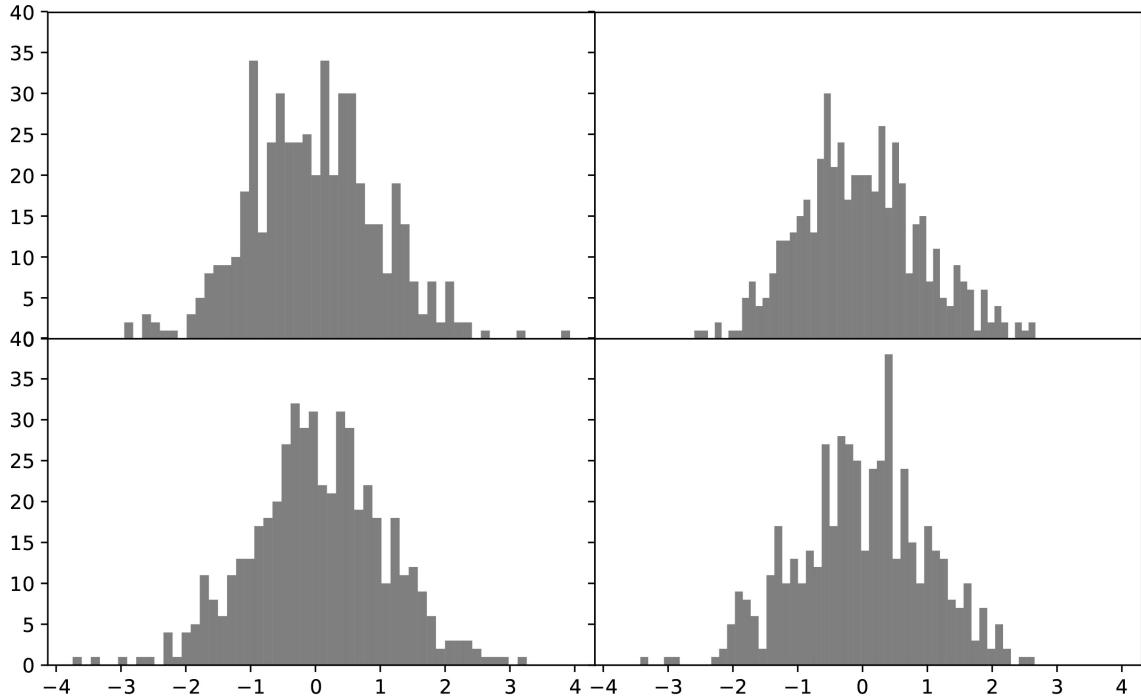
By default matplotlib leaves a certain amount of padding around the outside of the subplots and spacing between subplots. This spacing is all specified relative to the height and width of the plot, so that if you resize the plot either programmatically or manually using the GUI window, the plot will dynamically adjust itself. The spacing can be most easily changed using the `subplots_adjust` Figure method, also available as a top-level function:

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)
```

`wspace` and `hspace` controls the percent of the figure width and figure height, respectively, to use as spacing between subplots. Here is a small example where I shrink the spacing all the way to zero (see [Figure 9-4](#)):

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
    for j in range(2):
        axes[i, j].hist(np.random.randn(500), bins=50, color='l
```

```
plt.subplots_adjust(wspace=0, hspace=0)
```



**Figure 9-4. Figure with no inter-subplot spacing**

You may notice that the axis labels overlap. matplotlib doesn't check whether the labels overlap, so in a case like this you would need to fix the labels yourself by specifying explicit tick locations and tick labels. More on this in the coming sections.

# Colors, Markers, and Line Styles

Matplotlib's main `plot` function accepts arrays of X and Y coordinates and optionally a string abbreviation indicating color and line style. For example, to plot `x` versus `y` with green dashes, you would execute:

```
ax.plot(x, y, 'g--')
```

This way of specifying both color and linestyle in a string is provided as a convenience; in practice if you were creating plots programmatically you might prefer not to have to munge strings together to create plots with the desired style. The same plot could also have been expressed more explicitly as:

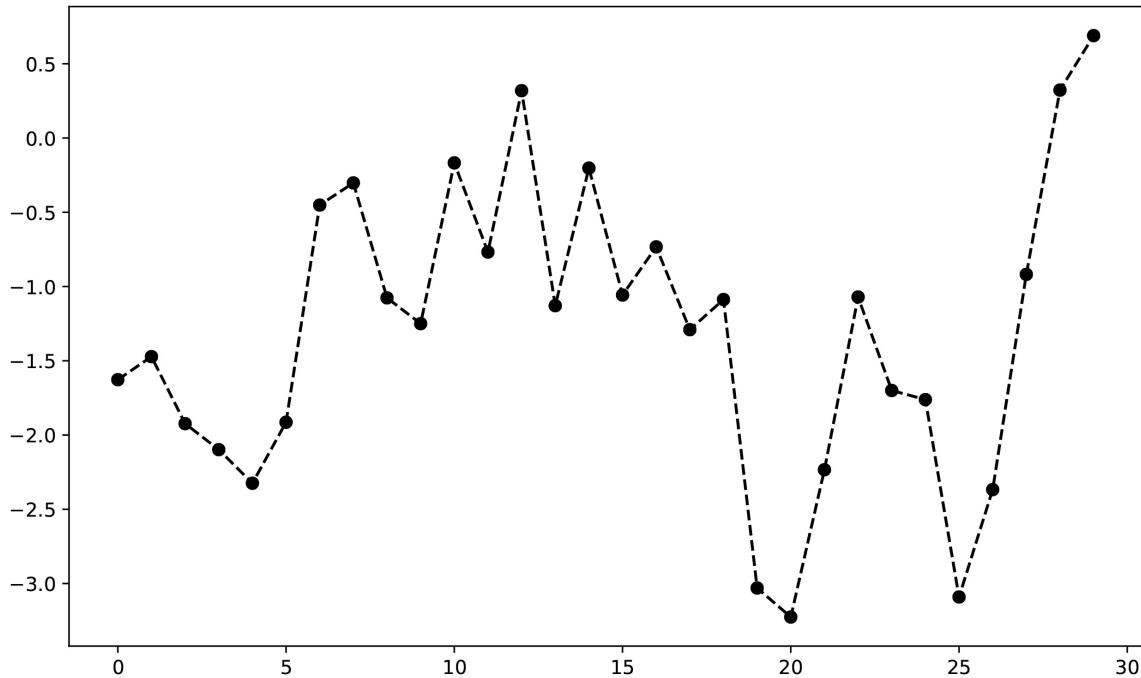
```
ax.plot(x, y, linestyle='--', color='g')
```

There are a number of color abbreviations provided for commonly-used colors, but any color on the spectrum can be used by specifying its RGB value (for example, '`#CECECE`'). You can see the full set of linestyles by looking at the docstring for `plot`.

Line plots can additionally have *markers* to highlight the actual data points. Since matplotlib creates a continuous line plot, interpolating between points, it can occasionally be unclear where the points lie. The marker can be part of the style string, which must have color followed by marker type and line style (see [Figure 9-5](#)):

```
In [25]: from numpy.random import randn
```

```
In [26]: plt.plot(randn(30).cumsum(), 'ko--')
```



**Figure 9-5. Line plot with markers example**

This could also have been written more explicitly as:

```
plot(randn(30).cumsum(), color='k', linestyle='dashed', marker=
```

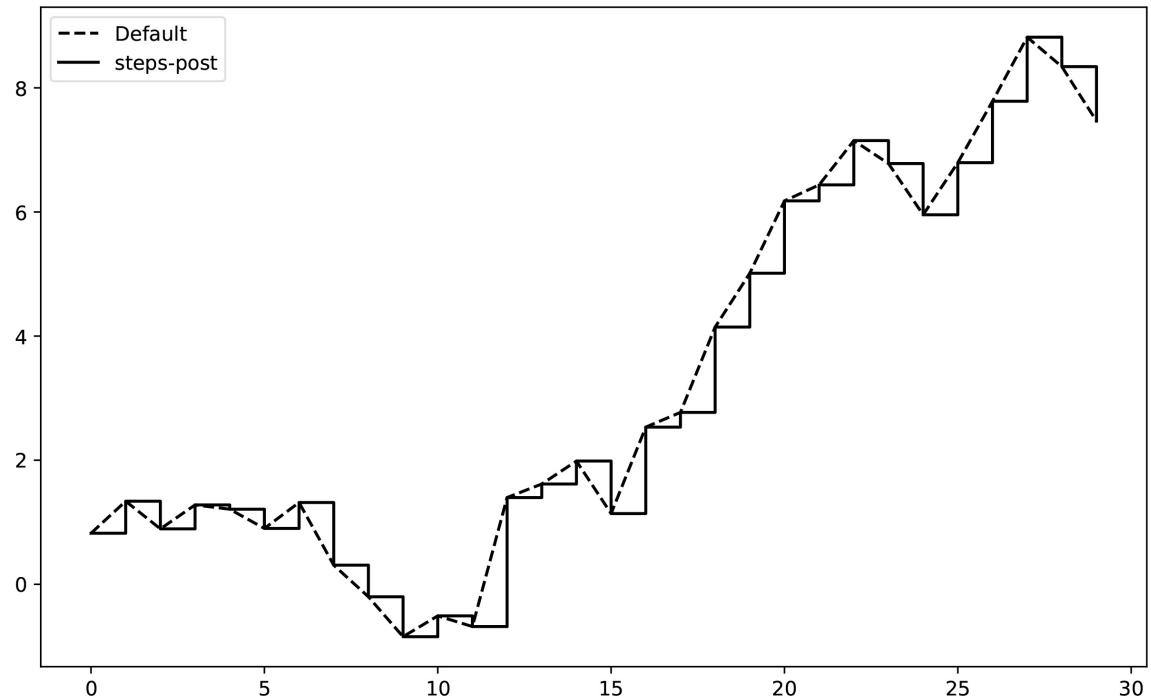
For line plots, you will notice that subsequent points are linearly interpolated by default. This can be altered with the `drawstyle` option:

```
In [28]: data = np.random.randn(30).cumsum()
```

```
In [29]: plt.plot(data, 'k--', label='Default')
Out[29]: [<matplotlib.lines.Line2D at 0x7f64b7859400>]
```

```
In [30]: plt.plot(data, 'k-', drawstyle='steps-post', label='st
Out[30]: [<matplotlib.lines.Line2D at 0x7f64b7859da0>]
```

```
In [31]: plt.legend(loc='best')
```



**Figure 9-6.** Line plot with different drawstyle options

# Ticks, Labels, and Legends

For most kinds of plot decorations, there are two main ways to do things: using the procedural `pyplot` interface (i.e. `matplotlib.pyplot`) and the more object-oriented native `matplotlib` API.

The `pyplot` interface, designed for interactive use, consists of methods like `xlim`, `xticks`, and `xticklabels`. These control the plot range, tick locations, and tick labels, respectively. They can be used in two ways:

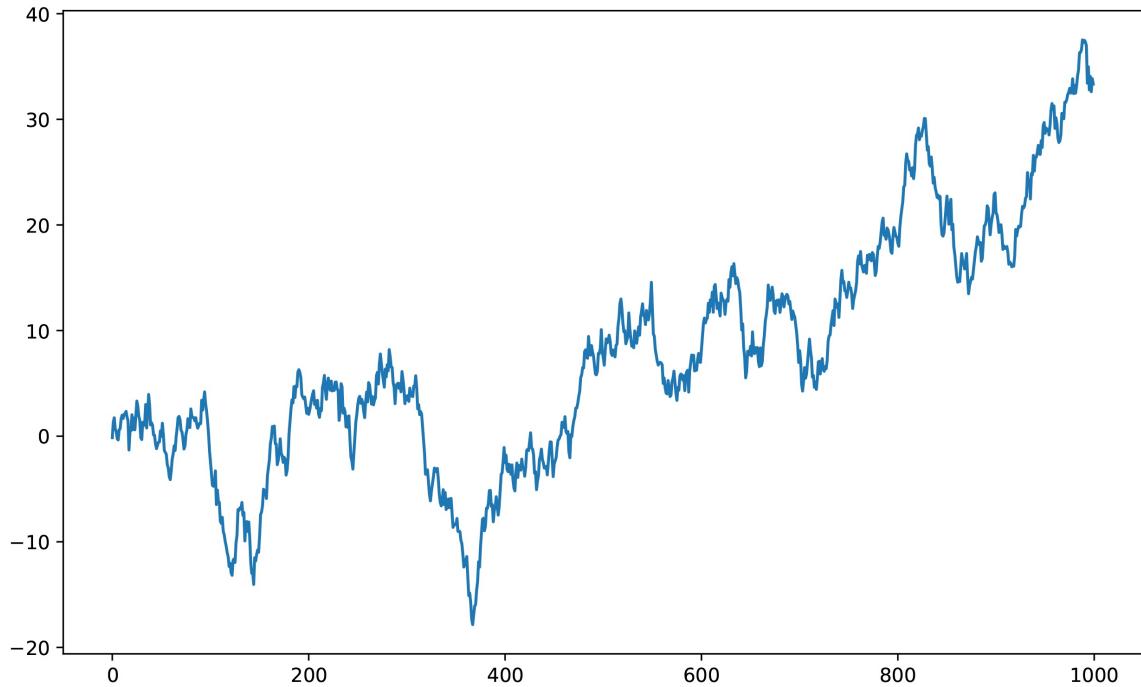
- Called with no arguments returns the current parameter value. For example `plt.xlim()` returns the current X axis plotting range
- Called with parameters sets the parameter value. So `plt.xlim([0, 10])`, sets the X axis range to 0 to 10

All such methods act on the active or most recently-created `AxesSubplot`. Each of them corresponds to two methods on the subplot object itself; in the case of `xlim` these are `ax.get_xlim` and `ax.set_xlim`. I prefer to use the subplot instance methods myself in the interest of being explicit (and especially when working with multiple subplots), but you can certainly use whichever you find more convenient.

## Setting the title, axis labels, ticks, and ticklabels

To illustrate customizing the axes, I'll create a simple figure and plot of a random walk (see [Figure 9-7](#)):

```
In [32]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)  
In [33]: ax.plot(np.random.randn(1000).cumsum())
```



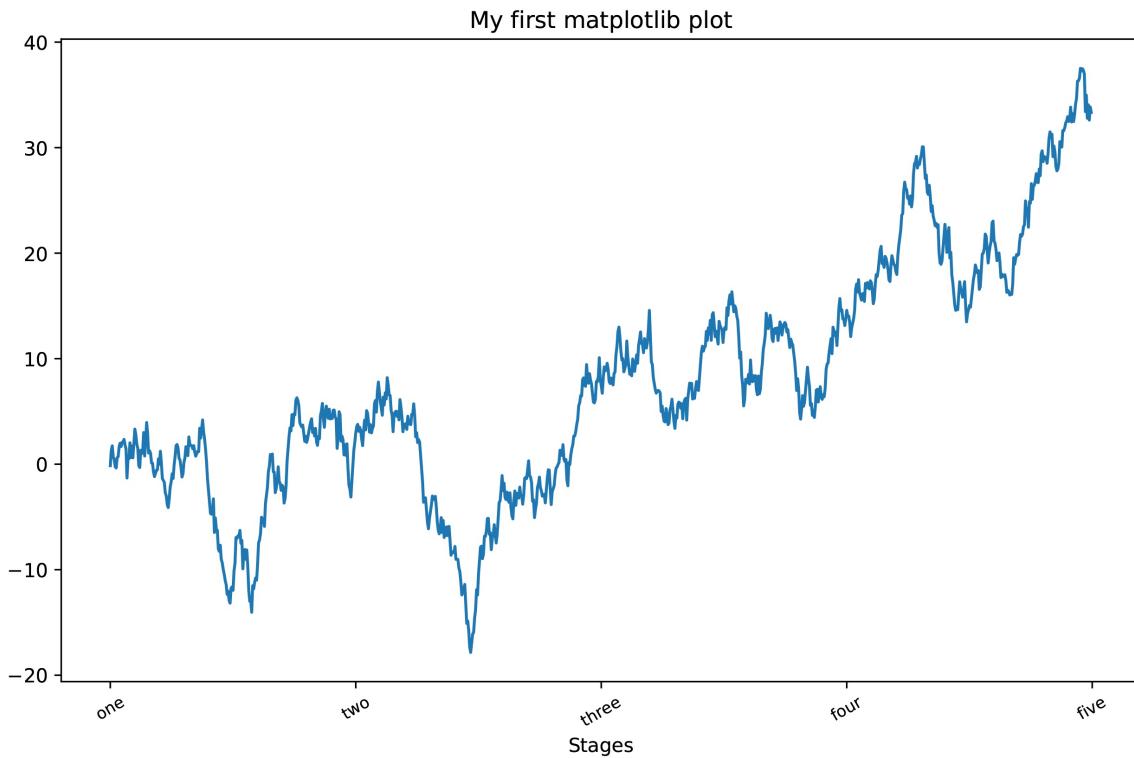
**Figure 9-7. Simple plot for illustrating xticks**

To change the X axis ticks, it's easiest to use `set_xticks` and `set_xticklabels`. The former instructs matplotlib where to place the ticks along the data range; by default these locations will also be the labels. But we can set any other values as the labels using `set_xticklabels`:

```
In [34]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])  
In [35]: labels = ax.set_xticklabels(['one', 'two', 'three', 'four',  
.....:                                     rotation=30, fontsize='smaller')
```

Lastly, `set_xlabel` gives a name to the X axis and `set_title` the subplot title:

```
In [36]: ax.set_title('My first matplotlib plot')  
Out[36]: <matplotlib.text.Text at 0x7f64b77d98d0>  
  
In [37]: ax.set_xlabel('Stages')
```



**Figure 9-8.** Simple plot for illustrating xticks

See [Figure 9-8](#) for the resulting figure. Modifying the Y axis consists of the same process, substituting `y` for `x` in the above.

## Adding legends

Legends are another critical element for identifying plot elements. There are a couple of ways to add one. The easiest is to pass the `label` argument when adding each piece of the plot:

```
In [38]: from numpy.random import randn
```

```
In [39]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)
```

```
In [40]: ax.plot(randn(1000).cumsum(), 'k', label='one')
Out[40]: [

```

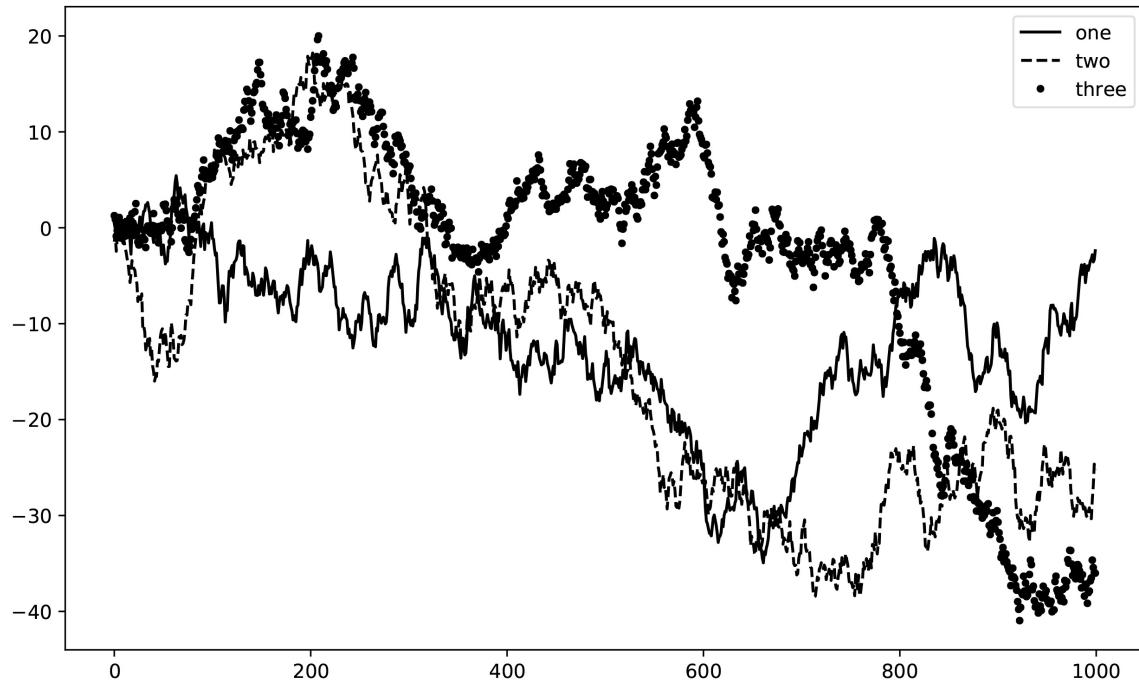
```
In [41]: ax.plot(randn(1000).cumsum(), 'k--', label='two')
Out[41]: [

```

```
In [42]: ax.plot(randn(1000).cumsum(), 'k.', label='three')
Out[42]: [
```

Once you've done this, you can either call `ax.legend()` or `plt.legend()` to automatically create a legend:

```
In [43]: ax.legend(loc='best')
```



**Figure 9-9.** Simple plot with 3 lines and legend

See [Figure 9-9](#). The `loc` tells matplotlib where to place the plot. If you aren't picky '`best`' is a good option, as it will choose a location that is most out of the way. To exclude one or more elements from the legend, pass no label or `label='nolegend'`.

# Annotations and Drawing on a Subplot

In addition to the standard plot types, you may wish to draw your own plot annotations, which could consist of text, arrows, or other shapes.

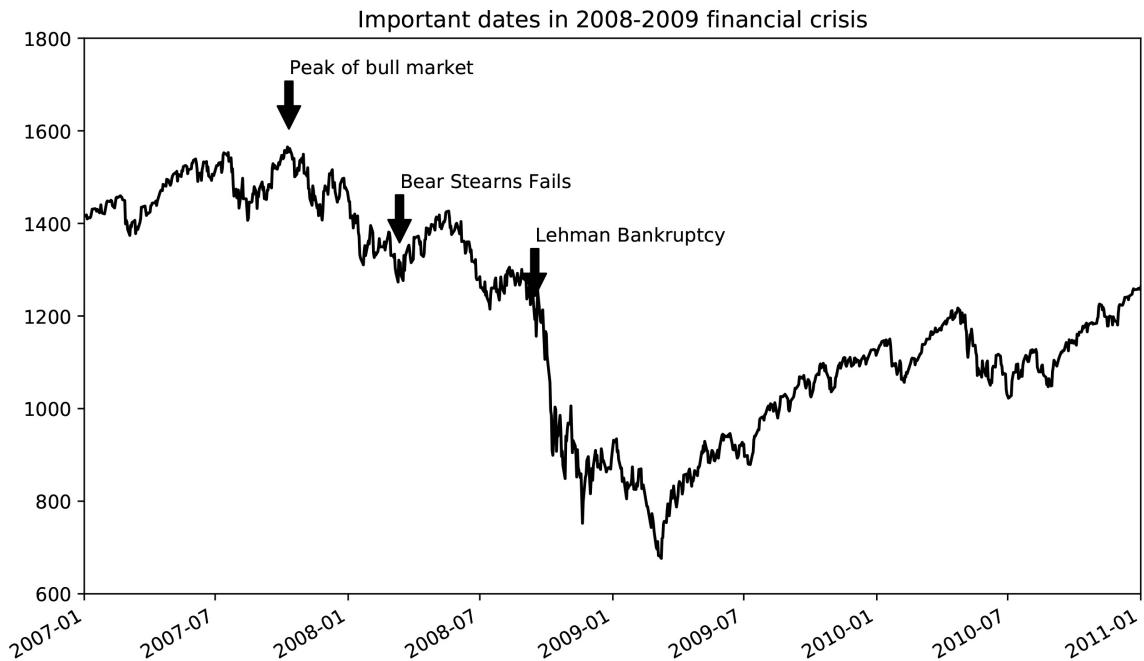
Annotations and text can be added using the `text`, `arrow`, and `annotate` functions. `text` draws text at given coordinates `(x, y)` on the plot with optional custom styling:

```
ax.text(x, y, 'Hello world!',  
        family='monospace', fontsize=10)
```

Annotations can draw both text and arrows arranged appropriately. As an example, let's plot the closing S&P 500 index price since 2007 (obtained from Yahoo! Finance) and annotate it with some of the important dates from the 2008-2009 financial crisis. See [Figure 9-10](#) for the result:

```
from datetime import datetime  
  
fig = plt.figure()  
ax = fig.add_subplot(1, 1, 1)  
  
data = pd.read_csv('examples/spx.csv', index_col=0, parse_dates=True)  
spx = data['SPX']  
  
spx.plot(ax=ax, style='k-')  
  
crisis_data = [  
    (datetime(2007, 10, 11), 'Peak of bull market'),  
    (datetime(2008, 3, 12), 'Bear Stearns Fails'),  
    (datetime(2008, 9, 15), 'Lehman Bankruptcy')  
]  
  
for date, label in crisis_data:  
    ax.annotate(label, xy=(date, spx.asof(date) + 50),  
               xytext=(date, spx.asof(date) + 200),  
               arrowprops=dict(facecolor='black'),  
               horizontalalignment='left', verticalalignment='bottom')  
  
# Zoom in on 2007-2010  
ax.set_xlim(['1/1/2007', '1/1/2011'])
```

```
ax.set_ylim([600, 1800])  
ax.set_title('Important dates in 2008-2009 financial crisis')
```

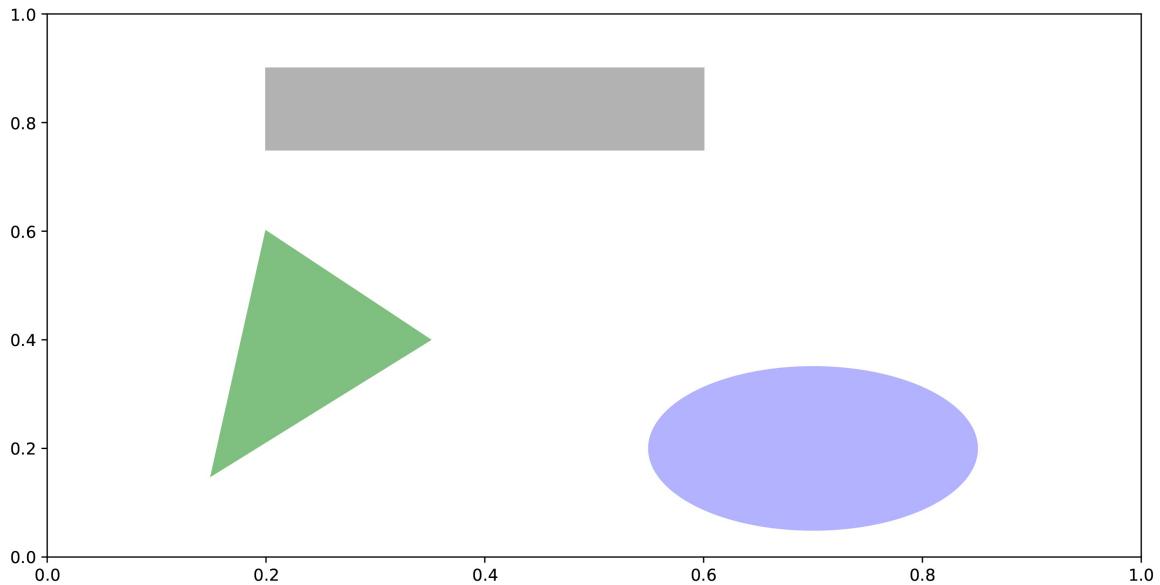


**Figure 9-10. Important dates in 2008-2009 financial crisis**

See the online matplotlib gallery for many more annotation examples to learn from.

Drawing shapes requires some more care. matplotlib has objects that represent many common shapes, referred to as *patches*. Some of these, like `Rectangle` and `Circle` are found in `matplotlib.pyplot`, but the full set is located in `matplotlib.patches`.

To add a shape to a plot, you create the patch object `shp` and add it to a subplot by calling `ax.add_patch(shp)` (see [Figure 9-11](#)):



**Figure 9-11. Figure composed from 3 different patches**

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.5)
circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]], 
                   color='g', alpha=0.5)

ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)
```

If you look at the implementation of many familiar plot types, you will see that they are assembled from patches.

# Saving Plots to File

The active figure can be saved to file using `plt.savefig`. This method is equivalent to the figure object's `savefig` instance method. For example, to save an SVG version of a figure, you need only type:

```
plt.savefig('figpath.svg')
```

The file type is inferred from the file extension. So if you used `.png` instead you would get a PDF. There are a couple of important options that I use frequently for publishing graphics: `dpi`, which controls the dots-per-inch resolution, and `bbox_inches`, which can trim the whitespace around the actual figure. To get the same plot as a PNG above with minimal whitespace around the plot and at 400 DPI, you would do:

```
plt.savefig('figpath.png', dpi=400, bbox_inches='tight')
```

`savefig` doesn't have to write to disk; it can also write to any file-like object, such as a `BytesIO`:

```
from io import BytesIO
buffer = BytesIO()
plt.savefig(buffer)
plot_data = buffer.getvalue()
```

This can be useful for serving dynamically-generated images over the web.

Table 9-2. `Figure.savefig` options

| Argument   | Description  |
|--|--|
| <code>fname</code>                                 | String containing a filepath or a Python file-like object. The figure format is inferred from the file extension, e.g. <code>.png</code> for PDF or <code>.png</code> for PNG. |
| <code>dpi</code>                                   | The figure resolution in dots per inch; defaults to 100 out of the box but can be configured   |
| <code>facecolor</code> ,<br><code>edgecolor</code> | The color of the figure background outside of the subplots. ' <code>w</code> ' (white), by default   |

`format`      The explicit file format to use ('png', 'pdf', 'svg',  
                  'ps', 'eps', ...)  
`bbox_inches`    The portion of the figure to save. If 'tight' is passed, will  
                  attempt to trim the empty space around the figure

# matplotlib Configuration

matplotlib comes configured with color schemes and defaults that are geared primarily toward preparing figures for publication. Fortunately, nearly all of the default behavior can be customized via an extensive set of global parameters governing figure size, subplot spacing, colors, font sizes, grid styles, and so on. One way to interact with the matplotlib configuration system. The first is programmatically from Python using the `rc` method. For example, to set the global default figure size to be 10 x 10, you could enter:

```
plt.rc('figure', figsize=(10, 10))
```

The first argument to `rc` is the component you wish to customize, such as `'figure'`, `'axes'`, `'xtick'`, `'ytick'`, `'grid'`, `'legend'` or many others. After that can follow a sequence of keyword arguments indicating the new parameters. An easy way to write down the options in your program is as a dict:

```
font_options = {'family' : 'monospace',
                'weight' : 'bold',
                'size'   : 'small'}
plt.rc('font', **font_options)
```

For more extensive customization and to see a list of all the options, matplotlib comes with a configuration file `matplotlibrc` in the `matplotlib/mpl-data` directory. If you customize this file and place it in your home directory titled `.matplotlibrc`, it will be loaded each time you use matplotlib.

As we'll see in the next section, the seaborn package has several built-in plot themes or *styles* that use matplotlib's configuration system internally.

# Plotting with pandas and seaborn

matplotlib can be a fairly low level tool. You assemble a plot from its base components: the data display (the type of plot: line, bar, box, scatter, contour, etc.), legend, title, tick labels, and other annotations. In pandas we may have multiple columns of data, along with row and column labels. pandas itself has built-in methods which simplify creating visualizations from DataFrame and Series objects. Another library we will use is `seaborn` (<https://seaborn.pydata.org/>), a statistical graphics library created by Michael Waskom. Together, these tools simplify creating many common visualization types.

## Line Plots

Series and DataFrame each have a `plot` attribute for making some basic plot types. By default, `plot()` makes line plots (see [Figure 9-12](#)):

```
In [54]: s = pd.Series(np.random.randn(10).cumsum(), index=np.arange(10))
```

```
In [55]: s.plot()
```

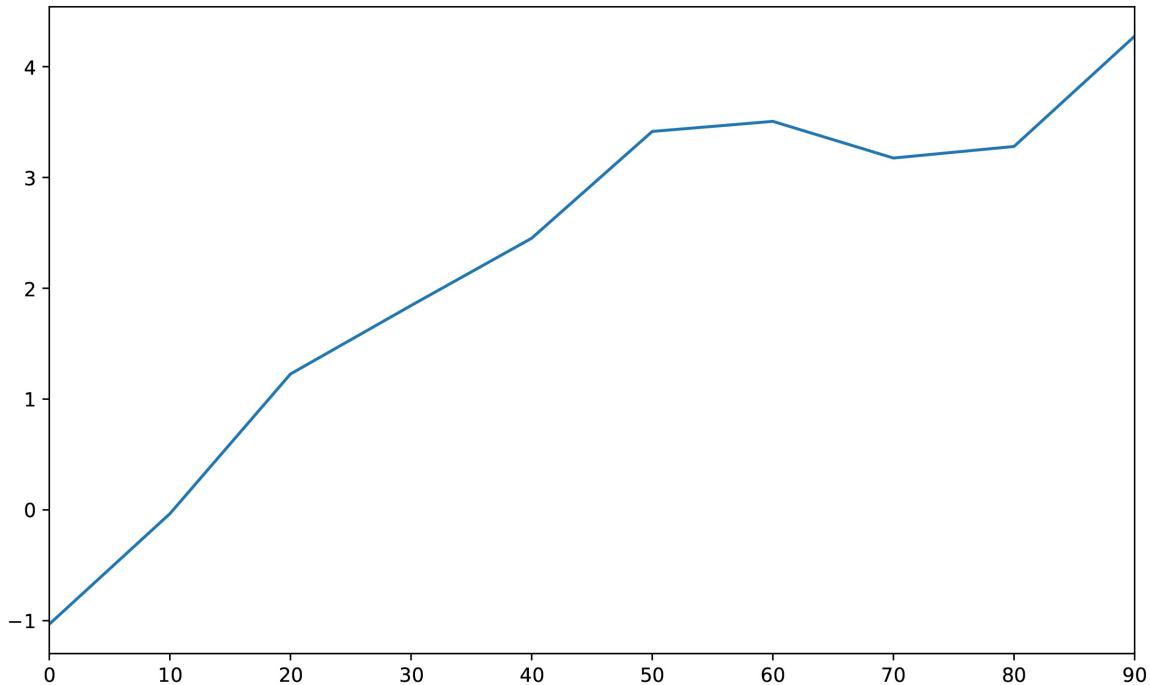


Figure 9-12. Simple Series plot example

The Series object's index is passed to matplotlib for plotting on the X axis, though this can be disabled by passing `use_index=False`. The X axis ticks and limits can be adjusted using the `xticks` and `xlim` options, and Y axis respectively using `yticks` and `ylim`. See [Table 9-3](#) for a full listing of `plot` options. I'll comment on a few more of them throughout this section and leave the rest to you to explore.

Most of pandas's plotting methods accept an optional `ax` parameter, which can be a matplotlib subplot object. This gives you more flexible placement of subplots in a grid layout.

DataFrame's `plot` method plots each of its columns as a different line on the same subplot, creating a legend automatically (see [Figure 9-13](#)):

```
In [56]: df = pd.DataFrame(np.random.randn(10, 4).cumsum(0),
....:                         columns=['A', 'B', 'C', 'D'],
....:                         index=np.arange(0, 100, 10))

In [57]: df.plot()
```

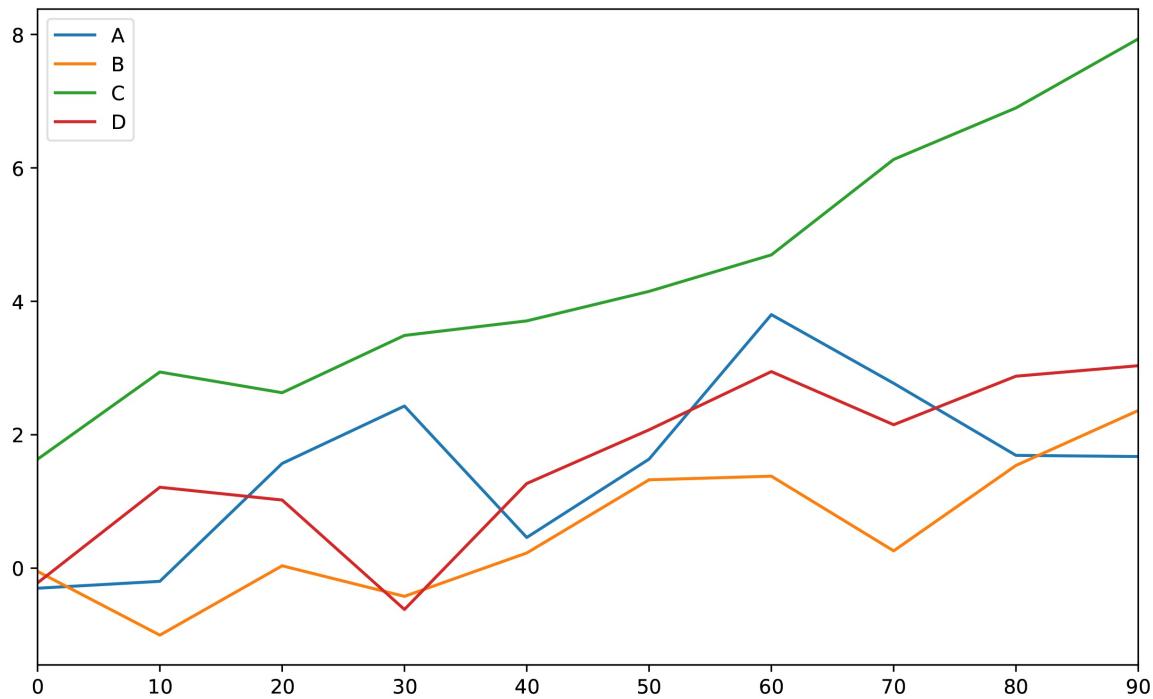


Figure 9-13. Simple DataFrame plot example

The `plot` attribute contains a “family” of methods for different plot types. For example, `df.plot()` is equivalent to `df.plot.line()`. We’ll explore some of these methods next.

#### Note

Additional keyword arguments to `plot` are passed through to the respective matplotlib plotting function, so you can further customize these plots by learning more about the matplotlib API.

Table 9-3. Series.plot method arguments

| Argument           | Description   |
|--------------------|---|
| <code>label</code> | Label for plot legend   |
| <code>ax</code>    | matplotlib subplot object to plot on. If nothing passed, uses active matplotlib subplot |
| <code>style</code> | Style string, like ' <code>'ko--'</code> ', to be passed to matplotlib.                 |
| <code>alpha</code> | The plot fill opacity (from 0 to 1)   |

|           |   |
|-----------|---|
| kind      | Can be 'area', 'bar', 'barh', 'density', 'hist', 'kde', 'line', 'pie' |
| logy      | Use logarithmic scaling on the Y axis                                 |
| use_index | Use the object index for tick labels                                  |
| rot       | Rotation of tick labels (0 through 360)                               |
| xticks    | Values to use for X axis ticks  |
| yticks    | Values to use for Y axis ticks  |
| xlim      | X axis limits (e.g. [0, 10])  |
| ylim      | Y axis limits   |
| grid      | Display axis grid (on by default)                                     |

DataFrame has a number of options allowing some flexibility with how the columns are handled; for example, whether to plot them all on the same subplot or to create separate subplots. See [Table 9-4](#) for more on these.

Table 9-4. DataFrame-specific plot arguments

| Argument     | Description   |
|--------------|---|
| subplots     | Plot each DataFrame column in a separate subplot                                |
| sharex       | If <code>subplots=True</code> , share the same X axis, linking ticks and limits |
| sharey       | If <code>subplots=True</code> , share the same Y axis                           |
| figsize      | Size of figure to create as tuple   |
| title        | Plot title as string  |
| legend       | Add a subplot legend ( <code>True</code> by default)                            |
| sort_columns | Plot columns in alphabetical order; by default uses existing column order       |

#### Note

For time series plotting, see [Chapter 11](#).

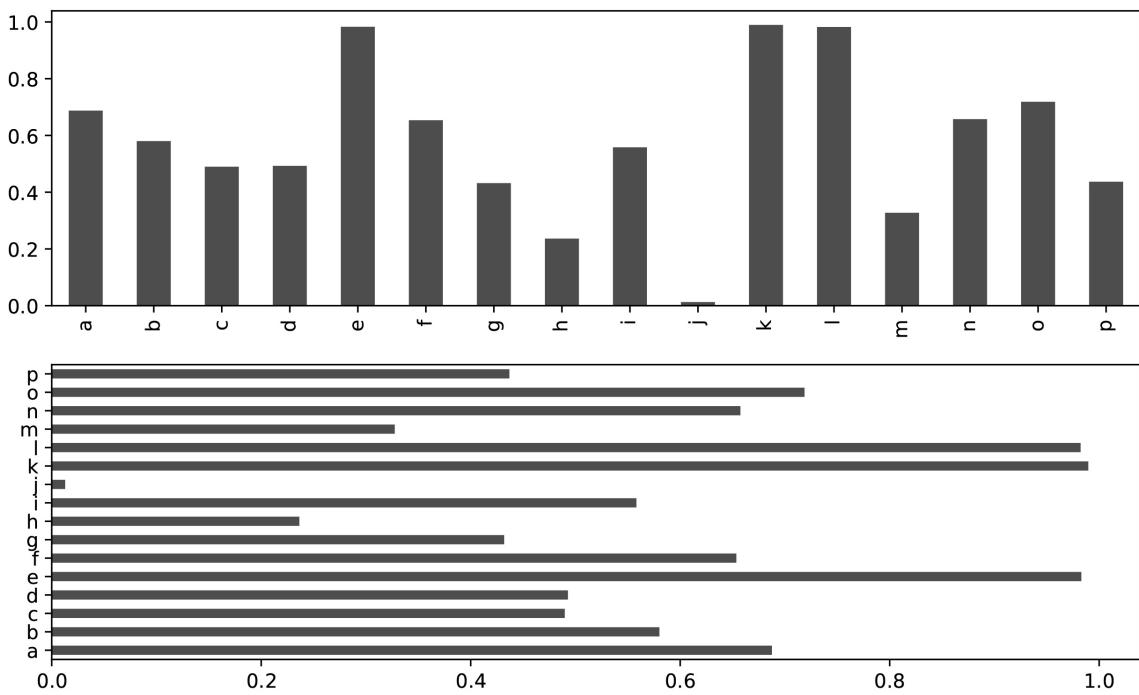
# Bar Plots

The `plot.bar()` and `plot.bart()` make vertical and horizontal bar plots, respectively. In this case, the Series or DataFrame index will be used as the X (`bar`) or Y (`bart`) ticks (see [Figure 9-14](#)):

```
In [58]: fig, axes = plt.subplots(2, 1)

In [59]: data = pd.Series(np.random.rand(16), index=list('abcde'
In [60]: data.plot.bar(ax=axes[0], color='k', alpha=0.7)
Out[60]: <matplotlib.axes._subplots.AxesSubplot at 0x7f64b73f4c>

In [61]: data.plot.bart(ax=axes[1], color='k', alpha=0.7)
```



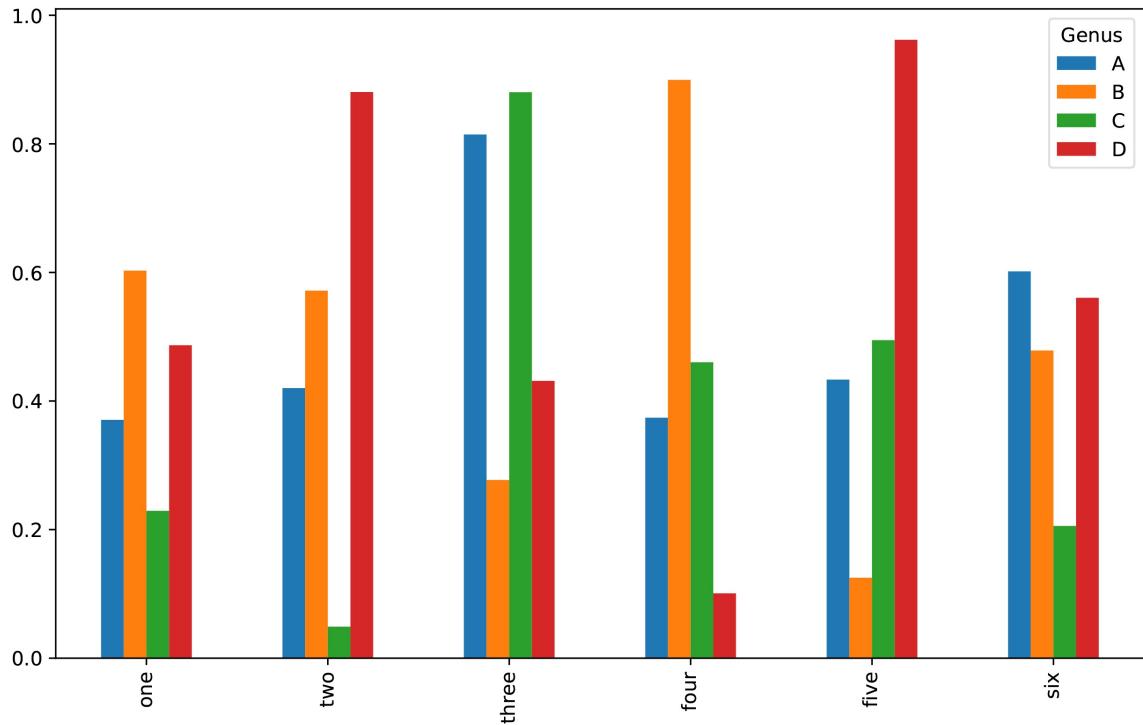
**Figure 9-14. Horizontal and vertical bar plot example**

With a DataFrame, bar plots group the values in each row together in a group in bars, side by side, for each value. See [Figure 9-15](#):

```
In [63]: df = pd.DataFrame(np.random.rand(6, 4),
....:                      index=['one', 'two', 'three', 'four',
....:                             columns=pd.Index(['A', 'B', 'C', 'D'])

In [64]: df
Out[64]:
   Genus      A      B      C      D
one    0.370670  0.602792  0.229159  0.486744
two    0.420082  0.571653  0.049024  0.880592
three  0.814568  0.277160  0.880316  0.431326
four   0.374020  0.899420  0.460304  0.100843
five   0.433270  0.125107  0.494675  0.961825
six    0.601648  0.478576  0.205690  0.560547

In [65]: df.plot.bar()
```

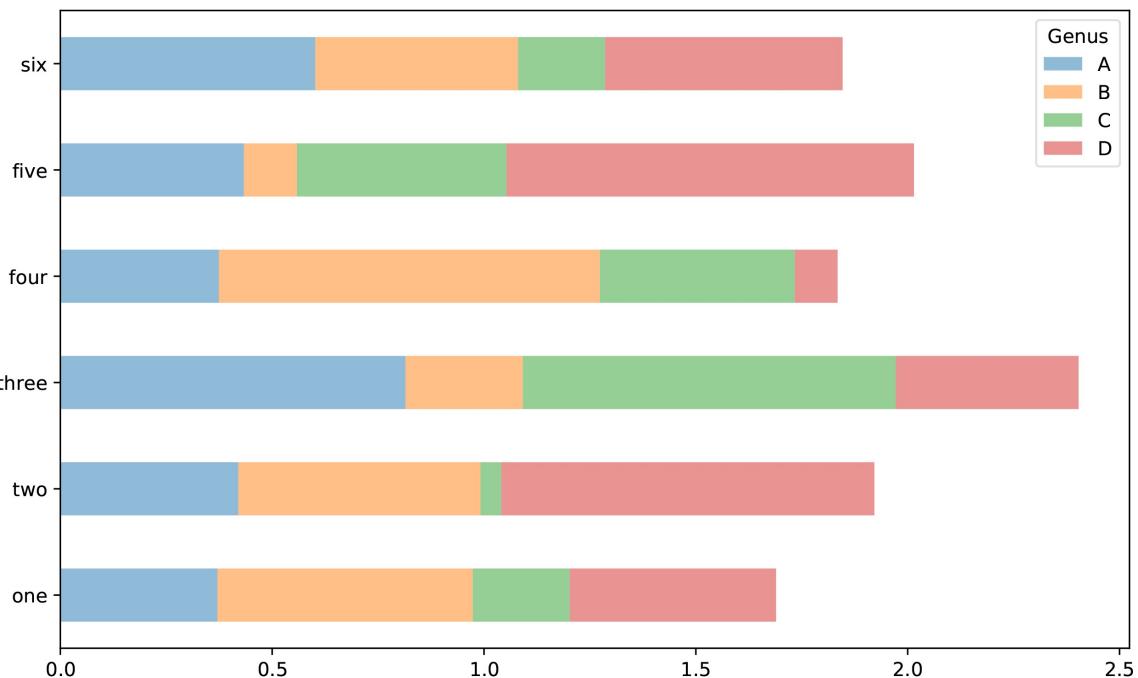


**Figure 9-15. DataFrame bar plot example**

Note that the name “Genus” on the DataFrame’s columns is used to title the legend.

Stacked bar plots are created from a DataFrame by passing `stacked=True`, resulting in the value in each row being stacked together (see [Figure 9-16](#)):

```
In [67]: df.plot.barh(stacked=True, alpha=0.5)
```



**Figure 9-16.** DataFrame stacked bar plot example

#### Note

A useful recipe for bar plots is to visualize a Series's value frequency using  
`value_counts: s.value_counts().plot.bar()`

Returning to the tipping data set used earlier in the book, suppose we wanted to make a stacked bar plot showing the percentage of data points for each party size on each day. I load the data using `read_csv` and make a cross-tabulation by day and party size:

```
In [69]: tips = pd.read_csv('examples/tips.csv')

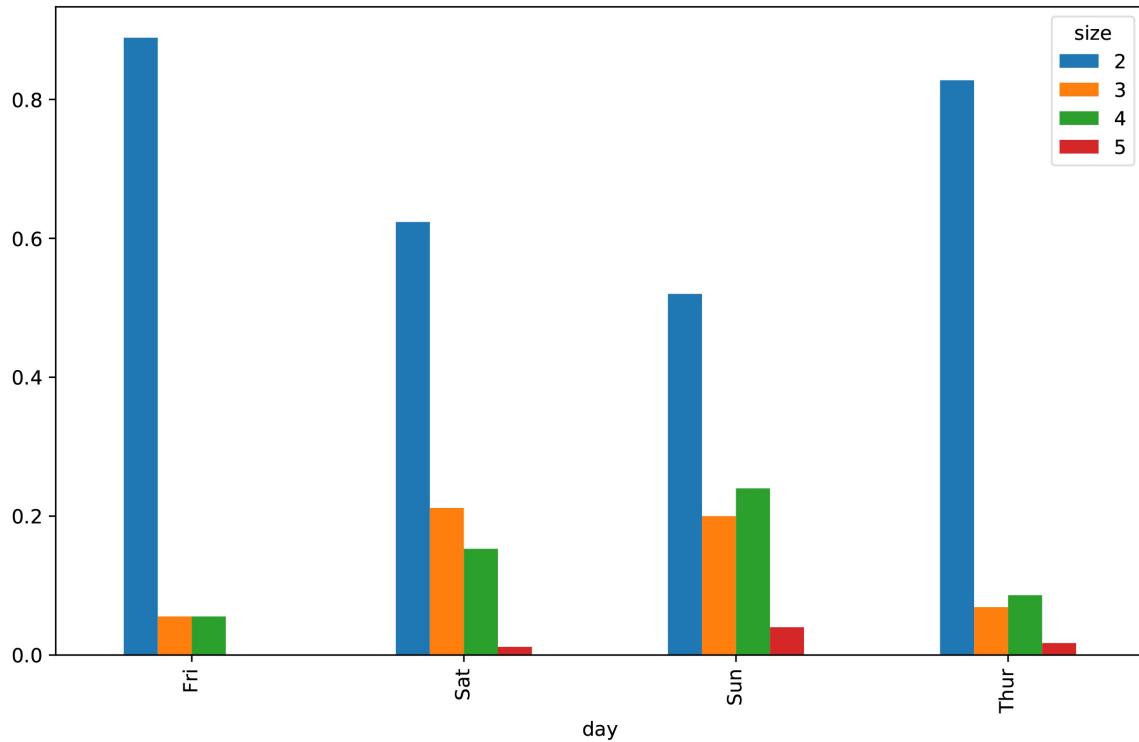
In [70]: party_counts = pd.crosstab(tips['day'], tips['size'])

In [71]: party_counts
Out[71]:
size   1    2    3    4    5    6
day
Fri     1   16    1    1    0    0
Sat     2   53   18   13    1    0
Sun     0   39   15   18    3    1
```

```
Thur   1   48    4    5    1    3  
  
# Not many 1- and 6-person parties  
In [72]: party_counts = party_counts.loc[:, 2:5]
```

Then, normalize so that each row sums to 1 and make the plot (see [Figure 9-17](#)):

```
# Normalize to sum to 1  
In [73]: party_pcts = party_counts.div(party_counts.sum(1), axis=1)  
  
In [74]: party_pcts  
Out[74]:  
size      2          3          4          5  
day  
Fri    0.8888889  0.0555556  0.0555556  0.0000000  
Sat    0.623529   0.211765   0.152941   0.011765  
Sun    0.520000   0.200000   0.240000   0.040000  
Thur   0.827586   0.068966   0.086207   0.017241  
  
In [75]: party_pcts.plot.bar()
```



**Figure 9-17. Fraction of parties by size on each day**

So you can see that party sizes appear to increase on the weekend in this data set.

With data that requires aggregation or summarization before making a plot, using the `seaborn` package can make things much simpler. Let's look now at the tipping percentage by day with `seaborn`:

```
In [77]: import seaborn as sns

In [78]: tips['tip_pct'] = tips['tip'] / (tips['total_bill'] - 

In [79]: tips.head()
Out[79]:
   total_bill    tip      sex smoker  day    time  size  tip_pct
0      16.99  1.01  Female     No  Sun  Dinner    2  0.063204
1      10.34  1.66    Male     No  Sun  Dinner    3  0.191244
2      21.01  3.50    Male     No  Sun  Dinner    3  0.199886
3      23.68  3.31    Male     No  Sun  Dinner    2  0.162494
4      24.59  3.61  Female     No  Sun  Dinner    4  0.172064
```

```
In [80]: sns.barplot(x='tip_pct', y='day', data=tips, orient='l')
```

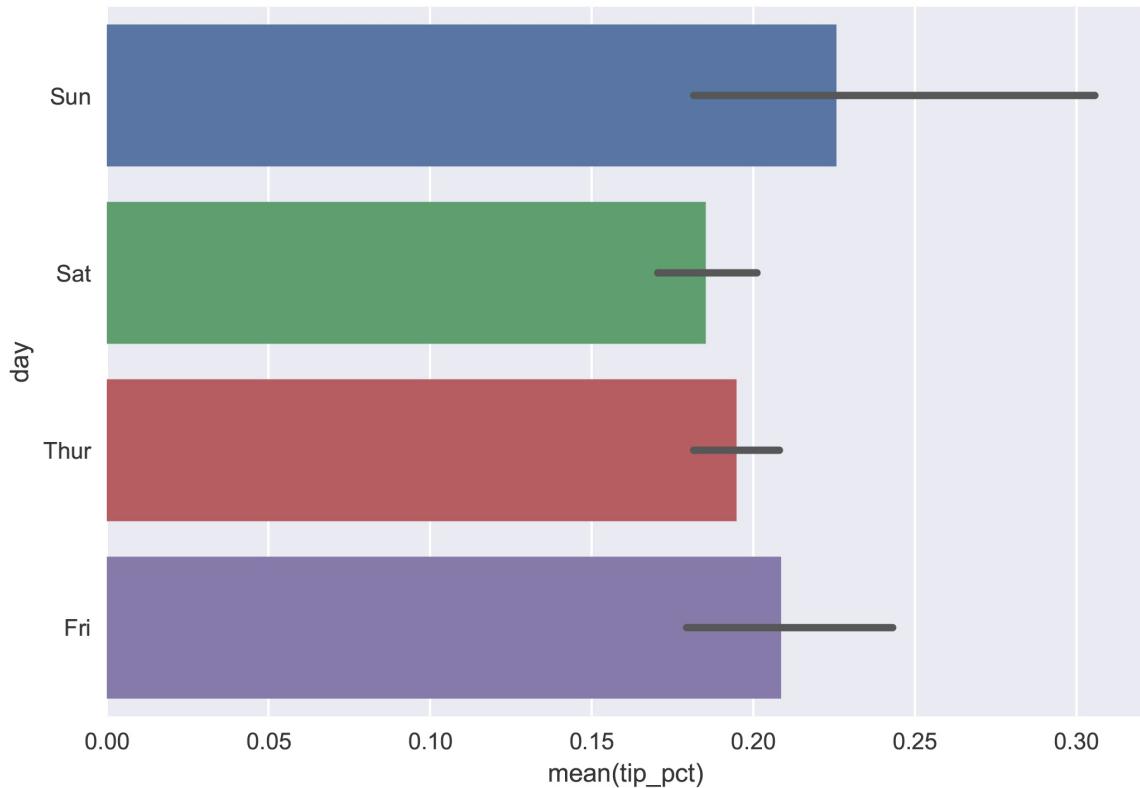


Figure 9-18. Tipping percentage by day with error bars

See [Figure 9-18](#) for the resulting plot. Plotting functions in seaborn take a `data` argument which can be a pandas DataFrame. The other arguments refer to column names. Because there are multiple observations for each value in the `day`, the bars are the average value of `tip_pct`. The black lines drawn on the bars represent the 95% confidence interval (this can be configured through optional arguments).

`seaborn.barplot` has a `hue` option which enables us to split by an additional categorical value:

```
In [82]: sns.barplot(x='tip_pct', y='day', hue='time', data=tip)
```

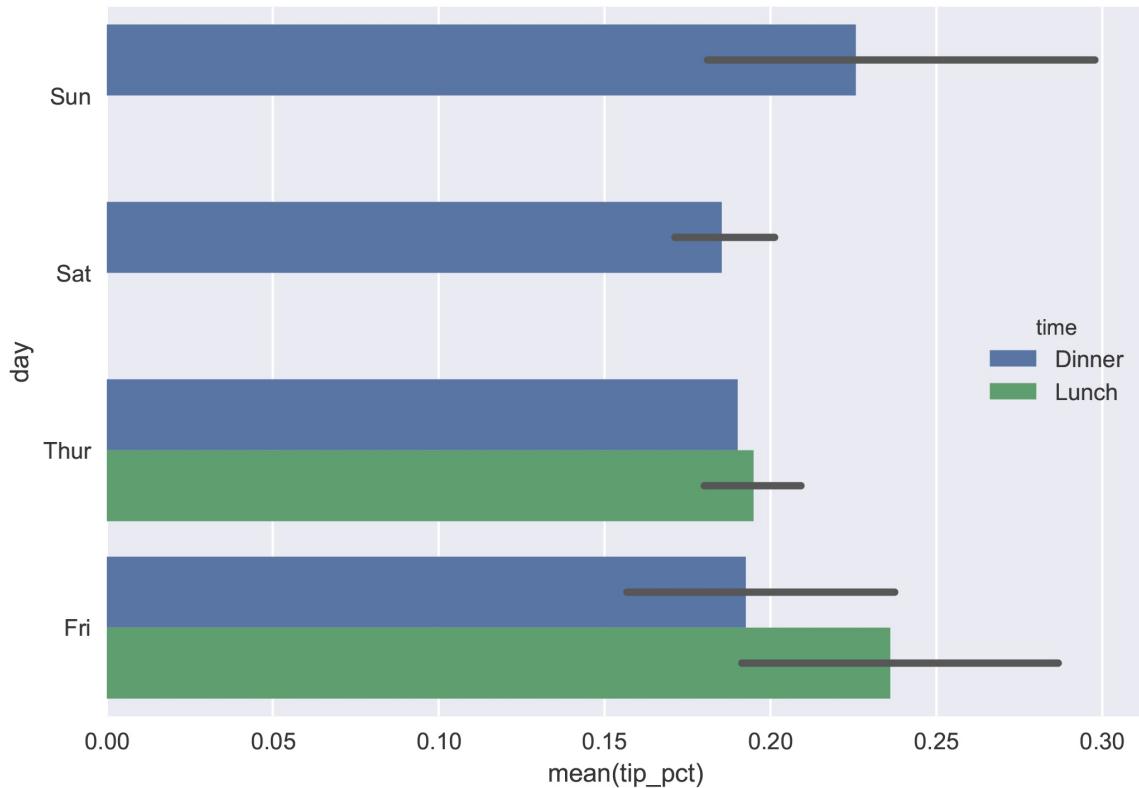


Figure 9-19. Tipping percentage by day and time

Notice that seaborn has automatically changed the aesthetics of plots: the default color palette, plot background, and grid line colors. You can switch between different plot appearances using `seaborn.set`:

```
In [84]: sns.set(style="whitegrid")
```

# Histograms and Density Plots

A histogram is a kind of bar plot that gives a discretized display of value frequency. The data points are split into discrete, evenly spaced bins, and the number of data points in each bin is plotted. Using the tipping data from before, we can make a histogram of tip percentages of the total bill using the `plot.hist` method on the Series (see [Figure 9-20](#)):

```
In [86]: tips['tip_pct'].plot.hist(bins=50)
```

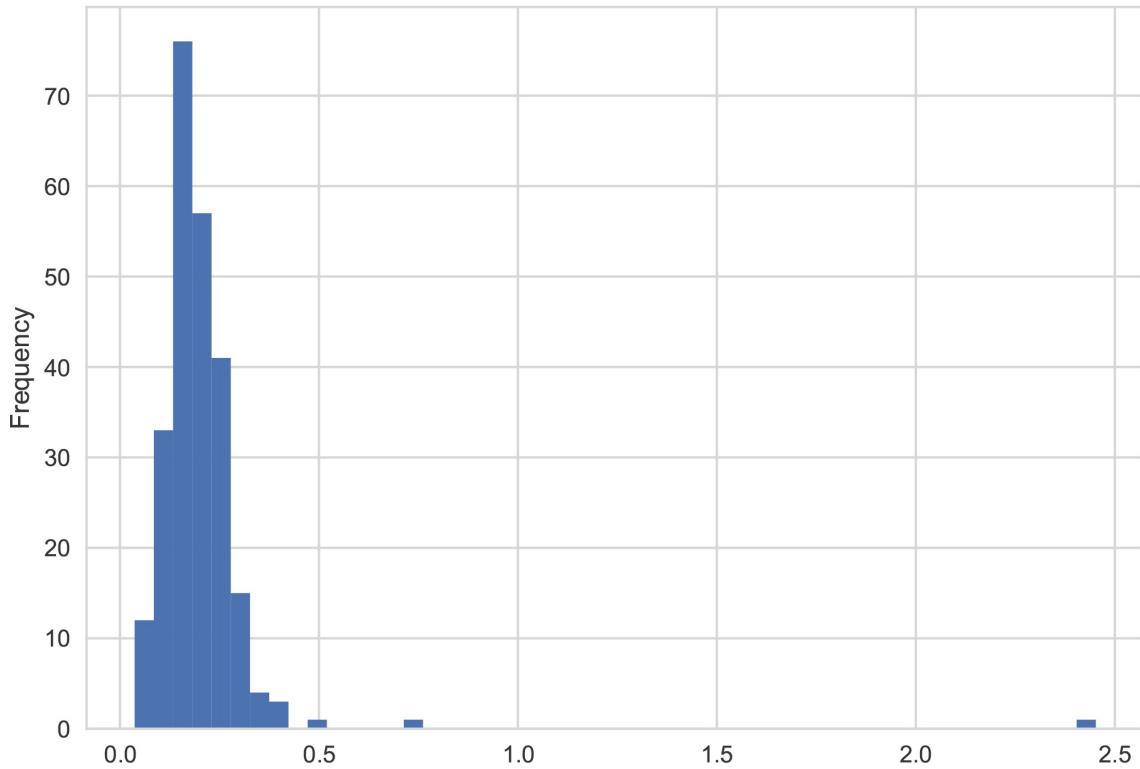
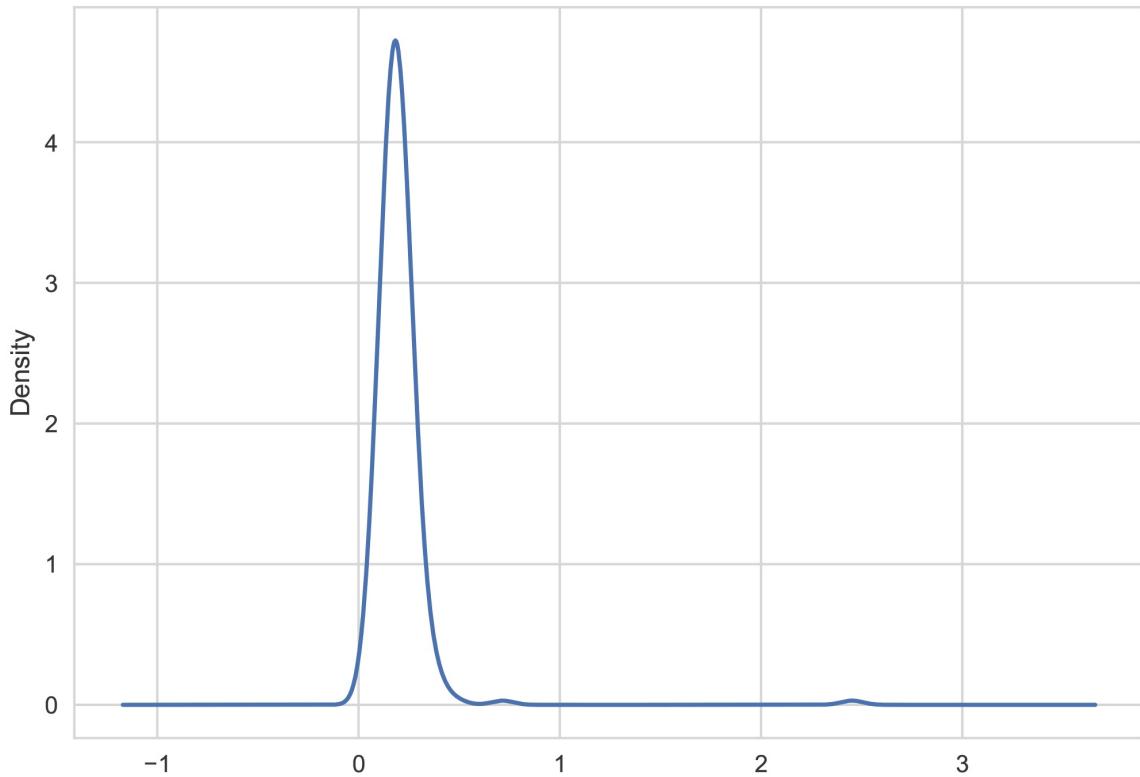


Figure 9-20. Histogram of tip percentages

A related plot type is a *density plot*, which is formed by computing an estimate of a continuous probability distribution that might have generated the observed data. A usual procedure is to approximate this distribution as a mixture of “kernels”, that is, simpler distributions like the normal distribution. Thus, density plots are also known as KDE (kernel density estimate) plots. Using `plot.kde` makes a density plot using the conventional mixture-of-normals estimate (see [Figure 9-21](#)):

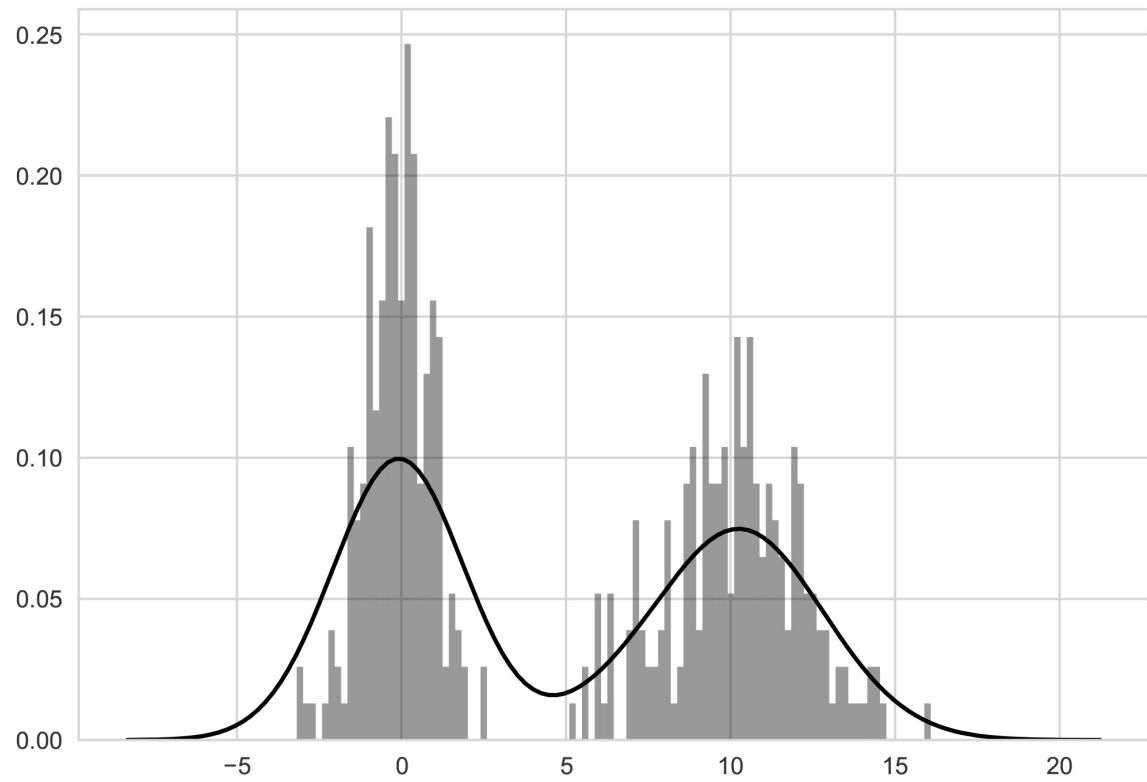
```
In [88]: tips['tip_pct'].plot.density()
```



**Figure 9-21.** Density plot of tip percentages

Seaborn makes histograms and density plots even easier through its `distplot` method, which can plot both a histogram and a continuous density estimate simultaneously. As an example, consider a bimodal distribution consisting of draws from two different standard normal distributions (see [Figure 9-22](#)):

```
In [90]: comp1 = np.random.normal(0, 1, size=200) # Normal(0,  
In [91]: comp2 = np.random.normal(10, 2, size=200) # Normal(10,  
In [92]: values = pd.Series(np.concatenate([comp1, comp2]))  
In [93]: sns.distplot(values, bins=100, color='k')
```



**Figure 9-22. Normalized histogram of normal mixture with density estimate**

# Scatter or Point Plots

Point plots (also sometimes called *scatter* plots) can be a useful way of examining the relationship between two one-dimensional data series. To give an example, I load the `macrodata` dataset from the `statsmodels` project, select a few variables, then compute log differences:

```
In [94]: macro = pd.read_csv('examples/macrodata.csv')

In [95]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]

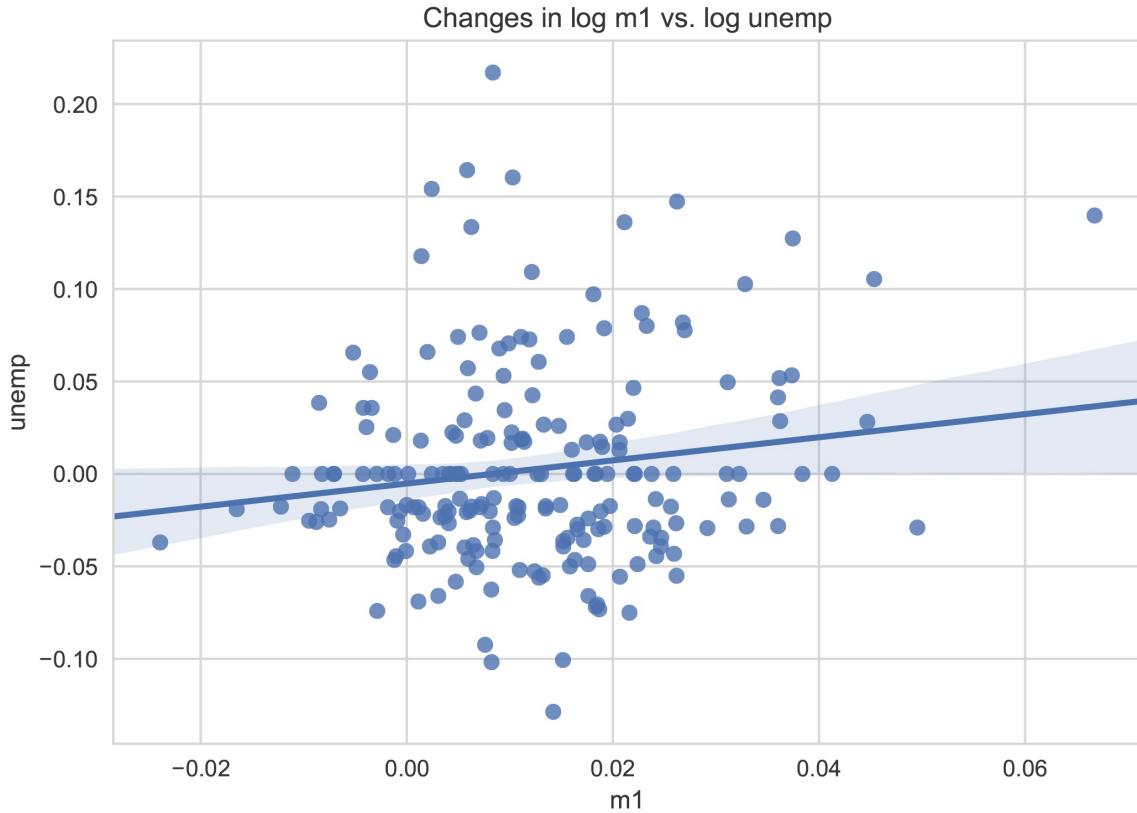
In [96]: trans_data = np.log(data).diff().dropna()

In [97]: trans_data[-5:]
Out[97]:
      cpi      m1    tbilrate    unemp
198 -0.007904  0.045361 -0.396881  0.105361
199 -0.021979  0.066753 -2.277267  0.139762
200  0.002340  0.010286  0.606136  0.160343
201  0.008419  0.037461 -0.200671  0.127339
202  0.008894  0.012202 -0.405465  0.042560
```

Now, I use `seaborn`'s `regplot` method, which make a scatter plot and fits a linear regression line (see [Figure 9-23](#)):

```
In [99]: sns.regplot('m1', 'unemp', data=trans_data)
Out[99]: <matplotlib.axes._subplots.AxesSubplot at 0x7f64ace03c>

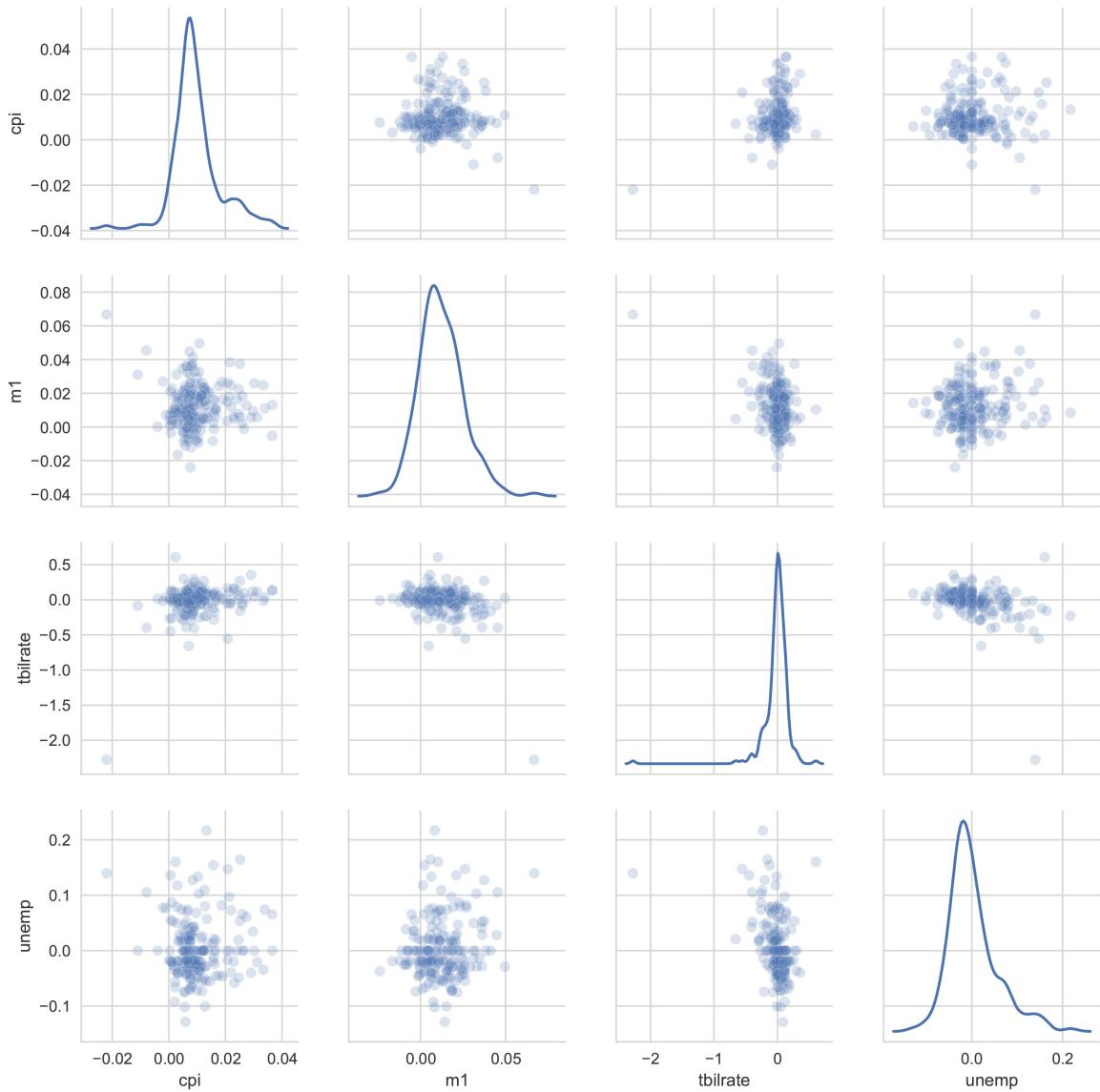
In [100]: plt.title('Changes in log %s vs. log %s' % ('m1', 'ur'))
```



**Figure 9-23.** A Seaborn regression / scatter plot

In exploratory data analysis it's helpful to be able to look at all the scatter plots among a group of variables; this is known as a *pairs* plot or *scatter plot matrix*. Making such a plot from scratch is a bit of work, so Seaborn has a convenient `pairplot` function. It supports placing histograms or density estimates of each variable along the diagonal. See [Figure 9-24](#) for the resulting plot:

```
In [101]: sns.pairplot(trans_data, diag_kind='kde', plot_kws={
```



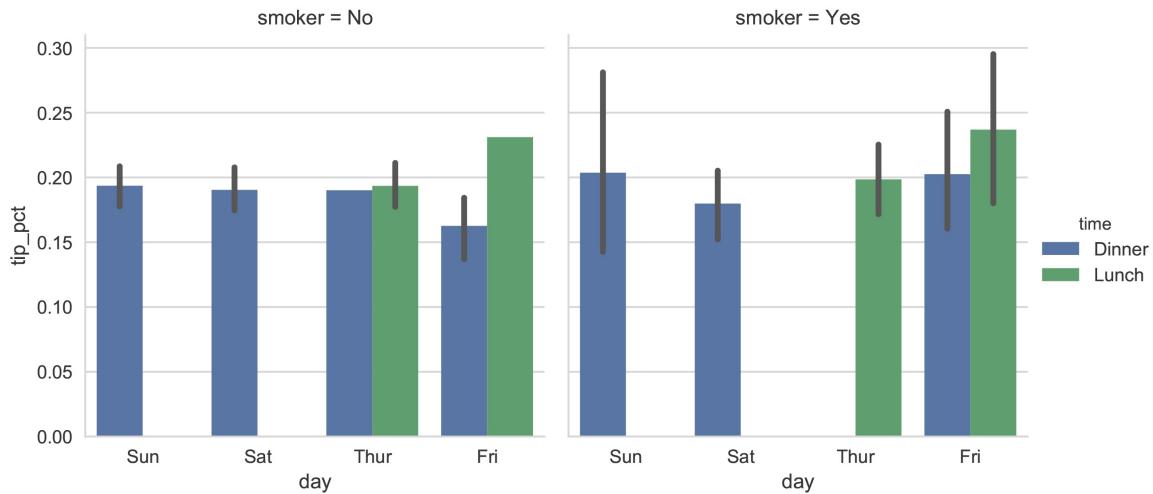
**Figure 9-24. Pair plot matrix of statsmodels macro data**

You may notice the `plot_kws` argument. This enables us to pass down configuration options to the individual plotting calls on the off-diagonal elements. Check out the `seaborn.pairplot` docstring for more granular configuration options.

## Facet grids and categorical data

Above we've shown grouped bar plots using seaborn. What about datasets where we have additional grouping dimensions? One way to visualize data with many categorical variables is to use a *facet grid*. Seaborn has a useful built-in function `factorplot` that simplifies making many kinds of faceted plots:

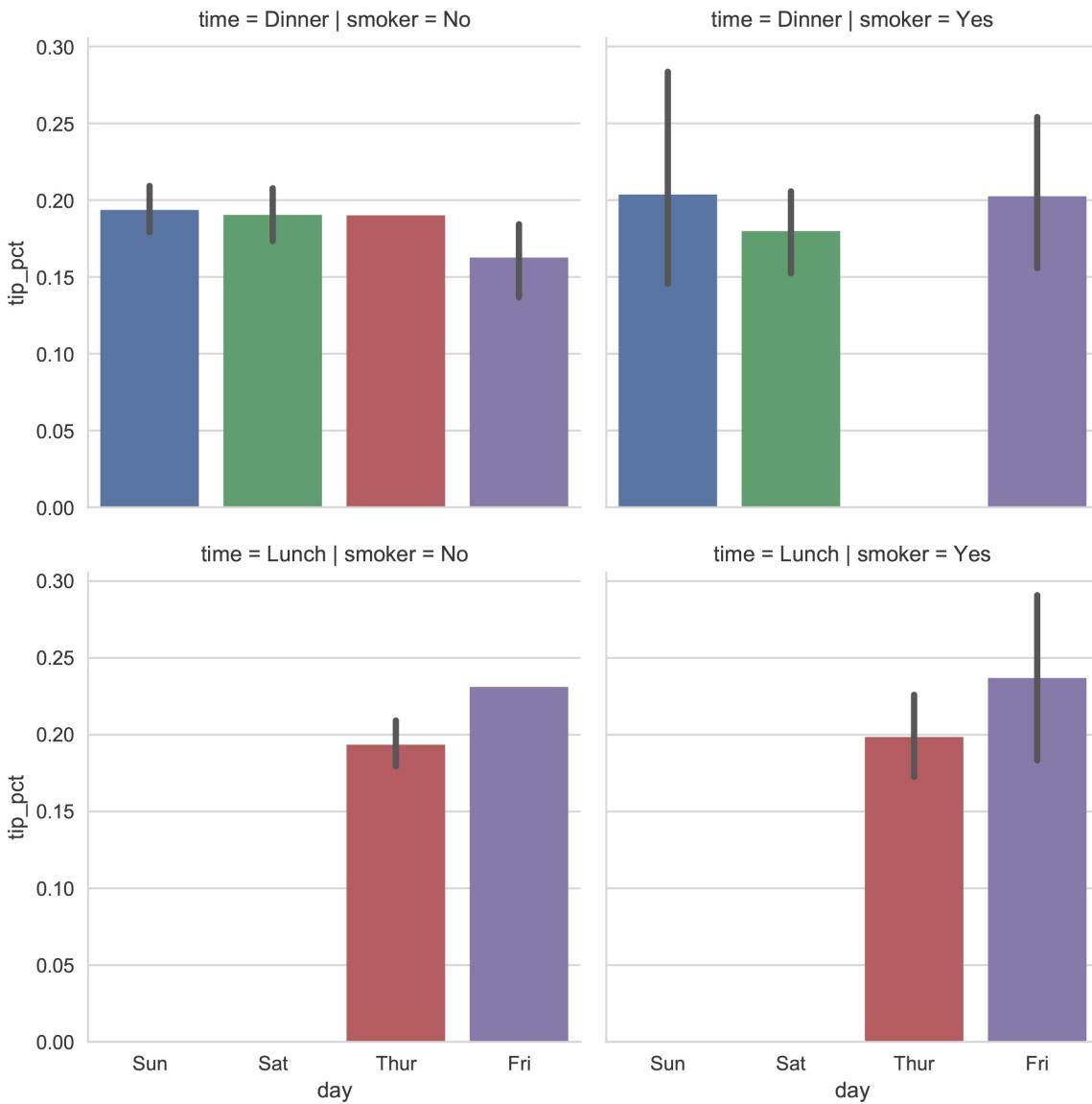
```
In [102]: sns.factorplot(x='day', y='tip_pct', hue='time', col=.....:  
.....: kind='bar', data=tips[tips.tip_pct < :
```



**Figure 9-25. Tipping percentage by day / time / smoker**

Instead of grouping by 'time' by different bar colors within a facet, we can also expand the facet grid by adding one row per `time` value:

```
In [103]: sns.factorplot(x='day', y='tip_pct', row='time',
.....:                 col='smoker',
.....:                 kind='bar', data=tips[tips.tip_pct < :
```



**Figure 9-26.** tip\_pct by day; facet by time / smoker

`factorplot` supports other plot types which may be useful depending on what you are trying to display. For example, box plots (which show the median, quartiles, and outliers) can be an effective visualization type:

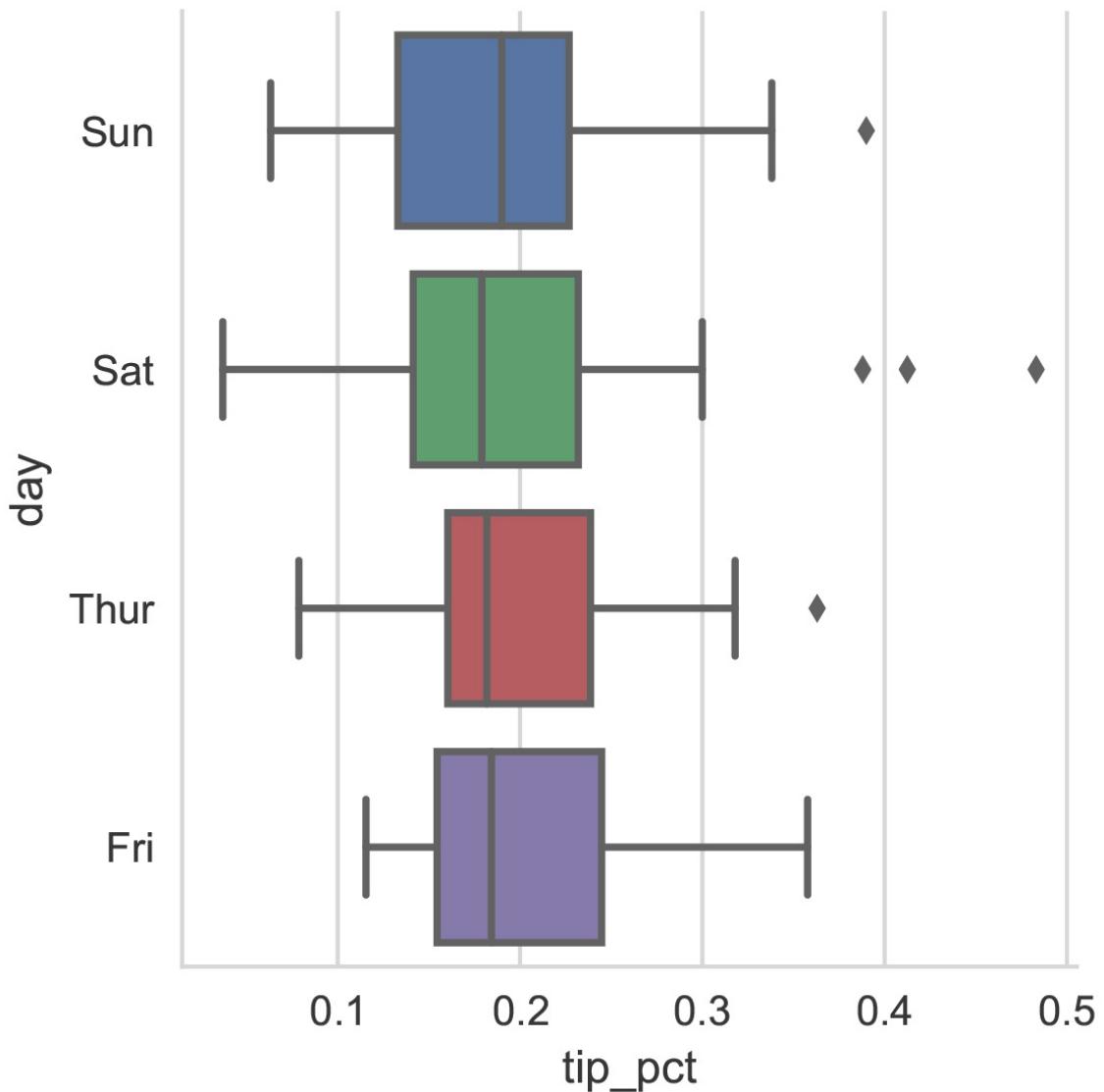


Figure 9-27. Box plot of tip\_pct by day

You can create your own facet grid plots using the more general `seaborn.FacetGrid` class. See the seaborn documentation for more (<https://seaborn.pydata.org/>).

# Other Python Visualization Tools

As is common with open source, there are a plethora of options for creating graphics in Python (too many to list). Since 2010, much development effort has been focused on creating interactive graphics for publication on the web. With tools like Bokeh (<http://bokeh.pydata.org/>) and Plotly (<https://github.com/plotly/plotly.py>), it's now possible to specify dynamic, interactive graphics in Python that are destined for a web browser.

For creating static graphics for print or web, I recommend defaulting to matplotlib and add-on libraries like pandas and seaborn for your needs. For other data visualization requirements, it may be useful to learn one of the other available tools out there. I encourage you to explore the ecosystem as it continues to involve and innovate into the future.

# Chapter 10. Data Aggregation and Group Operations

Categorizing a data set and applying a function to each group, whether an aggregation or transformation, is often a critical component of a data analysis workflow. After loading, merging, and preparing a data set, one may need to compute group statistics or possibly *pivot tables* for reporting or visualization purposes. pandas provides a flexible `groupby` interface, enabling you to slice and dice, and summarize data sets in a natural way.

One reason for the popularity of relational databases and SQL (which stands for “structured query language”) is the ease with which data can be joined, filtered, transformed, and aggregated. However, query languages like SQL are somewhat constrained in the kinds of group operations that can be performed. As you will see, with the expressiveness of Python and pandas, we can perform quite complex group operations by utilizing any function that accepts a pandas object or NumPy array. In this chapter, you will learn how to:

- Split a pandas object into pieces using one or more keys (in the form of functions, arrays, or DataFrame column names)
- Computing group summary statistics, like count, mean, or standard deviation, or a user-defined function
- Apply within-group transformations or other manipulations, like normalization, linear regression, rank, or subset selection
- Compute pivot tables and cross-tabulations
- Perform quantile analysis and other statistical group analyses

## Note

Aggregation of time series data, a special use case of `groupby`, is referred to

as *resampling* in this book and will receive separate treatment in [Chapter 11](#).

# GroupBy Mechanics

Hadley Wickham, an author of many popular packages for the R programming language, coined the term *split-apply-combine* for describing group operations. In the first stage of the process, data contained in a pandas object, whether a Series, DataFrame, or otherwise, is *split* into groups based on one or more *keys* that you provide. The splitting is performed on a particular axis of an object. For example, a DataFrame can be grouped on its rows (`axis=0`) or its columns (`axis=1`). Once this is done, a function is *applied* to each group, producing a new value. Finally, the results of all those function applications are *combined* into a result object. The form of the resulting object will usually depend on what's being done to the data. See [Figure 10-1](#) for a mockup of a simple group aggregation.

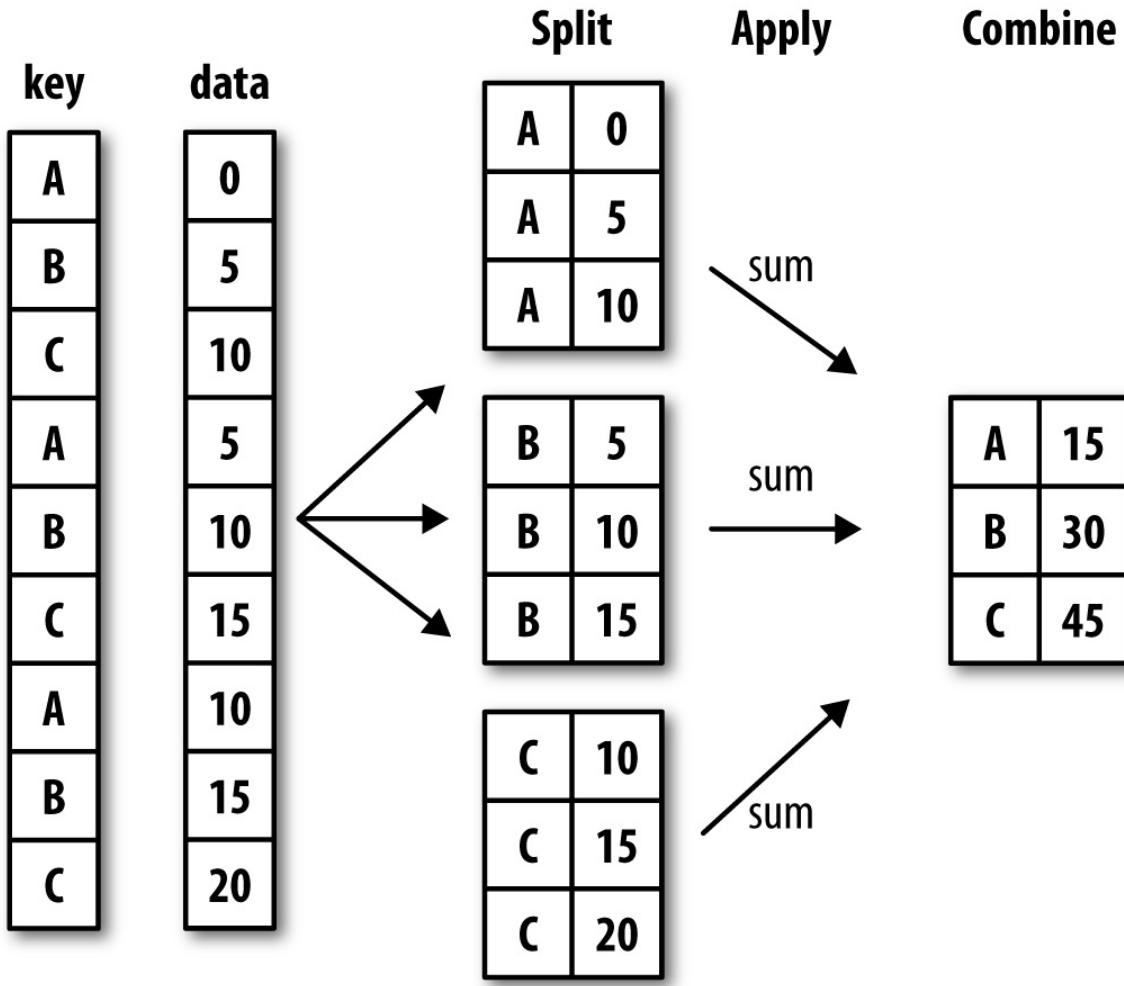


Figure 10-1. Illustration of a group aggregation

Each grouping key can take many forms, and the keys do not have to be all of the same type:

- A list or array of values that is the same length as the axis being grouped
- A value indicating a column name in a DataFrame
- A dict or Series giving a correspondence between the values on the axis being grouped and the group names
- A function to be invoked on the axis index or the individual labels in the index

Note that the latter three methods are shortcuts for producing an array of values to be used to split up the object. Don't worry if this all seems abstract.

Throughout this chapter, I will give many examples of all of these methods. To get started, here is a small tabular dataset as a DataFrame:

```
In [10]: df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
....:                   'key2' : ['one', 'two', 'one', 'two',
....:                   'data1' : np.random.randn(5),
....:                   'data2' : np.random.randn(5)})

In [11]: df
Out[11]:
      data1    data2  key1  key2
0 -0.204708  1.393406     a   one
1  0.478943  0.092908     a   two
2 -0.519439  0.281746     b   one
3 -0.555730  0.769023     b   two
4  1.965781  1.246435     a   one
```

Suppose you wanted to compute the mean of the `data1` column using the groups labels from `key1`. There are a number of ways to do this. One is to access `data1` and call `groupby` with the column (a Series) at `key1`:

```
In [12]: grouped = df['data1'].groupby(df['key1'])

In [13]: grouped
Out[13]: <pandas.core.groupby.SeriesGroupBy object at 0x7f34ef>
```

This `grouped` variable is now a *GroupBy* object. It has not actually computed anything yet except for some intermediate data about the group key `df['key1']`. The idea is that this object has all of the information needed to then apply some operation to each of the groups. For example, to compute group means we can call the GroupBy's `mean` method:

```
In [14]: grouped.mean()
Out[14]:
key1
a    0.746672
b   -0.537585
Name: data1, dtype: float64
```

Later, I'll explain more about what happens when you call `.mean()`. The important thing here is that the data (a Series) has been aggregated according to

the group key, producing a new Series that is now indexed by the unique values in the `key1` column. The result index has the name '`key1`' because the DataFrame column `df['key1']` did.

If instead we had passed multiple arrays as a list, we get something different:

```
In [15]: means = df['data1'].groupby([df['key1'], df['key2']]).  
  
In [16]: means  
Out[16]:  
key1   key2  
a      one      0.880536  
        two      0.478943  
b      one     -0.519439  
        two     -0.555730  
Name: data1, dtype: float64
```

In this case, we grouped the data using two keys, and the resulting Series now has a hierarchical index consisting of the unique pairs of keys observed:

```
In [17]: means.unstack()  
Out[17]:  
key2      one      two  
key1  
a      0.880536  0.478943  
b     -0.519439 -0.555730
```

In these examples, the group keys are all Series, though they could be any arrays of the right length:

```
In [18]: states = np.array(['Ohio', 'California', 'California',  
  
In [19]: years = np.array([2005, 2005, 2006, 2005, 2006])  
  
In [20]: df['data1'].groupby([states, years]).mean()  
Out[20]:  
California  2005      0.478943  
              2006     -0.519439  
Ohio        2005     -0.380219  
              2006      1.965781  
Name: data1, dtype: float64
```

Frequently the grouping information to be found in the same DataFrame as the data you want to work on. In that case, you can pass column names (whether

those are strings, numbers, or other Python objects) as the group keys:

```
In [21]: df.groupby('key1').mean()
Out[21]:
          data1      data2
key1
a      0.746672  0.910916
b     -0.537585  0.525384

In [22]: df.groupby(['key1', 'key2']).mean()
Out[22]:
          data1      data2
key1 key2
a    one    0.880536  1.319920
      two    0.478943  0.092908
b    one   -0.519439  0.281746
      two   -0.555730  0.769023
```

You may have noticed in the first case `df.groupby('key1').mean()` that there is no `key2` column in the result. Because `df['key2']` is not numeric data, it is said to be a *nuisance column*, which is therefore excluded from the result. By default, all of the numeric columns are aggregated, though it is possible to filter down to a subset as you'll see soon.

Regardless of the objective in using `groupby`, a generally useful GroupBy method is `size` which return a Series containing group sizes:

```
In [23]: df.groupby(['key1', 'key2']).size()
Out[23]:
key1 key2
a    one    2
      two    1
b    one    1
      two    1
dtype: int64
```

Take note that any missing values in a group key will be excluded from the result.

# Iterating Over Groups

The GroupBy object supports iteration, generating a sequence of 2-tuples containing the group name along with the chunk of data. Consider the following small example data set:

```
In [24]: for name, group in df.groupby('key1'):  
....:     print(name)  
....:     print(group)  
....:  
a  
    data1      data2 key1 key2  
0 -0.204708  1.393406   a  one  
1  0.478943  0.092908   a  two  
4  1.965781  1.246435   a  one  
b  
    data1      data2 key1 key2  
2 -0.519439  0.281746   b  one  
3 -0.555730  0.769023   b  two
```

In the case of multiple keys, the first element in the tuple will be a tuple of key values:

```
In [25]: for (k1, k2), group in df.groupby(['key1', 'key2']):  
....:     print((k1, k2))  
....:     print(group)  
....:  
('a', 'one')  
    data1      data2 key1 key2  
0 -0.204708  1.393406   a  one  
4  1.965781  1.246435   a  one  
('a', 'two')  
    data1      data2 key1 key2  
1  0.478943  0.092908   a  two  
('b', 'one')  
    data1      data2 key1 key2  
2 -0.519439  0.281746   b  one  
('b', 'two')  
    data1      data2 key1 key2  
3 -0.555730  0.769023   b  two
```

Of course, you can choose to do whatever you want with the pieces of data. A

recipe you may find useful is computing a dict of the data pieces as a one-liner:

```
In [26]: pieces = dict(list(df.groupby('key1')))
```

```
In [27]: pieces['b']
```

```
Out[27]:
```

|   | data1     | data2    | key1 | key2 |
|---|-----------|----------|------|------|
| 2 | -0.519439 | 0.281746 | b    | one  |
| 3 | -0.555730 | 0.769023 | b    | two  |

By default `groupby` groups on `axis=0`, but you can group on any of the other axes. For example, we could group the columns of our example `df` here by `dtype` like so:

```
In [28]: df.dtypes
```

```
Out[28]:
```

|  | data1   | data2   | key1   | key2   | dtype  |
|--|---------|---------|--------|--------|--------|
|  | float64 | float64 | object | object | object |

```
In [29]: grouped = df.groupby(df.dtypes, axis=1)
```

```
In [30]: dict(list(grouped))
```

```
Out[30]:
```

|   | {dtype('float64') : | data1    | data2 | dtype('O') : | key1 | key2 |
|---|---------------------|----------|-------|--------------|------|------|
| 0 | -0.204708           | 1.393406 |       |              |      |      |
| 1 | 0.478943            | 0.092908 |       |              |      |      |
| 2 | -0.519439           | 0.281746 |       |              |      |      |
| 3 | -0.555730           | 0.769023 |       |              |      |      |
| 4 | 1.965781            | 1.246435 |       |              |      |      |
| 0 | a                   | one      |       |              |      |      |
| 1 | a                   | two      |       |              |      |      |
| 2 | b                   | one      |       |              |      |      |
| 3 | b                   | two      |       |              |      |      |
| 4 | a                   | one      |       |              |      |      |

# Selecting a Column or Subset of Columns

Indexing a GroupBy object created from a DataFrame with a column name or array of column names has the effect of *selecting those columns* for aggregation. This means that:

```
df.groupby('key1')['data1']
df.groupby('key1')[['data2']]
```

are syntactic sugar for:

```
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])
```

Especially for large data sets, it may be desirable to aggregate only a few columns. For example, in the above data set, to compute means for just the data2 column and get the result as a DataFrame, we could write:

```
In [31]: df.groupby(['key1', 'key2'])[['data2']].mean()
Out[31]:
          data2
key1  key2
a      one    1.319920
      two    0.092908
b      one    0.281746
      two    0.769023
```

The object returned by this indexing operation is a grouped DataFrame if a list or array is passed and a grouped Series if only a single column name that is passed as a scalar:

```
In [32]: s_grouped = df.groupby(['key1', 'key2'])['data2']

In [33]: s_grouped
Out[33]: <pandas.core.groupby.SeriesGroupBy object at 0x7f34ef{

In [34]: s_grouped.mean()
Out[34]:
          data2
key1  key2
a      one    1.319920
      two    0.092908
```

```
b      one      0.281746
      two      0.769023
Name: data2, dtype: float64
```

# Grouping with Dicts and Series

Grouping information may exist in a form other than an array. Let's consider another example DataFrame:

```
In [35]: people = pd.DataFrame(np.random.randn(5, 5),
....:                               columns=['a', 'b', 'c', 'd', 'e']
....:                               index=['Joe', 'Steve', 'Wes', 'Jim',
....:                                     'Travis'])

In [36]: people.iloc[2:3, [1, 2]] = np.nan # Add a few NA values

In [37]: people
Out[37]:
      a        b        c        d        e
Joe  1.007189 -1.296221  0.274992  0.228913  1.352917
Steve 0.886429 -2.001637 -0.371843  1.669025 -0.438570
Wes  -0.539741       NaN       NaN -1.021228 -0.577087
Jim   0.124121  0.302614  0.523772  0.000940  1.343810
Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Now, suppose I have a group correspondence for the columns and want to sum together the columns by group:

```
In [38]: mapping = {'a': 'red', 'b': 'red', 'c': 'blue',
....:                 'd': 'blue', 'e': 'red', 'f': 'orange'}
```

Now, you could easily construct an array from this dict to pass to `groupby`, but instead we can just pass the dict:

```
In [39]: by_column = people.groupby(mapping, axis=1)

In [40]: by_column.sum()
Out[40]:
      blue        red
Joe    0.503905  1.063885
Steve  1.297183 -1.553778
Wes   -1.021228 -1.116829
Jim    0.524712  1.770545
Travis -4.230992 -2.405455
```

The same functionality holds for Series, which can be viewed as a fixed size

mapping. When I used Series as group keys in the above examples, pandas inspects each Series to ensure that its index is aligned with the axis it's grouping:

```
In [41]: map_series = pd.Series(mapping)

In [42]: map_series
Out[42]:
a      red
b      red
c    blue
d    blue
e      red
f  orange
dtype: object

In [43]: people.groupby(map_series, axis=1).count()
Out[43]:
      blue   red
Joe      2     3
Steve    2     3
Wes      1     2
Jim      2     3
Travis   2     3
```

# Grouping with Functions

Using Python functions is a more generic way of defining a group mapping compared with a dict or Series. Any function passed as a group key will be called once per index value, with the return values being used as the group names. More concretely, consider the example DataFrame from the previous section, which has people's first names as index values. Suppose you wanted to group by the length of the names; you could compute an array of string lengths, but instead you can just pass the `len` function:

```
In [44]: people.groupby(len).sum()
Out[44]:
      a          b          c          d          e
3  0.591569 -0.993608  0.798764 -0.791374  2.119639
5  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Mixing functions with arrays, dicts, or Series is not a problem as everything gets converted to arrays internally:

```
In [45]: key_list = ['one', 'one', 'one', 'two', 'two']
In [46]: people.groupby([len, key_list]).min()
Out[46]:
      a          b          c          d          e
3 one -0.539741 -1.296221  0.274992 -1.021228 -0.577087
     two  0.124121  0.302614  0.523772  0.000940  1.343810
5 one  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 two -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

# Grouping by Index Levels

A final convenience for hierarchically-indexed data sets is the ability to aggregate using one of the levels of an axis index. To do this, pass the level number or name using the `level` keyword:

```
In [47]: columns = pd.MultiIndex.from_arrays([[ 'US', 'US', 'US',
....:                                              [1, 3, 5, 1, 3]],
....:                                              names=['cty', 'ter'])

In [48]: hier_df = pd.DataFrame(np.random.randn(4, 5), columns=columns)

In [49]: hier_df
Out[49]:
      cty          US           JP
      tenor        1            3            5            1            3
0     0.560145 -1.265934  0.119827 -1.063512  0.332883
1    -2.359419 -0.199543 -1.541996 -0.970736 -1.307030
2     0.286350  0.377984 -0.753887  0.331286  1.349742
3     0.069877  0.246674 -0.011862  1.004812  1.327195

In [50]: hier_df.groupby(level='cty', axis=1).count()
Out[50]:
      cty  JP  US
      0   2   3
      1   2   3
      2   2   3
      3   2   3
```

# Data Aggregation

Aggregations refer to any data transformation that produces scalar values from arrays. In the examples above I have used several of them, such as `mean`, `count`, `min` and `sum`. You may wonder what is going on when you invoke `mean()` on a `GroupBy` object. Many common aggregations, such as those found in [Table 10-1](#), have optimized implementations. However, you are not limited to only this set of methods. You can use aggregations of your own devising and additionally call any method that is also defined on the grouped object. For example, as you recall `quantile` computes sample quantiles of a `Series` or a `DataFrame`'s columns [1](#):

```
In [51]: df
Out[51]:
    data1      data2  key1  key2
0 -0.204708  1.393406     a   one
1  0.478943  0.092908     a   two
2 -0.519439  0.281746     b   one
3 -0.555730  0.769023     b   two
4  1.965781  1.246435     a   one

In [52]: grouped = df.groupby('key1')

In [53]: grouped['data1'].quantile(0.9)
Out[53]:
key1
a      1.668413
b     -0.523068
Name: data1, dtype: float64
```

While `quantile` is not explicitly implemented for `GroupBy`, it is a `Series` method and thus available for use. Internally, `GroupBy` efficiently slices up the `Series`, calls `piece.quantile(0.9)` for each piece, then assembles those results together into the result object.

To use your own aggregation functions, pass any function that aggregates an array to the `aggregate` or `agg` method:

```
In [54]: def peak_to_peak(arr):
```

```

....:         return arr.max() - arr.min()

In [55]: grouped.agg(peak_to_peak)
Out[55]:
          data1      data2
key1
a      2.170488  1.300498
b      0.036292  0.487276

```

You may notice that some methods like `describe` also work, even though they are not aggregations, strictly speaking:

```

In [56]: grouped.describe()
Out[56]:
          data1
          count      mean       std      min    25%    50%
key1
a      3.0  0.746672  1.109736 -0.204708  0.137118  0.478943 ...
b      2.0 -0.537585  0.025662 -0.555730 -0.546657 -0.537585 ...
          data2
          max  count      mean       std      min    25%
key1
a      1.965781    3.0  0.910916  0.712217  0.092908  0.669671 ...
b     -0.519439    2.0  0.525384  0.344556  0.281746  0.403565 ...
          75%      max
key1
a      1.319920  1.393406
b      0.647203  0.769023

```

I will explain in more detail what has happened here in the next major section on group-wise operations and transformations.

#### Note

Custom aggregation functions are generally much slower than the optimized functions found in [Table 10-1](#). This is because there is some extra overhead (function calls, data rearrangement) in constructing the intermediate group data chunks.

Table 10-1. Optimized groupby methods

| Function name | Description                          |
|---------------|--------------------------------------|
| count         | Number of non-NA values in the group |

|             |  |
|-------------|--|
| sum         | Sum of non-NA values   |
| mean        | Mean of non-NA values  |
| median      | Arithmetic median of non-NA values                           |
| std, var    | Unbiased (n - 1 denominator) standard deviation and variance |
| min, max    | Minimum and maximum of non-NA values                         |
| prod        | Product of non-NA values                                     |
| first, last | First and last non-NA values                                 |

To illustrate some more advanced aggregation features, I'll use a less trivial dataset, a dataset on restaurant tipping. I obtained it from the R `reshape2` package; it was originally found in Bryant & Smith's 1995 text on business statistics (and found in the book's GitHub repository). After loading it with `read_csv`, I add a tipping percentage column `tip_pct`.

```
In [57]: tips = pd.read_csv('examples/tips.csv')

# Add tip percentage of total bill
In [58]: tips['tip_pct'] = tips['tip'] / tips['total_bill']

In [59]: tips[:6]
Out[59]:
   total_bill    tip     sex smoker  day    time    size  tip_pct
0      16.99  1.01  Female     No  Sun  Dinner      2  0.05944
1      10.34  1.66    Male     No  Sun  Dinner      3  0.16054
2      21.01  3.50    Male     No  Sun  Dinner      3  0.16658
3      23.68  3.31    Male     No  Sun  Dinner      2  0.13978
4      24.59  3.61  Female     No  Sun  Dinner      4  0.14680
5      25.29  4.71    Male     No  Sun  Dinner      4  0.18624
```

# Column-wise and Multiple Function Application

As you've seen above, aggregating a Series or all of the columns of a DataFrame is a matter of using `aggregate` with the desired function or calling a method like `mean` or `std`. However, you may want to aggregate using a different function depending on the column or multiple functions at once. Fortunately, this is possible to do, which I'll illustrate through a number of examples. First, I'll group the `tips` by `sex` and `smoker`:

```
In [60]: grouped = tips.groupby(['sex', 'smoker'])
```

Note that for descriptive statistics like those in [Table 10-1](#), you can pass the name of the function as a string:

```
In [61]: grouped_pct = grouped['tip_pct']
```

```
In [62]: grouped_pct.agg('mean')
Out[62]:
sex      smoker
Female    No        0.156921
          Yes       0.182150
Male     No        0.160669
          Yes       0.152771
Name: tip_pct, dtype: float64
```

If you pass a list of functions or function names instead, you get back a DataFrame with column names taken from the functions:

```
In [63]: grouped_pct.agg(['mean', 'std', 'peak_to_peak'])
Out[63]:
               mean        std  peak_to_peak
sex      smoker
Female    No        0.156921  0.036421   0.195876
          Yes       0.182150  0.071595   0.360233
Male     No        0.160669  0.041849   0.220186
          Yes       0.152771  0.090588   0.674707
```

You don't need to accept the names that GroupBy gives to the columns; notably

`lambda` functions have the name '`<lambda>`' which make them hard to identify (you can see for yourself by looking at a function's `__name__` attribute). As such, if you pass a list of `(name, function)` tuples, the first element of each tuple will be used as the `DataFrame` column names (you can think of a list of 2-tuples as an ordered mapping):

```
In [64]: grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])
Out[64]:
          foo        bar
sex   smoker
Female No      0.156921  0.036421
      Yes      0.182150  0.071595
Male   No      0.160669  0.041849
      Yes      0.152771  0.090588
```

With a `DataFrame`, you have more options as you can specify a list of functions to apply to all of the columns or different functions per column. To start, suppose we wanted to compute the same three statistics for the `tip_pct` and `total_bill` columns:

```
In [65]: functions = ['count', 'mean', 'max']

In [66]: result = grouped[['tip_pct', 'total_bill']].agg(functions)

In [67]: result
Out[67]:
           tip_pct                         total_bill
           count       mean       max       count       mean
sex   smoker
Female No      54  0.156921  0.252672      54  18.105185
      Yes      33  0.182150  0.416667      33  17.977875
Male   No      97  0.160669  0.291990      97  19.791235
      Yes      60  0.152771  0.710345      60  22.284500
```

As you can see, the resulting `DataFrame` has hierarchical columns, the same as you would get aggregating each column separately and using `concat` to glue the results together using the `keys` argument:

```
In [68]: result['tip_pct']
Out[68]:
           count       mean       max
sex   smoker
Female No      54  0.156921  0.252672
      Yes      33  0.182150  0.416667
```

```

Male    No      97  0.160669  0.291990
       Yes     60  0.152771  0.710345

```

As above, a list of tuples with custom names can be passed:

```
In [69]: ftuples = [('Durchschnitt', 'mean'), ('Abweichung', 'std')]
```

```
In [70]: grouped['tip_pct', 'total_bill'].agg(ftuples)
Out[70]:
```

|        |        | tip_pct      | total_bill |              |            |
|--------|--------|--------------|------------|--------------|------------|
|        |        | Durchschnitt | Abweichung | Durchschnitt | Abweichung |
| sex    | smoker |              |            |              |            |
| Female | No     | 0.156921     | 0.001327   | 18.105185    | 53.092422  |
|        | Yes    | 0.182150     | 0.005126   | 17.977879    | 84.451517  |
| Male   | No     | 0.160669     | 0.001751   | 19.791237    | 76.152961  |
|        | Yes    | 0.152771     | 0.008206   | 22.284500    | 98.244673  |

Now, suppose you wanted to apply potentially different functions to one or more of the columns. To do this, pass a dict to `agg` that contains a mapping of column names to any of the function specifications listed so far:

```
In [71]: grouped.agg({'tip' : np.max, 'size' : 'sum'})
```

```
Out[71]:
```

|        |        | tip  | size |
|--------|--------|------|------|
| sex    | smoker |      |      |
| Female | No     | 5.2  | 140  |
|        | Yes    | 6.5  | 74   |
| Male   | No     | 9.0  | 263  |
|        | Yes    | 10.0 | 150  |

```
In [72]: grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],
...:                   'size' : 'sum'})
```

```
Out[72]:
```

|        |        | tip_pct  |          | size     |          |     |
|--------|--------|----------|----------|----------|----------|-----|
|        |        | min      | max      | mean     | std      | sum |
| sex    | smoker |          |          |          |          |     |
| Female | No     | 0.056797 | 0.252672 | 0.156921 | 0.036421 | 140 |
|        | Yes    | 0.056433 | 0.416667 | 0.182150 | 0.071595 | 74  |
| Male   | No     | 0.071804 | 0.291990 | 0.160669 | 0.041849 | 263 |
|        | Yes    | 0.035638 | 0.710345 | 0.152771 | 0.090588 | 150 |

A DataFrame will have hierarchical columns only if multiple functions are applied to at least one column.

# Returning Aggregated Data without Row Indexes

In all of the examples up until now, the aggregated data comes back with an index, potentially hierarchical, composed from the unique group key combinations observed. Since this isn't always desirable, you can disable this behavior in most cases by passing `as_index=False` to `groupby`:

```
In [73]: tips.groupby(['sex', 'smoker'], as_index=False).mean()
Out[73]:
   sex  smoker  total_bill      tip      size  tip_pct
0  Female     No    18.105185  2.773519  2.592593  0.156921
1  Female    Yes    17.977879  2.931515  2.242424  0.182150
2   Male     No    19.791237  3.113402  2.711340  0.160669
3   Male    Yes    22.284500  3.051167  2.500000  0.152771
```

Of course, it's always possible to obtain the result in this format by calling `reset_index` on the result.

# Apply: General split-apply-combine

The most general purpose GroupBy method is `apply`, which is the subject of the rest of this section. As in [Figure 10-1](#), `apply` splits the object being manipulated into pieces, invokes the passed function on each piece, then attempts to concatenate the pieces together.

Returning to the tipping data set above, suppose you wanted to select the top five `tip_pct` values by group. First, write a function that selects the rows with the largest values in a particular column:

```
In [74]: def top(df, n=5, column='tip_pct'):
    ....:     return df.sort_values(by=column)[-n:]

In [75]: top(tips, n=6)
Out[75]:
   total_bill      tip        sex smoker  day   time  size  tip_pct
109       14.31    4.00  Female    Yes  Sat  Dinner    2  0.2795
183       23.17    6.50    Male    Yes  Sun  Dinner    4  0.2805
232       11.61    3.39    Male     No  Sat  Dinner    2  0.2919
67        3.07    1.00  Female    Yes  Sat  Dinner    1  0.3250
178       9.60    4.00  Female    Yes  Sun  Dinner    2  0.4160
172       7.25    5.15    Male    Yes  Sun  Dinner    2  0.7105
```

Now, if we group by `smoker`, say, and call `apply` with this function, we get the following:

```
In [76]: tips.groupby('smoker').apply(top)
Out[76]:
   total_bill      tip        sex smoker  day   time  size
smoker
No      88       24.71    5.85    Male    No  Thur  Lunch    2
       185      20.69    5.00    Male    No  Sun  Dinner    5
       51       10.29    2.60  Female    No  Sun  Dinner    2
       149       7.51    2.00    Male    No  Thur  Lunch    2
       232      11.61    3.39    Male    No  Sat  Dinner    2
Yes     109      14.31    4.00  Female   Yes  Sat  Dinner    2
       183      23.17    6.50    Male   Yes  Sun  Dinner    4
       67       3.07    1.00  Female   Yes  Sat  Dinner    1
       178      9.60    4.00  Female   Yes  Sun  Dinner    2
       172      7.25    5.15    Male   Yes  Sun  Dinner    2
```

What has happened here? The `top` function is called on each piece of the DataFrame, then the results are glued together using `pandas.concat`, labeling the pieces with the group names. The result therefore has a hierarchical index whose inner level contains index values from the original DataFrame.

If you pass a function to `apply` that takes other arguments or keywords, you can pass these after the function:

```
In [77]: tips.groupby(['smoker', 'day']).apply(top, n=1, column='tip')
Out[77]:
          total_bill      tip     sex smoker   day    time
smoker day
No      Fri    94      22.75    3.25 Female    No   Fri  Dinner
                  Sat    212      48.33    9.00 Male     No   Sat  Dinner
                  Sun    156      48.17    5.00 Male     No   Sun  Dinner
                  Thur   142      41.19    5.00 Male     No Thur  Lunch
Yes     Fri    95      40.17    4.73 Male      Yes   Fri  Dinner
                  Sat   170      50.81   10.00 Male      Yes   Sat  Dinner
                  Sun   182      45.35    3.50 Male      Yes   Sun  Dinner
                  Thur   197      43.11    5.00 Female    Yes Thur  Lunch
          tip_pct
smoker day
No      Fri    94    0.142857
                  Sat   212    0.186220
                  Sun   156    0.103799
                  Thur  142    0.121389
Yes     Fri    95    0.117750
                  Sat   170    0.196812
                  Sun   182    0.077178
                  Thur  197    0.115982
```

#### Note

Beyond these basic usage mechanics, getting the most out of `apply` may require some creativity. What occurs inside the function passed is up to you; it only needs to return a pandas object or a scalar value. The rest of this chapter will mainly consist of examples showing you how to solve various problems using `groupby`.

You may recall above I called `describe` on a GroupBy object:

```
In [78]: result = tips.groupby('smoker')['tip_pct'].describe()
```

```
In [79]: result
Out[79]:
   count      mean       std      min      25%      50%
smoker
No      151.0  0.159328  0.039910  0.056797  0.136906  0.155625
Yes     93.0   0.163196  0.085119  0.035638  0.106771  0.153846
          max
smoker
No      0.291990
Yes     0.710345

In [80]: result.unstack('smoker')
Out[80]:
           smoker
count    No      151.000000
           Yes     93.000000
mean     No      0.159328
           Yes     0.163196
std      No      0.039910
           Yes     0.085119
min      No      0.056797
           Yes     0.035638
25%     No      0.136906
           Yes     0.106771
50%     No      0.155625
           Yes     0.153846
75%     No      0.185014
           Yes     0.195059
max     No      0.291990
           Yes     0.710345
dtype: float64
```

Inside GroupBy, when you invoke a method like `describe`, it is actually just a shortcut for:

```
f = lambda x: x.describe()
grouped.apply(f)
```

# Suppressing the group keys

In the examples above, you see that the resulting object has a hierarchical index formed from the group keys along with the indexes of each piece of the original object. This can be disabled by passing `group_keys=False` to `groupby`:

```
In [81]: tips.groupby('smoker', group_keys=False).apply(top)
Out[81]:
   total_bill    tip      sex smoker     day    time  size  tip_
88      24.71  5.85    Male     No Thur  Lunch     2  0.236
185     20.69  5.00    Male     No Sun   Dinner    5  0.241
51      10.29  2.60  Female    No Sun   Dinner    2  0.252
149      7.51  2.00    Male     No Thur  Lunch     2  0.266
232     11.61  3.39    Male     No Sat   Dinner    2  0.291
109     14.31  4.00  Female    Yes Sat   Dinner    2  0.279
183     23.17  6.50    Male    Yes Sun   Dinner    4  0.280
67      3.07  1.00  Female    Yes Sat   Dinner    1  0.325
178      9.60  4.00  Female    Yes Sun   Dinner    2  0.416
172      7.25  5.15    Male    Yes Sun   Dinner    2  0.710
```

# Quantile and Bucket Analysis

As you may recall from [Chapter 8](#), pandas has some tools, in particular `cut` and `qcut`, for slicing data up into buckets with bins of your choosing or by sample quantiles. Combining these functions with `groupby`, it becomes convenient to perform bucket or quantile analysis on a data set. Consider a simple random data set and an equal-length bucket categorization using `cut`:

```
In [82]: frame = pd.DataFrame({'data1': np.random.randn(1000),
....:                           'data2': np.random.randn(1000)})

In [83]: quartiles = pd.cut(frame.data1, 4)

In [84]: quartiles[:10]
Out[84]:
0      (-1.23, 0.489]
1      (-2.956, -1.23]
2      (-1.23, 0.489]
3      (0.489, 2.208]
4      (-1.23, 0.489]
5      (0.489, 2.208]
6      (-1.23, 0.489]
7      (-1.23, 0.489]
8      (0.489, 2.208]
9      (0.489, 2.208]
Name: data1, dtype: category
Categories (4, interval[float64]): [(-2.956, -1.23] < (-1.23, (
```

The `Categorical` object returned by `cut` can be passed directly to `groupby`. So we could compute a set of statistics for the `data2` column like so:

```
In [85]: def get_stats(group):
....:     return {'min': group.min(), 'max': group.max(),
....:            'count': group.count(), 'mean': group.mean()}

In [86]: grouped = frame.data2.groupby(quartiles)

In [87]: grouped.apply(get_stats).unstack()
Out[87]:
              count        max        mean        min
data1
(-2.956, -1.23]    95.0  1.670835 -0.039521 -3.399312
```

```
(-1.23, 0.489]    598.0  3.260383 -0.002051 -2.989741
(0.489, 2.208]    297.0  2.954439  0.081822 -3.745356
(2.208, 3.928]     10.0   1.765640  0.024750 -1.929776
```

These were equal-length buckets; to compute equal-size buckets based on sample quantiles, use `qcut`. I'll pass `labels=False` to just get quantile numbers.

```
# Return quantile numbers
In [88]: grouping = pd.qcut(frame.data1, 10, labels=False)

In [89]: grouped = frame.data2.groupby(grouping)

In [90]: grouped.apply(get_stats).unstack()
Out[90]:
          count      max       mean       min
data1
0        100.0  1.670835 -0.049902 -3.399312
1        100.0  2.628441  0.030989 -1.950098
2        100.0  2.527939 -0.067179 -2.925113
3        100.0  3.260383  0.065713 -2.315555
4        100.0  2.074345 -0.111653 -2.047939
5        100.0  2.184810  0.052130 -2.989741
6        100.0  2.458842 -0.021489 -2.223506
7        100.0  2.954439 -0.026459 -3.056990
8        100.0  2.735527  0.103406 -3.745356
9        100.0  2.377020  0.220122 -2.064111
```

We will take a closer look at pandas's Categorical type in .

# Example: Filling Missing Values with Group-specific Values

When cleaning up missing data, in some cases you will replace data observations using `dropna`, but in others you may want to impute (fill in) the null (NA) values using a fixed value or some value derived from the data. `fillna` is the right tool to use; for example here I fill in NA values with the mean:

```
In [91]: s = pd.Series(np.random.randn(6))
```

```
In [92]: s[::2] = np.nan
```

```
In [93]: s
```

```
Out[93]:
```

```
0      NaN
1    -0.125921
2      NaN
3    -0.884475
4      NaN
5     0.227290
dtype: float64
```

```
In [94]: s.fillna(s.mean())
```

```
Out[94]:
```

```
0    -0.261035
1    -0.125921
2    -0.261035
3    -0.884475
4    -0.261035
5     0.227290
dtype: float64
```

Suppose you need the fill value to vary by group. One way to do this is to group the data and use `apply` with a function that calls `fillna` on each data chunk. Here is some sample data on some US states divided into eastern and western states:

```
In [95]: states = ['Ohio', 'New York', 'Vermont', 'Florida',
....:                 'Oregon', 'Nevada', 'California', 'Idaho']
```

```
In [96]: group_key = ['East'] * 4 + ['West'] * 4
In [97]: data = pd.Series(np.random.randn(8), index=states)
In [98]: data[['Vermont', 'Nevada', 'Idaho']] = np.nan
In [99]: data
Out[99]:
Ohio          0.922264
New York     -2.153545
Vermont        NaN
Florida       -0.375842
Oregon        0.329939
Nevada         NaN
California    1.105913
Idaho          NaN
dtype: float64

In [100]: data.groupby(group_key).mean()
Out[100]:
East   -0.535707
West    0.717926
dtype: float64
```

We can fill the NA values using the group means like so:

```
In [101]: fill_mean = lambda g: g.fillna(g.mean())
In [102]: data.groupby(group_key).apply(fill_mean)
Out[102]:
Ohio          0.922264
New York     -2.153545
Vermont      -0.535707
Florida       -0.375842
Oregon        0.329939
Nevada        0.717926
California    1.105913
Idaho         0.717926
dtype: float64
```

In another case, you might have pre-defined fill values in your code that vary by group. Since the groups have a `name` attribute set internally, we can use that:

```
In [103]: fill_values = {'East': 0.5, 'West': -1}
```

```
In [104]: fill_func = lambda g: g.fillna(fill_values[g.name])

In [105]: data.groupby(group_key).apply(fill_func)
Out[105]:
Ohio          0.922264
New York     -2.153545
Vermont       0.500000
Florida      -0.375842
Oregon        0.329939
Nevada        -1.000000
California    1.105913
Idaho         -1.000000
dtype: float64
```

# Example: Random Sampling and Permutation

Suppose you wanted to draw a random sample (with or without replacement) from a large dataset for Monte Carlo simulation purposes or some other application. There are a number of ways to perform the “draws”; some are much more efficient than others. One way is to select the first  $k$  elements of `np.random.permutation(N)`, where  $N$  is the size of your complete dataset and  $k$  the desired sample size. Here’s a way to construct a deck of English-style playing cards:

```
# Hearts, Spades, Clubs, Diamonds
suits = ['H', 'S', 'C', 'D']
card_val = (list(range(1, 11)) + [10] * 3) * 4
base_names = ['A'] + list(range(2, 11)) + ['J', 'K', 'Q']
cards = []
for suit in ['H', 'S', 'C', 'D']:
    cards.extend(str(num) + suit for num in base_names)

deck = pd.Series(card_val, index=cards)
```

So now we have a Series of length 52 whose index contains card names and values are the ones used in blackjack and other games (to keep things simple, I just let the ace ‘A’ be 1):

```
In [107]: deck[:13]
Out[107]:
AH      1
2H      2
3H      3
4H      4
5H      5
6H      6
7H      7
8H      8
9H      9
10H     10
JH      10
KH      10
QH      10
```

```
dtype: int64
```

Now, based on what I said above, drawing a hand of 5 cards from the deck could be written as:

```
In [108]: def draw(deck, n=5):
.....:     return deck.take(np.random.permutation(len(deck))
```

```
In [109]: draw(deck)
Out[109]:
AD      1
8C      8
5H      5
KC     10
2C      2
dtype: int64
```

Suppose you wanted two random cards from each suit. Because the suit is the last character of each card name, we can group based on this and use `apply`:

```
In [110]: get_suit = lambda card: card[-1] # last letter is su:
In [111]: deck.groupby(get_suit).apply(draw, n=2)
Out[111]:
C  2C      2
   3C      3
D  KD      10
   8D      8
H  KH      10
   3H      3
S  2S      2
   4S      4
dtype: int64
```

```
# alternatively
In [112]: deck.groupby(get_suit, group_keys=False).apply(draw,
Out[112]:
KC      10
JC      10
AD      1
5D      5
5H      5
6H      6
7S      7
KS     10
dtype: int64
```

# Example: Group Weighted Average and Correlation

Under the split-apply-combine paradigm of `groupby`, operations between columns in a DataFrame or two Series, such a group weighted average, are possible. As an example, take this dataset containing group keys, values, and some weights:

```
In [113]: df = pd.DataFrame({'category': ['a', 'a', 'a', 'a',
.....:                               'data': np.random.randn(8),
.....:                               'weights': np.random.rand(8)})
```

```
In [114]: df
Out[114]:
   category      data    weights
0          a  1.561587  0.957515
1          a  1.219984  0.347267
2          a -0.482239  0.581362
3          a  0.315667  0.217091
4          b -0.047852  0.894406
5          b -0.454145  0.918564
6          b -0.556774  0.277825
7          b  0.253321  0.955905
```

The group weighted average by `category` would then be:

```
In [115]: grouped = df.groupby('category')

In [116]: get_wavg = lambda g: np.average(g['data'], weights=g['weights'])

In [117]: grouped.apply(get_wavg)
Out[117]:
category
a      0.811643
b     -0.122262
dtype: float64
```

As another example, consider a financial data set originally obtained from Yahoo! Finance containing end of day prices for a few stocks and the S&P 500 index (the `SPX` symbol):

```
In [118]: close_px = pd.read_csv('examples/stock_px_2.csv', parse_dates=True)
In [119]: close_px.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 to 2011-10-14
Data columns (total 4 columns):
AAPL    2214 non-null float64
MSFT    2214 non-null float64
XOM    2214 non-null float64
SPX     2214 non-null float64
dtypes: float64(4)
memory usage: 86.5 KB

In [120]: close_px[-4:]
Out[120]:
          AAPL    MSFT      XOM      SPX
2011-10-11  400.29  27.00   76.27  1195.54
2011-10-12  402.19  26.96   77.16  1207.25
2011-10-13  408.43  27.18   76.37  1203.66
2011-10-14  422.00  27.27   78.11  1224.58
```

One task of interest might be to compute a DataFrame consisting of the yearly correlations of daily returns (computed from percent changes) with SPX. Here is one way to do it:

```
In [121]: rets = close_px.pct_change().dropna()

In [122]: spx_corr = lambda x: x.corrwith(x['SPX'])

In [123]: by_year = rets.groupby(lambda x: x.year)

In [124]: by_year.apply(spx_corr)
Out[124]:
          AAPL      MSFT      XOM      SPX
2003  0.541124  0.745174  0.661265  1.0
2004  0.374283  0.588531  0.557742  1.0
2005  0.467540  0.562374  0.631010  1.0
2006  0.428267  0.406126  0.518514  1.0
2007  0.508118  0.658770  0.786264  1.0
2008  0.681434  0.804626  0.828303  1.0
2009  0.707103  0.654902  0.797921  1.0
2010  0.710105  0.730118  0.839057  1.0
2011  0.691931  0.800996  0.859975  1.0
```

You could also compute inter-column correlations:

```
# Annual correlation of Apple with Microsoft
In [125]: by_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))
Out[125]:
2003    0.480868
2004    0.259024
2005    0.300093
2006    0.161735
2007    0.417738
2008    0.611901
2009    0.432738
2010    0.571946
2011    0.581987
dtype: float64
```

# Example: Group-wise Linear Regression

In the same theme as the previous example, you can use `groupby` to perform more complex group-wise statistical analysis, as long as the function returns a pandas object or scalar value. For example, I can define the following `regress` function (using the `statsmodels` econometrics library) which executes an ordinary least squares (OLS) regression on each chunk of data:

```
import statsmodels.api as sm
def regress(data, yvar, xvars):
    Y = data[yvar]
    X = data[xvars]
    X['intercept'] = 1.
    result = sm.OLS(Y, X).fit()
    return result.params
```

Now, to run a yearly linear regression of AAPL on SPX returns, execute:

```
In [127]: by_year.apply(regress, 'AAPL', ['SPX'])
Out[127]:
```

|      | SPX      | intercept |
|------|----------|-----------|
| 2003 | 1.195406 | 0.000710  |
| 2004 | 1.363463 | 0.004201  |
| 2005 | 1.766415 | 0.003246  |
| 2006 | 1.645496 | 0.000080  |
| 2007 | 1.198761 | 0.003438  |
| 2008 | 0.968016 | -0.001110 |
| 2009 | 0.879103 | 0.002954  |
| 2010 | 1.052608 | 0.001261  |
| 2011 | 0.806605 | 0.001514  |

# Pivot Tables and Cross-Tabulation

A *pivot table* is a data summarization tool frequently found in spreadsheet programs and other data analysis software. It aggregates a table of data by one or more keys, arranging the data in a rectangle with some of the group keys along the rows and some along the columns. Pivot tables in Python with pandas are made possible using the `groupby` facility described in this chapter combined with reshape operations utilizing hierarchical indexing. DataFrame has a `pivot_table` method, and additionally there is a top-level `pandas.pivot_table` function. In addition to providing a convenience interface to `groupby`, `pivot_table` also can add partial totals, also known as *margins*.

Returning to the tipping data set, suppose you wanted to compute a table of group means (the default `pivot_table` aggregation type) arranged by `sex` and `smoker` on the rows:

```
In [128]: tips.pivot_table(index=['sex', 'smoker'])
Out[128]:
          size      tip  tip_pct  total_bill
sex   smoker
Female  No       2.592593  2.773519  0.156921  18.105185
        Yes      2.242424  2.931515  0.182150  17.977879
Male   No       2.711340  3.113402  0.160669  19.791237
        Yes      2.500000  3.051167  0.152771  22.284500
```

This could have been produced using `groupby` directly. Now, suppose we want to aggregate only `tip_pct` and `size`, and additionally group by `day`. I'll put `smoker` in the table columns and `day` in the rows:

```
In [129]: tips.pivot_table(['tip_pct', 'size'], index=['sex',
.....:                   columns='smoker')
Out[129]:
          size      tip_pct
smoker
sex   day
Female Fri     2.500000  0.165296  0.209129
      Sat     2.307692  0.147993  0.163817
      Sun     3.071429  0.165710  0.237075
```

|      |      |          |          |          |          |
|------|------|----------|----------|----------|----------|
|      | Thur | 2.480000 | 2.428571 | 0.155971 | 0.163073 |
| Male | Fri  | 2.000000 | 2.125000 | 0.138005 | 0.144730 |
|      | Sat  | 2.656250 | 2.629630 | 0.162132 | 0.139067 |
|      | Sun  | 2.883721 | 2.600000 | 0.158291 | 0.173964 |
|      | Thur | 2.500000 | 2.300000 | 0.165706 | 0.164417 |

This table could be augmented to include partial totals by passing `margins=True`. This has the effect of adding `All` row and column labels, with corresponding values being the group statistics for all the data within a single tier. In this below example, the `All` values are means without taking into account smoker vs. non-smoker (the `All` columns) or any of the two levels of grouping on the rows (the `All` row):

```
In [130]: tips.pivot_table(['tip_pct', 'size'], index=['sex'],
....:                               columns='smoker', margins=True)
Out[130]:
```

| smoker | sex  | day      | size     |          |          | tip_pct  |     |
|--------|------|----------|----------|----------|----------|----------|-----|
|        |      |          | No       | Yes      | All      | No       | Yes |
| Female | Fri  | 2.500000 | 2.000000 | 2.111111 | 0.165296 | 0.209129 |     |
|        | Sat  | 2.307692 | 2.200000 | 2.250000 | 0.147993 | 0.163817 |     |
|        | Sun  | 3.071429 | 2.500000 | 2.944444 | 0.165710 | 0.237075 |     |
|        | Thur | 2.480000 | 2.428571 | 2.468750 | 0.155971 | 0.163073 |     |
| Male   | Fri  | 2.000000 | 2.125000 | 2.100000 | 0.138005 | 0.144730 |     |
|        | Sat  | 2.656250 | 2.629630 | 2.644068 | 0.162132 | 0.139067 |     |
|        | Sun  | 2.883721 | 2.600000 | 2.810345 | 0.158291 | 0.173964 |     |
|        | Thur | 2.500000 | 2.300000 | 2.433333 | 0.165706 | 0.164417 |     |
| All    |      | 2.668874 | 2.408602 | 2.569672 | 0.159328 | 0.163196 |     |

To use a different aggregation function, pass it to `aggfunc`. For example, '`count`' or `len` will give you a cross-tabulation (count or frequency) of group sizes:

```
In [131]: tips.pivot_table('tip_pct', index=['sex', 'smoker'],
....:                           aggfunc=len, margins=True)
Out[131]:
```

| day    | sex | smoker |      |      |      |       |
|--------|-----|--------|------|------|------|-------|
|        |     |        | Fri  | Sat  | Sun  | Thur  |
| Female | No  | 2.0    | 13.0 | 14.0 | 25.0 | 54.0  |
|        | Yes | 7.0    | 15.0 | 4.0  | 7.0  | 33.0  |
| Male   | No  | 2.0    | 32.0 | 43.0 | 20.0 | 97.0  |
|        | Yes | 8.0    | 27.0 | 15.0 | 10.0 | 60.0  |
| All    |     | 19.0   | 87.0 | 76.0 | 62.0 | 244.0 |

If some combinations are empty (or otherwise NA), you may wish to pass a `fill_value`:

```
In [132]: tips.pivot_table('size', index=['time', 'sex', 'smoker'],
....:                                         columns='day', aggfunc='sum', fill_
Out[132]:
      day               Fri   Sat   Sun  Thur
time sex   smoker
Dinner Female No       2    30    43     2
       Yes      8    33    10     0
             Male No       4    85   124     0
                   Yes     12   71    39     0
Lunch Female No       3     0     0    60
       Yes      6     0     0    17
             Male No       0     0     0    50
                   Yes     5     0     0    23
```

See [Table 10-2](#) for a summary of `pivot_table` methods.

Table 10-2. `pivot_table` options

| Function name           | Description  |
|-------------------------|--|
| <code>values</code>     | Column name or names to aggregate. By default aggregates all numeric columns   |
| <code>index</code>      | Column names or other group keys to group on the rows of the resulting pivot table   |
| <code>columns</code>    | Column names or other group keys to group on the columns of the resulting pivot table  |
| <code>aggfunc</code>    | Aggregation function or list of functions; ' <code>mean</code> ' by default. Can be any function valid in a <code>groupby</code> context |
| <code>fill_value</code> | Replace missing values in result table   |
| <code>dropna</code>     | If <code>True</code> , do not include columns whose entries are all <code>NA</code> .  |
| <code>margins</code>    | Add row/column subtotals and grand total, <code>False</code> by default  |

# Cross-Tabulations: Crosstab

A cross-tabulation (or *crosstab* for short) is a special case of a pivot table that computes group frequencies. Here is an example taken from the Wikipedia page on cross-tabulation:

```
In [136]: data
Out[136]:
   Sample  Gender      Handedness
0        1  Female  Right-handed
1        2    Male  Left-handed
2        3  Female  Right-handed
3        4    Male  Right-handed
4        5    Male  Left-handed
5        6    Male  Right-handed
6        7  Female  Right-handed
7        8  Female  Left-handed
8        9    Male  Right-handed
9       10  Female  Right-handed
```

As part of some survey analysis, we might want to summarize this data by gender and handedness. You could use `pivot_table` to do this, but the `pandas.crosstab` function can be more convenient:

```
In [137]: pd.crosstab(data.Gender, data.Handedness, margins=True)
Out[137]:
Handedness  Left-handed  Right-handed  All
Gender
Female            1          4          5
Male              2          3          5
All               3          7         10
```

The first two arguments to `crosstab` can each either be an array or Series or a list of arrays. As in the tips data:

```
In [138]: pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)
Out[138]:
smoker      No  Yes  All
time   day
Dinner  Fri     3     9    12
          Sat    45    42   87
```

|       |      |     |    |     |
|-------|------|-----|----|-----|
|       | Sun  | 57  | 19 | 76  |
|       | Thur | 1   | 0  | 1   |
| Lunch | Fri  | 1   | 6  | 7   |
|       | Thur | 44  | 17 | 61  |
| All   |      | 151 | 93 | 244 |

[1](#) Note that `quantile` performs linear interpolation if there is no value at exactly the passed percentile.

# Chapter 11. Time Series

Time series data is an important form of structured data in many different fields, such as finance, economics, ecology, neuroscience, or physics. Anything that is observed or measured at many points in time forms a time series. Many time series are *fixed frequency*, which is to say that data points occur at regular intervals according to some rule, such as every 15 seconds, every 5 minutes, or once per month. Time series can also be *irregular* without a fixed unit or time or offset between units. How you mark and refer to time series data depends on the application and you may have one of the following:

- *Timestamps*, specific instants in time
- Fixed *periods*, such as the month January 2007 or the full year 2010
- *Intervals* of time, indicated by a start and end timestamp. Periods can be thought of as special cases of intervals
- Experiment or elapsed time; each timestamp is a measure of time relative to a particular start time. For example, the diameter of a cookie baking each second since being placed in the oven

In this chapter, I am mainly concerned with time series in the first 3 categories, though many of the techniques can be applied to experimental time series where the index may be an integer or floating point number indicating elapsed time from the start of the experiment. The simplest and most widely used kind of time series are those indexed by timestamp.

pandas provides many built-in time series tools and data algorithms. You can efficiently work with very large time series and easily slice and dice, aggregate, and resample irregular and fixed frequency time series. Some of these tools are especially useful for financial and economics applications, but you could certainly use them to analyze server log data, too.

# Date and Time Data Types and Tools

The Python standard library includes data types for date and time data, as well as calendar-related functionality. The `datetime`, `time`, and `calendar` modules are the main places to start. The `datetime.datetime` type, or simply `datetime`, is widely used:

```
In [10]: from datetime import datetime
```

```
In [11]: now = datetime.now()
```

```
In [12]: now
```

```
Out[12]: datetime.datetime(2017, 5, 14, 22, 40, 21, 105457)
```

```
In [13]: now.year, now.month, now.day
```

```
Out[13]: (2017, 5, 14)
```

`datetime` stores both the date and time down to the microsecond. `timedelta` represents the temporal difference between two `datetime` objects:

```
In [14]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, {
```

```
In [15]: delta
```

```
Out[15]: datetime.timedelta(926, 56700)
```

```
In [16]: delta.days
```

```
Out[16]: 926
```

```
In [17]: delta.seconds
```

```
Out[17]: 56700
```

You can add (or subtract) a `timedelta` or multiple thereof to a `datetime` object to yield a new shifted object:

```
In [18]: from datetime import timedelta
```

```
In [19]: start = datetime(2011, 1, 7)
```

```
In [20]: start + timedelta(12)
```

```
Out[20]: datetime.datetime(2011, 1, 19, 0, 0)
```

```
In [21]: start - 2 * timedelta(12)
Out[21]: datetime.datetime(2010, 12, 14, 0, 0)
```

The data types in the `datetime` module are summarized in [Table 11-1](#). While this chapter is mainly concerned with the data types in pandas and higher level time series manipulation, you may encounter the `datetime`-based types in many other places in Python the wild.

Table 11-1. Types in datetime module

| Type                   | Description  |
|------------------------|--|
| <code>date</code>      | Store calendar date (year, month, day) using the Gregorian calendar.                       |
| <code>time</code>      | Store time of day as hours, minutes, seconds, and microseconds                             |
| <code>datetime</code>  | Stores both date and time  |
| <code>timedelta</code> | Represents the difference between two datetime values (as days, seconds, and microseconds) |
| <code>tzinfo</code>    | Base type for storing time zone information  |

# Converting between string and datetime

`datetime` objects and pandas `Timestamp` objects, which I'll introduce later, can be formatted as strings using `str` or the `strftime` method, passing a format specification:

```
In [22]: stamp = datetime(2011, 1, 3)
```

```
In [23]: str(stamp)
Out[23]: '2011-01-03 00:00:00'
```

```
In [24]: stamp.strftime('%Y-%m-%d')
Out[24]: '2011-01-03'
```

See [Table 2-5](#) for a complete list of the format codes. These same format codes can be used to convert strings to dates using `datetime.strptime`:

```
In [25]: value = '2011-01-03'
```

```
In [26]: datetime.strptime(value, '%Y-%m-%d')
Out[26]: datetime(2011, 1, 3, 0, 0)
```

```
In [27]: datestrs = ['7/6/2011', '8/6/2011']
```

```
In [28]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]
Out[28]: [datetime.datetime(2011, 7, 6, 0, 0), datetime.datetime(2011, 8, 6, 0, 0)]
```

`datetime.strptime` is a good way to parse a date with a known format. However, it can be a bit annoying to have to write a format spec each time, especially for common date formats. In this case, you can use the `parser.parse` method in the third party `dateutil` package (this is installed automatically when installing pandas):

```
In [29]: from dateutil.parser import parse
```

```
In [30]: parse('2011-01-03')
Out[30]: datetime.datetime(2011, 1, 3, 0, 0)
```

`dateutil` is capable of parsing most human-intelligible date representations:

```
In [31]: parse('Jan 31, 1997 10:45 PM')
Out[31]: datetime.datetime(1997, 1, 31, 22, 45)
```

In international locales, day appearing before month is very common, so you can pass `dayfirst=True` to indicate this:

```
In [32]: parse('6/12/2011', dayfirst=True)
Out[32]: datetime.datetime(2011, 12, 6, 0, 0)
```

pandas is generally oriented toward working with arrays of dates, whether used as an axis index or a column in a DataFrame. The `to_datetime` method parses many different kinds of date representations. Standard date formats like ISO 8601 can be parsed very quickly.

```
In [33]: datestrs = ['2011-07-06 12:00:00', '2011-08-06 00:00:00']
```

```
In [34]: pd.to_datetime(datestrs)
Out[34]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00'])
```

It also handles values that should be considered missing (`None`, empty string, etc.):

```
In [35]: idx = pd.to_datetime(datestrs + [None])
```

```
In [36]: idx
Out[36]: DatetimeIndex(['2011-07-06 12:00:00', '2011-08-06 00:00:00', NaT])
```

```
In [37]: idx[2]
Out[37]: NaT
```

```
In [38]: pd.isnull(idx)
Out[38]: array([False, False, True], dtype=bool)
```

`NaT` (Not a Time) is pandas's null value for timestamp data.

### Caution

`dateutil.parser` is a useful, but not perfect tool. Notably, it will recognize some strings as dates that you might prefer that it didn't, like '`42`' will be parsed as the year `2042` with today's calendar date.

`datetime` objects also have a number of locale-specific formatting options for

systems in other countries or languages. For example, the abbreviated month names will be different on German or French systems compared with English systems.

See [Table 11-2](#) for a listing.

Table 11-2. Locale-specific date formatting

| Type | Description   |
|------|---|
| %a   | Abbreviated weekday name  |
| %A   | Full weekday name   |
| %b   | Abbreviated month name  |
| %B   | Full month name   |
| %c   | Full date and time, for example ‘Tue 01 May 2012 04:20:57 PM’                 |
| %p   | Locale equivalent of AM or PM   |
| %x   | Locale-appropriate formatted date; e.g. in US May 1, 2012 yields ‘05/01/2012’ |
| %X   | Locale-appropriate time, e.g. ’04:24:12 PM’                                   |

# Time Series Basics

A basic kind of time series object in pandas is a Series indexed by timestamps, which is often represented external to pandas as Python strings or `datetime` objects:

```
In [39]: from datetime import datetime  
  
In [40]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5),  
....:           datetime(2011, 1, 8), datetime(2011, 1, 10),  
  
In [41]: ts = pd.Series(np.random.randn(6), index=dates)  
  
In [42]: ts  
Out[42]:  
2011-01-02    -0.204708  
2011-01-05     0.478943  
2011-01-07    -0.519439  
2011-01-08    -0.555730  
2011-01-10     1.965781  
2011-01-12     1.393406  
dtype: float64
```

Under the hood, these `datetime` objects have been put in a `DatetimeIndex`:

```
In [43]: ts.index  
Out[43]:  
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-  
'2011-01-10', '2011-01-12'],  
              dtype='datetime64[ns]', freq=None)
```

Like other Series, arithmetic operations between differently-indexed time series automatically align on the dates:

```
In [44]: ts + ts[::2]  
Out[44]:  
2011-01-02    -0.409415  
2011-01-05      NaN  
2011-01-07   -1.038877  
2011-01-08      NaN  
2011-01-10    3.931561  
2011-01-12      NaN
```

```
dtype: float64
```

pandas stores timestamps using NumPy's `datetime64` data type at the nanosecond resolution:

```
In [45]: ts.index.dtype
Out[45]: dtype('<M8[ns]')
```

Scalar values from a `DatetimeIndex` are pandas `Timestamp` objects

```
In [46]: stamp = ts.index[0]
```

```
In [47]: stamp
Out[47]: Timestamp('2011-01-02 00:00:00')
```

A `Timestamp` can be substituted anywhere you would use a `datetime` object. Additionally, it can store frequency information (if any) and understands how to do time zone conversions and other kinds of manipulations. More on both of these things later.

# Indexing, Selection, Subsetting

Time series behaves like any other `pandas.Series` when indexing and selecting data based on label:

```
In [48]: stamp = ts.index[2]
```

```
In [49]: ts[stamp]  
Out[49]: -0.51943871505673811
```

As a convenience, you can also pass a string that is interpretable as a date:

```
In [50]: ts['1/10/2011']  
Out[50]: 1.9657805725027142
```

```
In [51]: ts['20110110']  
Out[51]: 1.9657805725027142
```

For longer time series, a year or only a year and month can be passed to easily select slices of data:

```
In [52]: longer_ts = pd.Series(np.random.randn(1000),  
.....  
.....  
index=pd.date_range('1/1/2000',  
  
In [53]: longer_ts  
Out[53]:  
2000-01-01    0.092908  
2000-01-02    0.281746  
2000-01-03    0.769023  
2000-01-04    1.246435  
2000-01-05    1.007189  
2000-01-06   -1.296221  
2000-01-07    0.274992  
2000-01-08    0.228913  
2000-01-09    1.352917  
2000-01-10    0.886429  
.....  
2002-09-17   -0.139298  
2002-09-18   -1.159926  
2002-09-19    0.618965  
2002-09-20    1.373890  
2002-09-21   -0.983505
```

```
2002-09-22      0.930944
2002-09-23     -0.811676
2002-09-24     -1.830156
2002-09-25     -0.138730
2002-09-26      0.334088
Freq: D, Length: 1000, dtype: float64
```

```
In [54]: longer_ts['2001']
Out[54]:
2001-01-01      1.599534
2001-01-02      0.474071
2001-01-03      0.151326
2001-01-04     -0.542173
2001-01-05     -0.475496
2001-01-06      0.106403
2001-01-07     -1.308228
2001-01-08      2.173185
2001-01-09      0.564561
2001-01-10     -0.190481
...
2001-12-22      0.000369
2001-12-23      0.900885
2001-12-24     -0.454869
2001-12-25     -0.864547
2001-12-26      1.129120
2001-12-27      0.057874
2001-12-28     -0.433739
2001-12-29      0.092698
2001-12-30     -1.397820
2001-12-31      1.457823
Freq: D, Length: 365, dtype: float64
```

```
In [55]: longer_ts['2001-05']
Out[55]:
2001-05-01     -0.622547
2001-05-02      0.936289
2001-05-03      0.750018
2001-05-04     -0.056715
2001-05-05      2.300675
2001-05-06      0.569497
2001-05-07      1.489410
2001-05-08      1.264250
2001-05-09     -0.761837
2001-05-10     -0.331617
...
2001-05-22      0.503699
2001-05-23     -1.387874
```

```
2001-05-24      0.204851
2001-05-25      0.603705
2001-05-26      0.545680
2001-05-27      0.235477
2001-05-28      0.111835
2001-05-29     -1.251504
2001-05-30     -2.949343
2001-05-31      0.634634
Freq: D, Length: 31, dtype: float64
```

Slicing with `datetime` objects works also:

```
In [56]: ts[datetime(2011, 1, 7):]
Out[56]:
2011-01-07    -0.519439
2011-01-08    -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64
```

Because most time series data is ordered chronologically, you can slice with timestamps not contained in a time series to perform a range query:

```
In [57]: ts
Out[57]:
2011-01-02    -0.204708
2011-01-05    0.478943
2011-01-07    -0.519439
2011-01-08    -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64
```

```
In [58]: ts['1/6/2011':'1/11/2011']
Out[58]:
2011-01-07    -0.519439
2011-01-08    -0.555730
2011-01-10    1.965781
dtype: float64
```

As before you can pass either a string date, `datetime`, or `Timestamp`. Remember that slicing in this manner produces views on the source time series like slicing NumPy arrays. There is an equivalent instance method `truncate` which slices a `TimeSeries` between two dates:

```
In [59]: ts.truncate(after='1/9/2011')
Out[59]:
2011-01-02    -0.204708
2011-01-05     0.478943
2011-01-07    -0.519439
2011-01-08    -0.555730
dtype: float64
```

All of the above holds true for DataFrame as well, indexing on its rows:

```
In [60]: dates = pd.date_range('1/1/2000', periods=100, freq='W-WED')
In [61]: long_df = pd.DataFrame(np.random.randn(100, 4),
....:                           index=dates,
....:                           columns=['Colorado', 'Texas',
....:                                     'New York', 'Ohio'])
In [62]: long_df.loc['5-2001']
Out[62]:
          Colorado      Texas   New York      Ohio
2001-05-02 -0.006045  0.490094 -0.277186 -0.707213
2001-05-09 -0.560107  2.735527  0.927335  1.513906
2001-05-16  0.538600  1.273768  0.667876 -0.969206
2001-05-23  1.676091 -0.817649  0.050188  1.951312
2001-05-30  3.260383  0.963301  1.201206 -1.852001
```

# Time Series with Duplicate Indices

In some applications, there may be multiple data observations falling on a particular timestamp. Here is an example:

```
In [63]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000', '1/3/2000'])  
....:  
  
In [64]: dup_ts = pd.Series(np.arange(5), index=dates)  
  
In [65]: dup_ts  
Out[65]:  
2000-01-01    0  
2000-01-02    1  
2000-01-02    2  
2000-01-02    3  
2000-01-03    4  
dtype: int64
```

We can tell that the index is not unique by checking its `is_unique` property:

```
In [66]: dup_ts.index.is_unique  
Out[66]: False
```

Indexing into this time series will now either produce scalar values or slices depending on whether a timestamp is duplicated:

```
In [67]: dup_ts['1/3/2000'] # not duplicated  
Out[67]: 4  
  
In [68]: dup_ts['1/2/2000'] # duplicated  
Out[68]:  
2000-01-02    1  
2000-01-02    2  
2000-01-02    3  
dtype: int64
```

Suppose you wanted to aggregate the data having non-unique timestamps. One way to do this is to use `groupby` and pass `level=0` (the only level of indexing!):

```
In [69]: grouped = dup_ts.groupby(level=0)
```

```
In [70]: grouped.mean()
```

```
Out[70]:
```

```
2000-01-01    0
```

```
2000-01-02    2
```

```
2000-01-03    4
```

```
dtype: int64
```

```
In [71]: grouped.count()
```

```
Out[71]:
```

```
2000-01-01    1
```

```
2000-01-02    3
```

```
2000-01-03    1
```

```
dtype: int64
```

# Date Ranges, Frequencies, and Shifting

Generic time series in pandas are assumed to be irregular; that is, they have no fixed frequency. For many applications this is sufficient. However, it's often desirable to work relative to a fixed frequency, such as daily, monthly, or every 15 minutes, even if that means introducing missing values into a time series. Fortunately pandas has a full suite of standard time series frequencies and tools for resampling, inferring frequencies, and generating fixed frequency date ranges. For example, in the example time series, converting it to be fixed daily frequency can be accomplished by calling `resample`:

```
In [72]: ts
Out[72]:
2011-01-02    -0.204708
2011-01-05     0.478943
2011-01-07    -0.519439
2011-01-08    -0.555730
2011-01-10     1.965781
2011-01-12     1.393406
dtype: float64

In [73]: resampler = ts.resample('D')
```

Conversion between frequencies or *resampling* is a big enough topic to have its own section later. Here I'll show you how to use the base frequencies and multiples thereof.

# Generating Date Ranges

While I used it previously without explanation, `pandas.date_range` is responsible for generating a `DatetimeIndex` with an indicated length according to a particular frequency:

```
In [74]: index = pd.date_range('2012-04-01', '2012-06-01')

In [75]: index
Out[75]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-
              '2012-04-05', '2012-04-06', '2012-04-07', '2012-
              '2012-04-09', '2012-04-10', '2012-04-11', '2012-
              '2012-04-13', '2012-04-14', '2012-04-15', '2012-
              '2012-04-17', '2012-04-18', '2012-04-19', '2012-
              '2012-04-21', '2012-04-22', '2012-04-23', '2012-
              '2012-04-25', '2012-04-26', '2012-04-27', '2012-
              '2012-04-29', '2012-04-30', '2012-05-01', '2012-
              '2012-05-03', '2012-05-04', '2012-05-05', '2012-
              '2012-05-07', '2012-05-08', '2012-05-09', '2012-
              '2012-05-11', '2012-05-12', '2012-05-13', '2012-
              '2012-05-15', '2012-05-16', '2012-05-17', '2012-
              '2012-05-19', '2012-05-20', '2012-05-21', '2012-
              '2012-05-23', '2012-05-24', '2012-05-25', '2012-
              '2012-05-27', '2012-05-28', '2012-05-29', '2012-
              '2012-05-31', '2012-06-01'],
               dtype='datetime64[ns]', freq='D')
```

By default, `date_range` generates daily timestamps. If you pass only a start or end date, you must pass a number of periods to generate:

```
In [76]: pd.date_range(start='2012-04-01', periods=20)
Out[76]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-03', '2012-
              '2012-04-05', '2012-04-06', '2012-04-07', '2012-
              '2012-04-09', '2012-04-10', '2012-04-11', '2012-
              '2012-04-13', '2012-04-14', '2012-04-15', '2012-
              '2012-04-17', '2012-04-18', '2012-04-19', '2012-
              '2012-04-21', '2012-04-22', '2012-04-23', '2012-
              '2012-04-25', '2012-04-26', '2012-04-27', '2012-
              '2012-04-29', '2012-04-30', '2012-05-01', '2012-
              '2012-05-03', '2012-05-04', '2012-05-05', '2012-
              '2012-05-07', '2012-05-08', '2012-05-09', '2012-
              '2012-05-11', '2012-05-12', '2012-05-13', '2012-
              '2012-05-15', '2012-05-16', '2012-05-17', '2012-
              '2012-05-19', '2012-05-20', '2012-05-21', '2012-
              '2012-05-23', '2012-05-24', '2012-05-25', '2012-
              '2012-05-27', '2012-05-28', '2012-05-29', '2012-
              '2012-05-31', '2012-06-01'],
               dtype='datetime64[ns]', freq='D')
```

```
In [77]: pd.date_range(end='2012-06-01', periods=20)
Out[77]:
DatetimeIndex(['2012-05-13', '2012-05-14', '2012-05-15', '2012-
```

```
'2012-05-17', '2012-05-18', '2012-05-19', '2012-
'2012-05-21', '2012-05-22', '2012-05-23', '2012-
'2012-05-25', '2012-05-26', '2012-05-27', '2012-
'2012-05-29', '2012-05-30', '2012-05-31', '2012-
dtype='datetime64[ns]', freq='D')
```

The start and end dates define strict boundaries for the generated date index. For example, if you wanted a date index containing the last business day of each month, you would pass the '`BM`' frequency (business end of month) and only dates falling on or inside the date interval will be included:

```
In [78]: pd.date_range('2000-01-01', '2000-12-01', freq='BM')
Out[78]:
DatetimeIndex(['2000-01-31', '2000-02-29', '2000-03-31', '2000-
              '2000-05-31', '2000-06-30', '2000-07-31', '2000-
              '2000-09-29', '2000-10-31', '2000-11-30'],
              dtype='datetime64[ns]', freq='BM')
```

`date_range` by default preserves the time (if any) of the start or end timestamp:

```
In [79]: pd.date_range('2012-05-02 12:56:31', periods=5)
Out[79]:
DatetimeIndex(['2012-05-02 12:56:31', '2012-05-03 12:56:31',
               '2012-05-04 12:56:31', '2012-05-05 12:56:31',
               '2012-05-06 12:56:31'],
               dtype='datetime64[ns]', freq='D')
```

Sometimes you will have start or end dates with time information but want to generate a set of timestamps *normalized* to midnight as a convention. To do this, there is a `normalize` option:

```
In [80]: pd.date_range('2012-05-02 12:56:31', periods=5, normalize=True)
Out[80]:
DatetimeIndex(['2012-05-02', '2012-05-03', '2012-05-04', '2012-
               '2012-05-06'],
               dtype='datetime64[ns]', freq='D')
```

# Frequencies and Date Offsets

Frequencies in pandas are composed of a *base frequency* and a multiplier. Base frequencies are typically referred to by a string alias, like '`M`' for monthly or '`H`' for hourly. For each base frequency, there is an object defined generally referred to as a *date offset*. For example, hourly frequency can be represented with the `Hour` class:

```
In [81]: from pandas.tseries.offsets import Hour, Minute
```

```
In [82]: hour = Hour()
```

```
In [83]: hour
Out[83]: <Hour>
```

You can define a multiple of an offset by passing an integer:

```
In [84]: four_hours = Hour(4)
```

```
In [85]: four_hours
Out[85]: <4 * Hours>
```

In most applications, you would never need to explicitly create one of these objects, instead using a string alias like '`H`' or '`4H`'. Putting an integer before the base frequency creates a multiple:

```
In [86]: pd.date_range('2000-01-01', '2000-01-03 23:59', freq='H')
Out[86]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 04:00:00',
               '2000-01-01 08:00:00', '2000-01-01 12:00:00',
               '2000-01-01 16:00:00', '2000-01-01 20:00:00',
               '2000-01-02 00:00:00', '2000-01-02 04:00:00',
               '2000-01-02 08:00:00', '2000-01-02 12:00:00',
               '2000-01-02 16:00:00', '2000-01-02 20:00:00',
               '2000-01-03 00:00:00', '2000-01-03 04:00:00',
               '2000-01-03 08:00:00', '2000-01-03 12:00:00',
               '2000-01-03 16:00:00', '2000-01-03 20:00:00'],
              dtype='datetime64[ns]', freq='4H')
```

Many offsets can be combined together by addition:

```
In [87]: Hour(2) + Minute(30)
Out[87]: <150 * Minutes>
```

Similarly, you can pass frequency strings like '2h30min' which will effectively be parsed to the same expression:

```
In [88]: pd.date_range('2000-01-01', periods=10, freq='1h30min')
Out[88]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:30:00',
                '2000-01-01 03:00:00', '2000-01-01 04:30:00',
                '2000-01-01 06:00:00', '2000-01-01 07:30:00',
                '2000-01-01 09:00:00', '2000-01-01 10:30:00',
                '2000-01-01 12:00:00', '2000-01-01 13:30:00'],
               dtype='datetime64[ns]', freq='90T')
```

Some frequencies describe points in time that are not evenly spaced. For example, 'M' (calendar month end) and 'BM' (last business/weekday of month) depend on the number of days in a month and, in the latter case, whether the month ends on a weekend or not. We refer to these as *anchored* offsets.

See [Table 11-3](#) for a listing of frequency codes and date offset classes available in pandas.

#### Note

Users can define their own custom frequency classes to provide date logic not available in pandas, though the full details of that are outside the scope of this book.

Table 11-3. Base Time Series Frequencies (Not Comprehensive)

| Alias    | Offset Type | Description                           |
|----------|-------------|---------------------------------------|
| D        | Day         | Calendar daily                        |
| B        | BusinessDay | Business daily                        |
| H        | Hour        | Hourly                                |
| T or min | Minute      | Minutely                              |
| S        | Second      | Secondly                              |
| L or ms  | Milli       | Millisecond (1/1000th of 1 second)    |
| U        | Micro       | Microsecond (1/1000000th of 1 second) |

|   |   |  |
|---|---|--|
| M   | MonthEnd  | Last calendar day of month   |
| BM  | BusinessMonthEnd  | Last business day (weekday) of month   |
| MS  | MonthBegin  | First calendar day of month  |
| BMS   | BusinessMonthBegin  | First weekday of month   |
| W-MON,<br>W-TUE,<br>...<br>WOM-<br>1MON,<br>WOM-<br>2MON,<br>...  | Week  | Weekly on given day of week: MON, TUE, WED, THU, FRI, SAT, or SUN.   |
| WOM-<br>3MON,<br>WOM-<br>4MON,<br>...   | WeekOfMonth   | Generate weekly dates in the first, second, third, or fourth week of the month. For example, WOM-3FRI for the 3rd Friday of each month.  |
| Q-JAN,<br>Q-FEB,<br>...<br>BQ-JAN,<br>BQ-FEB,<br>...<br>QS-JAN,<br>QS-FEB,<br>...<br>BQS-<br>JAN,<br>BQS-<br>FEB,<br>...<br>A-JAN,<br>A-FEB,<br>...<br>BA-JAN,<br>BA-FEB,<br>...<br>AS-JAN,<br>AS-FEB,<br>...<br>BAS- | QuarterEnd<br>BusinessQuarterEnd<br>QuarterBegin<br>BusinessQuarterBegin<br>YearEnd<br>BusinessYearEnd<br>YearBegin | Quarterly dates anchored on last calendar day of each month, for year ending in indicated month: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC.<br>Quarterly dates anchored on last weekday day of each month, for year ending in indicated month<br>Quarterly dates anchored on first calendar day of each month, for year ending in indicated month<br>Quarterly dates anchored on first weekday day of each month, for year ending in indicated month<br>Annual dates anchored on last calendar day of given month: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC.<br>Annual dates anchored on last weekday of given month<br>Annual dates anchored on first day of given month |

|      |                   |   |
|------|-------------------|---|
| JAN, | BusinessYearBegin | Annual dates anchored on first weekday of given month |
| BAS- |                   |   |
| FEB, |                   |   |
| ...  |                   |   |

## Week of month dates

One useful frequency class is “week of month”, starting with `WOM`. This enables you to get dates like the third Friday of each month:

```
In [89]: rng = pd.date_range('2012-01-01', '2012-09-01', freq='WOM-3FRI')

In [90]: list(rng)
Out[90]:
[Timestamp('2012-01-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-02-17 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-03-16 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-04-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-05-18 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-06-15 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-07-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-08-17 00:00:00', freq='WOM-3FRI')]
```

# Shifting (Leading and Lagging) Data

“Shifting” refers to moving data backward and forward through time. Both Series and DataFrame have a `shift` method for doing naive shifts forward or backward, leaving the index unmodified:

```
In [91]: ts = pd.Series(np.random.randn(4),
....:                     index=pd.date_range('1/1/2000', periods=4))
Out[91]:
2000-01-31    -0.066748
2000-02-29     0.838639
2000-03-31    -0.117388
2000-04-30    -0.517795
Freq: M, dtype: float64

In [92]: ts.shift(2)
Out[92]:
2000-01-31        NaN
2000-02-29        NaN
2000-03-31    -0.066748
2000-04-30     0.838639
Freq: M, dtype: float64

In [93]: ts.shift(-2)
Out[93]:
2000-01-31    -0.117388
2000-02-29    -0.517795
2000-03-31        NaN
2000-04-30        NaN
Freq: M, dtype: float64

In [94]: ts.shift(-2)
```

A common use of `shift` is computing percent changes in a time series or multiple time series as DataFrame columns. This is expressed as

```
ts / ts.shift(1) - 1
```

Because naive shifts leave the index unmodified, some data is discarded. Thus if the frequency is known, it can be passed to `shift` to advance the timestamps instead of simply the data:

```
In [95]: ts.shift(2, freq='M')
Out[95]:
2000-03-31    -0.066748
2000-04-30     0.838639
2000-05-31    -0.117388
2000-06-30    -0.517795
Freq: M, dtype: float64
```

Other frequencies can be passed, too, giving you some flexibility in how to lead and lag the data:

```
In [96]: ts.shift(3, freq='D')
Out[96]:
2000-02-03    -0.066748
2000-03-03     0.838639
2000-04-03    -0.117388
2000-05-03    -0.517795
dtype: float64
```

```
In [97]: ts.shift(1, freq='3D')
Out[97]:
2000-02-03    -0.066748
2000-03-03     0.838639
2000-04-03    -0.117388
2000-05-03    -0.517795
dtype: float64
```

```
In [98]: ts.shift(1, freq='90T')
Out[98]:
2000-01-31 01:30:00    -0.066748
2000-02-29 01:30:00     0.838639
2000-03-31 01:30:00    -0.117388
2000-04-30 01:30:00    -0.517795
Freq: M, dtype: float64
```

## Shifting dates with offsets

The pandas date offsets can also be used with `datetime` or `Timestamp` objects:

```
In [99]: from pandas.tseries.offsets import Day, MonthEnd
In [100]: now = datetime(2011, 11, 17)
```

```
In [101]: now + 3 * Day()
Out[101]: Timestamp('2011-11-20 00:00:00')
```

If you add an anchored offset like `MonthEnd`, the first increment will `roll` forward a date to the next date according to the frequency rule:

```
In [102]: now + MonthEnd()
Out[102]: Timestamp('2011-11-30 00:00:00')
```

```
In [103]: now + MonthEnd(2)
Out[103]: Timestamp('2011-12-31 00:00:00')
```

Anchored offsets can explicitly “roll” dates forward or backward using their `rollforward` and `rollback` methods, respectively:

```
In [104]: offset = MonthEnd()
```

```
In [105]: offset.rollforward(now)
Out[105]: Timestamp('2011-11-30 00:00:00')
```

```
In [106]: offset.rollback(now)
Out[106]: Timestamp('2011-10-31 00:00:00')
```

A creative use of date offsets is to use these methods with `groupby`:

```
In [107]: ts = pd.Series(np.random.randn(20),
.....:                  index=pd.date_range('1/15/2000', period='M'))
In [108]: ts.groupby(offset.rollforward).mean()
Out[108]:
2000-01-31    -0.005833
2000-02-29     0.015894
2000-03-31     0.150209
dtype: float64
```

Of course, an easier and faster way to do this is using `resample` (much more on this later):

```
In [109]: ts.resample('M').mean()
Out[109]:
2000-01-31    -0.005833
2000-02-29     0.015894
2000-03-31     0.150209
Freq: M, dtype: float64
```

# Time Zone Handling

Working with time zones is generally considered one of the most unpleasant parts of time series manipulation. As such, many time series users choose to work with time series in *coordinated universal time* or *UTC*, which is the successor to Greenwich Mean Time and is the current international standard. Time zones are expressed as offsets from UTC; for example, New York is four hours behind UTC during daylight savings time and 5 hours the rest of the year.

In Python, time zone information comes from the 3rd party `pytz` library, which exposes the *Olson database*, a compilation of world time zone information. This is especially important for historical data because the DST transition dates (and even UTC offsets) have been changed numerous times depending on the whims of local governments. In the United States, the DST transition times have been changed many times since 1900!

For detailed information about `pytz` library, you'll need to look at that library's documentation. As far as this book is concerned, pandas wraps `pytz`'s functionality so you can ignore its API outside of the time zone names. Time zone names can be found interactively and in the docs:

```
In [110]: import pytz  
  
In [111]: pytz.common_timezones[-5:]  
Out[111]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacif:
```

To get a time zone object from `pytz`, use `pytz.timezone`:

```
In [112]: tz = pytz.timezone('America/New_York')  
  
In [113]: tz  
Out[113]: <DstTzInfo 'America/New_York' LMT-1 day, 19:04:00 STI
```

Methods in pandas will accept either time zone names or these objects, but I recommend using the names.

# Time Zone Localization and Conversion

By default, time series in pandas are *time zone naive*. Consider the following time series:

```
In [114]: rng = pd.date_range('3/9/2012 9:30', periods=6, freq='D')  
In [115]: ts = pd.Series(np.random.randn(len(rng)), index=rng)
```

The index's tz field is None:

```
In [116]: print(ts.index.tz)  
None
```

Date ranges can be generated with a time zone set:

```
In [117]: pd.date_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC')  
Out[117]:  
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',  
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',  
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',  
               '2012-03-15 09:30:00+00:00', '2012-03-16 09:30:00+00:00',  
               '2012-03-17 09:30:00+00:00', '2012-03-18 09:30:00+00:00'],  
              dtype='datetime64[ns, UTC]', freq='D')
```

Conversion from naive to *localized* is handled by the tz\_localize method:

```
In [118]: ts_utc = ts.tz_localize('UTC')  
  
In [119]: ts_utc  
Out[119]:  
2012-03-09 09:30:00+00:00    -0.202469  
2012-03-10 09:30:00+00:00     0.050718  
2012-03-11 09:30:00+00:00     0.639869  
2012-03-12 09:30:00+00:00     0.597594  
2012-03-13 09:30:00+00:00    -0.797246  
2012-03-14 09:30:00+00:00     0.472879  
Freq: D, dtype: float64  
  
In [120]: ts_utc.index  
Out[120]:  
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-10 09:30:00+00:00',  
               '2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',  
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',  
               '2012-03-15 09:30:00+00:00', '2012-03-16 09:30:00+00:00',  
               '2012-03-17 09:30:00+00:00', '2012-03-18 09:30:00+00:00'],  
              dtype='datetime64[ns, UTC]', freq='D')
```

```
'2012-03-11 09:30:00+00:00', '2012-03-12 09:30:00+00:00',
'2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00+00:00',
dtype='datetime64[ns, UTC]', freq='D')
```

Once a time series has been localized to a particular time zone, it can be converted to another time zone using `tz_convert`:

```
In [121]: ts_utc.tz_convert('America/New_York')
Out[121]:
2012-03-09 04:30:00-05:00    -0.202469
2012-03-10 04:30:00-05:00     0.050718
2012-03-11 05:30:00-04:00     0.639869
2012-03-12 05:30:00-04:00     0.597594
2012-03-13 05:30:00-04:00    -0.797246
2012-03-14 05:30:00-04:00     0.472879
Freq: D, dtype: float64
```

In the case of the above time series, which straddles a DST transition in the America/New\_York time zone, we could localize to EST and convert to, say, UTC or Berlin time:

```
In [122]: ts_eastern = ts.tz_localize('America/New_York')

In [123]: ts_eastern.tz_convert('UTC')
Out[123]:
2012-03-09 14:30:00+00:00    -0.202469
2012-03-10 14:30:00+00:00     0.050718
2012-03-11 13:30:00+00:00     0.639869
2012-03-12 13:30:00+00:00     0.597594
2012-03-13 13:30:00+00:00    -0.797246
2012-03-14 13:30:00+00:00     0.472879
Freq: D, dtype: float64

In [124]: ts_eastern.tz_convert('Europe/Berlin')
Out[124]:
2012-03-09 15:30:00+01:00    -0.202469
2012-03-10 15:30:00+01:00     0.050718
2012-03-11 14:30:00+01:00     0.639869
2012-03-12 14:30:00+01:00     0.597594
2012-03-13 14:30:00+01:00    -0.797246
2012-03-14 14:30:00+01:00     0.472879
Freq: D, dtype: float64
```

`tz_localize` and `tz_convert` are also instance methods on `DatetimeIndex`:

```
In [125]: ts.index.tz_localize('Asia/Shanghai')
Out[125]:
DatetimeIndex(['2012-03-09 09:30:00+08:00', '2012-03-10 09:30:00',
               '2012-03-11 09:30:00+08:00', '2012-03-12 09:30:00',
               '2012-03-13 09:30:00+08:00', '2012-03-14 09:30:00'],
              dtype='datetime64[ns, Asia/Shanghai]', freq='D')
```

#### Caution

Localizing naive timestamps also checks for ambiguous or non-existent times around daylight savings time transitions.

# Operations with Time Zone-aware Timestamp Objects

Similar to time series and date ranges, individual Timestamp objects similarly can be localized from naive to time zone-aware and converted from one time zone to another:

```
In [126]: stamp = pd.Timestamp('2011-03-12 04:00')

In [127]: stamp_utc = stamp.tz_localize('utc')

In [128]: stamp_utc.tz_convert('America/New_York')
Out[128]: Timestamp('2011-03-11 23:00:00-0500', tz='America/New_York')
```

You can also pass a time zone when creating the Timestamp:

```
In [129]: stamp_moscow = pd.Timestamp('2011-03-12 04:00', tz='Europe/Moscow')

In [130]: stamp_moscow
Out[130]: Timestamp('2011-03-12 04:00:00+0300', tz='Europe/Moscow')
```

Time zone-aware Timestamp objects internally store a UTC timestamp value as nanoseconds since the UNIX epoch (January 1, 1970); this UTC value is invariant between time zone conversions:

```
In [131]: stamp_utc.value
Out[131]: 12999024000000000000

In [132]: stamp_utc.tz_convert('America/New_York').value
Out[132]: 12999024000000000000
```

When performing time arithmetic using pandas's `DateOffset` objects, daylight savings time transitions are respected where possible:

```
# 30 minutes before DST transition
In [133]: from pandas.tseries.offsets import Hour

In [134]: stamp = pd.Timestamp('2012-03-12 01:30', tz='US/Eastern')

In [135]: stamp
```

```
Out[135]: Timestamp('2012-03-12 01:30:00-0400', tz='US/Eastern'

In [136]: stamp + Hour()
Out[136]: Timestamp('2012-03-12 02:30:00-0400', tz='US/Eastern'

# 90 minutes before DST transition
In [137]: stamp = pd.Timestamp('2012-11-04 00:30', tz='US/Eastern')

In [138]: stamp
Out[138]: Timestamp('2012-11-04 00:30:00-0400', tz='US/Eastern'

In [139]: stamp + 2 * Hour()
Out[139]: Timestamp('2012-11-04 01:30:00-0500', tz='US/Eastern')
```

# Operations between Different Time Zones

If two time series with different time zones are combined, the result will be UTC. Since the timestamps are stored under the hood in UTC, this is a straightforward operation and requires no conversion to happen:

```
In [140]: rng = pd.date_range('3/7/2012 9:30', periods=10, freq='B')
In [141]: ts = pd.Series(np.random.randn(len(rng)), index=rng)
In [142]: ts
Out[142]:
2012-03-07 09:30:00    0.522356
2012-03-08 09:30:00   -0.546348
2012-03-09 09:30:00   -0.733537
2012-03-12 09:30:00    1.302736
2012-03-13 09:30:00    0.022199
2012-03-14 09:30:00    0.364287
2012-03-15 09:30:00   -0.922839
2012-03-16 09:30:00    0.312656
2012-03-19 09:30:00   -1.128497
2012-03-20 09:30:00   -0.333488
Freq: B, dtype: float64

In [143]: ts1 = ts[:7].tz_localize('Europe/London')
In [144]: ts2 = ts1[2:].tz_convert('Europe/Moscow')
In [145]: result = ts1 + ts2

In [146]: result.index
Out[146]:
DatetimeIndex(['2012-03-07 09:30:00+00:00', '2012-03-08 09:30:00',
               '2012-03-09 09:30:00+00:00', '2012-03-12 09:30:00',
               '2012-03-13 09:30:00+00:00', '2012-03-14 09:30:00',
               '2012-03-15 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='B')
```

# Periods and Period Arithmetic

*Periods* represent time spans, like days, months, quarters, or years. The `Period` class represents this data type, requiring a string or integer and a frequency from the above table:

```
In [147]: p = pd.Period(2007, freq='A-DEC')  
  
In [148]: p  
Out[148]: Period('2007', 'A-DEC')
```

In this case, the `Period` object represents the full timespan from January 1, 2007 to December 31, 2007, inclusive. Conveniently, adding and subtracting integers from periods has the effect of shifting by their frequency:

```
In [149]: p + 5  
Out[149]: Period('2012', 'A-DEC')  
  
In [150]: p - 2  
Out[150]: Period('2005', 'A-DEC')
```

If two periods have the same frequency, their difference is the number of units between them:

```
In [151]: pd.Period('2014', freq='A-DEC') - p  
Out[151]: 7
```

Regular ranges of periods can be constructed using the `period_range` function:

```
In [152]: rng = pd.period_range('2000-01-01', '2000-06-30', freq='M')  
  
In [153]: rng  
Out[153]: PeriodIndex(['2000-01', '2000-02', '2000-03', '2000-04', '2000-05', '2000-06'], freq='M')
```

The `PeriodIndex` class stores a sequence of periods and can serve as an axis index in any pandas data structure:

```
In [154]: pd.Series(np.random.randn(6), index=rng)  
Out[154]:
```

```
2000-01    -0.514551
2000-02    -0.559782
2000-03    -0.783408
2000-04    -1.797685
2000-05    -0.172670
2000-06     0.680215
Freq: M, dtype: float64
```

If you have an array of strings, you can also appeal to the `PeriodIndex` class itself:

```
In [155]: values = ['2001Q3', '2002Q2', '2003Q1']

In [156]: index = pd.PeriodIndex(values, freq='Q-DEC')

In [157]: index
Out[157]: PeriodIndex(['2001Q3', '2002Q2', '2003Q1'], dtype='p
```

# Period Frequency Conversion

Periods and `PeriodIndex` objects can be converted to another frequency using their `asfreq` method. As an example, suppose we had an annual period and wanted to convert it into a monthly period either at the start or end of the year. This is fairly straightforward:

```
In [158]: p = pd.Period('2007', freq='A-DEC')
```

```
In [159]: p.asfreq('M', how='start')
Out[159]: Period('2007-01', 'M')
```

```
In [160]: p.asfreq('M', how='end')
Out[160]: Period('2007-12', 'M')
```

You can think of `Period('2007', 'A-DEC')` as being a sort of cursor pointing to a span of time, subdivided by monthly periods. See [Figure 11-1](#) for an illustration of this. For a *fiscal year* ending on a month other than December, the monthly subperiods belonging are different:

```
In [161]: p = pd.Period('2007', freq='A-JUN')
```

```
In [162]: p.asfreq('M', 'start')
Out[162]: Period('2006-07', 'M')
```

```
In [163]: p.asfreq('M', 'end')
Out[163]: Period('2007-06', 'M')
```

When converting from high to low frequency, the superperiod will be determined depending on where the subperiod “belongs”. For example, in `A-JUN` frequency, the month `Aug-2007` is actually part of the 2008 period:

```
In [164]: p = pd.Period('Aug-2007', 'M')
```

```
In [165]: p.asfreq('A-JUN')
Out[165]: Period('2008', 'A-JUN')
```

Whole `PeriodIndex` objects or time series can be similarly converted with the same semantics:

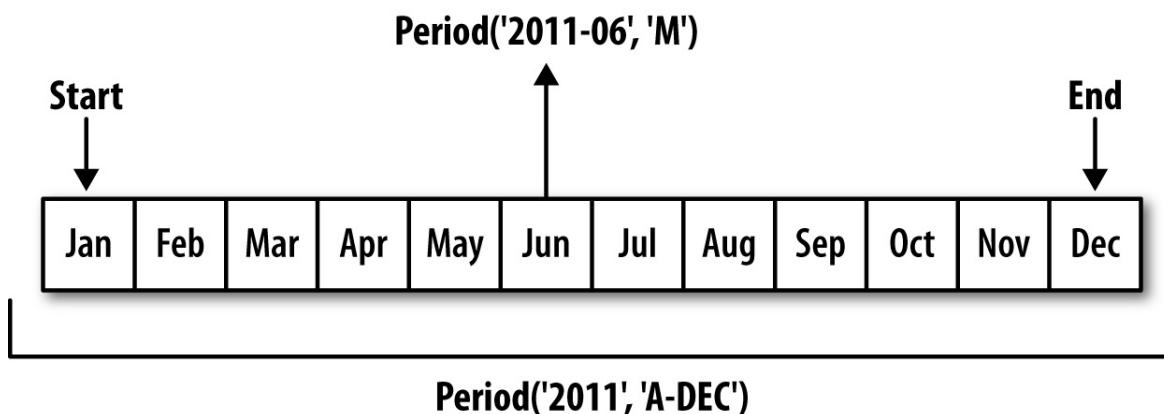
```
In [166]: rng = pd.period_range('2006', '2009', freq='A-DEC')

In [167]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [168]: ts
Out[168]:
2006    1.607578
2007    0.200381
2008   -0.834068
2009   -0.302988
Freq: A-DEC, dtype: float64

In [169]: ts.asfreq('M', how='start')
Out[169]:
2006-01    1.607578
2007-01    0.200381
2008-01   -0.834068
2009-01   -0.302988
Freq: M, dtype: float64

In [170]: ts.asfreq('B', how='end')
Out[170]:
2006-12-29    1.607578
2007-12-31    0.200381
2008-12-31   -0.834068
2009-12-31   -0.302988
Freq: B, dtype: float64
```



**Figure 11-1. Period frequency conversion illustration**

# Quarterly Period Frequencies

Quarterly data is standard in accounting, finance, and other fields. Much quarterly data is reported relative to a *fiscal year end*, typically the last calendar or business day of one of the 12 months of the year. As such, the period 2012Q4 has a different meaning depending on fiscal year end. pandas supports all 12 possible quarterly frequencies as Q-JAN through Q-DEC:

```
In [171]: p = pd.Period('2012Q4', freq='Q-JAN')
```

```
In [172]: p  
Out[172]: Period('2012Q4', 'Q-JAN')
```

In the case of fiscal year ending in January, 2012Q4 runs from November through January, which you can check by converting to daily frequency. See [Figure 11-2](#) for an illustration:

```
In [173]: p.asfreq('D', 'start')  
Out[173]: Period('2011-11-01', 'D')
```

```
In [174]: p.asfreq('D', 'end')  
Out[174]: Period('2012-01-31', 'D')
```

Thus, it's possible to do easy period arithmetic; for example, to get the timestamp at 4PM on the 2nd to last business day of the quarter, you could do:

```
In [175]: p4pm = (p.asfreq('B', 'e') - 1).asfreq('T', 's') + 168 * 9600000
```

```
In [176]: p4pm  
Out[176]: Period('2012-01-30 16:00', 'T')
```

```
In [177]: p4pm.to_timestamp()  
Out[177]: Timestamp('2012-01-30 16:00:00')
```

| Year 2012    |        |     |        |     |        |     |        |     |     |     |     |     |
|--------------|--------|-----|--------|-----|--------|-----|--------|-----|-----|-----|-----|-----|
| M            | JAN    | FEB | MAR    | APR | MAY    | JUN | JUL    | AUG | SEP | OCT | NOV | DEC |
| <b>Q-DEC</b> | 2012Q1 |     | 2012Q2 |     | 2012Q3 |     | 2012Q4 |     |     |     |     |     |
| <b>Q-SEP</b> | 2012Q2 |     | 2012Q3 |     | 2012Q4 |     | 2013Q1 |     |     |     |     |     |
| <b>Q-FEB</b> | 2012Q4 |     | 2013Q1 |     | 2013Q2 |     | 2013Q3 |     | Q4  |     |     |     |

**Figure 11-2. Different quarterly frequency conventions**

You can generate quarterly ranges works using `period_range`. Arithmetic is identical, too:

```
In [178]: rng = pd.period_range('2011Q3', '2012Q4', freq='Q-JAN')

In [179]: ts = pd.Series(np.arange(len(rng)), index=rng)

In [180]: ts
Out[180]:
2011Q3    0
2011Q4    1
2012Q1    2
2012Q2    3
2012Q3    4
2012Q4    5
Freq: Q-JAN, dtype: int64

In [181]: new_rng = (rng.asfreq('B', 'e') - 1).asfreq('T', 's')

In [182]: ts.index = new_rng.to_timestamp()

In [183]: ts
Out[183]:
2010-10-28 16:00:00    0
2011-01-28 16:00:00    1
2011-04-28 16:00:00    2
2011-07-28 16:00:00    3
2011-10-28 16:00:00    4
2012-01-30 16:00:00    5
dtype: int64
```

# Converting Timestamps to Periods (and Back)

Series and DataFrame objects indexed by timestamps can be converted to periods using the `to_period` method:

```
In [184]: rng = pd.date_range('2000-01-01', periods=3, freq='M')
In [185]: ts = pd.Series(np.random.randn(3), index=rng)
In [186]: pts = ts.to_period()

In [187]: ts
Out[187]:
2000-01-31    1.663261
2000-02-29   -0.996206
2000-03-31    1.521760
Freq: M, dtype: float64

In [188]: pts
Out[188]:
2000-01    1.663261
2000-02   -0.996206
2000-03    1.521760
Freq: M, dtype: float64
```

Since periods refer to non-overlapping timespans, a timestamp can only belong to a single period for a given frequency. While the frequency of the new `PeriodIndex` is inferred from the timestamps by default, you can specify any frequency you want. There is also no problem with having duplicate periods in the result:

```
In [189]: rng = pd.date_range('1/29/2000', periods=6, freq='D')
In [190]: ts2 = pd.Series(np.random.randn(6), index=rng)

In [191]: ts2.to_period('M')
Out[191]:
2000-01    0.244175
2000-01    0.423331
2000-01   -0.654040
```

```
2000-02      2.089154
2000-02     -0.060220
2000-02     -0.167933
Freq: M, dtype: float64
```

To convert back to timestamps, use `to_timestamp`:

```
In [192]: pts = ts.to_period()

In [193]: pts
Out[193]:
2000-01      1.663261
2000-02     -0.996206
2000-03      1.521760
Freq: M, dtype: float64

In [194]: pts.to_timestamp(how='end')
Out[194]:
2000-01-31    1.663261
2000-02-29   -0.996206
2000-03-31    1.521760
Freq: M, dtype: float64
```

# Creating a PeriodIndex from Arrays

Fixed frequency data sets are sometimes stored with timespan information spread across multiple columns. For example, in this macroeconomic data set, the year and quarter are in different columns:

```
In [195]: data = pd.read_csv('examples/macrodata.csv')
```

```
In [196]: data.year
```

```
Out[196]:
```

```
0      1959.0
```

```
1      1959.0
```

```
2      1959.0
```

```
3      1959.0
```

```
4      1960.0
```

```
5      1960.0
```

```
6      1960.0
```

```
7      1960.0
```

```
8      1961.0
```

```
9      1961.0
```

```
...
```

```
193     2007.0
```

```
194     2007.0
```

```
195     2007.0
```

```
196     2008.0
```

```
197     2008.0
```

```
198     2008.0
```

```
199     2008.0
```

```
200     2009.0
```

```
201     2009.0
```

```
202     2009.0
```

```
Name: year, Length: 203, dtype: float64
```

```
In [197]: data.quarter
```

```
Out[197]:
```

```
0      1.0
```

```
1      2.0
```

```
2      3.0
```

```
3      4.0
```

```
4      1.0
```

```
5      2.0
```

```
6      3.0
```

```
7      4.0
```

```
8      1.0
9      2.0
...
193     2.0
194     3.0
195     4.0
196     1.0
197     2.0
198     3.0
199     4.0
200     1.0
201     2.0
202     3.0
Name: quarter, Length: 203, dtype: float64
```

By passing these arrays to `PeriodIndex` with a frequency, they can be combined to form an index for the DataFrame:

```
In [198]: index = pd.PeriodIndex(year=data.year, quarter=data.quarter)

In [199]: index
Out[199]:
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1',
             '1960Q3', '1960Q4', '1961Q1', '1961Q2',
             ...
             '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2',
             '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', length=203, freq='Q-DEC')

In [200]: data.index = index

In [201]: data.infl
Out[201]:
1959Q1      0.00
1959Q2      2.34
1959Q3      2.74
1959Q4      0.27
1960Q1      2.31
1960Q2      0.14
1960Q3      2.70
1960Q4      1.21
1961Q1     -0.40
1961Q2      1.47
...
2007Q2      2.75
2007Q3      3.45
2007Q4      6.38
```

```
2008Q1      2.82
2008Q2      8.53
2008Q3     -3.16
2008Q4     -8.79
2009Q1      0.94
2009Q2      3.37
2009Q3      3.56
Freq: Q-DEC, Name: infl, Length: 203, dtype: float64
```

# Resampling and Frequency Conversion

*Resampling* refers to the process of converting a time series from one frequency to another. Aggregating higher frequency data to lower frequency is called *downsampling*, while converting lower frequency to higher frequency is called *upsampling*. Not all resampling falls into either of these categories; for example, converting `W-WED` (weekly on Wednesday) to `W-FRI` is neither upsampling nor downsampling.

pandas objects are equipped with a `resample` method, which is the workhorse function for all frequency conversion. `resample` has a similar API to `groupby`; you call `resample` to group the data, then call an aggregation function:

```
In [202]: rng = pd.date_range('2000-01-01', periods=100, freq='H')
In [203]: ts = pd.Series(np.random.randn(len(rng)), index=rng)
In [204]: ts.resample('M').mean()
Out[204]:
2000-01-31    -0.165893
2000-02-29     0.078606
2000-03-31     0.223811
2000-04-30    -0.063643
Freq: M, dtype: float64

In [205]: ts.resample('M', kind='period').mean()
Out[205]:
2000-01    -0.165893
2000-02     0.078606
2000-03     0.223811
2000-04    -0.063643
Freq: M, dtype: float64
```

`resample` is a flexible and high-performance method that can be used to process very large time series. I'll illustrate its semantics and use through a series of examples.

Table 11-4. Resample method arguments

| <b>Argument</b>               | <b>Description</b>   |
|-------------------------------|--|
| <code>freq</code>             | String or DateOffset indicating desired resampled frequency, e.g. 'M', '5min', or <code>Second(15)</code>  |
| <code>axis=0</code>           | Axis to resample on, default axis=0  |
| <code>fill_method=None</code> | How to interpolate when upsampling, as in ' <code>ffill</code> ' or ' <code>bfill</code> '. By default does no interpolation.  |
| <code>closed='None'</code>    | In downsampling, which end of each interval is closed (inclusive), ' <code>right</code> ' or ' <code>left</code> '.  |
| <code>label='None'</code>     | In downsampling, how to label the aggregated result, with the ' <code>right</code> ' or ' <code>left</code> ' bin edge. For example, the 9:30 to 9:35 5-minute interval could be labeled 9:30 or 9:35. |
| <code>loffset=None</code>     | Time adjustment to the bin labels, such as ' <code>-1s</code> ' / <code>Second(-1)</code> to shift the aggregate labels one second earlier   |
| <code>limit=None</code>       | When forward or backward filling, the maximum number of periods to fill  |
| <code>kind=None</code>        | Aggregate to periods (' <code>period</code> ') or timestamps (' <code>timestamp</code> '); defaults to kind of index the time series has   |
| <code>convention=None</code>  | When resampling periods, the convention (' <code>start</code> ' or ' <code>end</code> ') for converting the low frequency period to high frequency. Defaults to ' <code>end</code> '                   |

# Downsampling

Aggregating data to a regular, lower frequency is a pretty normal time series task. The data you're aggregating doesn't need to be fixed frequently; the desired frequency defines *bin edges* that are used to slice the time series into pieces to aggregate. For example, to convert to monthly, '`M`' or '`BM`', the data need to be chopped up into one month intervals. Each interval is said to be *half-open*; a data point can only belong to one interval, and the union of the intervals must make up the whole time frame. There are a couple things to think about when using `resample` to downsample data:

- Which side of each interval is *closed*
- How to label each aggregated bin, either with the start of the interval or the end

To illustrate, let's look at some one-minute data:

```
In [206]: rng = pd.date_range('2000-01-01', periods=12, freq='T')

In [207]: ts = pd.Series(np.arange(12), index=rng)

In [208]: ts
Out[208]:
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
2000-01-01 00:09:00    9
2000-01-01 00:10:00   10
2000-01-01 00:11:00   11
Freq: T, dtype: int64
```

Suppose you wanted to aggregate this data into five-minute chunks or *bars* by taking the sum of each group:

```
In [209]: ts.resample('5min', closed='right').sum()
Out[209]:
1999-12-31 23:55:00      0
2000-01-01 00:00:00     15
2000-01-01 00:05:00     40
2000-01-01 00:10:00     11
Freq: 5T, dtype: int64
```

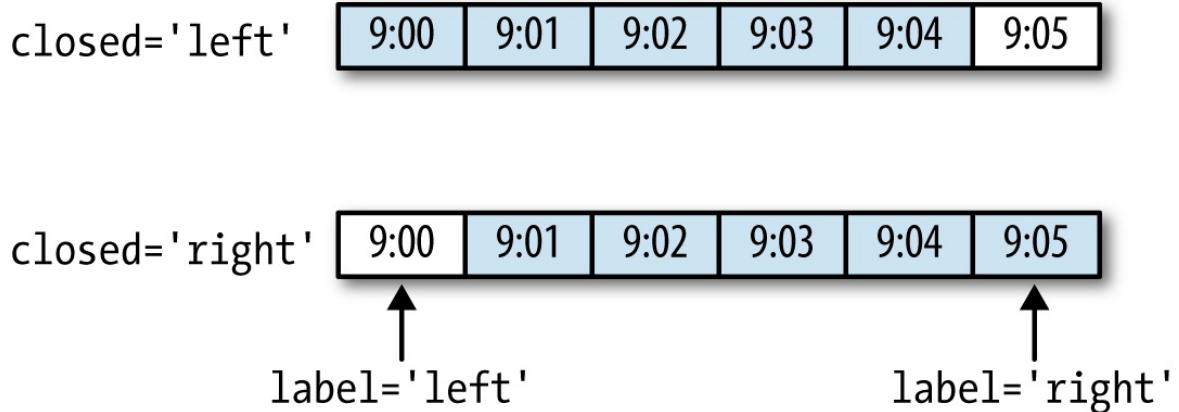
The frequency you pass defines bin edges in five-minute increments. By default, the *left* bin edge is inclusive, so the 00:00 value is included in the 00:00 to 00:05 interval.<sup>1</sup> Passing `closed='right'` changes the interval to be closed on the right:

```
In [210]: ts.resample('5min', closed='right').sum()
Out[210]:
1999-12-31 23:55:00      0
2000-01-01 00:00:00     15
2000-01-01 00:05:00     40
2000-01-01 00:10:00     11
Freq: 5T, dtype: int64
```

The resulting time series is labeled by the timestamps from the left side of each bin. By passing `label='right'` you can label them with the right bin edge:

```
In [211]: ts.resample('5min', closed='right', label='right').sum()
Out[211]:
2000-01-01 00:00:00      0
2000-01-01 00:05:00     15
2000-01-01 00:10:00     40
2000-01-01 00:15:00     11
Freq: 5T, dtype: int64
```

See [Figure 11-3](#) for an illustration of minutely data being resampled to five-minute.



**Figure 11-3. 5-minute resampling illustration of closed, label conventions**

Lastly, you might want to shift the result index by some amount, say subtracting one second from the right edge to make it more clear which interval the timestamp refers to. To do this, pass a string or date offset to `loffset`:

```
In [212]: ts.resample('5min', closed='right',
.....:                      label='right', loffset='-1s').sum()
Out[212]:
1999-12-31 23:59:59      0
2000-01-01 00:04:59     15
2000-01-01 00:09:59     40
2000-01-01 00:14:59     11
Freq: 5T, dtype: int64
```

This also could have been accomplished by calling the `shift` method on the result without the `loffset`.

## Open-High-Low-Close (OHLC) resampling

In finance, a popular way to aggregate a time series is to compute four values for each bucket: the first (open), last (close), maximum (high), and minimal (low) values. By using the `ohlc` aggregate function you will obtain a DataFrame having columns containing these four aggregates, which are efficiently computed in a single sweep of the data:

```
In [213]: ts.resample('5min').ohlc()
Out[213]:
          open   high   low   close
```

|                     |    |    |    |    |
|---------------------|----|----|----|----|
| 2000-01-01 00:00:00 | 0  | 4  | 0  | 4  |
| 2000-01-01 00:05:00 | 5  | 9  | 5  | 9  |
| 2000-01-01 00:10:00 | 10 | 11 | 10 | 11 |

# Upsampling and Interpolation

When converting from a low frequency to a higher frequency, no aggregation is needed. Let's consider a DataFrame with some weekly data:

```
In [214]: frame = pd.DataFrame(np.random.randn(2, 4),
.....:                               index=pd.date_range('1/1/2000',
.....:                               columns=['Colorado', 'Texas', 'New York', 'Ohio']

In [215]: frame
Out[215]:
          Colorado      Texas    New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

When using an aggregation function with this data, there is only one value per group, and missing values result in the gaps:

```
In [216]: df_daily = frame.resample('D').sum()

In [217]: df_daily
Out[217]:
          Colorado      Texas    New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06       NaN        NaN        NaN        NaN
2000-01-07       NaN        NaN        NaN        NaN
2000-01-08       NaN        NaN        NaN        NaN
2000-01-09       NaN        NaN        NaN        NaN
2000-01-10       NaN        NaN        NaN        NaN
2000-01-11       NaN        NaN        NaN        NaN
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

Suppose you wanted to fill forward each weekly value on the non-Wednesdays. The same filling or interpolation methods available in the `fillna` and `reindex` methods are available for resampling:

```
In [218]: frame.resample('D').ffill()
Out[218]:
          Colorado      Texas    New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-07 -0.896431  0.677263  0.036503  0.087102
```

```
2000-01-08 -0.896431 0.677263 0.036503 0.087102
2000-01-09 -0.896431 0.677263 0.036503 0.087102
2000-01-10 -0.896431 0.677263 0.036503 0.087102
2000-01-11 -0.896431 0.677263 0.036503 0.087102
2000-01-12 -0.046662 0.927238 0.482284 -0.867130
```

You can similarly choose to only fill a certain number of periods forward to limit how far to continue using an observed value:

```
In [219]: frame.resample('D').ffill(limit=2)
Out[219]:
```

|            | Colorado  | Texas    | New York | Ohio      |
|------------|-----------|----------|----------|-----------|
| 2000-01-05 | -0.896431 | 0.677263 | 0.036503 | 0.087102  |
| 2000-01-06 | -0.896431 | 0.677263 | 0.036503 | 0.087102  |
| 2000-01-07 | -0.896431 | 0.677263 | 0.036503 | 0.087102  |
| 2000-01-08 | NaN       | NaN      | NaN      | NaN       |
| 2000-01-09 | NaN       | NaN      | NaN      | NaN       |
| 2000-01-10 | NaN       | NaN      | NaN      | NaN       |
| 2000-01-11 | NaN       | NaN      | NaN      | NaN       |
| 2000-01-12 | -0.046662 | 0.927238 | 0.482284 | -0.867130 |

Notably, the new date index need not overlap with the old one at all:

```
In [220]: frame.resample('W-THU').ffill()
Out[220]:
```

|            | Colorado  | Texas    | New York | Ohio      |
|------------|-----------|----------|----------|-----------|
| 2000-01-06 | -0.896431 | 0.677263 | 0.036503 | 0.087102  |
| 2000-01-13 | -0.046662 | 0.927238 | 0.482284 | -0.867130 |

# Resampling with Periods

Resampling data indexed by periods is similar to timestamps:

```
In [221]: frame = pd.DataFrame(np.random.randn(24, 4),
.....:                               index=pd.period_range('1-2000',
.....:                                         columns=['Colorado', 'Texas', 'I
In [222]: frame[:5]
Out[222]:
    Colorado      Texas     New York      Ohio
2000-01  0.493841 -0.155434  1.397286  1.507055
2000-02 -1.179442  0.443171  1.395676 -0.529658
2000-03  0.787358  0.248845  0.743239  1.267746
2000-04  1.302395 -0.272154 -0.051532 -0.467740
2000-05 -1.040816  0.426419  0.312945 -1.115689

In [223]: annual_frame = frame.resample('A-DEC').mean()

In [224]: annual_frame
Out[224]:
    Colorado      Texas     New York      Ohio
2000  0.556703  0.016631  0.111873 -0.027445
2001  0.046303  0.163344  0.251503 -0.157276
```

Upsampling is more nuanced as you must make a decision about which end of the timespan in the new frequency to place the values before resampling, just like the `asfreq` method. The `convention` argument defaults to '`end`' but can also be '`start`':

```
# Q-DEC: Quarterly, year ending in December
In [225]: annual_frame.resample('Q-DEC').ffill()
Out[225]:
    Colorado      Texas     New York      Ohio
2000Q1  0.556703  0.016631  0.111873 -0.027445
2000Q2  0.556703  0.016631  0.111873 -0.027445
2000Q3  0.556703  0.016631  0.111873 -0.027445
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.046303  0.163344  0.251503 -0.157276
2001Q2  0.046303  0.163344  0.251503 -0.157276
2001Q3  0.046303  0.163344  0.251503 -0.157276
2001Q4  0.046303  0.163344  0.251503 -0.157276
```

```
In [226]: annual_frame.resample('Q-DEC', convention='start').ffill()
Out[226]:
          Colorado      Texas   New York      Ohio
2000Q1    0.556703  0.016631  0.111873 -0.027445
2000Q2    0.556703  0.016631  0.111873 -0.027445
2000Q3    0.556703  0.016631  0.111873 -0.027445
2000Q4    0.556703  0.016631  0.111873 -0.027445
2001Q1    0.046303  0.163344  0.251503 -0.157276
2001Q2    0.046303  0.163344  0.251503 -0.157276
2001Q3    0.046303  0.163344  0.251503 -0.157276
2001Q4    0.046303  0.163344  0.251503 -0.157276
```

Since periods refer to timespans, the rules about upsampling and downsampling are more rigid:

- In downsampling, the target frequency must be a *subperiod* of the source frequency.
- In upsampling, the target frequency must be a *superperiod* of the source frequency.

If these rules are not satisfied, an exception will be raised. This mainly affects the quarterly, annual, and weekly frequencies; for example, the timespans defined by Q-MAR only line up with A-MAR, A-JUN, A-SEP, and A-DEC:

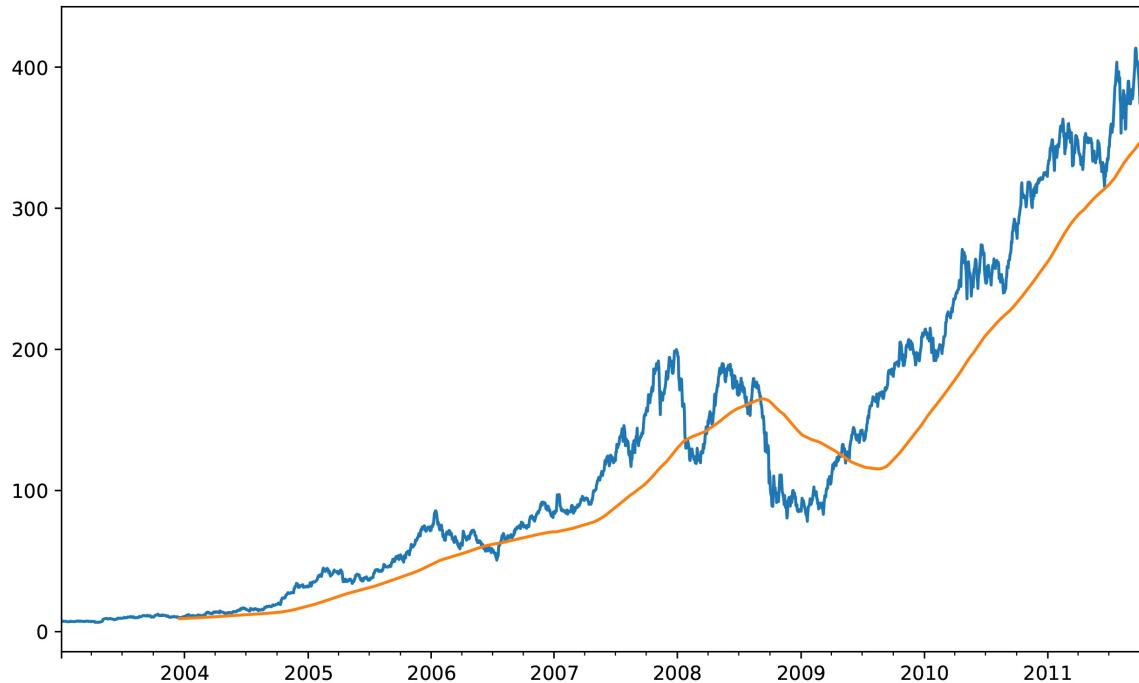
```
In [227]: annual_frame.resample('Q-MAR').ffill()
Out[227]:
          Colorado      Texas   New York      Ohio
2000Q4    0.556703  0.016631  0.111873 -0.027445
2001Q1    0.556703  0.016631  0.111873 -0.027445
2001Q2    0.556703  0.016631  0.111873 -0.027445
2001Q3    0.556703  0.016631  0.111873 -0.027445
2001Q4    0.046303  0.163344  0.251503 -0.157276
2002Q1    0.046303  0.163344  0.251503 -0.157276
2002Q2    0.046303  0.163344  0.251503 -0.157276
2002Q3    0.046303  0.163344  0.251503 -0.157276
```

# Moving Window Functions

An important class of array transformations used for time series operations are statistics and other functions evaluated over a sliding window or with exponentially decaying weights. I call these *moving window functions*, even though it includes functions without a fixed-length window like exponentially-weighted moving average. Like other statistical functions, these also automatically exclude missing data.

I introduce the `rolling` operator, which behaves similarly to `resample` and `groupby`. It can be called on a Series or DataFrame along with a `window` (expressed as a number of periods):

```
In [228]: close_px_all = pd.read_csv('examples/stock_px_2.csv',  
In [229]: close_px = close_px_all[['AAPL', 'MSFT', 'XOM']]  
In [230]: close_px = close_px.resample('B').ffill()  
In [231]: close_px.AAPL.plot()  
Out[231]: <matplotlib.axes._subplots.AxesSubplot at 0x7f6873091  
In [232]: close_px.AAPL.rolling(250).mean().plot()
```



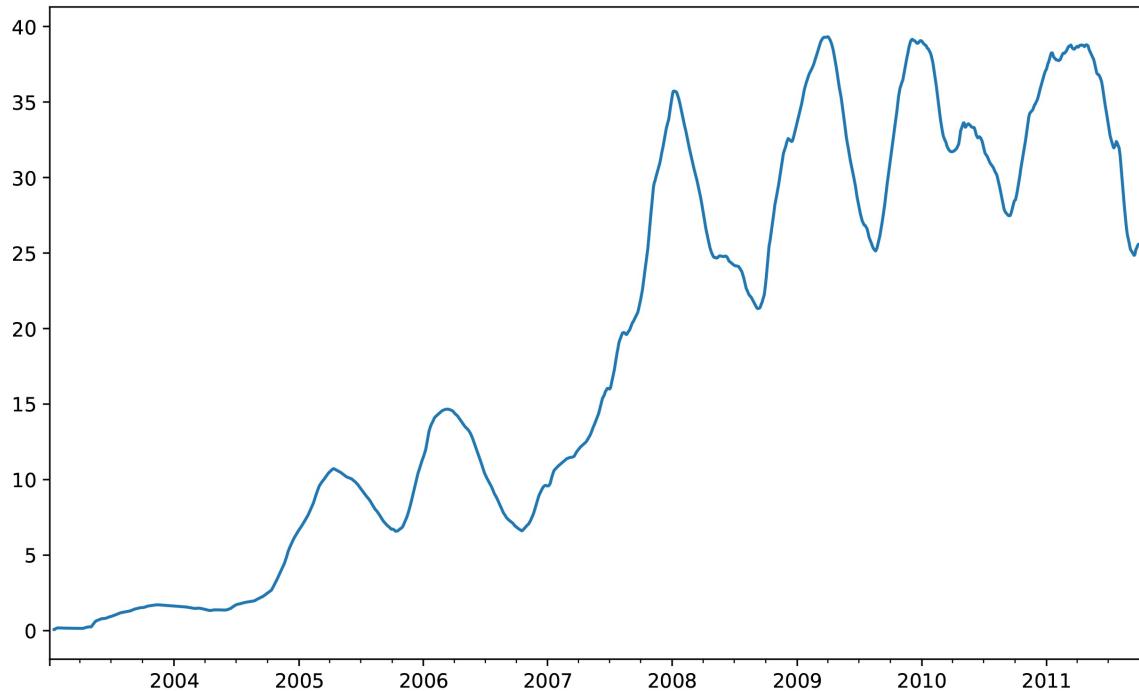
**Figure 11-4. Apple Price with 250-day MA**

See [Figure 11-4](#) for the plot. By default rolling functions require all of the values in the window to be non-NA. This behavior can be changed to account for missing data and, in particular, the fact that you will have fewer than `window` periods of data at the beginning of the time series (see [Figure 11-5](#)):

```
In [234]: appl_std250 = close_px.AAPL.rolling(250, min_periods=5).std()

In [235]: appl_std250[5:12]
Out[235]:
2003-01-09      NaN
2003-01-10      NaN
2003-01-13      NaN
2003-01-14      NaN
2003-01-15    0.077496
2003-01-16    0.074760
2003-01-17    0.112368
Freq: B, Name: AAPL, dtype: float64

In [236]: appl_std250.plot()
```



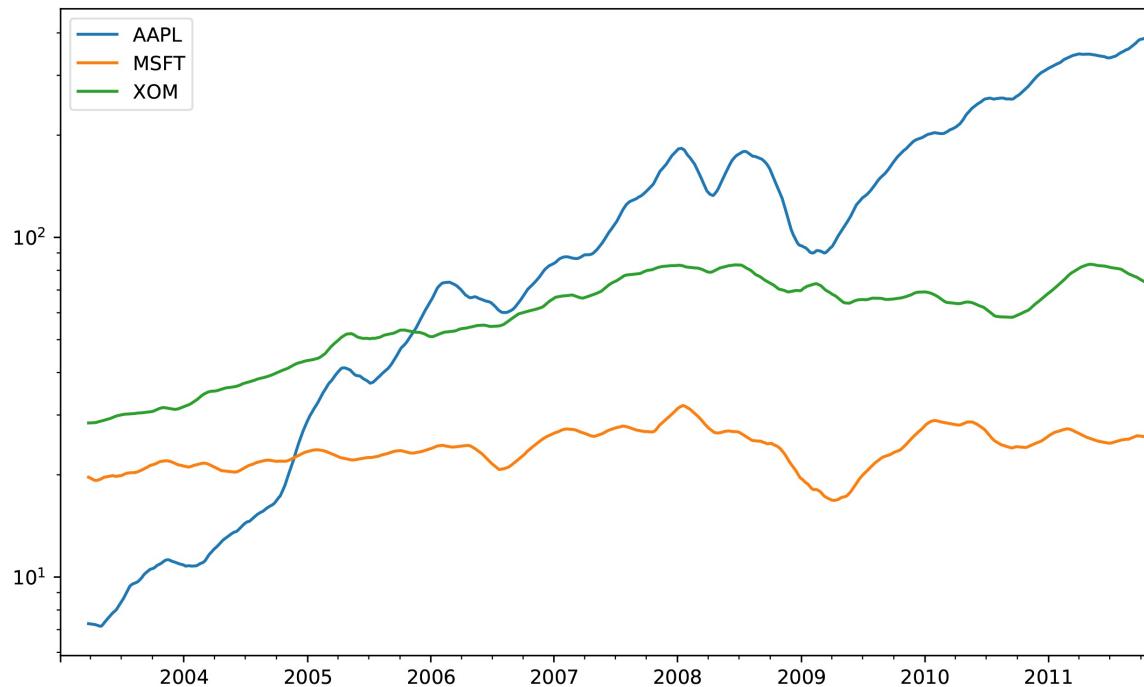
**Figure 11-5. Apple 250-day daily return standard deviation**

To compute an *expanding window mean*, use the `expanding` operator instead of `rolling`:

```
In [237]: expanding_mean = appl_std250.expanding().mean()
```

Calling `rolling_mean` and friends on a DataFrame applies the transformation to each column (see [Figure 11-6](#)):

```
In [239]: close_px.rolling(60).mean().plot(logy=True)
```



**Figure 11-6. Stocks Prices 60-day MA (log Y-axis)**

# Exponentially-weighted functions

An alternative to using a static window size with equally-weighted observations is to specify a constant *decay factor* to give more weight to more recent observations. In mathematical terms, if  $ma_t$  is the moving average result at time  $t$  and  $x$  is the time series in question, each value in the result is computed as  $ma_t = a * ma_{t-1} + (1 - a) * x_t$ , where  $a$  is the decay factor. There are a couple of ways to specify the decay factor, a popular one is using a *span*, which makes the result comparable to a simple moving window function with window size equal to the span.

Since an exponentially-weighted statistic places more weight on more recent observations, it “adapts” faster to changes compared with the equal-weighted version.

pandas has the `ewm` operator to go along with `rolling` and `expanding`. Here’s an example comparing a 60-day moving average of Apple’s stock price with an EW moving average with `span=60` (see [Figure 11-7](#)):

```
In [241]: aapl_px = close_px.AAPL['2006':'2007']

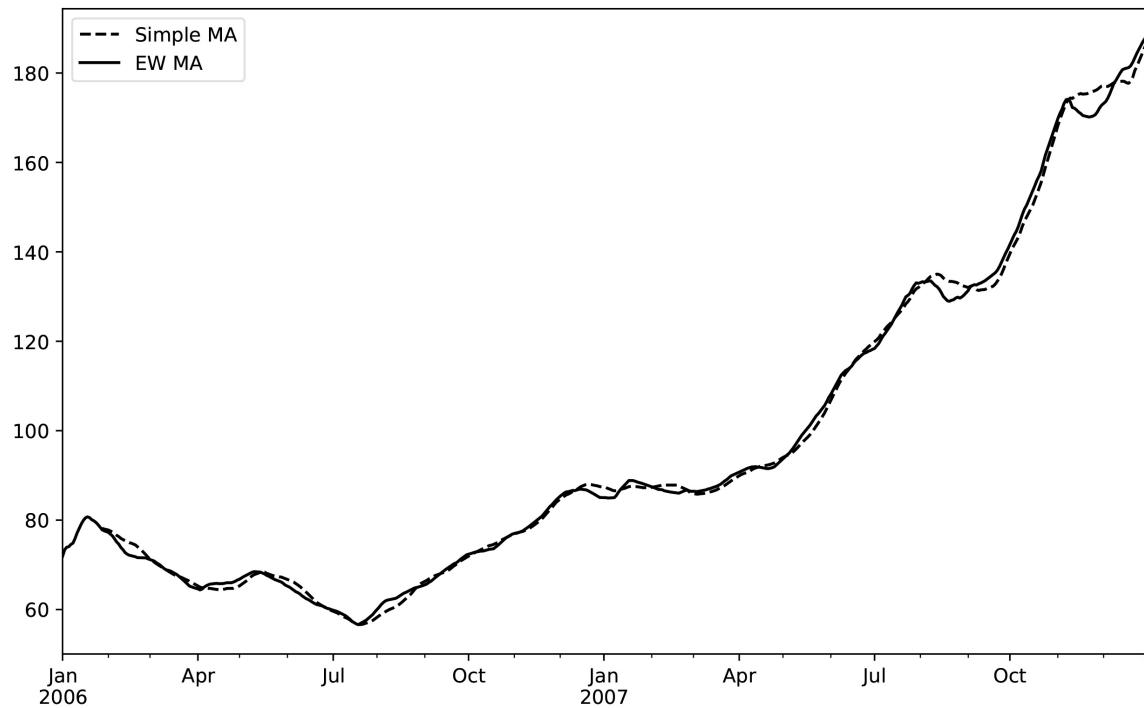
In [242]: ma60 = aapl_px.rolling(30, min_periods=20).mean()

In [243]: ewma60 = aapl_px.ewm(span=30).mean()

In [244]: ma60.plot(style='k--', label='Simple MA')
Out[244]: <matplotlib.axes._subplots.AxesSubplot at 0x7f686d08e

In [245]: ewma60.plot(style='k-', label='EW MA')
Out[245]: <matplotlib.axes._subplots.AxesSubplot at 0x7f686d08e

In [246]: plt.legend()
```



**Figure 11-7. Simple moving average versus exponentially-weighted**

# Binary Moving Window Functions

Some statistical operators, like correlation and covariance, need to operate on two time series. As an example, financial analysts are often interested in a stock's correlation to a benchmark index like the S&P 500. The `corr` aggregation function after calling `rolling` can do this for us (see [Figure 11-8](#)):

```
In [248]: spx_px = close_px_all['SPX']

In [249]: spx_rets = spx_px / spx_px.shift(1) - 1

In [250]: returns = close_px.pct_change()

In [251]: corr = returns.AAPL.rolling(125, min_periods=100).corr

In [252]: corr.plot()
```

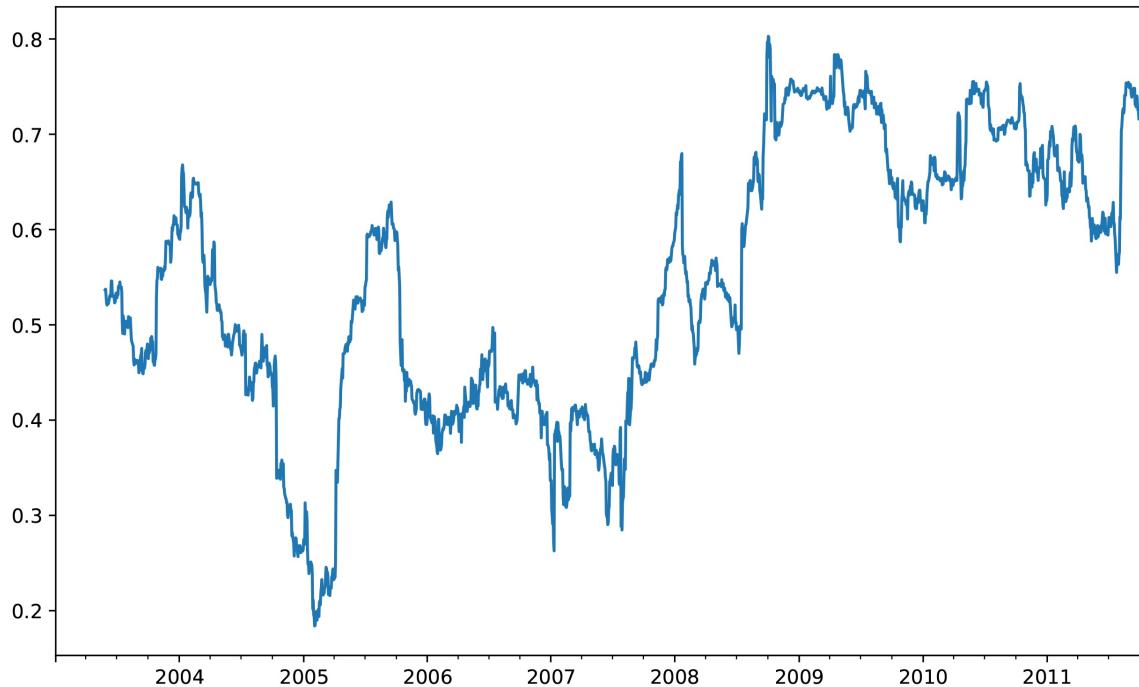
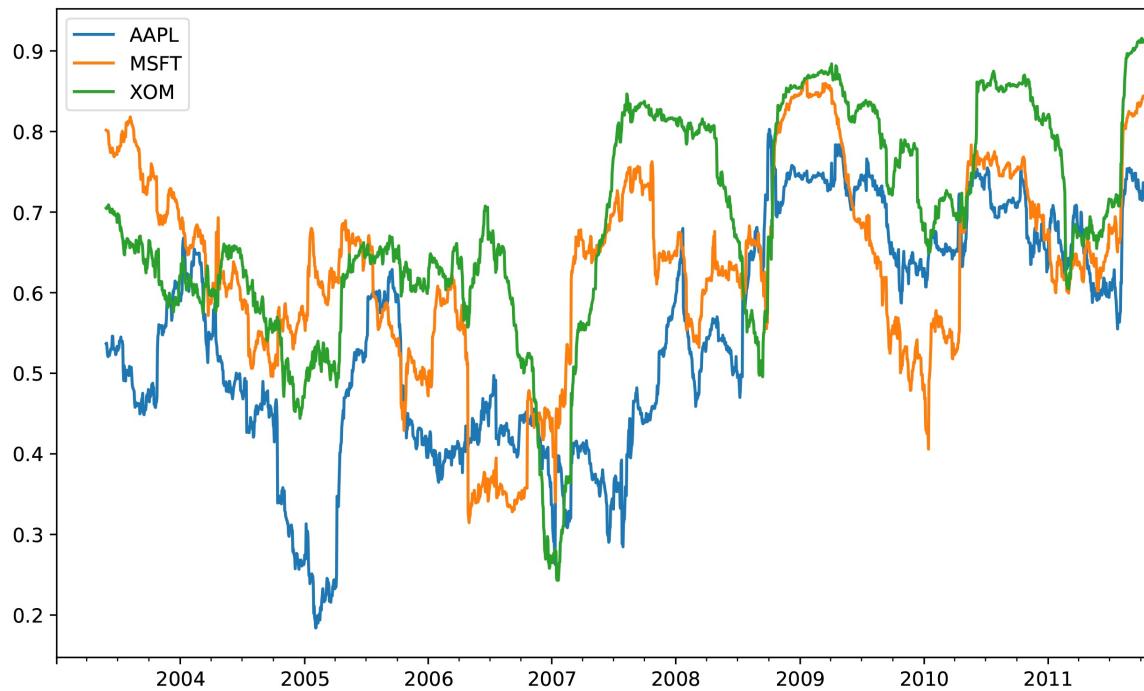


Figure 11-8. 6-month AAPL return correlation to S&P 500

Suppose you wanted to compute the correlation of the S&P 500 index with many stocks at once. Writing a loop and creating a new DataFrame would be easy but maybe get repetitive, so if you pass a TimeSeries and a DataFrame, a function like `rolling_corr` will compute the correlation of the TimeSeries (`spx_rets` in this case) with each column in the DataFrame. See [Figure 11-9](#) for the plot of the result:

```
In [254]: corr = returns.rolling(125, min_periods=100).corr(spx_rets)
```

```
In [255]: corr.plot()
```

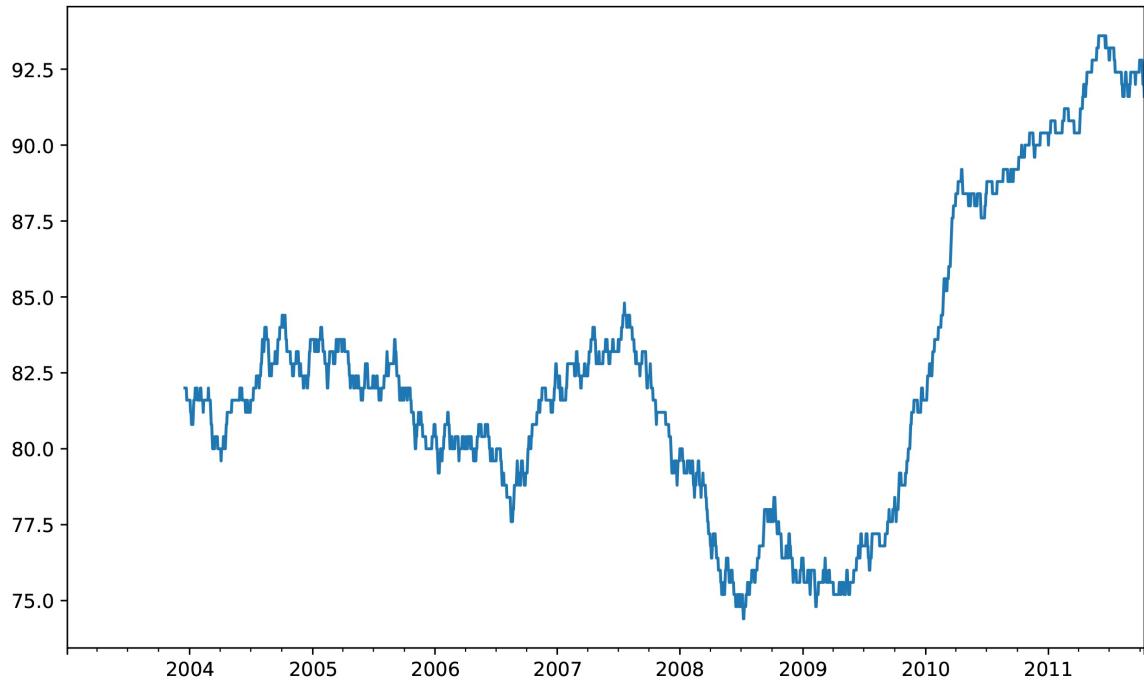


**Figure 11-9. 6-month return correlations to S&P 500**

# User-Defined Moving Window Functions

The `apply` method on `rolling` and related methods provides a means to apply an array function of your own devising over a moving window. The only requirement is that the function produce a single value (a reduction) from each piece of the array. For example, while we can compute sample quantiles using `rolling(...).quantile(q)`, we might be interested in the percentile rank of a particular value over the sample. The `scipy.stats.percentileofscore` function does just this:

```
In [257]: from scipy.stats import percentileofscore  
In [258]: score_at_2percent = lambda x: percentileofscore(x, 0.  
In [259]: result = returns.AAPL.rolling(250).apply(score_at_2per  
In [260]: result.plot()
```



**Figure 11-10. Percentile rank of 2% AAPL return over 1 year window**

<sup>1</sup> The choice of the default values for `closed` and `label` might seem a bit odd to some users. In practice the choice is somewhat arbitrary; for some target frequencies, `closed='left'` is preferable, while for others `closed='right'` makes more sense. The important thing is that you keep in mind exactly how you are segmenting the data.

# Chapter 12. Advanced NumPy

In this chapter, I will go deeper into the NumPy library for array computing. This will include more internal detail about the `ndarray` type and more advanced array manipulations and algorithms.

# ndarray Object Internals

The NumPy ndarray provides a means to interpret a block of homogeneous data (either contiguous or strided, more on this later) as a multidimensional array object. The data type, or *dtype*, determines how the data is interpreted as being floating point, integer, boolean, or any of the other types we've been looking at.

Part of what makes ndarray flexible is that every array object is a *strided* view on a block of data. You might wonder, for example, how the array view `arr[::2, ::-1]` does not copy any data. The reason the ndarray is more than just a chunk of memory and a dtype; it also has “striding” information which enables the array to move through memory with varying step sizes. More precisely, the ndarray internally consists of the following:

- A *pointer to data*, that is a block of data in RAM or in a memory-mapped file
- The *data type* or *dtype*, describing fixed-size value cells in the array
- A tuple indicating the array’s *shape*
- A tuple of *strides*, integers indicating the number of bytes to “step” in order to advance one element along a dimension.

While it is rare that a typical NumPy user would be interested in the array strides, they are the critical ingredient in constructing “zero-copy” array views. Strides can even be negative which enables an array to move “backward” through memory, which would be the case in a slice like `obj[::-1]` or `obj[:, ::-1]`.

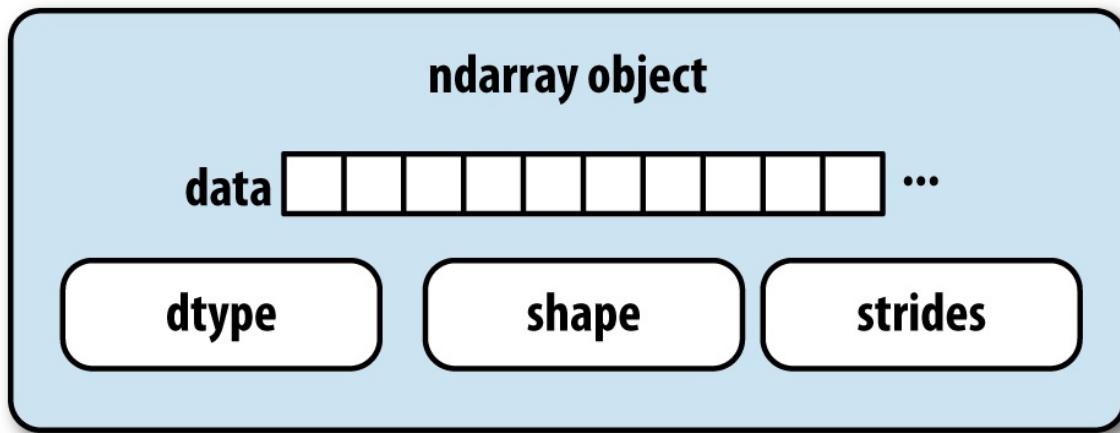
For example, a 10 by 5 array would have shape `(10, 5)`:

```
In [10]: np.ones((10, 5)).shape  
Out[10]: (10, 5)
```

A typical (C order, more on this later)  $3 \times 4 \times 5$  array of `float64` (8-byte) values has strides `(160, 40, 8)`

```
In [11]: np.ones((3, 4, 5), dtype=np.float64).strides  
Out[11]: (160, 40, 8)
```

See [Figure 12-1](#) for a simple mockup the ndarray innards.



**Figure 12-1.** The NumPy ndarray object

# NumPy dtype Hierarchy

You may occasionally have code which needs to check whether an array contains integers, floating point numbers, strings, or Python objects. Because there are multiple types of floating point numbers (`float16` through `float128`), checking that the `dtype` is among a list of types would be very verbose. Fortunately, the dtypes have superclasses such as `np.integer` and `np.floating` which can be used in conjunction with the `np.issubdtype` function:

```
In [12]: ints = np.ones(10, dtype=np.uint16)
In [13]: floats = np.ones(10, dtype=np.float32)
In [14]: np.issubdtype(ints.dtype, np.integer)
Out[14]: True
In [15]: np.issubdtype(floats.dtype, np.floating)
Out[15]: True
```

You can see all of the parent classes of a specific `dtype` by calling the type's `mro` method:

```
In [16]: np.float64.mro()
Out[16]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
 numpy.generic,
 float,
 object]
```

Most NumPy users will never have to know about this, but it occasionally comes in handy. See [Figure 12-2](#) for a graph of the `dtype` hierarchy and parent-subclass relationships <sup>1</sup>.

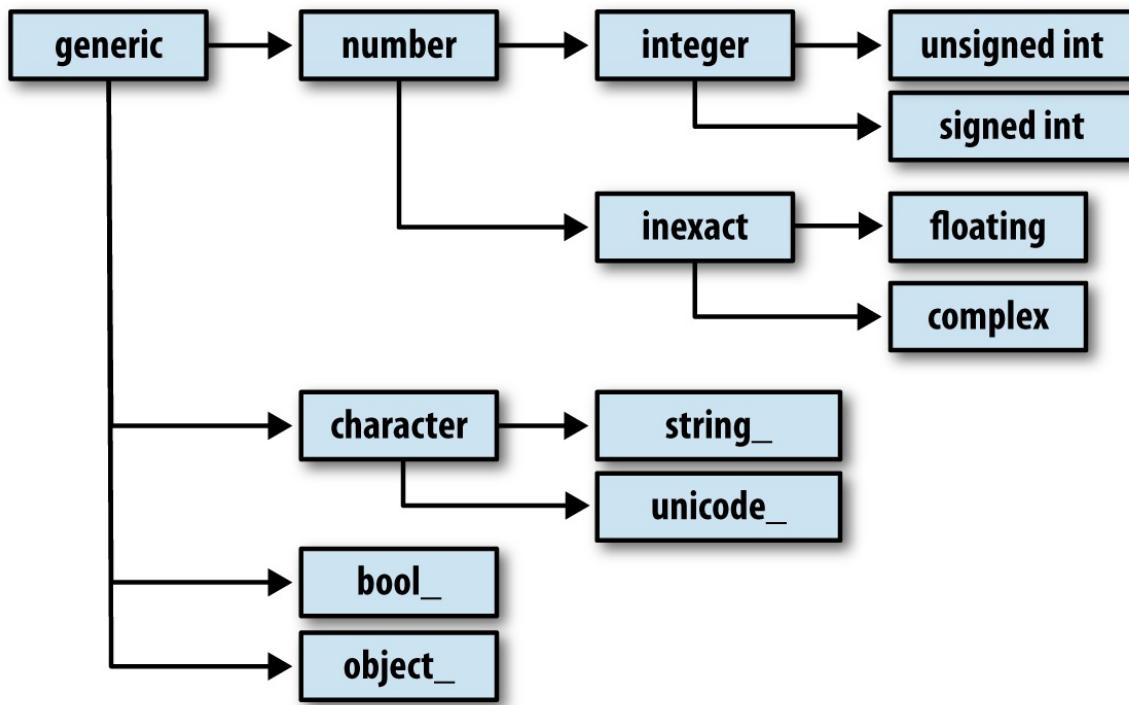


Figure 12-2. The NumPy dtype class hierarchy

# Advanced Array Manipulation

There are many ways to work with arrays beyond fancy indexing, slicing, and boolean subsetting. While much of the heavy lifting for data analysis applications is handled by higher level functions in pandas, you may at some point need to write a data algorithm that is not found in one of the existing libraries.

# Reshaping Arrays

In many cases, you can convert an array from one shape to another without copying any data. To do this, pass a tuple indicating the new shape to the `reshape` array instance method. For example, suppose we had a one-dimensional array of values that we wished to rearrange into a matrix:

```
In [17]: arr = np.arange(8)

In [18]: arr
Out[18]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [19]: arr.reshape((4, 2))
Out[19]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

A multidimensional array can also be reshaped:

```
In [20]: arr.reshape((4, 2)).reshape((2, 4))
Out[20]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

One of the passed shape dimensions can be `-1`, in which case the value used for that dimension will be inferred from the data:

```
In [21]: arr = np.arange(15)

In [22]: arr.reshape((5, -1))
Out[22]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

Since an array's `shape` attribute is a tuple, it can be passed to `reshape`, too:

```
In [23]: other_arr = np.ones((3, 5))

In [24]: other_arr.shape
Out[24]: (3, 5)

In [25]: arr.reshape(other_arr.shape)
Out[25]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

The opposite operation of `reshape` from one-dimensional to a higher dimension is typically known as *flattening* or *raveling*:

```
In [26]: arr = np.arange(15).reshape((5, 3))

In [27]: arr
Out[27]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])

In [28]: arr.ravel()
Out[28]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
```

`ravel` does not produce a copy of the underlying data if it does not have to (more on this below). The `flatten` method behaves like `ravel` except it always returns a copy of the data:

```
In [29]: arr.flatten()
Out[29]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
```

The data can be reshaped or raveled in different orders. This is a slightly nuanced topic for new NumPy users and is therefore the next subtopic.

# C versus Fortran Order

NumPy gives you control and flexibility over the layout of your data in memory. By default, NumPy arrays are created in *row major* order. Spatially this means that if you have a two-dimensional array of data, the items in each row of the array are stored in adjacent memory locations. The alternative to row major ordering is *column major* order, which means that values within each column of data are stored in adjacent memory locations.

For historical reasons, row and column major order are also known as C and Fortran order, respectively. In the FORTRAN 77 language, matrices are all column major.

Functions like `reshape` and `ravel`, accept an `order` argument indicating the order to use the data in the array. This can be '`C`' or '`F`' in most cases (there are also less commonly-used options '`A`' and '`K`'; see the NumPy documentation). These are illustrated in [Figure 12-3](#).

```
In [30]: arr = np.arange(12).reshape((3, 4))

In [31]: arr
Out[31]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [32]: arr.ravel()
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11]

In [33]: arr.ravel('F')
Out[33]: array([ 0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7, 11])
```

Reshaping arrays with more than two dimensions can be a bit mind-bending. The key difference between C and Fortran order is the order in which the dimensions are walked:

- *C / row major order*: traverse higher dimensions *first* (e.g. axis 1 before advancing on axis 0).

- *Fortran / column major order*: traverse higher dimensions *last* (e.g. axis 0 before advancing on axis 1).

# Concatenating and Splitting Arrays

`numpy.concatenate` takes a sequence (tuple, list, etc.) of arrays and joins them together in order along the input axis.

```
In [34]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])  
  
In [35]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])  
  
In [36]: np.concatenate([arr1, arr2], axis=0)  
Out[36]:  
array([[ 1,  2,  3],  
       [ 4,  5,  6],  
       [ 7,  8,  9],  
       [10, 11, 12]])  
  
In [37]: np.concatenate([arr1, arr2], axis=1)  
Out[37]:  
array([[ 1,  2,  3,  7,  8,  9],  
       [ 4,  5,  6, 10, 11, 12]])
```

|   |   |   |   |   |   |   |   |   |   |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|----|----|

`arr.reshape((4, 3), order=?)`

C order (row major)

|   |    |    |
|---|----|----|
| 0 | 1  | 2  |
| 3 | 4  | 5  |
| 6 | 7  | 8  |
| 9 | 10 | 11 |

`order='C'`

Fortran order (column major)

|   |   |    |
|---|---|----|
| 0 | 4 | 8  |
| 1 | 5 | 9  |
| 2 | 6 | 10 |
| 3 | 7 | 11 |

`order='F'`

**Figure 12-3. Reshaping in C (row major) or Fortran (column major) order**

There are some convenience functions, like `vstack` and `hstack`, for common kinds of concatenation. The above operations could have been expressed as:

```
In [38]: np.vstack((arr1, arr2))
Out[38]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
In [39]: np.hstack((arr1, arr2))
Out[39]:
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

`split`, on the other hand, slices apart an array into multiple arrays along an axis:

```
In [40]: arr = np.random.randn(5, 2)
```

```
In [41]: arr
Out[41]:
array([[-0.2047,  0.4789],
       [-0.5194, -0.5557],
       [ 1.9658,  1.3934],
       [ 0.0929,  0.2817],
       [ 0.769 ,  1.2464]])
```

```
In [42]: first, second, third = np.split(arr, [1, 3])
```

```
In [43]: first
Out[43]: array([[-0.2047,  0.4789]])
```

```
In [44]: second
Out[44]:
array([[-0.5194, -0.5557],
       [ 1.9658,  1.3934]])
```

```
In [45]: third
Out[45]:
array([[ 0.0929,  0.2817],
       [ 0.769 ,  1.2464]])
```

See [Table 12-1](#) for a list of all relevant concatenation and splitting functions, some of which are provided only as a convenience of the very general purpose `concatenate`.

Table 12-1. Array concatenation functions

| Function  | Description  |
|---|--|
| <code>concatenate</code>                        | Most general function, concatenates collection of arrays along one axis      |
| <code>vstack</code> ,<br><code>row_stack</code> | Stack arrays row-wise (along axis 0)   |
| <code>hstack</code>                             | Stack arrays column-wise (along axis 1)                                      |
| <code>column_stack</code>                       | Like <code>hstack</code> , but converts 1D arrays to 2D column vectors first |
| <code>dstack</code>                             | Stack arrays “depth”-wise (along axis 2)                                     |
| <code>split</code>                              | Split array at passed locations along a particular axis                      |
| <code>hsplit</code> /<br><code>vsplit</code>    | Convenience functions for splitting on axis 0 and 1, respectively.           |

## Stacking helpers: `r_` and `c_`

There are two special objects in the NumPy namespace, `r_` and `c_`, that make stacking arrays more concise:

```
In [46]: arr = np.arange(6)

In [47]: arr1 = arr.reshape((3, 2))

In [48]: arr2 = np.random.randn(3, 2)

In [49]: np.r_[arr1, arr2]
Out[49]:
array([[ 0.      ,  1.      ],
       [ 2.      ,  3.      ],
       [ 4.      ,  5.      ],
       [ 1.0072  , -1.2962 ],
       [ 0.275   ,  0.2289],
       [ 1.3529 ,  0.8864]])
```

```
In [50]: np.c_[np.r_[arr1, arr2], arr]
```

```
Out[50]:  
array([[ 0.        ,  1.        ,  0.        ],  
       [ 2.        ,  3.        ,  1.        ],  
       [ 4.        ,  5.        ,  2.        ],  
       [ 1.0072   , -1.2962   ,  3.        ],  
       [ 0.275    ,  0.2289   ,  4.        ],  
       [ 1.3529   ,  0.8864   ,  5.        ]])
```

These additionally can translate slices to arrays:

```
In [51]: np.c_[1:6, -10:-5]  
Out[51]:  
array([[ 1, -10],  
       [ 2, -9],  
       [ 3, -8],  
       [ 4, -7],  
       [ 5, -6]])
```

See the docstring for more on what you can do with `c_` and `r_`.

# Repeating Elements: Tile and Repeat

## Note

The need to replicate or repeat arrays can be less common with NumPy than it is with other array programming frameworks like MATLAB. One reason for this is that *broadcasting* often fills this need better, which is the subject of the next section.

Two useful tools for repeating or replicating arrays to produce larger arrays are the `repeat` and `tile` functions. `repeat` replicates each element in an array some number of times, producing a larger array:

```
In [52]: arr = np.arange(3)

In [53]: arr.repeat(3)
Out[53]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

By default, if you pass an integer, each element will be repeated that number of times. If you pass an array of integers, each element can be repeated a different number of times:

```
In [54]: arr.repeat([2, 3, 4])
Out[54]: array([0, 0, 1, 1, 1, 2, 2, 2])
```

Multidimensional arrays can have their elements repeated along a particular axis.

```
In [55]: arr = np.random.randn(2, 2)

In [56]: arr
Out[56]:
array([-2.0016, -0.3718],
      [ 1.669 , -0.4386])

In [57]: arr.repeat(2, axis=0)
Out[57]:
array([-2.0016, -0.3718],
      [-2.0016, -0.3718],
      [ 1.669 , -0.4386],
```

```
[ 1.669 , -0.4386]])
```

Note that if no axis is passed, the array will be flattened first, which is likely not what you want. Similarly you can pass an array of integers when repeating a multidimensional array to repeat a given slice a different number of times:

```
In [58]: arr.repeat([2, 3], axis=0)
```

```
Out[58]:
```

```
array([[-2.0016, -0.3718],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386]])
```

```
In [59]: arr.repeat([2, 3], axis=1)
```

```
Out[59]:
```

```
array([[ -2.0016, -2.0016, -0.3718, -0.3718, -0.3718],
       [ 1.669 ,  1.669 , -0.4386, -0.4386, -0.4386]])
```

`tile`, on the other hand, is a shortcut for stacking copies of an array along an axis. You can visually think about it as like “laying down tiles”:

```
In [60]: arr
```

```
Out[60]:
```

```
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])
```

```
In [61]: np.tile(arr, 2)
```

```
Out[61]:
```

```
array([[ -2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386]])
```

The second argument is the number of tiles; with a scalar, the tiling is made row-by-row, rather than column by column: The second argument to `tile` can be a tuple indicating the layout of the “tiling”:

```
In [62]: arr
```

```
Out[62]:
```

```
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])
```

```
In [63]: np.tile(arr, (2, 1))
```

```
Out[63]:
```

```
array([[ -2.0016, -0.3718],
       [ 1.669 , -0.4386],
```

```
[ -2.0016, -0.3718],  
[ 1.669 , -0.4386]] )  
  
In [64]: np.tile(arr, (3, 2))  
Out[64]:  
array([[ -2.0016, -0.3718, -2.0016, -0.3718],  
       [ 1.669 , -0.4386,  1.669 , -0.4386],  
       [-2.0016, -0.3718, -2.0016, -0.3718],  
       [ 1.669 , -0.4386,  1.669 , -0.4386],  
       [-2.0016, -0.3718, -2.0016, -0.3718],  
       [ 1.669 , -0.4386,  1.669 , -0.4386]])
```

# Fancy Indexing Equivalents: Take and Put

As you may recall from [Chapter 4](#), one way to get and set subsets of arrays is by *fancy* indexing using integer arrays:

```
In [65]: arr = np.arange(10) * 100  
In [66]: inds = [7, 1, 2, 6]  
In [67]: arr[inds]  
Out[67]: array([700, 100, 200, 600])
```

There are alternate ndarray methods that are useful in the special case of only making a selection on a single axis:

```
In [68]: arr.take(inds)  
Out[68]: array([700, 100, 200, 600])  
  
In [69]: arr.put(inds, 42)  
  
In [70]: arr  
Out[70]: array([ 0,  42,  42, 300, 400, 500,  42,  42, 800, 900])  
  
In [71]: arr.put(inds, [40, 41, 42, 43])  
  
In [72]: arr  
Out[72]: array([ 0,  41,  42, 300, 400, 500,  43,  40, 800, 900])
```

To use `take` along other axes, you can pass the `axis` keyword:

```
In [73]: inds = [2, 0, 2, 1]  
  
In [74]: arr = np.random.randn(2, 4)  
  
In [75]: arr  
Out[75]:  
array([[ -0.5397,   0.477 ,   3.2489,  -1.0212],  
      [-0.5771,   0.1241,   0.3026,   0.5238]])  
  
In [76]: arr.take(inds, axis=1)  
Out[76]:  
array([[ 3.2489, -0.5397,   3.2489,   0.477 ],
```

```
[ 0.3026, -0.5771,  0.3026,  0.1241] ])
```

`put` does not accept an `axis` argument but rather indexes into the flattened (one-dimensional, C order) version of the array. Thus, when you need to set elements using an index array on other axes, it is often easiest to use fancy indexing.

# Broadcasting

*Broadcasting* describes how arithmetic works between arrays of different shapes. It can be a powerful feature, but one that can cause confusion, even by experienced users. The simplest example of broadcasting occurs when combining a scalar value with an array:

```
In [77]: arr = np.arange(5)

In [78]: arr
Out[78]: array([0, 1, 2, 3, 4])

In [79]: arr * 4
Out[79]: array([ 0,  4,  8, 12, 16])
```

Here we say that the scalar value 4 has been *broadcast* to all of the other elements in the multiplication operation.

For example, we can demean each column of an array by subtracting the column means. In this case, it is very simple:

```
In [80]: arr = np.random.randn(4, 3)

In [81]: arr.mean(0)
Out[81]: array([-0.3928, -0.3824, -0.8768])

In [82]: demeaned = arr - arr.mean(0)

In [83]: demeaned
Out[83]:
array([[ 0.3937,  1.7263,  0.1633],
       [-0.4384, -1.9878, -0.9839],
       [-0.468 ,  0.9426, -0.3891],
       [ 0.5126, -0.6811,  1.2097]])
```

```
In [84]: demeaned.mean(0)
Out[84]: array([-0.,  0., -0.])
```

See [Figure 12-4](#) for an illustration of this operation. Demeaning the rows as a broadcast operation requires a bit more care. Fortunately, broadcasting

potentially lower dimensional values across any dimension of an array (like subtracting the row means from each column of a two-dimensional array) is possible as long as you follow the rules. This brings us to:

### The Broadcasting Rule

Two arrays are compatible for broadcasting if for each *trailing dimension* (that is, starting from the end), the axis lengths match or if either of the lengths is 1. Broadcasting is then performed over the missing and / or length 1 dimensions.

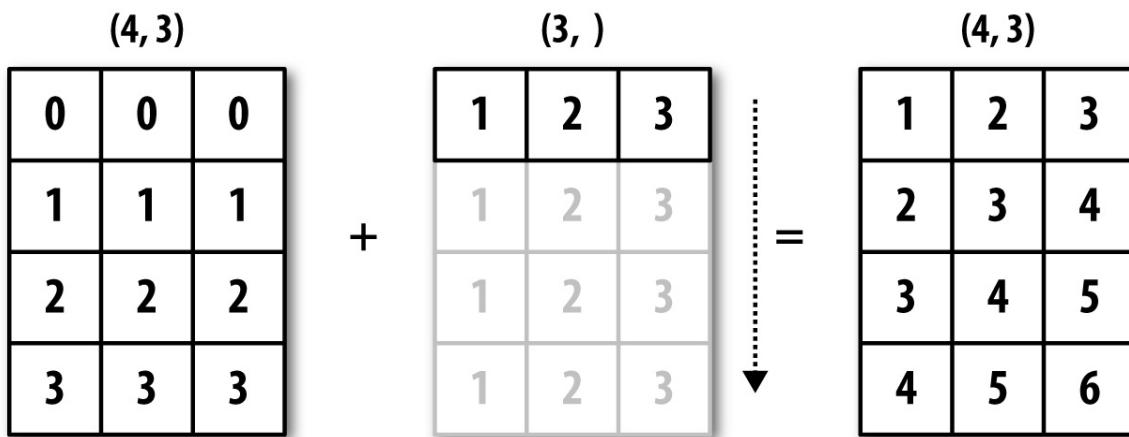


Figure 12-4. Broadcasting over axis 0 with a 1D array

Even as an experienced NumPy user, I often must stop to draw pictures and think about the broadcasting rule. Consider the last example and suppose we wished instead to subtract the mean value from each row. Since `arr.mean(0)` has length 3, it is compatible for broadcasting across axis 0 because the trailing dimension in `arr` is 3 and therefore matches. According to the rules, to subtract over axis 1 (that is, subtract the row mean from each row), the smaller array must have shape `(4, 1)`:

```
In [85]: arr
Out[85]:
array([[ 0.0009,  1.3438, -0.7135],
       [-0.8312, -2.3702, -1.8608],
       [-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329]])
```

```

In [86]: row_means = arr.mean(1)

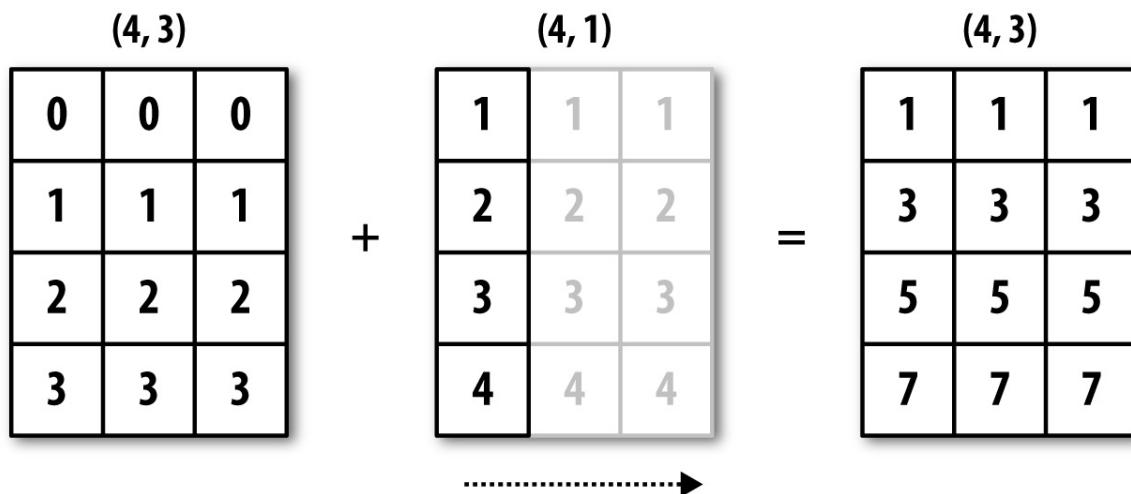
In [87]: row_means.reshape((4, 1))
Out[87]:
array([[ 0.2104],
       [-1.6874],
       [-0.5222],
       [-0.2036]])

In [88]: demeaned = arr - row_means.reshape((4, 1))

In [89]: demeaned.mean(1)
Out[89]: array([ 0., -0.,  0.,  0.])

```

See [Figure 12-5](#) for an illustration of this operation.



**Figure 12-5.** Broadcasting over axis 1 of a 2D array

See [Figure 12-6](#) for another illustration, this time adding a two-dimensional array to a three-dimensional one across axis 0.

# Broadcasting Over Other Axes

Broadcasting with higher dimensional arrays can seem even more mind-bending, but it is really a matter of following the rules. If you don't, you'll get an error like this:

```
In [90]: arr - arr.mean(1)
-----
ValueError                                     Traceback (most recent call last)
<ipython-input-90-7b87b85a20b2> in <module>()
----> 1 arr - arr.mean(1)
ValueError: operands could not be broadcast together with shapes
```

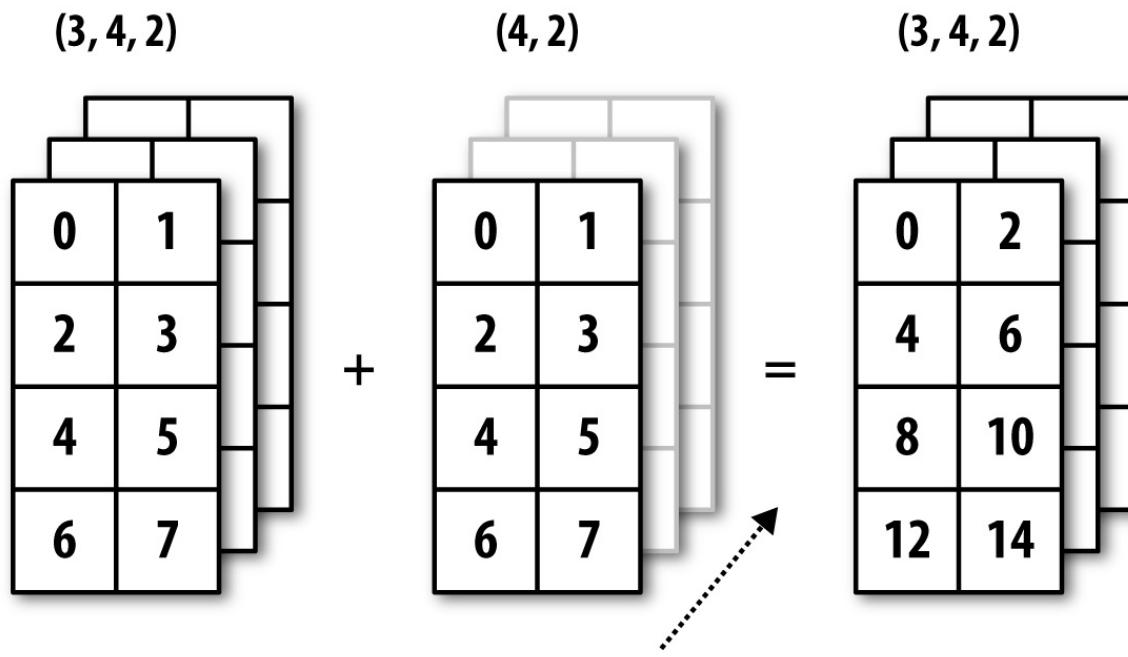


Figure 12-6. Broadcasting over axis 0 of a 3D array

It's quite common to want to perform an arithmetic operation with a lower dimensional array across axes other than axis 0. According to the broadcasting rule, the "broadcast dimensions" must be 1 in the smaller array. In the example of row demeaning above this meant reshaping the row means to be shape (4, 1) instead of (4,):

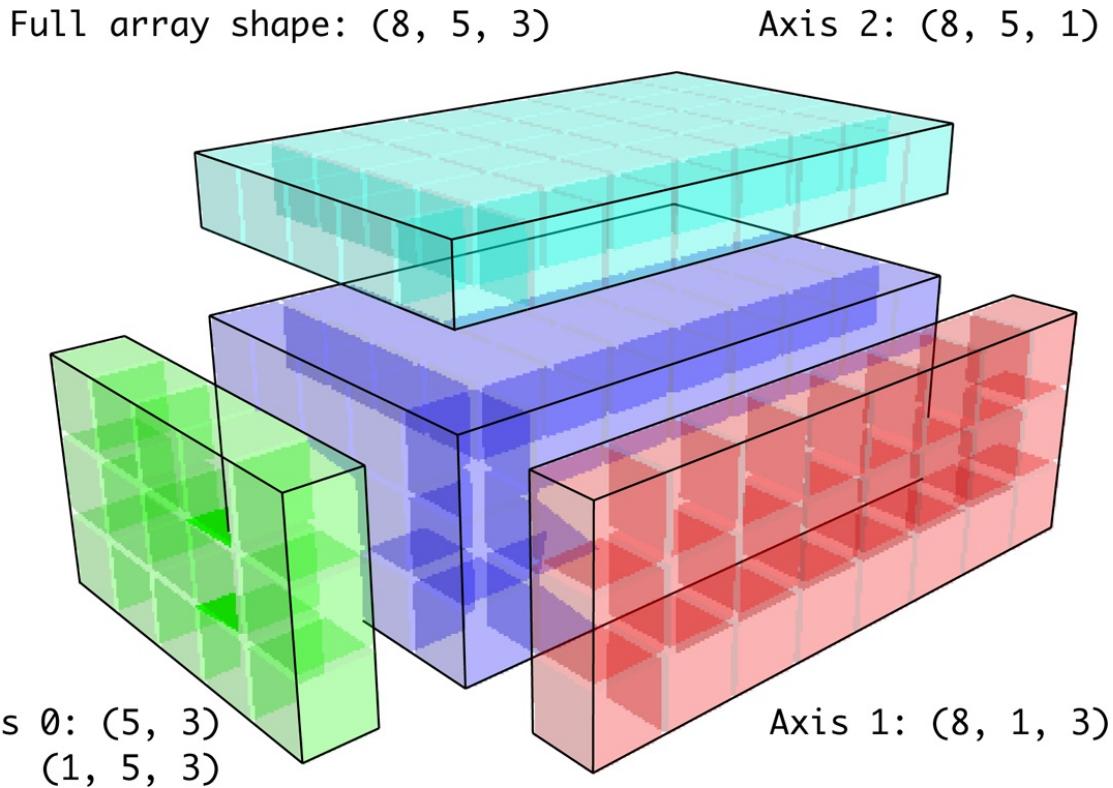
```
In [91]: arr - arr.mean(1).reshape((4, 1))
```

```
Out[91]:  
array([[-0.2095,  1.1334, -0.9239],  
       [ 0.8562, -0.6828, -0.1734],  
       [-0.3386,  1.0823, -0.7438],  
       [ 0.3234, -0.8599,  0.5365]])
```

In the three-dimensional case, broadcasting over any of the three dimensions is only a matter of reshaping the data to be shape-compatible. See [Figure 12-7](#) for a nice visualization of the shapes required to broadcast over each axis of a three-dimensional array.

A common problem, therefore, is needing to add a new axis with length 1 specifically for broadcasting purposes. Using `reshape` is one option, but inserting an axis requires constructing a tuple indicating the new shape. This can often be a tedious exercise. Thus, NumPy arrays offer a special syntax for inserting new axes by indexing. We use the special `np.newaxis` attribute along with “full” slices to insert the new axis:

```
In [92]: arr = np.zeros((4, 4))  
  
In [93]: arr_3d = arr[:, np.newaxis, :]  
  
In [94]: arr_3d.shape  
Out[94]: (4, 1, 4)  
  
In [95]: arr_1d = np.random.normal(size=3)  
  
In [96]: arr_1d[:, np.newaxis]  
Out[96]:  
array([-2.3594],  
      [-0.1995],  
      [-1.542 ])  
  
In [97]: arr_1d[np.newaxis, :]  
Out[97]: array([[-2.3594, -0.1995, -1.542 ]])
```



**Figure 12-7. Compatible 2D array shapes for broadcasting over a 3D array**

Thus, if we had a three-dimensional array and wanted to demean axis 2, say, we would need to write:

```
In [98]: arr = np.random.randn(3, 4, 5)

In [99]: depth_means = arr.mean(2)

In [100]: depth_means
Out[100]:
array([[-0.4735,  0.3971, -0.0228,  0.2001],
       [-0.3521, -0.281 , -0.071 , -0.1586],
       [ 0.6245,  0.6047,  0.4396, -0.2846]])

In [101]: demeaned = arr - depth_means[:, :, np.newaxis]

In [102]: demeaned.mean(2)
Out[102]:
array([[ 0.,  0., -0., -0.],
       [ 0.,  0., -0.,  0.],
       [ 0.,  0., -0., -0.]])
```

Some readers might wonder if there's a way to generalize demeaning over an axis without sacrificing performance. There is, but it requires some indexing gymnastics:

```
def demean_axis(arr, axis=0):
    means = arr.mean(axis)

    # This generalizes things like [:, :, np.newaxis] to N dimensions
    indexer = [slice(None)] * arr.ndim
    indexer[axis] = np.newaxis
    return arr - means[indexer]
```

# Setting Array Values by Broadcasting

The same broadcasting rule governing arithmetic operations also applies to setting values via array indexing. In a simple case, we can do things like:

```
In [103]: arr = np.zeros((4, 3))
```

```
In [104]: arr[:] = 5
```

```
In [105]: arr
Out[105]:
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

However, if we had a one-dimensional array of values we wanted to set into the columns of the array, we can do that as long as the shape is compatible:

```
In [106]: col = np.array([1.28, -0.42, 0.44, 1.6])
```

```
In [107]: arr[:] = col[:, np.newaxis]
```

```
In [108]: arr
Out[108]:
array([[ 1.28,  1.28,  1.28],
       [-0.42, -0.42, -0.42],
       [ 0.44,  0.44,  0.44],
       [ 1.6 ,  1.6 ,  1.6 ]])
```

```
In [109]: arr[:2] = [[-1.37], [0.509]]
```

```
In [110]: arr
Out[110]:
array([[-1.37, -1.37, -1.37],
       [ 0.509,  0.509,  0.509],
       [ 0.44 ,  0.44 ,  0.44 ],
       [ 1.6 ,  1.6 ,  1.6 ]])
```

# Advanced ufunc Usage

While many NumPy users will only make use of the fast element-wise operations provided by the universal functions, there are a number of additional features that occasionally can help you write more concise code without loops.

# ufunc Instance Methods

Each of NumPy's binary ufuncs has special methods for performing certain kinds of special vectorized operations. These are summarized in [Table 12-2](#), but I'll give a few concrete examples to illustrate how they work.

`reduce` takes a single array and aggregates its values, optionally along an axis, by performing a sequence of binary operations. For example, an alternate way to sum elements in an array is to use `np.add.reduce`:

```
In [111]: arr = np.arange(10)
```

```
In [112]: np.add.reduce(arr)
Out[112]: 45
```

```
In [113]: arr.sum()
Out[113]: 45
```

The starting value (0 for `add`) depends on the ufunc. If an axis is passed, the reduction is performed along that axis. This allows you to answer certain kinds of questions in a concise way. As a less trivial example, we can use `np.logical_and` to check whether the values in each row of an array are sorted:

```
In [114]: np.random.seed(12346) # for reproducibility
```

```
In [115]: arr = np.random.randn(5, 5)
```

```
In [116]: arr[::-2].sort(1) # sort a few rows
```

```
In [117]: arr[:, :-1] < arr[:, 1:]
Out[117]:
array([[ True,  True,  True,  True],
       [False,  True, False, False],
       [ True,  True,  True,  True],
       [ True, False,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
```

```
In [118]: np.logical_and.reduce(arr[:, :-1] < arr[:, 1:], axis=0)
Out[118]: array([ True, False,  True, False,  True], dtype=bool)
```

Note that `logical_and.reduce` is equivalent to the `all` method.

`accumulate` is related to `reduce` like `cumsum` is related to `sum`. It produces an array of the same size with the intermediate “accumulated” values:

```
In [119]: arr = np.arange(15).reshape((3, 5))
```

```
In [120]: np.add.accumulate(arr, axis=1)
Out[120]:
array([[ 0,  1,  3,  6, 10],
       [ 5, 11, 18, 26, 35],
       [10, 21, 33, 46, 60]])
```

`outer` performs a pairwise cross-product between two arrays:

```
In [121]: arr = np.arange(3).repeat([1, 2, 2])
```

```
In [122]: arr
Out[122]: array([0, 1, 1, 2, 2])
```

```
In [123]: np.multiply.outer(arr, np.arange(5))
Out[123]:
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  2,  4,  6,  8]])
```

The output of `outer` will have a dimension that is the sum of the dimensions of the inputs:

```
In [124]: x, y = np.random.randn(3, 4), np.random.randn(5)
```

```
In [125]: result = np.subtract.outer(x, y)
```

```
In [126]: result.shape
Out[126]: (3, 4, 5)
```

The last method, `reduceat`, performs a “local reduce”, in essence an array `groupby` operation in which slices of the array are aggregated together. While it’s less flexible than the `GroupBy` capabilities in pandas, it can be very fast and powerful in the right circumstances. It accepts a sequence of “bin edges” which indicate how to split and aggregate the values:

```
In [127]: arr = np.arange(10)

In [128]: np.add.reduceat(arr, [0, 5, 8])
Out[128]: array([10, 18, 17])
```

The results are the reductions (here, sums) performed over `arr[0:5]`, `arr[5:8]`, and `arr[8:]`. Like the other methods, you can pass an `axis` argument:

```
In [129]: arr = np.multiply.outer(np.arange(4), np.arange(5))

In [130]: arr
Out[130]:
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  3,  6,  9, 12]])

In [131]: np.add.reduceat(arr, [0, 2, 4], axis=1)
Out[131]:
array([[ 0,  0,  0],
       [ 1,  5,  4],
       [ 2, 10,  8],
       [ 3, 15, 12]])
```

Table 12-2. ufunc methods

| Method                         | Description   |
|--------------------------------|---|
| <code>reduce(x)</code>         | Aggregate values by successive applications of the operation  |
| <code>accumulate(x)</code>     | Aggregate values, preserving all partial aggregates   |
| <code>reduceat(x, bins)</code> | “Local” reduce or “group by”. Reduce contiguous slices of data to produce aggregated array.   |
| <code>outer(x, y)</code>       | Apply operation to all pairs of elements in <code>x</code> and <code>y</code> . Result array has <code>shape x.shape + y.shape</code> |

# Writing new ufuncs in Python

There are a number of facilities for creating your own NumPy ufuncs. The most general is to use the NumPy C API, but that is beyond the scope of this book. In this section, we will look at pure Python ufuncs.

`numpy.frompyfunc` accepts a Python function along with a specification for the number of inputs and outputs. For example, a simple function that adds element-wise would be specified as:

```
In [132]: def add_elements(x, y):
....:     return x + y

In [133]: add_them = np.frompyfunc(add_elements, 2, 1)

In [134]: add_them(np.arange(8), np.arange(8))
Out[134]: array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)
```

Functions created using `frompyfunc` always return arrays of Python objects which can be inconvenient. Fortunately, there is an alternate, but slightly less featureful function `numpy.vectorize` that allows you to specify the output type:

```
In [135]: add_them = np.vectorize(add_elements, otypes=[np.float])

In [136]: add_them(np.arange(8), np.arange(8))
Out[136]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14.])
```

These functions provide a way to create ufunc-like functions, but they are very slow because they require a Python function call to compute each element, which is a lot slower than NumPy's C-based ufunc loops:

```
In [137]: arr = np.random.randn(10000)

In [138]: %timeit add_them(arr, arr)
1.61 ms +- 24 us per loop (mean +- std. dev. of 7 runs, 1000 loops)

In [139]: %timeit np.add(arr, arr)
3.34 us +- 128 ns per loop (mean +- std. dev. of 7 runs, 10000 loops)
```

Later in this chapter we'll show how to create fast ufuncs in Python using the Numba project ([\(\*http://numba.pydata.org/\*\)](http://numba.pydata.org/)).

# Structured and Record Arrays

You may have noticed up until now that ndarray is a *homogeneous* data container; that is, it represents a block of memory in which each element takes up the same number of bytes, determined by the dtype. On the surface, this would appear to not allow you to represent heterogeneous or tabular-like data. A *structured* array is an ndarray in which each element can be thought of as representing a *struct* in C (hence the “structured” name) or a row in a SQL table with multiple named fields:

```
In [140]: dtype = [('x', np.float64), ('y', np.int32)]  
In [141]: sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=dtype)  
In [142]: sarr  
Out[142]:  
array([( 1.5, 6), ( 3.1416, -2)],  
      dtype=[('x', '<f8'), ('y', '<i4')])
```

There are several ways to specify a structured dtype (see the online NumPy documentation). One typical way is as a list of tuples with (field\_name, field\_data\_type). Now, the elements of the array are tuple-like objects whose elements can be accessed like a dictionary:

```
In [143]: sarr[0]  
Out[143]: ( 1.5, 6)  
  
In [144]: sarr[0]['y']  
Out[144]: 6
```

The field names are stored in the dtype.names attribute. On accessing a field on the structured array, a strided view on the data is returned thus copying nothing:

```
In [145]: sarr['x']  
Out[145]: array([ 1.5, 3.1416])
```

# Nested dtypes and Multidimensional Fields

When specifying a structured dtype, you can additionally pass a shape (as an int or tuple):

```
In [146]: dtype = [('x', np.int64, 3), ('y', np.int32)]  
In [147]: arr = np.zeros(4, dtype=dtype)  
  
In [148]: arr  
Out[148]:  
array([[0, 0, 0], 0), [[0, 0, 0], 0), [[0, 0, 0], 0), [[0, 0,  
           dtype=[('x', '<i8', (3,)), ('y', '<i4')])]
```

In this case, the `x` field now refers to an array of length three for each record:

```
In [149]: arr[0]['x']  
Out[149]: array([0, 0, 0])
```

Conveniently, accessing `arr['x']` then returns a two-dimensional array instead of a one-dimensional array as in prior examples:

```
In [150]: arr['x']  
Out[150]:  
array([[0, 0, 0],  
      [0, 0, 0],  
      [0, 0, 0],  
      [0, 0, 0]])
```

This enables you to express more complicated, nested structures as a single block of memory in an array. You can also nest dtypes to make more complex structures. Here is an example:

```
In [151]: dtype = [('x', [('a', 'f8'), ('b', 'f4')]), ('y', np.  
In [152]: data = np.array([(1, 2), 5), ((3, 4), 6)], dtype=dty  
  
In [153]: data['x']  
Out[153]:  
array([(1., 2.), (3., 4.)],  
      dtype=[('a', '<f8'), ('b', '<f4')])
```

```
In [154]: data['y']
Out[154]: array([5, 6], dtype=int32)
```

```
In [155]: data['x']['a']
Out[155]: array([ 1.,  3.])
```

Variable-shape fields and nested records are a flexible tool that can be very useful in some circumstances. A DataFrame from pandas, by contrast, does not support this feature directly, though it is similar to hierarchical indexing.

# Why Use Structured Arrays?

Compared with, say, a pandas DataFrame, NumPy structured arrays are a comparatively low-level tool. They provide a means to interpreting a block of memory as a tabular structure with arbitrarily complex nested columns. Since each element in the array is represented in memory as a fixed number of bytes, structured arrays provide a very fast and efficient way of writing data to and from disk (including memory maps, more on this later), transporting it over the network, and other such use.

As another common use for structured arrays, writing data files as fixed length record byte streams is a common way to serialize data in C and C++ code, which is commonly found in legacy systems in industry. As long as the format of the file is known (the size of each record and the order, byte size, and data type of each element), the data can be read into memory using `np.fromfile`. Specialized uses like this are beyond the scope of this book, but it's worth knowing that such things are possible.

# More About Sorting

Like Python's built-in list, the ndarray `sort` instance method is an *in-place* sort, meaning that the array contents are rearranged without producing a new array:

```
In [156]: arr = np.random.randn(6)

In [157]: arr.sort()

In [158]: arr
Out[158]: array([-1.082 ,  0.3759,  0.8014,  1.1397,  1.2888,
```

When sorting arrays in-place, remember that if the array is a view on a different ndarray, the original array will be modified:

```
In [159]: arr = np.random.randn(3, 5)

In [160]: arr
Out[160]:
array([[ -0.3318, -1.4711,  0.8705, -0.0847, -1.1329],
       [-1.0111, -0.3436,  2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424,  0.8548,  1.038 , -0.329 ]])

In [161]: arr[:, 0].sort()    # Sort first column values in-place

In [162]: arr
Out[162]:
array([[ -1.0111, -1.4711,  0.8705, -0.0847, -1.1329],
       [-0.3318, -0.3436,  2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424,  0.8548,  1.038 , -0.329 ]])
```

On the other hand, `numpy.sort` creates a new, sorted copy of an array. Otherwise it accepts the same arguments (such as `kind`, more on this below) as `ndarray.sort`:

```
In [163]: arr = np.random.randn(5)

In [164]: arr
Out[164]: array([-1.1181, -0.2415, -2.0051,  0.7379, -1.0614])
```

```
In [165]: np.sort(arr)
Out[165]: array([-2.0051, -1.1181, -1.0614, -0.2415,  0.7379])
```

```
In [166]: arr
Out[166]: array([-1.1181, -0.2415, -2.0051,  0.7379, -1.0614])
```

All of these sort methods take an axis argument for sorting the sections of data along the passed axis independently:

```
In [167]: arr = np.random.randn(3, 5)
```

```
In [168]: arr
Out[168]:
array([[ 0.5955, -0.2682,  1.3389, -0.1872,  0.9111],
       [-0.3215,  1.0054, -0.5168,  1.1925, -0.1989],
       [ 0.3969, -1.7638,  0.6071, -0.2222, -0.2171]])
```

```
In [169]: arr.sort(axis=1)
```

```
In [170]: arr
Out[170]:
array([[-0.2682, -0.1872,  0.5955,  0.9111,  1.3389],
       [-0.5168, -0.3215, -0.1989,  1.0054,  1.1925],
       [-1.7638, -0.2222, -0.2171,  0.3969,  0.6071]])
```

You may notice that none of the sort methods have an option to sort in descending order. This is a problem in practice because array slicing produces views, thus not producing a copy or requiring any computational work. Many Python users are familiar with the “trick” that for a list `values`, `values[::-1]` returns a list in reverse order. The same is true for ndarrays:

```
In [171]: arr[:, ::-1]
Out[171]:
array([[ 1.3389,  0.9111,  0.5955, -0.1872, -0.2682],
       [ 1.1925,  1.0054, -0.1989, -0.3215, -0.5168],
       [ 0.6071,  0.3969, -0.2171, -0.2222, -1.7638]])
```

# Indirect Sorts: argsort and lexsort

In data analysis one may need to reorder data sets by one or more keys. For example, a table of data about some students might need to be sorted by last name then by first name. This is an example of an *indirect* sort, and if you've read the pandas-related chapters you have already seen many higher-level examples. Given a key or keys (an array or values or multiple arrays of values), you wish to obtain an array of integer *indices* (I refer to them colloquially as *indexers*) that tells you how to reorder the data to be in sorted order. Two methods for this are `argsort` and `numpy.lexsort`. As an example:

```
In [172]: values = np.array([5, 0, 1, 3, 2])  
  
In [173]: indexer = values.argsort()  
  
In [174]: indexer  
Out[174]: array([1, 2, 4, 3, 0])  
  
In [175]: values[indexer]  
Out[175]: array([0, 1, 2, 3, 5])
```

As a more complicated example, this code reorders a 2D array by its first row:

```
In [176]: arr = np.random.randn(3, 5)  
  
In [177]: arr[0] = values  
  
In [178]: arr  
Out[178]:  
array([[ 5.        ,  0.        ,  1.        ,  3.        ,  2.        ],  
       [-0.3636, -0.1378,  2.1777, -0.4728,  0.8356],  
       [-0.2089,  0.2316,  0.728 , -1.3918,  1.9956]])  
  
In [179]: arr[:, arr[0].argsort()]  
Out[179]:  
array([[ 0.        ,  1.        ,  2.        ,  3.        ,  5.        ],  
       [-0.1378,  2.1777,  0.8356, -0.4728, -0.3636],  
       [ 0.2316,  0.728 ,  1.9956, -1.3918, -0.2089]])
```

`lexsort` is similar to `argsort`, but it performs an indirect *lexicographical* sort on multiple key arrays. Suppose we wanted to sort some data identified by first

and last names:

```
In [180]: first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill']
In [181]: last_name = np.array(['Jones', 'Arnold', 'Arnold', 'Jones'])
In [182]: sorter = np.lexsort((first_name, last_name))
In [183]: zip(last_name[sorter], first_name[sorter])
Out[183]: <zip at 0x7f107f0e4188>
```

`lexsort` can be a bit confusing the first time you use it because the order in which the keys are used to order the data starts with the *last* array passed. Here, `last_name` was used before `first_name`.

#### Note

pandas methods like Series's and DataFrame's `sort_values` method are implemented with variants of these functions (which also must take into account missing values)

# Alternate Sort Algorithms

A *stable* sorting algorithm preserves the relative position of equal elements. This can be especially important in indirect sorts where the relative ordering is meaningful:

```
In [184]: values = np.array(['2:first', '2:second', '1:first',  
In [185]: key = np.array([2, 2, 1, 1])  
In [186]: indexer = key.argsort(kind='mergesort')  
In [187]: indexer  
Out[187]: array([2, 3, 4, 0, 1])  
  
In [188]: values.take(indexer)  
Out[188]:  
array(['1:first', '1:second', '1:third', '2:first', '2:second']  
      dtype='<U8')
```

The only stable sort available is *mergesort* which has guaranteed  $O(n \log n)$  performance (for complexity buffs), but its performance is on average worse than the default quicksort method. See [Table 12-3](#) for a summary of available methods and their relative performance (and performance guarantees). This is not something that most users will ever have to think about but useful to know that it's there.

Table 12-3. Array sorting methods

| Kind        | Speed | Stable | Work space | Worst-case    |
|-------------|-------|--------|------------|---------------|
| 'quicksort' | 1     | No     | 0          | $O(n^2)$      |
| 'mergesort' | 2     | Yes    | $n / 2$    | $O(n \log n)$ |
| 'heapsort'  | 3     | No     | 0          | $O(n \log n)$ |

# Partially sorting arrays

One of the goals of sorting can be to determine the largest or smallest elements in an array. NumPy has optimized methods, `numpy.partition` and `np.argpartition` for partitioning an array around the  $k$ -th smallest element.

```
In [189]: np.random.seed(12345)

In [190]: arr = np.random.randn(20)

In [191]: arr
Out[191]:
array([-0.2047,  0.4789, -0.5194, -0.5557,  1.9658,  1.3934,  (
       0.2817,  0.769 ,  1.2464,  1.0072, -1.2962,  0.275 ,  (
       1.3529,  0.8864, -2.0016, -0.3718,  1.669 , -0.4386])

In [192]: np.partition(arr, 3)
Out[192]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386, -( 
       0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  0.275 ,  (
       1.3529,  0.8864,  1.3934,  1.9658,  1.669 ,  1.2464])
```

After calling `partition(arr, 3)`, the first 3 elements in the result are the smallest 3 values in no particular order. `numpy.argpartition`, similar to `numpy.argsort`, returns the indices that rearrange the data into the equivalent order:

```
In [193]: indices = np.argpartition(arr, 3)

In [194]: indices
Out[194]:
array([16, 11,  3,  2, 17, 19,  0,  7,  8,  1, 10,  6, 12, 13,
       4, 18,  9])

In [195]: arr.take(indices)
Out[195]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386, -( 
       0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  0.275 ,  (
       1.3529,  0.8864,  1.3934,  1.9658,  1.669 ,  1.2464)])
```

# **numpy.searchsorted: Finding elements in a Sorted Array**

`searchsorted` is an array method that performs a binary search on a sorted array, returning the location in the array where the value would need to be inserted to maintain sortedness:

```
In [196]: arr = np.array([0, 1, 7, 12, 15])
```

```
In [197]: arr.searchsorted(9)
Out[197]: 3
```

You can also pass an array of values to get an array of indices back:

```
In [198]: arr.searchsorted([0, 8, 11, 16])
Out[198]: array([0, 3, 3, 5])
```

You might have noticed that `searchsorted` returned 0 for the 0 element. This is because the default behavior is to return the index at the left side of a group of equal values:

```
In [199]: arr = np.array([0, 0, 0, 1, 1, 1, 1])
```

```
In [200]: arr.searchsorted([0, 1])
Out[200]: array([0, 3])
```

```
In [201]: arr.searchsorted([0, 1], side='right')
Out[201]: array([3, 7])
```

As another application of `searchsorted`, suppose we had an array of values between 0 and 10,000) and a separate array of “bucket edges” that we wanted to use to bin the data:

```
In [202]: data = np.floor(np.random.uniform(0, 10000, size=50))
```

```
In [203]: bins = np.array([0, 100, 1000, 5000, 10000])
```

```
In [204]: data
Out[204]:
array([ 9940.,   6768.,   7908.,   1709.,    268.,   8003.,   9037.,
```

```
4917., 5262., 5963., 519., 8950., 7282., 8183.,
8101., 959., 2189., 2587., 4681., 4593., 7095.,
5314., 1677., 7688., 9281., 6094., 1501., 4896.,
8486., 9110., 3838., 3154., 5683., 1878., 1258.,
7996., 5735., 9732., 6340., 8884., 4954., 3516.,
5039., 2256.])
```

To then get a labeling of which interval each data point belongs to (where 1 would mean the bucket [0, 100]), we can simply use `searchsorted`:

```
In [205]: labels = bins.searchsorted(data)
```

```
In [206]: labels
```

```
Out[206]:
```

```
array([4, 4, 4, 3, 2, 4, 4, 2, 3, 4, 4, 2, 4, 4, 4, 4, 4, 4, 2, 3,
       3, 4, 3, 4, 4, 4, 3, 3, 3, 4, 4, 3, 3, 3, 4, 3, 3, 3, 4, 4, 4,
       3, 4, 4, 3])
```

This, combined with pandas's `groupby`, can be used to bin data:

```
In [207]: pd.Series(data).groupby(labels).mean()
```

```
Out[207]:
```

```
2    498.000000
3    3064.277778
4    7389.035714
dtype: float64
```

# Writing Fast NumPy Functions with Numba

Numba (<http://numba.pydata.org/>) is an open source project that creates fast functions for NumPy-like data using CPUs, GPUs, or other hardware. It uses the LLVM Project (<http://llvm.org/>) to translate Python code into compiled machine code.

To introduce Numba, let's consider a pure Python function that computes the expression `(x - y).mean()` using a for loop:

```
import numpy as np

def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

This function is very slow:

```
In [209]: x = np.random.randn(10000000)
In [210]: y = np.random.randn(10000000)
In [211]: %timeit mean_distance(x, y)
1 loop, best of 3: 2 s per loop
In [212]: %timeit (x - y).mean()
100 loops, best of 3: 14.7 ms per loop
```

The NumPy version is over 100 times faster. We can turn this function into a compiled Numba function using the `numba.jit` function:

```
In [213]: import numba as nb
```

```
In [214]: numba_mean_distance = nb.jit(mean_distance)
```

We could also have written this as a decorator:

```
@nb.jit
def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

The resulting function is actually faster than the vectorized NumPy version:

```
In [215]: %timeit numba_mean_distance(x, y)
100 loops, best of 3: 10.3 ms per loop
```

Numba is a fairly deep library, supporting different kinds of hardware, modes of compilation, and user extensions. It is also able to compile a substantial subset of the NumPy Python API without explicit for-loops. Numba is able to recognize constructs that can be compiled to machine code, while substituting calls to the CPython API for functions that it does not know how to compile. Numba's `jit` function has an option `nopython=True` which restricts allowed code to Python code that can be compiled to LLVM without any Python C API calls. `jit(nopython=True)` has a shorter alias `numba.njit`.

In the above example, we could have written:

```
from numba import float64, njit

@njit(float64(float64[:, :], float64[:, :]))
def mean_distance(x, y):
    return (x - y).mean()
```

I encourage you to learn more by reading the online documentation for Numba at <http://numba.pydata.org/>. I will show some examples in this next section about creating custom NumPy ufunc objects.

# Creating Custom `numpy.ufunc` Objects with Numba

The `numba.vectorize` function creates compiled NumPy ufuncs, which behave like built-in ufuncs. Let's consider a Python implementation of `numpy.add`:

```
from numba import vectorize

@vectorize
def nb_add(x, y):
    return x + y
```

Now we have:

```
In [13]: x = np.arange(10)

In [14]: nb_add(x, x)
Out[14]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18.])

In [15]: nb_add.accumulate(x, 0)
Out[15]: array([ 0.,  1.,  3.,  6., 10., 15., 21., 28., 36., 45.])
```

# Advanced Array Input and Output

In [Chapter 4](#), I introduced you to `np.save` and `np.load` for storing arrays in binary format on disk. There are a number of additional options to consider for more sophisticated use. In particular, memory maps have the additional benefit of enabling you to work with data sets that do not fit into RAM.

# Memory-mapped Files

A *memory-mapped* file is a method for interacting with binary data on disk as though an in-memory array. NumPy implements a `memmap` object that is ndarray-like, enabling small segments of a large file to be read and written without reading the whole array into memory. Additionally, a `memmap` has the same methods as an in-memory array and thus can be substituted into many algorithms where an ndarray would be expected.

To create a new memory map, use the function `np.memmap` and pass a file path, `dtype`, `shape`, and `file mode`:

```
In [209]: mmap = np.memmap('mymmap', dtype='float64', mode='w+')

In [210]: mmap
Out[210]:
memmap([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

Slicing a `memmap` returns views on the data on disk:

```
In [211]: section = mmap[:5]
```

If you assign data to these, it will be buffered in memory (like a Python file object), but can be written to disk by calling `flush`:

```
In [212]: section[:] = np.random.randn(5, 10000)
```

```
In [213]: mmap.flush()
```

```
In [214]: mmap
Out[214]:
memmap([[ 0.7584, -0.6605,  0.8626, ...,  0.6046, -0.6212,  2.0
          [-1.2113, -1.0375,  0.7093, ..., -1.4117, -0.1719, -0.89
          [-0.1419, -0.3375,  0.4329, ...,  1.2914, -0.752 , -0.44
          ...,
```

```
[ 0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.  
[ 0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.  
[ 0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.
```

```
In [215]: del mmap
```

Whenever a memory map falls out of scope and is garbage-collected, any changes will be flushed to disk also. When *opening an existing memory map*, you still have to specify the dtype and shape as the file is only a block of binary data with no metadata on disk:

```
In [216]: mmap = np.memmap('my mmap', dtype='float64', shape=(100, 100))
```

```
In [217]: mmap
```

```
Out[217]:
```

```
memmap([[ 0.7584, -0.6605,  0.8626, ...,  0.6046, -0.6212,  2.0111,  
[-1.2113, -1.0375,  0.7093, ..., -1.4117, -0.1719, -0.8911,  
[-0.1419, -0.3375,  0.4329, ...,  1.2914, -0.752 , -0.4411,  
...,  
[ 0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.  
[ 0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.  
[ 0.      ,  0.      ,  0.      , ...,  0.      ,  0.      ,  0.
```

Memory maps also work with structured or nested dtypes as described in a previous section.

# HDF5 and Other Array Storage Options

PyTables and h5py are two Python projects providing NumPy-friendly interfaces for storing array data in the efficient and compressible HDF5 format (HDF stands for *hierarchical data format*). You can safely store hundreds of gigabytes or even terabytes of data in HDF5 format. To learn more about using HDF5 with Python, I would recommend reading the pandas online documentation.

# Performance Tips

Getting good performance out of code utilizing NumPy is often straightforward, as array operations typically replace otherwise comparatively extremely slow pure Python loops. Here is a brief list of some of the things to keep in mind:

- Convert Python loops and conditional logic to array operations and boolean array operations
- Use broadcasting whenever possible
- Avoid copying data using array views (slicing)
- Utilize ufuncs and ufunc methods

If you can't get the performance you require after exhausting the capabilities provided by NumPy alone, writing code in C, Fortran, or especially Cython (see a bit more on this below) may be in order. I personally use Cython (<http://cython.org>) heavily in my own work as an easy way to get C-like performance with minimal development.

# The Importance of Contiguous Memory

While the full extent of this topic is a bit outside the scope of this book, in some applications the memory layout of an array can significantly affect the speed of computations. This is based partly on performance differences having to do with the cache hierarchy of the CPU; operations accessing contiguous blocks of memory (for example, summing the rows of a C order array) will generally be the fastest because the memory subsystem will buffer the appropriate blocks of memory into the ultrafast L1 or L2 CPU cache. Also, certain code paths inside NumPy's C codebase have been optimized for the contiguous case in which generic strided memory access can be avoided.

To say that an array's memory layout is *contiguous* means that the elements are stored in memory in the order that they appear in the array with respect to Fortran (column major) or C (row major) ordering. By default, NumPy arrays are created as *C-contiguous* or just simply contiguous. A column major array, such as the transpose of a C-contiguous array, is thus said to be Fortran-contiguous. These properties can be explicitly checked via the flags attribute on the ndarray:

```
In [220]: arr_c = np.ones((1000, 1000), order='C')
```

```
In [221]: arr_f = np.ones((1000, 1000), order='F')
```

```
In [222]: arr_c.flags
```

```
Out[222]:
```

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

```
In [223]: arr_f.flags
```

```
Out[223]:
```

```
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
```

```
UPDATEIFCOPY : False  
In [224]: arr_f.flags.f_contiguous  
Out[224]: True
```

In this example, summing the rows of these arrays should, in theory, be faster for `arr_c` than `arr_f` since the rows are contiguous in memory. Here I check for sure using `%timeit` in IPython:

```
In [225]: %timeit arr_c.sum(1)  
340 us +- 25.5 us per loop (mean +- std. dev. of 7 runs, 1000 :  
  
In [226]: %timeit arr_f.sum(1)  
500 us +- 68.8 us per loop (mean +- std. dev. of 7 runs, 1000 :
```

When looking to squeeze more performance out of NumPy, this is often a place to invest some effort. If you have an array that does not have the desired memory order, you can use `copy` and pass either '`C`' or '`F`':

```
In [227]: arr_f.copy('C').flags  
Out[227]:  
C_CONTIGUOUS : True  
F_CONTIGUOUS : False  
OWNDATA : True  
WRITEABLE : True  
ALIGNED : True  
UPDATEIFCOPY : False
```

When constructing a view on an array, keep in mind that the result is not guaranteed to be contiguous:

```
In [228]: arr_c[:50].flags.contiguous  
Out[228]: True  
  
In [229]: arr_c[:, :50].flags  
Out[229]:  
C_CONTIGUOUS : False  
F_CONTIGUOUS : False  
OWNDATA : False  
WRITEABLE : True  
ALIGNED : True  
UPDATEIFCOPY : False
```

<sup>1</sup> Some of the dtypes have trailing underscores in their names. These are there

to avoid variable name conflicts between the NumPy-specific types and the Python built-in ones.

# **Chapter 13. Examples Data Sets**

This chapter contains a collection of miscellaneous example datasets that you can use for practice with the libraries and tools in this book.

# 1.usa.gov data from bit.ly

In 2011, URL shortening service bit.ly partnered with the United States government website `usa.gov` to provide a feed of anonymous data gathered from users who shorten links ending with `.gov` or `.mil`. In 2011, a live feed as well as hourly snapshots were available as downloadable text files. I've included a sample file in the accompanying materials for this book on GitHub.<sup>1</sup>

In the case of the hourly snapshots, each line in each file contains a common form of web data known as JSON, which stands for JavaScript Object Notation. For example, if we read just the first line of a file you may see something like

```
In [5]: path = 'datasets/bitly_usagov/example.txt'

In [6]: open(path).readline()
Out[6]: '{ "a": "Mozilla\\5.0 (Windows NT 6.1; WOW64) AppleWebKit\\535.11 (KHTML, like Gecko) Chrome\\17.0.963.78 Safari\\535.11", "c": "tz": "America\\New_York", "gr": "MA", "g": "A6qOVH", "h": "wzorofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r": "http:\\\\www.facebook.com\\1\\7AQEFzjSi\\1.usa.gov\\wfL", "http:\\\\www.ncbi.nlm.nih.gov\\pubmed\\22415991", "t": 131331822918, "cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }'
```

Python has both built-in and third party libraries for converting a JSON string into a Python dictionary object. Here I'll use the `json` module and its `loads` function invoked on each line in the sample file I downloaded:

```
import json
path = 'datasets/bitly_usagov/example.txt'
records = [json.loads(line) for line in open(path)]
```

The resulting object `records` is now a list of Python dicts:

```
In [18]: records[0]
Out[18]:
{'a': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 Chrome/17.0.963.78 Safari/535.11',
 'al': 'en-US,en;q=0.8',
```

```
'c': 'US',
'cy': 'Danvers',
'g': 'A6qOVH',
'gr': 'MA',
'h': 'wfLQtf',
'hc': 1331822918,
'hh': '1.usa.gov',
'l': 'orofrog',
'll': [42.576698, -70.954903],
'nk': 1,
'r': 'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/wfLQtf',
't': 1331923247,
'tz': 'America/New_York',
'u': 'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

Note that Python indices start at 0 and not 1 like some other languages (like R). You can now access individual values within records by passing a string for the key you wish to access:

# Counting Time Zones in Pure Python

Suppose we were interested in the most often-occurring time zones in the data set (the `tz` field). There are many ways we could do this. First, let's extract a list of time zones again using a list comprehension:

```
In [12]: time_zones = [rec['tz'] for rec in records]
-----
KeyError Traceback (most recent call last)
<ipython-input-12-db4fbd348da9> in <module>()
----> 1 time_zones = [rec['tz'] for rec in records]
<ipython-input-12-db4fbd348da9> in <listcomp>(.0)
----> 1 time_zones = [rec['tz'] for rec in records]
KeyError: 'tz'
```

Oops! Turns out that not all of the records have a time zone field. This is easy to handle as we can add the check `if 'tz' in rec` at the end of the list comprehension:

```
In [13]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]

In [14]: time_zones[:10]
Out[14]:
['America/New_York',
 'America/Denver',
 'America/New_York',
 'America/Sao_Paulo',
 'America/New_York',
 'America/New_York',
 'Europe/Warsaw',
 '',
 '',
 '']
```

Just looking at the first 10 time zones we see that some of them are unknown (empty string). You can filter these out also but I'll leave them in for now. Now, to produce counts by time zone I'll show two approaches: the harder way (using just the Python standard library) and the easier way (using pandas). One way to do the counting is to use a dict to store counts while we iterate through the time zones:

```
def get_counts(sequence):
    counts = {}
    for x in sequence:
        if x in counts:
            counts[x] += 1
        else:
            counts[x] = 1
    return counts
```

Using more advanced tools in the Python standard library, you can write the same thing more briefly:

```
from collections import defaultdict

def get_counts2(sequence):
    counts = defaultdict(int) # values will initialize to 0
    for x in sequence:
        counts[x] += 1
    return counts
```

I put this logic in a function just to make it more reusable. To use it on the time zones, just pass the `time_zones` list:

```
In [17]: counts = get_counts(time_zones)
```

```
In [18]: counts['America/New_York']
Out[18]: 1251
```

```
In [19]: len(time_zones)
Out[19]: 3440
```

If we wanted the top 10 time zones and their counts, we can do a bit of dictionary acrobatics:

```
def top_counts(count_dict, n=10):
    value_key_pairs = [(count, tz) for tz, count in count_dict.items()]
    value_key_pairs.sort()
    return value_key_pairs[-n:]
```

We have then:

```
In [21]: top_counts(counts)
Out[21]:
[(33, 'America/Sao_Paulo'),
 (35, 'Europe/Madrid'),
```

```
(36, 'Pacific/Honolulu'),  
(37, 'Asia/Tokyo'),  
(74, 'Europe/London'),  
(191, 'America/Denver'),  
(382, 'America/Los_Angeles'),  
(400, 'America/Chicago'),  
(521, ''),  
(1251, 'America/New_York')]
```

If you search the Python standard library, you may find the `collections.Counter` class, which makes this task a lot easier:

```
In [22]: from collections import Counter
```

```
In [23]: counts = Counter(time_zones)
```

```
In [24]: counts.most_common(10)
```

```
Out[24]:
```

```
[('America/New_York', 1251),  
 ('', 521),  
 ('America/Chicago', 400),  
 ('America/Los_Angeles', 382),  
 ('America/Denver', 191),  
 ('Europe/London', 74),  
 ('Asia/Tokyo', 37),  
 ('Pacific/Honolulu', 36),  
 ('Europe/Madrid', 35),  
 ('America/Sao_Paulo', 33)]
```

# Counting Time Zones with pandas

Creating a DataFrame from the original set of records is as easy as passing the list of records to `pandas.DataFrame`:

```
In [25]: import pandas as pd
```

```
In [26]: frame = pd.DataFrame(records)
```

```
In [27]: frame.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3560 entries, 0 to 3559
Data columns (total 18 columns):
_heartbeat_    120 non-null float64
a              3440 non-null object
al             3094 non-null object
c              2919 non-null object
cy             2919 non-null object
g              3440 non-null object
gr             2919 non-null object
h              3440 non-null object
hc             3440 non-null float64
hh             3440 non-null object
kw             93 non-null object
l              3440 non-null object
ll             2919 non-null object
nk             3440 non-null float64
r              3440 non-null object
t              3440 non-null float64
tz             3440 non-null object
u              3440 non-null object
dtypes: float64(4), object(14)
memory usage: 500.7+ KB
```

```
In [28]: frame['tz'][:10]
```

```
Out[28]:
```

```
0      America/New_York
1      America/Denver
2      America/New_York
3      America/Sao_Paulo
4      America/New_York
5      America/New_York
6      Europe/Warsaw
7
```

```
8  
9  
Name: tz, dtype: object
```

The output shown for the `frame` is the *summary view*, shown for large `DataFrame` objects. We can then use the `value_counts` method for `Series`:

```
In [29]: tz_counts = frame['tz'].value_counts()
```

```
In [30]: tz_counts[:10]  
Out[30]:  
America/New_York      1251  
                    521  
America/Chicago       400  
America/Los_Angeles  382  
America/Denver        191  
Europe/London         74  
Asia/Tokyo            37  
Pacific/Honolulu      36  
Europe/Madrid          35  
America/Sao_Paulo     33  
Name: tz, dtype: int64
```

We can visualize this data using `matplotlib`. You can do a bit of munging to fill in a substitute value for unknown and missing time zone data in the records. We replace the missing values with the `fillna` method and use boolean array indexing for the empty strings:

```
In [31]: clean_tz = frame['tz'].fillna('Missing')
```

```
In [32]: clean_tz[clean_tz == ''] = 'Unknown'
```

```
In [33]: tz_counts = clean_tz.value_counts()
```

```
In [34]: tz_counts[:10]  
Out[34]:  
America/New_York      1251  
Unknown                521  
America/Chicago        400  
America/Los_Angeles   382  
America/Denver          191  
Missing                 120  
Europe/London           74  
Asia/Tokyo              37  
Pacific/Honolulu         36
```

```
Europe/Madrid          35  
Name: tz, dtype: int64
```

I now use the Seaborn package (<http://seaborn.pydata.org/>) to make a horizontal bar plot:

```
In [36]: import seaborn as sns  
  
In [37]: subset = tz_counts[:10]  
  
In [38]: sns.barplot(y=subset.index, x=subset.values)
```

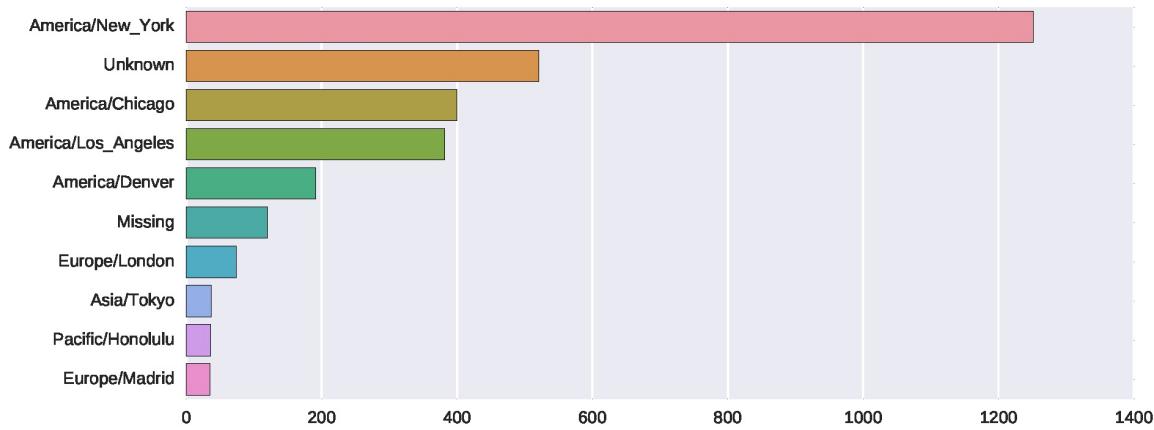


Figure 13-1. Top time zones in the 1.usa.gov sample data

See [Figure 13-1](#) for the resulting figure.

The `a` field contains information about the browser, device, or application used to perform the URL shortening:

```
In [39]: frame['a'][1]  
Out[39]: 'GoogleMaps/RochesterNY'  
  
In [40]: frame['a'][50]  
Out[40]: 'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101  
Firefox/10.0.2'  
  
In [41]: frame['a'][51][:50] # long line  
Out[41]: 'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P910)'
```

Parsing all of the interesting information in these “agent” strings may seem like a daunting task. One possible strategy is to split off the first token in the string

(corresponding roughly to the browser capability) and make another summary of the user behavior:

```
In [42]: results = pd.Series([x.split()[0] for x in frame.a.str])  
  
In [43]: results[:5]  
Out[43]:  
0           Mozilla/5.0  
1    GoogleMaps/RochesterNY  
2           Mozilla/4.0  
3           Mozilla/5.0  
4           Mozilla/5.0  
dtype: object  
  
In [44]: results.value_counts()[:8]  
Out[44]:  
Mozilla/5.0                2594  
Mozilla/4.0                  601  
GoogleMaps/RochesterNY        121  
Opera/9.80                     34  
TEST_INTERNET_AGENT                 24  
GoogleProducer                   21  
Mozilla/6.0                      5  
BlackBerry8520/5.0.0.681            4  
dtype: int64
```

Now, suppose you wanted to decompose the top time zones into Windows and non-Windows users. As a simplification, let's say that a user is on Windows if the string 'Windows' is in the agent string. Since some of the agents are missing, I'll exclude these from the data:

```
In [45]: cframe = frame[frame.a.notnull()]
```

We want to then compute a value whether each row is Windows or not:

```
In [47]: cframe['os'] = np.where(cframe['a'].str.contains('Windows', 'Not Windows'))  
.....
```

```
In [48]: cframe['os'][:5]  
Out[48]:  
0           Windows  
1    Not Windows  
2           Windows  
3    Not Windows  
4           Windows
```

```
Name: os, dtype: object
```

Then, you can group the data by its time zone column and this new list of operating systems:

```
In [49]: by_tz_os = cframe.groupby(['tz', 'os'])
```

The group counts, analogous to the `value_counts` function above, can be computed using `size`. This result is then reshaped into a table with `unstack`:

```
In [50]: agg_counts = by_tz_os.size().unstack().fillna(0)
```

```
In [51]: agg_counts[:10]
```

```
Out[51]:
```

|                                | os | Not Windows | Windows |
|--------------------------------|----|-------------|---------|
| tz                             |    | 245.0       | 276.0   |
| Africa/Cairo                   |    | 0.0         | 3.0     |
| Africa/Casablanca              |    | 0.0         | 1.0     |
| Africa/Ceuta                   |    | 0.0         | 2.0     |
| Africa/Johannesburg            |    | 0.0         | 1.0     |
| Africa/Lusaka                  |    | 0.0         | 1.0     |
| America/Anchorage              |    | 4.0         | 1.0     |
| America/Argentina/Buenos_Aires |    | 1.0         | 0.0     |
| America/Argentina/Cordoba      |    | 0.0         | 1.0     |
| America/Argentina/Mendoza      |    | 0.0         | 1.0     |

Finally, let's select the top overall time zones. To do so, I construct an indirect index array from the row counts in `agg_counts`:

```
# Use to sort in ascending order
```

```
In [52]: indexer = agg_counts.sum(1).argsort()
```

```
In [53]: indexer[:10]
```

```
Out[53]:
```

| tz                             |    |
|--------------------------------|----|
| Africa/Cairo                   | 24 |
| Africa/Casablanca              | 20 |
| Africa/Ceuta                   | 21 |
| Africa/Johannesburg            | 92 |
| Africa/Lusaka                  | 87 |
| America/Anchorage              | 53 |
| America/Argentina/Buenos_Aires | 54 |
| America/Argentina/Cordoba      | 57 |
| America/Argentina/Mendoza      | 26 |

```
America/Argentina/Mendoza          55  
dtype: int64
```

I then use `take` to select the rows in that order, then slice off the last 10 rows (largest values):

```
In [54]: count_subset = agg_counts.take(indexer[-10:])
```

```
In [55]: count_subset  
Out[55]:
```

| os                  | Not Windows | Windows |
|---------------------|-------------|---------|
| tz                  |             |         |
| America/Sao_Paulo   | 13.0        | 20.0    |
| Europe/Madrid       | 16.0        | 19.0    |
| Pacific/Honolulu    | 0.0         | 36.0    |
| Asia/Tokyo          | 2.0         | 35.0    |
| Europe/London       | 43.0        | 31.0    |
| America/Denver      | 132.0       | 59.0    |
| America/Los_Angeles | 130.0       | 252.0   |
| America/Chicago     | 115.0       | 285.0   |
|                     | 245.0       | 276.0   |
| America/New_York    | 339.0       | 912.0   |

pandas has a convenient `nlargest` which does the same thing:

```
In [56]: agg_counts.sum(1).nlargest(10)  
Out[56]:
```

| tz                  |        |
|---------------------|--------|
| America/New_York    | 1251.0 |
|                     | 521.0  |
| America/Chicago     | 400.0  |
| America/Los_Angeles | 382.0  |
| America/Denver      | 191.0  |
| Europe/London       | 74.0   |
| Asia/Tokyo          | 37.0   |
| Pacific/Honolulu    | 36.0   |
| Europe/Madrid       | 35.0   |
| America/Sao_Paulo   | 33.0   |

```
dtype: float64
```

Then, as shown in the preceding code block, this can be plotted in a bar plot; I'll make it a stacked bar plot by passing an additional argument to Seaborn's `barplot` function (see [Figure 13-2](#)):

```
# Rearrange the data for plotting
```

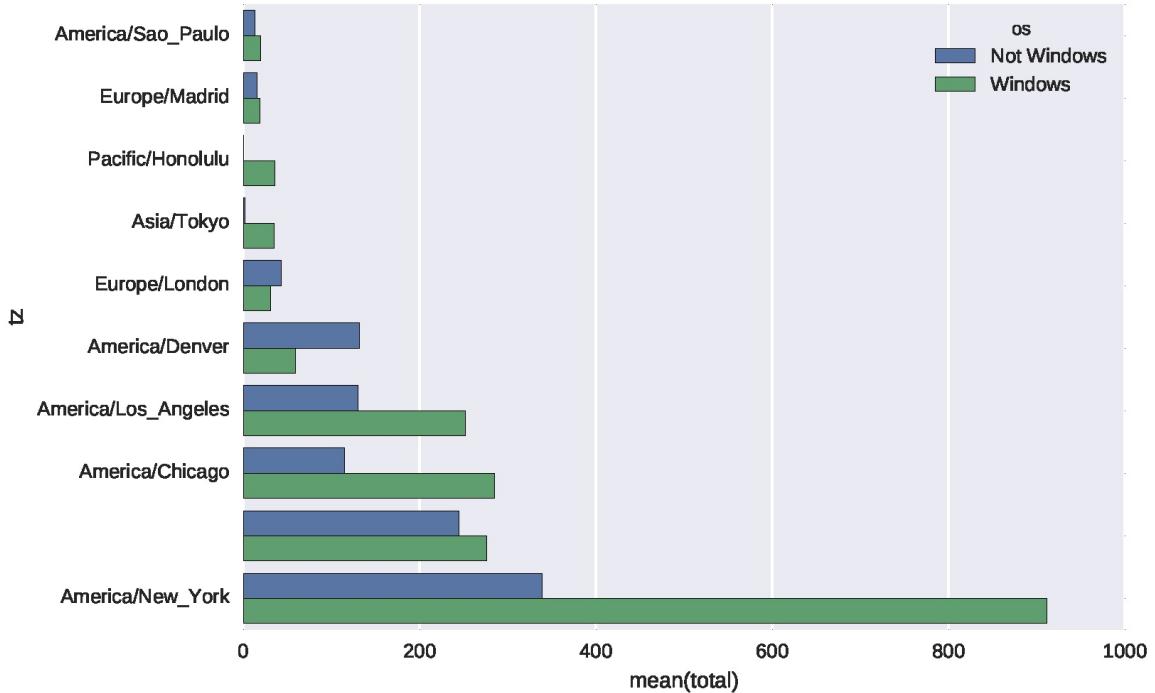
```
In [58]: count_subset = count_subset.stack()

In [59]: count_subset.name = 'total'

In [60]: count_subset = count_subset.reset_index()

In [61]: count_subset[:10]
Out[61]:
      tz          os  total
0  America/Sao_Paulo  Not Windows   13.0
1  America/Sao_Paulo       Windows   20.0
2    Europe/Madrid  Not Windows   16.0
3    Europe/Madrid       Windows   19.0
4  Pacific/Honolulu  Not Windows    0.0
5  Pacific/Honolulu       Windows   36.0
6    Asia/Tokyo  Not Windows    2.0
7    Asia/Tokyo       Windows   35.0
8  Europe/London  Not Windows   43.0
9  Europe/London       Windows   31.0

In [62]: sns.barplot(x='total', y='tz', hue='os', data=count_
```



**Figure 13-2. Top time zones by Windows and non-Windows users**

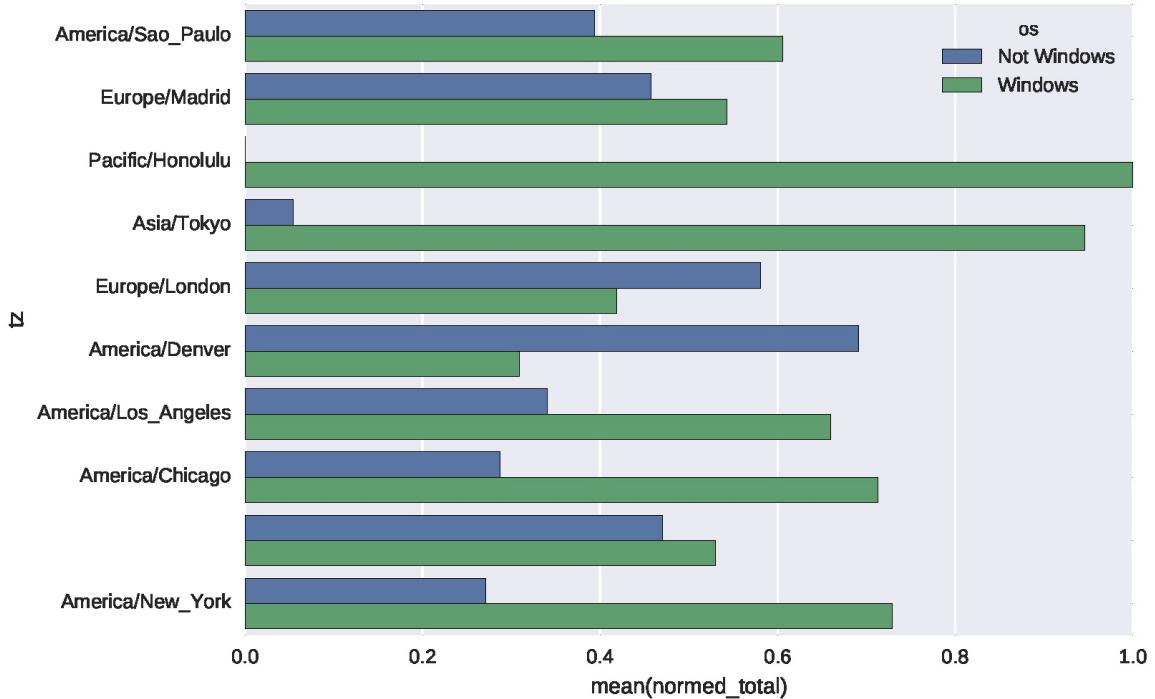
The plot doesn't make it easy to see the relative percentage of Windows users in the smaller groups, so let's normalize the group percentages to sum to 1:

```
def norm_total(group):
    group['normed_total'] = group.total / group.total.sum()
    return group

results = count_subset.groupby('tz').apply(norm_total)
```

Then plot this in [Figure 13-3](#):

```
In [65]: sns.barplot(x='normed_total', y='tz', hue='os', data=
```



**Figure 13-3. Percentage Windows and non-Windows users in top-occurring time zones**

The normalized sum could have been computed more efficiently using the `transform` method with `groupby`:

```
In [66]: g = count_subset.groupby('tz')
```

```
In [67]: results2 = count_subset.total / g.total.transform('sur
```

# MovieLens 1M Data Set

GroupLens Research (<http://www.grouplens.org/node/73>) provides a number of collections of movie ratings data collected from users of MovieLens in the late 1990s and early 2000s. The data provide movie ratings, movie metadata (genres and year), and demographic data about the users (age, zip code, gender, and occupation). Such data is often of interest in the development of recommendation systems based on machine learning algorithms. While I will not be exploring machine learning techniques in detail in this book, I will show you how to slice and dice data sets like these into the exact form you need.

The MovieLens 1M data set contains 1 million ratings collected from 6000 users on 4000 movies. It's spread across 3 tables: ratings, user information, and movie information. After extracting the data from the zip file, each table can be loaded into a pandas DataFrame object using `pandas.read_table`:

```
import pandas as pd

# Make display smaller
pd.options.display.max_rows = 10

unames = ['user_id', 'gender', 'age', 'occupation', 'zip']
users = pd.read_table('datasets/movielens/users.dat', sep='::',
                      names=unames)

rnames = ['user_id', 'movie_id', 'rating', 'timestamp']
ratings = pd.read_table('datasets/movielens/ratings.dat', sep='::',
                        names=rnames)

mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
                       names=mnames)
```

You can verify that everything succeeded by looking at the first few rows of each DataFrame with Python's slice syntax:

```
In [69]: users[:5]
Out[69]:
```

```

      user_id gender  age occupation      zip
0            1      F    1           10  48067
1            2      M   56           16  70072
2            3      M   25           15  55117
3            4      M   45            7  02460
4            5      M   25           20  55455

In [70]: ratings[:5]
Out[70]:
      user_id  movie_id  rating  timestamp
0            1       1193      5  978300760
1            1        661      3  978302109
2            1        914      3  978301968
3            1       3408      4  978300275
4            1       2355      5  978824291

In [71]: movies[:5]
Out[71]:
      movie_id                      title
0            1          Toy Story (1995)  Animation|Chil
1            2          Jumanji (1995)  Adventure|Chi
2            3  Grumpier Old Men (1995)
3            4      Waiting to Exhale (1995)
4            5  Father of the Bride Part II (1995)

In [72]: ratings
Out[72]:
      user_id  movie_id  rating  timestamp
0            1       1193      5  978300760
1            1        661      3  978302109
2            1        914      3  978301968
3            1       3408      4  978300275
4            1       2355      5  978824291
...
1000204     ...     ...     ...
1000205     6040     1091      1  956716541
1000205     6040     1094      5  956704887
1000206     6040      562      5  956704746
1000207     6040     1096      4  956715648
1000208     6040     1097      4  956715569
[1000209 rows x 4 columns]

```

Note that ages and occupations are coded as integers indicating groups described in the data set's README file. Analyzing the data spread across three tables is not a simple task; for example, suppose you wanted to compute mean ratings for a particular movie by sex and age. As you will see, this is much easier to do with all of the data merged together into a single table. Using

pandas's merge function, we first merge ratings with users then merging that result with the movies data. pandas infers which columns to use as the merge (or *join*) keys based on overlapping names:

```
In [73]: data = pd.merge(pd.merge(ratings, users), movies)
```

```
In [74]: data
```

```
Out[74]:
```

```
      user_id  movie_id  rating  timestamp  gender  age  occi
0            1       1193      5  978300760      F     1
1            2       1193      5  978298413      M    56
2           12       1193      4  978220179      M    25
3           15       1193      4  978199279      M    25
4           17       1193      5  978158471      M    50
...
1000204      ...      ...    ...      ...      ...
1000205      5949     2198      5  958846401      M    18
1000205      5675     2703      3  976029116      M    35
1000206      5780     2845      1  958153068      M    18
1000207      5851     3607      5  957756608      F    18
1000208      5938     2909      4  957273353      M    25
                                         title
0          One Flew Over the Cuckoo's Nest (1975)
1          One Flew Over the Cuckoo's Nest (1975)
2          One Flew Over the Cuckoo's Nest (1975)
3          One Flew Over the Cuckoo's Nest (1975)
4          One Flew Over the Cuckoo's Nest (1975)
...
1000204          Modulations (1998)
1000205          Broken Vessels (1998)
1000206          White Boys (1999)
1000207          One Little Indian (1973)  Comedy | D
1000208  Five Wives, Three Secretaries and Me (1998)
[1000209 rows x 10 columns]
```

```
In [75]: data.iloc[0]
```

```
Out[75]:
```

```
user_id                  1
movie_id                 1193
rating                   5
timestamp                978300760
gender                   F
age                      1
occupation               10
zip                      48067
title        One Flew Over the Cuckoo's Nest (1975)
genres                  Drama
Name: 0, dtype: object
```

In this form, aggregating the ratings grouped by one or more user or movie attributes is straightforward once you build some familiarity with pandas. To get mean movie ratings for each film grouped by gender, we can use the `pivot_table` method:

```
In [76]: mean_ratings = data.pivot_table('rating', index='title',
.....:                                         columns='gender', aggfunc='mean')

In [77]: mean_ratings[:5]
Out[77]:
   gender          F          M
   title
$1,000,000 Duck (1971)  3.375000  2.761905
'Night Mother (1986)      3.388889  3.352941
'Til There Was You (1997) 2.675676  2.733333
'burbs, The (1989)        2.793478  2.962085
...And Justice for All (1979) 3.828571  3.689024
```

This produced another DataFrame containing mean ratings with movie titles as row labels (the “index”) and gender as column labels. First, I’m going to filter down to movies that received at least 250 ratings (a completely arbitrary number); to do this, I group the data by title and use `size()` to get a Series of group sizes for each title:

```
In [78]: ratings_by_title = data.groupby('title').size()

In [79]: ratings_by_title[:10]
Out[79]:
   title
$1,000,000 Duck (1971)      37
'Night Mother (1986)         70
'Til There Was You (1997)    52
'burbs, The (1989)           303
...And Justice for All (1979) 199
1-900 (1994)                  2
10 Things I Hate About You (1999) 700
101 Dalmatians (1961)        565
101 Dalmatians (1996)        364
12 Angry Men (1957)           616
dtype: int64

In [80]: active_titles = ratings_by_title.index[ratings_by_title >= 250]

In [81]: active_titles
Out[81]:
```

```

Index([''burbs, The (1989)'', '10 Things I Hate About You (1999)'',
       '101 Dalmatians (1961)'', '101 Dalmatians (1996)'', '12 Angry Men (1957)'',
       '13th Warrior, The (1999)'', '2 Days in the Valley (1996)'',
       '20,000 Leagues Under the Sea (1954)'', '2001: A Space Odyssey (1997)'',
       '2010 (1984)'',
       ...
       'X-Men (2000)'', 'Year of Living Dangerously (1982)'',
       'Yellow Submarine (1968)'', 'You've Got Mail (1998)'',
       'Young Frankenstein (1974)'', 'Young Guns (1988)'',
       'Young Guns II (1990)'', 'Young Sherlock Holmes (1985)'',
       'Zero Effect (1998)'', 'eXistenZ (1999)''],
      dtype='object', name='title', length=1216)

```

The index of titles receiving at least 250 ratings can then be used to select rows from `mean_ratings` above:

```
# Select rows on the index
In [82]: mean_ratings = mean_ratings.loc[active_titles]
```

```
In [83]: mean_ratings
```

```
Out[83]:
```

|                                   | F        | M        |
|-----------------------------------|----------|----------|
| gender                            |          |          |
| title                             |          |          |
| 'burbs, The (1989)                | 2.793478 | 2.962085 |
| 10 Things I Hate About You (1999) | 3.646552 | 3.311966 |
| 101 Dalmatians (1961)             | 3.791444 | 3.500000 |
| 101 Dalmatians (1996)             | 3.240000 | 2.911215 |
| 12 Angry Men (1957)               | 4.184397 | 4.328421 |
| ...                               | ...      | ...      |
| Young Guns (1988)                 | 3.371795 | 3.425620 |
| Young Guns II (1990)              | 2.934783 | 2.904025 |
| Young Sherlock Holmes (1985)      | 3.514706 | 3.363344 |
| Zero Effect (1998)                | 3.864407 | 3.723140 |
| eXistenZ (1999)                   | 3.098592 | 3.289086 |
| [1216 rows x 2 columns]           |          |          |

To see the top films among female viewers, we can sort by the F column in descending order:

```
In [85]: top_female_ratings = mean_ratings.sort_values(by='F', ascending=False)
```

```
In [86]: top_female_ratings[:10]
```

```
Out[86]:
```

|                       | F        |
|-----------------------|----------|
| gender                |          |
| title                 |          |
| Close Shave, A (1995) | 4.644444 |

|  |          |   |
|--|----------|---|
| Wrong Trousers, The (1993)                         | 4.588235 | ' |
| Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)      | 4.572650 | ' |
| Wallace & Gromit: The Best of Aardman Animation... | 4.563107 | ' |
| Schindler's List (1993)                            | 4.562602 | ' |
| Shawshank Redemption, The (1994)                   | 4.539075 | ' |
| Grand Day Out, A (1992)                            | 4.537879 | ' |
| To Kill a Mockingbird (1962)                       | 4.536667 | ' |
| Creature Comforts (1990)                           | 4.513889 | ' |
| Usual Suspects, The (1995)                         | 4.513317 | ' |

# Measuring rating disagreement

Suppose you wanted to find the movies that are most divisive between male and female viewers. One way is to add a column to `mean_ratings` containing the difference in means, then sort by that:

```
In [87]: mean_ratings['diff'] = mean_ratings['M'] - mean_ratings['F']
```

Sorting by `'diff'` gives us the movies with the greatest rating difference and which were preferred by women:

```
In [88]: sorted_by_diff = mean_ratings.sort_values(by='diff')
```

```
In [89]: sorted_by_diff[:10]
```

```
Out[89]:
```

| gender                                | F        | M              |
|---------------------------------------|----------|----------------|
| title                                 |          |                |
| Dirty Dancing (1987)                  | 3.790378 | 2.959596 -0.83 |
| Jumpin' Jack Flash (1986)             | 3.254717 | 2.578358 -0.67 |
| Grease (1978)                         | 3.975265 | 3.367041 -0.61 |
| Little Women (1994)                   | 3.870588 | 3.321739 -0.54 |
| Steel Magnolias (1989)                | 3.901734 | 3.365957 -0.51 |
| Anastasia (1997)                      | 3.800000 | 3.281609 -0.51 |
| Rocky Horror Picture Show, The (1975) | 3.673016 | 3.160131 -0.51 |
| Color Purple, The (1985)              | 4.158192 | 3.659341 -0.49 |
| Age of Innocence, The (1993)          | 3.827068 | 3.339506 -0.48 |
| Free Willy (1993)                     | 2.921348 | 2.438776 -0.48 |

Reversing the order of the rows and again slicing off the top 10 rows, we get the movies preferred by men that women didn't rate as highly:

```
# Reverse order of rows, take first 10 rows
```

```
In [90]: sorted_by_diff[::-1][:10]
```

```
Out[90]:
```

| gender                                 | F        | M             |
|--|----------|---------------|
| title                                  |          |               |
| Good, The Bad and The Ugly, The (1966) | 3.494949 | 4.221300 0.71 |
| Kentucky Fried Movie, The (1977)       | 2.878788 | 3.555147 0.68 |
| Dumb & Dumber (1994)                   | 2.697987 | 3.336595 0.66 |
| Longest Day, The (1962)                | 3.411765 | 4.031447 0.66 |
| Cable Guy, The (1996)                  | 2.250000 | 2.863787 0.66 |
| Evil Dead II (Dead By Dawn) (1987)     | 3.297297 | 3.909283 0.66 |

|                               |          |          |     |
|-------------------------------|----------|----------|-----|
| Hidden, The (1987)            | 3.137931 | 3.745098 | 0.6 |
| Rocky III (1982)              | 2.361702 | 2.943503 | 0.5 |
| Caddyshack (1980)             | 3.396135 | 3.969737 | 0.5 |
| For a Few Dollars More (1965) | 3.409091 | 3.953795 | 0.5 |

Suppose instead you wanted the movies that elicited the most disagreement among viewers, independent of gender identification. Disagreement can be measured by the variance or standard deviation of the ratings:

```
# Standard deviation of rating grouped by title
In [91]: rating_std_by_title = data.groupby('title')['rating']

# Filter down to active_titles
In [92]: rating_std_by_title = rating_std_by_title.loc[active_t]

# Order Series by value in descending order
In [93]: rating_std_by_title.sort_values(ascending=False)[:10]
Out[93]:
title
Dumb & Dumber (1994)           1.321333
Blair Witch Project, The (1999) 1.316368
Natural Born Killers (1994)     1.307198
Tank Girl (1995)                1.277695
Rocky Horror Picture Show, The (1975) 1.260177
Eyes Wide Shut (1999)            1.259624
Evita (1996)                   1.253631
Billy Madison (1995)            1.249970
Fear and Loathing in Las Vegas (1998) 1.246408
Bicentennial Man (1999)          1.245533
Name: rating, dtype: float64
```

You may have noticed that movie genres are given as a pipe-separated (|) string. If you wanted to do some analysis by genre, more work would be required to transform the genre information into a more usable form.

# US Baby Names 1880-2010

The United States Social Security Administration (SSA) has made available data on the frequency of baby names from 1880 through the present. Hadley Wickham, an author of several popular R packages, has often made use of this data set in illustrating data manipulation in R.

```
In [4]: names.head(10)
Out[4]:
      name sex  births   year
0     Mary   F    7065 1880
1     Anna   F    2604 1880
2     Emma   F    2003 1880
3 Elizabeth   F    1939 1880
4    Minnie   F    1746 1880
5 Margaret   F    1578 1880
6      Ida   F    1472 1880
7     Alice   F    1414 1880
8    Bertha   F    1320 1880
9     Sarah   F    1288 1880
```

There are many things you might want to do with the data set:

- Visualize the proportion of babies given a particular name (your own, or another name) over time.
- Determine the relative rank of a name.
- Determine the most popular names in each year or the names with largest increases or decreases.
- Analyze trends in names: vowels, consonants, length, overall diversity, changes in spelling, first and last letters
- Analyze external sources of trends: biblical names, celebrities, demographic changes

Using the tools in this book at so far, many of these kinds of analyses are within reach, so I will walk you through some of them.

As of this writing, the US Social Security Administration makes available data files, one per year, containing the total number of births for each sex/name combination. The raw archive of these files can be obtained here:

```
http://www.ssa.gov/oact/babynames/limits.html
```

In the event that this page has been moved by the time you're reading this, it can most likely be located again by Internet search. After downloading the "National data" file `names.zip` and unzipping it, you will have a directory containing a series of files like `yob1880.txt`. I use the `UNIX head` command to look at the first 10 lines of one of the files (on Windows, you can use the `more` command or open it in a text editor):

```
In [94]: !head -n 10 datasets/babynames/yob1880.txt
Mary,F,7065
Anna,F,2604
Emma,F,2003
Elizabeth,F,1939
Minnie,F,1746
Margaret,F,1578
Ida,F,1472
Alice,F,1414
Bertha,F,1320
Sarah,F,1288
```

As this is a nicely comma-separated form, it can be loaded into a DataFrame with `pandas.read_csv`:

```
In [95]: import pandas as pd
In [96]: names1880 = pd.read_csv('datasets/babynames/yob1880.txt',
...:                           names=['name', 'sex', 'births']
In [97]: names1880
Out[97]:
      name  sex  births
0       Mary    F     7065
1        Anna    F     2604
2        Emma    F     2003
3   Elizabeth    F     1939
4       Minnie    F     1746
...
1995     Woodie    M       5
1996    Worthy    M       5
```

```
1997      Wright    M      5
1998      York     M      5
1999  Zachariah   M      5
[2000 rows x 3 columns]
```

These files only contain names with at least 5 occurrences in each year, so for simplicity's sake we can use the sum of the births column by sex as the total number of births in that year:

```
In [98]: names1880.groupby('sex').births.sum()
Out[98]:
sex
F      90993
M     110493
Name: births, dtype: int64
```

Since the data set is split into files by year, one of the first things to do is to assemble all of the data into a single DataFrame and further to add a `year` field. You can do this using `pandas.concat`:

```
years = range(1880, 2011)

pieces = []
columns = ['name', 'sex', 'births']

for year in years:
    path = 'datasets/babynames/yob%d.txt' % year
    frame = pd.read_csv(path, names=columns)

    frame['year'] = year
    pieces.append(frame)

# Concatenate everything into a single DataFrame
names = pd.concat(pieces, ignore_index=True)
```

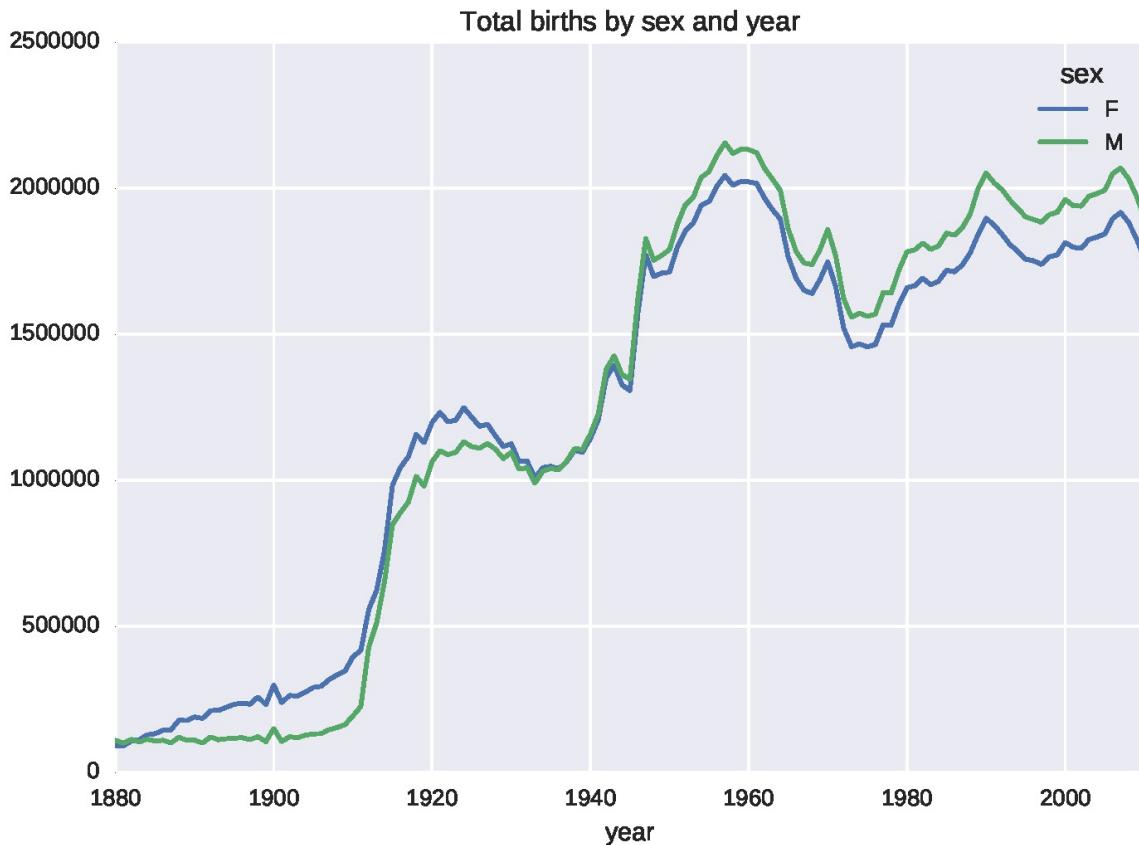
There are a couple things to note here. First, remember that `concat` glues the DataFrame objects together row-wise by default. Secondly, you have to pass `ignore_index=True` because we're not interested in preserving the original row numbers returned from `read_csv` (this will be explained in more detail in [Chapter 8](#)). So we now have a very large DataFrame containing all of the names data:

Now the `names` DataFrame looks like:

```
In [100]: names
Out[100]:
      name  sex  births  year
0       Mary    F     7065  1880
1       Anna    F     2604  1880
2       Emma    F     2003  1880
3  Elizabeth    F     1939  1880
4     Minnie    F     1746  1880
...
1690779   Zymaire    M      5  2010
1690780     Zyonne    M      5  2010
1690781   Zyquarius    M      5  2010
1690782      Zyran    M      5  2010
1690783     Zzyzx    M      5  2010
[1690784 rows x 4 columns]
```

With this data in hand, we can already start aggregating the data at the year and sex level using `groupby` or `pivot_table`, see [Figure 13-4](#):

```
In [101]: total_births = names.pivot_table('births', index='year',
                                             columns='sex', aggfunc='sum')
.....
In [102]: total_births.tail()
Out[102]:
sex          F          M
year
2006  1896468  2050234
2007  1916888  2069242
2008  1883645  2032310
2009  1827643  1973359
2010  1759010  1898382
In [103]: total_births.plot(title='Total births by sex and year')
```



**Figure 13-4.** Total births by sex and year

Next, let's insert a column `prop` with the fraction of babies given each name relative to the total number of births. A `prop` value of 0.02 would indicate that 2 out of every 100 babies was given a particular name. Thus, we group the data by year and sex, then add the new column to each group:

```
def add_prop(group):
    group['prop'] = group.births / group.births.sum()
    return group
names = names.groupby(['year', 'sex']).apply(add_prop)
```

The resulting complete data set now has the following columns:

```
In [105]: names
Out[105]:
   name  sex  births  year      prop
0   Mary    F     7065  1880  0.077643
1   Anna    F     2604  1880  0.028618
2   Emma    F     2003  1880  0.022013
```

```

3      Elizabeth    F    1939   1880   0.021309
4      Minnie      F    1746   1880   0.019188
...
1690779     Zymaire    M      5   2010   0.000003
1690780     Zyonne     M      5   2010   0.000003
1690781     Zyquarius   M      5   2010   0.000003
1690782     Zyran      M      5   2010   0.000003
1690783     Zzyzx      M      5   2010   0.000003
[1690784 rows x 5 columns]

```

When performing a group operation like this, it's often valuable to do a sanity check, like verifying that the `prop` column sums to 1 within all the groups:

```

In [106]: names.groupby(['year', 'sex']).prop.sum()
Out[106]:
year   sex
1880   F      1.0
        M      1.0
1881   F      1.0
        M      1.0
1882   F      1.0
        ...
2008   M      1.0
2009   F      1.0
        M      1.0
2010   F      1.0
        M      1.0
Name: prop, Length: 262, dtype: float64

```

Now that this is done, I'm going to extract a subset of the data to facilitate further analysis: the top 1000 names for each sex/year combination. This is yet another group operation:

```

def get_top1000(group):
    return group.sort_values(by='births', ascending=False)[:1000]
grouped = names.groupby(['year', 'sex'])
top1000 = grouped.apply(get_top1000)
# Drop the group index, not needed
top1000.reset_index(inplace=True, drop=True)

```

If you prefer a do-it-yourself approach, you could also do:

```

pieces = []
for year, group in names.groupby(['year', 'sex']):
    pieces.append(group.sort_values(by='births', ascending=False))

```

```
top1000 = pd.concat(pieces, ignore_index=True)
```

The resulting data set is now quite a bit smaller:

```
In [108]: top1000
```

```
Out[108]:
```

```
      name  sex  births  year      prop
0       Mary    F     7065  1880  0.077643
1        Anna    F     2604  1880  0.028618
2       Emma    F     2003  1880  0.022013
3   Elizabeth    F     1939  1880  0.021309
4      Minnie    F     1746  1880  0.019188
...
261872    Camilo    M      194  2010  0.000102
261873    Destin    M      194  2010  0.000102
261874    Jaquan    M      194  2010  0.000102
261875    Jaydan    M      194  2010  0.000102
261876    Maxton    M      193  2010  0.000102
[261877 rows x 5 columns]
```

We'll use this Top 1,000 data set in the following investigations into the data.

# Analyzing Naming Trends

With the full data set and Top 1,000 data set in hand, we can start analyzing various naming trends of interest. Splitting the Top 1,000 names into the boy and girl portions is easy to do first:

```
In [109]: boys = top1000[top1000.sex == 'M']
```

```
In [110]: girls = top1000[top1000.sex == 'F']
```

Simple time series, like the number of Johns or Marys for each year can be plotted but require a bit of munging to be a bit more useful. Let's form a pivot table of the total number of births by year and name:

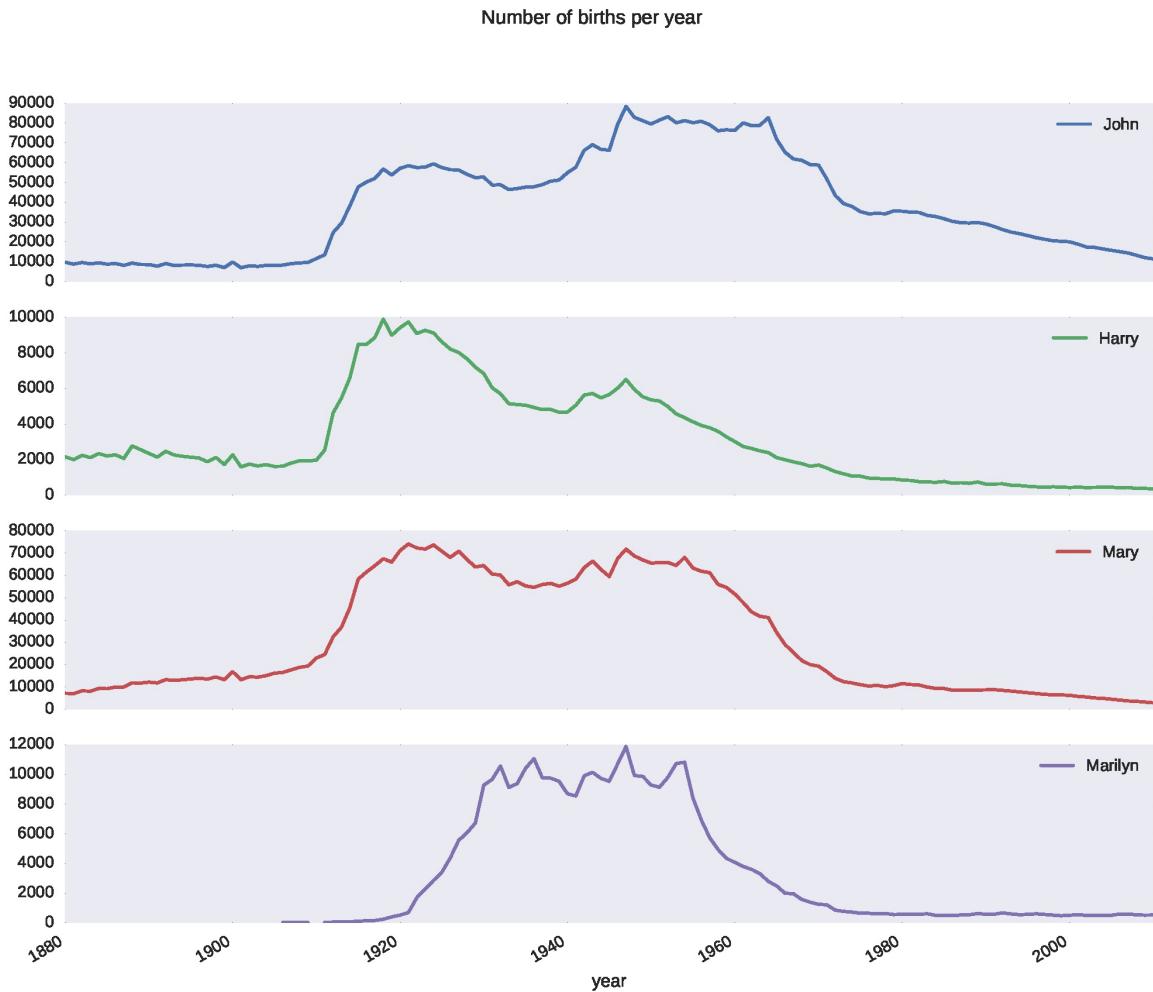
```
In [111]: total_births = top1000.pivot_table('births', index='name', aggfunc='sum')
```

Now, this can be plotted for a handful of names using DataFrame's `plot` method:

```
In [112]: total_births.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 131 entries, 1880 to 2010
Columns: 6868 entries, Aaden to Zuri
dtypes: float64(6868)
memory usage: 6.9 MB
```

```
In [113]: subset = total_births[['John', 'Harry', 'Mary', 'Mark', 'Sarah']]
```

```
In [114]: subset.plot(subplots=True, figsize=(12, 10), grid=False, title="Number of births per year")
```



**Figure 13-5. A few boy and girl names over time**

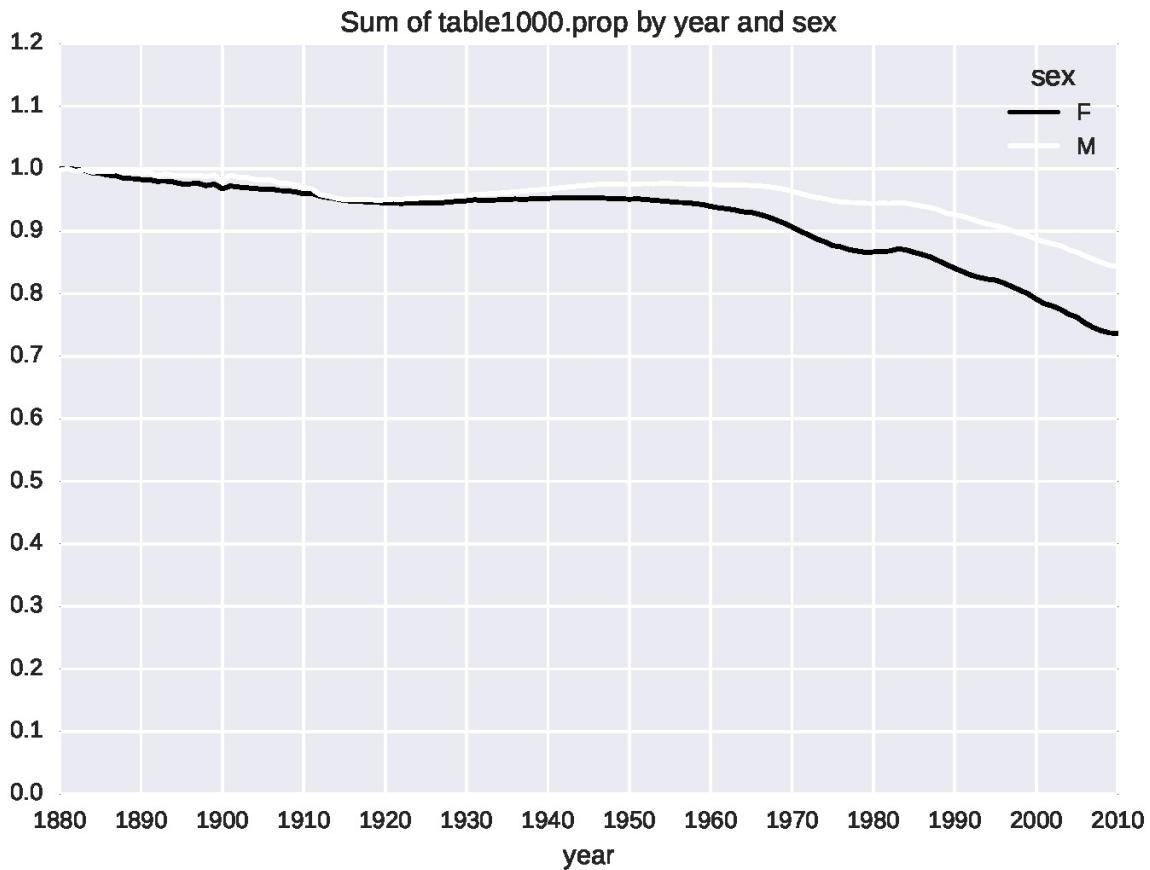
See [Figure 13-5](#) for the result. On looking at this, you might conclude that these names have grown out of favor with the American population. But the story is actually more complicated than that, as will be explored in the next section.

## Measuring the increase in naming diversity

One explanation for the decrease in plots above is that fewer parents are choosing common names for their children. This hypothesis can be explored and confirmed in the data. One measure is the proportion of births represented by the top 1000 most popular names, which I aggregate and plot by year and sex:

```
In [116]: table = top1000.pivot_table('prop', index='year',
.....:                                     columns='sex', aggfunc=si
```

```
In [117]: table.plot(title='Sum of table1000.prop by year and s
.....:             yticks=np.linspace(0, 1.2, 13), xticks=rar
.....:             colormap='gray')
```



**Figure 13-6. Proportion of births represented in top 1000 names by sex**

See [Figure 13-6](#) for this plot. So you can see that, indeed, there appears to be increasing name diversity (decreasing total proportion in the top 1,000). Another interesting metric is the number of distinct names, taken in order of popularity from highest to lowest, in the top 50% of births. This number is a bit more tricky to compute. Let's consider just the boy names from 2010:

```
In [118]: df = boys[boys.year == 2010]
```

```
In [119]: df
```

```
Out[119]:
```

|        |  | name    | sex | births | year | prop     |
|--------|--|---------|-----|--------|------|----------|
| 260877 |  | Jacob   | M   | 21875  | 2010 | 0.011523 |
| 260878 |  | Ethan   | M   | 17866  | 2010 | 0.009411 |
| 260879 |  | Michael | M   | 17133  | 2010 | 0.009025 |
| 260880 |  | Jayden  | M   | 17030  | 2010 | 0.008971 |
| 260881 |  | William | M   | 16870  | 2010 | 0.008887 |
| ...    |  | ...     | ... | ...    | ...  | ...      |

```
261872    Camilo    M      194  2010  0.000102
261873    Destin    M      194  2010  0.000102
261874    Jaquan    M      194  2010  0.000102
261875    Jaydan    M      194  2010  0.000102
261876    Maxton    M      193  2010  0.000102
[1000 rows x 5 columns]
```

After sorting `prop` in descending order, we want to know how many of the most popular names it takes to reach 50%. You could write a `for` loop to do this, but a vectorized NumPy way is a bit more clever. Taking the cumulative sum, `cumsum`, of `prop` then calling the method `searchsorted` returns the position in the cumulative sum at which 0.5 would need to be inserted to keep it in sorted order:

```
In [120]: prop_cumsum = df.sort_values(by='prop', ascending=False)

In [121]: prop_cumsum[:10]
Out[121]:
260877    0.011523
260878    0.020934
260879    0.029959
260880    0.038930
260881    0.047817
260882    0.056579
260883    0.065155
260884    0.073414
260885    0.081528
260886    0.089621
Name: prop, dtype: float64

In [122]: prop_cumsum.values.searchsorted(0.5)
Out[122]: 116
```

Since arrays are zero-indexed, adding 1 to this result gives you a result of 117. By contrast, in 1900 this number was much smaller:

```
In [123]: df = boys[boys.year == 1900]

In [124]: in1900 = df.sort_values(by='prop', ascending=False).values

In [125]: in1900.searchsorted(0.5) + 1
Out[125]: 25
```

You can now apply this operation to each year/sex combination; `groupby` those

fields and apply a function returning the count for each group:

```
def get_quantile_count(group, q=0.5):
    group = group.sort_values(by='prop', ascending=False)
    return group.prop.cumsum().values.searchsorted(q) + 1

diversity = top1000.groupby(['year', 'sex']).apply(get_quantile_count)
diversity = diversity.unstack('sex')
```

This resulting DataFrame diversity now has two time series, one for each sex, indexed by year. This can be inspected in IPython and plotted as before (see [Figure 13-7](#)):

```
In [128]: diversity.head()
Out[128]:
   sex      F      M
  year
1880  38    14
1881  38    14
1882  38    15
1883  39    15
1884  39    16

In [129]: diversity.plot(title="Number of popular names in top
```

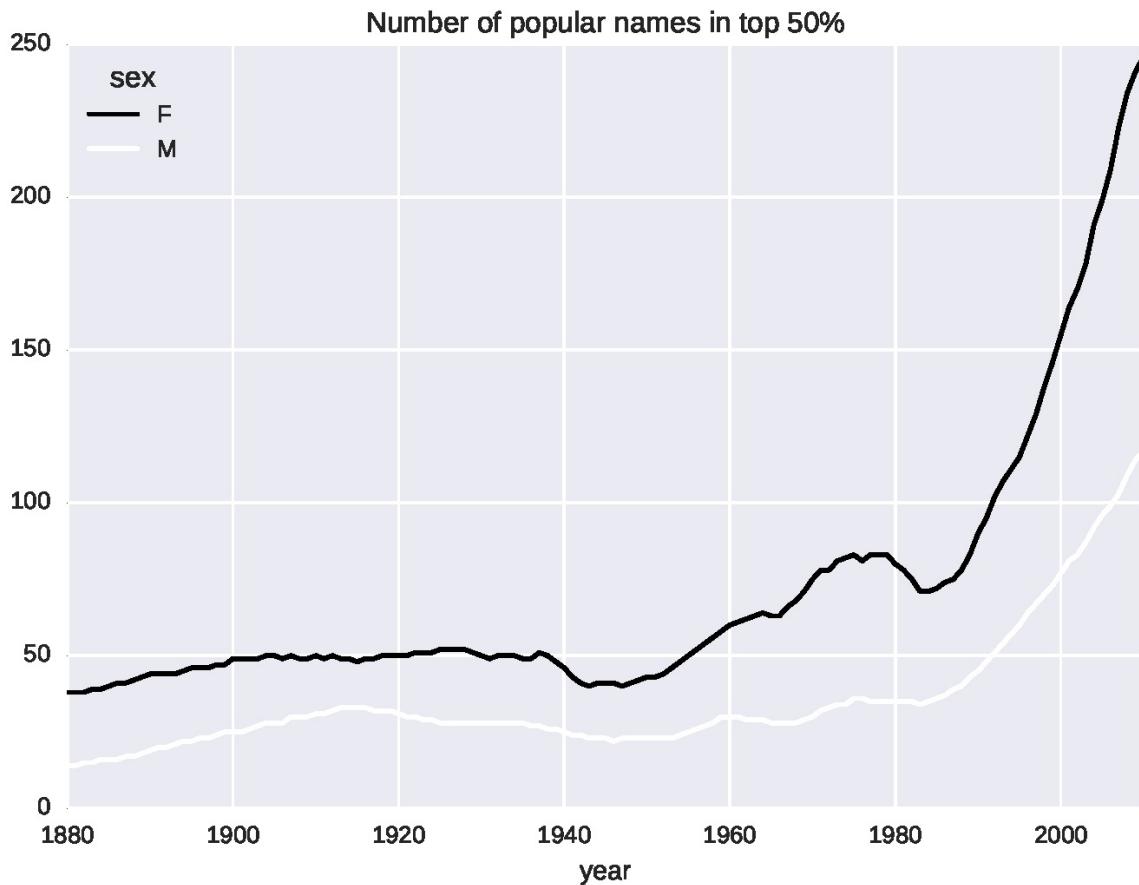


Figure 13-7. Plot of diversity metric by year

As you can see, girl names have always been more diverse than boy names, and they have only become more so over time. Further analysis of what exactly is driving the diversity, like the increase of alternate spellings, is left to the reader.

## The “Last letter” Revolution

In 2007, a baby name researcher Laura Wattenberg pointed out on her website (<http://www.babynamewizard.com>) that the distribution of boy names by final letter has changed significantly over the last 100 years. To see this, I first aggregate all of the births in the full data set by year, sex, and final letter:

```
# extract last letter from name column
get_last_letter = lambda x: x[-1]
```

```

last_letters = names.name.map(get_last_letter)
last_letters.name = 'last_letter'

table = names.pivot_table('births', index=last_letters,
                           columns=['sex', 'year'], aggfunc=sum)

```

Then, I select out three representative years spanning the history and print the first few rows:

```

In [131]: subtable = table.reindex(columns=[1910, 1960, 2010],)

In [132]: subtable.head()
Out[132]:
   sex          F          M
   year      1910      1960
last_letter
a      108376.0    691247.0    670605.0    977.0    5204.0
b          NaN       694.0       450.0    411.0    3912.0
c          5.0        49.0       946.0    482.0   15476.0
d       6750.0     3729.0     2607.0   22111.0   262112.0
e      133569.0    435013.0    313833.0    28655.0   178823.0 ...

```

Next, normalize the table by total births to compute a new table containing proportion of total births for each sex ending in each letter:

```

In [133]: subtable.sum()
Out[133]:
   sex  year
   F      396416.0
         1910      396416.0
         1960      2022062.0
         2010      1759010.0
   M      194198.0
         1910      194198.0
         1960      2132588.0
         2010      1898382.0
dtype: float64

In [134]: letter_prop = subtable / subtable.sum()

In [135]: letter_prop
Out[135]:
   sex          F          M
   year      1910      1960
last_letter
a      0.273390  0.341853  0.381240  0.005031  0.002440
b          NaN  0.000343  0.000256  0.002116  0.001834
c      0.000013  0.000024  0.000538  0.002482  0.007257

```

```

d          0.017028  0.001844  0.001482  0.113858  0.122908
e          0.336941  0.215133  0.178415  0.147556  0.083853
...
...          ...
v           NaN     0.000060  0.000117  0.000113  0.000037
w          0.000020  0.000031  0.001182  0.006329  0.007711
x          0.000015  0.000037  0.000727  0.003965  0.001851
y          0.110972  0.152569  0.116828  0.077349  0.160987
z          0.002439  0.000659  0.000704  0.000170  0.000184
[26 rows x 6 columns]

```

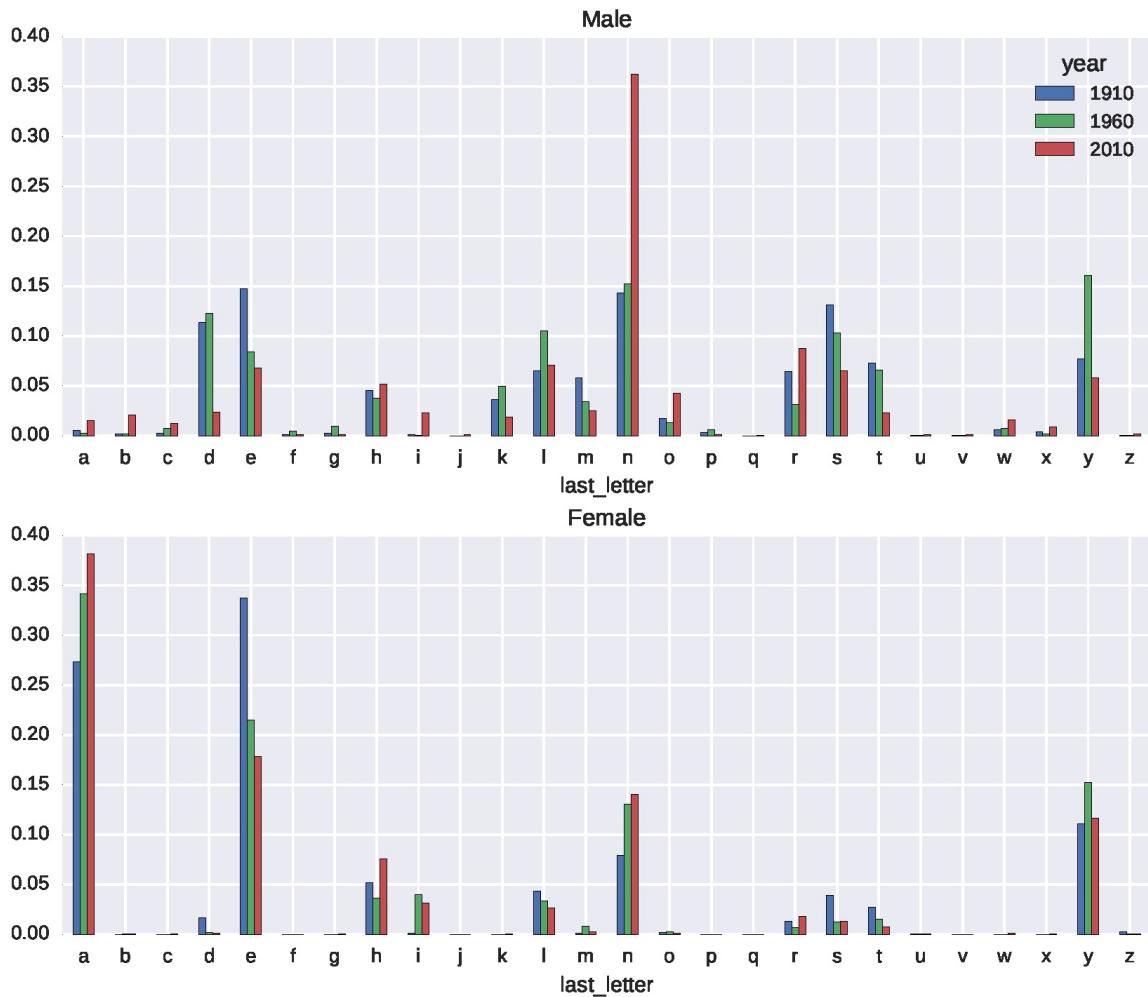
With the letter proportions now in hand, I can make bar plots for each sex broken down by year. See [Figure 13-8](#):

```

import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
letter_prop['M'].plot(kind='bar', rot=0, ax=axes[0], title='Male Letter Proportions')
letter_prop['F'].plot(kind='bar', rot=0, ax=axes[1], title='Female Letter Proportions')
plt.tight_layout()
plt.show()

```



**Figure 13-8. Proportion of boy and girl names ending in each letter**

As you can see, boy names ending in “n” have experienced significant growth since the 1960s. Going back to the full table created above, I again normalize by year and sex and select a subset of letters for the boy names, finally transposing to make each column a time series:

```
In [138]: letter_prop = table / table.sum()

In [139]: dny_ts = letter_prop.loc[['d', 'n', 'y'], 'M'].T

In [140]: dny_ts.head()
Out[140]:
last_letter          d          n          y
year
1880      0.083055  0.153213  0.075760
```

```
1881      0.083247  0.153214  0.077451
1882      0.085340  0.149560  0.077537
1883      0.084066  0.151646  0.079144
1884      0.086120  0.149915  0.080405
```

With this DataFrame of time series in hand, I can make a plot of the trends over time again with its `plot` method (see [Figure 13-9](#)):

```
In [143]: dny_ts.plot(colormap='gray')
```

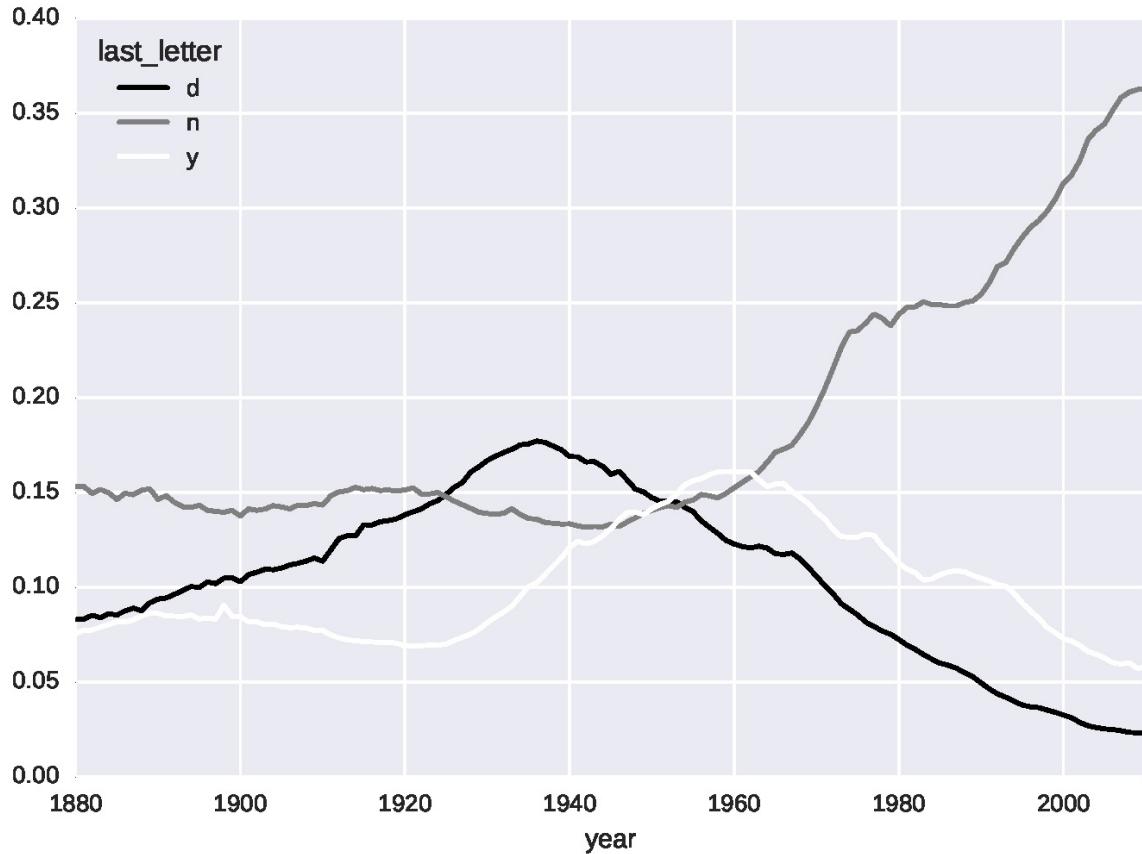


Figure 13-9. Proportion of boys born with names ending in d/n/y over time

## Boy names that became girl names (and vice versa)

Another fun trend is looking at boy names that were more popular with one sex earlier in the sample but have “changed sexes” in the present. One example is the name Lesley or Leslie. Going back to the `top1000` dataset, I compute a list of names occurring in the dataset starting with ‘lesl’:

```
In [144]: all_names = pd.Series(top1000.name.unique())
In [145]: lesley_like = all_names[all_names.str.lower().str.contains('lesl')]
In [146]: lesley_like
Out[146]:
632      Leslie
2294     Lesley
4262     Leslie
4728     Lesli
```

```
6103      Lesly
dtype: object
```

From there, we can filter down to just those names and sum births grouped by name to see the relative frequencies:

```
In [147]: filtered = top1000[top1000.name.isin(lesley_like)]
```

```
In [148]: filtered.groupby('name').births.sum()
```

```
Out[148]:
```

```
name
Leslee      1082
Lesley     35022
Lesli       929
Leslie    370429
Lesly      10067
Name: births, dtype: int64
```

Next, let's aggregate by sex and year and normalize within year:

```
In [149]: table = filtered.pivot_table('births', index='year',
                                         ....,
                                         columns='sex', aggfunc=
```

```
In [150]: table = table.div(table.sum(1), axis=0)
```

```
In [151]: table.tail()
```

```
Out[151]:
```

```
sex      F      M
year
2006    1.0  NaN
2007    1.0  NaN
2008    1.0  NaN
2009    1.0  NaN
2010    1.0  NaN
```

Lastly, it's now possible to make a plot of the breakdown by sex over time ([Figure 13-10](#)):

```
In [153]: table.plot(style={'M': 'k-', 'F': 'k--'})
```

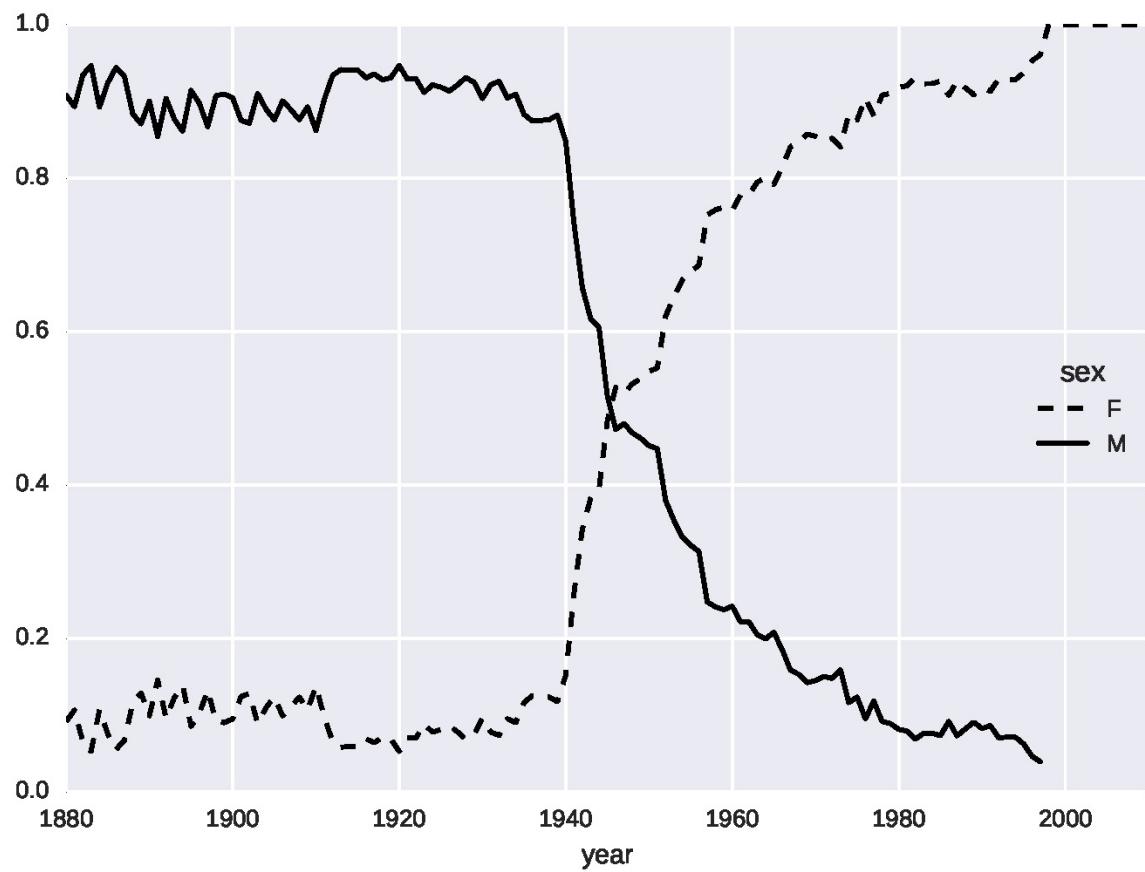


Figure 13-10. Proportion of male/female Lesley-like names over time

# USDA Food Database

The US Department of Agriculture makes available a database of food nutrient information. Ashley Williams, an English hacker, in 2011 made available a version of this database in JSON format. The records look like this:

```
{  
    "id": 21441,  
    "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA  
Wing, meat and skin with breading",  
    "tags": ["KFC"],  
    "manufacturer": "Kentucky Fried Chicken",  
    "group": "Fast Foods",  
    "portions": [  
        {  
            "amount": 1,  
            "unit": "wing, with skin",  
            "grams": 68.0  
        },  
  
        ...  
    ],  
    "nutrients": [  
        {  
            "value": 20.8,  
            "units": "g",  
            "description": "Protein",  
            "group": "Composition"  
        },  
  
        ...  
    ]  
}
```

Each food has a number of identifying attributes along with two lists of nutrients and portion sizes. Having the data in this form is not particularly amenable for analysis, so we need to do some work to wrangle the data into a better form.

After downloading and extracting the data from the link above, you can load it into Python with any JSON library of your choosing. I'll use the built-in Python

json module:

```
In [154]: import json  
  
In [155]: db = json.load(open('datasets/usda_food/database.json'))  
  
In [156]: len(db)  
Out[156]: 6636
```

Each entry in `db` is a dict containing all the data for a single food. The 'nutrients' field is a list of dicts, one for each nutrient:

```
In [157]: db[0].keys()  
Out[157]: dict_keys(['id', 'description', 'tags', 'manufacturer',  
                     'nutrients'])  
  
In [158]: db[0]['nutrients'][0]  
Out[158]:  
{'description': 'Protein',  
 'group': 'Composition',  
 'units': 'g',  
 'value': 25.18}  
  
In [159]: nutrients = pd.DataFrame(db[0]['nutrients'])  
  
In [160]: nutrients[:7]  
Out[160]:  
          description      group units    value  
0           Protein  Composition     g   25.18  
1  Total lipid (fat)  Composition     g   29.20  
2  Carbohydrate, by difference  Composition     g    3.06  
3            Ash        Other     g    3.28  
4            Energy       Energy   kcal  376.00  
5            Water  Composition     g   39.28  
6            Energy       Energy    kJ 1573.00
```

When converting a list of dicts to a DataFrame, we can specify a list of fields to extract. We'll take the food names, group, id, and manufacturer:

```
In [161]: info_keys = ['description', 'group', 'id', 'manufacturer']  
  
In [162]: info = pd.DataFrame(db, columns=info_keys)  
  
In [163]: info[:5]  
Out[163]:  
          description      group  
0  Cheese, caraway  Dairy and Egg Products
```

```
1           Cheese, cheddar  Dairy and Egg Products
2           Cheese, edam    Dairy and Egg Products
3           Cheese, feta    Dairy and Egg Products
4 Cheese, mozzarella, part skim milk  Dairy and Egg Products
   manufacturer
0
1
2
3
4
```

```
In [164]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
description      6636 non-null object
group            6636 non-null object
id               6636 non-null int64
manufacturer     5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB
```

You can see the distribution of food groups with `value_counts`:

```
In [165]: pd.value_counts(info.group)[:10]
Out[165]:
Vegetables and Vegetable Products      812
Beef Products                          618
Baked Products                        496
Breakfast Cereals                     403
Legumes and Legume Products          365
Fast Foods                            365
Lamb, Veal, and Game Products        345
Sweets                                341
Fruits and Fruit Juices              328
Pork Products                         328
Name: group, dtype: int64
```

Now, to do some analysis on all of the nutrient data, it's easiest to assemble the nutrients for each food into a single large table. To do so, we need to take several steps. First, I'll convert each list of food nutrients to a DataFrame, add a column for the food `id`, and append the DataFrame to a list. Then, these can be concatenated together with `concat`:

If all goes well, `nutrients` should look like this:

```
In [167]: nutrients
Out[167]:
          description      group units
0             Protein Composition   g
1    Total lipid (fat) Composition   g
2 Carbohydrate, by difference Composition   g
3                  Ash        Other   g
4                 Energy       Energy kcal
...
389350      Vitamin B-12, added  Vitamins mcg
389351           Cholesterol     Other   mg
389352  Fatty acids, total saturated  Other   g
389353  Fatty acids, total monounsaturated  Other   g
389354  Fatty acids, total polyunsaturated  Other   g
[389355 rows x 5 columns]
```

I noticed that there are duplicates in this DataFrame, so it makes things easier to drop them:

```
In [168]: nutrients.duplicated().sum() # number of duplicates
Out[168]: 14179
```

```
In [169]: nutrients = nutrients.drop_duplicates()
```

Since 'group' and 'description' is in both DataFrame objects, we can rename them to make it clear what is what:

```
In [170]: col_mapping = {'description' : 'food',
.....:                 'group'       : 'fgroup'}
```

```
In [171]: info = info.rename(columns=col_mapping, copy=False)
```

```
In [172]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
food            6636 non-null object
fgroup          6636 non-null object
id              6636 non-null int64
manufacturer    5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB
```

```
In [173]: col_mapping = {'description' : 'nutrient',
.....:                 'group'       : 'nutgroup'}
```

```
In [174]: nutrients = nutrients.rename(columns=col_mapping, copy=True)

In [175]: nutrients
Out[175]:

```

|                           | nutrient                           | nutgroup    | units |
|---------------------------|------------------------------------|-------------|-------|
| 0                         | Protein                            | Composition | g     |
| 1                         | Total lipid (fat)                  | Composition | g     |
| 2                         | Carbohydrate, by difference        | Composition | g     |
| 3                         | Ash                                | Other       | g     |
| 4                         | Energy                             | Energy      | kcal  |
| ...                       | ...                                | ...         | ...   |
| 389350                    | Vitamin B-12, added                | Vitamins    | mcg   |
| 389351                    | Cholesterol                        | Other       | mg    |
| 389352                    | Fatty acids, total saturated       | Other       | g     |
| 389353                    | Fatty acids, total monounsaturated | Other       | g     |
| 389354                    | Fatty acids, total polyunsaturated | Other       | g     |
| [375176 rows x 5 columns] |                                    |             |       |

With all of this done, we're ready to merge `info` with `nutrients`:

```
In [176]: ndata = pd.merge(nutrients, info, on='id', how='outer')
```

```
In [177]: ndata.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 375175
Data columns (total 8 columns):
nutrient      375176 non-null object
nutgroup       375176 non-null object
units          375176 non-null object
value          375176 non-null float64
id             375176 non-null int64
food           375176 non-null object
fgroup         375176 non-null object
manufacturer   293054 non-null object
dtypes: float64(1), int64(1), object(6)
memory usage: 25.8+ MB
```

```
In [178]: ndata.iloc[30000]
```

```
Out[178]:
```

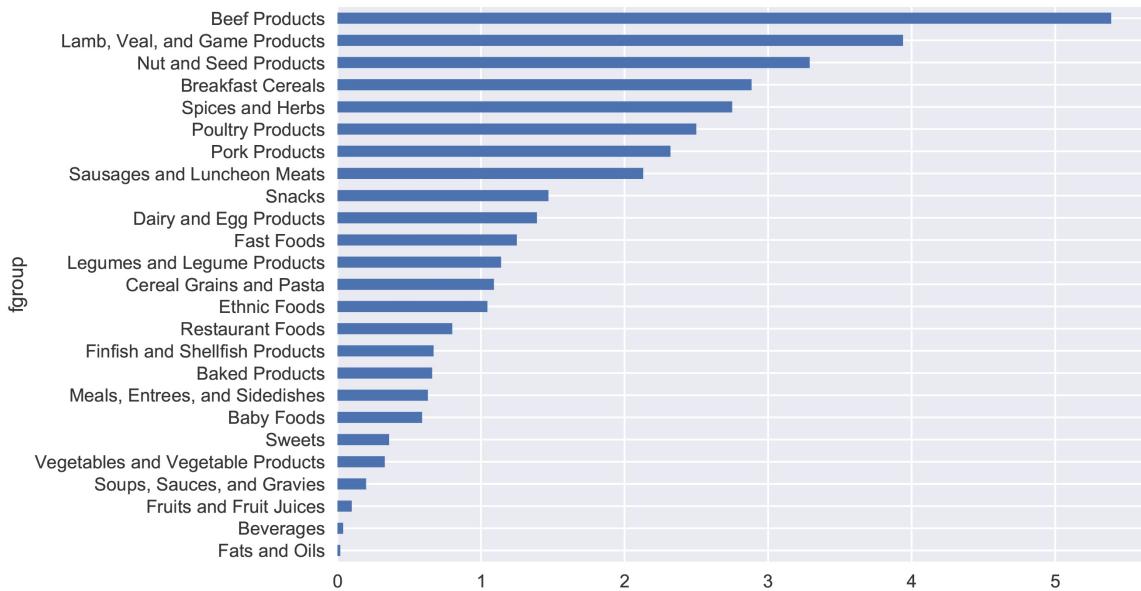
|              |  |
|--------------|--|
| nutrient     | Glycine                                |
| nutgroup     | Amino Acids                            |
| units        | g                                      |
| value        | 0.04                                   |
| id           | 6158                                   |
| food         | Soup, tomato bisque, canned, condensed |
| fgroup       | Soups, Sauces, and Gravies             |
| manufacturer |  |

```
Name: 30000, dtype: object
```

The tools that you need to slice and dice, aggregate, and visualize this dataset will be explored in detail in the next two chapters, so after you get a handle on those methods you might return to this dataset. For example, we could a plot of median values by food group and nutrient type (see [Figure 13-11](#)):

```
In [180]: result = ndata.groupby(['nutrient', 'fgroup'])['value'].median()
```

```
In [181]: result['Zinc, Zn'].sort_values().plot(kind='barh')
```



**Figure 13-11. Median Zinc values by nutrient group**

With a little cleverness, you can find which food is most dense in each nutrient:

```
by_nutrient = ndata.groupby(['nutgroup', 'nutrient'])

get_maximum = lambda x: x.loc[x.value.idxmax()]
get_minimum = lambda x: x.loc[x.value.idxmin()]

max_foods = by_nutrient.apply(get_maximum)[['value', 'food']]

# make the food a little smaller
max_foods.food = max_foods.food.str[:50]
```

The resulting DataFrame is a bit too large to display in the book; here is only the 'Amino Acids' nutrient group:

```
In [183]: max_foods.loc['Amino Acids']['food']
Out[183]:
nutrient
Alanine           Gelatins, dry powder, unsweetened
Arginine          Seeds, sesame flour, lowfat
Aspartic acid     Soy protein isolate
Cystine           Seeds, cottonseed flour, low fat (glandular)
Glutamic acid    Soy protein isolate
...
Serine            Soy protein isolate, PROTEIN TECHNOLOGIES INTERNATIONAL
```

Threonine           Soy protein isolate, PROTEIN TECHNOLOGIES INTL  
Tryptophan          Sea lion, Steller, meat with fat (Alaska Nat:  
Tyrosine           Soy protein isolate, PROTEIN TECHNOLOGIES INTL  
Valine             Soy protein isolate, PROTEIN TECHNOLOGIES INTL  
Name: food, Length: 19, dtype: object

# 2012 Federal Election Commission Database

The US Federal Election Commission publishes data on contributions to political campaigns. This includes contributor names, occupation and employer, address, and contribution amount. An interesting dataset is from the 2012 US presidential election

(<http://www.fec.gov/disclosurep/PDownload.do>). As of this writing (June 2012), the full dataset for all states is a 150 megabyte CSV file P00000001-ALL.csv, which can be loaded with `pandas.read_csv`:

```
In [184]: fec = pd.read_csv('datasets/fec/P00000001-ALL.csv')

In [185]: fec.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001731 entries, 0 to 1001730
Data columns (total 16 columns):
cmte_id           1001731 non-null object
cand_id           1001731 non-null object
cand_nm           1001731 non-null object
contbr_nm         1001731 non-null object
contbr_city       1001712 non-null object
contbr_st          1001727 non-null object
contbr_zip         1001620 non-null object
contbr_employer    988002 non-null object
contbr_occupation  993301 non-null object
contb_receipt_amt  1001731 non-null float64
contb_receipt_dt   1001731 non-null object
receipt_desc       14166 non-null object
memo_cd            92482 non-null object
memo_text          97770 non-null object
form_tp            1001731 non-null object
file_num           1001731 non-null int64
dtypes: float64(1), int64(1), object(14)
memory usage: 122.3+ MB
```

A sample record in the DataFrame looks like this:

```
In [186]: fec.iloc[123456]
Out[186]:
```

```
cmte_id          C00431445
cand_id          P80003338
cand_nm          Obama, Barack
contbr_nm        ELLMAN, IRA
contbr_city      TEMPE
...
receipt_desc     NaN
memo_cd          NaN
memo_text        NaN
form_tp          SA17A
file_num         772372
Name: 123456, Length: 16, dtype: object
```

You may think of some ways to start slicing and dicing this data to extract informative statistics about donors and patterns in the campaign contributions. I'll show you a number of different analyses that apply techniques in this book.

You can see that there are no political party affiliations in the data, so this would be useful to add. You can get a list of all the unique political candidates using `unique` (note that NumPy suppresses the quotes around the strings in the output):

```
In [187]: unique_cands = fec.cand_nm.unique()

In [188]: unique_cands
Out[188]:
array(['Bachmann, Michelle', 'Romney, Mitt', 'Obama, Barack',
       'Roemer, Charles E. 'Buddy' III', 'Pawlenty, Timothy',
       'Johnson, Gary Earl', 'Paul, Ron', 'Santorum, Rick', 'Cain,
       'Gingrich, Newt', 'McCotter, Thaddeus G', 'Huntsman, Jon
       'Perry, Rick'], dtype=object)

In [189]: unique_cands[2]
Out[189]: 'Obama, Barack'
```

One way to indicate party affiliation is using a dict:<sup>2</sup>

```
parties = {'Bachmann, Michelle': 'Republican',
           'Cain, Herman': 'Republican',
           'Gingrich, Newt': 'Republican',
           'Huntsman, Jon': 'Republican',
           'Johnson, Gary Earl': 'Republican',
           'McCotter, Thaddeus G': 'Republican',
           'Obama, Barack': 'Democrat',
           'Paul, Ron': 'Republican',
```

```
'Pawlenty, Timothy': 'Republican',
'Perry, Rick': 'Republican',
"Roemer, Charles E. 'Buddy' III": 'Republican',
'Romney, Mitt': 'Republican',
'Santorum, Rick': 'Republican'}
```

Now, using this mapping and the `map` method on Series objects, you can compute an array of political parties from the candidate names:

```
In [191]: fec.cand_nm[123456:123461]
Out[191]:
123456    Obama, Barack
123457    Obama, Barack
123458    Obama, Barack
123459    Obama, Barack
123460    Obama, Barack
Name: cand_nm, dtype: object

In [192]: fec.cand_nm[123456:123461].map(parties)
Out[192]:
123456    Democrat
123457    Democrat
123458    Democrat
123459    Democrat
123460    Democrat
Name: cand_nm, dtype: object
```

```
# Add it as a column
In [193]: fec['party'] = fec.cand_nm.map(parties)

In [194]: fec['party'].value_counts()
Out[194]:
Democrat      593746
Republican    407985
Name: party, dtype: int64
```

A couple of data preparation points. First, this data includes both contributions and refunds (negative contribution amount):

```
In [195]: (fec.contb_receipt_amt > 0).value_counts()
Out[195]:
True      991475
False     10256
Name: contb_receipt_amt, dtype: int64
```

To simplify the analysis, I'll restrict the data set to positive contributions:

```
In [196]: fec = fec[fec.contb_receipt_amt > 0]
```

Since Barack Obama and Mitt Romney are the main two candidates, I'll also prepare a subset that just has contributions to their campaigns:

```
In [197]: fec_mrbo = fec[fec.cand_nm.isin(['Obama', 'Barack', 'Romney'])]
```

# Donation Statistics by Occupation and Employer

Donations by occupation is another oft-studied statistic. For example, lawyers (attorneys) tend to donate more money to Democrats, while business executives tend to donate more to Republicans. You have no reason to believe me; you can see for yourself in the data. First, the total number of donations by occupation is easy:

```
In [198]: fec.contbr_occupation.value_counts()[:10]
Out[198]:
RETIRED                               233990
INFORMATION REQUESTED                 35107
ATTORNEY                                34286
HOMEMAKER                                29931
PHYSICIAN                                 23432
INFORMATION REQUESTED PER BEST EFFORTS  21138
ENGINEER                                    14334
TEACHER                                     13990
CONSULTANT                                  13273
PROFESSOR                                   12555
Name: contbr_occupation, dtype: int64
```

You will notice by looking at the occupations that many refer to the same basic job type, or there are several variants of the same thing. Here is a code snippet illustrates a technique for cleaning up a few of them by mapping from one occupation to another; note the “trick” of using `dict.get` to allow occupations with no mapping to “pass through”:

```
occ_mapping = {
    'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
    'INFORMATION REQUESTED' : 'NOT PROVIDED',
    'INFORMATION REQUESTED (BEST EFFORTS)' : 'NOT PROVIDED',
    'C.E.O.' : 'CEO'
}

# If no mapping provided, return x
f = lambda x: occ_mapping.get(x, x)
fec.contbr_occupation = fec.contbr_occupation.map(f)
```

I'll also do the same thing for employers:

```
emp_mapping = {
    'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
    'INFORMATION REQUESTED' : 'NOT PROVIDED',
    'SELF' : 'SELF-EMPLOYED',
    'SELF EMPLOYED' : 'SELF-EMPLOYED',
}

# If no mapping provided, return x
f = lambda x: emp_mapping.get(x, x)
fec.contbr_employer = fec.contbr_employer.map(f)
```

Now, you can use `pivot_table` to aggregate the data by party and occupation, then filter down to the subset that donated at least \$2 million overall:

```
In [201]: by_occupation = fec.pivot_table('contb_receipt_amt',
                                             index='contbr_occupat',
                                             columns='party', agg)

In [202]: over_2mm = by_occupation[by_occupation.sum(1) > 2000000]

In [203]: over_2mm
Out[203]:
party           Democrat      Republican
contbr_occupation
ATTORNEY        11141982.97  7.477194e+06
CEO              2074974.79   4.211041e+06
CONSULTANT       2459912.71   2.544725e+06
ENGINEER         951525.55   1.818374e+06
EXECUTIVE        1355161.05   4.138850e+06
...
PRESIDENT        1878509.95   4.720924e+06
PROFESSOR        2165071.08   2.967027e+05
REAL ESTATE       528902.09   1.625902e+06
RETIRED          25305116.38  2.356124e+07
SELF-EMPLOYED    672393.40   1.640253e+06
[17 rows x 2 columns]
```

It can be easier to look at this data graphically as a bar plot ('`barh`' means horizontal bar plot, see [Figure 13-12](#)):

```
In [205]: over_2mm.plot(kind='barh')
```

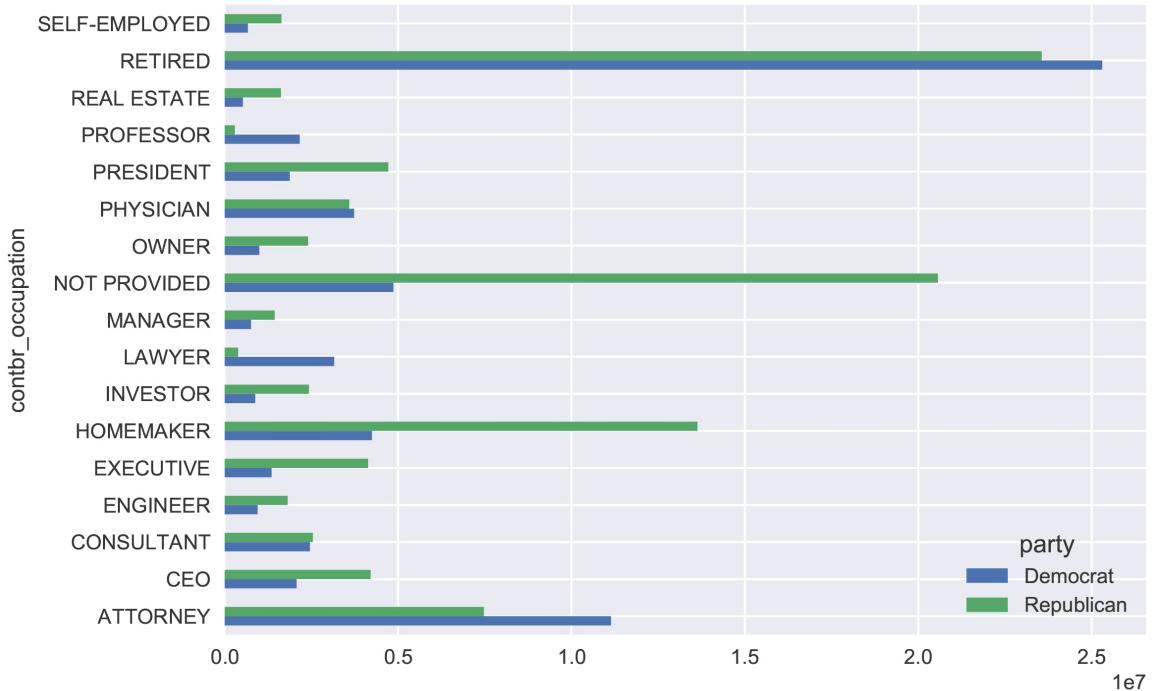


Figure 13-12. Total donations by party for top occupations

You might be interested in the top donor occupations or top companies donating to Obama and Romney. To do this, you can group by candidate name and use a variant of the `top` method from earlier in the chapter:

```
def get_top_amounts(group, key, n=5):
    totals = group.groupby(key) ['contb_receipt_amt'].sum()
    return totals.nlargest(n)
```

Then aggregated by occupation and employer:

```
In [207]: grouped = fec_mrbo.groupby('cand_nm')

In [208]: grouped.apply(get_top_amounts, 'contbr_occupation', 1)
Out[208]:
cand_nm      contbr_occupation
Obama, Barack  RETIRED           25305116.38
                  ATTORNEY        11141982.97
                  INFORMATION REQUESTED 4866973.96
                  HOMEMAKER        4248875.80
                  PHYSICIAN         3735124.94
                               ...
Romney, Mitt     HOMEMAKER       8147446.22
```

|           |            |
|-----------|------------|
| ATTORNEY  | 5364718.82 |
| PRESIDENT | 2491244.89 |
| EXECUTIVE | 2300947.03 |
| C.E.O.    | 1968386.11 |

Name: contb\_receipt\_amt, Length: 14, dtype: float64

In [209]: grouped.apply(get\_top\_amounts, 'contbr\_employer', n=1)

Out[209]:

| cand_nm       | contbr_employer       |             |
|---------------|-----------------------|-------------|
| Obama, Barack | RETIRED               | 22694358.85 |
|               | SELF-EMPLOYED         | 17080985.96 |
|               | NOT EMPLOYED          | 8586308.70  |
|               | INFORMATION REQUESTED | 5053480.37  |
|               | HOMEMAKER             | 2605408.54  |

...

|              |                    |           |
|--------------|--------------------|-----------|
| Romney, Mitt | CREDIT SUISSE      | 281150.00 |
|              | MORGAN STANLEY     | 267266.00 |
|              | GOLDMAN SACH & CO. | 238250.00 |
|              | BARCLAYS CAPITAL   | 162750.00 |
|              | H.I.G. CAPITAL     | 139500.00 |

Name: contb\_receipt\_amt, Length: 20, dtype: float64

# Bucketing Donation Amounts

A useful way to analyze this data is to use the `cut` function to discretize the contributor amounts into buckets by contribution size:

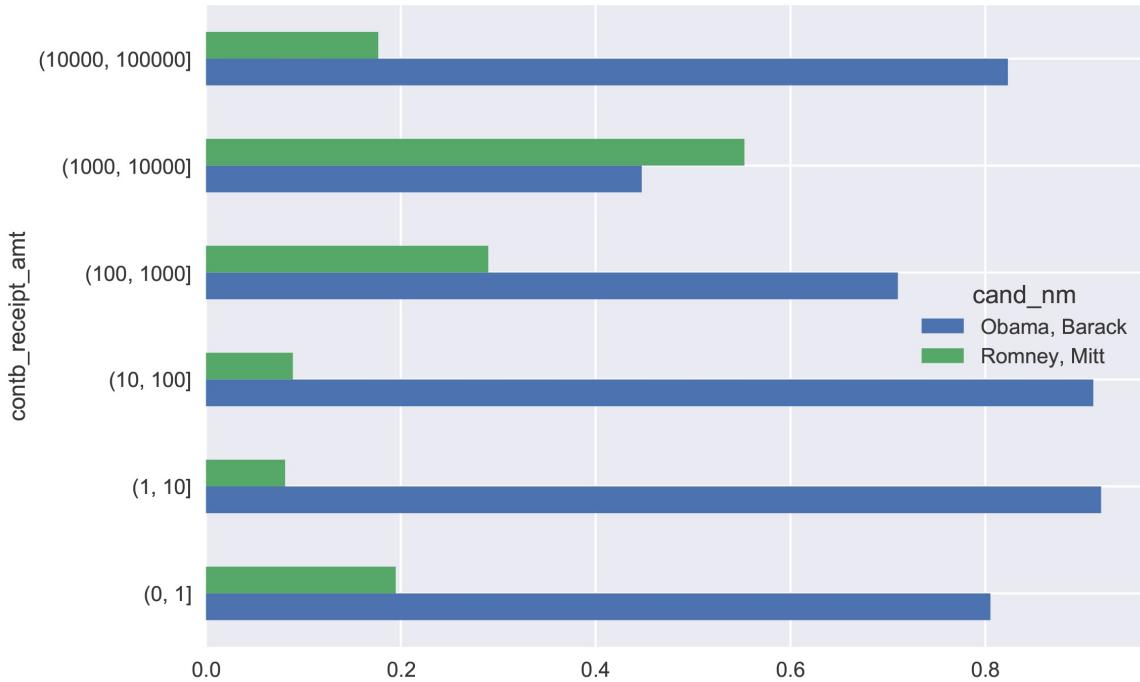
```
In [210]: bins = np.array([0, 1, 10, 100, 1000, 10000, 100000,  
In [211]: labels = pd.cut(fec_mrbo.contb_receipt_amt, bins)  
In [212]: labels  
Out[212]:  
411      (10, 100]  
412      (100, 1000]  
413      (100, 1000]  
414      (10, 100]  
415      (10, 100]  
        ...  
701381      (10, 100]  
701382      (100, 1000]  
701383      (1, 10]  
701384      (10, 100]  
701385      (100, 1000]  
Name: contb_receipt_amt, Length: 694282, dtype: category  
Categories (8, interval[int64]): [(0, 1] < (1, 10] < (10, 100]  
                                (10000, 100000] < (100000, 1000000]
```

We can then group the data for Obama and Romney by name and bin label to get a histogram by donation size:

```
In [213]: grouped = fec_mrbo.groupby(['cand_nm', labels])  
In [214]: grouped.size().unstack(0)  
Out[214]:  
cand_nm          Obama, Barack   Romney, Mitt  
contb_receipt_amt  
(0, 1]                  493.0       77.0  
(1, 10]                 40070.0      3681.0  
(10, 100]                372280.0     31853.0  
(100, 1000]              153991.0     43357.0  
(1000, 10000]             22284.0      26186.0  
(10000, 100000]            2.0         1.0  
(100000, 1000000]           3.0         NaN  
(1000000, 10000000]          4.0         NaN
```

This data shows that Obama has received a significantly larger number of small donations than Romney. You can also sum the contribution amounts and normalize within buckets to visualize percentage of total donations of each size by candidate:

```
In [216]: bucket_sums = grouped.contb_receipt_amt.sum().unstack  
In [217]: normed_sums = bucket_sums.div(bucket_sums.sum(axis=1))  
In [218]: normed_sums  
Out[218]:  
cand_nm          Obama, Barack  Romney, Mitt  
contb_receipt_amt  
(0, 1]           0.805182    0.194818  
(1, 10]          0.918767    0.081233  
(10, 100]         0.910769    0.089231  
(100, 1000]       0.710176    0.289824  
(1000, 10000]     0.447326    0.552674  
(10000, 100000]   0.823120    0.176880  
(100000, 1000000] 1.000000      NaN  
(1000000, 10000000] 1.000000      NaN  
  
In [219]: normed_sums[:-2].plot(kind='barh')
```



**Figure 13-13. Percentage of total donations received by candidates for each donation size**

I excluded the two largest bins as these are not donations by individuals. See [Figure 13-13](#) for the resulting figure.

There are of course many refinements and improvements of this analysis. For example, you could aggregate donations by donor name and zip code to adjust for donors who gave many small amounts versus one or more large donations. I encourage you to download it and explore it yourself.

# Donation Statistics by State

Aggregating the data by candidate and state is a routine affair:

```
In [220]: grouped = fec_mrbo.groupby(['cand_nm', 'contbr_st'])

In [221]: totals = grouped.contb_receipt_amt.sum().unstack(0).T

In [222]: totals = totals[totals.sum(1) > 100000]

In [223]: totals[:10]
Out[223]:
cand_nm      Obama, Barack    Romney, Mitt
contbr_st
AK           281840.15       86204.24
AL           543123.48      527303.51
AR           359247.28      105556.00
AZ           1506476.98     1888436.23
CA          23824984.24     11237636.60
CO           2132429.49     1506714.12
CT           2068291.26     3499475.45
DC           4373538.80     1025137.50
DE           336669.14      82712.00
FL           7318178.58     8338458.81
```

If you divide each row by the total contribution amount, you get the relative percentage of total donations by state for each candidate:

```
In [224]: percent = totals.div(totals.sum(1), axis=0)

In [225]: percent[:10]
Out[225]:
cand_nm      Obama, Barack    Romney, Mitt
contbr_st
AK           0.765778       0.234222
AL           0.507390       0.492610
AR           0.772902       0.227098
AZ           0.443745       0.556255
CA           0.679498       0.320502
CO           0.585970       0.414030
CT           0.371476       0.628524
DC           0.810113       0.189887
DE           0.802776       0.197224
```

FL 0.467417 0.532583

<sup>1</sup> <http://github.com/wesm/pydata-book>

<sup>2</sup> This makes the simplifying assumption that Gary Johnson is a Republican even though he later became the Libertarian party candidate.

# Chapter 14. Appendix: Advanced IPython and Jupyter

Act without doing; work without effort. Think of the small as large and the few as many. Confront the difficult while it is still easy; accomplish the great task by a series of small acts.

Laozi

Some of the features in this chapter are difficult to fully illustrate without a live IPython or Jupyter session. If this is your first time learning about IPython, I recommend that you follow along with the examples to get a feel for how things work. As with any keyboard-driven console-like environment, developing muscle-memory for the common commands is part of the learning curve.

While I'll give an overview of how to get started with Jupyter notebooks, most of this chapter focused on the *IPython system* which can be used either in the console (terminal) or within Jupyter (web notebook).

# Using the Command History

IPython maintains a small on-disk database containing the text of each command that you execute. This serves various purposes:

- Searching, completing, and executing previously-executed commands with minimal typing
- Persisting the command history between sessions.
- Logging the input/output history to a file

These features are more useful in the shell than in the notebook, since the notebook by design keeps a log of the input and output in each code cell.

# Searching and Reusing the Command History

The IPython shell lets you search and execute previous code or other commands. This is useful as you may often find yourself repeating the same commands, such as a `%run` command or some other code snippet. Suppose you had run:

```
In[7]: %run first/second/third/data_script.py
```

and then explored the results of the script (assuming it ran successfully), only to find that you made an incorrect calculation. After figuring out the problem and modifying `data_script.py`, you can start typing a few letters of the `%run` command then press either the `<Ctrl-P>` key combination or the `<up arrow>` key. This will search the command history for the first prior command matching the letters you typed. Pressing either `<Ctrl-P>` or `<up arrow>` multiple times will continue to search through the history. If you pass over the command you wish to execute, fear not. You can move *forward* through the command history by pressing either `<Ctrl-N>` or `<down arrow>`. After doing this a few times you may start pressing these keys without thinking!

Using `<Ctrl-R>` gives you the same partial incremental searching capability provided by the `readline` used in UNIX-style shells, such as the bash shell. On Windows, `readline` functionality is emulated by IPython. To use this, press `<Ctrl-R>` then type a few characters contained in the input line you want to search for:

```
In [1]: a_command = foo(x, y, z)  
(reverse-i-search)`com': a_command = foo(x, y, z)
```

Pressing `<Ctrl-R>` will cycle through the history for each line matching the characters you've typed.

# Input and Output Variables

Forgetting to assign the result of a function call to a variable can be very annoying. An IPython session stores references to *both* the input commands and output Python objects in special variables. The previous two outputs are stored in the `_` (one underscore) and `__` (two underscores) variables, respectively:

```
In [556]: 2 ** 27
Out[556]: 134217728

In [557]:
Out[557]: 134217728
```

Input variables are stored in variables named like `_ix`, where `x` is the input line number. For each such input variables there is a corresponding output variable `_x`. So after input line 27, say, there will be two new variables `_27` (for the output) and `_i27` for the input.

```
In [26]: foo = 'bar'

In [27]: foo
Out[27]: 'bar'

In [28]: _i27
Out[28]: u'foo'

In [29]: _27
Out[29]: 'bar'
```

Since the input variables are strings, that can be executed again using the Python `exec` keyword:

```
In [30]: exec _i27
```

Several magic functions allow you to work with the input and output history. `%hist` is capable of printing all or part of the input history, with or without line numbers. `%reset` is for clearing the interactive namespace and optionally the input and output caches. The `%xdel` magic function is intended for removing all references to a *particular* object from the IPython machinery. See the

documentation for both of these magics for more details.

**Warning**

When working with very large data sets, keep in mind that IPython's input and output history causes any object referenced there to not be garbage collected (freeing up the memory), even if you delete the variables from the interactive namespace using the `del` keyword. In such cases, careful usage of `%xdel` and `%reset` can help you avoid running into memory problems.

# Interacting with the Operating System

Another use feature of IPython is that it allows you to seamlessly access the file system and operating system shell. This means, among other things, that you can perform most standard command line actions as you would in the Windows or UNIX (Linux, OS X) shell without having to exit IPython. This includes shell commands, changing directories, and storing the results of a command in a Python object (list or string). There are also simple command aliasing and directory bookmarking features.

See [Table 14-1](#) for a summary of magic functions and syntax for calling shell commands. I'll briefly visit these features in the next few sections.

Table 14-1. IPython system-related commands

| Command                            | Description  |
|------------------------------------|--|
| <code>!cmd</code>                  | Execute <code>cmd</code> in the system shell                     |
| <code>output = !cmd args</code>    | Run <code>cmd</code> and store the stdout in <code>output</code> |
| <code>%alias alias_name cmd</code> | Define an alias for a system (shell) command                     |
| <code>%bookmark</code>             | Utilize IPython's directory bookmarking system                   |
| <code>%cd directory</code>         | Change system working directory to passed directory              |
| <code>%pwd</code>                  | Return the current system working directory                      |
| <code>%pushd directory</code>      | Place current directory on stack and change to target directory  |
| <code>%popd</code>                 | Change to directory popped off the top of the stack              |
| <code>%dirs</code>                 | Return a list containing the current directory stack             |
| <code>%dhist</code>                | Print the history of visited directories                         |
| <code>%env</code>                  | Return the system environment variables as a dict                |
| <code>%matplotlib</code>           | Configure matplotlib integration options                         |

# Shell Commands and Aliases

Starting a line in IPython with an exclamation point !, or bang, tells IPython to execute everything after the bang in the system shell. This means that you can delete files (using `rm` or `del`, depending on your OS), change directories, or execute any other process. It's even possible to start processes that take control away from IPython, or even start another Python interpreter:

```
Python 3.5.1 | packaged by conda-forge | (default, May 20 2016,
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more info
>>>
```

The console output of a shell command can be stored in a variable by assigning the !-escaped expression to a variable. For example, on my Linux-based machine connected to the Internet via ethernet, I can get my IP address as a Python variable:

```
In [1]: ip_info = !ifconfig wlan0 | grep "inet "
In [2]: ip_info[0].strip()
Out[2]: 'inet addr:10.0.0.11 Bcast:10.0.0.255 Mask:255.255.255.0'
```

The returned Python object `ip_info` is actually a custom list type containing various versions of the console output.

IPython can also substitute in Python values defined in the current environment when using !. To do this, preface the variable name by the dollar sign \$:

```
In [3]: foo = 'test*'
In [4]: !ls $foo
test4.py test.py test.xml
```

The `%alias` magic function can define custom shortcuts for shell commands. As a simple example:

```
In [1]: %alias ll ls -l
```

```
In [2]: ll /usr
total 332
drwxr-xr-x    2 root root   69632 2012-01-29 20:36 bin/
drwxr-xr-x    2 root root    4096 2010-08-23 12:05 games/
drwxr-xr-x 123 root root   20480 2011-12-26 18:08 include/
drwxr-xr-x 265 root root  126976 2012-01-29 20:36 lib/
drwxr-xr-x   44 root root   69632 2011-12-26 18:08 lib32/
lrwxrwxrwx    1 root root      3 2010-08-23 16:02 lib64 -> lib/
drwxr-xr-x   15 root root    4096 2011-10-13 19:03 local/
drwxr-xr-x    2 root root   12288 2012-01-12 09:32 sbin/
drwxr-xr-x 387 root root   12288 2011-11-04 22:53 share/
drwxrwsr-x   24 root src     4096 2011-07-17 18:38 src/
```

Multiple commands can be executed just as on the command line by separating them with semicolons:

```
In [558]: %alias test_alias (cd ch08; ls; cd ..)
```

```
In [559]: test_alias
macrodata.csv  spx.csv  tips.csv
```

You'll notice that IPython "forgets" any aliases you define interactively as soon as the session is closed. To create permanent aliases, you will need to use the configuration system. See later in the chapter.

# Directory Bookmark System

IPython has a simple directory bookmarking system to enable you to save aliases for common directories so that you can jump around very easily. For example, suppose you wanted to create a bookmark that points to the supplementary materials for this book:

```
In [6]: %bookmark py4da /home/wesm/code/pydata-book
```

Once I've done this, when I use the `%cd` magic, I can use any bookmarks I've defined

```
In [7]: cd py4da  
(bookmark:py4da) -> /home/wesm/code/pydata-book  
/home/wesm/code/pydata-book
```

If a bookmark name conflicts with a directory name in your current working directory, you can use the `-b` flag to override and use the bookmark location. Using the `-l` option with `%bookmark` lists all of your bookmarks:

```
In [8]: %bookmark -l  
Current bookmarks:  
py4da -> /home/wesm/code/pydata-book-source
```

Bookmarks, unlike aliases, are automatically persisted between IPython sessions.

# Software Development Tools

In addition to being a comfortable environment for interactive computing and data exploration, IPython can also be a useful companion for general Python software development. In data analysis applications, it's important first to have *correct* code. Fortunately, IPython has closely integrated and enhanced the built-in Python `pdb` debugger. Secondly you want your code to be *fast*. For this IPython has easy-to-use code timing and profiling tools. I will give an overview of these tools in detail here.

# Interactive Debugger

IPython's debugger enhances `pdb` with tab completion, syntax highlighting, and context for each line in exception tracebacks. One of the best times to debug code is right after an error has occurred. The `%debug` command, when entered immediately after an exception, invokes the “post-mortem” debugger and drops you into the stack frame where the exception was raised:

```
In [2]: run ch03/ipython_bug.py
-----
AssertionError                                     Traceback (most recent call
/home/wesm/code/pydata-book/ch03/ipython_bug.py in <module>()
    13     throws_an_exception()
    14
--> 15 calling_things()

/home/wesm/code/pydata-book/ch03/ipython_bug.py in calling_thi
    11 def calling_things():
    12     works_fine()
--> 13     throws_an_exception()
    14
    15 calling_things()

/home/wesm/code/pydata-book/ch03/ipython_bug.py in throws_an_e
    7     a = 5
    8     b = 6
--> 9     assert(a + b == 10)
   10
   11 def calling_things():

AssertionError:

In [3]: %debug
> /home/wesm/code/pydata-book/ch03/ipython_bug.py(9) throws_an_e
    8     b = 6
--> 9     assert(a + b == 10)
   10

ipdb>
```

Once inside the debugger, you can execute arbitrary Python code and explore all of the objects and data (which have been “kept alive” by the interpreter)

inside each stack frame. By default you start in the lowest level, where the error occurred. By pressing `u` (up) and `d` (down), you can switch between the levels of the stack trace:

```
ipdb> u
> /home/wesm/code/pydata-book/ch03/ipython_bug.py(13)calling_th
  12      works_fine()
---> 13      throws_an_exception()
  14
```

Executing the `%pdb` command makes it so that IPython automatically invokes the debugger after any exception, a mode that many users will find especially useful.

It's also easy to use the debugger to help develop code, especially when you wish to set breakpoints or step through the execution of a function or script to examine the state at each stage. There are several ways to accomplish this. The first is by using `%run` with the `-d` flag, which invokes the debugger before executing any code in the passed script. You must immediately press `s` (step) to enter the script:

```
In [5]: run -d ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/ch03/ipython_bug.py
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> s
--Call--
> /home/wesm/code/pydata-book/ch03/ipython_bug.py(1)<module>()
1---> 1 def works_fine():
      2     a = 5
      3     b = 6
```

After this point, it's up to you how you want to work your way through the file. For example, in the above exception, we could set a breakpoint right before calling the `works_fine` method and run the script until we reach the breakpoint by pressing `c` (continue):

```
ipdb> b 12
ipdb> c
> /home/wesm/code/pydata-book/ch03/ipython_bug.py(12)calling_th
  11 def calling_things():
```

```
2--> 12      works_fine()
13      throws_an_exception()
```

At this point, you can step into `works_fine()` or execute `works_fine()` by pressing `n` (next) to advance to the next line:

```
ipdb> n
> /home/wesm/code/pydata-book/ch03/ipython_bug.py(13)calling_tl
2    12      works_fine()
---> 13      throws_an_exception()
14
```

Then, we could step into `throws_an_exception` and advance to the line where the error occurs and look at the variables in the scope. Note that debugger commands take precedence over variable names; in such cases preface the variables with `!` to examine their contents.

```
ipdb> s
--Call--
> /home/wesm/code/pydata-book/ch03/ipython_bug.py(6)throws_an_e
    5
----> 6 def throws_an_exception():
    7     a = 5

ipdb> n
> /home/wesm/code/pydata-book/ch03/ipython_bug.py(7)throws_an_e
    6 def throws_an_exception():
----> 7     a = 5
    8     b = 6

ipdb> n
> /home/wesm/code/pydata-book/ch03/ipython_bug.py(8)throws_an_e
    7     a = 5
----> 8     b = 6
    9     assert(a + b == 10)

ipdb> n
> /home/wesm/code/pydata-book/ch03/ipython_bug.py(9)throws_an_e
    8     b = 6
----> 9     assert(a + b == 10)
10

ipdb> !a
5
ipdb> !b
```

Developing proficiency with the interactive debugger is largely a matter of practice and experience. See [Table 14-2](#) for a full catalogue of the debugger commands. If you are used to an IDE, you might find the terminal-driven debugger to be a bit unforgiving at first, but that will improve in time. Some of the Python IDEs have excellent GUI debuggers, so most users can find something that works for them.

Table 14-2. (I)Python debugger commands

| Command                            | Action   |
|------------------------------------|--|
| h (elp)                            | Display command list   |
| help <i>command</i>                | Show documentation for <i>command</i>                          |
| c (ontinue)                        | Resume program execution                                       |
| q (uit)                            | Exit debugger without executing any more code                  |
| b (reak) <i>number</i>             | Set breakpoint at <i>number</i> in current file                |
| b<br><i>path/to/file.py:number</i> | Set breakpoint at line <i>number</i> in specified file         |
| s (tep)                            | Step <i>into</i> function call                                 |
| n (ext)                            | Execute current line and advance to next line at current level |
| u (p) / d (own)                    | Move up/down in function call stack                            |
| a (rgs)                            | Show arguments for current function                            |
| debug <i>statement</i>             | Invoke statement <i>statement</i> in new (recursive) debugger  |
| l (ist) <i>statement</i>           | Show current position and context at current level of stack    |
| w (here)                           | Print full stack trace with context at current position        |

## Other ways to make use of the debugger

There are a couple of other useful ways to invoke the debugger. The first is by using a special `set_trace` function (named after `pdb.set_trace`), which is basically a “poor man’s breakpoint”. Here are two small recipes you might

want to put somewhere for your general use (potentially adding them to your IPython profile as I do):

```
def set_trace():
    from IPython.core.debugger import Pdb
    Pdb(color_scheme='Linux').set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):
    from IPython.core.debugger import Pdb
    pdb = Pdb(color_scheme='Linux')
    return pdb.runcall(f, *args, **kwargs)
```

The first function, `set_trace`, is very simple. Put `set_trace()` anywhere in your code that you want to stop and take a look around (for example, right before an exception occurs):

```
In [7]: run ch03/ipython_bug.py
> /home/wesm/code/pydata-book/ch03/ipython_bug.py(16)calling_tl
    15      set_trace()
--> 16      throws_an_exception()
    17
```

Pressing `c` (continue) will cause the code to resume normally with no harm done.

The `debug` function above enables you to invoke the interactive debugger easily on an arbitrary function call. Suppose we had written a function like

```
def f(x, y, z=1):
    tmp = x + y
    return tmp / z
```

and we wished to step through its logic. Ordinarily using `f` would look like `f(1, 2, z=3)`. To instead step into `f`, pass `f` as the first argument to `debug` followed by the positional and keyword arguments to be passed to `f`:

```
In [6]: debug(f, 1, 2, z=3)
> <ipython-input>(2)f()
    1 def f(x, y, z):
--> 2     tmp = x + y
    3     return tmp / z

ipdb>
```

I find that these two simple recipes save me a lot of time on a day-to-day basis.

Lastly, the debugger can be used in conjunction with `%run`. By running a script with `%run -d`, you will be dropped directly into the debugger, ready to set any breakpoints and start the script:

```
In [1]: %run -d ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/ch03/ipython_bug.py
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb>
```

Adding `-b` with a line number starts the debugger with a breakpoint set already:

```
In [2]: %run -d -b2 ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/ch03/ipython_bug.py
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> c
> /home/wesm/code/pydata-book/ch03/ipython_bug.py(2)works_fine
      1 def works_fine():
1---> 2      a = 5
      3      b = 6

ipdb>
```

# Timing Code: %time and %timeit

For larger-scale or longer-running data analysis applications, you may wish to measure the execution time of various components or of individual statements or function calls. You may want a report of which functions are taking up the most time in a complex process. Fortunately, IPython enables you to get this information very easily while you are developing and testing your code.

Timing code by hand using the built-in `time` module and its functions `time.clock` and `time.time` is often tedious and repetitive, as you must write the same uninteresting boilerplate code:

```
import time
start = time.time()
for i in range(iterations):
    # some code to run here
elapsed_per = (time.time() - start) / iterations
```

Since this is such a common operation, IPython has two magic functions `%time` and `%timeit` to automate this process for you. `%time` runs a statement once, reporting the total execution time. Suppose we had a large list of strings and we wanted to compare different methods of selecting all strings starting with a particular prefix. Here is a simple list of 600,000 strings and two identical methods of selecting only the ones that start with '`foo`':

```
# a very large list of strings
strings = ['foo', 'foobar', 'baz', 'qux',
           'python', 'Guido Van Rossum'] * 100000

method1 = [x for x in strings if x.startswith('foo')]

method2 = [x for x in strings if x[:3] == 'foo']
```

It looks like they should be about the same performance-wise, right? We can check for sure using `%time`:

```
In [561]: %time method1 = [x for x in strings if x.startswith('foo')]
CPU times: user 0.19 s, sys: 0.00 s, total: 0.19 s
Wall time: 0.19 s
```

```
In [562]: %time method2 = [x for x in strings if x[:3] == 'foo'
CPU times: user 0.09 s, sys: 0.00 s, total: 0.09 s
Wall time: 0.09 s
```

The `Wall time` (short for “wall-clock time”) is the main number of interest. So, it looks like the first method takes more than twice as long, but it’s not a very precise measurement. If you try `%time`-ing those statements multiple times yourself, you’ll find that the results are somewhat variable. To get a more precise measurement, use the `%timeit` magic function. Given an arbitrary statement, it has a heuristic to run a statement multiple times to produce a more accurate average runtime.

```
In [563]: %timeit [x for x in strings if x.startswith('foo')]
10 loops, best of 3: 159 ms per loop
```

```
In [564]: %timeit [x for x in strings if x[:3] == 'foo']
10 loops, best of 3: 59.3 ms per loop
```

This seemingly innocuous example illustrates that it is worth understanding the performance characteristics of the Python standard library, NumPy, pandas, and other libraries used in this book. In larger-scale data analysis applications, those milliseconds will start to add up!

`%timeit` is especially useful for analyzing statements and functions with very short execution times, even at the level of microseconds (millionths of a second) or nanoseconds (billions of a second). These may seem like insignificant amounts of time, but of course a 20 microsecond function invoked 1 million times takes 15 seconds longer than a 5 microsecond function. In the above example, we could very directly compare the two string operations to understand their performance characteristics:

```
In [565]: x = 'foobar'
```

```
In [566]: y = 'foo'
```

```
In [567]: %timeit x.startswith(y)
1000000 loops, best of 3: 267 ns per loop
```

```
In [568]: %timeit x[:3] == y
10000000 loops, best of 3: 147 ns per loop
```

# Basic Profiling: %prun and %run -p

Profiling code is closely related to timing code, except it is concerned with determining *where* time is spent. The main Python profiling tool is the `cProfile` module, which is not specific to IPython at all. `cProfile` executes a program or any arbitrary block of code while keeping track of how much time is spent in each function.

A common way to use `cProfile` is on the command line, running an entire program and outputting the aggregated time per function. Suppose we had a simple script which does some linear algebra in a loop (computing the maximum absolute eigenvalues of a series of  $100 \times 100$  matrices):

```
import numpy as np
from numpy.linalg import eigvals

def run_experiment(niter=100):
    K = 100
    results = []
    for _ in xrange(niter):
        mat = np.random.randn(K, K)
        max_eigenvalue = np.abs(eigvals(mat)).max()
        results.append(max_eigenvalue)
    return results
some_results = run_experiment()
print 'Largest one we saw: %s' % np.max(some_results)
```

Don't worry if you are not familiar with NumPy. You can run this script through `cProfile` by running the following in the command line:

```
python -m cProfile cprof_example.py
```

If you try that, you'll find that the results are outputted sorted by function name. This makes it a bit hard to get an idea of where the most time is spent, so it's very common to specify a *sort order* using the `-s` flag:

```
$ python -m cProfile -s cumulative cprof_example.py
Largest one we saw: 11.923204422
15116 function calls (14927 primitive calls) in 0.720 seconds
```

```
Ordered by: cumulative time
```

| ncalls | tottime | percall | cumtime | percall | filename:lineno(function)     |
|--------|---------|---------|---------|---------|-------------------------------|
| 1      | 0.001   | 0.001   | 0.721   | 0.721   | cprof_example.py:1(<module>)  |
| 100    | 0.003   | 0.000   | 0.586   | 0.006   | linalg.py:702(eigvals)        |
| 200    | 0.572   | 0.003   | 0.572   | 0.003   | {numpy.linalg.lapack.eigvals} |
| 1      | 0.002   | 0.002   | 0.075   | 0.075   | __init__.py:106(<module>)     |
| 100    | 0.059   | 0.001   | 0.059   | 0.001   | {method 'randn'}              |
| 1      | 0.000   | 0.000   | 0.044   | 0.044   | add_newdocs.py:9(<module>)    |
| 2      | 0.001   | 0.001   | 0.037   | 0.019   | __init__.py:1(<module>)       |
| 2      | 0.003   | 0.002   | 0.030   | 0.015   | __init__.py:2(<module>)       |
| 1      | 0.000   | 0.000   | 0.030   | 0.030   | type_check.py:3(<module>)     |
| 1      | 0.001   | 0.001   | 0.021   | 0.021   | __init__.py:15(<module>)      |
| 1      | 0.013   | 0.013   | 0.013   | 0.013   | numeric.py:1(<module>)        |
| 1      | 0.000   | 0.000   | 0.009   | 0.009   | __init__.py:6(<module>)       |
| 1      | 0.001   | 0.001   | 0.008   | 0.008   | __init__.py:45(<module>)      |
| 262    | 0.005   | 0.000   | 0.007   | 0.000   | function_base.py:31(<module>) |
| 100    | 0.003   | 0.000   | 0.005   | 0.000   | linalg.py:162(_assess)        |
| ...    |         |         |         |         |                               |

Only the first 15 rows of the output are shown. It's easiest to read by scanning down the `cumtime` column to see how much total time was spent *inside* each function. Note that if a function calls some other function, *the clock does not stop running*. `cProfile` records the start and end time of each function call and uses that to produce the timing.

In addition to the above command-line usage, `cProfile` can also be used programmatically to profile arbitrary blocks of code without having to run a new process. IPython has a convenient interface to this capability using the `%prun` command and the `-p` option to `%run`. `%prun` takes the same “command line options” as `cProfile` but will profile an arbitrary Python statement instead of a whole `.py` file:

```
In [4]: %prun -l 7 -s cumulative run_experiment()
        4203 function calls in 0.643 seconds
```

```
Ordered by: cumulative time
List reduced from 32 to 7 due to restriction <7>
```

| ncalls | tottime | percall | cumtime | percall | filename:lineno(function)     |
|--------|---------|---------|---------|---------|-------------------------------|
| 1      | 0.000   | 0.000   | 0.643   | 0.643   | <string>:1(<module>)          |
| 1      | 0.001   | 0.001   | 0.643   | 0.643   | cprof_example.py:4()          |
| 100    | 0.003   | 0.000   | 0.583   | 0.006   | linalg.py:702(eigvals)        |
| 200    | 0.569   | 0.003   | 0.569   | 0.003   | {numpy.linalg.lapack.eigvals} |

```
100    0.058    0.001    0.058    0.001 {method 'randn'}
```

```
100    0.003    0.000    0.005    0.000 linalg.py:162(_asse
```

```
200    0.002    0.000    0.002    0.000 {method 'all' of 'n
```

Similarly, calling `%run -p -s cumulative cprof_example.py` has the same effect as the command-line approach above, except you never have to leave IPython.

In the Jupyter notebook, you can use the `%%prun` magic (two % signs) to profile an entire code block. This pops up a separate window with the profile output. This can be useful in getting possibly quick answers to questions like, “Why did that code block take so long to run?”

# Profiling a Function Line-by-Line

In some cases the information you obtain from `%prun` (or another `cProfile`-based profile method) may not tell the whole story about a function's execution time, or it may be so complex that the results, aggregated by function name, are hard to interpret. For this case, there is a small library called `line_profiler` (obtainable via PyPI or one of the package management tools). It contains an IPython extension enabling a new magic function `%lprun` that computes a line-by-line-profiling of one or more functions. You can enable this extension by modifying your IPython configuration (see the IPython documentation or the section on configuration later in this chapter) to include the following line:

```
# A list of dotted module names of IPython extensions to load.  
c.TerminalIPythonApp.extensions = ['line_profiler']
```

You can also run the command:

```
%load_ext line_profiler
```

`line_profiler` can be used programmatically (see the full documentation), but it is perhaps most powerful when used interactively in IPython. Suppose you had a module `prof_mod` with the following code doing some NumPy array operations:

```
from numpy.random import randn  
  
def add_and_sum(x, y):  
    added = x + y  
    summed = added.sum(axis=1)  
    return summed  
  
def call_function():  
    x = randn(1000, 1000)  
    y = randn(1000, 1000)  
    return add_and_sum(x, y)
```

If we wanted to understand the performance of the `add_and_sum` function, `%prun` gives us the following:

```
In [569]: %run prof_mod

In [570]: x = randn(3000, 3000)

In [571]: y = randn(3000, 3000)

In [572]: %prun add_and_sum(x, y)
           4 function calls in 0.049 seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall  filename:lineno()
1        0.036    0.036    0.046    0.046  prof_mod.py:3(add_
1        0.009    0.009    0.009    0.009  {method 'sum' of
1        0.003    0.003    0.049    0.049  <string>:1(<modu
1        0.000    0.000    0.000    0.000  {method 'disable'
```

This is not especially enlightening. With the `line_profiler` IPython extension activated, a new command `%lprun` is available. The only difference in usage is that we must instruct `%lprun` which function or functions we wish to profile. The general syntax is:

```
%lprun -f func1 -f func2 statement_to_profile
```

In this case, we want to profile `add_and_sum`, so we run:

```
In [573]: %lprun -f add_and_sum add_and_sum(x, y)
Timer unit: 1e-06 s
File: prof_mod.py
Function: add_and_sum at line 3
Total time: 0.045936 s
Line #      Hits          Time  Per Hit   % Time  Line Contents
=====
3                      def add_and_...
4            1      36510  36510.0     79.5      added = x
5            1      9425   9425.0     20.5      summed = c
6            1          1       1.0      0.0      return sur
```

This can be much easier to interpret. In this case we profiled the same function we used in the statement. Looking at the module code above, we could call `call_function` and profile that as well as `add_and_sum`, thus getting a full picture of the performance of the code:

```
In [574]: %lprun -f add_and_sum -f call_function call_function
Timer unit: 1e-06 s
File: prof_mod.py
```

```

Function: add_and_sum at line 3
Total time: 0.005526 s
Line #      Hits          Time  Per Hit   % Time  Line Contents
=====
3                      def add_and_su
4          1        4375    4375.0     79.2  added = x
5          1        1149   1149.0     20.8  summed = a
6          1            2       2.0      0.0  return sum

File: prof_mod.py
Function: call_function at line 8
Total time: 0.121016 s
Line #      Hits          Time  Per Hit   % Time  Line Contents
=====
8                      def call_funct
9          1        57169   57169.0     47.2  x = randn
10         1        58304   58304.0     48.2  y = randn
11         1        5543    5543.0      4.6  return add(x,y)

```

As a general rule of thumb, I tend to prefer `%prun (cProfile)` for “macro” profiling and `%lprun (line_profiler)` for “micro” profiling. It’s worthwhile to have a good understanding of both tools.

#### Note

The reason that you have to specify explicitly the names of the functions you want to profile with `%lprun` is that the overhead of “tracing” the execution time of each line is significant. Tracing functions that are not of interest could potentially significantly alter the profile results.

# Tips for Productive Code Development Using IPython

Writing code in a way that makes it easy to develop, debug, and ultimately *use* interactively may be a paradigm shift for many users. There are procedural details like code reloading that may require some adjustment as well as coding style concerns.

As such, implementing most of the strategies described in this section is more of an art than a science and will require some experimentation on your part to determine a way to write your Python code that is effective for you. Ultimately you want to structure your code in a way that makes it easy to use iteratively and to be able to explore the results of running a program or function as effortlessly as possible. I have found software designed with IPython in mind to be easier to work with than code intended only to be run as a standalone command-line application. This becomes especially important when something goes wrong and you have to diagnose an error in code that you or someone else might have written months or years beforehand.

# Reloading Module Dependencies

In Python, when you type `import some_lib`, the code in `some_lib` is executed and all the variables, functions, and imports defined within are stored in the newly created `some_lib` module namespace. The next time you type `import some_lib`, you will get a reference to the existing module namespace. The potential difficulty in interactive code development in IPython comes when you, say, `%run` a script that depends on some other module where you may have made changes. Suppose I had the following code in `test_script.py`:

```
import some_lib

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

If you were to execute `%run test_script.py` then modify `some_lib.py`, the next time you execute `%run test_script.py` you will still get the *old version* of `some_lib` because of Python’s “load-once” module system. This behavior differs from some other data analysis environments, like MATLAB, which automatically propagate code changes.<sup>1</sup> To cope with this, you have a couple of options. The first way is to use Python’s built-in `reload` function, altering `test_script.py` to look like the following:

```
import some_lib
reload(some_lib)

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

This guarantees that you will get a fresh copy of `some_lib` every time you run `test_script.py`. Obviously, if the dependencies go deeper, it might be a bit tricky to be inserting usages of `reload` all over the place. For this problem, IPython has a special `dreload` function (*not* a magic function) for “deep” (recursive) reloading of modules. If I were to run `import some_lib` then type `dreload(some_lib)`, it will attempt to reload `some_lib` as well as all of its

dependencies. This will not work in all cases, unfortunately, but when it does it beats having to restart IPython.

# Code Design Tips

There's no simple recipe for this, but here are some high-level principles I have found effective in my own work.

## Keep relevant objects and data alive

It's not unusual to see a program written for the command line with a structure somewhat like the following trivial example:

```
from my_functions import g

def f(x, y):
    return g(x + y)

def main():
    x = 6
    y = 7.5
    result = x + y

if __name__ == '__main__':
    main()
```

Do you see what might be wrong with this program if we were to run it in IPython? After it's done, none of the results or objects defined in the `main` function will be accessible in the IPython shell. A better way is to have whatever code is in `main` execute directly in the module's global namespace (or in the `if __name__ == '__main__':` block, if you want the module to also be importable). That way, when you `%run` the code, you'll be able to look at all of the variables defined in `main`. This is equivalent to defining top-level variables in cells in the Jupyter notebook. It's less meaningful in this simple example, but in this book we'll be looking at some complex data analysis problems involving large data sets that you will want to be able to play with in IPython.

## Flat is better than nested

Deeply nested code makes me think about the many layers of an onion. When testing or debugging a function, how many layers of the onion must you peel back in order to reach the code of interest? The idea that “flat is better than nested” is a part of the Zen of Python, and it applies generally to developing code for interactive use as well. Making functions and classes as decoupled and modular as possible makes them easier to test (if you are writing unit tests), debug, and use interactively.

## Overcome a fear of longer files

If you come from a Java (or another such language) background, you may have been told to keep files short. In many languages, this is sound advice; long length is usually a bad “code smell”, indicating refactoring or reorganization may be necessary. However, while developing code using IPython, working with 10 small, but interconnected files (under, say, 100 lines each) is likely to cause you more headache in general than a single large file or two or three longer files. Fewer files means fewer modules to reload and less jumping between files while editing, too. I have found maintaining larger modules, each with high *internal* cohesion, to be much more useful and pythonic. After iterating toward a solution, it sometimes will make sense to refactor larger files into smaller ones.

Obviously, I don’t support taking this argument to the extreme, which would be to put all of your code in a single monstrous file. Finding a sensible and intuitive module and package structure for a large codebase often takes a bit of work, but it is especially important to get right in teams. Each module should be internally cohesive, and it should be as obvious as possible where to find functions and classes responsible for each area of functionality.

# Advanced IPython Features

Making full use of the IPython system may lead you to write your code in a slightly different way, or to dig into the configuration.

# Making Your Own Classes IPython-friendly

IPython makes every effort to display a console-friendly string representation of any object that you inspect. For many objects, like dicts, lists, and tuples, the built-in `pprint` module is used to do the nice formatting. In user-defined classes, however, you have to generate the desired string output yourself. Suppose we had the following simple class:

```
class Message:  
    def __init__(self, msg):  
        self.msg = msg
```

If you wrote this, you would be disappointed to discover that the default output for your class isn't very nice:

```
In [576]: x = Message('I have a secret')  
  
In [577]: x  
Out[577]: <__main__.Message instance at 0x60ebbd8>
```

IPython takes the string returned by the `__repr__` magic method (by doing `output = repr(obj)`) and prints that to the console. Thus, we can add a simple `__repr__` method to the above class to get a more helpful output:

```
class Message:  
    def __init__(self, msg):  
        self.msg = msg  
  
    def __repr__(self):  
        return 'Message: %s' % self.msg  
  
In [579]: x = Message('I have a secret')  
  
In [580]: x  
Out[580]: Message: I have a secret
```

# Profiles and Configuration

Most aspects of the appearance (colors, prompt, spacing between lines, etc.) and behavior of the IPython and Jupyter environments are configurable through an extensive configuration system. Here are some of the things you can do via configuration:

- Change the color scheme
- Change how the input and output prompts look, or remove the blank line after `Out` and before the next `In` prompt
- Execute an arbitrary list of Python statements. These could be imports that you use all the time or anything else you want to happen each time you launch IPython
- Enable always-on IPython extensions, like the `%lprun` magic in `line_profiler`
- Enabling Jupyter extensions
- Define your own magics or system aliases

Configurations for the IPython shell are specified in special `ipython_config.py` files which are usually found in the `.ipython/` directory in your user home directory. Configuration is performed based on a particular *profile*. When you start IPython normally, you load up, by default, the *default profile*, stored in the `profile_default` directory. Thus, on my Linux OS the full path to my default IPython configuration file is:

```
/home/wesm/.ipython/profile_default/ipython_config.py
```

To initialize this file on your system, run in the terminal:

```
ipython profile create
```

I'll spare you the gory details of what's in this file. Fortunately it has comments

describing what each configuration option is for, so I will leave it to the reader to tinker and customize. One additional useful feature is that it's possible to have *multiple profiles*. Suppose you wanted to have an alternate IPython configuration tailored for a particular application or project. Creating a new profile is as simple as typing something like

```
ipython profile create secret_project
```

Once you've done this, edit the config files in the newly-created `profile_secret_project` directory then launch IPython like so

```
$ ipython --profile=secret_project
Python 3.5.1 | packaged by conda-forge | (default, May 20 2016,
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?            -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help        -> Python's own help system.
object?     -> Details about 'object', use 'object??' for extra c

IPython profile: secret_project
```

As always, the online IPython documentation is an excellent resource for more on profiles and configuration.

Configuration for Jupyter works a little differently because you can use its notebooks with languages other than Python. To create an analogous Jupyter config file, run

```
jupyter notebook --generate-config
```

This writes a default config file to the

`.jupyter/jupyter_notebook_config.py` directory in your home directory. After editing this to suit your needs, you may rename it to a different file, like:

```
$ mv ~/.jupyter/jupyter_notebook_config.py ~/.jupyter/my_custom_config.py
```

When launching Jupyter, you can then add the `--config` argument:

```
jupyter notebook --config=~/.jupyter/my_custom_config.py
```

# Wrapping up

As you work through the code examples in the rest of the book and grow your skills as a Python programmer, I encourage you to keep learning about the IPython and Jupyter ecosystems. Since these projects have been designed to assist user productivity, you may discover tools which enable you to do your work more easily than using the Python language and its computational libraries by themselves.

You can also find a wealth of interesting Jupyter notebooks on the nbviewer website: <https://nbviewer.jupyter.org/>.

<sup>1</sup> Since a module or package may be imported in many different places in a particular program, Python caches a module's code the first time it is imported rather than executing the code in the module every time. Otherwise, modularity and good code organization could potentially cause inefficiency in an application.