

SECOND EDITION

Python for Data Analysis

*Data Wrangling with Pandas, NumPy,
and IPython*

Wes McKinney

O'REILLY®

Python for Data Analysis

by Wes McKinney

Copyright © 2018 William McKinney

Printed in the United States of America

October 2012: First Edition

October 2017: Second Edition

Revision History for the Second Edition

2017-09-25: First Release

<http://oreilly.com/catalog/errata.csp?isbn=9781491957660>

978-1-491-95766-0

[LSI]

Contents

Preface.....	xi
1. Preliminaries.....	1
1.1 What Is This Book About?	1
What Kinds of Data?	1
1.2 Why Python for Data Analysis?	2
Python as Glue	2
Solving the “Two-Language” Problem	3
Why Not Python?	3
1.3 Essential Python Libraries	4
NumPy	4
pandas	4
matplotlib	5
IPython and Jupyter	6
SciPy	6
scikit-learn	7
statsmodels	8
1.4 Installation and Setup	8
Windows	9
Apple (OS X, macOS)	9
GNU/Linux	9
Installing or Updating Python Packages	10
Python 2 and Python 3	11
Integrated Development Environments (IDEs) and Text Editors	11
1.5 Community and Conferences	12
1.6 Navigating This Book	12
Code Examples	13
Data for Examples	13

Import Conventions	14
Jargon	14
2. Python Language Basics, IPython, and Jupyter Notebooks.	15
2.1 The Python Interpreter	16
2.2 IPython Basics	17
Running the IPython Shell	17
Running the Jupyter Notebook	18
Tab Completion	21
Introspection	23
The %run Command	25
Executing Code from the Clipboard	26
Terminal Keyboard Shortcuts	27
About Magic Commands	28
Matplotlib Integration	29
2.3 Python Language Basics	30
Language Semantics	30
Scalar Types	38
Control Flow	46
3. Built-in Data Structures, Functions, and Files.	51
3.1 Data Structures and Sequences	51
Tuple	51
List	54
Built-in Sequence Functions	59
dict	61
set	65
List, Set, and Dict Comprehensions	67
3.2 Functions	69
Namespaces, Scope, and Local Functions	70
Returning Multiple Values	71
Functions Are Objects	72
Anonymous (Lambda) Functions	73
Currying: Partial Argument Application	74
Generators	75
Errors and Exception Handling	77
3.3 Files and the Operating System	80
Bytes and Unicode with Files	83
3.4 Conclusion	84
4. NumPy Basics: Arrays and Vectorized Computation.	85
4.1 The NumPy ndarray: A Multidimensional Array Object	87

Creating ndarrays	88
Data Types for ndarrays	90
Arithmetic with NumPy Arrays	93
Basic Indexing and Slicing	94
Boolean Indexing	99
Fancy Indexing	102
Transposing Arrays and Swapping Axes	103
4.2 Universal Functions: Fast Element-Wise Array Functions	105
4.3 Array-Oriented Programming with Arrays	108
Expressing Conditional Logic as Array Operations	109
Mathematical and Statistical Methods	111
Methods for Boolean Arrays	113
Sorting	113
Unique and Other Set Logic	114
4.4 File Input and Output with Arrays	115
4.5 Linear Algebra	116
4.6 Pseudorandom Number Generation	118
4.7 Example: Random Walks	119
Simulating Many Random Walks at Once	121
4.8 Conclusion	122
5. Getting Started with pandas.....	123
5.1 Introduction to pandas Data Structures	124
Series	124
DataFrame	128
Index Objects	134
5.2 Essential Functionality	136
Reindexing	136
Dropping Entries from an Axis	138
Indexing, Selection, and Filtering	140
Integer Indexes	145
Arithmetic and Data Alignment	146
Function Application and Mapping	151
Sorting and Ranking	153
Axis Indexes with Duplicate Labels	157
5.3 Summarizing and Computing Descriptive Statistics	158
Correlation and Covariance	160
Unique Values, Value Counts, and Membership	162
5.4 Conclusion	165
6. Data Loading, Storage, and File Formats.....	167
6.1 Reading and Writing Data in Text Format	167

Reading Text Files in Pieces	173
Writing Data to Text Format	175
Working with Delimited Formats	176
JSON Data	178
XML and HTML: Web Scraping	180
6.2 Binary Data Formats	183
Using HDF5 Format	184
Reading Microsoft Excel Files	186
6.3 Interacting with Web APIs	187
6.4 Interacting with Databases	188
6.5 Conclusion	190
7. Data Cleaning and Preparation.....	191
7.1 Handling Missing Data	191
Filtering Out Missing Data	193
Filling In Missing Data	195
7.2 Data Transformation	197
Removing Duplicates	197
Transforming Data Using a Function or Mapping	198
Replacing Values	200
Renaming Axis Indexes	201
Discretization and Binning	203
Detecting and Filtering Outliers	205
Permutation and Random Sampling	206
Computing Indicator/Dummy Variables	208
7.3 String Manipulation	211
String Object Methods	211
Regular Expressions	213
Vectorized String Functions in pandas	216
7.4 Conclusion	219
8. Data Wrangling: Join, Combine, and Reshape.....	221
8.1 Hierarchical Indexing	221
Reordering and Sorting Levels	224
Summary Statistics by Level	225
Indexing with a DataFrame's columns	225
8.2 Combining and Merging Datasets	227
Database-Style DataFrame Joins	227
Merging on Index	232
Concatenating Along an Axis	236
Combining Data with Overlap	241
8.3 Reshaping and Pivoting	242

Reshaping with Hierarchical Indexing	243
Pivoting “Long” to “Wide” Format	246
Pivoting “Wide” to “Long” Format	249
8.4 Conclusion	251
9. Plotting and Visualization.....	253
9.1 A Brief matplotlib API Primer	253
Figures and Subplots	255
Colors, Markers, and Line Styles	259
Ticks, Labels, and Legends	261
Annotations and Drawing on a Subplot	265
Saving Plots to File	267
matplotlib Configuration	268
9.2 Plotting with pandas and seaborn	268
Line Plots	269
Bar Plots	272
Histograms and Density Plots	277
Scatter or Point Plots	280
Facet Grids and Categorical Data	283
9.3 Other Python Visualization Tools	285
9.4 Conclusion	286
10. Data Aggregation and Group Operations.....	287
10.1 GroupBy Mechanics	288
Iterating Over Groups	291
Selecting a Column or Subset of Columns	293
Grouping with Dicts and Series	294
Grouping with Functions	295
Grouping by Index Levels	295
10.2 Data Aggregation	296
Column-Wise and Multiple Function Application	298
Returning Aggregated Data Without Row Indexes	301
10.3 Apply: General split-apply-combine	302
Suppressing the Group Keys	304
Quantile and Bucket Analysis	305
Example: Filling Missing Values with Group-Specific Values	306
Example: Random Sampling and Permutation	308
Example: Group Weighted Average and Correlation	310
Example: Group-Wise Linear Regression	312
10.4 Pivot Tables and Cross-Tabulation	313
Cross-Tabulations: Crosstab	315
10.5 Conclusion	316

11. Time Series.....	317
11.1 Date and Time Data Types and Tools	318
Converting Between String and Datetime	319
11.2 Time Series Basics	322
Indexing, Selection, Subsetting	323
Time Series with Duplicate Indices	326
11.3 Date Ranges, Frequencies, and Shifting	327
Generating Date Ranges	328
Frequencies and Date Offsets	330
Shifting (Leading and Lagging) Data	332
11.4 Time Zone Handling	335
Time Zone Localization and Conversion	335
Operations with Time Zone–Aware Timestamp Objects	338
Operations Between Different Time Zones	339
11.5 Periods and Period Arithmetic	339
Period Frequency Conversion	340
Quarterly Period Frequencies	342
Converting Timestamps to Periods (and Back)	344
Creating a PeriodIndex from Arrays	345
11.6 Resampling and Frequency Conversion	348
Downsampling	349
Upsampling and Interpolation	352
Resampling with Periods	353
11.7 Moving Window Functions	354
Exponentially Weighted Functions	358
Binary Moving Window Functions	359
User-Defined Moving Window Functions	361
11.8 Conclusion	362
12. Advanced pandas.....	363
12.1 Categorical Data	363
Background and Motivation	363
Categorical Type in pandas	365
Computations with Categoricals	367
Categorical Methods	370
12.2 Advanced GroupBy Use	373
Group Transforms and “Unwrapped” GroupBys	373
Grouped Time Resampling	377
12.3 Techniques for Method Chaining	378
The pipe Method	380
12.4 Conclusion	381

13. Introduction to Modeling Libraries in Python.....	383
13.1 Interfacing Between pandas and Model Code	383
13.2 Creating Model Descriptions with Patsy	386
Data Transformations in Patsy Formulas	389
Categorical Data and Patsy	390
13.3 Introduction to statsmodels	393
Estimating Linear Models	393
Estimating Time Series Processes	396
13.4 Introduction to scikit-learn	397
13.5 Continuing Your Education	401
14. Data Analysis Examples.....	403
14.1 1.USA.gov Data from Bitly	403
Counting Time Zones in Pure Python	404
Counting Time Zones with pandas	406
14.2 MovieLens 1M Dataset	413
Measuring Rating Disagreement	418
14.3 US Baby Names 1880–2010	419
Analyzing Naming Trends	425
14.4 USDA Food Database	434
14.5 2012 Federal Election Commission Database	440
Donation Statistics by Occupation and Employer	442
Bucketing Donation Amounts	445
Donation Statistics by State	447
14.6 Conclusion	448
A. Advanced NumPy.....	449
A.1 ndarray Object Internals	449
NumPy dtype Hierarchy	450
A.2 Advanced Array Manipulation	451
Reshaping Arrays	452
C Versus Fortran Order	454
Concatenating and Splitting Arrays	454
Repeating Elements: tile and repeat	457
Fancy Indexing Equivalents: take and put	459
A.3 Broadcasting	460
Broadcasting Over Other Axes	462
Setting Array Values by Broadcasting	465
A.4 Advanced ufunc Usage	466
ufunc Instance Methods	466
Writing New ufuncs in Python	468
A.5 Structured and Record Arrays	469

Nested dtypes and Multidimensional Fields	469
Why Use Structured Arrays?	470
A.6 More About Sorting	471
Indirect Sorts: argsort and lexsort	472
Alternative Sort Algorithms	474
Partially Sorting Arrays	474
numpy.searchsorted: Finding Elements in a Sorted Array	475
A.7 Writing Fast NumPy Functions with Numba	476
Creating Custom numpy.ufunc Objects with Numba	478
A.8 Advanced Array Input and Output	478
Memory-Mapped Files	478
HDF5 and Other Array Storage Options	480
A.9 Performance Tips	480
The Importance of Contiguous Memory	480
B. More on the IPython System.....	483
B.1 Using the Command History	483
Searching and Reusing the Command History	483
Input and Output Variables	484
B.2 Interacting with the Operating System	485
Shell Commands and Aliases	486
Directory Bookmark System	487
B.3 Software Development Tools	487
Interactive Debugger	488
Timing Code: %time and %timeit	492
Basic Profiling: %prun and %run -p	494
Profiling a Function Line by Line	496
B.4 Tips for Productive Code Development Using IPython	498
Reloading Module Dependencies	498
Code Design Tips	499
B.5 Advanced IPython Features	500
Making Your Own Classes IPython-Friendly	500
Profiles and Configuration	501
B.6 Conclusion	503
Index.....	505

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 second edition, I have overhauled the chapters to account both for incompatible changes and deprecations as well as new features that have occurred in the last five years. I've also added fresh content to introduce tools that either did not exist in 2012 or had not matured enough to make the first cut. Finally, I have tried to avoid writing about new or cutting-edge open source projects that 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.

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 the latest versions of the pandas library in 2017
- A new chapter on some more advanced pandas tools, and some other usage tips
- A brief introduction to using statsmodels and scikit-learn

I also reorganized a significant portion of the content from the first edition to make the book more accessible to newcomers.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

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

Constant width

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

Constant width bold

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

Constant width italic

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



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

Using Code Examples

You can find data files and related material for each chapter is available in this book's GitHub repository at <http://github.com/wesm/pydata-book>.

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: “*Python for Data Analysis* by Wes McKinney (O'Reilly). Copyright 2017 Wes McKinney, 978-1-491-95766-0.”

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.

1.1 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 datasets 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 dataset

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.

1.2 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, for various historical and cultural reasons, Python has developed a large and active scientific computing and data analysis community. In the last 10 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 support for libraries (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.

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 and 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 for building the production systems. 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 trade-off. However, in an application with very low latency or demanding resource utilization requirements (e.g., 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 that 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 dataset 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.

1.3 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, 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 datasets 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 more efficient for 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. Thus, many numerical computing tools for Python either assume NumPy arrays as a primary data structure or else target seamless interoperability with NumPy.

pandas

pandas 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 objects in pandas that will be used in this book are the `DataFrame`, a tabular, column-oriented data structure with both row and column labels, and the `Series`, a one-dimensional labeled array object.

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 the 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 that preserve metadata
- Flexible handling of missing data
- Merge and other relational operations found in popular 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 into 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 datasets, and a play on the phrase *Python data analysis* itself.

matplotlib

matplotlib is the most popular Python library for producing plots and other two-dimensional data visualizations. It was originally created by John D. Hunter and is now maintained by a large team of developers. It is designed 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 inte-

gration 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 16 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 filesystem. 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**, 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 open source project, which 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.

The Jupyter notebook system also allows you to author content in Markdown and HTML, providing you a means to create rich documents with code and text. Other programming languages have also implemented kernels for Jupyter to enable you to use languages other than Python in Jupyter.

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**, you will find Jupyter notebooks containing all the code examples from each chapter.

SciPy

SciPy 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

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` has become the premier general-purpose machine learning toolkit for Python programmers. In just seven years, it has had over 1,500 contributors from around the world. It includes submodules for such models as:

- Classification: SVM, nearest neighbors, random forest, logistic regression, etc.
- Regression: Lasso, ridge regression, 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 include a comprehensive guide to scikit-learn in this book, I will give a brief introduction to some of its models and how to use them with the other tools presented in the book.

statsmodels

statsmodels 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.
- Analysis of variance (ANOVA)
- Time series analysis: AR, ARMA, ARIMA, VAR, and other models
- Nonparametric methods: Kernel density estimation, kernel regression
- Visualization of statistical model results

statsmodels is more focused on statistical inference, providing uncertainty estimates and p -values for parameters. scikit-learn, by contrast, is more prediction-focused.

As with scikit-learn, I will give a brief introduction to statsmodels and how to use it with NumPy and pandas.

1.4 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. 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](#). I recommend following the installation instructions for Windows available on the Anaconda download page, which may have changed between the time this book was published and when you are reading this.

Now, let's verify that things are configured correctly. To open the Command Prompt application (also known as *cmd.exe*), right-click the Start menu and select Command Prompt. 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, 11:41:13)
[MSC v.1900 64 bit (AMD64)] on win32
>>>
```

To exit the shell, press Ctrl-D (on Linux or macOS), Ctrl-Z (on Windows), or type the command **exit()** and press Enter.

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
```

To exit the shell, press Ctrl-D or type **exit()** and press Enter.

GNU/Linux

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
```



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

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` (or `.zshrc`, if you are using the `zsh` shell) 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`.

Installing or Updating Python Packages

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

```
conda install package_name
```

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

```
pip install package_name
```

You can update packages by using the `conda update` command:

```
conda update package_name
```

`pip` also supports upgrades using the `--upgrade` flag:

```
pip install --upgrade package_name
```

You will have several opportunities to try out these commands throughout the book.



While you can use both `conda` and `pip` to install packages, you should not attempt to update `conda` packages with `pip`, as doing so can lead to environment problems. When using Anaconda or Miniconda, it's best to first try updating with `conda`.

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. Because 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 during that time.

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. Therefore, 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. Some code examples may work differently or not at all in Python 2.7.

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 is 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 (subscription-based 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.

1.5 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 to take a look at include:

- `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` (scikit-learn@python.org) 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 an 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. Here are some to consider:

- 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)

1.6 Navigating This Book

If you have never programmed in Python before, you will want to spend some time in Chapters 2 and 3, 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 [Appendix A](#). 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 a few isolated cases where concepts are used that 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 datasets to derive new datasets (e.g., 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 EXAMPLE
Out[5]: OUTPUT
```

When you see a code example like this, the intent is for you to type in the example code in the In block in your coding environment and execute it by pressing the Enter key (or Shift-Enter in Jupyter). You should see output similar to what is shown in the Out block.

Data for Examples

Datasets for the examples in each chapter are hosted in [a GitHub repository](#). 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. If you run into problems, navigate to [my website](#) for up-to-date instructions about obtaining the book materials.

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 email: book@wesmckinney.com. The best way to report errors in the book is on the [errata page on the O'Reilly website](#).

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 statsmodels as sm
```

This means that when you see `np.arange`, this is a reference to the `arange` function in NumPy. This is done because 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 that does not add new features, but makes something more convenient or easier to type.

Python Language Basics, IPython, and Jupyter Notebooks

When I wrote the first 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 2017, there is now a growing literature on data science, data analysis, and machine learning, supplementing the prior works on general-purpose scientific computing geared toward computational scientists, physicists, and professionals in other research fields. There are also 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. So, I will only present roughly enough information in this chapter and [Chapter 3](#) 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 datasets. 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 datasets 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.



There are introductory Python concepts 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](#) and potentially one of the many excellent books on general-purpose Python programming. Some recommendations to get you started include:

- *Python Cookbook*, Third Edition, by David Beazley and Brian K. Jones (O'Reilly)
- *Fluent Python* by Luciano Ramalho (O'Reilly)
- *Effective Python* by Brett Slatkin (Pearson)

2.1 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, 23:17:12)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> print(a)
5
```

The `>>>` you see is the *prompt* where you'll type code 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')
```

You can run it by executing the following command (the *hello_world.py* file must be in your current working terminal directory):

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

While some Python programmers execute all of their Python code in this way, those doing data analysis or scientific computing make use of IPython, an enhanced Python interpreter, or Jupyter notebooks, web-based code notebooks 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 in [Appendix A](#). When you use 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, 23:17:12)
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 details.

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.

2.2 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, 23:17:12)
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 details.

```
In [1]: a = 5
```

```
In [2]: a
```

```
Out[2]: 5
```

You can execute arbitrary Python statements by typing them in and pressing Return (or Enter). When you type just a variable into IPython, it renders a string representation of the object:

```
In [5]: import numpy as np
```

```
In [6]: data = {i : np.random.randn() for i in range(7)}
```

```
In [7]: data
```

```
Out[7]:
```

```
{0: -0.20470765948471295,  
1: 0.47894333805754824,  
2: -0.5194387150567381,  
3: -0.55573030434749,  
4: 1.9657805725027142,  
5: 1.3934058329729904,  
6: 0.09290787674371767}
```

The first two lines are Python code statements; the second statement creates a variable named `data` that refers to a newly created Python dictionary. The last line prints the value of `data` in the console.

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 the above `data` variable 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: 1.972367135977295,  
3: 0.15455217573074576, 4: -0.24058577449429575, 5: -1.2904897053651216,  
6: 0.3308507317325902}
```

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

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:
http://localhost:8888/
[I 15:20:52.740 NotebookApp] Use Control-C to stop this server and shut down
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 in Google Chrome.



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.

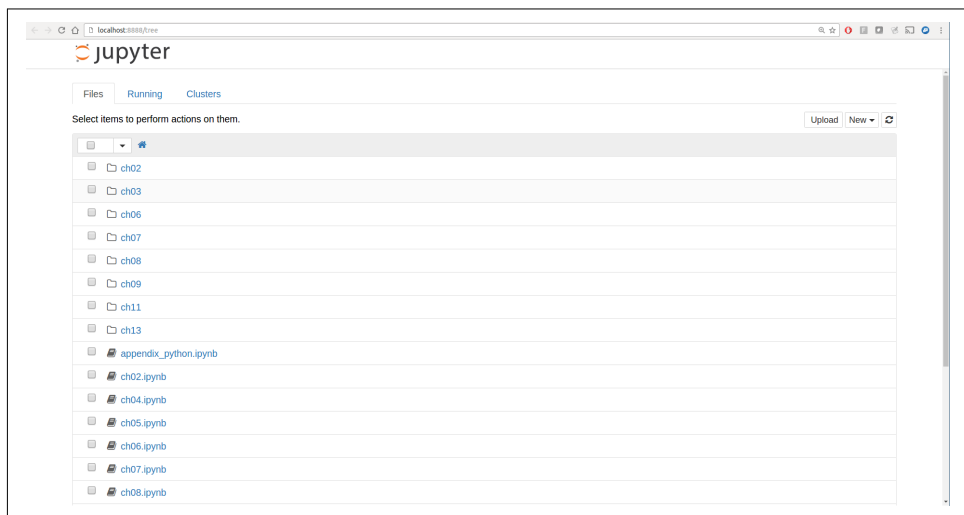


Figure 2-1. Jupyter notebook landing page

To create a new notebook, click the New button and select the “Python 3” or “conda [default]” 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.

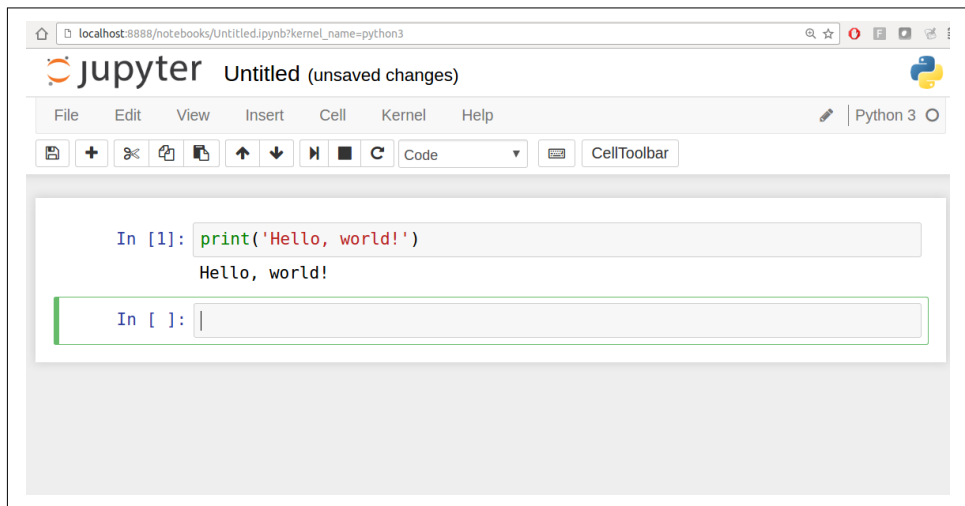


Figure 2-2. Jupyter new notebook view

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 where you started the notebook process (or in a subfolder within it), 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](#).

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.

Introductory examples

1.usa.gov data from bit.ly ¶

```
In [ ]: %pwd
```

```
In [ ]: path = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'
```

```
In [ ]: open(path).readline()
```

```
In [ ]: import json
path = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'
records = [json.loads(line) for line in open(path)]
```

```
In [ ]: records[0]
```

```
In [ ]: records[0]['tz']
```

```
In [ ]: print(records[0]['tz'])
```

Counting time zones in pure Python

Figure 2-3. Jupyter example view for an existing notebook

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 the Tab key 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 auto-completions show up in a drop-down box rather than as text output.



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. See the IPython documentation to find out how to do this.

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 the Tab key will complete anything on your computer’s filesystem 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 [7]: 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 “[The %run Command](#)” on page 25), 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](#).

```
In [12]: def func_with_keywords(abra=1, abbra=2, abbbra=3):  
         return abra, abbra, abbbra
```

```
In [ ]: func_with_keywords(ab)
```

```
abbbra=  
abbra=  
abra=  
abs
```

Figure 2-4. Autocomplete function keywords in Jupyter notebook

We'll have a closer look at functions in a little bit.

Introspection

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

```
In [8]: b = [1, 2, 3]
```

```
In [9]: b?
```

```
Type: list
```

```
String Form:[1, 2, 3]
```

```
Length: 3
```

```
Docstring:
```

```
list() -> new empty list
```

```
list(iterable) -> new list initialized from iterable's items
```

```
In [10]: print?
```

```
Docstring:
```

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

```
Prints the values to a stream, or to sys.stdout by default.
```

```
Optional keyword arguments:
```

```
file: a file-like object (stream); defaults to the current sys.stdout.
```

```
sep: string inserted between values, default a space.
```

```
end: string appended after the last value, default a newline.
```

```
flush: whether to forcibly flush the stream.
```

```
Type: builtin_function_or_method
```

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 (which you can reproduce in IPython or Jupyter):

```
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 [11]: 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 [12]: 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 [13]: np.*load*?
np.__loader__
np.load
np.loads
np.loadtxt
np.pkgload
```

The %run Command

You can run any file 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)
```

You can execute this by passing the filename to `%run`:

```
In [14]: %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 [15]: c  
Out [15]: 7.5  
  
In [16]: result  
Out[16]: 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.



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 unusual cases.



When a piece of Python code has called into some compiled extension modules, pressing Ctrl-C will not always cause the program execution to stop immediately. 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 [17]: %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 [18]: %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. [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.

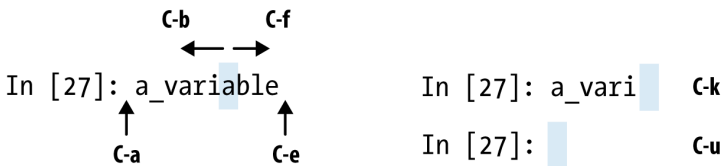


Figure 2-5. Illustration of some keyboard shortcuts in the IPython shell

Table 2-1. Standard IPython keyboard shortcuts

Keyboard shortcut	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. Since these shortcuts have evolved more rapidly than IPython's, I encourage you to explore the integrated help system in the Jupyter notebook's menus.

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 [20]: a = np.random.randn(100, 100)

In [20]: %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 [21]: %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, you can set a **break** point, to step through the code **from the point**. You can use this mode by giving statements to execute **and** optionally a breakpoint.

The other one **is** to activate debugger **in** post-mortem mode. You can 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 the last traceback that occurred, so you must call this quickly after an exception that you wish to inspect has fired, because **if** another one occurs, it clobbers the previous one.

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

positional arguments:

statement	Code to run in debugger. You can omit this in cell magic mode.
-----------	--

optional arguments:

<code>--breakpoint <FILE:LINE></code> , <code>-b <FILE:LINE></code>	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 *automatic* and can be enabled or disabled with `%automatic`.

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

```
In [22]: %pwd
Out[22]: '/home/wesm/code/pydata-book'
```

```
In [23]: foo = %pwd
```

```
In [24]: foo
Out[24]: '/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`. [Table 2-2](#) highlights some 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
<code>%quickref</code>	Display the IPython Quick Reference Card
<code>%magic</code>	Display detailed documentation for all of the available magic commands
<code>%debug</code>	Enter the interactive debugger at the bottom of the last exception traceback
<code>%hist</code>	Print command input (and optionally output) history
<code>%pdb</code>	Automatically enter debugger after any exception
<code>%paste</code>	Execute preformatted Python code from clipboard
<code>%cpaste</code>	Open a special prompt for manually pasting Python code to be executed
<code>%reset</code>	Delete all variables/names defined in interactive namespace
<code>%page OBJECT</code>	Pretty-print the object and display it through a pager
<code>%run script.py</code>	Run a Python script inside IPython
<code>%prun statement</code>	Execute <i>statement</i> with <code>cProfile</code> and report the profiler output
<code>%time statement</code>	Report the execution time of a single statement
<code>%timeit statement</code>	Run a statement multiple times to compute an ensemble average execution time; useful for timing code with very short execution time
<code>%who</code> , <code>%who_ls</code> , <code>%whos</code>	Display variables defined in interactive namespace, with varying levels of information/verbosity
<code>%xdel variable</code>	Delete a variable and attempt to clear any references to the object in the IPython internals

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 [26]: %matplotlib
Using matplotlib backend: Qt4Agg
```

In Jupyter, the command is a little different (Figure 2-6):

```
In [26]: %matplotlib inline
```

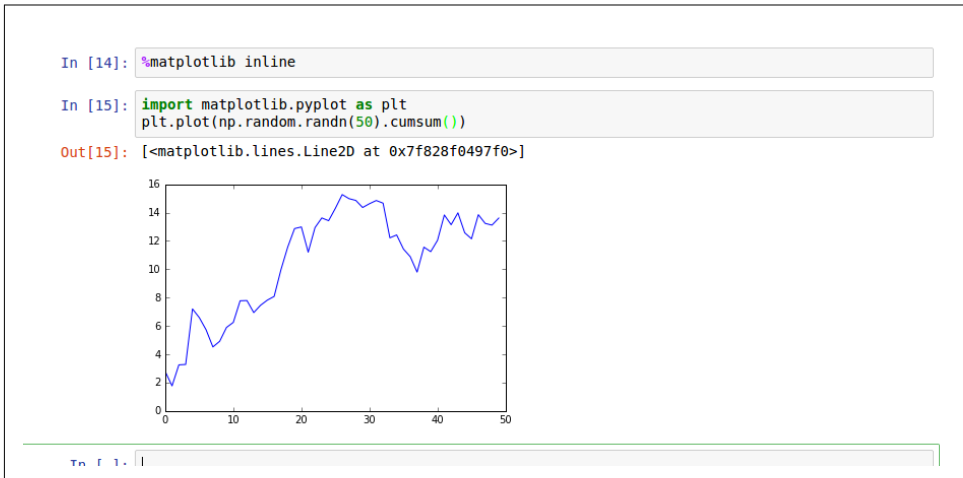


Figure 2-6. Jupyter inline matplotlib plotting

2.3 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 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. Consider a for loop from a sorting 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.

Love it or hate it, significant whitespace is a fact of life for Python programmers, and in my experience it can make Python code more readable than other languages I've used. While it may seem foreign at first, you will hopefully grow accustomed in time.



I strongly recommend using *four spaces* as your default indentation and replacing tabs with four 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 two spaces not being terribly uncommon. By and large, four spaces is 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* (e.g., *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

You call functions 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. You can call them using the following 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 argument passing

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

```
In [8]: a = [1, 2, 3]
```

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

```
In [9]: 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 [10]: a.append(4)
```

```
In [11]: b
Out[11]: [1, 2, 3, 4]
```

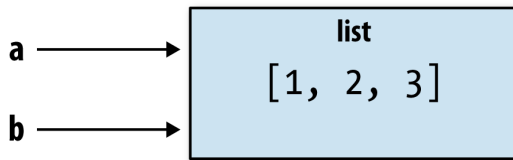


Figure 2-7. Two references for the same object

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



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, new local variables are created referencing the original objects without any copying. If you bind a new object to a variable inside a function, that change will not be reflected in the parent scope. It is therefore possible to alter the internals of a mutable argument. Suppose we had the following function:

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

Then we have:

```
In [27]: data = [1, 2, 3]
```

```
In [28]: append_element(data, 4)
```

```
In [29]: data
```

```
Out[29]: [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 [12]: a = 5
```

```
In [13]: type(a)
```

```
Out[13]: int
```

```
In [14]: a = 'foo'
```

```
In [15]: type(a)
Out[15]: 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 [16]: '5' + 5
-----
TypeError                                Traceback (most recent call last)
<ipython-input-16-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 [17]: a = 4.5

In [18]: b = 2

# String formatting, to be visited later
In [19]: print('a is {0}, b is {1}'.format(type(a), type(b)))
a is <class 'float'>, b is <class 'int'>

In [20]: a / b
Out[20]: 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 [21]: a = 5

In [22]: isinstance(a, int)
Out[22]: 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 [23]: a = 5; b = 4.5

In [24]: isinstance(a, (int, float))
Out[24]: True

In [25]: isinstance(b, (int, float))
Out[25]: 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 that 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.ispace     a.replace     a.splitlines
a.find        a.istitle    a.rfind       a.startswith
```

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

```
In [27]: getattr(a, 'split')
```

```
Out[27]: <function str.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. This is sometimes called “duck typing,” after the saying “If it walks like a duck and quacks like a duck, then it’s a duck.” 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 [29]: isiterable('a string')
```

```
Out[29]: True
```

```
In [30]: isiterable([1, 2, 3])
```



```
Out[30]: True
```

```
In [31]: iterable(5)
```

```
Out[31]: 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 iterable(x):  
    x = list(x)
```

Imports

In Python a *module* is simply a file with the *.py* extension containing Python code. 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 [32]: 5 - 7
```

```
Out[32]: -2
```

```
In [33]: 12 + 21.5
Out[33]: 33.5
```

```
In [34]: 5 <= 2
Out[34]: 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 [35]: a = [1, 2, 3]
```

```
In [36]: b = a
```

```
In [37]: c = list(a)
```

```
In [38]: a is b
Out[38]: True
```

```
In [39]: a is not c
Out[39]: True
```

Since `list` always creates a new Python list (i.e., a copy), we can be sure that `c` is distinct from `a`. Comparing with `is` is not the same as the `==` operator, because in this case we have:

```
In [40]: a == c
Out[40]: 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 [41]: a = None
```

```
In [42]: a is None
Out[42]: True
```

Table 2-3. Binary operators

Operation	Description
<code>a + b</code>	Add a and b
<code>a - b</code>	Subtract b from a
<code>a * b</code>	Multiply a by b
<code>a / b</code>	Divide a by b
<code>a // b</code>	Floor-divide a by b, dropping any fractional remainder
<code>a ** b</code>	Raise a to the b power
<code>a & b</code>	True if both a and b are True; for integers, take the bitwise AND
<code>a b</code>	True if either a or b is True; for integers, take the bitwise OR
<code>a ^ b</code>	For booleans, True if a or b is True, but not both; for integers, take the bitwise EXCLUSIVE-OR

Operation	Description
<code>a == b</code>	True if a equals b
<code>a != b</code>	True if a is not equal to b
<code>a <= b, a < b</code>	True if a is less than (less than or equal) to b
<code>a > b, a >= b</code>	True if a is greater than (greater than or equal) to b
<code>a is b</code>	True if a and b reference the same Python object
<code>a is not b</code>	True if a and b reference different Python objects

Mutable and immutable objects

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

```
In [43]: a_list = ['foo', 2, [4, 5]]
```

```
In [44]: a_list[2] = (3, 4)
```

```
In [45]: a_list
```

```
Out[45]: ['foo', 2, (3, 4)]
```

Others, like strings and tuples, are immutable:

```
In [46]: a_tuple = (3, 5, (4, 5))
```

```
In [47]: a_tuple[1] = 'four'
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-47-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 along with its standard library has a small set of built-in types for handling numerical data, strings, boolean (True or False) values, and dates and time. These “single value” types are sometimes called *scalar types* and we refer to them in this book as scalars. 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
None	The Python “null” value (only one instance of the None object exists)
str	String type; holds Unicode (UTF-8 encoded) strings
bytes	Raw ASCII bytes (or Unicode encoded as bytes)
float	Double-precision (64-bit) floating-point number (note there is no separate double type)
bool	A True or False value
int	Arbitrary precision signed integer

Numeric types

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

```
In [48]: ival = 17239871
```

```
In [49]: ival ** 6
```

```
Out[49]: 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 with scientific notation:

```
In [50]: fval = 7.243
```

```
In [51]: fval2 = 6.78e-5
```

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

```
In [52]: 3 / 2
```

```
Out[52]: 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 [53]: 3 // 2
```

```
Out[53]: 1
```

Strings

Many people use Python for its powerful and flexible built-in string processing capabilities. You can write *string literals* 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 four lines of text; the line breaks after `"""` and after `lines` are included in the string. We can count the new line characters with the `count` method on `c`:

```
In [55]: c.count('\n')
Out[55]: 3
```

Python strings are immutable; you cannot modify a string:

```
In [56]: a = 'this is a string'
```

```
In [57]: a[10] = 'f'
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-57-5ca625d1e504> in <module>()
----> 1 a[10] = 'f'
TypeError: 'str' object does not support item assignment
```

```
In [58]: b = a.replace('string', 'longer string')
```

```
In [59]: b
Out[59]: 'this is a longer string'
```

After this operation, the variable `a` is unmodified:

```
In [60]: a
Out[60]: 'this is a string'
```

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

```
In [61]: a = 5.6
```

```
In [62]: s = str(a)
```

```
In [63]: print(s)
5.6
```

Strings are a sequence of Unicode characters and therefore can be treated like other sequences, such as lists and tuples (which we will explore in more detail in the next chapter):

```
In [64]: s = 'python'
```

```
In [65]: list(s)
Out[65]: ['p', 'y', 't', 'h', 'o', 'n']
```

```
In [66]: s[:3]
Out[66]: 'pyt'
```

The syntax `s[:3]` is called *slicing* and is implemented for many kinds of Python sequences. This will be explained in more detail later on, as it is used extensively in this book.

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 [67]: s = '12\\34'
```

```
In [68]: 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 [69]: s = r'this\has\no\special\characters'
```

```
In [70]: s
Out[70]: 'this\\has\\no\\special\\characters'
```

The `r` stands for *raw*.

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

```
In [71]: a = 'this is the first half '
```

```
In [72]: b = 'and this is the second half'
```

```
In [73]: a + b
Out[73]: '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, and here I will briefly describe the mechanics of one of the main interfaces. String objects have a `format` method that can be used to substitute formatted arguments into the string, producing a new string:

```
In [74]: 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 two decimal places.
- `{1:s}` means to format the second argument as a string.
- `{2:d}` means to format the third argument as an exact integer.

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

```
In [75]: template.format(4.5560, 'Argentine Pesos', 1)
Out[75]: '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 consulting 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 (i.e., 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 [76]: val = "español"

In [77]: val
Out[77]: 'español'
```

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

```
In [78]: val_utf8 = val.encode('utf-8')

In [79]: val_utf8
Out[79]: b'espa\xc3\xb1ol'

In [80]: type(val_utf8)
Out[80]: bytes
```

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

```
In [81]: val_utf8.decode('utf-8')
Out[81]: '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 [82]: val.encode('latin1')
Out[82]: b'espa\xff1ol'

In [83]: val.encode('utf-16')
Out[83]: b'\xff\xfee\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'

In [84]: val.encode('utf-16le')
Out[84]: b'e\x00s\x00p\x00a\x00\xf1\x00o\x00l\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.

Though you may seldom need to do so, you can define your own byte literals by prefixing a string with `b`:

```
In [85]: bytes_val = b'this is bytes'

In [86]: bytes_val
Out[86]: b'this is bytes'

In [87]: decoded = bytes_val.decode('utf8')

In [88]: decoded # this is str (Unicode) now
Out[88]: '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` or `or` keywords:

```
In [89]: True and True
Out[89]: True

In [90]: False or True
Out[90]: True
```

Type casting

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

```
In [91]: s = '3.14159'

In [92]: fval = float(s)

In [93]: type(fval)
Out[93]: float

In [94]: int(fval)
Out[94]: 3

In [95]: bool(fval)
Out[95]: True

In [96]: bool(0)
Out[96]: False
```


None

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

```
In [97]: a = None
```

```
In [98]: a is None
Out[98]: True
```

```
In [99]: b = 5
```

```
In [100]: b is not None
Out[100]: True
```

None is also a common default value for 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 [101]: type(None)
Out[101]: 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 [102]: from datetime import datetime, date, time
```

```
In [103]: dt = datetime(2011, 10, 29, 20, 30, 21)
```

```
In [104]: dt.day
Out[104]: 29
```

```
In [105]: dt.minute
Out[105]: 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 [106]: dt.date()
Out[106]: datetime.date(2011, 10, 29)
```

```
In [107]: dt.time()
Out[107]: datetime.time(20, 30, 21)
```

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

```
In [108]: dt.strftime('%m/%d/%Y %H:%M')
Out[108]: '10/29/2011 20:30'
```

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

```
In [109]: datetime.strptime('20091031', '%Y%m%d')
Out[109]: datetime.datetime(2009, 10, 31, 0, 0)
```

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

When you are 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 [110]: dt.replace(minute=0, second=0)
Out[110]: 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 [111]: dt2 = datetime(2011, 11, 15, 22, 30)
```

```
In [112]: delta = dt2 - dt
```

```
In [113]: delta
Out[113]: datetime.timedelta(17, 7179)
```

```
In [114]: type(delta)
Out[114]: datetime.timedelta
```

The output `timedelta(17, 7179)` indicates that the `timedelta` encodes an offset of 17 days and 7,179 seconds.

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

```
In [115]: dt
Out[115]: datetime.datetime(2011, 10, 29, 20, 30, 21)
```

```
In [116]: dt + delta
Out[116]: datetime.datetime(2011, 11, 15, 22, 30)
```

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

Type	Description
%Y	Four-digit year
%y	Two-digit year

Type	Description
%m	Two-digit month [01, 12]
%d	Two-digit day [01, 31]
%H	Hour (24-hour clock) [00, 23]
%I	Hour (12-hour clock) [01, 12]
%M	Two-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 (e.g., 2012-4-18)
%D	Shortcut for %m/%d/%y (e.g., 04/18/12)

Control Flow

Python has several built-in keywords 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 that, 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 [117]: a = 5; b = 7
In [118]: c = 8; d = 4
```

```
In [119]: 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 [120]: 4 > 3 > 2 > 1
Out[120]: 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
```

You can advance a for loop 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 with 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 [121]: 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!')
```

range

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

```
In [122]: range(10)
Out[122]: range(0, 10)
```

```
In [123]: list(range(10))
Out[123]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Both a start, end, and step (which may be negative) can be given:

```
In [124]: list(range(0, 20, 2))
Out[124]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
In [125]: list(range(5, 0, -1))
Out[125]: [5, 4, 3, 2, 1]
```

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 99,999 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 to combine an `if-else` block that 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 [126]: x = 5

In [127]: 'Non-negative' if x >= 0 else 'Negative'
Out[127]: '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 as the true and false expressions are very complex.

Built-in Data Structures, Functions, and Files

This chapter discusses capabilities built into the Python language that 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.

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

3.1 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 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 you're 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))
```

You can convert any sequence or iterator 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. As in 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 the tuple is 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
```

If an object inside a tuple is mutable, such as a list, you can modify it in-place:

```
In [11]: tup[1].append(3)
```

```
In [12]: tup
```

```
Out[12]: ('foo', [1, 2, 3], True)
```

You can concatenate tuples 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 righthand 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 you can easily swap variable names, a task which in many languages might look like:

```
tmp = a
```

```
a = b
```

```
b = tmp
```

But, in Python, the swap can be done like this:

```
In [21]: a, b = 1, 2
```

```
In [22]: a
```

```
Out[22]: 1
```

```
In [23]: b
```

```
Out[23]: 2
```

```
In [24]: b, a = a, b
```

```
In [25]: a
```

```
Out[25]: 2
```

```
In [26]: b
```

```
Out[26]: 1
```

A common use of variable unpacking is iterating over sequences of tuples or lists:

```
In [27]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
In [28]: 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 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 [29]: values = 1, 2, 3, 4, 5
```

```
In [30]: a, b, *rest = values
```

```
In [31]: a, b  
Out[31]: (1, 2)
```

```
In [32]: rest  
Out[32]: [3, 4, 5]
```

This `rest` bit is sometimes something you want to discard; there is nothing special about the `rest` name. As a matter of convention, many Python programmers will use the underscore (`_`) for unwanted variables:

```
In [33]: a, b, *_ = values
```

Tuple methods

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

```
In [34]: a = (1, 2, 2, 2, 3, 4, 2)
```

```
In [35]: a.count(2)  
Out[35]: 4
```

List

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

```
In [36]: a_list = [2, 3, 7, None]
```

```
In [37]: tup = ('foo', 'bar', 'baz')
```

```
In [38]: b_list = list(tup)
```

```
In [39]: b_list  
Out[39]: ['foo', 'bar', 'baz']
```

```
In [40]: b_list[1] = 'peekaboo'
```

```
In [41]: b_list
Out[41]: ['foo', 'peekaboo', 'baz']
```

Lists and tuples are semantically similar (though tuples cannot be modified) and 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 [42]: gen = range(10)

In [43]: gen
Out[43]: range(0, 10)

In [44]: list(gen)
Out[44]: [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 [45]: b_list.append('dwarf')

In [46]: b_list
Out[46]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

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

```
In [47]: b_list.insert(1, 'red')

In [48]: b_list
Out[48]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

The insertion index must be between 0 and the length of the list, inclusive.



`insert` is computationally expensive compared with `append`, because 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 wish to 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 [49]: b_list.pop(2)
Out[49]: 'peekaboo'

In [50]: b_list
Out[50]: ['foo', 'red', 'baz', 'dwarf']
```

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

```
In [51]: b_list.append('foo')

In [52]: b_list
Out[52]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [53]: b_list.remove('foo')

In [54]: b_list
Out[54]: ['red', 'baz', 'dwarf', 'foo']
```

If performance is not a concern, by using `append` and `remove`, you can use a Python list as a perfectly suitable “multiset” data structure.

Check if a list contains a value using the `in` keyword:

```
In [55]: 'dwarf' in b_list
Out[55]: True
```

The keyword `not` can be used to negate `in`:

```
In [56]: 'dwarf' not in b_list
Out[56]: False
```

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

Concatenating and combining lists

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

```
In [57]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[57]: [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 [58]: x = [4, None, 'foo']

In [59]: x.extend([7, 8, (2, 3)])

In [60]: x
Out[60]: [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

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

```
In [61]: a = [7, 2, 5, 1, 3]
```

```
In [62]: a.sort()
```

```
In [63]: a
```

```
Out[63]: [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*—that is, 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 [64]: b = ['saw', 'small', 'He', 'foxes', 'six']
```

```
In [65]: b.sort(key=len)
```

```
In [66]: b
```

```
Out[66]: ['He', 'saw', 'six', 'small', 'foxes']
```

Soon, we'll look at the `sorted` function, which can produce a sorted copy of a general sequence.

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 [67]: import bisect
```

```
In [68]: c = [1, 2, 2, 2, 3, 4, 7]
```

```
In [69]: bisect.bisect(c, 2)
```

```
Out[69]: 4
```

```
In [70]: bisect.bisect(c, 5)
```

```
Out[70]: 6
```

```
In [71]: bisect.insort(c, 6)
```

```
In [72]: c
```

```
Out[72]: [1, 2, 2, 2, 3, 4, 6, 7]
```



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 most sequence types by using slice notation, which in its basic form consists of `start:stop` passed to the indexing operator `[]`:

```
In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]
```

```
In [74]: seq[1:5]
```

```
Out[74]: [2, 3, 7, 5]
```

Slices can also be assigned to with a sequence:

```
In [75]: seq[3:4] = [6, 3]
```

```
In [76]: seq
```

```
Out[76]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

While the 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 [77]: seq[:5]
```

```
Out[77]: [7, 2, 3, 6, 3]
```

```
In [78]: seq[3:]
```

```
Out[78]: [6, 3, 5, 6, 0, 1]
```

Negative indices slice the sequence relative to the end:

```
In [79]: seq[-4:]
```

```
Out[79]: [5, 6, 0, 1]
```

```
In [80]: seq[-6:-2]
```

```
Out[80]: [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 illustration of slicing with positive and negative integers. In the figure, the indices are shown at the “bin edges” to help show where the slice selections start and stop using positive or negative indices.

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

```
In [81]: seq[::2]
Out[81]: [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 [82]: seq[::-1]
Out[82]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

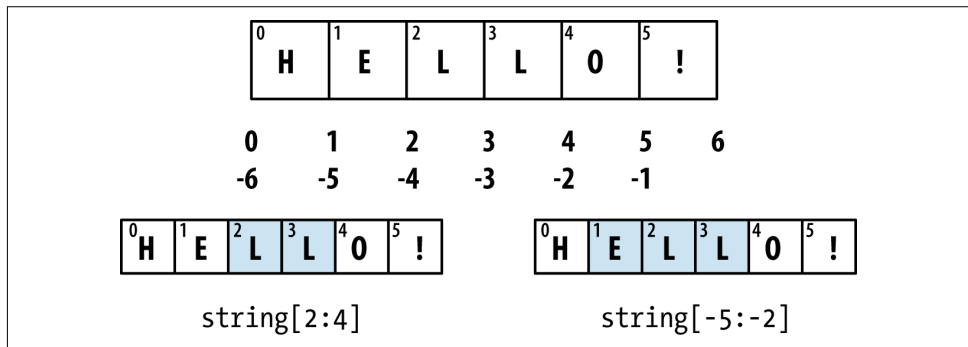


Figure 3-1. Illustration of Python slicing conventions

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 you are indexing data, a helpful 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 [83]: some_list = ['foo', 'bar', 'baz']
```

```
In [84]: mapping = {}
```



```
In [85]: for i, v in enumerate(some_list):
        ....:     mapping[v] = i
```

```
In [86]: mapping
Out[86]: {'bar': 1, 'baz': 2, 'foo': 0}
```

sorted

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

```
In [87]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[87]: [0, 1, 2, 2, 3, 6, 7]
```

```
In [88]: sorted('horse race')
Out[88]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

The `sorted` function accepts the same arguments as the `sort` method on lists.

zip

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

```
In [89]: seq1 = ['foo', 'bar', 'baz']
```

```
In [90]: seq2 = ['one', 'two', 'three']
```

```
In [91]: zipped = zip(seq1, seq2)
```

```
In [92]: list(zipped)
```

```
Out[92]: [('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 [93]: seq3 = [False, True]
```

```
In [94]: list(zip(seq1, seq2, seq3))
```

```
Out[94]: [('foo', 'one', False), ('bar', 'two', True)]
```

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

```
In [95]: 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 [96]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
....:                ('Schilling', 'Curt')]
```

```
In [97]: first_names, last_names = zip(*pitchers)
```

```
In [98]: first_names
Out[98]: ('Nolan', 'Roger', 'Schilling')
```

```
In [99]: last_names
Out[99]: ('Ryan', 'Clemens', 'Curt')
```

reversed

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

```
In [100]: list(reversed(range(10)))
Out[100]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Keep in mind that `reversed` is a generator (to be discussed in some more detail later), so it does not create the reversed sequence 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 approach for creating one is to use curly braces `{}` and colons to separate keys and values:

```
In [101]: empty_dict = {}
```

```
In [102]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}
```

```
In [103]: d1
Out[103]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

You can access, insert, or set elements using the same syntax as for accessing elements of a list or tuple:

```
In [104]: d1[7] = 'an integer'
```

```
In [105]: d1
Out[105]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

```
In [106]: d1['b']
Out[106]: [1, 2, 3, 4]
```

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

```
In [107]: 'b' in d1
Out[107]: True
```

You can delete values either using the `del` keyword or the `pop` method (which simultaneously returns the value and deletes the key):

```
In [108]: d1[5] = 'some value'
```

```
In [109]: d1
Out[109]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value'}
```

```
In [110]: d1['dummy'] = 'another value'
```

```
In [111]: d1
Out[111]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value',
 'dummy': 'another value'}
```

```
In [112]: del d1[5]
```

```
In [113]: d1
Out[113]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 'dummy': 'another value'}
```

```
In [114]: ret = d1.pop('dummy')
```

```
In [115]: ret
Out[115]: 'another value'
```

```
In [116]: d1
Out[116]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

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 [117]: list(d1.keys())
Out[117]: ['a', 'b', 7]
```

```
In [118]: list(d1.values())
Out[118]: ['some value', [1, 2, 3, 4], 'an integer']
```

You can merge one dict into another using the update method:

```
In [119]: d1.update({'b' : 'foo', 'c' : 12})

In [120]: d1
Out[120]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

The update method changes dicts in-place, so any existing keys in the data passed to update will have their old values discarded.

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, the `dict` function accepts a list of 2-tuples:

```
In [121]: mapping = dict(zip(range(5), reversed(range(5))))

In [122]: mapping
Out[122]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Later 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 [123]: words = ['apple', 'bat', 'bar', 'atom', 'book']
```

```
In [124]: by_letter = {}
```

```
In [125]: for word in words:
.....:     letter = word[0]
.....:     if letter not in by_letter:
.....:         by_letter[letter] = [word]
.....:     else:
.....:         by_letter[letter].append(word)
.....:
```

```
In [126]: by_letter
```

```
Out[126]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

The setdefault dict method is for precisely this purpose. The preceding for loop can be rewritten as:

```
for word in words:
    letter = word[0]
    by_letter.setdefault(letter, []).append(word)
```

The built-in collections module has a useful class, `defaultdict`, which makes this even easier. To create one, you pass 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)
```

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 [127]: hash('string')
Out[127]: 5023931463650008331
```

```
In [128]: hash((1, 2, (2, 3)))
Out[128]: 1097636502276347782
```

```
In [129]: hash((1, 2, [2, 3])) # fails because lists are mutable
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-129-800cd14ba8be> 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 [130]: d = {}
```

```
In [131]: d[tuple([1, 2, 3])] = 5
```

```
In [132]: d
```

```
Out[132]: {(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 via a *set literal* with curly braces:

```
In [133]: set([2, 2, 2, 1, 3, 3])
```

```
Out[133]: {1, 2, 3}
```

```
In [134]: {2, 2, 2, 1, 3, 3}
```

```
Out[134]: {1, 2, 3}
```

Sets support mathematical *set operations* like union, intersection, difference, and symmetric difference. Consider these two example sets:

```
In [135]: a = {1, 2, 3, 4, 5}
```

```
In [136]: b = {3, 4, 5, 6, 7, 8}
```

The union of these two sets is the set of distinct elements occurring in either set. This can be computed with either the union method or the | binary operator:

```
In [137]: a.union(b)
```

```
Out[137]: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
In [138]: a | b
```

```
Out[138]: {1, 2, 3, 4, 5, 6, 7, 8}
```

The intersection contains the elements occurring in both sets. The & operator or the intersection method can be used:

```
In [139]: a.intersection(b)
```

```
Out[139]: {3, 4, 5}
```

```
In [140]: a & b
```

```
Out[140]: {3, 4, 5}
```

See [Table 3-1](#) for a list of commonly used set methods.

Table 3-1. Python set operations

Function	Alternative syntax	Description
<code>a.add(x)</code>	N/A	Add element <code>x</code> to the set <code>a</code>
<code>a.clear()</code>	N/A	Reset the set <code>a</code> 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 & 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 &= 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

All 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 may be more efficient:

```
In [141]: c = a.copy()
```

```
In [142]: c |= b
```

```
In [143]: c
```

```
Out[143]: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
In [144]: d = a.copy()
```

```
In [145]: d &= b
```

```
In [146]: d
```

```
Out[146]: {3, 4, 5}
```

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

```
In [147]: my_data = [1, 2, 3, 4]
```

```
In [148]: my_set = {tuple(my_data)}
```

```
In [149]: my_set
```

```
Out[149]: {(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 [150]: a_set = {1, 2, 3, 4, 5}
```

```
In [151]: {1, 2, 3}.issubset(a_set)
```

```
Out[151]: True
```

```
In [152]: a_set.issuperset({1, 2, 3})
```

```
Out[152]: True
```

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

```
In [153]: {1, 2, 3} == {3, 2, 1}
```

```
Out[153]: True
```

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, 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 [154]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
```

```
In [155]: [x.upper() for x in strings if len(x) > 2]
```

```
Out[155]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Set and dict comprehensions are a natural extension, producing sets and dicts in an 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 from before. Suppose we wanted a set containing just the lengths of the strings contained in the collection; we could easily compute this using a set comprehension:

```
In [156]: unique_lengths = {len(x) for x in strings}
```

```
In [157]: unique_lengths
Out[157]: {1, 2, 3, 4, 6}
```

We could also express this more functionally using the map function, introduced shortly:

```
In [158]: set(map(len, strings))
Out[158]: {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 [159]: loc_mapping = {val : index for index, val in enumerate(strings)}
```

```
In [160]: loc_mapping
Out[160]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Nested list comprehensions

Suppose we have a list of lists containing some English and Spanish names:

```
In [161]: all_data = [['John', 'Emily', 'Michael', 'Mary', 'Steven'],
.....:                ['Maria', 'Juan', 'Javier', 'Natalia', 'Pilar']]
```

You might have gotten these names from a couple of files and decided to organize them by language. 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 [162]: result = [name for names in all_data for name in names
.....:                if name.count('e') >= 2]
```

```
In [163]: result
Out[163]: ['Steven']
```

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 [164]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
In [165]: flattened = [x for tup in some_tuples for x in tup]
In [166]: flattened
Out[166]: [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 syntax just shown from a list comprehension inside a list comprehension, which is also perfectly valid:

```
In [167]: [[x for x in tup] for tup in some_tuples]
Out[167]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

This produces a list of lists, rather than a flattened list of all of the inner elements.

3.2 Functions

Functions are the primary and most important method of code organization and reuse in Python. As a rule of thumb, if you anticipate needing to repeat the same or very similar code more than once, it may be worth writing a reusable function. Functions can also help make your code more readable by giving a name to a group of Python statements.

Functions are declared with the `def` keyword and returned from with 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 Python reaches the end of a function without encountering a return statement, None is returned automatically.

Each function can have *positional* arguments and *keyword* arguments. Keyword arguments are most commonly used to specify default values or optional arguments. In the preceding function, x and y are positional arguments while z is a keyword argument. This means that the function can be called in any of these ways:

```
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
my_function(10, 20)
```

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.



It is possible to use keywords for passing positional arguments as well. In the preceding example, we could also have written:

```
my_function(x=5, y=6, z=7)
my_function(y=6, x=5, z=7)
```

In some cases this can help with readability.

Namespaces, Scope, and Local Functions

Functions can access variables in two different scopes: *global* and *local*. An alternative 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 that are outside the purview of this chapter). Consider the following function:

```
def func():
    a = []
    for i in range(5):
        a.append(i)
```

When func() is called, the empty list a is created, five elements are appended, and then a is destroyed when the function exits. Suppose instead we had declared a as follows:

```
a = []
def func():
    for i in range(5):
        a.append(i)
```

Assigning variables outside of the function's scope is possible, but those variables must be declared as global via the `global` keyword:

```
In [168]: a = None

In [169]: def bind_a_variable():
.....:     global a
.....:     a = []
.....:     bind_a_variable()
.....:

In [170]: print(a)
[]
```



I generally discourage use of the `global` keyword. Typically global variables are used to store some kind of state in a system. If you find yourself using a lot of them, it may indicate a need for object-oriented programming (using classes).

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 with simple syntax. Here's an 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 may find yourself doing this often. 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 preceding example, we could have done this 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 before might be to return a dict instead:

```
def f():
    a = 5
    b = 6
    c = 7
    return {'a' : a, 'b' : b, 'c' : c}
```

This alternative technique can be useful depending on what you are trying to do.

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:

```
In [171]: states = [' Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FlOrIda',
.....:              'south carolina##', 'West virginia?']
```

Anyone who has ever worked with user-submitted survey data has seen messy results like these. Lots of things need to happen to make this list of strings uniform and ready for analysis: stripping whitespace, removing punctuation symbols, and standardizing on proper capitalization. One way to do this is to use built-in string methods along with the `re` standard library module for regular expressions:

```
import re

def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub('[!#?]', '', value)
        value = value.title()
        result.append(value)
    return result
```

The result looks like this:

```
In [173]: clean_strings(states)
Out[173]:
['Alabama',
'Georgia',
'Georgia',
'Georgia',
'Florida',
'South Carolina',
'West Virginia']
```

An alternative 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 the following:

```
In [175]: clean_strings(states, clean_ops)
Out[175]:
['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 and generic.

You can use functions as arguments to other functions like the built-in `map` function, which applies a function to a sequence of some kind:

```
In [176]: for x in map(remove_punctuation, states):
.....:     print(x)
Alabama
Georgia
Georgia
georgia
FLOrIda
south carolina
West virginia
```

Anonymous (Lambda) Functions

Python has support for so-called *anonymous* or *lambda* functions, which are a way of writing functions consisting of a single statement, the result of which is the return value. They are defined with 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 [177]: strings = ['foo', 'card', 'bar', 'aaa', 'abab']
```

Here we could pass a lambda function to the list's `sort` method:

```
In [178]: strings.sort(key=lambda x: len(set(list(x))))
```

```
In [179]: strings  
Out[179]: ['aaa', 'foo', 'abab', 'bar', 'card']
```



One reason lambda functions are called anonymous functions is that, unlike functions declared with the `def` keyword, the function object itself is never given an explicit `__name__` attribute.

Currying: Partial Argument Application

Currying is computer science jargon (named after the mathematician Haskell Curry) that 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 all we've really done is define 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 [180]: some_dict = {'a': 1, 'b': 2, 'c': 3}

In [181]: 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 [182]: dict_iterator = iter(some_dict)

In [183]: dict_iterator
Out[183]: <dict_keyiterator at 0x7fbbd5a9f908>
```

An 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 [184]: list(dict_iterator)
Out[184]: ['a', 'b', 'c']
```

A *generator* is a concise way to construct a new iterable object. Whereas normal functions execute and return a single result at a time, generators return a sequence of multiple results 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 {0}'.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 [186]: gen = squares()

In [187]: gen
Out[187]: <generator object squares at 0x7fbbd5ab4570>
```

It is not until you request elements from the generator that it begins executing its code:


```
In [188]: for x in gen:
.....:     print(x, end=' ')
Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

Generator expressions

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 within parentheses instead of brackets:

```
In [189]: gen = (x ** 2 for x in range(100))

In [190]: gen
Out[190]: <generator object <genexpr> at 0x7fbbd5ab29e8>
```

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 instead of list comprehensions as function arguments in many cases:

```
In [191]: sum(x ** 2 for x in range(100))
Out[191]: 328350

In [192]: dict((i, i **2) for i in range(5))
Out[192]: {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, grouping consecutive elements in the sequence by return value of the function. Here's an example:

```
In [193]: import itertools

In [194]: first_letter = lambda x: x[0]

In [195]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']

In [196]: for letter, names in itertools.groupby(names, first_letter):
.....:     print(letter, list(names)) # names is a generator
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 <code>k</code> -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 <code>k</code> -tuples of elements in the iterable, respecting order
<code>groupby(iterable[, keyfunc])</code>	Generates (key, sub-iterator) for each unique key
<code>product(*iterables, repeat=1)</code>	Generates the Cartesian product of the input iterables as tuples, similar to a nested for loop

Errors and 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 [197]: float('1.2345')
Out[197]: 1.2345
```

```
In [198]: float('something')
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-198-439904410854> 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 [200]: attempt_float('1.2345')
Out[200]: 1.2345
```

```
In [201]: attempt_float('something')
Out[201]: 'something'
```

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

```
In [202]: float((1, 2))
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-202-842079ebb635> 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 [204]: attempt_float((1, 2))
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-204-9bdfd730cead> in <module>()
----> 1 attempt_float((1, 2))
<ipython-input-203-3e06b8379b6b> 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()
```

Exceptions in IPython

If an exception is raised while you are %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 [10]: %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). You can control the amount of context shown 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.

3.3 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 one reason 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 [207]: path = 'examples/segismundo.txt'
```

```
In [208]: 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 [209]: lines = [x.rstrip() for x in open(path)]
```

```
In [210]: lines
```

```
Out[210]:
['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.',
 '']
```

When you use `open` to create file objects, it is important to explicitly close the file when you are finished with it. Closing the file releases its resources back to the operating system:

```
In [211]: f.close()
```

One of the ways to make it easier to clean up open files is to use the `with` statement:

```
In [212]: with open(path) as f:
.....:     lines = [x.rstrip() for x in f]
```

This will automatically close the file `f` when exiting the `with` block.

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 writable file but fails if the file path already exists. See [Table 3-3](#) 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 [213]: f = open(path)
```

```
In [214]: f.read(10)
Out[214]: 'Sueña e l r'
```

```
In [215]: f2 = open(path, 'rb') # Binary mode
```

```
In [216]: f2.read(10)
Out[216]: b'Sue\xc3\xb1a e l '
```

The `read` method advances the file handle’s position by the number of bytes read. `tell` gives you the current position:

```
In [217]: f.tell()
Out[217]: 11
```

```
In [218]: f2.tell()
Out[218]: 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 encoding. You can check the default encoding in the `sys` module:

```
In [219]: import sys
```

```
In [220]: sys.getdefaultencoding()
Out[220]: 'utf-8'
```

`seek` changes the file position to the indicated byte in the file:

```
In [221]: f.seek(3)
Out[221]: 3
```

```
In [222]: f.read(1)
Out[222]: 'ñ'
```

Lastly, we remember to close the files:

```
In [223]: f.close()
```

```
In [224]: f2.close()
```

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 the file if it does not already exist)
r+	Read and write
b	Add to mode for binary files (i.e., '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 (i.e., 'rt' or 'xt')

To write text to a file, you can use 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 [225]: with open('tmp.txt', 'w') as handle:
.....:     handle.writelines(x for x in open(path) if len(x) > 1)
```

```
In [226]: with open('tmp.txt') as f:
.....:     lines = f.readlines()
```

```
In [227]: lines
```

```
Out[227]:
```

```
['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>write(str)</code>	Write passed string to file

Method	Description
writelines(strings)	Write passed sequence of strings to the file
close()	Close the handle
flush()	Flush the internal I/O buffer to disk
seek(pos)	Move to indicated file position (integer)
tell()	Return current file position as integer
closed	True if the file is closed

Bytes and Unicode with Files

The default behavior for Python files (whether readable or writable) 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 [230]: with open(path) as f:
.....:     chars = f.read(10)
```

```
In [231]: chars
Out[231]: '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 [232]: with open(path, 'rb') as f:
.....:     data = f.read(10)
```

```
In [233]: data
Out[233]: 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 is fully formed:

```
In [234]: data.decode('utf8')
Out[234]: 'Sueña el '
```

```
In [235]: data[:4].decode('utf8')
```

```
-----
UnicodeDecodeError                                Traceback (most recent call last)
<ipython-input-235-300e0af10bb7> in <module>()
----> 1 data[:4].decode('utf8')
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in position 3: unexpected end of data
```

Text mode, combined with the encoding option of `open`, provides a convenient way to convert from one Unicode encoding to another:


```
In [236]: sink_path = 'sink.txt'
```

```
In [237]: with open(path) as source:
```

```
.....:     with open(sink_path, 'xt', encoding='iso-8859-1') as sink:
.....:         sink.write(source.read())
```

```
In [238]: with open(sink_path, encoding='iso-8859-1') as f:
```

```
.....:     print(f.read(10))
```

```
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 [240]: f = open(path)
```

```
In [241]: f.read(5)
```

```
Out[241]: 'Sueña'
```

```
In [242]: f.seek(4)
```

```
Out[242]: 4
```

```
In [243]: f.read(1)
```

```
-----
UnicodeDecodeError                                Traceback (most recent call last)
<ipython-input-243-7841103e33f5> in <module>()
----> 1 f.read(1)
/miniconda/envs/book-env/lib/python3.6/codecs.py in decode(self, input, final)
    319     # decode input (taking the buffer into account)
    320     data = self.buffer + input
--> 321     (result, consumed) = self._buffer_decode(data, self.errors, final)
    )
    322     # keep undecoded input until the next call
    323     self.buffer = data[consumed:]
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in position 0: invalid s
tart byte
```

```
In [244]: f.close()
```

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.

3.4 Conclusion

With some of the basics and the Python environment and language now under our belt, it's time to move on and learn about NumPy and array-oriented computing in Python.

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. Since

NumPy is a large topic, I will cover many advanced NumPy features like broadcasting in more depth later (see [Appendix A](#)).

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 datasets
- Expressing conditional logic as array expressions instead of loops with `if-elif-else` branches
- Group-wise data manipulations (aggregation, transformation, function application)

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.



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 the community together 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 one million integers, and the equivalent Python list:

```
In [7]: import numpy as np

In [8]: my_arr = np.arange(1000000)

In [9]: my_list = list(range(1000000))
```

Now let's multiply each sequence by 2:

```
In [10]: %time for _ in range(10): my_arr2 = my_arr * 2
CPU times: user 20 ms, sys: 50 ms, total: 70 ms
Wall time: 72.4 ms

In [11]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
CPU times: user 760 ms, sys: 290 ms, total: 1.05 s
Wall time: 1.05 s
```

NumPy-based algorithms are generally 10 to 100 times faster (or more) than their pure Python counterparts and use significantly less memory.

4.1 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 datasets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

To give you a flavor of how NumPy enables batch computations with similar syntax to scalar values on built-in Python objects, I first import NumPy and generate a small array of random data:

```
In [12]: import numpy as np

# Generate some random data
In [13]: data = np.random.randn(2, 3)

In [14]: data
Out[14]:
array([[ -0.2047,  0.4789, -0.5194],
       [-0.5557,  1.9658,  1.3934]])
```

I then write mathematical operations with data:

```
In [15]: data * 10
Out[15]:
array([[ -2.0471,  4.7894, -5.1944],
       [-5.5573, 19.6578, 13.9341]])

In [16]: data + data
Out[16]:
```

```
array([[ -0.4094,  0.9579, -1.0389],
       [-1.1115,  3.9316,  2.7868]])
```

In the first example, all of the elements have been multiplied by 10. In the second, the corresponding values in each “cell” in the array have been added to each other.



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 against making a habit of this. The `numpy` namespace is 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 [17]: data.shape
Out[17]: (2, 3)
```

```
In [18]: data.dtype
Out[18]: 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.



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 [19]: data1 = [6, 7.5, 8, 0, 1]
```

```
In [20]: arr1 = np.array(data1)
```

```
In [21]: arr1
Out[21]: array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
```

```
In [23]: arr2 = np.array(data2)
```

```
In [24]: arr2
```

```
Out[24]:
```

```
array([[1, 2, 3, 4],  
       [5, 6, 7, 8]])
```

Since `data2` was a list of lists, the NumPy array `arr2` has two dimensions with shape inferred from the data. We can confirm this by inspecting the `ndim` and `shape` attributes:

```
In [25]: arr2.ndim
```

```
Out[25]: 2
```

```
In [26]: arr2.shape
```

```
Out[26]: (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 previous two examples we have:

```
In [27]: arr1.dtype
```

```
Out[27]: dtype('float64')
```

```
In [28]: arr2.dtype
```

```
Out[28]: 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 0s or 1s, 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 [29]: np.zeros(10)
```

```
Out[29]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

```
In [30]: np.zeros((3, 6))
```

```
Out[30]:
```

```
array([[ 0.,  0.,  0.,  0.,  0.,  0.],  
       [ 0.,  0.,  0.,  0.,  0.,  0.],  
       [ 0.,  0.,  0.,  0.,  0.,  0.]])
```

```
In [31]: np.empty((2, 3, 2))
```

```
Out[31]:
```

```
array([[ [ 0.,  0.],  
        [ 0.,  0.],  
        [ 0.,  0.]])
```

```
[[ 0.,  0.],  
 [ 0.,  0.],  
 [ 0.,  0.]])
```



It's not safe to assume that `np.empty` will return an array of all zeros. In some cases, it may return uninitialized “garbage” values.

`arange` is an array-valued version of the built-in Python `range` function:

```
In [32]: np.arange(15)  
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

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>array</code>	Convert input data (list, tuple, array, or other sequence type) to 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> , <code>ones_like</code>	Produce an array of all 1s with the given shape and dtype; <code>ones_like</code> takes another array and produces a ones array of the same shape and dtype
<code>zeros</code> , <code>zeros_like</code>	Like <code>ones</code> and <code>ones_like</code> but producing arrays of 0s instead
<code>empty</code> , <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> , <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 filled array of the same shape and dtype
<code>eye</code> , <code>identity</code>	Create a square $N \times N$ identity matrix (1s on the diagonal and 0s 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 [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)
```

```
In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)
```

```
In [35]: arr1.dtype  
Out[35]: dtype('float64')
```

```
In [36]: arr2.dtype
Out[36]: 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.



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 datasets, it is good to know that you have control over the storage type.

Table 4-2. NumPy data types

Type	Type code	Description
<code>int8</code> , <code>uint8</code>	<code>i1</code> , <code>u1</code>	Signed and unsigned 8-bit (1 byte) integer types
<code>int16</code> , <code>uint16</code>	<code>i2</code> , <code>u2</code>	Signed and unsigned 16-bit integer types
<code>int32</code> , <code>uint32</code>	<code>i4</code> , <code>u4</code>	Signed and unsigned 32-bit integer types
<code>int64</code> , <code>uint64</code>	<code>i8</code> , <code>u8</code>	Signed and unsigned 64-bit integer types
<code>float16</code>	<code>f2</code>	Half-precision floating point
<code>float32</code>	<code>f4</code> or <code>f</code>	Standard single-precision floating point; compatible with C <code>float</code>
<code>float64</code>	<code>f8</code> or <code>d</code>	Standard double-precision floating point; compatible with C double and Python <code>float</code> object
<code>float128</code>	<code>f16</code> or <code>g</code>	Extended-precision floating point
<code>complex64</code> , <code>complex128</code> , <code>complex256</code>	<code>c8</code> , <code>c16</code> , <code>c32</code>	Complex numbers represented by two 32, 64, or 128 floats, respectively
<code>bool</code>	<code>?</code>	Boolean type storing <code>True</code> and <code>False</code> values
<code>object</code>	<code>0</code>	Python object type; a value can be any Python object
<code>string_</code>	<code>S</code>	Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use <code>'S10'</code>
<code>unicode_</code>	<code>U</code>	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 [37]: arr = np.array([1, 2, 3, 4, 5])
In [38]: arr.dtype
Out[38]: dtype('int64')
In [39]: float_arr = arr.astype(np.float64)
In [40]: float_arr.dtype
Out[40]: 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 [41]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
In [42]: arr
Out[42]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])
In [43]: arr.astype(np.int32)
Out[43]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

If you have an array of strings representing numbers, you can use `astype` to convert them to numeric form:

```
In [44]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
In [45]: numeric_strings.astype(float)
Out[45]: array([ 1.25, -9.6,  42.  ])
```



It's important to be cautious when using the `numpy.string_` type, as string data in NumPy is fixed size and may truncate input without warning. pandas has more intuitive out-of-the-box behavior on non-numeric data.

If casting were to fail for some reason (like a string that cannot be converted to `float64`), a `ValueError` will be raised. Here 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 [46]: int_array = np.arange(10)
In [47]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
In [48]: int_array.astype(calibers.dtype)
Out[48]: 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 [49]: empty_uint32 = np.empty(8, dtype='u4')

In [50]: empty_uint32
Out[50]:
array([          0, 1075314688,           0, 1075707904,           0,
        1075838976,           0, 1072693248], dtype=uint32)
```



Calling `astype` *always* creates a new array (a copy of the data), even if the new dtype is the same as the old dtype.

Arithmetic with NumPy Arrays

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 element-wise:

```
In [51]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])
```

```
In [52]: arr
Out[52]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

```
In [53]: arr * arr
Out[53]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

```
In [54]: arr - arr
Out[54]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Arithmetic operations with scalars propagate the scalar argument to each element in the array:

```
In [55]: 1 / arr
Out[55]:
array([[ 1.    ,  0.5   ,  0.3333],
       [ 0.25  ,  0.2   ,  0.1667]])
```

```
In [56]: arr ** 0.5
Out[56]:
array([[ 1.    ,  1.4142,  1.7321],
       [ 2.    ,  2.2361,  2.4495]])
```

Comparisons between arrays of the same size yield boolean arrays:

```
In [57]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
```

```
In [58]: arr2
```

```
Out[58]:  
array([[ 0.,  4.,  1.],  
       [ 7.,  2., 12.]])
```

```
In [59]: arr2 > arr
```

```
Out[59]:  
array([[False,  True, False],  
       [ True, False,  True]], dtype=bool)
```

Operations between differently sized arrays is called *broadcasting* and will be discussed in more detail in [Appendix A](#). 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 [60]: arr = np.arange(10)
```

```
In [61]: arr
```

```
Out[61]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [62]: arr[5]
```

```
Out[62]: 5
```

```
In [63]: arr[5:8]
```

```
Out[63]: array([5, 6, 7])
```

```
In [64]: arr[5:8] = 12
```

```
In [65]: arr
```

```
Out[65]: 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.

To give an example of this, I first create a slice of `arr`:

```
In [66]: arr_slice = arr[5:8]
```

```
In [67]: arr_slice
```

```
Out[67]: array([12, 12, 12])
```

Now, when I change values in `arr_slice`, the mutations are reflected in the original array `arr`:

```
In [68]: arr_slice[1] = 12345
In [69]: arr
Out[69]: array([ 0,  1,  2,  3,  4, 12, 12345, 12,  8,
  9])
```

The “bare” slice `[:]` will assign to all values in an array:

```
In [70]: arr_slice[:] = 64
In [71]: arr
Out[71]: 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 that 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.



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 [72]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
In [73]: arr2d[2]
Out[73]: 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 [74]: arr2d[0][2]
Out[74]: 3
In [75]: arr2d[0, 2]
Out[75]: 3
```

See [Figure 4-1](#) for an illustration of indexing on a two-dimensional array. I find it helpful to think of axis 0 as the “rows” of the array and axis 1 as the “columns.”

		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 [76]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

```
In [77]: arr3d
```

```
Out[77]:
```

```
array([[[ 1,  2,  3],
         [ 4,  5,  6]],
       [[ 7,  8,  9],
         [10, 11, 12]]])
```

`arr3d[0]` is a 2×3 array:

```
In [78]: arr3d[0]
```

```
Out[78]:
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

Both scalar values and arrays can be assigned to `arr3d[0]`:

```
In [79]: old_values = arr3d[0].copy()
```

```
In [80]: arr3d[0] = 42
```

```
In [81]: arr3d
```

```
Out[81]:
```

```
array([[[42, 42, 42],
         [42, 42, 42]],
       [[ 7,  8,  9],
         [10, 11, 12]]])
```

```
In [82]: arr3d[0] = old_values
```

```
In [83]: arr3d
Out[83]:
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 [84]: arr3d[1, 0]
Out[84]: array([7, 8, 9])
```

This expression is the same as though we had indexed in two steps:

```
In [85]: x = arr3d[1]
```

```
In [86]: x
Out[86]:
array([[ 7,  8,  9],
       [10, 11, 12]])
```

```
In [87]: x[0]
Out[87]: 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 with the familiar syntax:

```
In [88]: arr
Out[88]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

```
In [89]: arr[1:6]
Out[89]: array([ 1,  2,  3,  4, 64])
```

Consider the two-dimensional array from before, `arr2d`. Slicing this array is a bit different:

```
In [90]: arr2d
Out[90]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
In [91]: arr2d[:2]
Out[91]:
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. It can be helpful to read the expression `arr2d[:2]` as “select the first two rows of `arr2d`.”

You can pass multiple slices just like you can pass multiple indexes:

```
In [92]: arr2d[:2, 1:]
Out[92]:
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.

For example, I can select the second row but only the first two columns like so:

```
In [93]: arr2d[1, :2]
Out[93]: array([4, 5])
```

Similarly, I can select the third column but only the first two rows like so:

```
In [94]: arr2d[:2, 2]
Out[94]: array([3, 6])
```

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 [95]: arr2d[:, :1]
Out[95]:
array([[1],
       [4],
       [7]])
```

Of course, assigning to a slice expression assigns to the whole selection:

```
In [96]: arr2d[:2, 1:] = 0

In [97]: arr2d
Out[97]:
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

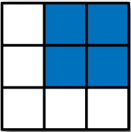
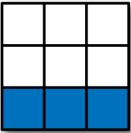
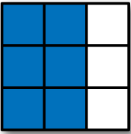
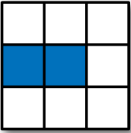
	Expression	Shape
	<code>arr[:2, 1:]</code>	<code>(2, 2)</code>
	<code>arr[2]</code> <code>arr[2, :]</code> <code>arr[2:, :]</code>	<code>(3,)</code> <code>(3,)</code> <code>(1, 3)</code>
	<code>arr[:, :2]</code>	<code>(3, 2)</code>
	<code>arr[1, :2]</code> <code>arr[1:2, :2]</code>	<code>(2,)</code> <code>(1, 2)</code>

Figure 4-2. Two-dimensional array slicing

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 [98]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
```

```
In [99]: data = np.random.randn(7, 4)
```

```
In [100]: names
```

```
Out[100]:
```

```
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
      dtype='<U4')
```

```
In [101]: data
```

```
Out[101]:
```

```
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]])
```

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 [102]: names == 'Bob'  
Out[102]: array([ True, False, False,  True, False, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```
In [103]: data[names == 'Bob']  
Out[103]:  
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 array axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers; more on this later).



Boolean selection will not fail if the boolean array is not the correct length, so I recommend care when using this feature.

In these examples, I select from the rows where `names == 'Bob'` and index the columns, too:

```
In [104]: data[names == 'Bob', 2:]  
Out[104]:  
array([[ 0.769 ,  1.2464],  
       [-0.5397,  0.477 ]])
```

```
In [105]: data[names == 'Bob', 3]  
Out[105]: array([ 1.2464,  0.477 ])
```

To select everything but 'Bob', you can either use `!=` or negate the condition using `~`:

```
In [106]: names != 'Bob'  
Out[106]: array([False,  True,  True, False,  True,  True,  True], dtype=bool)
```

```
In [107]: data[~(names == 'Bob')]  
Out[107]:  
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]])
```

The `~` operator can be useful when you want to invert a general condition:

```
In [108]: cond = names == 'Bob'

In [109]: data[~cond]
Out[109]:
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 [110]: mask = (names == 'Bob') | (names == 'Will')

In [111]: mask
Out[111]: array([ True, False,  True,  True,  True, False, False], dtype=bool)

In [112]: data[mask]
Out[112]:
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.



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 [113]: data[data < 0] = 0

In [114]: data
Out[114]:
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 one-dimensional boolean array is also easy:

```
In [115]: data[names != 'Joe'] = 7

In [116]: data
Out[116]:
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.    ]])
```

As we will see later, these types of operations on two-dimensional data are convenient to do with pandas.

Fancy Indexing

Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays. Suppose we had an 8×4 array:

```
In [117]: arr = np.empty((8, 4))

In [118]: for i in range(8):
.....:     arr[i] = i

In [119]: arr
Out[119]:
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 [120]: arr[[4, 3, 0, 6]]
Out[120]:
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 selects rows from the end:

```
In [121]: arr[[-3, -5, -7]]
Out[121]:
```

```
array([[ 5.,  5.,  5.,  5.],
       [ 3.,  3.,  3.,  3.],
       [ 1.,  1.,  1.,  1.]])
```

Passing multiple index arrays does something slightly different; it selects a one-dimensional array of elements corresponding to each tuple of indices:

```
In [122]: arr = np.arange(32).reshape((8, 4))
```

```
In [123]: arr
```

```
Out[123]:
```

```
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 [124]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
```

```
Out[124]: array([ 4, 23, 29, 10])
```

We'll look at the reshape method in more detail in [Appendix A](#).

Here the elements (1, 0), (5, 3), (7, 1), and (2, 2) were selected. Regardless of how many dimensions the array has (here, only 2), the result of fancy indexing is always one-dimensional.

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 [125]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
```

```
Out[125]:
```

```
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 that similarly returns a view on the underlying data without copying anything. Arrays have the transpose method and also the special T attribute:

```
In [126]: arr = np.arange(15).reshape((3, 5))
```

```
In [127]: arr
```

```
Out[127]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
In [128]: arr.T
Out[128]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

When doing matrix computations, you may do this very often—for example, when computing the inner matrix product using `np.dot`:

```
In [129]: arr = np.random.randn(6, 3)
```

```
In [130]: arr
Out[130]:
array([[ -0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329],
       [-2.3594, -0.1995, -1.542 ],
       [-0.9707, -1.307 ,  0.2863],
       [ 0.378 , -0.7539,  0.3313],
       [ 1.3497,  0.0699,  0.2467]])
```

```
In [131]: np.dot(arr.T, arr)
Out[131]:
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 [132]: arr = np.arange(16).reshape((2, 2, 4))
```

```
In [133]: arr
Out[133]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]])
```

```
In [134]: arr.transpose((1, 0, 2))
Out[134]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]])
```

Here, the axes have been reordered with the second axis first, the first axis second, and the last axis unchanged.

Simple transposing with `.T` is a special case of swapping axes. `ndarray` has the method `swapaxes`, which takes a pair of axis numbers and switches the indicated axes to rearrange the data:

```
In [135]: arr
Out[135]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7]],
      [[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
In [136]: arr.swapaxes(1, 2)
Out[136]:
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.

4.2 Universal Functions: Fast Element-Wise Array Functions

A universal function, or *ufunc*, is a function that performs element-wise 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 element-wise transformations, like `sqrt` or `exp`:

```
In [137]: arr = np.arange(10)
```

```
In [138]: arr
Out[138]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [139]: np.sqrt(arr)
Out[139]:
array([ 0.    ,  1.    ,  1.4142,  1.7321,  2.    ,  2.2361,  2.4495,
        2.6458,  2.8284,  3.    ])
```

```
In [140]: np.exp(arr)
Out[140]:
array([ 1.    ,  2.7183,  7.3891,  20.0855,  54.5982,
        148.4132,  403.4288, 1096.6332, 2980.958 , 8103.0839])
```

These are referred to as *unary* ufuncs. Others, such as `add` or `maximum`, take two arrays (thus, *binary* ufuncs) and return a single array as the result:

```
In [141]: x = np.random.randn(8)
```

```
In [142]: y = np.random.randn(8)
```

```
In [143]: x
```

```
Out[143]:
```

```
array([-0.0119,  1.0048,  1.3272, -0.9193, -1.5491,  0.0222,  0.7584,  
       -0.6605])
```

```
In [144]: y
```

```
Out[144]:
```

```
array([ 0.8626, -0.01  ,  0.05  ,  0.6702,  0.853  , -0.9559, -0.0235,  
       -2.3042])
```

```
In [145]: np.maximum(x, y)
```

```
Out[145]:
```

```
array([ 0.8626,  1.0048,  1.3272,  0.6702,  0.853  ,  0.0222,  0.7584,  
       -0.6605])
```

Here, `numpy.maximum` computed the element-wise maximum of the elements in `x` and `y`.

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 [146]: arr = np.random.randn(7) * 5
```

```
In [147]: arr
```

```
Out[147]: array([-3.2623, -6.0915, -6.663  ,  5.3731,  3.6182,  3.45  ,  5.0077])
```

```
In [148]: remainder, whole_part = np.modf(arr)
```

```
In [149]: remainder
```

```
Out[149]: array([-0.2623, -0.0915, -0.663  ,  0.3731,  0.6182,  0.45  ,  0.0077])
```

```
In [150]: whole_part
```

```
Out[150]: array([-3., -6., -6.,  5.,  3.,  3.,  5.])
```

Ufuncs accept an optional `out` argument that allows them to operate in-place on arrays:

```
In [151]: arr
```

```
Out[151]: array([-3.2623, -6.0915, -6.663  ,  5.3731,  3.6182,  3.45  ,  5.0077])
```

```
In [152]: np.sqrt(arr)
```

```
Out[152]: array([ nan,      nan,      nan,  2.318  ,  1.9022,  1.8574,  2.2378])
```

```
In [153]: np.sqrt(arr, arr)
```

```
Out[153]: array([ nan,   nan,   nan,  2.318 ,  1.9022,  1.8574,  2.2378])
```

```
In [154]: arr
```

```
Out[154]: array([ nan,   nan,   nan,  2.318 ,  1.9022,  1.8574,  2.2378])
```

See Tables 4-3 and 4-4 for a listing of available ufuncs.

Table 4-3. *Unary ufuncs*

Function	Description
abs, fabs	Compute the absolute value element-wise for integer, floating-point, or complex values
sqrt	Compute the square root of each element (equivalent to <code>arr ** 0.5</code>)
square	Compute the square of each element (equivalent to <code>arr ** 2</code>)
exp	Compute the exponent e^x of each element
log, log10, log2, log1p	Natural logarithm (base e), log base 10, log base 2, and $\log(1+x)$, respectively
sign	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
ceil	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
floor	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
rint	Round elements to the nearest integer, preserving the dtype
modf	Return fractional and integral parts of array as a separate array
isnan	Return boolean array indicating whether each value is NaN (Not a Number)
isfinite, isinf	Return boolean array indicating whether each element is finite (non- <code>inf</code> , non-NaN) or infinite, respectively
cos, cosh, sin, sinh, tan, tanh	Regular and hyperbolic trigonometric functions
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Inverse trigonometric functions
logical_not	Compute truth value of <code>not x</code> element-wise (equivalent to <code>~arr</code>).

Table 4-4. *Binary universal functions*

Function	Description
add	Add corresponding elements in arrays
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide, floor_divide	Divide or floor divide (truncating the remainder)
power	Raise elements in first array to powers indicated in second array
maximum, fmax	Element-wise maximum; <code>fmax</code> ignores NaN
minimum, fmin	Element-wise minimum; <code>fmin</code> ignores NaN
mod	Element-wise modulus (remainder of division)
copysign	Copy sign of values in second argument to values in first argument

Function	Description
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding boolean array (equivalent to infix operators >, >=, <, <=, ==, !=)
logical_and, logical_or, logical_xor	Compute element-wise truth value of logical operation (equivalent to infix operators &, , ^)

4.3 Array-Oriented 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 [Appendix A](#), I 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 [155]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points
```

```
In [156]: xs, ys = np.meshgrid(points, points)
```

```
In [157]: ys
```

```
Out[157]:
```

```
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 matter of writing the same expression you would write with two points:

```
In [158]: z = np.sqrt(xs ** 2 + ys ** 2)
```

```
In [159]: z
```

```
Out[159]:
```

```
array([[ 7.0711,  7.064 ,  7.0569, ...,  7.0499,  7.0569,  7.064 ],
       [ 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 ]])
```

As a preview of [Chapter 9](#), I use `matplotlib` to create visualizations of this two-dimensional array:

```
In [160]: import matplotlib.pyplot as plt  
  
In [161]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()  
Out[161]: <matplotlib.colorbar.Colorbar at 0x7f715e3fa630>  
  
In [162]: plt.title("Image plot of  $\sqrt{x^2 + y^2}$  for a grid of values")  
Out[162]: <matplotlib.text.Text at 0x7f715d2de748>
```

See [Figure 4-3](#). Here I used the `matplotlib` function `imshow` to create an image plot from a two-dimensional array of function values.

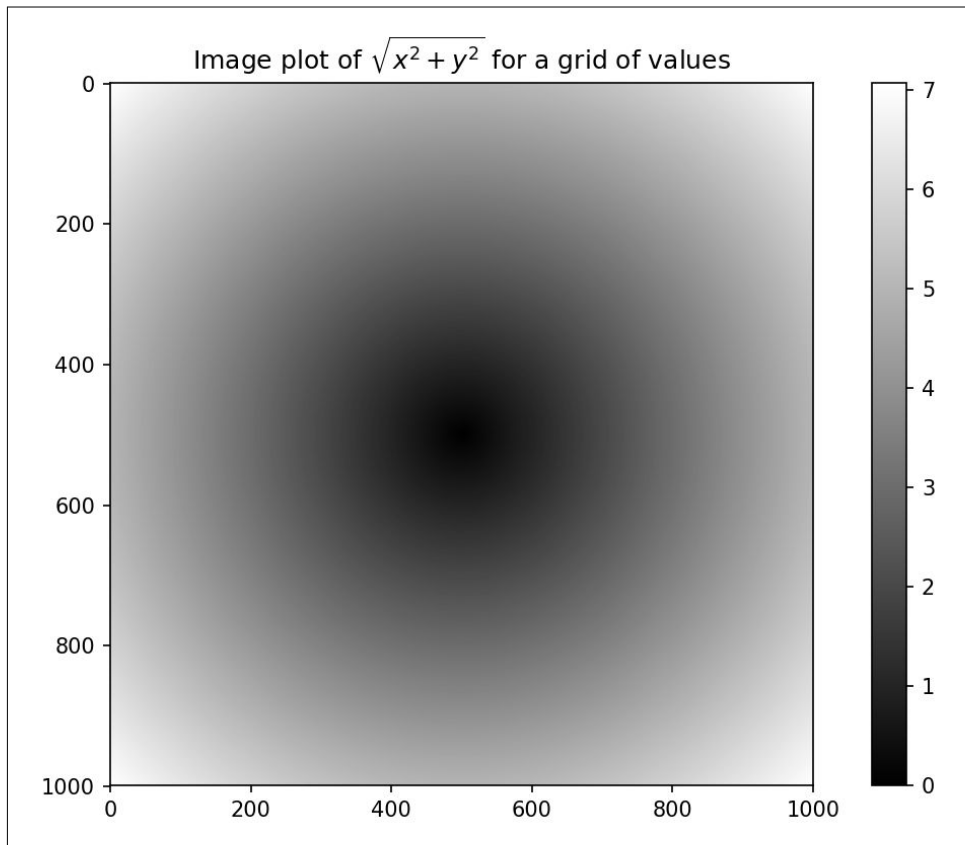


Figure 4-3. Plot of function evaluated on grid

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 [165]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
```

```
In [166]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
```

```
In [167]: 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`, and otherwise take the value from `yarr`. A list comprehension doing this might look like:

```
In [168]: result = [(x if c else y)
.....:                  for x, y, c in zip(xarr, yarr, cond)]
```

```
In [169]: result
```

```
Out[169]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

This has multiple problems. First, it will not be very fast for large arrays (because all the work is being done in interpreted Python code). Second, it will not work with multidimensional arrays. With `np.where` you can write this very concisely:

```
In [170]: result = np.where(cond, xarr, yarr)
```

```
In [171]: result
```

```
Out[171]: 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 [172]: arr = np.random.randn(4, 4)
```

```
In [173]: arr
```

```
Out[173]:
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 [174]: arr > 0
```

```
Out[174]:
array([[False, False, False, False],
       [ True,  True, False,  True],
       [ True,  True,  True, False],
       [ True, False,  True,  True]], dtype=bool)
```

```
In [175]: np.where(arr > 0, 2, -2)
```

```
Out[175]:
array([[ -2,  -2,  -2,  -2],
       [  2,   2,  -2,   2],
```

```
[ 2,  2,  2, -2],
 [ 2, -2,  2,  2]])
```

You can combine scalars and arrays when using `np.where`. For example, I can replace all positive values in `arr` with the constant 2 like so:

```
In [176]: np.where(arr > 0, 2, arr) # set only positive values to 2
Out[176]:
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 `np.where` can be more than just equal-sized arrays or scalars.

Mathematical and Statistical Methods

A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class. You can use aggregations (often called *reductions*) like `sum`, `mean`, and `std` (standard deviation) either by calling the array instance method or using the top-level NumPy function.

Here I generate some normally distributed random data and compute some aggregate statistics:

```
In [177]: arr = np.random.randn(5, 4)
```

```
In [178]: arr
Out[178]:
array([[ 2.1695, -0.1149,  2.0037,  0.0296],
       [ 0.7953,  0.1181, -0.7485,  0.585 ],
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363,  1.0954],
       [ 0.9809, -0.5895,  1.5817, -0.5287]])
```

```
In [179]: arr.mean()
Out[179]: 0.19607051119998253
```

```
In [180]: np.mean(arr)
Out[180]: 0.19607051119998253
```

```
In [181]: arr.sum()
Out[181]: 3.9214102239996507
```

Functions like `mean` and `sum` take an optional `axis` argument that computes the statistic over the given axis, resulting in an array with one fewer dimension:

```
In [182]: arr.mean(axis=1)
Out[182]: array([ 1.022 ,  0.1875, -0.502 , -0.0881,  0.3611])
```

```
In [183]: arr.sum(axis=0)
Out[183]: array([ 3.1693, -2.6345,  2.2381,  1.1486])
```

Here, `arr.mean(1)` means “compute mean across the columns” where `arr.sum(0)` means “compute sum down the rows.”

Other methods like `cumsum` and `cumprod` do not aggregate, instead producing an array of the intermediate results:

```
In [184]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
In [185]: arr.cumsum()
```

```
Out[185]: 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 [186]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
```

```
In [187]: arr
```

```
Out[187]:
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
In [188]: arr.cumsum(axis=0)
```

```
Out[188]:
```

```
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])
```

```
In [189]: arr.cumprod(axis=1)
```

```
Out[189]:
```

```
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
<code>sum</code>	Sum of all the elements in the array or along an axis; zero-length arrays have sum 0
<code>mean</code>	Arithmetic mean; zero-length arrays have NaN mean
<code>std</code> , <code>var</code>	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator <code>n</code>)
<code>min</code> , <code>max</code>	Minimum and maximum
<code>argmin</code> , <code>argmax</code>	Indices of minimum and maximum elements, respectively
<code>cumsum</code>	Cumulative sum of elements starting from 0
<code>cumprod</code>	Cumulative product of elements starting from 1

Methods for Boolean Arrays

Boolean values are coerced to 1 (True) and 0 (False) in the preceding methods. Thus, `sum` is often used as a means of counting True values in a boolean array:

```
In [190]: arr = np.random.randn(100)
```

```
In [191]: (arr > 0).sum() # Number of positive values
```

```
Out[191]: 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 [192]: bools = np.array([False, False, True, False])
```

```
In [193]: bools.any()
```

```
Out[193]: True
```

```
In [194]: bools.all()
```

```
Out[194]: 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 with the `sort` method:

```
In [195]: arr = np.random.randn(6)
```

```
In [196]: arr
```

```
Out[196]: array([ 0.6095, -0.4938,  1.24   , -0.1357,  1.43   , -0.8469])
```

```
In [197]: arr.sort()
```

```
In [198]: arr
```

```
Out[198]: array([-0.8469, -0.4938, -0.1357,  0.6095,  1.24   ,  1.43   ])
```

You can sort each one-dimensional section of values in a multidimensional array in-place along an axis by passing the axis number to `sort`:

```
In [199]: arr = np.random.randn(5, 3)
```

```
In [200]: arr
```

```
Out[200]:
```

```
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 [201]: arr.sort(1)
```

```
In [202]: arr
```

```
Out[202]:
```

```
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 [203]: large_arr = np.random.randn(1000)
```

```
In [204]: large_arr.sort()
```

```
In [205]: large_arr[int(0.05 * len(large_arr))] # 5% quantile
```

```
Out[205]: -1.5311513550102103
```

For more details on using NumPy's sorting methods, and more advanced techniques like indirect sorts, see [Appendix A](#). Several other kinds of data manipulations related to sorting (e.g., sorting a table of data by one or more columns) can also 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 [206]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
```

```
In [207]: np.unique(names)
```

```
Out[207]:
```

```
array(['Bob', 'Joe', 'Will'],  
      dtype='<U4')
```

```
In [208]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
```

```
In [209]: np.unique(ints)
```

```
Out[209]: array([1, 2, 3, 4])
```

Contrast `np.unique` with the pure Python alternative:

```
In [210]: sorted(set(names))
```

```
Out[210]: ['Bob', 'Joe', 'Will']
```

Another function, `np.in1d`, tests membership of the values in one array in another, returning a boolean array:

```
In [211]: values = np.array([6, 0, 0, 3, 2, 5, 6])
```

```
In [212]: np.in1d(values, [2, 3, 6])
```

```
Out[212]: array([ True, False, False,  True,  True, False,  True], dtype=bool)
```

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

4.4 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 [213]: arr = np.arange(10)
```

```
In [214]: 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 with `np.load`:

```
In [215]: np.load('some_array.npy')
```

```
Out[215]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

You save multiple arrays in an uncompressed archive using `np.savez` and passing the arrays as keyword arguments:

```
In [216]: np.savez('array_archive.npz', a=arr, b=arr)
```

When loading an `.npz` file, you get back a dict-like object that loads the individual arrays lazily:

```
In [217]: arch = np.load('array_archive.npz')
```

```
In [218]: arch['b']
```

```
Out[218]: 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 [219]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

4.5 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. Thus, there is a function `dot`, both an array method and a function in the `numpy` namespace, for matrix multiplication:

```
In [223]: x = np.array([[1., 2., 3.], [4., 5., 6.]])
```

```
In [224]: y = np.array([[6., 23.], [-1., 7.], [8., 9.]])
```

```
In [225]: x
```

```
Out[225]:
```

```
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

```
In [226]: y
```

```
Out[226]:
```

```
array([[ 6., 23.],
       [-1.,  7.],
       [ 8.,  9.]])
```

```
In [227]: x.dot(y)
```

```
Out[227]:
```

```
array([[ 28.,  64.],
       [ 67., 181.]])
```

`x.dot(y)` is equivalent to `np.dot(x, y)`:

```
In [228]: np.dot(x, y)
```

```
Out[228]:
```

```
array([[ 28.,  64.],
       [ 67., 181.]])
```

A matrix product between a two-dimensional array and a suitably sized one-dimensional array results in a one-dimensional array:

```
In [229]: np.dot(x, np.ones(3))
```

```
Out[229]: array([ 6., 15.])
```

The `@` symbol (as of Python 3.5) also works as an infix operator that performs matrix multiplication:

```
In [230]: x @ np.ones(3)
```

```
Out[230]: array([ 6., 15.])
```

`numpy.linalg` has a standard set of matrix decompositions and things like inverse and determinant. These are implemented under the hood via the same industry-standard linear algebra libraries used in other languages like MATLAB and R, such as BLAS, LAPACK, or possibly (depending on your NumPy build) the proprietary Intel MKL (Math Kernel Library):

```
In [231]: from numpy.linalg import inv, qr
```

```
In [232]: X = np.random.randn(5, 5)
```

```
In [233]: mat = X.T.dot(X)
```

```
In [234]: inv(mat)
```

```
Out[234]:
```

```
array([[ 933.1189,   871.8258, -1417.6902, -1460.4005,  1782.1391],
       [ 871.8258,   815.3929, -1325.9965, -1365.9242,  1666.9347],
       [-1417.6902, -1325.9965,  2158.4424,  2222.0191, -2711.6822],
       [-1460.4005, -1365.9242,  2222.0191,  2289.0575, -2793.422 ],
       [ 1782.1391,  1666.9347, -2711.6822, -2793.422 ,  3409.5128]])
```

```
In [235]: mat.dot(inv(mat))
```

```
Out[235]:
```

```
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 [236]: q, r = qr(mat)
```

```
In [237]: r
```

```
Out[237]:
```

```
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]])
```

The expression `X.T.dot(X)` computes the dot product of `X` with its transpose `X.T`.

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>diag</code>	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
<code>dot</code>	Matrix multiplication
<code>trace</code>	Compute the sum of the diagonal elements
<code>det</code>	Compute the matrix determinant

Function	Description
----------	-------------

<code>eig</code>	Compute the eigenvalues and eigenvectors of a square matrix
<code>inv</code>	Compute the inverse of a square matrix
<code>pinv</code>	Compute the Moore–Penrose pseudo-inverse of a matrix
<code>qr</code>	Compute the QR decomposition
<code>svd</code>	Compute the singular value decomposition (SVD)
<code>solve</code>	Solve the linear system $Ax = b$ for x , where A is a square matrix
<code>lstsq</code>	Compute the least-squares solution to $Ax = b$

4.6 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 4×4 array of samples from the standard normal distribution using `normal`:

```
In [238]: samples = np.random.normal(size=(4, 4))
```

```
In [239]: samples
```

```
Out[239]:
```

```
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 [240]: from random import normalvariate
```

```
In [241]: N = 1000000
```

```
In [242]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.77 s +- 126 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

```
In [243]: %timeit np.random.normal(size=N)
61.7 ms +- 1.32 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

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 [244]: np.random.seed(1234)
```

The data generation functions in `numpy.random` use a global random seed. To avoid global state, you can use `numpy.random.RandomState` to create a random number generator isolated from others:

```
In [245]: rng = np.random.RandomState(1234)

In [246]: rng.randn(10)
Out[246]:
array([ 0.4714, -1.191 ,  1.4327, -0.3127, -0.7206,  0.8872,  0.8596,
        -0.6365,  0.0157, -2.2427])
```

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
<code>seed</code>	Seed the random number generator
<code>permutation</code>	Return a random permutation of a sequence, or return a permuted range
<code>shuffle</code>	Randomly permute a sequence in-place
<code>rand</code>	Draw samples from a uniform distribution
<code>randint</code>	Draw random integers from a given low-to-high range
<code>randn</code>	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
<code>binomial</code>	Draw samples from a binomial distribution
<code>normal</code>	Draw samples from a normal (Gaussian) distribution
<code>beta</code>	Draw samples from a beta distribution
<code>chisquare</code>	Draw samples from a chi-square distribution
<code>gamma</code>	Draw samples from a gamma distribution
<code>uniform</code>	Draw samples from a uniform [0, 1) distribution

4.7 Example: Random Walks

The simulation of **random walks** provides an illustrative application of utilizing array operations. Let's first consider a simple random walk starting at 0 with steps of 1 and -1 occurring with equal probability.

Here is a pure Python way to implement a single random walk with 1,000 steps using the built-in `random` module:

```
In [247]: import random
.....: position = 0
.....: walk = [position]
.....: steps = 1000
.....: for i in range(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:

```
In [249]: plt.plot(walk[:100])
```

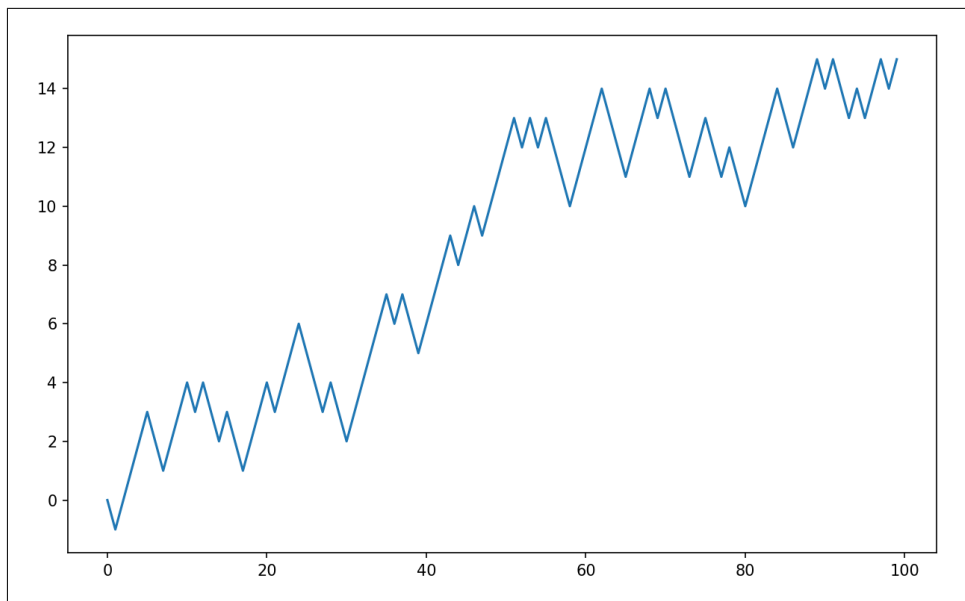


Figure 4-4. A simple random walk

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 [251]: nsteps = 1000
```

```
In [252]: draws = np.random.randint(0, 2, size=nsteps)
```

```
In [253]: steps = np.where(draws > 0, 1, -1)
```

```
In [254]: walk = steps.cumsum()
```

From this we can begin to extract statistics like the minimum and maximum value along the walk's trajectory:

```
In [255]: walk.min()
```

```
Out[255]: -3
```

```
In [256]: walk.max()
Out[256]: 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, we can compute this using `argmax`, which returns the first index of the maximum value in the boolean array (True is the maximum value):

```
In [257]: (np.abs(walk) >= 10).argmax()
Out[257]: 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 preceding code. If passed a 2-tuple, the `numpy.random` functions will generate a two-dimensional array of draws, and we can compute the cumulative sum across the rows to compute all 5,000 random walks in one shot:

```
In [258]: nwalks = 5000
```

```
In [259]: nsteps = 1000
```

```
In [260]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 or 1
```

```
In [261]: steps = np.where(draws > 0, 1, -1)
```

```
In [262]: walks = steps.cumsum(1)
```

```
In [263]: walks
```

```
Out[263]:
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 [264]: walks.max()
Out[264]: 138
```

```
In [265]: walks.min()
Out[265]: -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 [266]: hits30 = (np.abs(walks) >= 30).any(1)
```

```
In [267]: hits30
Out[267]: array([False,  True,  False, ...,  False,  True,  False], dtype=bool)
```

```
In [268]: hits30.sum() # Number that hit 30 or -30
Out[268]: 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 [269]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)
```

```
In [270]: crossing_times.mean()
Out[270]: 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:

```
In [271]: steps = np.random.normal(loc=0, scale=0.25,
.....:                               size=(nwalks, nsteps))
```

4.8 Conclusion

While much of the rest of the book will focus on building data wrangling skills with `pandas`, we will continue to work in a similar array-based style. In [Appendix A](#), we will dig deeper into NumPy features to help you further develop your array computing skills.

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.

While pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with tabular or heterogeneous data. NumPy, by contrast, is best suited for working with homogeneous numerical array data.

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 800 distinct contributors, who've been helping 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
```


5.1 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', 'c'])
```

```
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 labels 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
```

Here `['c', 'a', 'd']` is interpreted as a list of indices, even though it contains strings instead of integers.

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, 'Utah': 5000}
```

```
In [27]: obj3 = pd.Series(sdata)
```

```
In [28]: obj3
Out[28]:
Ohio      35000
Oregon    16000
Texas     71000
Utah      5000
dtype: int64
```

When you are 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, three 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. Since 'Utah' was not included in `states`, it is excluded from the resulting object.

I will use the terms “missing” or “NA” interchangeably 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 by index label 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 will be addressed in more detail later. 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
```

```
Out[41]:
```

```
0    4
```

```
1    7
```

```
2   -5
```

```
3    3
```

```
dtype: int64
```

```
In [42]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
```

```
In [43]: obj
```

```
Out[43]:
```

```
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.



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 we will discuss in [Chapter 8](#) 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', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
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 [45]: frame
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
5	3.2	Nevada	2003

If you are using the Jupyter notebook, pandas DataFrame objects will be displayed as a more browser-friendly HTML table.

For large DataFrames, the head method selects only the first five rows:

```
In [46]: frame.head()
Out[46]:
```

	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 [47]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[47]:
```

	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
5	2003	Nevada	3.2

If you pass a column that isn't contained in the dict, it will appear with missing values in the result:

```
In [48]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
    ....:                          index=['one', 'two', 'three', 'four',
    ....:                          'five', 'six'])
```

```
In [49]: frame2
Out[49]:
```

	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
six	2003	Nevada	3.2	NaN

```
In [50]: frame2.columns
Out[50]: 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 [51]: frame2['state']
Out[51]:
```

one	Ohio
two	Ohio
three	Ohio
four	Nevada
five	Nevada
six	Nevada

```
Name: state, dtype: object
```

```
In [52]: frame2.year
Out[52]:
```

one	2000
two	2001
three	2002
four	2001
five	2002
six	2003

```
Name: year, dtype: int64
```



Attribute-like access (e.g., `frame2.year`) and tab completion of column names in IPython is provided as a convenience.

`frame2[column]` works for any column name, but `frame2.column` only works when the column name is a valid Python variable name.

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 with the special `loc` attribute (much more on this later):

```
In [53]: frame2.loc['three']
Out[53]:
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 [54]: frame2['debt'] = 16.5
```

```
In [55]: frame2
Out[55]:
```

	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
six	2003	Nevada	3.2	16.5

```
In [56]: frame2['debt'] = np.arange(6.)
```

```
In [57]: frame2
Out[57]:
```

	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
six	2003	Nevada	3.2	5.0

When you are 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 [58]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
```

```
In [59]: frame2['debt'] = val
```

```
In [60]: frame2
Out[60]:
```

	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
six	2003	Nevada	3.2	NaN

Assigning a column that doesn't exist will create a new column. The `del` keyword will delete columns as with a dict.

As an example of `del`, I first add a new column of boolean values where the state column equals 'Ohio':

```
In [61]: frame2['eastern'] = frame2.state == 'Ohio'
```

```
In [62]: frame2
```

```
Out[62]:
```

	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
six	2003	Nevada	3.2	NaN	False



New columns cannot be created with the `frame2.eastern` syntax.

The `del` method can then be used to remove this column:

```
In [63]: del frame2['eastern']
```

```
In [64]: frame2.columns
```

```
Out[64]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```



The column returned from 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 with the Series's `copy` method.

Another common form of data is a nested dict of dicts:

```
In [65]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
.....:         'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

If the nested dict is passed to the DataFrame, pandas will interpret the outer dict keys as the columns and the inner keys as the row indices:

```
In [66]: frame3 = pd.DataFrame(pop)
```

```
In [67]: frame3
```

```
Out[67]:
```

	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 [68]: frame3.T
Out[68]:
      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 [69]: pd.DataFrame(pop, index=[2001, 2002, 2003])
Out[69]:
      Nevada  Ohio
2001    2.4    1.7
2002    2.9    3.6
2003    NaN    NaN
```

Dicts of Series are treated in much the same way:

```
In [70]: pdata = {'Ohio': frame3['Ohio'][::-1],
....:            'Nevada': frame3['Nevada'][:2]}
```

```
In [71]: pd.DataFrame(pdata)
Out[71]:
      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 [72]: frame3.index.name = 'year'; frame3.columns.name = 'state'
```

```
In [73]: frame3
Out[73]:
state  Nevada  Ohio
year
2000    NaN   1.5
2001    2.4   1.7
2002    2.9   3.6
```

As with Series, the `values` attribute returns the data contained in the DataFrame as a two-dimensional ndarray:

```
In [74]: frame3.values
Out[74]:
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 [75]: frame2.values  
Out[75]:  
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],  
       [2003, 'Nevada', 3.2, nan]], 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, lists, or tuples	Each sequence becomes a column in the DataFrame; all sequences must be the same length
NumPy structured/record 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 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 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 you use when constructing a Series or DataFrame is internally converted to an Index:

```
In [76]: obj = pd.Series(range(3), index=['a', 'b', 'c'])
```

```
In [77]: index = obj.index
```

```
In [78]: index
```

```
Out[78]: Index(['a', 'b', 'c'], dtype='object')
```

```
In [79]: index[1:]
```

```
Out[79]: 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 [80]: labels = pd.Index(np.arange(3))
```

```
In [81]: labels
```

```
Out[81]: Int64Index([0, 1, 2], dtype='int64')
```

```
In [82]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)
```

```
In [83]: obj2
```

```
Out[83]:
```

```
0    1.5
```

```
1   -2.5
```

```
2    0.0
```

```
dtype: float64
```

```
In [84]: obj2.index is labels
```

```
Out[84]: True
```



Some users will not often take advantage of the capabilities provided by indexes, but because some operations will yield results containing indexed data, it's important to understand how they work.

In addition to being array-like, an Index also behaves like a fixed-size set:

```
In [85]: frame3
```

```
Out[85]:
```

```
state Nevada Ohio
```

```
year
```

```
2000      NaN  1.5
```

```
2001      2.4  1.7
```

```
2002      2.9  3.6
```

```
In [86]: frame3.columns
```

```
Out[86]: Index(['Nevada', 'Ohio'], dtype='object', name='state')
```

```
In [87]: 'Ohio' in frame3.columns
```

```
Out[87]: True
```

```
In [88]: 2003 in frame3.index
```

```
Out[88]: False
```

Unlike Python sets, a pandas Index can contain duplicate labels:

```
In [89]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])
```

```
In [90]: dup_labels
```

```
Out[90]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Selections with duplicate labels will select all occurrences of that label.

Each Index has a number of methods and properties for set logic, which answer other common questions about the data it contains. Some useful ones are summarized in Table 5-2.

Table 5-2. Some Index methods and properties

Method	Description
<code>append</code>	Concatenate with additional Index objects, producing a new Index
<code>difference</code>	Compute set difference as an Index
<code>intersection</code>	Compute set intersection
<code>union</code>	Compute set union
<code>isin</code>	Compute boolean array indicating whether each value is contained in the passed collection
<code>delete</code>	Compute new Index with element at index <code>i</code> deleted
<code>drop</code>	Compute new Index by deleting passed values
<code>insert</code>	Compute new Index by inserting element at index <code>i</code>
<code>is_monotonic</code>	Returns <code>True</code> if each element is greater than or equal to the previous element
<code>is_unique</code>	Returns <code>True</code> if the Index has no duplicate values
<code>unique</code>	Compute the array of unique values in the Index

5.2 Essential Functionality

This section will walk you through the fundamental mechanics of interacting with the data contained in a Series or DataFrame. In the chapters to come, we 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; instead, we'll focus on the most important features, leaving the less common (i.e., 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 an example:

```
In [91]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
```

```
In [92]: obj
```

```
Out[92]:
```

```
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 [93]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
```

```
In [94]: obj2
```

```
Out[94]:
```

```
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 [95]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
```

```
In [96]: obj3
```

```
Out[96]:
```

```
0      blue
```

```
2     purple
```

```
4     yellow
```

```
dtype: object
```

```
In [97]: obj3.reindex(range(6), method='ffill')
```

```
Out[97]:
```

```
0      blue
```

```
1      blue
```

```
2     purple
```

```
3     purple
```

```
4     yellow
```

```
5     yellow
```

```
dtype: object
```

With `DataFrame`, `reindex` can alter either the (row) index, columns, or both. When passed only a sequence, it reindexes the rows in the result:

```
In [98]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
.....:                        index=['a', 'c', 'd'],
.....:                        columns=['Ohio', 'Texas', 'California'])
```

```
In [99]: frame
```

```
Out[99]:
```

```
   Ohio  Texas  California
```

```
a     0     1           2
```

```
c     3     4           5
```

```
d     6     7           8
```

```
In [100]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])
```

```
In [101]: frame2
Out[101]:
   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 with the columns keyword:

```
In [102]: states = ['Texas', 'Utah', 'California']
```

```
In [103]: frame.reindex(columns=states)
```

```
Out[103]:
   Texas  Utah  California
a      1  NaN         2
c      4  NaN         5
d      7  NaN         8
```

See [Table 5-3](#) for more about the arguments to `reindex`.

As we'll explore in more detail, you can reindex more succinctly by label-indexing with `loc`, and many users prefer to use it exclusively:

```
In [104]: frame.loc[['a', 'b', 'c', 'd'], states]
```

```
Out[104]:
   Texas  Utah  California
a      1  NaN         2
b      NaN NaN         NaN
c      4  NaN         5
d      7  NaN         8
```

Table 5-3. reindex function arguments

Argument	Description
<code>index</code>	New sequence to use as index. Can be Index instance or any other sequence-like Python data structure. An Index will be used exactly as is without any copying.
<code>method</code>	Interpolation (fill) method; 'ffill' fills forward, while 'bfill' fills backward.
<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 Index on level of MultiIndex; otherwise select subset of.
<code>copy</code>	If True, always copy underlying data even if new index is equivalent to old index; if False, 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 [105]: obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [106]: obj
```

```
Out[106]:
```

```
a    0.0
```

```
b    1.0
```

```
c    2.0
```

```
d    3.0
```

```
e    4.0
```

```
dtype: float64
```

```
In [107]: new_obj = obj.drop('c')
```

```
In [108]: new_obj
```

```
Out[108]:
```

```
a    0.0
```

```
b    1.0
```

```
d    3.0
```

```
e    4.0
```

```
dtype: float64
```

```
In [109]: obj.drop(['d', 'c'])
```

```
Out[109]:
```

```
a    0.0
```

```
b    1.0
```

```
e    4.0
```

```
dtype: float64
```

With DataFrame, index values can be deleted from either axis. To illustrate this, we first create an example DataFrame:

```
In [110]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
.....:                        index=['Ohio', 'Colorado', 'Utah', 'New York'],
.....:                        columns=['one', 'two', 'three', 'four'])
```

```
In [111]: data
```

```
Out[111]:
```

```
      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
```

Calling drop with a sequence of labels will drop values from the row labels (axis 0):

```
In [112]: data.drop(['Colorado', 'Ohio'])
```

```
Out[112]:
```

```
      one  two  three  four
```

```
Utah    8    9    10   11
```

```
New York 12   13    14   15
```


You can drop values from the columns by passing `axis=1` or `axis='columns'`:

```
In [113]: data.drop('two', axis=1)
Out[113]:
```

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

```
In [114]: data.drop(['two', 'four'], axis='columns')
Out[114]:
```

	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 [115]: obj.drop('c', inplace=True)

In [116]: obj
Out[116]:
```

a	0.0
b	1.0
d	3.0
e	4.0

```
dtype: float64
```

Be careful with the `inplace`, as it destroys any data that is dropped.

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 [117]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [118]: obj
Out[118]:
```

a	0.0
b	1.0
c	2.0
d	3.0

```
dtype: float64

In [119]: obj['b']
Out[119]: 1.0
```

```
In [120]: obj[1]
Out[120]: 1.0
```

```
In [121]: obj[2:4]
Out[121]:
c    2.0
d    3.0
dtype: float64
```

```
In [122]: obj[['b', 'a', 'd']]
Out[122]:
b    1.0
a    0.0
d    3.0
dtype: float64
```

```
In [123]: obj[[1, 3]]
Out[123]:
b    1.0
d    3.0
dtype: float64
```

```
In [124]: obj[obj < 2]
Out[124]:
a    0.0
b    1.0
dtype: float64
```

Slicing with labels behaves differently than normal Python slicing in that the end-point is inclusive:

```
In [125]: obj['b':'c']
Out[125]:
b    1.0
c    2.0
dtype: float64
```

Setting using these methods modifies the corresponding section of the Series:

```
In [126]: obj['b':'c'] = 5
```

```
In [127]: obj
Out[127]:
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

```
In [128]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
.....:                        index=['Ohio', 'Colorado', 'Utah', 'New York'],
.....:                        columns=['one', 'two', 'three', 'four'])
```

```
In [129]: data
Out[129]:
```

	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 [130]: data['two']
Out[130]:
```

Ohio	1
Colorado	5
Utah	9
New York	13

Name: two, dtype: int64

```
In [131]: data[['three', 'one']]
Out[131]:
```

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

Indexing like this has a few special cases. First, slicing or selecting data with a boolean array:

```
In [132]: data[:2]
Out[132]:
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

```
In [133]: data[data['three'] > 5]
Out[133]:
```

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

The row selection syntax `data[:2]` is provided as a convenience. Passing a single element or a list to the `[]` operator selects columns.

Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

```
In [134]: data < 5
Out[134]:
```

	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 [135]: data[data < 5] = 0
```

```
In [136]: data
Out[136]:
```

	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.

Selection with loc and iloc

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`).

As a preliminary example, let's select a single row and multiple columns by label:

```
In [137]: data.loc['Colorado', ['two', 'three']]
Out[137]:
```

two	5
three	6

Name: Colorado, dtype: int64

We'll then perform some similar selections with integers using `iloc`:

```
In [138]: data.iloc[2, [3, 0, 1]]
Out[138]:
```

four	11
one	8
two	9

Name: Utah, dtype: int64

```
In [139]: data.iloc[2]
Out[139]:
```

one	8
two	9
three	10
four	11

Name: Utah, dtype: int64

```
In [140]: data.iloc[[1, 2], [3, 0, 1]]
Out[140]:
```

	four	one	two
Colorado	7	0	5
Utah	11	8	9

Both indexing functions work with slices in addition to single labels or lists of labels:

```
In [141]: data.loc[:'Utah', 'two']
Out[141]:
```

Ohio	0
Colorado	5
Utah	9

Name: two, dtype: int64

```
In [142]: data.iloc[:, :3][data.three > 5]
Out[142]:
```

	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, [Table 5-4](#) provides a short summary of many of them. As you'll see later, there are a number of additional options for working with hierarchical indexes.



When originally 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 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.

The `ix` indexing operator still exists, but it is deprecated. I do not recommend using it.

Table 5-4. 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 or 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 integer position

Type	Notes
<code>df.iloc[:, where]</code>	Selects single column or 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)
reindex method	Select either rows or columns by labels
<code>get_value</code> , <code>set_value</code> methods	Select single value by row and column label

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 might not expect the following code to generate an error:

```
ser = pd.Series(np.arange(3.))
ser
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 [144]: ser
Out[144]:
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 [145]: ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])

In [146]: ser2[-1]
Out[146]: 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 [147]: ser[:1]
Out[147]:
0    0.0
dtype: float64

In [148]: ser.loc[:1]
Out[148]:
0    0.0
1    1.0
```

```
dtype: float64
```

```
In [149]: ser.iloc[:1]
```

```
Out[149]:
```

```
0    0.0
```

```
dtype: float64
```

Arithmetic and Data Alignment

An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes. When you are 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 [150]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
```

```
In [151]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],  
.....:                  index=['a', 'c', 'e', 'f', 'g'])
```

```
In [152]: s1
```

```
Out[152]:
```

```
a    7.3
```

```
c   -2.5
```

```
d    3.4
```

```
e    1.5
```

```
dtype: float64
```

```
In [153]: s2
```

```
Out[153]:
```

```
a   -2.1
```

```
c    3.6
```

```
e   -1.5
```

```
f    4.0
```

```
g    3.1
```

```
dtype: float64
```

Adding these together yields:

```
In [154]: s1 + s2
```

```
Out[154]:
```

```
a    5.2
```

```
c    1.1
```

```
d   NaN
```

```
e    0.0
```

```
f   NaN
```

```
g   NaN
```

```
dtype: float64
```

The internal data alignment introduces missing 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 [155]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
.....:                       index=['Ohio', 'Texas', 'Colorado'])
```

```
In [156]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
.....:                       index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

```
In [157]: df1
```

```
Out[157]:
```

	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 [158]: df2
```

```
Out[158]:
```

	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 [159]: df1 + df2
```

```
Out[159]:
```

	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

Since the 'c' and 'e' columns are not found in both DataFrame objects, they appear as all missing in the result. The same holds for the rows whose labels are not common to both objects.

If you add DataFrame objects with no column or row labels in common, the result will contain all nulls:

```
In [160]: df1 = pd.DataFrame({'A': [1, 2]})
```

```
In [161]: df2 = pd.DataFrame({'B': [3, 4]})
```

```
In [162]: df1
```

```
Out[162]:
```

	A
0	1
1	2

```
In [163]: df2
```



```
Out[163]:
  B
0  3
1  4
```

```
In [164]: df1 - df2
```

```
Out[164]:
  A  B
0 NaN NaN
1 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 [165]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
.....:                       columns=list('abcd'))
```

```
In [166]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
.....:                       columns=list('abcde'))
```

```
In [167]: df2.loc[1, 'b'] = np.nan
```

```
In [168]: df1
```

```
Out[168]:
   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 [169]: df2
```

```
Out[169]:
   a  b  c  d  e
0  0.0  1.0  2.0  3.0  4.0
1  5.0  NaN  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 [170]: df1 + df2
```

```
Out[170]:
   a  b  c  d  e
0  0.0  2.0  4.0  6.0 NaN
1  9.0  NaN 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 [171]: df1.add(df2, fill_value=0)
```

```
Out[171]:
```

```

      a      b      c      d      e
0  0.0  2.0  4.0  6.0  4.0
1  9.0  5.0 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

```

See [Table 5-5](#) for a listing of Series and DataFrame methods for arithmetic. Each of them has a counterpart, starting with the letter `r`, that has arguments flipped. So these two statements are equivalent:

```

In [172]: 1 / df1
Out[172]:
      a      b      c      d
0  inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909

```

```

In [173]: df1.rdiv(1)
Out[173]:
      a      b      c      d
0  inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909

```

Relatedly, when reindexing a Series or DataFrame, you can also specify a different fill value:

```

In [174]: df1.reindex(columns=df2.columns, fill_value=0)
Out[174]:
      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 5-5. Flexible arithmetic methods

Method	Description
<code>add</code> , <code>radd</code>	Methods for addition (+)
<code>sub</code> , <code>rsub</code>	Methods for subtraction (-)
<code>div</code> , <code>rdiv</code>	Methods for division (/)
<code>floordiv</code> , <code>rfloordiv</code>	Methods for floor division (//)
<code>mul</code> , <code>rmul</code>	Methods for multiplication (*)
<code>pow</code> , <code>rpow</code>	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 two-dimensional array and one of its rows:

```
In [175]: arr = np.arange(12.).reshape((3, 4))
```

```
In [176]: arr
```

```
Out[176]:  
array([[ 0.,  1.,  2.,  3.],  
       [ 4.,  5.,  6.,  7.],  
       [ 8.,  9., 10., 11.]])
```

```
In [177]: arr[0]
```

```
Out[177]: array([ 0.,  1.,  2.,  3.])
```

```
In [178]: arr - arr[0]
```

```
Out[178]:  
array([[ 0.,  0.,  0.,  0.],  
       [ 4.,  4.,  4.,  4.],  
       [ 8.,  8.,  8.,  8.]])
```

When we subtract `arr[0]` from `arr`, the subtraction is performed once for each row. This is referred to as *broadcasting* and is explained in more detail as it relates to general NumPy arrays in [Appendix A](#). Operations between a DataFrame and a Series are similar:

```
In [179]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),  
.....:                        columns=list('bde'),  
.....:                        index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

```
In [180]: series = frame.iloc[0]
```

```
In [181]: frame
```

```
Out[181]:  
      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 [182]: series
```

```
Out[182]:  
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 [183]: frame - series
```

```
Out[183]:  
      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 [184]: series2 = pd.Series(range(3), index=['b', 'e', 'f'])
```

```
In [185]: frame + series2
```

```
Out[185]:
```

	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 [186]: series3 = frame['d']
```

```
In [187]: frame
```

```
Out[187]:
```

	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 [188]: series3
```

```
Out[188]:
```

Utah	1.0
Ohio	4.0
Texas	7.0
Oregon	10.0

Name: d, dtype: float64

```
In [189]: frame.sub(series3, axis='index')
```

```
Out[189]:
```

	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='index'` or `axis=0`) and broadcast across.

Function Application and Mapping

NumPy ufuncs (element-wise array methods) also work with pandas objects:

```
In [190]: frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
.....:                          index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

```
In [191]: frame
Out[191]:
```

	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 [192]: np.abs(frame)
Out[192]:
```

	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 one-dimensional arrays to each column or row. DataFrame's `apply` method does exactly this:

```
In [193]: f = lambda x: x.max() - x.min()
```

```
In [194]: frame.apply(f)
Out[194]:
```

b	1.802165
d	1.684034
e	2.689627

dtype: float64

Here the function `f`, which computes the difference between the maximum and minimum of a Series, is invoked once on each column in `frame`. The result is a Series having the columns of `frame` as its index.

If you pass `axis='columns'` to `apply`, the function will be invoked once per row instead:

```
In [195]: frame.apply(f, axis='columns')
Out[195]:
```

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 [196]: def f(x):
.....:     return pd.Series([x.min(), x.max()], index=['min', 'max'])
```

```
In [197]: frame.apply(f)
```

```
Out[197]:
```

	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 `apply` `map`:

```
In [198]: format = lambda x: '%.2f' % x
```

```
In [199]: frame.applymap(format)
```

```
Out[199]:
```

	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 [200]: frame['e'].map(format)
```

```
Out[200]:
```

Utah	-0.52
Ohio	1.39
Texas	0.77
Oregon	-1.30

Name: e, dtype: object

Sorting and Ranking

Sorting a dataset 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 [201]: obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
```

```
In [202]: obj.sort_index()
```

```
Out[202]:
```

a	1
b	2
c	3
d	0

dtype: int64

With a DataFrame, you can sort by index on either axis:

```
In [203]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
.....:                          index=['three', 'one'],
.....:                          columns=['d', 'a', 'b', 'c'])
```

```
In [204]: frame.sort_index()
```

```
Out[204]:
   d  a  b  c
one  4  5  6  7
three 0  1  2  3
```

```
In [205]: frame.sort_index(axis=1)
```

```
Out[205]:
   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 [206]: frame.sort_index(axis=1, ascending=False)
```

```
Out[206]:
   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 [207]: obj = pd.Series([4, 7, -3, 2])
```

```
In [208]: obj.sort_values()
```

```
Out[208]:
2    -3
3     2
0     4
1     7
dtype: int64
```

Any missing values are sorted to the end of the Series by default:

```
In [209]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
```

```
In [210]: obj.sort_values()
```

```
Out[210]:
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 [211]: frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
```

```
In [212]: frame
```

```
Out[212]:
   a  b
0  0  4
1  1  7
2  0 -3
3  1  2
```

```
0 0 4
1 1 7
2 0 -3
3 1 2
```

```
In [213]: frame.sort_values(by='b')
```

```
Out[213]:
```

```
  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 [214]: frame.sort_values(by=['a', 'b'])
```

```
Out[214]:
```

```
  a  b
2  0 -3
0  0  4
3  1  2
1  1  7
```

Ranking assigns 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 [215]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
```

```
In [216]: obj.rank()
```

```
Out[216]:
```

```
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 in which they're observed in the data:

```
In [217]: obj.rank(method='first')
```

```
Out[217]:
```

```
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
```

```
dtype: float64
```


Here, instead of using the average rank 6.5 for the entries 0 and 2, they instead have been set to 6 and 7 because label 0 precedes label 2 in the data.

You can rank in descending order, too:

```
# Assign tie values the maximum rank in the group
In [218]: obj.rank(ascending=False, method='max')
Out[218]:
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 5-6](#) for a list of tie-breaking methods available.

DataFrame can compute ranks over the rows or the columns:

```
In [219]: frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
.....:                          'c': [-2, 5, 8, -2.5]})
```

```
In [220]: frame
Out[220]:
```

```
   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 [221]: frame.rank(axis='columns')
Out[221]:
```

```
   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 5-6. 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 Labels

Up until now all of the examples we've looked at 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 [222]: obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
```

```
In [223]: obj
```

```
Out[223]:
```

```
a    0
```

```
a    1
```

```
b    2
```

```
b    3
```

```
c    4
```

```
dtype: int64
```

The index's `is_unique` property can tell you whether its labels are unique or not:

```
In [224]: obj.index.is_unique
```

```
Out[224]: False
```

Data selection is one of the main things that behaves differently with duplicates. Indexing a label with multiple entries returns a Series, while single entries return a scalar value:

```
In [225]: obj['a']
```

```
Out[225]:
```

```
a    0
```

```
a    1
```

```
dtype: int64
```

```
In [226]: obj['c']
```

```
Out[226]: 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 [227]: df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])
```

```
In [228]: df
```

```
Out[228]:
```

```
      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 [229]: df.loc['b']
```

```
Out[229]:
```

```
      0          1          2
```

```
b 1.669025 -0.438570 -0.539741
b 0.476985 3.248944 -1.021228
```

5.3 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 [230]: 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 [231]: df
Out[231]:
   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 [232]: df.sum()
Out[232]:
one    9.25
two   -5.80
dtype: float64
```

Passing `axis='columns'` or `axis=1` sums across the columns instead:

```
In [233]: df.sum(axis='columns')
Out[233]:
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 with the `skipna` option:

```
In [234]: df.mean(axis='columns', skipna=False)
Out[234]:
a    NaN
b    1.300
c    NaN
```

```
d -0.275
dtype: float64
```

See [Table 5-7](#) for a list of common options for each reduction method.

Table 5-7. Options for reduction methods

Method	Description
axis	Axis to reduce over; 0 for DataFrame's rows and 1 for columns
skipna	Exclude missing values; True by default
level	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 [235]: df.idxmax()
Out[235]:
one    b
two    d
dtype: object
```

Other methods are *accumulations*:

```
In [236]: df.cumsum()
Out[236]:
   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 [237]: df.describe()
Out[237]:
   count  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 alternative summary statistics:

```
In [238]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)
```

```
In [239]: obj.describe()
Out[239]:
count    16
```

```
unique      3
top         a
freq       8
dtype: object
```

See [Table 5-8](#) for a full list of summary statistics and related methods.

Table 5-8. 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 labels 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 (third moment) of values
kurt	Sample kurtosis (fourth moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute first 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-datareader` package. If you don't have it installed already, it can be obtained via `conda` or `pip`:

```
conda install pandas-datareader
```

I use the `pandas_datareader` module to download some data for a few stock tickers:

```
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()})
```



It's possible by the time you are reading this that Yahoo! Finance no longer exists since Yahoo! was acquired by Verizon in 2017. Refer to the pandas-datareader documentation online for the latest functionality.

I now compute percent changes of the prices, a time series operation which will be explored further in [Chapter 11](#):

```
In [242]: returns = price.pct_change()

In [243]: returns.tail()
Out[243]:
```

	AAPL	GOOG	IBM	MSFT
Date				
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 [244]: returns['MSFT'].corr(returns['IBM'])
Out[244]: 0.49976361144151144

In [245]: returns['MSFT'].cov(returns['IBM'])
Out[245]: 8.8706554797035462e-05
```

Since `MSFT` is a valid Python attribute, we can also select these columns using more concise syntax:

```
In [246]: returns.MSFT.corr(returns.IBM)
Out[246]: 0.49976361144151144
```

`DataFrame`'s `corr` and `cov` methods, on the other hand, return a full correlation or covariance matrix as a `DataFrame`, respectively:

```
In [247]: returns.corr()
Out[247]:
```

	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 [248]: returns.cov()
Out[248]:
```

	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 [249]: returns.corrwith(returns.IBM)
Out[249]:
```

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 [250]: returns.corrwith(volume)
Out[250]:
```

AAPL	-0.075565
GOOG	-0.007067
IBM	-0.204849
MSFT	-0.092950

dtype: float64

Passing `axis='columns'` does things row-by-row instead. In all cases, the data points are aligned by label before the correlation is computed.

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 [251]: obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

The first function is `unique`, which gives you an array of the unique values in a Series:

```
In [252]: uniques = obj.unique()

In [253]: uniques
Out[253]: 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 [254]: obj.value_counts()
Out[254]:
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 [255]: pd.value_counts(obj.values, sort=False)
Out[255]:
a    3
b    2
c    3
d    1
dtype: int64
```

`isin` performs a vectorized set membership check and can be useful in filtering a dataset down to a subset of values in a Series or column in a DataFrame:

```
In [256]: obj
Out[256]:
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object
```

```
In [257]: mask = obj.isin(['b', 'c'])
```

```
In [258]: mask
Out[258]:
0    True
1    False
2    False
3    False
4    False
5    True
6    True
7    True
8    True
dtype: bool
```

```
In [259]: obj[mask]
Out[259]:
```



```
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 [260]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])
```

```
In [261]: unique_vals = pd.Series(['c', 'b', 'a'])
```

```
In [262]: pd.Index(unique_vals).get_indexer(to_match)
```

```
Out[262]: array([0, 2, 1, 1, 0, 2])
```

See [Table 5-9](#) for a reference on these methods.

Table 5-9. Unique, value counts, and set membership methods

Method	Description
<code>isin</code>	Compute boolean array indicating whether each Series value is contained in the passed sequence of values
<code>match</code>	Compute integer indices for each value in an array into another array of distinct values; helpful for data alignment and join-type operations
<code>unique</code>	Compute array of unique values in a Series, returned in the order observed
<code>value_counts</code>	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 [263]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
.....:                        'Qu2': [2, 3, 1, 2, 3],
.....:                        'Qu3': [1, 5, 2, 4, 4]})
```

```
In [264]: data
```

```
Out[264]:
```

```
   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 [265]: result = data.apply(pd.value_counts).fillna(0)
```

```
In [266]: result
```

```
Out[266]:
```

	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

Here, the row labels in the result are the distinct values occurring in all of the columns. The values are the respective counts of these values in each column.

5.4 Conclusion

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.

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.

6.1 Reading and Writing Data in Text Format

pandas features a number of functions for reading tabular data as a DataFrame object. [Table 6-1](#) summarizes 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; use comma as default delimiter
<code>read_table</code>	Load delimited data from a file, URL, or file-like object; use tab (' <code>\t</code> ') as default delimiter
<code>read_fwf</code>	Read data in fixed-width column format (i.e., no delimiters)
<code>read_clipboard</code>	Version of <code>read_table</code> that reads data from the clipboard; 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 HDF5 files written by pandas
<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

Function	Description
<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 works, so if you're struggling to read a particular file, there might be a similar enough example to help you find the right parameters.

Some of these functions, like `pandas.read_csv`, perform *type inference*, because the column data types are not part of the data format. That means you don't necessarily have to specify which columns are numeric, integer, boolean, or string. Other data formats, like HDF5, Feather, and msgpack, have the data types stored in the format.

Handling dates and other custom types can require extra effort. 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
```



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.

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 specified 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

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', 'd', 'message'])
```

```
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=['key1', 'key2'])

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. 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\n']
```

While you could do some munging by hand, the fields here are separated by a variable amount of whitespace. In these cases, you can pass a regular expression as a delimiter for `read_table`. 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 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=['NULL'])
```

```
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': ['two']}
```

```
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 lists some frequently used options in `pandas.read_csv` and `pd.read_table`.

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 <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>

Argument	Description
<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(s) to split comments off the end of lines.
<code>parse_dates</code>	Attempt to parse data to <code>datetime</code> ; <code>False</code> by default. If <code>True</code> , will attempt to parse all columns. Otherwise can specify a 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 (e.g., if date/time split across two columns).
<code>keep_date_col</code>	If joining columns to parse date, keep the joined columns; <code>False</code> by default.
<code>converters</code>	Dict containing column number of name mapping to functions (e.g., <code>{ 'foo' : f }</code> would apply the function <code>f</code> to all values in the 'foo' column).
<code>dayfirst</code>	When parsing potentially ambiguous dates, treat as international format (e.g., 7/6/2012 -> June 7, 2012); <code>False</code> by default.
<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 <code>TextParser</code> 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 (e.g., 'utf-8' for UTF-8 encoded text).
<code>squeeze</code>	If the parsed data only contains one column, return a <code>Series</code> .
<code>thousands</code>	Separator for thousands (e.g., ',' or '.').

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.

Before we look at a large file, we make the pandas display settings more compact:

```
In [33]: pd.options.display.max_rows = 10
```

Now we have:

```
In [34]: result = pd.read_csv('examples/ex6.csv')
```

```
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 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 a file in pieces, specify a `chunksize` as a number of rows:

```
In [37]: chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

In [38]: chunker
Out[38]: <pandas.io.parsers.TextFileReader at 0x7f6b1e2672e8>
```

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 that enables you to read pieces of an arbitrary size.

Writing Data to Text Format

Data can also be exported to a delimited format. Let's consider one of the CSV files read before:

```
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', 'c'])
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
```

Working with Delimited Formats

It's possible to load most forms of tabular data from disk using functions like `pd.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"
```

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 with any quote characters removed:

```
In [56]: for line in reader:
...:     print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']
```

From there, it's up to you to do the wrangling necessary to put the data in the form that you need it. Let's take this step by step. First, we read the file into a list of lines:

```
In [57]: with open('examples/ex7.csv') as f:
...:     lines = list(csv.reader(f))
```

Then, we split the lines into the header line and the data lines:

```
In [58]: header, values = lines[0], lines[1:]
```

Then we can create a dictionary of data columns using a dictionary comprehension and the expression `zip(*values)`, which transposes rows to columns:

```
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. To define a new format with a different delimiter, string quoting convention, or line terminator, we define 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)
```

We can also give individual CSV dialect parameters 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> .

Argument	Description
quoting	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> .
skipinitialspace	Ignore whitespace after each delimiter; default is <code>False</code> .
doublequote	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 <code>quoting</code> is set to <code>csv.QUOTE_NONE</code> ; disabled by default.



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": 30, "pets": ["Zeus", "Zuko"]},
               {"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': 30, 'name': 'Scott', 'pets': ['Zeus', 'Zuko']},  
              {'age': 38, 'name': 'Katie', 'pets': ['Sixes', 'Stache', 'Cisco']}]}
```

`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 dicts (which were previously 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   30  
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 examples/example.json
```

```
[{"a": 1, "b": 2, "c": 3},  
 {"a": 4, "b": 5, "c": 6},  
 {"a": 7, "b": 8, "c": 9}]
```

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 [Chapter 7](#).

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`, Beautiful Soup, and `html5lib`. While `lxml` is comparatively much faster in general, 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 Beautiful Soup 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.¹ First, you must install some additional libraries used by `read_html`:

```
conda install lxml
pip install beautifulsoup4 html5lib
```

If you are not using `conda`, `pip install lxml` will likely also work.

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	

¹ For the full list, see <https://www.fdic.gov/bank/individual/failed/banklist.html>.

3	Trust Company Bank	Memphis TN	9956
4	North Milwaukee State Bank	Milwaukee WI	20364
	Acquiring Institution	Closing Date	Updated Date
0	Today's Bank	September 23, 2016	November 17, 2016
1	United Bank	August 19, 2016	November 17, 2016
2	First-Citizens Bank & Trust Company	May 6, 2016	September 6, 2016
3	The Bank of Fayette County	April 29, 2016	September 6, 2016
4	First-Citizens Bank & Trust Company	March 11, 2016	June 16, 2016

Because `failures` has many columns, pandas inserts a line break character `\`.

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
```

Parsing XML with `lxml.objectify`

XML (eXtensible Markup Language) is another common structured data format supporting hierarchical, nested data with metadata. The book you are currently reading was actually created from a series of large XML documents.

Earlier, I showed the `pandas.read_html` function, which uses either `lxml` or `BeautifulSoup` 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**. 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 operational
systemwide. The availability rate is based on physical observations performed
the morning of regular business days only. This is a new indicator the agency
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` returns 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 0x7f6b15817748>
```

```
In [85]: root.get('href')  
Out[85]: 'http://www.google.com'
```

```
In [86]: root.text  
Out[86]: 'Google'
```

6.2 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 that 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')
```

You can read any “pickled” object stored in a file 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
```



`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 backward 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

A compressable column-oriented binary format based on the Blosc compression library.

Feather

A cross-language column-oriented file format I designed with the R programming community’s [Hadley Wickham](#). Feather uses the [Apache Arrow](#) columnar memory format.

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, including 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. HDF5 can be a good choice for working with very 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,ncols->1,indexers->
[index])
/obj3          frame_table  (typ->appendable,nrows->100,ncols->1,indexers->
[index])

```

Objects contained in the HDF5 file can then be retrieved with 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 <= 15'])
Out[99]:
      a
10  1.007189
11 -1.296221
12  0.274992
13  0.228913
14  1.352917
15  0.886429

In [100]: store.close()

```

The put is an explicit version of the store['obj2'] = frame method but allows us to set other options like the storage format.

The pandas.read_hdf function gives you a shortcut to these tools:

```

In [101]: frame.to_hdf('mydata.h5', 'obj3', format='table')

```

```
In [102]: pd.read_hdf('mydata.h5', 'obj3', where=['index < 5'])
Out[102]:
      a
0 -0.204708
1  0.478943
2 -0.519439
3 -0.555730
4  1.965781
```



If you are processing data that is stored on remote servers, like Amazon S3 or HDFS, using a different binary format designed for distributed storage like [Apache Parquet](#) may be more suitable. Python for Parquet and other such storage formats is still developing, so I do not write about them in this book.

If you work with large quantities of data locally, I would encourage you to explore PyTables and h5py to see how they can suit your needs. Since many data analysis problems are I/O-bound (rather than CPU-bound), using a tool like HDF5 can massively accelerate your applications.



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` with `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 simply pass the filename 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()
```

You can also pass a file path to `to_excel` and avoid the `ExcelWriter`:

```
In [111]: frame.to_excel('examples/ex2.xlsx')
```

6.3 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.

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 [113]: import requests
```

```
In [114]: url = 'https://api.github.com/repos/pandas-dev/pandas/issues'
```

```
In [115]: resp = requests.get(url)
```

```
In [116]: resp
```

```
Out[116]: <Response [200]>
```

The `Response` object's `json` method will return a dictionary containing JSON parsed into native Python objects:

```
In [117]: data = resp.json()
```

```
In [118]: data[0]['title']
```

```
Out[118]: 'Period does not round down for frequencies less than 1 hour'
```

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 [119]: issues = pd.DataFrame(data, columns=['number', 'title',
.....:                                     'labels', 'state'])

In [120]: issues
Out[120]:
   number                                     title \
0    17666  Period does not round down for frequencies les...
1    17665                DOC: improve docstring of function where
2    17664                COMPAT: skip 32-bit test on int repr
3    17662                implement Delegator class
4    17654  BUG: Fix series rename called with str alterin...
..     ...
25   17603  BUG: Correctly localize naive datetime strings...
26   17599                core.dtypes.generic --> cython
27   17596  Merge cdate_range functionality into bdate_range
28   17587  Time Grouper bug fix when applied for list gro...
29   17583  BUG: fix tz-aware DatetimeIndex + TimedeltaInd...
      labels state
0      [] open
1  [{'id': 134699, 'url': 'https://api.github.com... open
2  [{'id': 563047854, 'url': 'https://api.github.... open
3      [] open
4  [{'id': 76811, 'url': 'https://api.github.com/... open
..     ...
25  [{'id': 76811, 'url': 'https://api.github.com/... open
26  [{'id': 49094459, 'url': 'https://api.github.c... open
27  [{'id': 35818298, 'url': 'https://api.github.c... open
28  [{'id': 233160, 'url': 'https://api.github.com... open
29  [{'id': 76811, 'url': 'https://api.github.com/... 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.

6.4 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:

```

In [121]: import sqlite3

In [122]: query = """
.....: CREATE TABLE test

```



```
.....: (a VARCHAR(20), b VARCHAR(20),
.....: c REAL,          d INTEGER
.....: );"""
```

```
In [123]: con = sqlite3.connect('mydata.sqlite')
```

```
In [124]: con.execute(query)
```

```
Out[124]: <sqlite3.Cursor at 0x7f6b12a50f10>
```

```
In [125]: con.commit()
```

Then, insert a few rows of data:

```
In [126]: data = [('Atlanta', 'Georgia', 1.25, 6),
.....:             ('Tallahassee', 'Florida', 2.6, 3),
.....:             ('Sacramento', 'California', 1.7, 5)]
```

```
In [127]: stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"
```

```
In [128]: con.executemany(stmt, data)
```

```
Out[128]: <sqlite3.Cursor at 0x7f6b15c66ce0>
```

```
In [129]: con.commit()
```

Most Python SQL drivers (PyODBC, pycpg2, MySQLdb, pymssql, etc.) return a list of tuples when selecting data from a table:

```
In [130]: cursor = con.execute('select * from test')
```

```
In [131]: rows = cursor.fetchall()
```

```
In [132]: rows
```

```
Out[132]:
[('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 [133]: cursor.description
```

```
Out[133]:
 (('a', None, None, None, None, None),
 ('b', None, None, None, None, None),
 ('c', None, None, None, None, None),
 ('d', None, None, None, None, None))
```

```
In [134]: pd.DataFrame(rows, columns=[x[0] for x in cursor.description])
```

```
Out[134]:
```

	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](#) 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 before:

```
In [135]: import sqlalchemy as sqla
```

```
In [136]: db = sqla.create_engine('sqlite:///mydata.sqlite')
```

```
In [137]: pd.read_sql('select * from test', db)
```

```
Out[137]:
```

	a	b	c	d
0	Atlanta	Georgia	1.25	6
1	Tallahassee	Florida	2.60	3
2	Sacramento	California	1.70	5

6.5 Conclusion

Getting access to data is frequently the first step in the data analysis process. We have looked at a number of useful tools in this chapter that should help you get started. In the upcoming chapters we will dig deeper into data wrangling, data visualization, time series analysis, and other topics.

Data Cleaning and Preparation

During the course of doing data analysis and modeling, a significant amount of time is spent on data preparation: loading, cleaning, transforming, and rearranging. Such tasks are often reported to take up 80% or more of an analyst's time. Sometimes the way that data is stored in files or databases is not in the right format for a particular task. Many researchers choose to do ad hoc processing of data from one form to another using a general-purpose programming language, 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, provides 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 the `pandas` 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.

7.1 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` objects is somewhat imperfect, but it is functional for a lot of users. 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, 'avocado'])
```

```
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. See [Table 7-1](#) for a list of some functions related to missing data handling.

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 'ffill' or 'bfill'.
<code>isnull</code>	Return 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 you always have the option to do it by hand using `pandas.isnull` and boolean indexing, 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 that 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
```

```
In [29]: df.iloc[:2, 2] = NA
```

```
In [30]: df
```

```
Out[30]:
   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 [31]: df.dropna()
```

```
Out[31]:
   0  1  2
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

```
In [32]: df.dropna(thresh=2)
```

```
Out[32]:
```

	0	1	2
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

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 [33]: df.fillna(0)
Out[33]:
```

	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 [34]: df.fillna({'1': 0.5, '2': 0})
Out[34]:
```

	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:

```
In [35]: _ = df.fillna(0, inplace=True)
```

```
In [36]: df
Out[36]:
```

	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 [37]: df = pd.DataFrame(np.random.randn(6, 3))
```

```
In [38]: df.iloc[2:, 1] = NA
```

```
In [39]: df.iloc[4:, 2] = NA
```

```
In [40]: df
```

```
Out[40]:
```

	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 [41]: df.fillna(method='ffill')
```

```
Out[41]:
```

	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 [42]: df.fillna(method='ffill', limit=2)
```

```
Out[42]:
```

	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 [43]: data = pd.Series([1., NA, 3.5, NA, 7])
```

```
In [44]: data.fillna(data.mean())
```

```
Out[44]:
```

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
value	Scalar value or dict-like object to use to fill missing values
method	Interpolation; by default 'ffill' if function called with no other arguments
axis	Axis to fill on; default axis=0
inplace	Modify the calling object without producing a copy
limit	For forward and backward filling, maximum number of consecutive periods to fill

7.2 Data Transformation

So far in this chapter we've been concerned with rearranging data. Filtering, cleaning, and other transformations 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 [45]: data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
    ....:                      'k2': [1, 1, 2, 3, 3, 4, 4]})
```

```
In [46]: data
```

```
Out[46]:
   k1 k2
0  one  1
1  two  1
2  one  2
3  two  3
4  one  3
5  two  4
6  two  4
```

The DataFrame method `data.duplicated` returns a boolean Series indicating whether each row is a duplicate (has been observed in a previous row) or not:

```
In [47]: data.duplicated()
```

```
Out[47]:
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

Relatedly, `drop_duplicates` returns a DataFrame where the duplicated array is False:

```
In [48]: data.drop_duplicates()
Out[48]:
   k1 k2
0  one  1
1  two  1
2  one  2
3  two  3
4  one  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 [49]: data['v1'] = range(7)

In [50]: data.drop_duplicates(['k1'])
Out[50]:
   k1 k2 v1
0  one  1  0
1  two  1  1
```

`drop_duplicates` and `drop_duplicates` by default keep the first observed value combination. Passing `keep='last'` will return the last one:

```
In [51]: data.drop_duplicates(['k1', 'k2'], keep='last')
Out[51]:
   k1 k2 v1
0  one  1  0
1  two  1  1
2  one  2  2
3  two  3  3
4  one  3  4
6  two  4  6
```

Transforming Data Using a Function or Mapping

For many datasets, 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 various kinds of meat:

```
In [52]: data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
    ....:                               'Pastrami', 'corned beef', 'Bacon',
    ....:                               'pastrami', 'honey ham', 'nova lox'],
    ....:                    'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
```

```
In [53]: data
Out[53]:
   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 are capitalized and others are not. Thus, we need to convert each value to lowercase using the `str.lower` Series method:

```
In [55]: lowercased = data['food'].str.lower()
```

```
In [56]: lowercased
```

```
Out[56]:
```

0	bacon
1	pulled pork
2	bacon
3	pastrami
4	corned beef
5	bacon
6	pastrami
7	honey ham
8	nova lox

```
Name: food, dtype: object
```

```
In [57]: data['animal'] = lowercased.map(meat_to_animal)
```

```
In [58]: data
```

```
Out[58]:
```

	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 [59]: data['food'].map(lambda x: meat_to_animal[x.lower()])
Out[59]:
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. As you've already seen, `map` can be used to modify a subset of values in an object but `replace` provides a simpler and more flexible way to do so. Let's consider this Series:

```
In [60]: data = pd.Series([1., -999., 2., -999., -1000., 3.])
```

```
In [61]: data
```

```
Out[61]:
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 (unless you pass `inplace=True`):

```
In [62]: data.replace(-999, np.nan)
```

```
Out[62]:
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 and then the substitute value:

```
In [63]: data.replace([-999, -1000], np.nan)
Out[63]:
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 [64]: data.replace([-999, -1000], [np.nan, 0])
Out[64]:
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 [65]: data.replace({-999: np.nan, -1000: 0})
Out[65]:
0      1.0
1      NaN
2      2.0
3      NaN
4      0.0
5      3.0
dtype: float64
```



The `data.replace` method is distinct from `data.str.replace`, which performs string substitution element-wise. We look at these string methods on Series later in the chapter.

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. You can also modify the axes in-place without creating a new data structure. Here's a simple example:

```
In [66]: 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 [67]: transform = lambda x: x[:4].upper()
```

```
In [68]: data.index.map(transform)
```

```
Out[68]: Index(['OHIO', 'COLO', 'NEW'], dtype='object')
```

You can assign to `index`, modifying the DataFrame in-place:

```
In [69]: data.index = data.index.map(transform)
```

```
In [70]: data
```

```
Out[70]:
```

	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 dataset without modifying the original, a useful method is `rename`:

```
In [71]: data.rename(index=str.title, columns=str.upper)
```

```
Out[71]:
```

	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 [72]: data.rename(index={'OHIO': 'INDIANA'},
....:                 columns={'three': 'peekaboo'})
```

```
Out[72]:
```

	one	two	peekaboo	four
INDIANA	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

`rename` saves you from the chore of copying the DataFrame manually and assigning to its `index` and `columns` attributes. Should you wish to modify a dataset in-place, pass `inplace=True`:

```
In [73]: data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
```

```
In [74]: data
```

```
Out[74]:
```

	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 [75]: 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 [76]: bins = [18, 25, 35, 60, 100]
```

```
In [77]: cats = pd.cut(ages, bins)
```

```
In [78]: cats
```

```
Out[78]:
```

```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35, 60], (35, 60], (25, 35]]
```

```
Length: 12
```

```
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

The object pandas returns is a special `Categorical` object. The output you see describes the bins computed by `pandas.cut`. You can treat it like an array of strings indicating the bin name; internally it contains a `categories` array specifying the distinct category names along with a labeling for the ages data in the `codes` attribute:

```
In [79]: cats.codes
```

```
Out[79]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
```

```
In [80]: cats.categories
```

```
Out[80]:
```

```
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]]
              closed='right',
              dtype='interval[int64]')
```

```
In [81]: pd.value_counts(cats)
```

```
Out[81]:
```

```
(18, 25]    5
(35, 60]    3
(25, 35]    3
(60, 100]   1
dtype: int64
```

Note that `pd.value_counts(cats)` are the bin counts for the result of `pandas.cut`.

Consistent with mathematical notation for intervals, a parenthesis means that the side is *open*, while the square bracket means it is *closed* (inclusive). You can change which side is closed by passing `right=False`:

```
In [82]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)
Out[82]:
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36,
 61), [36, 61), [26, 36)]
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

You can also pass your own bin names by passing a list or array to the `labels` option:

```
In [83]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
```

```
In [84]: pd.cut(ages, bins, labels=group_names)
Out[84]:
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, Mid
dleAged, YoungAdult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

If you pass an integer number of bins to `cut` 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 [85]: data = np.random.rand(20)

In [86]: pd.cut(data, 4, precision=2)
Out[86]:
[(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.97], (0.34, 0.55], ..., (0.34
, 0.55], (0.34, 0.55], (0.55, 0.76], (0.34, 0.55], (0.12, 0.34]]
Length: 20
Categories (4, interval[float64]): [(0.12, 0.34] < (0.34, 0.55] < (0.55, 0.76] <
(0.76, 0.97]]
```

The `precision=2` option limits the decimal precision to two digits.

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 [87]: data = np.random.randn(1000) # Normally distributed

In [88]: cats = pd.qcut(data, 4) # Cut into quartiles

In [89]: cats
Out[89]:
[(-0.0265, 0.62], (0.62, 3.928], (-0.68, -0.0265], (0.62, 3.928], (-0.0265, 0.62]
, ..., (-0.68, -0.0265], (-0.68, -0.0265], (-2.95, -0.68], (0.62, 3.928], (-0.68,
-0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.68] < (-0.68, -0.0265] < (-0.0265,
0.62] <
(0.62, 3.928]]
```



```
In [90]: pd.value_counts(cats)
Out[90]:
(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 [91]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
Out[91]:
[(-0.0265, 1.286], (-0.0265, 1.286], (-1.187, -0.0265], (-0.0265, 1.286], (-0.0265, 1.286], ..., (-1.187, -0.0265], (-1.187, -0.0265], (-2.95, -1.187], (-0.0265, 1.286], (-1.187, -0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -1.187] < (-1.187, -0.0265] < (-0.0265, 1.286] < (1.286, 3.928]]
```

We'll return to `cut` and `qcut` later in the chapter during our discussion of 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 [92]: data = pd.DataFrame(np.random.randn(1000, 4))

In [93]: data.describe()
Out[93]:
```

	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 3 in absolute value:

```
In [94]: col = data[2]

In [95]: col[np.abs(col) > 3]
Out[95]:
41    -3.399312
136   -3.745356
Name: 2, dtype: float64
```

To select all rows having a value exceeding 3 or -3, you can use the `any` method on a boolean DataFrame:

```
In [96]: data[(np.abs(data) > 3).any(1)]
Out[96]:
```

	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 [97]: data[np.abs(data) > 3] = np.sign(data) * 3
```

```
In [98]: data.describe()
```

```
Out[98]:
```

	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 statement `np.sign(data)` produces 1 and -1 values based on whether the values in `data` are positive or negative:

```
In [99]: np.sign(data).head()
```

```
Out[99]:
```

	0	1	2	3
0	-1.0	1.0	-1.0	1.0
1	1.0	-1.0	1.0	-1.0
2	1.0	1.0	1.0	-1.0
3	-1.0	-1.0	1.0	-1.0
4	-1.0	1.0	-1.0	-1.0

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 [100]: df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))
```

```
In [101]: sampler = np.random.permutation(5)
```

```
In [102]: sampler  
Out[102]: array([3, 1, 4, 2, 0])
```

That array can then be used in `iloc`-based indexing or the equivalent `take` function:

```
In [103]: df  
Out[103]:  
   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 [104]: df.take(sampler)  
Out[104]:  
   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, you can use the `sample` method on Series and DataFrame:

```
In [105]: df.sample(n=3)  
Out[105]:  
   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 (to allow repeat choices), pass `replace=True` to `sample`:

```
In [106]: choices = pd.Series([5, 7, -1, 6, 4])  
  
In [107]: draws = choices.sample(n=10, replace=True)
```

```
In [108]: draws  
Out[108]:  
4  4  
1  7  
4  4  
2 -1  
0  5  
3  6  
1  7
```

```
4 4
0 5
4 4
dtype: int64
```

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 with k columns containing all 1s and 0s. 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 [109]: df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
.....:                      'data1': range(6)})
```

```
In [110]: pd.get_dummies(df['key'])
```

```
Out[110]:
   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 [111]: dummies = pd.get_dummies(df['key'], prefix='key')
```

```
In [112]: df_with_dummy = df[['data1']].join(dummies)
```

```
In [113]: df_with_dummy
```

```
Out[113]:
   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 look at the MovieLens 1M dataset, which is investigated in more detail in [Chapter 14](#):

```

In [114]: mnames = ['movie_id', 'title', 'genres']

In [115]: movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
.....:                             header=None, names=mnames)

In [116]: movies[:10]
Out[116]:
  movie_id          title          genres
0         1  Toy Story (1995)  Animation|Children's|Comedy
1         2    Jumanji (1995)  Adventure|Children's|Fantasy
2         3  Grumpier Old Men (1995)  Comedy|Romance
3         4  Waiting to Exhale (1995)  Comedy|Drama
4         5  Father of the Bride Part II (1995)  Comedy
5         6      Heat (1995)  Action|Crime|Thriller
6         7    Sabrina (1995)  Comedy|Romance
7         8  Tom and Huck (1995)  Adventure|Children's
8         9  Sudden Death (1995)  Action
9         10  GoldenEye (1995)  Action|Adventure|Thriller

```

Adding indicator variables for each genre requires a little bit of wrangling. First, we extract the list of unique genres in the dataset:

```

In [117]: all_genres = []

In [118]: for x in movies.genres:
.....:     all_genres.extend(x.split('|'))

In [119]: genres = pd.unique(all_genres)

```

Now we have:

```

In [120]: genres
Out[120]:
array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
      'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
      'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir',
      'Western'], dtype=object)

```

One way to construct the indicator DataFrame is to start with a DataFrame of all zeros:

```

In [121]: zero_matrix = np.zeros((len(movies), len(genres)))

In [122]: 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 [123]: gen = movies.genres[0]
```

```
In [124]: gen.split('|')
```

```
Out[124]: ['Animation', 'Children's', 'Comedy']
```

```
In [125]: dummies.columns.get_indexer(gen.split('|'))
```

```
Out[125]: array([0, 1, 2])
```

Then, we can use `.iloc` to set values based on these indices:

```
In [126]: for i, gen in enumerate(movies.genres):
```

```
.....:     indices = dummies.columns.get_indexer(gen.split('|'))
```

```
.....:     dummies.iloc[i, indices] = 1
```

```
.....:
```

Then, as before, you can combine this with `movies`:

```
In [127]: movies_windic = movies.join(dummies.add_prefix('Genre_'))
```

```
In [128]: movies_windic.iloc[0]
```

```
Out[128]:
```

```
movie_id 1
```

```
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
```



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, and then wrap the result in a DataFrame.

A useful recipe for statistical applications is to combine `get_dummies` with a discretization function like `cut`:

```
In [129]: np.random.seed(12345)
```

```
In [130]: values = np.random.rand(10)
```

```
In [131]: values
```

```
Out[131]:
```

```
array([[ 0.9296,  0.3164,  0.1839,  0.2046,  0.5677,  0.5955,  0.9645,  
        0.6532,  0.7489,  0.6536]])
```

```
In [132]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
```

```
In [133]: pd.get_dummies(pd.cut(values, bins))
```

```
Out[133]:
```

	(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

We set the random seed with `numpy.random.seed` to make the example deterministic. We will look again at `pandas.get_dummies` later in the book.

7.3 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 [134]: val = 'a,b, guido'
```

```
In [135]: val.split(',')
```

```
Out[135]: ['a', 'b', ' guido']
```

`split` is often combined with `strip` to trim whitespace (including line breaks):

```
In [136]: pieces = [x.strip() for x in val.split(',')]
```

```
In [137]: pieces
```

```
Out[137]: ['a', 'b', 'guido']
```

These substrings could be concatenated together with a two-colon delimiter using addition:

```
In [138]: first, second, third = pieces
```

```
In [139]: first + '::' + second + '::' + third
```

```
Out[139]: '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 [140]: '::'.join(pieces)
```

```
Out[140]: '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 [141]: 'guido' in val
```

```
Out[141]: True
```

```
In [142]: val.index(',')
```

```
Out[142]: 1
```

```
In [143]: val.find('::')
```

```
Out[143]: -1
```

Note the difference between `find` and `index` is that `index` raises an exception if the string isn't found (versus returning `-1`):

```
In [144]: val.index('::')
```

```
-----  
ValueError                                Traceback (most recent call last)
```

```
<ipython-input-144-280f8b2856ce> in <module>()
```

```
----> 1 val.index('::')
```

```
ValueError: substring not found
```

Relatedly, `count` returns the number of occurrences of a particular substring:

```
In [145]: val.count(',')
```

```
Out[145]: 2
```

`replace` will substitute occurrences of one pattern for another. It is commonly used to delete patterns, too, by passing an empty string:

```
In [146]: val.replace(',', '::')
```

```
Out[146]: 'a::b:: guido'
```

```
In [147]: val.replace(',', '')
```

```
Out[147]: 'ab guido'
```


See [Table 7-3](#) for a listing of some of Python’s string methods.

Regular expressions can also be used with many of these operations, as you’ll see.

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.
<code>index</code>	Return position of first character in substring if found in the string; raises <code>ValueError</code> if not found.
<code>find</code>	Return position of first character of <i>first</i> occurrence of substring in the string; like <code>index</code> , but returns <code>-1</code> if not found.
<code>rfind</code>	Return position of first character of <i>last</i> occurrence of substring in the string; returns <code>-1</code> if not found.
<code>replace</code>	Replace occurrences of string with another string.
<code>strip</code> , <code>rstrip</code> , <code>rstrip</code>	Trim whitespace, including newlines; equivalent to <code>x.strip()</code> (and <code>rstrip</code> , <code>rstrip</code> , respectively) for each element.
<code>split</code>	Break string into list of substrings using passed delimiter.
<code>lower</code>	Convert alphabet characters to lowercase.
<code>upper</code>	Convert alphabet characters to uppercase.
<code>casefold</code>	Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form.
<code>ljust</code> , <code>rjust</code>	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.



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 we 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 [148]: import re
In [149]: text = "foo  bar\t baz  \tqux"
In [150]: re.split('\s+', text)
Out[150]: ['foo', 'bar', 'baz', 'qux']
```

When you call `re.split('\s+', text)`, the regular expression is first *compiled*, and 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 [151]: regex = re.compile('\s+')
In [152]: regex.split(text)
Out[152]: ['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 [153]: regex.findall(text)
Out[153]: [' ', '\t ', ' \t']
```



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 email addresses:

```
In [155]: regex.findall(text)
Out[155]:
['dave@google.com',
'steve@gmail.com',
'rob@gmail.com',
'ryan@yahoo.com']
```

search returns a special match object for the first email address in the text. For the preceding regex, the match object can only tell us the start and end position of the pattern in the string:

```
In [156]: m = regex.search(text)
```

```
In [157]: m
```

```
Out[157]: <_sre.SRE_Match object; span=(5, 20), match='dave@google.com'>
```

```
In [158]: text[m.start():m.end()]
```

```
Out[158]: 'dave@google.com'
```

regex.match returns None, as it only will match if the pattern occurs at the start of the string:

```
In [159]: print(regex.match(text))
```

```
None
```

Relatedly, sub will return a new string with occurrences of the pattern replaced by the a new string:

```
In [160]: 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 three components: username, domain name, and domain suffix. To do this, put parentheses around the parts of the pattern to segment:

```
In [161]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'
```

```
In [162]: 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 [163]: m = regex.match('wesm@bright.net')
```

```
In [164]: m.groups()
```

```
Out[164]: ('wesm', 'bright', 'net')
```

findall returns a list of tuples when the pattern has groups:

```
In [165]: regex.findall(text)
```

```
Out[165]:
```

```
[('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 corresponds to the second, and so forth:

```
In [166]: print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))  
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. Table 7-4 provides a brief summary.

Table 7-4. Regular expression methods

Argument	Description
findall	Return all non-overlapping matching patterns in a string as a list
finditer	Like findall, but returns an iterator
match	Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise None
search	Scan string for match to pattern; returning a match object if so; unlike match, the match can be anywhere in the string as opposed to only at the beginning
split	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 dataset for analysis often requires a lot of string munging and regularization. To complicate matters, a column containing strings will sometimes have missing data:

```
In [167]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',  
.....:          'Rob': 'rob@gmail.com', 'Wes': np.nan}
```

```
In [168]: data = pd.Series(data)
```

```
In [169]: data
```

```
Out[169]:
```

```
Dave    dave@google.com  
Rob      rob@gmail.com  
Steve   steve@gmail.com  
Wes                NaN  
dtype: object
```

```
In [170]: data.isnull()
Out[170]:
Dave      False
Rob       False
Steve     False
Wes       True
dtype: bool
```

You can apply 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 [171]: data.str.contains('gmail')
Out[171]:
Dave      False
Rob       True
Steve     True
Wes       NaN
dtype: object
```

Regular expressions can be used, too, along with any re options like `IGNORECASE`:

```
In [172]: pattern
Out[172]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [173]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[173]:
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 [174]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [175]: matches
Out[175]:
Dave      True
Rob       True
Steve     True
Wes       NaN
dtype: object
```

To access elements in the embedded lists, we can pass an index to either of these functions:

```
In [176]: matches.str.get(1)
Out[176]:
```

```
Dave    NaN
Rob     NaN
Steve   NaN
Wes     NaN
dtype: float64
```

```
In [177]: matches.str[0]
Out[177]:
Dave     NaN
Rob      NaN
Steve    NaN
Wes      NaN
dtype: float64
```

You can similarly slice strings using this syntax:

```
In [178]: data.str[:5]
Out[178]:
Dave     dave@
Rob      rob@g
Steve    steve
Wes      NaN
dtype: object
```

See [Table 7-5](#) for more pandas string methods.

Table 7-5. Partial listing of 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
extract	Use a regular expression with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group
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>i</i> -th element)
isalnum	Equivalent to built-in <code>str.alnum</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, upper	Convert cases; equivalent to <code>x.lower()</code> or <code>x.upper()</code> for each element

Method	Description
<code>match</code>	Use <code>re.match</code> with the passed regular expression on each element, returning matched groups as list
<code>pad</code>	Add whitespace to left, right, or both sides of strings
<code>center</code>	Equivalent to <code>pad(side='both')</code>
<code>repeat</code>	Duplicate values (e.g., <code>s.str.repeat(3)</code> is equivalent to <code>x * 3</code> for each string)
<code>replace</code>	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

7.4 Conclusion

Effective data preparation can significantly improve productive by enabling you to spend more time analyzing data and less time getting it ready for analysis. We have explored a number of tools in this chapter, but the coverage here is by no means comprehensive. In the next chapter, we will explore pandas's joining and grouping functionality.

Data Wrangling: Join, Combine, and Reshape

In many applications, data may be spread across a number of 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 14](#).

8.1 Hierarchical Indexing

Hierarchical indexing is an important feature of pandas that enables 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', 'c', 'd', 'd'],
...:                    [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, 0, 1, 1, 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 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, you could rearrange the data 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 later in this chapter.

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, 2, 1, 2]],
.....:                        columns=[['Ohio', 'Ohio', 'Colorado'],
.....:                                ['Green', 'Red', 'Green']])
```

```
In [19]: frame
Out[19]:
```

	Ohio		Colorado
	Green	Red	Green
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10

The hierarchical levels can have names (as strings or any Python objects). If so, these will show up in the console output:

```
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



Be careful to distinguish the index names 'state' and 'color' from the row labels.

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 preceding DataFrame with level names could be created like this:

```
MultiIndex.from_arrays([[ 'Ohio', 'Ohio', 'Colorado'], [ 'Green', 'Red', '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, sorts the data 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 by the indicated level:

```
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
```



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 aggregate by on a particular axis. Consider the above `DataFrame`; we can aggregate 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, -1),
.....:                        'c': ['one', 'one', 'one', 'two', 'two',
.....:                              'two', '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	
one	0	0	7
	1	1	6
	2	2	5
two	0	3	4
	1	4	3
	2	5	2
	3	6	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	
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

8.2 Combining and Merging Datasets

Data contained in pandas objects can be combined together in a number of 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.
- The `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 datasets 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 [35]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
.....:                      'data1': range(7)})
```

```
In [36]: df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
.....:                      'data2': range(3)})
```

```
In [37]: df1
Out[37]:
```

	data1	key
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a
5	5	a

```
6      6  b
In [38]: df2
Out[38]:
   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 [39]: pd.merge(df1, df2)
Out[39]:
   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 that information is not specified, `merge` uses the overlapping column names as the keys. It's a good practice to specify explicitly, though:

```
In [40]: pd.merge(df1, df2, on='key')
Out[40]:
   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 [41]: df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                    'data1': range(7)})
```

```
In [42]: df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
   ....:                    'data2': range(3)})
```

```
In [43]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[43]:
   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 [44]: pd.merge(df1, df2, how='outer')
Out[44]:
```

	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`.

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

Many-to-many merges have well-defined, though not necessarily intuitive, behavior. Here's an example:

```
In [45]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
.....:                      'data1': range(6)})
```

```
In [46]: df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
.....:                      'data2': range(5)})
```

```
In [47]: df1
```

```
Out[47]:
```

	data1	key
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a
5	5	b


```
In [48]: df2
```

```
Out[48]:
```

	data2	key
0	0	a
1	1	b
2	2	a
3	3	b
4	4	d

```
In [49]: pd.merge(df1, df2, on='key', how='left')
```

```
Out[49]:
```

	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 three 'b' rows in the left DataFrame and two in the right one, there are six 'b' rows in the result. The join method only affects the distinct key values appearing in the result:

```
In [50]: pd.merge(df1, df2, how='inner')
```

```
Out[50]:
```

	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 [51]: left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],  
.....:                      'key2': ['one', 'two', 'one'],  
.....:                      'lval': [1, 2, 3]})
```

```
In [52]: right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],  
.....:                       'key2': ['one', 'one', 'one', 'two'],  
.....:                       'rval': [4, 5, 6, 7]})
```

```
In [53]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

```
Out[53]:
  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).



When you're 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 earlier 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 [54]: pd.merge(left, right, on='key1')
```

```
Out[54]:
  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 [55]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
```

```
Out[55]:
  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 8-2. 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'; defaults to 'inner'.
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 left and right as the join keys.
left_on	Columns in left DataFrame to use as join keys.
right_on	Analogous to left_on for right DataFrame.
left_index	Use row index in left as its join key (or keys, if a MultiIndex).
right_index	Analogous to left_index.
sort	Sort merged data lexicographically by join keys; True 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') (e.g., if 'data' in both DataFrame objects, would appear as 'data_x' and 'data_y' in result).
copy	If False, avoid copying data into resulting data structure in some exceptional cases; by default always copies.
indicator	Adds a special column _merge that indicates the source of each row; values will be 'left_only', 'right_only', or 'both' based on the origin of the joined data in each row.

Merging on Index

In some cases, the merge key(s) 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 [56]: left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
.....:                        'value': range(6)})
```

```
In [57]: right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])
```

```
In [58]: left1
```

```
Out[58]:
   key  value
0    a      0
1    b      1
2    a      2
3    a      3
4    b      4
5    c      5
```

```
In [59]: right1
```

```
Out[59]:
   group_val
a          3.5
b          7.0
```

```
In [60]: pd.merge(left1, right1, left_on='key', right_index=True)
```

```
Out[60]:
```

	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 [61]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
```

```
Out[61]:
```

	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 [62]: lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio',  
.....:                               'Nevada', 'Nevada'],  
.....:                       'key2': [2000, 2001, 2002, 2001, 2002],  
.....:                       'data': np.arange(5.)})
```

```
In [63]: righth = pd.DataFrame(np.arange(12).reshape((6, 2)),  
.....:                        index=[['Nevada', 'Nevada', 'Ohio', 'Ohio',  
.....:                               'Ohio', 'Ohio'],  
.....:                        [2001, 2000, 2000, 2000, 2001, 2002]],  
.....:                        columns=['event1', 'event2'])
```

```
In [64]: lefth
```

```
Out[64]:
```

	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 [65]: righth
```

```
Out[65]:
```

		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 [66]: pd.merge(left, right, left_on=['key1', 'key2'], right_index=True)
```

```
Out[66]:
```

	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 [67]: pd.merge(left, right, left_on=['key1', 'key2'],
.....:             right_index=True, how='outer')
```

```
Out[67]:
```

	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 [68]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
.....:                        index=['a', 'c', 'e'],
.....:                        columns=['Ohio', 'Nevada'])
```

```
In [69]: right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
.....:                          index=['b', 'c', 'd', 'e'],
.....:                          columns=['Missouri', 'Alabama'])
```

```
In [70]: left2
```

```
Out[70]:
```

	Ohio	Nevada
a	1.0	2.0
c	3.0	4.0
e	5.0	6.0

```
In [71]: right2
```

```
Out[71]:
```

	Missouri	Alabama
b	7.0	8.0
c	9.0	10.0
d	11.0	12.0
e	13.0	14.0

```
In [72]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
```

```
Out[72]:
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 [73]: left2.join(right2, how='outer')
Out[73]:
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 (i.e., 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 [74]: left1.join(right1, on='key')
Out[74]:
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 in the next section:

```
In [75]: another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
....:                          index=['a', 'c', 'e', 'f'],
....:                          columns=['New York', 'Oregon'])
```

```
In [76]: another
Out[76]:
New York Oregon
a 7.0 8.0
c 9.0 10.0
e 11.0 12.0
f 16.0 17.0
```

```
In [77]: left2.join([right2, another])
Out[77]:
```

	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 [78]: left2.join([right2, another], how='outer')
Out[78]:
```

	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 [79]: arr = np.arange(12).reshape((3, 4))
```

```
In [80]: arr
Out[80]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
In [81]: np.concatenate([arr, arr], axis=1)
Out[81]:
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 [82]: s1 = pd.Series([0, 1], index=['a', 'b'])
```

```
In [83]: s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])
```

```
In [84]: s3 = pd.Series([5, 6], index=['f', 'g'])
```

Calling `concat` with these objects in a list glues together the values and indexes:

```
In [85]: pd.concat([s1, s2, s3])
```

```
Out[85]:
```

```
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 [86]: pd.concat([s1, s2, s3], axis=1)
```

```
Out[86]:
```

```
   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 [87]: s4 = pd.concat([s1, s3])
```

```
In [88]: s4
```

```
Out[88]:
```

```
a    0
b    1
f    5
g    6
dtype: int64
```

```
In [89]: pd.concat([s1, s4], axis=1)
```

```
Out[89]:
```



```
0 1
a 0.0 0
b 1.0 1
f NaN 5
g NaN 6
```

```
In [90]: pd.concat([s1, s4], axis=1, join='inner')
```

```
Out[90]:
```

```
0 1
a 0 0
b 1 1
```

In this last example, the 'f' and 'g' labels disappeared because of the `join='inner'` option.

You can even specify the axes to be used on the other axes with `join_axes`:

```
In [91]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
```

```
Out[91]:
```

```
0 1
a 0.0 0.0
c NaN NaN
b 1.0 1.0
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 [92]: result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])
```

```
In [93]: result
```

```
Out[93]:
```

```
one  a  0
     b  1
two  a  0
     b  1
three f  5
     g  6
dtype: int64
```

```
In [94]: result.unstack()
```

```
Out[94]:
```

```
      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 [95]: pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
```

```
Out[95]:
```

	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 [96]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],
...:                        columns=['one', 'two'])
```

```
In [97]: df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
...:                        columns=['three', 'four'])
```

```
In [98]: df1
```

```
Out[98]:
```

	one	two
a	0	1
b	2	3
c	4	5

```
In [99]: df2
```

```
Out[99]:
```

	three	four
a	5	6
c	7	8

```
In [100]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
```

```
Out[100]:
```

	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 [101]: pd.concat({'level1': df1, 'level2': df2}, axis=1)
```

```
Out[101]:
```

	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 [102]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
.....:               names=['upper', 'lower'])
```

```
Out[102]:
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 [103]: df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])
```

```
In [104]: df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])
```

```
In [105]: df1
```

```
Out[105]:
           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 [106]: df2
```

```
Out[106]:
           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 [107]: pd.concat([df1, df2], ignore_index=True)
```

```
Out[107]:
           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; this is the only required argument
<code>axis</code>	Axis to concatenate along; defaults to 0 (along rows)
<code>join</code>	Either 'inner' or 'outer' ('outer' by default); 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>)

Argument	Description
levels	Specific indexes to use as hierarchical index level or levels if keys passed
names	Names for created hierarchical levels if keys and/or levels passed
verify_integrity	Check new axis in concatenated object for duplicates and raise exception if so; by default (False) allows duplicates
ignore_index	Do not preserve indexes along concatenation axis, instead producing a new range(total_length) index

Combining Data with Overlap

There is another data combination situation that 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 [108]: a = pd.Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],
.....:                  index=['f', 'e', 'd', 'c', 'b', 'a'])
```

```
In [109]: b = pd.Series(np.arange(len(a), dtype=np.float64),
.....:                  index=['f', 'e', 'd', 'c', 'b', 'a'])
```

```
In [110]: b[-1] = np.nan
```

```
In [111]: a
Out[111]:
f    NaN
e    2.5
d    NaN
c    3.5
b    4.5
a    NaN
dtype: float64
```

```
In [112]: b
Out[112]:
f    0.0
e    1.0
d    2.0
c    3.0
b    4.0
a    NaN
dtype: float64
```

```
In [113]: np.where(pd.isnull(a), b, a)
Out[113]: 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 [114]: b[:-2].combine_first(a[2:])
Out[114]:
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 [115]: df1 = pd.DataFrame({'a': [1., np.nan, 5., np.nan],
.....:                      'b': [np.nan, 2., np.nan, 6.],
.....:                      'c': range(2, 18, 4)})
```

```
In [116]: df2 = pd.DataFrame({'a': [5., 4., np.nan, 3., 7.],
.....:                      'b': [np.nan, 3., 4., 6., 8.]})
```

```
In [117]: df1
Out[117]:
   a    b    c
0  1.0 NaN    2
1  NaN  2.0    6
2  5.0 NaN  10
3  NaN  6.0  14
```

```
In [118]: df2
Out[118]:
   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 [119]: df1.combine_first(df2)
Out[119]:
   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
```

8.3 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 [120]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),
.....:                        index=pd.Index(['Ohio', 'Colorado'], name='state'),
.....:                        columns=pd.Index(['one', 'two', 'three'],
.....:                                       name='number'))
```

```
In [121]: data
Out[121]:
```

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 [122]: result = data.stack()
```

```
In [123]: result
Out[123]:
```

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 [124]: result.unstack()
Out[124]:
```

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 [125]: result.unstack(0)
Out[125]:
state  Ohio  Colorado
number
one      0      3
two      1      4
three    2      5
```

```
In [126]: result.unstack('state')
Out[126]:
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 [127]: s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
```

```
In [128]: s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])
```

```
In [129]: data2 = pd.concat([s1, s2], keys=['one', 'two'])
```

```
In [130]: data2
Out[130]:
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: int64
```

```
In [131]: data2.unstack()
Out[131]:
     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 [132]: data2.unstack()
Out[132]:
     a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

```
In [133]: data2.unstack().stack()
Out[133]:
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 [134]: data2.unstack().stack(dropna=False)
Out[134]:
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 you unstack in a DataFrame, the level unstacked becomes the lowest level in the result:

```
In [135]: df = pd.DataFrame({'left': result, 'right': result + 5},
.....:                      columns=pd.Index(['left', 'right'], name='side'))
```

```
In [136]: df
Out[136]:
```

side		left	right
Ohio	number		
	one	0	5
	two	1	6
Colorado	three	2	7
	one	3	8
	two	4	9
	three	5	10

```
In [137]: df.unstack('state')
Out[137]:
```

side	left	right		
state	Ohio	Colorado	Ohio	Colorado
number				
one	0	3	5	8
two	1	4	6	9
three	2	5	7	10

When calling `stack`, we can indicate the name of the axis to stack:


```
In [138]: df.unstack('state').stack('side')
Out[138]:
```

state		Colorado	Ohio
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. Let’s load some example data and do a small amount of time series wrangling and other data cleaning:

```
In [139]: data = pd.read_csv('examples/macrodata.csv')
```

```
In [140]: data.head()
```

```
Out[140]:
```

	year	quarter	realgdp	realcons	realinv	realgovt	realdpi	cpi	\
0	1959.0	1.0	2710.349	1707.4	286.898	470.045	1886.9	28.98	
1	1959.0	2.0	2778.801	1733.7	310.859	481.301	1919.7	29.15	
2	1959.0	3.0	2775.488	1751.8	289.226	491.260	1916.4	29.35	
3	1959.0	4.0	2785.204	1753.7	299.356	484.052	1931.3	29.37	
4	1960.0	1.0	2847.699	1770.5	331.722	462.199	1955.5	29.54	
	m1	tbilrate	unemp	pop	infl	realint			
0	139.7	2.82	5.8	177.146	0.00	0.00			
1	141.7	3.08	5.1	177.830	2.34	0.74			
2	140.5	3.82	5.3	178.657	2.74	1.09			
3	140.0	4.33	5.6	179.386	0.27	4.06			
4	139.6	3.50	5.2	180.007	2.31	1.19			

```
In [141]: periods = pd.PeriodIndex(year=data.year, quarter=data.quarter,
.....:                               name='date')
```

```
In [142]: columns = pd.Index(['realgdp', 'infl', 'unemp'], name='item')
```

```
In [143]: data = data.reindex(columns=columns)
```

```
In [144]: data.index = periods.to_timestamp('D', 'end')
```

```
In [145]: ldata = data.stack().reset_index().rename(columns={0: 'value'})
```

We will look at `PeriodIndex` a bit more closely in [Chapter 11](#). In short, it combines the year and quarter columns to create a kind of time interval type.

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 previous 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]:
```

item	value			value2		
	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('item')
```

```
In [155]: unstacked[:7]
```

```
Out[155]:
```

item	value			value2		
	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

Pivoting “Wide” to “Long” Format

An inverse operation to pivot for DataFrames is `pandas.melt`. Rather than transforming one column into many in a new DataFrame, it merges multiple columns into one, producing a DataFrame that is longer than the input. Let’s look at an example:

```
In [157]: df = pd.DataFrame({'key': ['foo', 'bar', 'baz'],
.....:                       'A': [1, 2, 3],
.....:                       'B': [4, 5, 6],
.....:                       'C': [7, 8, 9]})
```

```
In [158]: df
```

```
Out[158]:
```

	A	B	C	key
0	1	4	7	foo
1	2	5	8	bar
2	3	6	9	baz

The 'key' column may be a group indicator, and the other columns are data values. When using `pandas.melt`, we must indicate which columns (if any) are group indicators. Let’s use 'key' as the only group indicator here:

```
In [159]: melted = pd.melt(df, ['key'])
```

```
In [160]: melted
```

```
Out[160]:
```

	key	variable	value
0	foo	A	1
1	bar	A	2
2	baz	A	3
3	foo	B	4
4	bar	B	5
5	baz	B	6
6	foo	C	7
7	bar	C	8
8	baz	C	9

Using pivot, we can reshape back to the original layout:

```
In [161]: reshaped = melted.pivot('key', 'variable', 'value')

In [162]: reshaped
Out[162]:
variable A B C
key
bar      2 5 8
baz      3 6 9
foo      1 4 7
```

Since the result of pivot creates an index from the column used as the row labels, we may want to use reset_index to move the data back into a column:

```
In [163]: reshaped.reset_index()
Out[163]:
variable key A B C
0        bar 2 5 8
1        baz 3 6 9
2        foo 1 4 7
```

You can also specify a subset of columns to use as value columns:

```
In [164]: pd.melt(df, id_vars=['key'], value_vars=['A', 'B'])
Out[164]:
   key variable  value
0  foo         A      1
1  bar         A      2
2  baz         A      3
3  foo         B      4
4  bar         B      5
5  baz         B      6
```

pandas.melt can be used without any group identifiers, too:

```
In [165]: pd.melt(df, value_vars=['A', 'B', 'C'])
Out[165]:
   variable  value
0         A      1
1         A      2
2         A      3
3         B      4
4         B      5
5         B      6
6         C      7
7         C      8
8         C      9

In [166]: pd.melt(df, value_vars=['key', 'A', 'B'])
Out[166]:
   variable value
0        key   foo
1        key   bar
```

	key	baz
2		
3	A	1
4	A	2
5	A	3
6	B	4
7	B	5
8	B	6

8.4 Conclusion

Now that you have some pandas basics for data import, cleaning, and reorganization under your belt, we are ready to move on to data visualization with matplotlib. We will return to pandas later in the book when we discuss more advanced analytics.

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, to help identify outliers or needed data transformations, or as a way of generating 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** and libraries that build on top of it.

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 visualizations to all of the common vector and raster graphics formats (PDF, SVG, JPG, PNG, BMP, GIF, etc.). With the exception of a few diagrams, nearly all of the graphics in this book were produced using matplotlib.

Over time, matplotlib has spawned a number of add-on toolkits for data visualization that use matplotlib for their underlying plotting. One of these is **seaborn**, which we explore later in this chapter.

The simplest way to follow the code examples in the chapter is to use interactive plotting in the Jupyter notebook. To set this up, execute the following statement in a Jupyter notebook:

```
%matplotlib notebook
```

9.1 A Brief matplotlib API Primer

With matplotlib, we use the following import convention:

```
In [11]: import matplotlib.pyplot as plt
```

After running `%matplotlib notebook` in Jupyter (or simply `%matplotlib` in IPython), we can try creating a simple plot. If everything is set up right, a line plot like **Figure 9-1** should appear:

```
In [12]: import numpy as np
```

```
In [13]: data = np.arange(10)
```

```
In [14]: data
```

```
Out[14]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [15]: plt.plot(data)
```

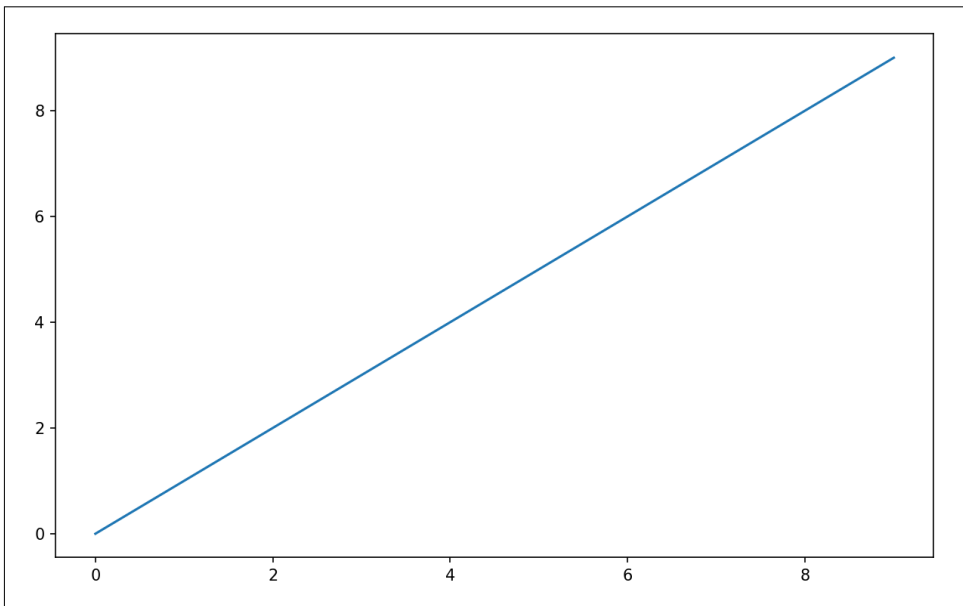


Figure 9-1. Simple line plot

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.



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 [16]: 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 [17]: ax1 = fig.add_subplot(2, 2, 1)
```

This means that the figure should be 2×2 (so up to four plots in total), and we're selecting the first of four subplots (numbered from 1). If you create the next two subplots, you'll end up with a visualization that looks like [Figure 9-2](#):

```
In [18]: ax2 = fig.add_subplot(2, 2, 2)
```

```
In [19]: ax3 = fig.add_subplot(2, 2, 3)
```

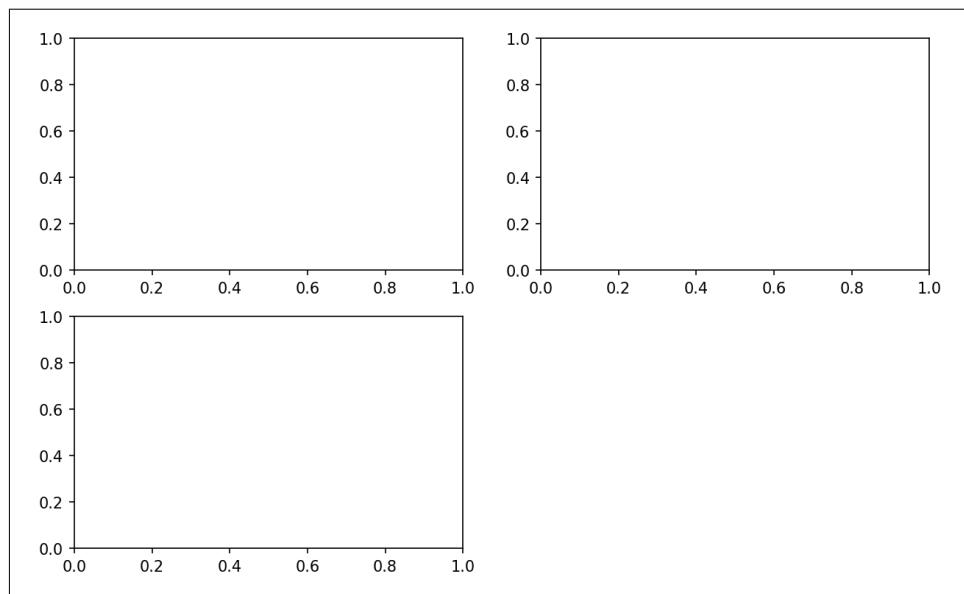


Figure 9-2. An empty matplotlib figure with three subplots



One nuance of using Jupyter notebooks is that plots are reset after each cell is evaluated, so for more complex plots you must put all of the plotting commands in a single notebook cell.

Here we 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. So if we add the following command, you'll get something like **Figure 9-3**:

```
In [20]: plt.plot(np.random.randn(50).cumsum(), 'k--')
```

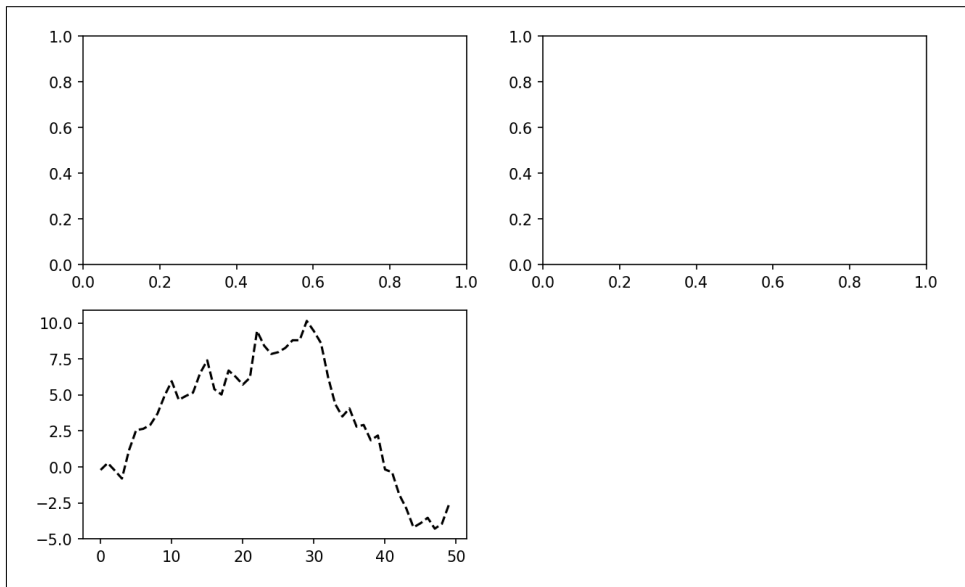


Figure 9-3. Data visualization after single plot

The `'k--'` is a *style* option instructing matplotlib to plot a black dashed line. The objects returned by `fig.add_subplot` here are `AxesSubplot` objects, on which you can directly plot on the other empty subplots by calling each one's instance method (see **Figure 9-4**):

```
In [21]: _ = ax1.hist(np.random.randn(100), bins=20, color='k', alpha=0.3)
```

```
In [22]: ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.randn(30))
```

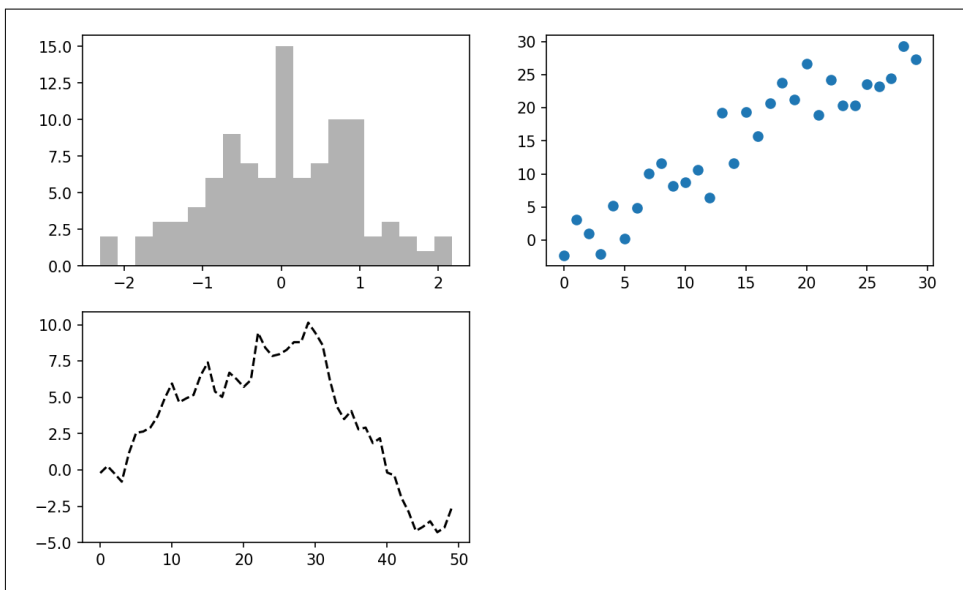


Figure 9-4. Data visualization after additional plots

You can find a comprehensive catalog of plot types in the [matplotlib documentation](#).

Creating a figure with a grid of subplots is a very common task, so matplotlib includes a convenience method, `plt.subplots`, that creates a new figure and returns a NumPy array containing the created subplot objects:

```
In [24]: fig, axes = plt.subplots(2, 3)
```

```
In [25]: axes
```

```
Out[25]:
```

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7fb626374048>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7fb62625db00>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7fb6262f6c88>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x7fb6261a36a0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7fb626181860>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7fb6260fd4e0>]], dtype=object)
```

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 you're comparing data on the same scale; otherwise, matplotlib autoscales plot limits independently. See [Table 9-1](#) for more on this method.

Table 9-1. *pyplot.subplots options*

Argument	Description
<code>nrows</code>	Number of rows of subplots
<code>ncols</code>	Number of columns of subplots
<code>sharex</code>	All subplots should use the same x-axis ticks (adjusting the <code>xlim</code> will affect all subplots)
<code>sharey</code>	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 subplots 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. You can change the spacing using the `subplots_adjust` method on Figure objects, 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 9-5](#)):

```
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='k', alpha=0.5)
plt.subplots_adjust(wspace=0, hspace=0)
```

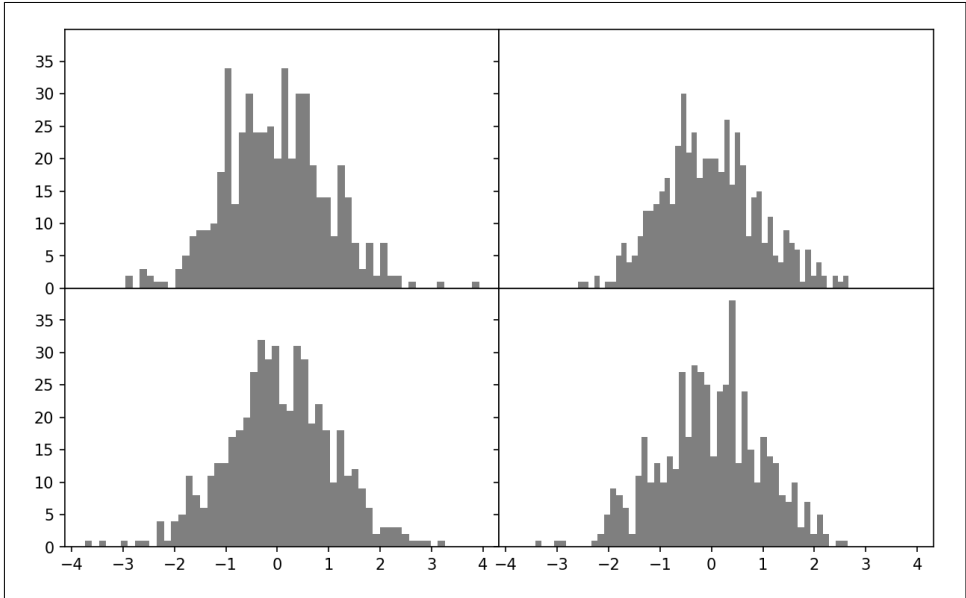


Figure 9-5. Data visualization with no inter-subplot spacing

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 (we'll look at how to do this in the following 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 line style 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 you can use any color on the spectrum by specifying its hex code (e.g., '#CECECE'). You can see the full set of line styles by looking at the docstring for plot (use `plot?` in IPython or Jupyter).

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 9-6](#)):

```
In [30]: from numpy.random import randn
In [31]: plt.plot(randn(30).cumsum(), 'ko--')
```

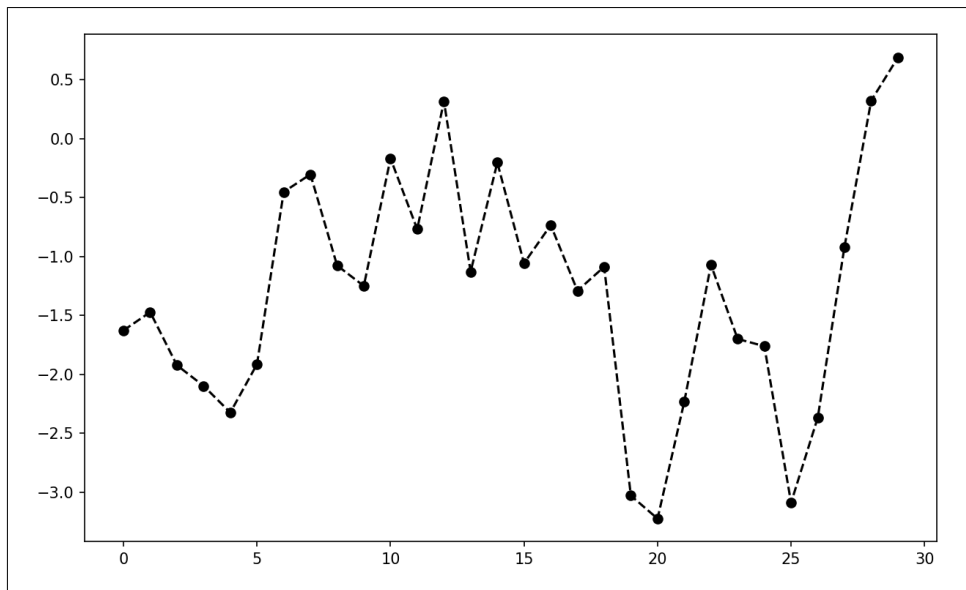


Figure 9-6. Line plot with markers

This could also have been written more explicitly as:

```
plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')
```

For line plots, you will notice that subsequent points are linearly interpolated by default. This can be altered with the `drawstyle` option ([Figure 9-7](#)):

```
In [33]: data = np.random.randn(30).cumsum()
In [34]: plt.plot(data, 'k--', label='Default')
Out[34]: [<matplotlib.lines.Line2D at 0x7fb624d86160>]
In [35]: plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')
Out[35]: [<matplotlib.lines.Line2D at 0x7fb624d869e8>]
In [36]: plt.legend(loc='best')
```

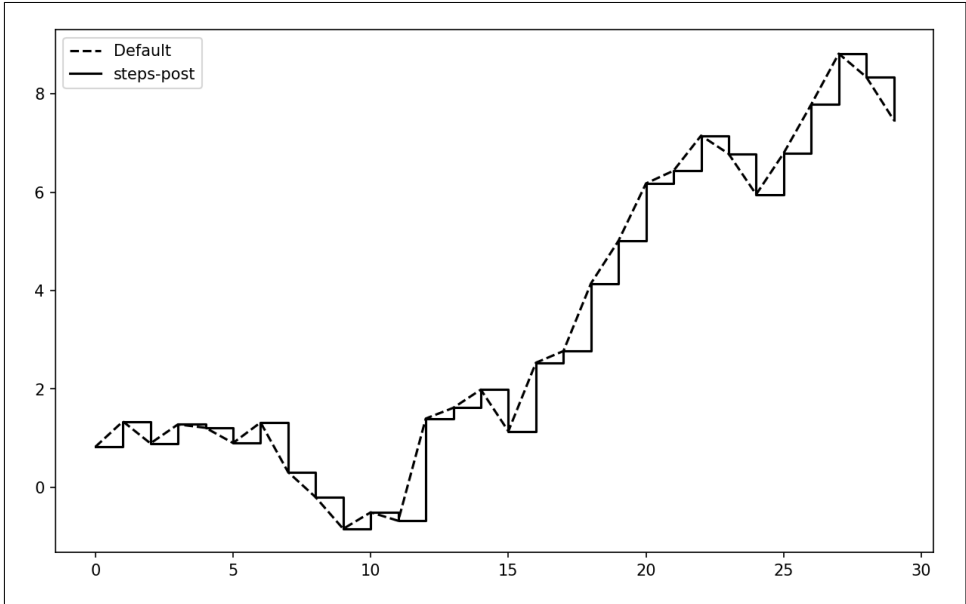


Figure 9-7. Line plot with different drawstyle options

You may notice output like `<matplotlib.lines.Line2D at ...>` when you run this. `matplotlib` returns objects that reference the plot subcomponent that was just added. A lot of the time you can safely ignore this output. Here, since we passed the `label` arguments to `plot`, we are able to create a plot legend to identify each line using `plt.legend`.



You must call `plt.legend` (or `ax.legend`, if you have a reference to the axes) to create the legend, whether or not you passed the `label` options when plotting the data.

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 (e.g., `plt.xlim()` returns the current x-axis plotting range)
- Called with parameters sets the parameter value (e.g., `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 9-8](#)):

```
In [37]: fig = plt.figure()
```

```
In [38]: ax = fig.add_subplot(1, 1, 1)
```

```
In [39]: ax.plot(np.random.randn(1000).cumsum())
```

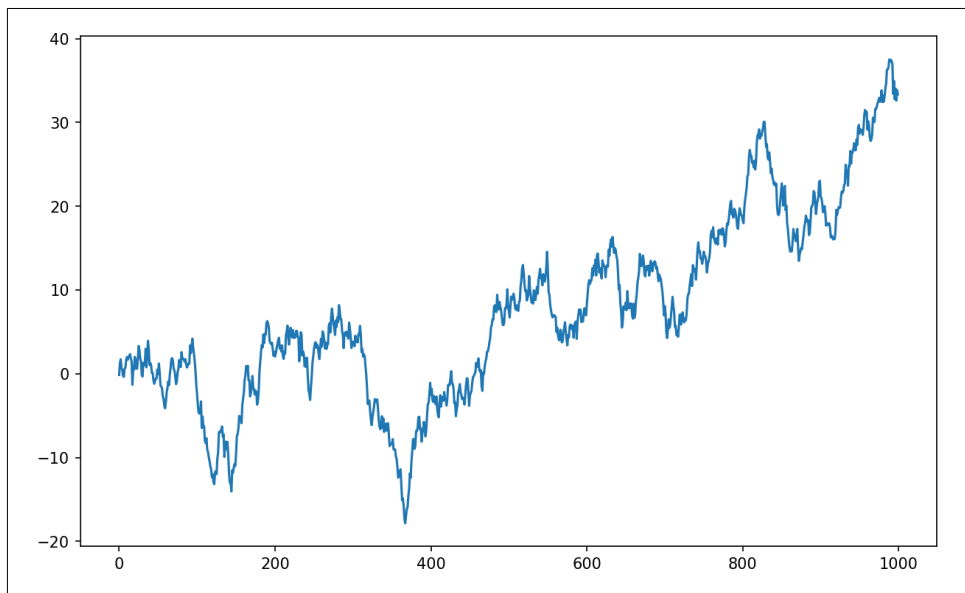


Figure 9-8. Simple plot for illustrating `xticks` (with label)

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 [40]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])  
  
In [41]: labels = ax.set_xticklabels(['one', 'two', 'three', 'four', 'five'],  
    ...:                             rotation=30, fontsize='small')
```

The `rotation` option sets the x tick labels at a 30-degree rotation. Lastly, `set_xlabel` gives a name to the x-axis and `set_title` the subplot title (see [Figure 9-9](#) for the resulting figure):

```
In [42]: ax.set_title('My first matplotlib plot')  
Out[42]: <matplotlib.text.Text at 0x7fb624d055f8>  
  
In [43]: ax.set_xlabel('Stages')
```

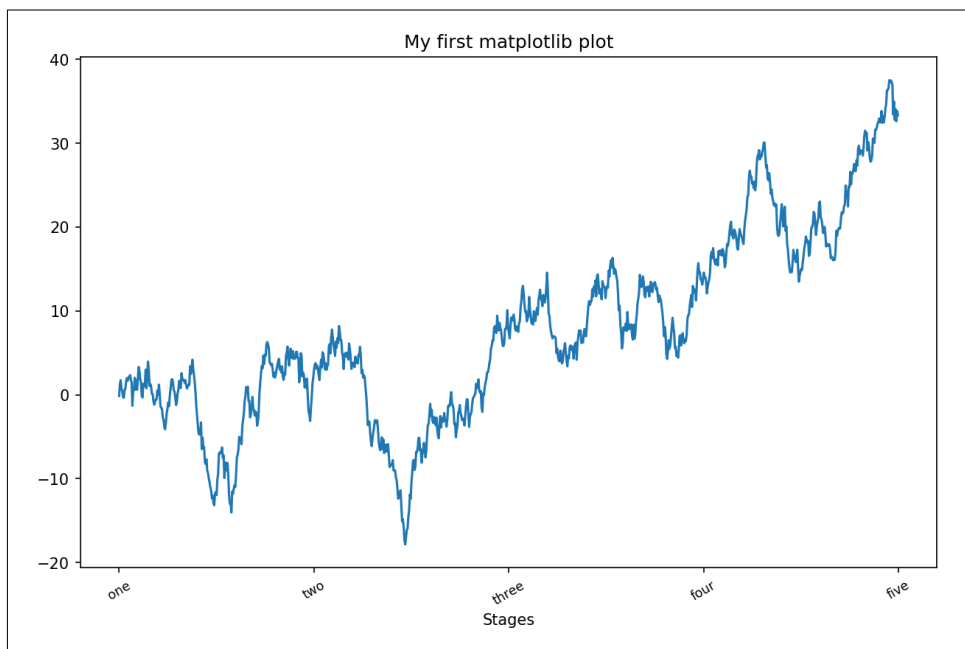


Figure 9-9. Simple plot for illustrating `xticks`

Modifying the y-axis consists of the same process, substituting `y` for `x` in the above. The axes class has a `set` method that allows batch setting of plot properties. From the prior example, we could also have written:

```
props = {  
    'title': 'My first matplotlib plot',  
    'xlabel': 'Stages'  
}  
ax.set(**props)
```

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 [44]: from numpy.random import randn

In [45]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)

In [46]: ax.plot(randn(1000).cumsum(), 'k', label='one')
Out[46]: [<matplotlib.lines.Line2D at 0x7fb624bdf860>]

In [47]: ax.plot(randn(1000).cumsum(), 'k--', label='two')
Out[47]: [<matplotlib.lines.Line2D at 0x7fb624be90f0>]

In [48]: ax.plot(randn(1000).cumsum(), 'k.', label='three')
Out[48]: [<matplotlib.lines.Line2D at 0x7fb624be9160>]
```

Once you've done this, you can either call `ax.legend()` or `plt.legend()` to automatically create a legend. The resulting plot is in [Figure 9-10](#):

```
In [49]: ax.legend(loc='best')
```

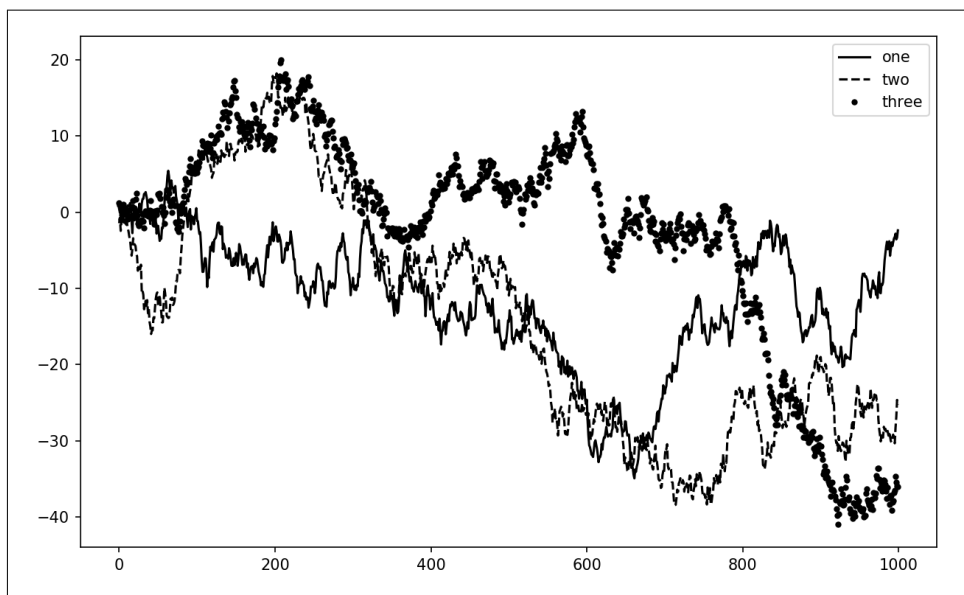


Figure 9-10. Simple plot with three lines and legend

The legend method has several other choices for the location `loc` argument. See the docstring (with `ax.legend?`) for more information.

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. You can add annotations and text 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. You can most easily reproduce this code example in a single cell in a Jupyter notebook. See [Figure 9-11](#) 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) + 75),
                xytext=(date, spx.asof(date) + 225),
                arrowprops=dict(facecolor='black', headwidth=4, width=2,
                                headlength=4),
                horizontalalignment='left', verticalalignment='top')

# 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 the 2008-2009 financial crisis')
```

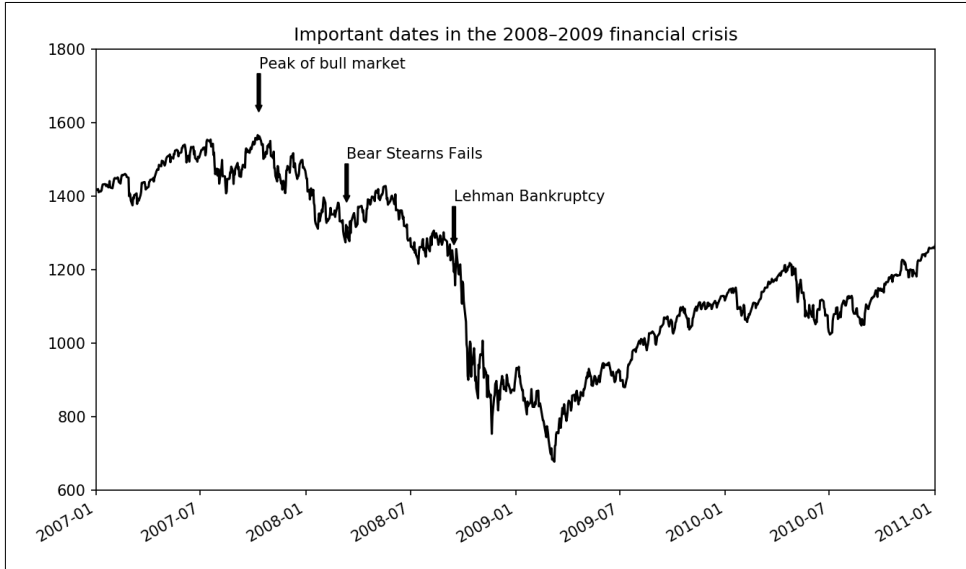


Figure 9-11. Important dates in the 2008–2009 financial crisis

There are a couple of important points to highlight in this plot: the `ax.annotate` method can draw labels at the indicated `x` and `y` coordinates. We use the `set_xlim` and `set_ylim` methods to manually set the start and end boundaries for the plot rather than using matplotlib's default. Lastly, `ax.set_title` adds a main title to the plot.

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 9-12):

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)
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)
```

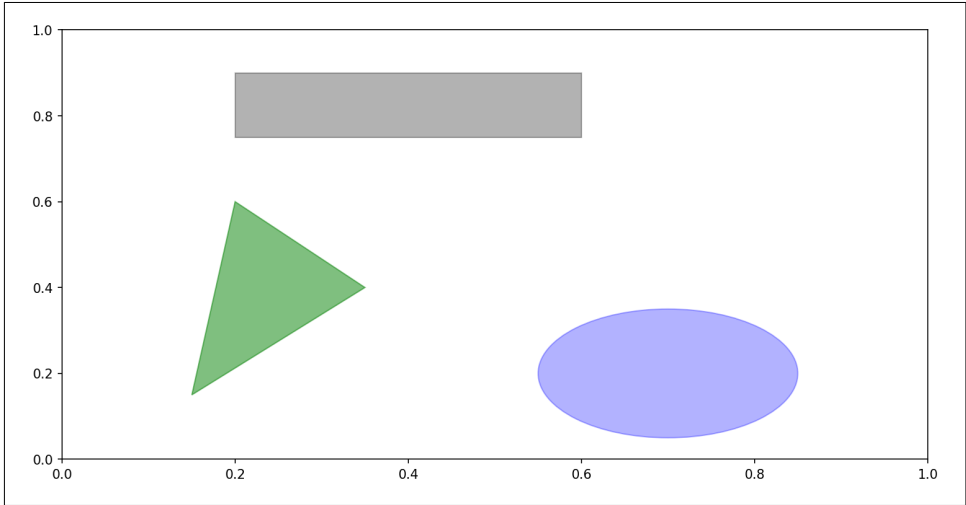


Figure 9-12. Data visualization composed from three different patches

If you look at the implementation of many familiar plot types, you will see that they are assembled from patches.

Saving Plots to File

You can save the active figure 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 `.pdf` 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 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()
```

See [Table 9-2](#) for a list of some other options for `savefig`.

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>.pdf</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> , <code>edgecolor</code>	The color of the figure background outside of the subplots; 'w' (white), by default
<code>format</code>	The explicit file format to use ('png', 'pdf', 'svg', 'ps', 'eps', ...)
<code>bbox_inches</code>	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 modify the configuration programmatically from Python is to use the `rc` method; for example, to set the global default figure size to be 10×10 , 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.

9.2 Plotting with pandas and seaborn

matplotlib can be a fairly low-level tool. You assemble a plot from its base components: the data display (i.e., 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 that simplify creating visualizations from DataFrame and Series objects. Another library is **seaborn**, a statistical graphics library created by Michael Waskom. Seaborn simplifies creating many common visualization types.



Importing seaborn modifies the default matplotlib color schemes and plot styles to improve readability and aesthetics. Even if you do not use the seaborn API, you may prefer to import seaborn as a simple way to improve the visual aesthetics of general matplotlib plots.

Line Plots

Series and DataFrame each have a `plot` attribute for making some basic plot types. By default, `plot()` makes line plots (see [Figure 9-13](#)):

```
In [60]: s = pd.Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))
```

```
In [61]: s.plot()
```

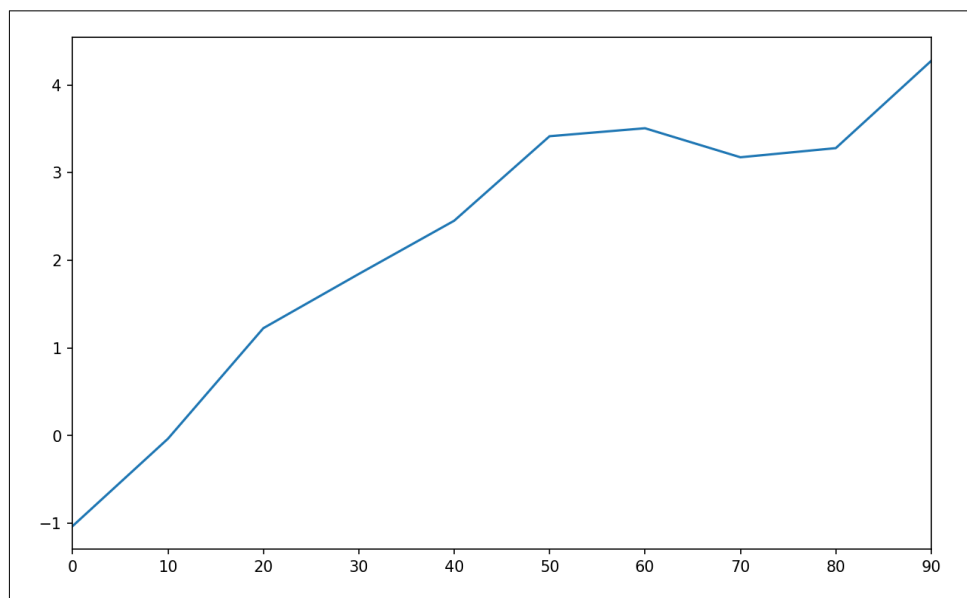


Figure 9-13. Simple Series plot

The Series object's index is passed to matplotlib for plotting on the x-axis, though you can disable this by passing `use_index=False`. The x-axis ticks and limits can be adjusted with the `xticks` and `xlim` options, and y-axis respectively with `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 9-14](#)):

```
In [62]: df = pd.DataFrame(np.random.randn(10, 4).cumsum(0),
.....:                    columns=['A', 'B', 'C', 'D'],
.....:                    index=np.arange(0, 100, 10))
```

```
In [63]: df.plot()
```

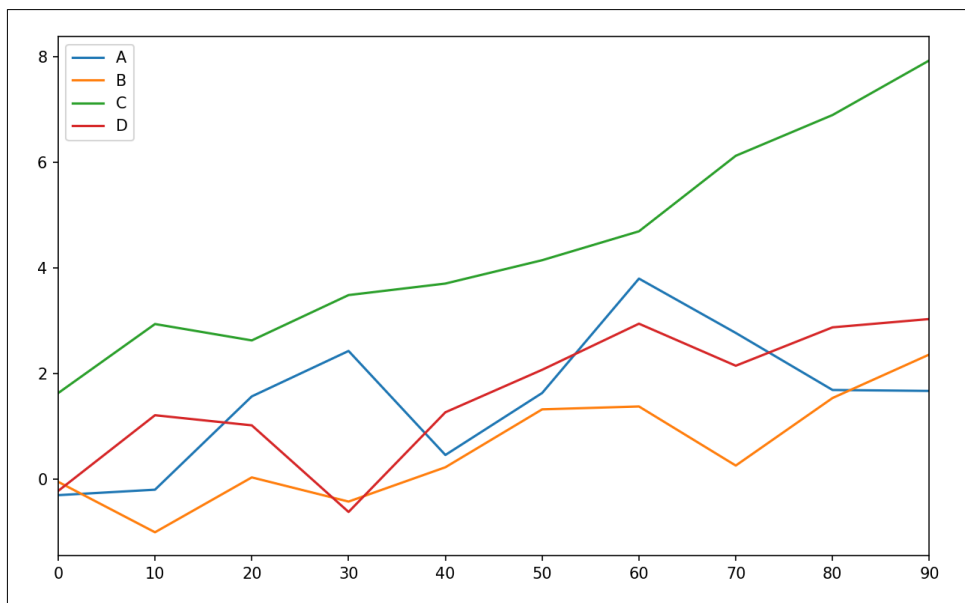


Figure 9-14. Simple DataFrame plot

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.



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
label	Label for plot legend
ax	matplotlib subplot object to plot on; if nothing passed, uses active matplotlib subplot
style	Style string, like 'ko--', to be passed to matplotlib
alpha	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 subplots=True, share the same x-axis, linking ticks and limits
sharey	If subplots=True, share the same y-axis
figsize	Size of figure to create as tuple
title	Plot title as string
legend	Add a subplot legend (True by default)
sort_columns	Plot columns in alphabetical order; by default uses existing column order



For time series plotting, see [Chapter 11](#).

Bar Plots

The `plot.bar()` and `plot.barh()` make vertical and horizontal bar plots, respectively. In this case, the Series or DataFrame index will be used as the x (bar) or y (barh) ticks (see [Figure 9-15](#)):

```
In [64]: fig, axes = plt.subplots(2, 1)
```

```
In [65]: data = pd.Series(np.random.rand(16), index=list('abcdefghijklmnop'))
```

```
In [66]: data.plot.bar(ax=axes[0], color='k', alpha=0.7)
```

```
Out[66]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb62493d470>
```

```
In [67]: data.plot.barh(ax=axes[1], color='k', alpha=0.7)
```

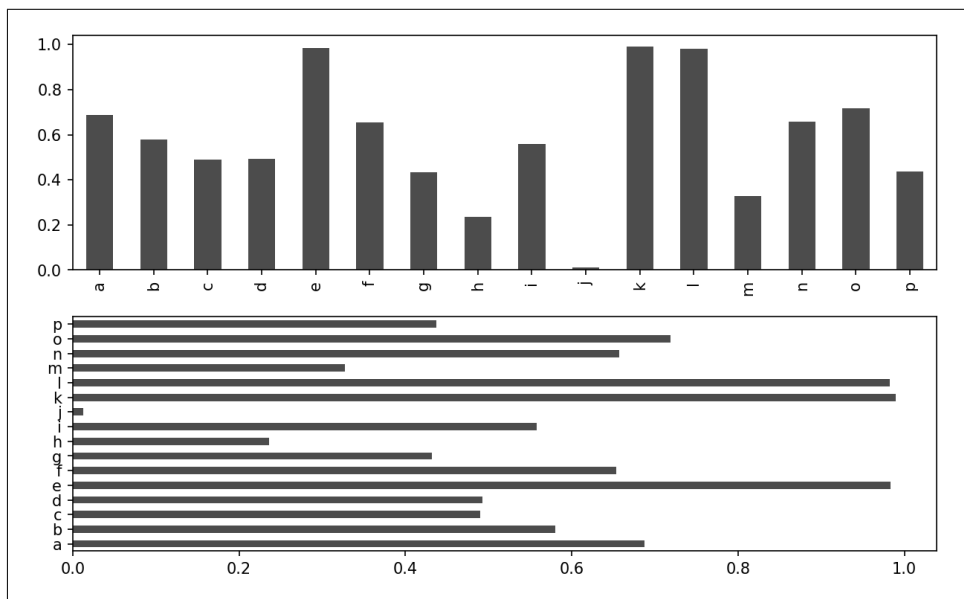


Figure 9-15. Horizontal and vertical bar plot

The options `color='k'` and `alpha=0.7` set the color of the plots to black and use partial transparency on the filling.

With a DataFrame, bar plots group the values in each row together in a group of bars, side by side, for each value. See [Figure 9-16](#):

```
In [69]: df = pd.DataFrame(np.random.rand(6, 4),
....:                      index=['one', 'two', 'three', 'four', 'five', 'six'],
....:                      columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))
```

```
In [70]: df
```

```
Out[70]:
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 [71]: df.plot.bar()
```

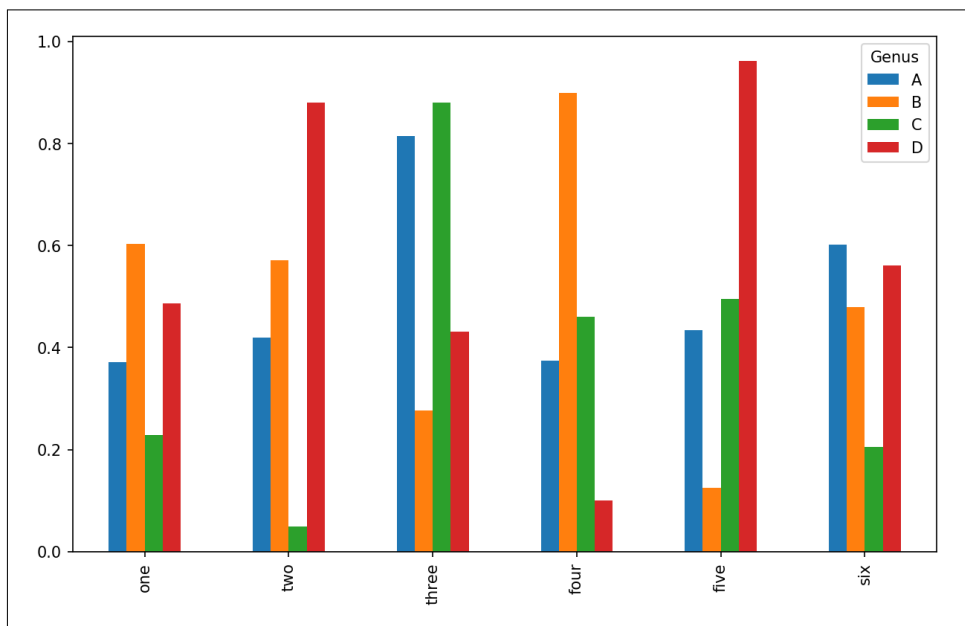


Figure 9-16. DataFrame bar plot

Note that the name “Genus” on the DataFrame’s columns is used to title the legend.

We create stacked bar plots from a DataFrame by passing `stacked=True`, resulting in the value in each row being stacked together (see [Figure 9-17](#)):

```
In [73]: df.plot.barh(stacked=True, alpha=0.5)
```

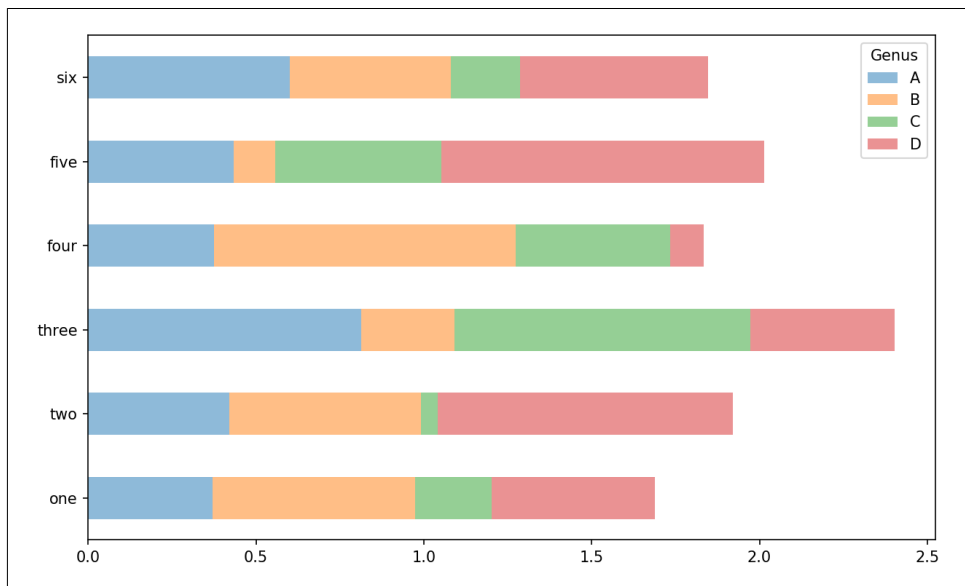


Figure 9-17. DataFrame stacked bar plot



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 dataset 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 [75]: tips = pd.read_csv('examples/tips.csv')
```

```
In [76]: party_counts = pd.crosstab(tips['day'], tips['size'])
```

```
In [77]: party_counts
```

```
Out[77]:
```

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 [78]: party_counts = party_counts.loc[:, 2:5]
```

Then, normalize so that each row sums to 1 and make the plot (see [Figure 9-18](#)):

```
# Normalize to sum to 1
```

```
In [79]: party_pcts = party_counts.div(party_counts.sum(1), axis=0)
```

```
In [80]: party_pcts
```

```
Out[80]:
```

```
size      2      3      4      5
day
Fri  0.888889  0.055556  0.055556  0.000000
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 [81]: party_pcts.plot.bar()
```

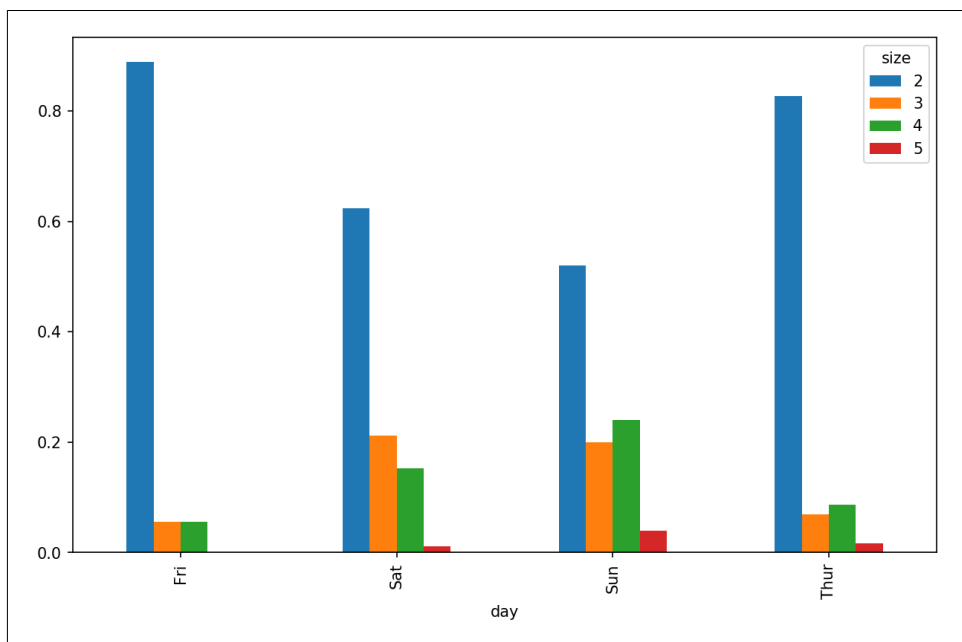


Figure 9-18. Fraction of parties by size on each day

So you can see that party sizes appear to increase on the weekend in this dataset.

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` (see [Figure 9-19](#) for the resulting plot):

```
In [83]: import seaborn as sns
```

```
In [84]: tips['tip_pct'] = tips['tip'] / (tips['total_bill'] - tips['tip'])
```

```
In [85]: tips.head()
```

```
Out[85]:
```

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.063204
1	10.34	1.66	No	Sun	Dinner	3	0.191244
2	21.01	3.50	No	Sun	Dinner	3	0.199886
3	23.68	3.31	No	Sun	Dinner	2	0.162494
4	24.59	3.61	No	Sun	Dinner	4	0.172069

```
In [86]: sns.barplot(x='tip_pct', y='day', data=tips, orient='h')
```

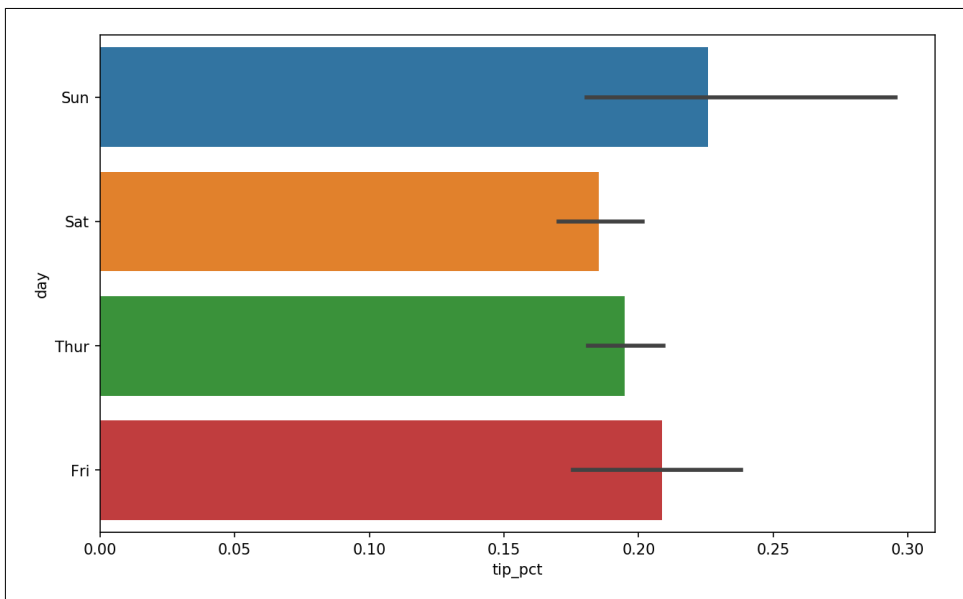


Figure 9-19. Tipping percentage by day with error bars

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 that enables us to split by an additional categorical value (Figure 9-20):

```
In [88]: sns.barplot(x='tip_pct', y='day', hue='time', data=tips, orient='h')
```

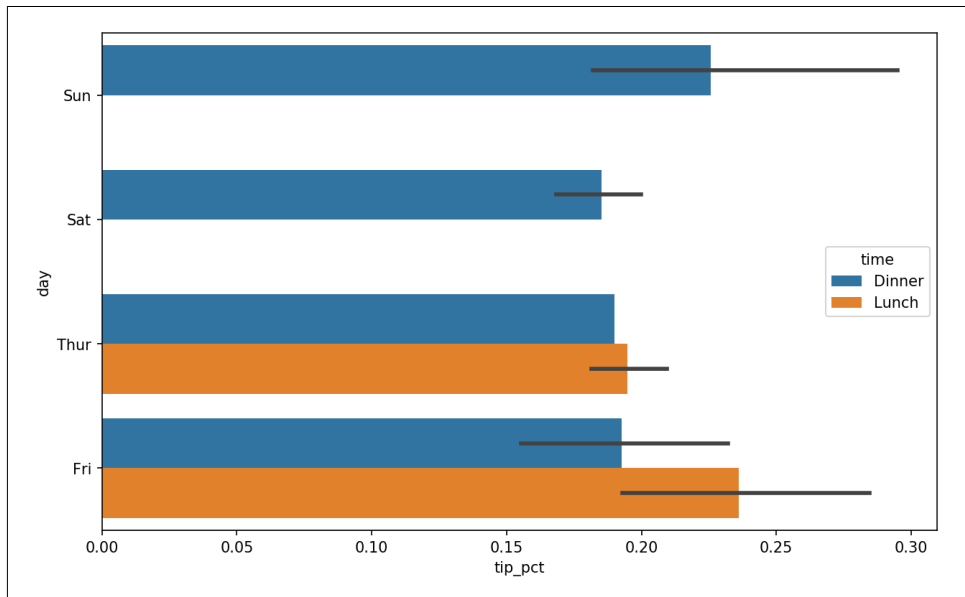


Figure 9-20. Tipping percentage by day and time

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 [90]: 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 9-21):

```
In [92]: tips['tip_pct'].plot.hist(bins=50)
```

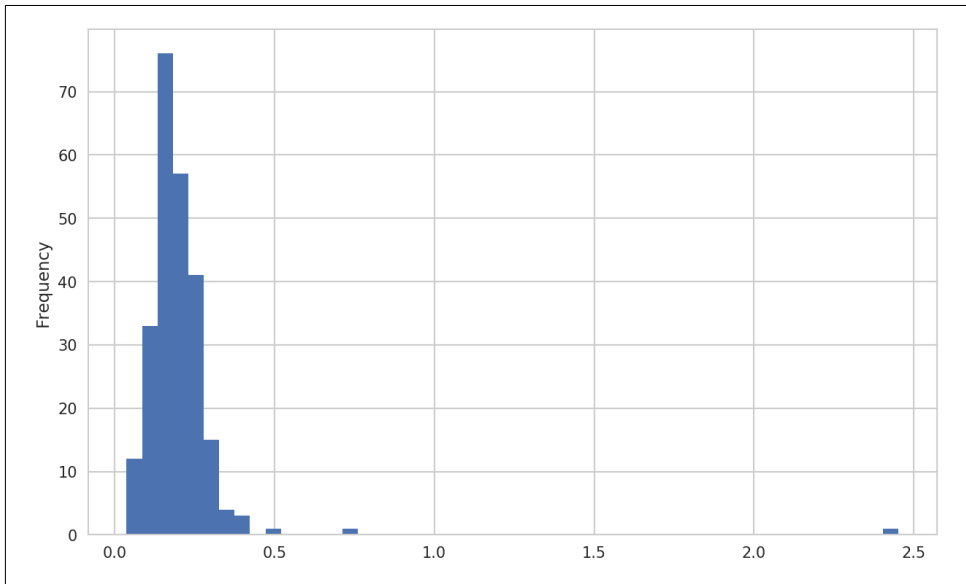


Figure 9-21. Histogram of tip percentages

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. The 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 kernel density estimate (KDE) plots. Using `plot.kde` makes a density plot using the conventional mixture-of-normals estimate (see [Figure 9-22](#)):

```
In [94]: tips['tip_pct'].plot.density()
```

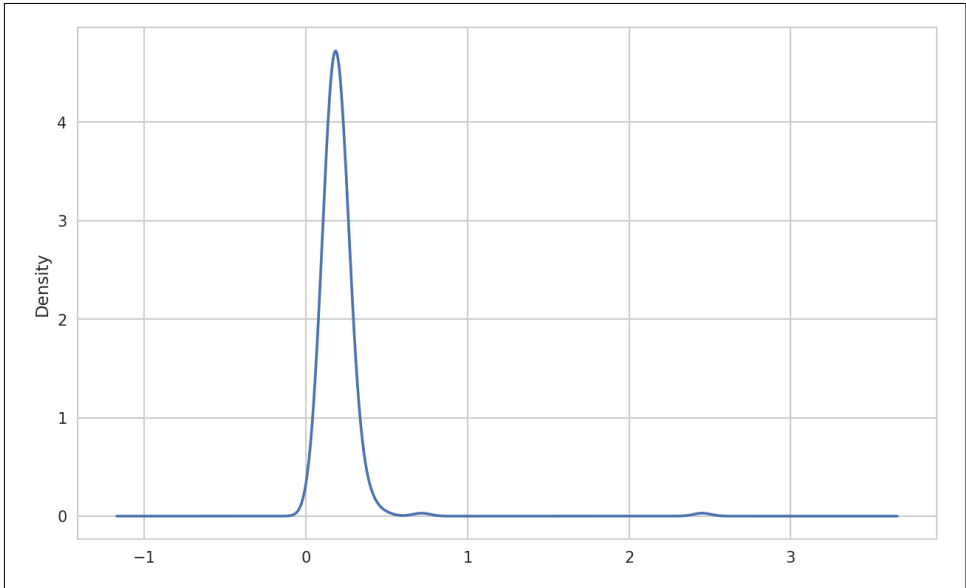



Figure 9-22. Density plot of tip percentages

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 9-23):

```
In [96]: comp1 = np.random.normal(0, 1, size=200)
In [97]: comp2 = np.random.normal(10, 2, size=200)
In [98]: values = pd.Series(np.concatenate([comp1, comp2]))
In [99]: sns.distplot(values, bins=100, color='k')
```

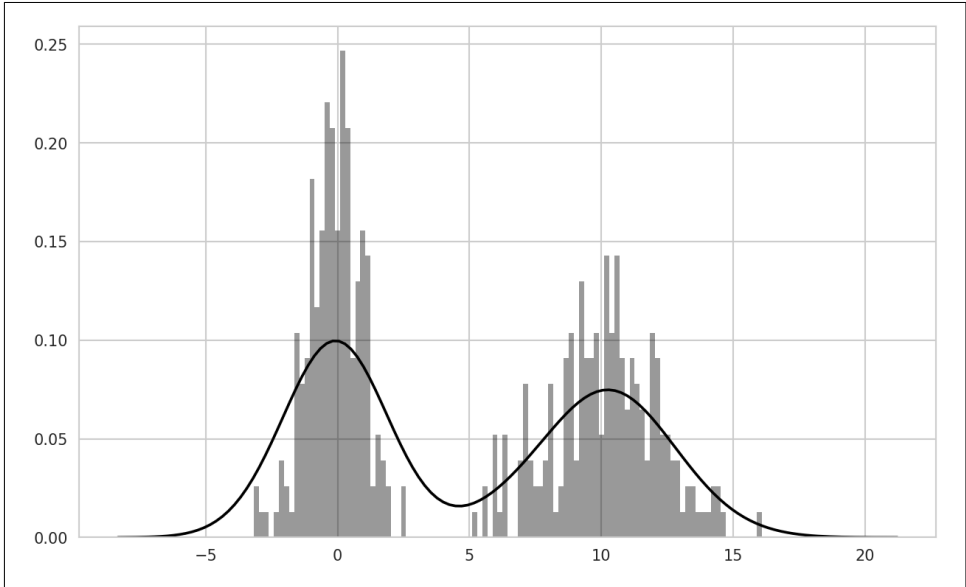


Figure 9-23. Normalized histogram of normal mixture with density estimate

Scatter or Point Plots

Point plots or scatter plots can be a useful way of examining the relationship between two one-dimensional data series. For example, here we load the `macrodata` dataset from the `statsmodels` project, select a few variables, then compute log differences:

```
In [100]: macro = pd.read_csv('examples/macrodata.csv')
In [101]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]
In [102]: trans_data = np.log(data).diff().dropna()
In [103]: trans_data[-5:]
Out[103]:
```

	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

We can then use seaborn's `regplot` method, which makes a scatter plot and fits a linear regression line (see [Figure 9-24](#)):

```
In [105]: sns.regplot('m1', 'unemp', data=trans_data)
Out[105]: <matplotlib.axes._subplots.AxesSubplot at 0x7fb613720be0>

In [106]: plt.title('Changes in log %s versus log %s' % ('m1', 'unemp'))
```

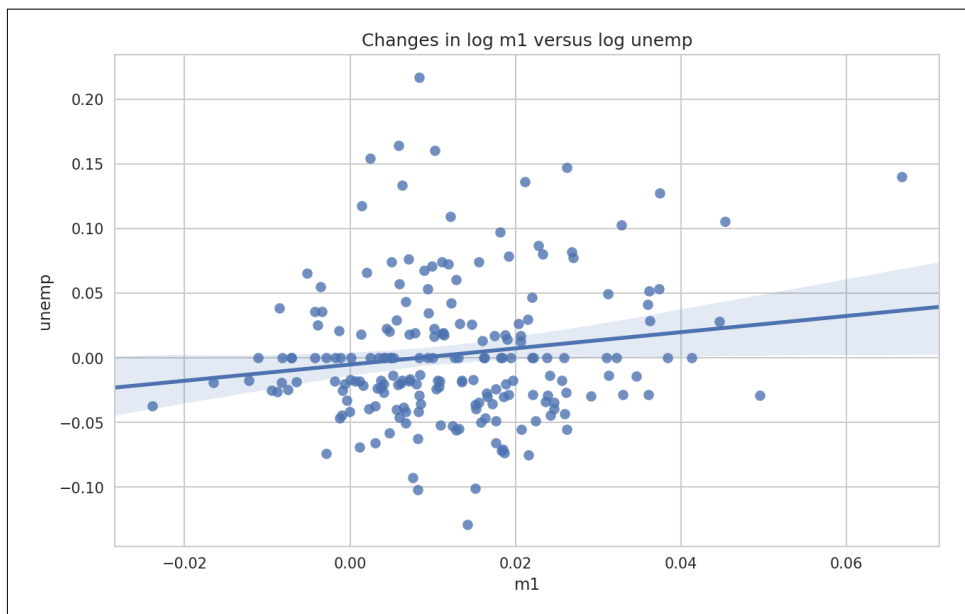


Figure 9-24. A seaborn regression/scatter plot

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, which supports placing histograms or density estimates of each variable along the diagonal (see [Figure 9-25](#) for the resulting plot):

```
In [107]: sns.pairplot(trans_data, diag_kind='kde', plot_kws={'alpha': 0.2})
```

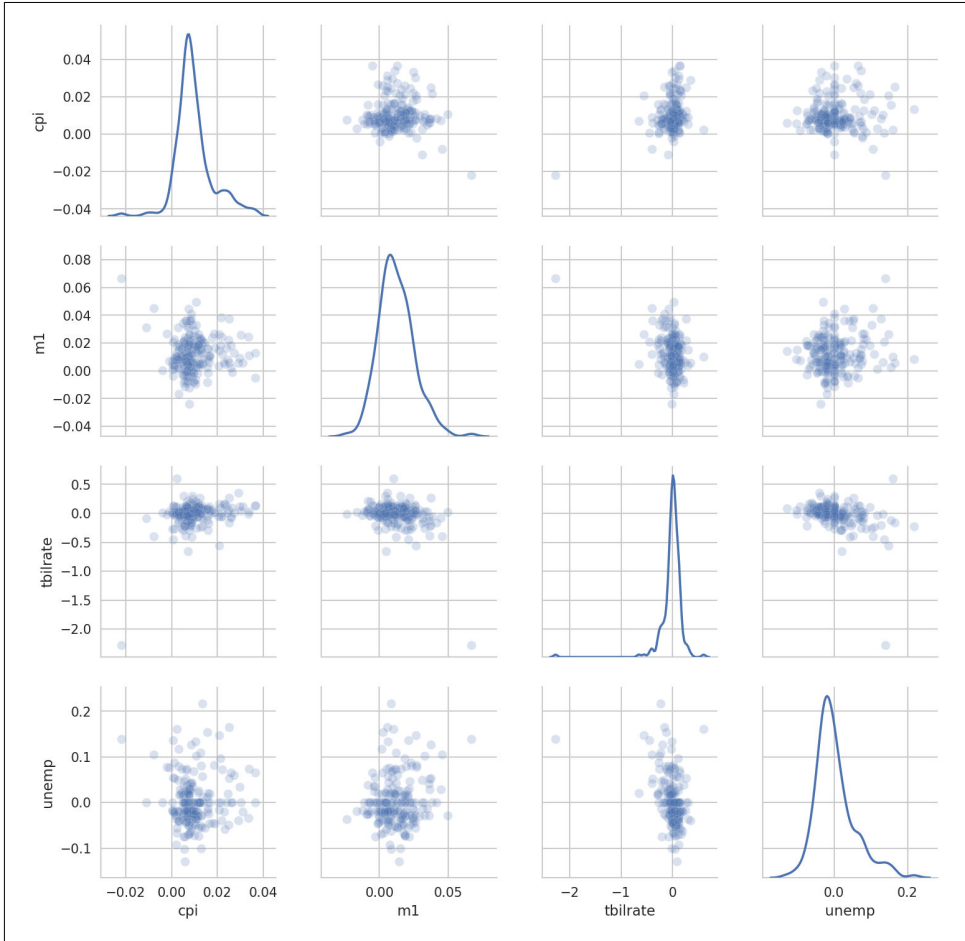


Figure 9-25. Pair plot matrix of statsmodels macro data

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

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 (see [Figure 9-26](#) for the resulting plot):

```
In [108]: sns.factorplot(x='day', y='tip_pct', hue='time', col='smoker',
.....:                  kind='bar', data=tips[tips.tip_pct < 1])
```

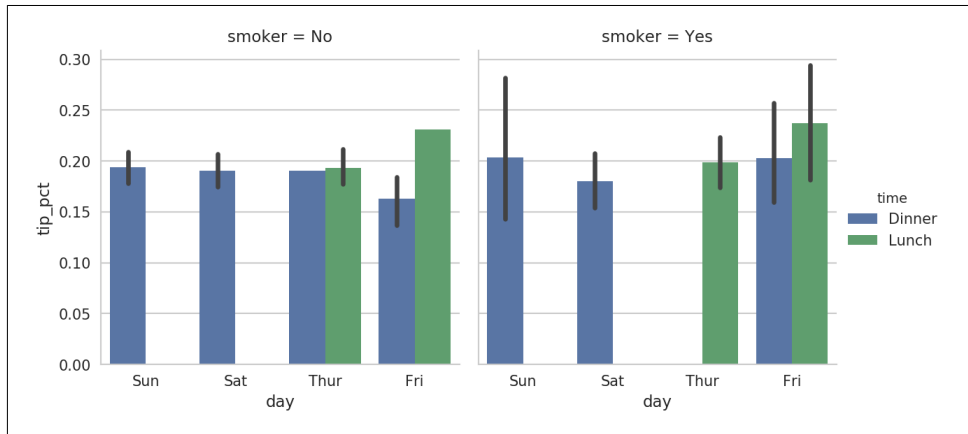


Figure 9-26. Tipping percentage by day/time/smoker

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 ([Figure 9-27](#)):

```
In [109]: sns.factorplot(x='day', y='tip_pct', row='time',
.....:                  col='smoker',
.....:                  kind='bar', data=tips[tips.tip_pct < 1])
```

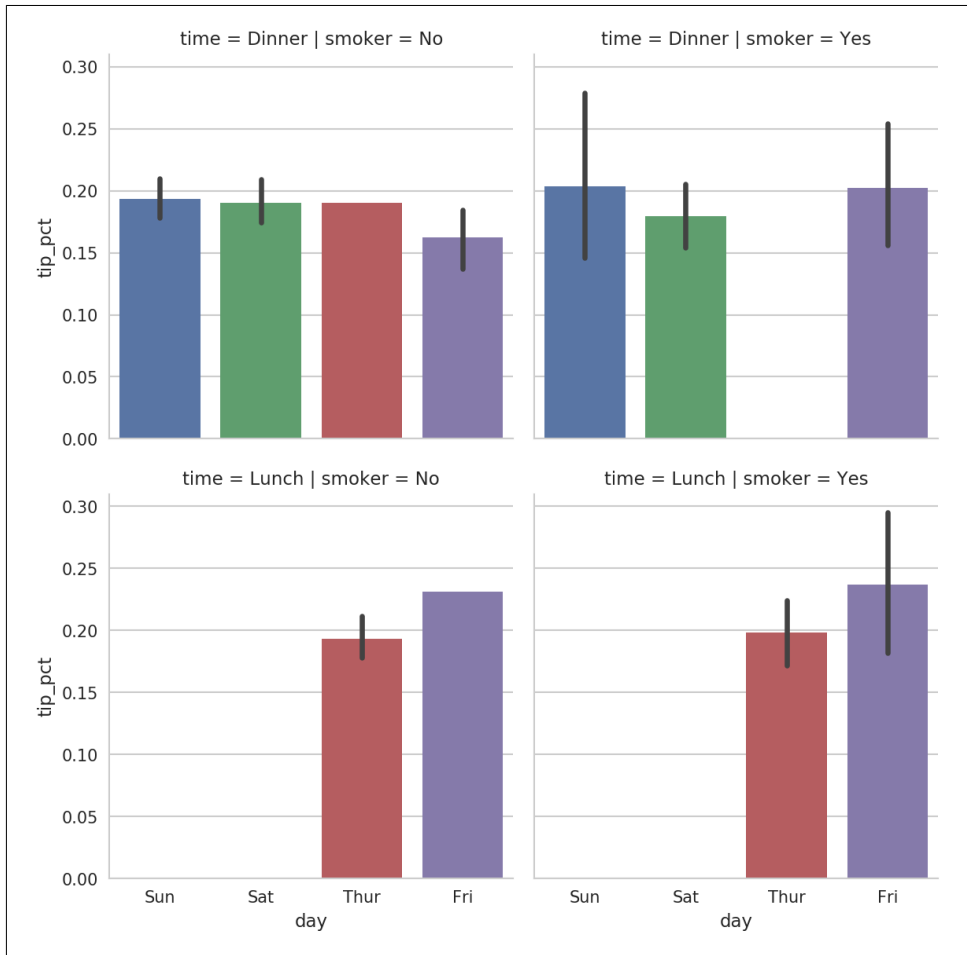


Figure 9-27. *tip_pct* by day; facet by time/smoker

`factorplot` supports other plot types that 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 9-28):

```
In [110]: sns.factorplot(x='tip_pct', y='day', kind='box',
.....:                  data=tips[tips.tip_pct < 0.5])
```

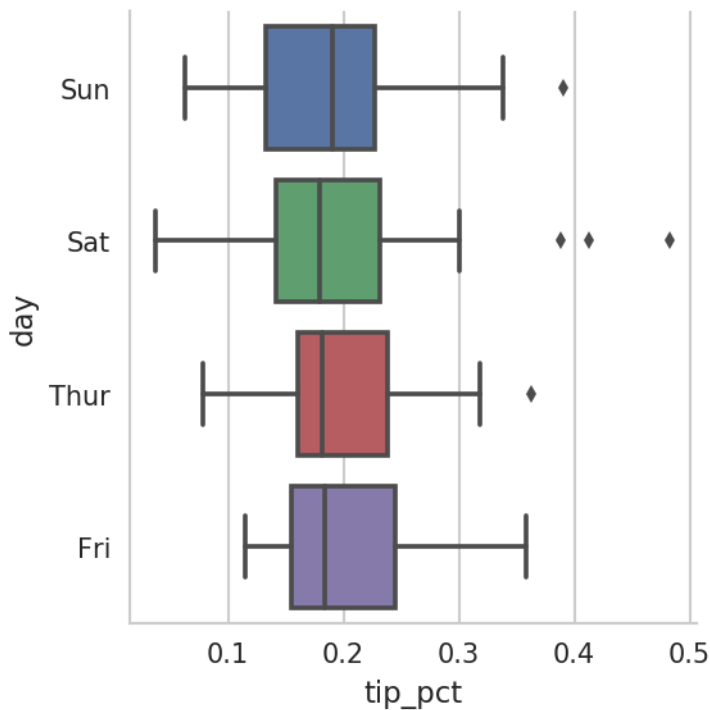


Figure 9-28. Box plot of `tip_pct` by day

You can create your own facet grid plots using the more general `seaborn.FacetGrid` class. See the [seaborn documentation](#) for more.

9.3 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** and **Plotly**, 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.

9.4 Conclusion

The goal of this chapter was to get your feet wet with some basic data visualization using pandas, matplotlib, and seaborn. If visually communicating the results of data analysis is important in your work, I encourage you to seek out resources to learn more about effective data visualization. It is an active field of research and you can practice with many excellent learning resources available online and in print form.

In the next chapter, we turn our attention to data aggregation and group operations with pandas.

Data Aggregation and Group Operations

Categorizing a dataset 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 dataset, you 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, dice, and summarize datasets 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)
- Calculate 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



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](#).

10.1 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.

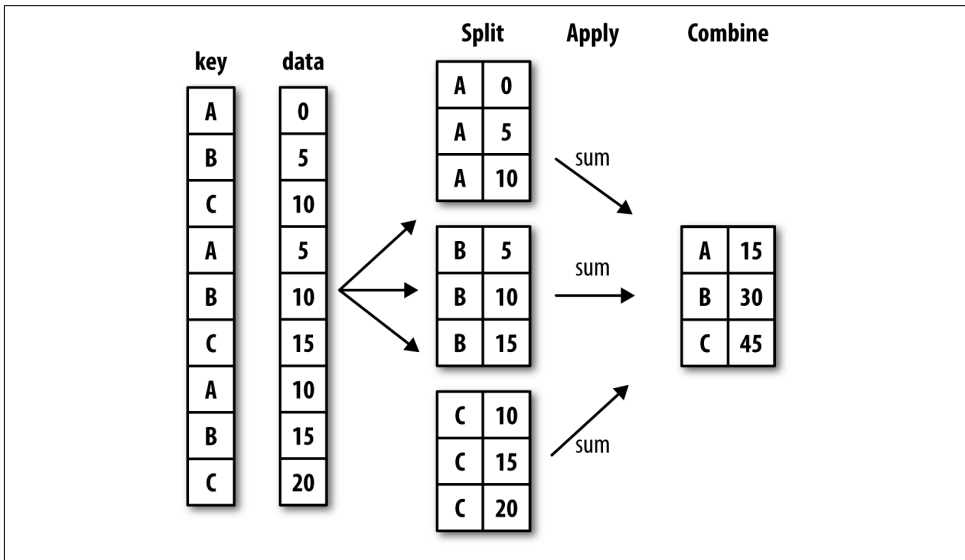


Figure 10-1. Illustration of a group aggregation

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 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', 'one'],
    ....:                   '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 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 0x7faa31537390>
```

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'd get something different:

```
In [15]: means = df['data1'].groupby([df['key1'], df['key2']]).mean()
```

```
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
```

Here 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 this example, the group keys are all Series, though they could be any arrays of the right length:

```
In [18]: states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])
```

```
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 is 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 returns a `Series` containing group sizes:

```
In [23]: df.groupby(['key1', 'key2']).size()
Out[23]:
```

key1	key2	size
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:

```
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 -0.55573  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      float64
data2      float64
key1       object
key2       object
dtype: object

```

```
In [29]: grouped = df.groupby(df.dtypes, axis=1)
```

We can print out the groups like so:

```
In [30]: for dtype, group in grouped:
```

```
....:     print(dtype)
```

```
....:     print(group)
```

```
....:
```

```
float64
```

```

      data1      data2
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

```

```
object
```

```
      key1 key2
```

```
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 column subsetting 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 datasets, it may be desirable to aggregate only a few columns. For example, in the preceding dataset, 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 or a grouped Series if only a single column name 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 0x7faa30c78da0>
```

```
In [34]: s_grouped.mean()
```

```
Out[34]:
```

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 construct an array from this dict to pass to groupby, but instead we can just pass the dict (I included the key 'f' to highlight that unused grouping keys are OK):

```
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:

```
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; while you could compute an array of string lengths, it's simpler to 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 datasets is the ability to aggregate using one of the levels of an axis index. Let's look at an example:

```
In [47]: columns = pd.MultiIndex.from_arrays([[ 'US', 'US', 'US', 'JP', 'JP'],  
.....:                                     [1, 3, 5, 1, 3]],  
.....:                                     names=['cty', 'tenor'])
```

```
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
```

To group by level, pass the level number or name using the `level` keyword:

```
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
```

10.2 Data Aggregation

Aggregations refer to any data transformation that produces scalar values from arrays. The preceding examples have used several of them, including `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.

Table 10-1. Optimized groupby methods

Function name	Description
<code>count</code>	Number of non-NA values in the group
<code>sum</code>	Sum of non-NA values
<code>mean</code>	Mean of non-NA values
<code>median</code>	Arithmetic median of non-NA values
<code>std</code> , <code>var</code>	Unbiased ($n - 1$ denominator) standard deviation and variance
<code>min</code> , <code>max</code>	Minimum and maximum of non-NA values
<code>prod</code>	Product of non-NA values
<code>first</code> , <code>last</code>	First and last non-NA values

You can use aggregations of your own devising and additionally call any method that is also defined on the grouped object. For example, you might recall that `quantile` computes sample quantiles of a `Series` or a `DataFrame`'s columns.

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, and then assembles those results together into the result object:

```
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
```

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%	75%	
key1								
a	3.0	0.746672	1.109736	-0.204708	0.137118	0.478943	1.222362	
b	2.0	-0.537585	0.025662	-0.555730	-0.546657	-0.537585	-0.528512	
		data2						
		max	count	mean	std	min	25%	50%
key1								
a	1.965781	3.0	0.910916	0.712217	0.092908	0.669671	1.246435	
b	-0.519439	2.0	0.525384	0.344556	0.281746	0.403565	0.525384	
		75%	max					
key1								

```
a    1.319920  1.393406
b    0.647203  0.769023
```

I will explain in more detail what has happened here in [Section 10.3](#), “Apply: General split-apply-combine,” on page 302.



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.

Column-Wise and Multiple Function Application

Let’s return to the tipping dataset from earlier examples. After loading it with `read_csv`, we 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	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808
5	25.29	4.71	No	Sun	Dinner	4	0.186240

As you’ve already seen, 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 day and smoker:

```
In [60]: grouped = tips.groupby(['day', '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]:
```

day	smoker	mean
Fri	No	0.151650
	Yes	0.174783
Sat	No	0.158048

```
Yes      0.147906
Sun No    0.160113
Yes      0.187250
Thur No   0.160298
Yes      0.163863
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
day	smoker			
Fri	No	0.151650	0.028123	0.067349
	Yes	0.174783	0.051293	0.159925
Sat	No	0.158048	0.039767	0.235193
	Yes	0.147906	0.061375	0.290095
Sun	No	0.160113	0.042347	0.193226
	Yes	0.187250	0.154134	0.644685
Thur	No	0.160298	0.038774	0.193350
	Yes	0.163863	0.039389	0.151240

Here we passed a list of aggregation functions to `agg` to evaluate independently on the data groups.

You don't need to accept the names that `GroupBy` gives to the columns; notably, lambda functions have the name '<lambda>', which makes them hard to identify (you can see for yourself by looking at a function's `__name__` attribute). Thus, 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
day	smoker		
Fri	No	0.151650	0.028123
	Yes	0.174783	0.051293
Sat	No	0.158048	0.039767
	Yes	0.147906	0.061375
Sun	No	0.160113	0.042347
	Yes	0.187250	0.154134
Thur	No	0.160298	0.038774
	Yes	0.163863	0.039389

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	max
day	smoker						
Fri	No	4	0.151650	0.187735	4	18.420000	22.75
	Yes	15	0.174783	0.263480	15	16.813333	40.17
Sat	No	45	0.158048	0.291990	45	19.661778	48.33
	Yes	42	0.147906	0.325733	42	21.276667	50.81
Sun	No	57	0.160113	0.252672	57	20.506667	48.17
	Yes	19	0.187250	0.710345	19	24.120000	45.35
Thur	No	45	0.160298	0.266312	45	17.113111	41.19
	Yes	17	0.163863	0.241255	17	19.190588	43.11

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 column names as the keys argument:

```
In [68]: result['tip_pct']
```

```
Out[68]:
```

		count	mean	max
day	smoker			
Fri	No	4	0.151650	0.187735
	Yes	15	0.174783	0.263480
Sat	No	45	0.158048	0.291990
	Yes	42	0.147906	0.325733
Sun	No	57	0.160113	0.252672
	Yes	19	0.187250	0.710345
Thur	No	45	0.160298	0.266312
	Yes	17	0.163863	0.241255

As before, a list of tuples with custom names can be passed:

```
In [69]: ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]
```

```
In [70]: grouped['tip_pct', 'total_bill'].agg(ftuples)
```

```
Out[70]:
```

		tip_pct		total_bill	
		Durchschnitt	Abweichung	Durchschnitt	Abweichung
day	smoker				
Fri	No	0.151650	0.000791	18.420000	25.596333
	Yes	0.174783	0.002631	16.813333	82.562438
Sat	No	0.158048	0.001581	19.661778	79.908965
	Yes	0.147906	0.003767	21.276667	101.387535
Sun	No	0.160113	0.001793	20.506667	66.099980
	Yes	0.187250	0.023757	24.120000	109.046044
Thur	No	0.160298	0.001503	17.113111	59.625081
	Yes	0.163863	0.001551	19.190588	69.808518

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
day	smoker		
Fri	No	3.50	9
	Yes	4.73	31
Sat	No	9.00	115
	Yes	10.00	104
Sun	No	6.00	167
	Yes	6.50	49
Thur	No	6.70	112
	Yes	5.00	40

```
In [72]: grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],
....:                  'size' : 'sum'})
Out[72]:
```

		tip_pct				size
		min	max	mean	std	sum
day	smoker					
Fri	No	0.120385	0.187735	0.151650	0.028123	9
	Yes	0.103555	0.263480	0.174783	0.051293	31
Sat	No	0.056797	0.291990	0.158048	0.039767	115
	Yes	0.035638	0.325733	0.147906	0.061375	104
Sun	No	0.059447	0.252672	0.160113	0.042347	167
	Yes	0.065660	0.710345	0.187250	0.154134	49
Thur	No	0.072961	0.266312	0.160298	0.038774	112
	Yes	0.090014	0.241255	0.163863	0.039389	40

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. 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(['day', 'smoker'], as_index=False).mean()
Out[73]:
```

	day	smoker	total_bill	tip	size	tip_pct
0	Fri	No	18.420000	2.812500	2.250000	0.151650
1	Fri	Yes	16.813333	2.714000	2.066667	0.174783
2	Sat	No	19.661778	3.102889	2.555556	0.158048
3	Sat	Yes	21.276667	2.875476	2.476190	0.147906
4	Sun	No	20.506667	3.167895	2.929825	0.160113
5	Sun	Yes	24.120000	3.516842	2.578947	0.187250

```

6 Thur No 17.113111 2.673778 2.488889 0.160298
7 Thur Yes 19.190588 3.030000 2.352941 0.163863

```

Of course, it's always possible to obtain the result in this format by calling `reset_index` on the result. Using the `as_index=False` method avoids some unnecessary computations.

10.3 Apply: General split-apply-combine

The most general-purpose `GroupBy` method is `apply`, which is the subject of the rest of this section. As illustrated in [Figure 10-2](#), `apply` splits the object being manipulated into pieces, invokes the passed function on each piece, and then attempts to concatenate the pieces together.

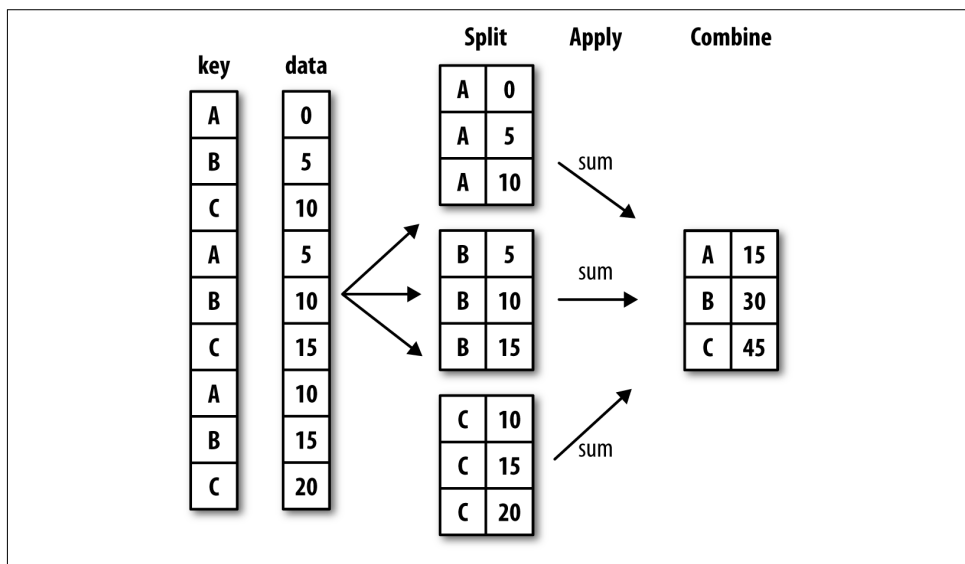


Figure 10-2. Illustration of a group aggregation

Returning to the tipping dataset from before, 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 smoker  day   time  size  tip_pct
109     14.31   4.00   Yes  Sat  Dinner    2  0.279525
183     23.17   6.50   Yes  Sun  Dinner    4  0.280535
232     11.61   3.39   No   Sat  Dinner    2  0.291990

```


67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
172	7.25	5.15	Yes	Sun	Dinner	2	0.710345

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	smoker	day	time	size	tip_pct	
smoker	No	88	24.71	5.85	No	Thur	Lunch	2	0.236746
		185	20.69	5.00	No	Sun	Dinner	5	0.241663
		51	10.29	2.60	No	Sun	Dinner	2	0.252672
		149	7.51	2.00	No	Thur	Lunch	2	0.266312
		232	11.61	3.39	No	Sat	Dinner	2	0.291990
Yes	109	109	14.31	4.00	Yes	Sat	Dinner	2	0.279525
		183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
		67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
		178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
		172	7.25	5.15	Yes	Sun	Dinner	2	0.710345

What has happened here? The `top` function is called on each row group from the DataFrame, and 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='total_bill')
Out[77]:
```

			total_bill	tip	smoker	day	time	size	tip_pct		
smoker	day	No	Fri	94	22.75	3.25	No	Fri	Dinner	2	0.142857
			Sat	212	48.33	9.00	No	Sat	Dinner	4	0.186220
			Sun	156	48.17	5.00	No	Sun	Dinner	6	0.103799
Yes	Thur	142	142	41.19	5.00	No	Thur	Lunch	5	0.121389	
			Fri	95	40.17	4.73	Yes	Fri	Dinner	4	0.117750
			Sat	170	50.81	10.00	Yes	Sat	Dinner	3	0.196812
			Sun	182	45.35	3.50	Yes	Sun	Dinner	3	0.077178
			Thur	197	43.11	5.00	Yes	Thur	Lunch	4	0.115982



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 that I earlier 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%	75%	\
smoker								
No	151.0	0.159328	0.039910	0.056797	0.136906	0.155625	0.185014	
Yes	93.0	0.163196	0.085119	0.035638	0.106771	0.153846	0.195059	
		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 preceding examples, 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. You can disable this by passing `group_keys=False` to `groupby`:

```
In [81]: tips.groupby('smoker', group_keys=False).apply(top)
Out[81]:
```

	total_bill	tip	smoker	day	time	size	tip_pct
88	24.71	5.85	No	Thur	Lunch	2	0.236746
185	20.69	5.00	No	Sun	Dinner	5	0.241663
51	10.29	2.60	No	Sun	Dinner	2	0.252672
149	7.51	2.00	No	Thur	Lunch	2	0.266312
232	11.61	3.39	No	Sat	Dinner	2	0.291990
109	14.31	4.00	Yes	Sat	Dinner	2	0.279525
183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
172	7.25	5.15	Yes	Sun	Dinner	2	0.710345

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` makes it convenient to perform bucket or quantile analysis on a dataset. Consider a simple random dataset 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, 0.489] < (0.489, 2.208] < (2.208, 3.928]]
```

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 [Chapter 12](#).

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 US states divided into eastern and western regions:

```
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
Out[98]:
Ohio          0.922264
New York     -2.153545
Vermont      -0.365757
Florida      -0.375842
Oregon        0.329939
Nevada        0.981994
California   1.105913
Idaho        -1.613716
dtype: float64
```

Note that the syntax `['East'] * 4` produces a list containing four copies of the elements in `['East']`. Adding lists together concatenates them.

Let's set some values in the data to be missing:

```
In [99]: data[['Vermont', 'Nevada', 'Idaho']] = np.nan

In [100]: data
Out[100]:
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 [101]: data.groupby(group_key).mean()
Out[101]:
East    -0.535707
West     0.717926
dtype: float64
```

We can fill the NA values using the group means like so:

```
In [102]: fill_mean = lambda g: g.fillna(g.mean())

In [103]: data.groupby(group_key).apply(fill_mean)
Out[103]:
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 predefined fill values in your code that vary by group. Since the groups have a `name` attribute set internally, we can use that:

```
In [104]: fill_values = {'East': 0.5, 'West': -1}

In [105]: fill_func = lambda g: g.fillna(fill_values[g.name])

In [106]: data.groupby(group_key).apply(fill_func)
Out[106]:
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”; here we use the `sample` method for Series.

To demonstrate, 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 [108]: deck[:13]
Out[108]:
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 before, drawing a hand of five cards from the deck could be written as:

```
In [109]: def draw(deck, n=5):
.....:     return deck.sample(n)

In [110]: draw(deck)
Out[110]:
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 [111]: get_suit = lambda card: card[-1] # last letter is suit

In [112]: deck.groupby(get_suit).apply(draw, n=2)
Out[112]:
```

```
C 2C 2
   3C 3
D  KD 10
   8D 8
H  KH 10
   3H 3
S  2S 2
   4S 4
dtype: int64
```

Alternatively, we could write:

```
In [113]: deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
Out[113]:
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 as a group weighted average, are possible. As an example, take this dataset containing group keys, values, and some weights:

```
In [114]: df = pd.DataFrame({'category': ['a', 'a', 'a', 'a',
.....:                                     'b', 'b', 'b', 'b'],
.....:                       'data': np.random.randn(8),
.....:                       'weights': np.random.rand(8)})
```

```
In [115]: df
Out[115]:
  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 [116]: grouped = df.groupby('category')

In [117]: get_wavg = lambda g: np.average(g['data'], weights=g['weights'])
```



```
In [118]: grouped.apply(get_wavg)
Out[118]:
category
a    0.811643
b   -0.122262
dtype: float64
```

As another example, consider a financial dataset originally obtained from Yahoo! Finance containing end-of-day prices for a few stocks and the S&P 500 index (the SPX symbol):

```
In [119]: close_px = pd.read_csv('examples/stock_px_2.csv', parse_dates=True,
.....:                             index_col=0)
```

```
In [120]: 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 [121]: close_px[-4:]
Out[121]:
```

	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. As one way to do this, we first create a function that computes the pairwise correlation of each column with the 'SPX' column:

```
In [122]: spx_corr = lambda x: x.corrwith(x['SPX'])
```

Next, we compute percent change on close_px using pct_change:

```
In [123]: rets = close_px.pct_change().dropna()
```

Lastly, we group these percent changes by year, which can be extracted from each row label with a one-line function that returns the year attribute of each datetime label:

```
In [124]: get_year = lambda x: x.year
```

```
In [125]: by_year = rets.groupby(get_year)
```

```
In [126]: by_year.apply(spx_corr)
Out[126]:
```

	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. Here we compute the annual correlation between Apple and Microsoft:

```
In [127]: by_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))
Out[127]:
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 [129]: by_year.apply(regress, 'AAPL', ['SPX'])
Out[129]:
      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

```

10.4 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 through the `groupby` facility described in this chapter combined with reshape operations utilizing hierarchical indexing. `DataFrame` has a `pivot_table` method, and there is also a top-level `pandas.pivot_table` function. In addition to providing a convenience interface to `groupby`, `pivot_table` can add partial totals, also known as *margins*.

Returning to the tipping dataset, suppose you wanted to compute a table of group means (the default `pivot_table` aggregation type) arranged by day and smoker on the rows:

```

In [130]: tips.pivot_table(index=['day', 'smoker'])
Out[130]:

```

day	smoker	size	tip	tip_pct	total_bill
Fri	No	2.250000	2.812500	0.151650	18.420000
	Yes	2.066667	2.714000	0.174783	16.813333
Sat	No	2.555556	3.102889	0.158048	19.661778
	Yes	2.476190	2.875476	0.147906	21.276667
Sun	No	2.929825	3.167895	0.160113	20.506667
	Yes	2.578947	3.516842	0.187250	24.120000
Thur	No	2.488889	2.673778	0.160298	17.113111
	Yes	2.352941	3.030000	0.163863	19.190588

This could have been produced with `groupby` directly. Now, suppose we want to aggregate only `tip_pct` and `size`, and additionally group by `time`. I'll put `smoker` in the table columns and `day` in the rows:

```

In [131]: tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'],
.....:                      columns='smoker')
Out[131]:

```

		size		tip_pct		
		No	Yes	No	Yes	
time	Dinner	Fri	2.000000	2.222222	0.139622	0.165347
	Sat	2.555556	2.476190	0.158048	0.147906	
	Sun	2.929825	2.578947	0.160113	0.187250	
	Thur	2.000000	NaN	0.159744	NaN	

```
Lunch Fri 3.000000 1.833333 0.187735 0.188937
      Thur 2.500000 2.352941 0.160311 0.163863
```

We could augment this table 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 [132]: tips.pivot_table(['tip_pct', 'size'], index=['time', 'day'],
.....:                    columns='smoker', margins=True)
Out[132]:
```

		size			tip_pct		
smoker		No	Yes	All	No	Yes	All
time	day						
Dinner	Fri	2.000000	2.222222	2.166667	0.139622	0.165347	0.158916
	Sat	2.555556	2.476190	2.517241	0.158048	0.147906	0.153152
	Sun	2.929825	2.578947	2.842105	0.160113	0.187250	0.166897
	Thur	2.000000	NaN	2.000000	0.159744	NaN	0.159744
Lunch	Fri	3.000000	1.833333	2.000000	0.187735	0.188937	0.188765
	Thur	2.500000	2.352941	2.459016	0.160311	0.163863	0.161301
All		2.668874	2.408602	2.569672	0.159328	0.163196	0.160803

Here, the All values are means without taking into account smoker versus non-smoker (the All columns) or any of the two levels of grouping on the rows (the All row).

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 [133]: tips.pivot_table('tip_pct', index=['time', 'smoker'], columns='day',
.....:                    aggfunc=len, margins=True)
Out[133]:
```

		Fri	Sat	Sun	Thur	All
time	smoker					
Dinner	No	3.0	45.0	57.0	1.0	106.0
	Yes	9.0	42.0	19.0	NaN	70.0
Lunch	No	1.0	NaN	NaN	44.0	45.0
	Yes	6.0	NaN	NaN	17.0	23.0
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 [134]: tips.pivot_table('tip_pct', index=['time', 'size', 'smoker'],
.....:                    columns='day', aggfunc='mean', fill_value=0)
Out[134]:
```

			Fri	Sat	Sun	Thur
time	size	smoker				
Dinner	1	No	0.000000	0.137931	0.000000	0.000000
		Yes	0.000000	0.325733	0.000000	0.000000
	2	No	0.139622	0.162705	0.168859	0.159744
		Yes	0.171297	0.148668	0.207893	0.000000
	3	No	0.000000	0.154661	0.152663	0.000000

```

    Yes    0.000000  0.144995  0.152660  0.000000
  4    No    0.000000  0.150096  0.148143  0.000000
    Yes    0.117750  0.124515  0.193370  0.000000
  5    No    0.000000  0.000000  0.206928  0.000000
    Yes    0.000000  0.106572  0.065660  0.000000
...
Lunch  1    No    0.000000  0.000000  0.000000  0.181728
    Yes    0.223776  0.000000  0.000000  0.000000
  2    No    0.000000  0.000000  0.000000  0.166005
    Yes    0.181969  0.000000  0.000000  0.158843
  3    No    0.187735  0.000000  0.000000  0.084246
    Yes    0.000000  0.000000  0.000000  0.204952
  4    No    0.000000  0.000000  0.000000  0.138919
    Yes    0.000000  0.000000  0.000000  0.155410
  5    No    0.000000  0.000000  0.000000  0.121389
  6    No    0.000000  0.000000  0.000000  0.173706

```

[21 rows x 4 columns]

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 ('mean' by default); can be any function valid in a groupby context
<code>fill_value</code>	Replace missing values in result table
<code>dropna</code>	If True, do not include columns whose entries are all NA
<code>margins</code>	Add row/column subtotals and grand total (False 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:

```

In [138]: data
Out[138]:
  Sample Nationality  Handedness
0      1          USA  Right-handed
1      2          Japan Left-handed
2      3          USA  Right-handed
3      4          Japan Right-handed
4      5          Japan Left-handed
5      6          Japan Right-handed
6      7          USA  Right-handed
7      8          USA  Left-handed
8      9          Japan Right-handed
9     10          USA  Right-handed

```

As part of some survey analysis, we might want to summarize this data by nationality and handedness. You could use `pivot_table` to do this, but the `pandas.crosstab` function can be more convenient:

```
In [139]: pd.crosstab(data.Nationality, data.Handedness, margins=True)
Out[139]:
Handedness  Left-handed  Right-handed  All
Nationality
Japan                2             3    5
USA                  1             4    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 [140]: pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)
Out[140]:
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
```

10.5 Conclusion

Mastering pandas's data grouping tools can help both with data cleaning as well as modeling or statistical analysis work. In [Chapter 14](#) we will look at several more example use cases for `groupby` on real data.

In the next chapter, we turn our attention to time series data.

Time Series

Time series data is an important form of structured data in many different fields, such as finance, economics, ecology, neuroscience, and 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 of 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 (e.g., 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 three 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 also supports indexes based on timedeltas, which can be a useful way of representing experiment or elapsed time. We do not explore timedelta indexes in this book, but you can learn more in the [pandas documentation](#).

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.

11.1 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, 9, 25, 14, 5, 52, 72973)
```

```
In [13]: now.year, now.month, now.day
```

```
Out[13]: (2017, 9, 25)
```

`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, 8, 15)
```

```
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)
```

Table 11-1 summarizes the data types in the `datetime` module. 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 in 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 <code>datetime</code> values (as days, seconds, and microseconds)
<code>tzinfo</code>	Base type for storing time zone information

Converting Between String and Datetime

You can format `datetime` objects and pandas `Timestamp` objects, which I'll introduce later, 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 11-2 for a complete list of the format codes (reproduced from Chapter 2).

Table 11-2. Datetime format specification (ISO C89 compatible)

Type	Description
<code>%Y</code>	Four-digit year
<code>%y</code>	Two-digit year
<code>%m</code>	Two-digit month [01, 12]
<code>%d</code>	Two-digit day [01, 31]
<code>%H</code>	Hour (24-hour clock) [00, 23]
<code>%I</code>	Hour (12-hour clock) [01, 12]
<code>%M</code>	Two-digit minute [00, 59]
<code>%S</code>	Second [00, 61] (seconds 60, 61 account for leap seconds)
<code>%w</code>	Weekday as integer [0 (Sunday), 6]

Type	Description
%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 (e.g., 2012-4-18)
%D	Shortcut for %m/%d/%y (e.g., 04/18/12)

You can use these same format codes 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.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 you install `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'], dtype='datetime64[ns]', freq=None)
```

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'], dtype='datetime64[ns]', freq=None)
```

```
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.



`dateutil.parser` is a useful but imperfect tool. Notably, it will recognize some strings as dates that you might prefer that it didn't—for example, '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 11-3](#) for a listing.

Table 11-3. 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 (e.g., 'Tue 01 May 2012 04:20:57 PM')
%p	Locale equivalent of AM or PM
%x	Locale-appropriate formatted date (e.g., in the United States, May 1, 2012 yields '05/01/2012')
%X	Locale-appropriate time (e.g., '04:24:12 PM')

11.2 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, 7), datetime(2011, 1, 8),
....:             datetime(2011, 1, 10), datetime(2011, 1, 12)]

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', '2011-01-08',
              '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
```

Recall that `ts[::-2]` selects every second element in `ts`.

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 you are 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', periods=1000))
```

```
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
```

Here, the string '2001' is interpreted as a year and selects that time period. This also works if you specify the month:

```
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 as well:

```
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 time-stamps 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. This means that no data is copied and modifications on the slice will be reflected in the original data.

There is an equivalent instance method, `truncate`, that slices a Series 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 this 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',
....:                              '1/2/2000', '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`:

```
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
```

11.3 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, you can convert the sample time series to be fixed daily frequency 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')
```

The string 'D' is interpreted as daily frequency.

Conversion between frequencies or *resampling* is a big enough topic to have its own section later (Section 11.6, “Resampling and Frequency Conversion,” on page 348). 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', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20',
               '2012-04-21', '2012-04-22', '2012-04-23', '2012-04-24',
               '2012-04-25', '2012-04-26', '2012-04-27', '2012-04-28',
               '2012-04-29', '2012-04-30', '2012-05-01', '2012-05-02',
               '2012-05-03', '2012-05-04', '2012-05-05', '2012-05-06',
               '2012-05-07', '2012-05-08', '2012-05-09', '2012-05-10',
               '2012-05-11', '2012-05-12', '2012-05-13', '2012-05-14',
               '2012-05-15', '2012-05-16', '2012-05-17', '2012-05-18',
               '2012-05-19', '2012-05-20', '2012-05-21', '2012-05-22',
               '2012-05-23', '2012-05-24', '2012-05-25', '2012-05-26',
               '2012-05-27', '2012-05-28', '2012-05-29', '2012-05-30',
               '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', '2012-04-04',
               '2012-04-05', '2012-04-06', '2012-04-07', '2012-04-08',
               '2012-04-09', '2012-04-10', '2012-04-11', '2012-04-12',
               '2012-04-13', '2012-04-14', '2012-04-15', '2012-04-16',
               '2012-04-17', '2012-04-18', '2012-04-19', '2012-04-20'],
              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', '2012-05-16',
               '2012-05-17', '2012-05-18', '2012-05-19', '2012-05-20',
               '2012-05-21', '2012-05-22', '2012-05-23', '2012-05-24',
               '2012-05-25', '2012-05-26', '2012-05-27', '2012-05-28',
               '2012-05-29', '2012-05-30', '2012-05-31', '2012-06-01'],
              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; see more complete listing

of frequencies in Table 11-4) 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', '2000-04-28',
               '2000-05-31', '2000-06-30', '2000-07-31', '2000-08-31',
               '2000-09-29', '2000-10-31', '2000-11-30'],
              dtype='datetime64[ns]', freq='BM')
```

Table 11-4. 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 (1/1,000 of 1 second)
U	Micro	Microsecond (1/1,000,000 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, W-TUE, ...	Week	Weekly on given day of week (MON, TUE, WED, THU, FRI, SAT, or SUN)
WOM-1MON, WOM-2MON, ...	WeekOfMonth	Generate weekly dates in the first, second, third, or fourth week of the month (e.g., WOM-3FRI for the third Friday of each month)
Q-JAN, Q-FEB, ...	QuarterEnd	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)
BQ-JAN, BQ-FEB, ...	BusinessQuarterEnd	Quarterly dates anchored on last weekday day of each month, for year ending in indicated month
QS-JAN, QS-FEB, ...	QuarterBegin	Quarterly dates anchored on first calendar day of each month, for year ending in indicated month
BQS-JAN, BQS-FEB, ...	BusinessQuarterBegin	Quarterly dates anchored on first weekday day of each month, for year ending in indicated month
A-JAN, A-FEB, ...	YearEnd	Annual dates anchored on last calendar day of given month (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC)
BA-JAN, BA-FEB, ...	BusinessYearEnd	Annual dates anchored on last weekday of given month
AS-JAN, AS-FEB, ...	YearBegin	Annual dates anchored on first day of given month
BAS-JAN, BAS-FEB, ...	BusinessYearBegin	Annual dates anchored on first weekday of given month

`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', '2012-05-05',
              '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='4h')
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'],
              dtype='datetime64[ns]', freq='4H')
```

```
'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 '1h30min', that 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.

Refer back to [Table 11-4](#) for a listing of frequency codes and date offset classes available in pandas.



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.

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, freq='M'))
```

```
In [92]: ts  
Out[92]:  
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 [93]: ts.shift(2)  
Out[93]:  
2000-01-31         NaN  
2000-02-29         NaN  
2000-03-31    -0.066748  
2000-04-30     0.838639  
Freq: M, dtype: float64
```

```
In [94]: ts.shift(-2)  
Out[94]:  
2000-01-31    -0.117388  
2000-02-29    -0.517795  
2000-03-31         NaN  
2000-04-30         NaN  
Freq: M, dtype: float64
```

When we shift like this, missing data is introduced either at the start or the end of the time series.

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='90T')
Out[97]:
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
```

The T here stands for minutes.

Shifting dates with offsets

The pandas date offsets can also be used with `datetime` or `Timestamp` objects:

```
In [98]: from pandas.tseries.offsets import Day, MonthEnd
```

```
In [99]: now = datetime(2011, 11, 17)
```

```
In [100]: now + 3 * Day()
Out[100]: 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 [101]: now + MonthEnd()
Out[101]: Timestamp('2011-11-30 00:00:00')
```

```
In [102]: now + MonthEnd(2)
Out[102]: Timestamp('2011-12-31 00:00:00')
```

Anchored offsets can explicitly “roll” dates forward or backward by simply using their `rollforward` and `rollback` methods, respectively:

```
In [103]: offset = MonthEnd()

In [104]: offset.rollforward(now)
Out[104]: Timestamp('2011-11-30 00:00:00')
```

```
In [105]: offset.rollback(now)
Out[105]: Timestamp('2011-10-31 00:00:00')
```

A creative use of date offsets is to use these methods with groupby:

```
In [106]: ts = pd.Series(np.random.randn(20),
.....:                    index=pd.date_range('1/15/2000', periods=20, freq='4d'))
```

```
In [107]: ts
```

```
Out[107]:
2000-01-15    -0.116696
2000-01-19     2.389645
2000-01-23    -0.932454
2000-01-27    -0.229331
2000-01-31    -1.140330
2000-02-04     0.439920
2000-02-08    -0.823758
2000-02-12    -0.520930
2000-02-16     0.350282
2000-02-20     0.204395
2000-02-24     0.133445
2000-02-28     0.327905
2000-03-03     0.072153
2000-03-07     0.131678
2000-03-11    -1.297459
2000-03-15     0.997747
2000-03-19     0.870955
2000-03-23    -0.991253
2000-03-27     0.151699
2000-03-31     1.266151
Freq: 4D, dtype: float64
```

```
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` (we'll discuss this in much more depth in [Section 11.6, "Resampling and Frequency Conversion,"](#) on page 348):

```
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
```


11.4 Time Zone Handling

Working with time zones is generally considered one of the most unpleasant parts of time series manipulation. As a result, 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 saving time and five hours behind the rest of the year.

In Python, time zone information comes from the third-party `pytz` library (installable with `pip` or `conda`), which exposes the *Olson database*, a compilation of world time zone information. This is especially important for historical data because the daylight saving time (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 the `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/Pacific', 'UTC']
```

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 STD>
```

Methods in `pandas` will accept either time zone names or these objects.

Time Zone Localization and Conversion

By default, time series in `pandas` are *time zone naive*. For example, 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)
```

```
In [116]: ts
```

```
Out[116]:
```

```
2012-03-09 09:30:00    -0.202469
```

```
2012-03-10 09:30:00     0.050718
```

```
2012-03-11 09:30:00     0.639869
```

```
2012-03-12 09:30:00     0.597594
```

```
2012-03-13 09:30:00    -0.797246
2012-03-14 09:30:00     0.472879
Freq: D, dtype: float64
```

The index's tz field is None:

```
In [117]: print(ts.index.tz)
None
```

Date ranges can be generated with a time zone set:

```
In [118]: pd.date_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC')
Out[118]:
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 [119]: ts
Out[119]:
2012-03-09 09:30:00    -0.202469
2012-03-10 09:30:00     0.050718
2012-03-11 09:30:00     0.639869
2012-03-12 09:30:00     0.597594
2012-03-13 09:30:00    -0.797246
2012-03-14 09:30:00     0.472879
Freq: D, dtype: float64
```

```
In [120]: ts_utc = ts.tz_localize('UTC')
```

```
In [121]: ts_utc
Out[121]:
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 [122]: ts_utc.index
Out[122]:
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'],
              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 with `tz_convert`:

```
In [123]: ts_utc.tz_convert('America/New_York')
Out[123]:
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 preceding 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 [124]: ts_eastern = ts.tz_localize('America/New_York')
```

```
In [125]: ts_eastern.tz_convert('UTC')
Out[125]:
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 [126]: ts_eastern.tz_convert('Europe/Berlin')
Out[126]:
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 [127]: ts.index.tz_localize('Asia/Shanghai')
Out[127]:
DatetimeIndex(['2012-03-09 09:30:00+08:00', '2012-03-10 09:30:00+08:00',
              '2012-03-11 09:30:00+08:00', '2012-03-12 09:30:00+08:00',
              '2012-03-13 09:30:00+08:00', '2012-03-14 09:30:00+08:00'],
              dtype='datetime64[ns, Asia/Shanghai]', freq='D')
```



Localizing naive timestamps also checks for ambiguous or non-existent times around daylight saving 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 [128]: stamp = pd.Timestamp('2011-03-12 04:00')
In [129]: stamp_utc = stamp.tz_localize('utc')
In [130]: stamp_utc.tz_convert('America/New_York')
Out[130]: Timestamp('2011-03-11 23:00:00-0500', tz='America/New_York')
```

You can also pass a time zone when creating the `Timestamp`:

```
In [131]: stamp_moscow = pd.Timestamp('2011-03-12 04:00', tz='Europe/Moscow')
In [132]: stamp_moscow
Out[132]: Timestamp('2011-03-12 04:00:00+0300', tz='Europe/Moscow')
```

Time zone–aware `Timestamp` objects internally store a UTC timestamp value as nano-seconds since the Unix epoch (January 1, 1970); this UTC value is invariant between time zone conversions:

```
In [133]: stamp_utc.value
Out[133]: 1299902400000000000

In [134]: stamp_utc.tz_convert('America/New_York').value
Out[134]: 1299902400000000000
```

When performing time arithmetic using pandas's `DateOffset` objects, pandas respects daylight saving time transitions where possible. Here we construct time-stamps that occur right before DST transitions (forward and backward). First, 30 minutes before transitioning to DST:

```
In [135]: from pandas.tseries.offsets import Hour
In [136]: stamp = pd.Timestamp('2012-03-12 01:30', tz='US/Eastern')
In [137]: stamp
Out[137]: Timestamp('2012-03-12 01:30:00-0400', tz='US/Eastern')
In [138]: stamp + Hour()
Out[138]: Timestamp('2012-03-12 02:30:00-0400', tz='US/Eastern')
```

Then, 90 minutes before transitioning out of DST:

```
In [139]: stamp = pd.Timestamp('2012-11-04 00:30', tz='US/Eastern')
In [140]: stamp
Out[140]: Timestamp('2012-11-04 00:30:00-0400', tz='US/Eastern')
```

```
In [141]: stamp + 2 * Hour()
Out[141]: 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 [142]: rng = pd.date_range('3/7/2012 9:30', periods=10, freq='B')
```

```
In [143]: ts = pd.Series(np.random.randn(len(rng)), index=rng)
```

```
In [144]: ts
Out[144]:
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 [145]: ts1 = ts[:7].tz_localize('Europe/London')
```

```
In [146]: ts2 = ts1[2:].tz_convert('Europe/Moscow')
```

```
In [147]: result = ts1 + ts2
```

```
In [148]: result.index
```

```
Out[148]:
DatetimeIndex(['2012-03-07 09:30:00+00:00', '2012-03-08 09:30:00+00:00',
               '2012-03-09 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'],
              dtype='datetime64[ns, UTC]', freq='B')
```

11.5 Periods and Period Arithmetic

Periods represent timespans, like days, months, quarters, or years. The `Period` class represents this data type, requiring a string or integer and a frequency from Table 11-4:

```
In [149]: p = pd.Period(2007, freq='A-DEC')
```

```
In [150]: p
Out[150]: 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 [151]: p + 5
Out[151]: Period('2012', 'A-DEC')
```

```
In [152]: p - 2
Out[152]: Period('2005', 'A-DEC')
```

If two periods have the same frequency, their difference is the number of units between them:

```
In [153]: pd.Period('2014', freq='A-DEC') - p
Out[153]: 7
```

Regular ranges of periods can be constructed with the `period_range` function:

```
In [154]: rng = pd.period_range('2000-01-01', '2000-06-30', freq='M')
```

```
In [155]: rng
Out[155]: PeriodIndex(['2000-01', '2000-02', '2000-03', '2000-04', '2000-05', '2000-06'], dtype='period[M]', freq='M')
```

The `PeriodIndex` class stores a sequence of periods and can serve as an axis index in any pandas data structure:

```
In [156]: pd.Series(np.random.randn(6), index=rng)
Out[156]:
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 use the `PeriodIndex` class:

```
In [157]: values = ['2001Q3', '2002Q2', '2003Q1']
```

```
In [158]: index = pd.PeriodIndex(values, freq='Q-DEC')
```

```
In [159]: index
Out[159]: PeriodIndex(['2001Q3', '2002Q2', '2003Q1'], dtype='period[Q-DEC]', freq='Q-DEC')
```

Period Frequency Conversion

Periods and `PeriodIndex` objects can be converted to another frequency with 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 [160]: p = pd.Period('2007', freq='A-DEC')
```

```
In [161]: p  
Out[161]: Period('2007', 'A-DEC')
```

```
In [162]: p.asfreq('M', how='start')  
Out[162]: Period('2007-01', 'M')
```

```
In [163]: p.asfreq('M', how='end')  
Out[163]: 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 corresponding monthly subperiods are different:

```
In [164]: p = pd.Period('2007', freq='A-JUN')
```

```
In [165]: p  
Out[165]: Period('2007', 'A-JUN')
```

```
In [166]: p.asfreq('M', 'start')  
Out[166]: Period('2006-07', 'M')
```

```
In [167]: p.asfreq('M', 'end')  
Out[167]: Period('2007-06', 'M')
```

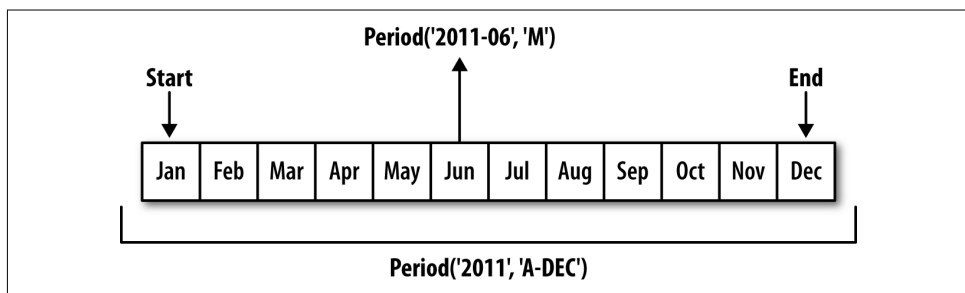


Figure 11-1. Period frequency conversion illustration

When you are converting from high to low frequency, pandas determines the super-period depending on where the subperiod “belongs.” For example, in `A-JUN` frequency, the month `Aug-2007` is actually part of the `2008` period:

```
In [168]: p = pd.Period('Aug-2007', 'M')
```

```
In [169]: p.asfreq('A-JUN')  
Out[169]: Period('2008', 'A-JUN')
```

Whole `PeriodIndex` objects or time series can be similarly converted with the same semantics:

```
In [170]: rng = pd.period_range('2006', '2009', freq='A-DEC')
In [171]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [172]: ts
Out[172]:
2006    1.607578
2007    0.200381
2008   -0.834068
2009   -0.302988
Freq: A-DEC, dtype: float64
```

```
In [173]: ts.asfreq('M', how='start')
Out[173]:
2006-01    1.607578
2007-01    0.200381
2008-01   -0.834068
2009-01   -0.302988
Freq: M, dtype: float64
```

Here, the annual periods are replaced with monthly periods corresponding to the first month falling within each annual period. If we instead wanted the last business day of each year, we can use the 'B' frequency and indicate that we want the end of the period:

```
In [174]: ts.asfreq('B', how='end')
Out[174]:
2006-12-29    1.607578
2007-12-31    0.200381
2008-12-31   -0.834068
2009-12-31   -0.302988
Freq: B, dtype: float64
```

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. Thus, 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 [175]: p = pd.Period('2012Q4', freq='Q-JAN')

In [176]: p
Out[176]: 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.

Year 2012												
M	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
Q-DEC	2012Q1			2012Q2			2012Q3			2012Q4		
Q-SEP	2012Q2			2012Q3			2012Q4			2013Q1		
Q-FEB	2012Q4		2013Q1			2013Q2			2013Q3		Q4	

Figure 11-2. Different quarterly frequency conventions

```
In [177]: p.asfreq('D', 'start')
Out[177]: Period('2011-11-01', 'D')
```

```
In [178]: p.asfreq('D', 'end')
Out[178]: Period('2012-01-31', 'D')
```

Thus, it's possible to do easy period arithmetic; for example, to get the timestamp at 4 PM on the second-to-last business day of the quarter, you could do:

```
In [179]: p4pm = (p.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60
```

```
In [180]: p4pm
Out[180]: Period('2012-01-30 16:00', 'T')
```

```
In [181]: p4pm.to_timestamp()
Out[181]: Timestamp('2012-01-30 16:00:00')
```

You can generate quarterly ranges using `period_range`. Arithmetic is identical, too:

```
In [182]: rng = pd.period_range('2011Q3', '2012Q4', freq='Q-JAN')
```

```
In [183]: ts = pd.Series(np.arange(len(rng)), index=rng)
```

```
In [184]: ts
Out[184]:
2011Q3    0
2011Q4    1
2012Q1    2
2012Q2    3
2012Q3    4
2012Q4    5
Freq: Q-JAN, dtype: int64
```

```
In [185]: new_rng = (rng.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60
```

```
In [186]: ts.index = new_rng.to_timestamp()
```

```
In [187]: ts
Out[187]:
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 with the `to_period` method:

```
In [188]: rng = pd.date_range('2000-01-01', periods=3, freq='M')
```

```
In [189]: ts = pd.Series(np.random.randn(3), index=rng)
```

```
In [190]: ts
Out[190]:
2000-01-31    1.663261
2000-02-29   -0.996206
2000-03-31    1.521760
Freq: M, dtype: float64
```

```
In [191]: pts = ts.to_period()
```

```
In [192]: pts
Out[192]:
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 [193]: rng = pd.date_range('1/29/2000', periods=6, freq='D')
```

```
In [194]: ts2 = pd.Series(np.random.randn(6), index=rng)
```

```
In [195]: ts2
Out[195]:
2000-01-29    0.244175
2000-01-30    0.423331
2000-01-31   -0.654040
2000-02-01    2.089154
2000-02-02   -0.060220
```

```
2000-02-03    -0.167933
Freq: D, dtype: float64
```

```
In [196]: ts2.to_period('M')
Out[196]:
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 [197]: pts = ts2.to_period()
```

```
In [198]: pts
Out[198]:
2000-01-29     0.244175
2000-01-30     0.423331
2000-01-31    -0.654040
2000-02-01     2.089154
2000-02-02    -0.060220
2000-02-03    -0.167933
Freq: D, dtype: float64
```

```
In [199]: pts.to_timestamp(how='end')
Out[199]:
2000-01-29     0.244175
2000-01-30     0.423331
2000-01-31    -0.654040
2000-02-01     2.089154
2000-02-02    -0.060220
2000-02-03    -0.167933
Freq: D, dtype: float64
```

Creating a PeriodIndex from Arrays

Fixed frequency datasets are sometimes stored with timespan information spread across multiple columns. For example, in this macroeconomic dataset, the year and quarter are in different columns:

```
In [200]: data = pd.read_csv('examples/macrodta.csv')
```

```
In [201]: data.head(5)
Out[201]:
```

	year	quarter	realgdp	realcons	realinv	realgovt	realdpi	cpi	\
0	1959.0	1.0	2710.349	1707.4	286.898	470.045	1886.9	28.98	
1	1959.0	2.0	2778.801	1733.7	310.859	481.301	1919.7	29.15	
2	1959.0	3.0	2775.488	1751.8	289.226	491.260	1916.4	29.35	
3	1959.0	4.0	2785.204	1753.7	299.356	484.052	1931.3	29.37	

```
4  1960.0      1.0  2847.699    1770.5  331.722  462.199  1955.5  29.54
   m1  tbilrate  unemp      pop  infl  realint
0  139.7      2.82   5.8  177.146  0.00   0.00
1  141.7      3.08   5.1  177.830  2.34   0.74
2  140.5      3.82   5.3  178.657  2.74   1.09
3  140.0      4.33   5.6  179.386  0.27   4.06
4  139.6      3.50   5.2  180.007  2.31   1.19
```

```
In [202]: data.year
```

```
Out[202]:
```

```
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 [203]: data.quarter
```

```
Out[203]:
```

```
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, you can combine them to form an index for the `DataFrame`:

```
In [204]: index = pd.PeriodIndex(year=data.year, quarter=data.quarter,
.....:                          freq='Q-DEC')
```

```
In [205]: index
```

```
Out[205]:
```

```
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960Q2',
            '1960Q3', '1960Q4', '1961Q1', '1961Q2',
            ...
            '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',
            '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', length=203, freq='Q-DEC')
```

```
In [206]: data.index = index
```

```
In [207]: data.infl
```

```
Out[207]:
```

```
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
```

11.6 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 [208]: rng = pd.date_range('2000-01-01', periods=100, freq='D')
```

```
In [209]: ts = pd.Series(np.random.randn(len(rng)), index=rng)
```

```
In [210]: ts
```

```
Out[210]:
```

```
2000-01-01    0.631634
2000-01-02   -1.594313
2000-01-03   -1.519937
2000-01-04    1.108752
2000-01-05    1.255853
2000-01-06   -0.024330
2000-01-07   -2.047939
2000-01-08   -0.272657
2000-01-09   -1.692615
2000-01-10    1.423830
```

```
...
```

```
2000-03-31   -0.007852
2000-04-01   -1.638806
2000-04-02    1.401227
2000-04-03    1.758539
2000-04-04    0.628932
2000-04-05   -0.423776
2000-04-06    0.789740
2000-04-07    0.937568
2000-04-08   -2.253294
2000-04-09   -1.772919
```

```
Freq: D, Length: 100, dtype: float64
```

```
In [211]: ts.resample('M').mean()
```

```
Out[211]:
```

```
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 [212]: ts.resample('M', kind='period').mean()
```

```
Out[212]:
```

```
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. The examples in the following sections illustrate its semantics and use. [Table 11-5](#) summarizes some of its options.

Table 11-5. Resample method arguments

Argument	Description
<code>freq</code>	String or <code>DateOffset</code> indicating desired resampled frequency (e.g., 'M', '5min', or <code>Second(15)</code>)
<code>axis</code>	Axis to resample on; default <code>axis=0</code>
<code>fill_method</code>	How to interpolate when upsampling, as in 'ffill' or 'bfill'; by default does no interpolation
<code>closed</code>	In downsampling, which end of each interval is closed (inclusive), 'right' or 'left'
<code>label</code>	In downsampling, how to label the aggregated result, with the 'right' or 'left' bin edge (e.g., the 9:30 to 9:35 five-minute interval could be labeled 9:30 or 9:35)
<code>loffset</code>	Time adjustment to the bin labels, such as '-1s' / <code>Second(-1)</code> to shift the aggregate labels one second earlier
<code>limit</code>	When forward or backward filling, the maximum number of periods to fill
<code>kind</code>	Aggregate to periods ('period') or timestamps ('timestamp'); defaults to the type of index the time series has
<code>convention</code>	When resampling periods, the convention ('start' or 'end') for converting the low-frequency period to high frequency; defaults to 'end'

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', you need to chop up the data 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 [213]: rng = pd.date_range('2000-01-01', periods=12, freq='T')
```

```
In [214]: ts = pd.Series(np.arange(12), index=rng)
```

```
In [215]: ts
Out[215]:
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 [216]: ts.resample('5min', closed='right').sum()
Out[216]:
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.¹ Passing `closed='right'` changes the interval to be closed on the right:

```
In [217]: ts.resample('5min', closed='right').sum()
Out[217]:
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 [218]: ts.resample('5min', closed='right', label='right').sum()
Out[218]:
2000-01-01 00:00:00    0
2000-01-01 00:05:00   15
```

¹ 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.


```

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 minute frequency data being resampled to five-minute frequency.

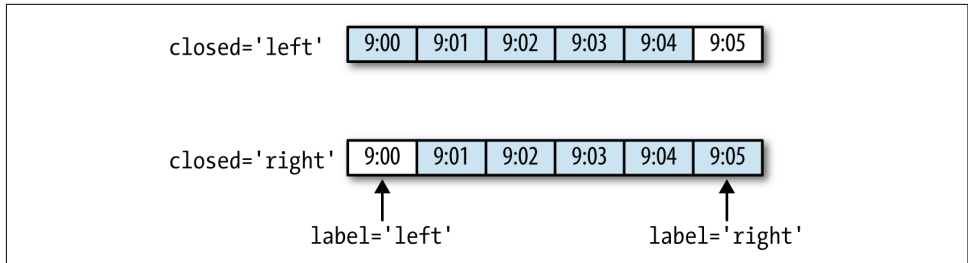


Figure 11-3. Five-minute resampling illustration of closed, label conventions

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 [219]: ts.resample('5min', closed='right',
.....:                label='right', loffset='-1s').sum()
Out[219]:
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

```

You also could have accomplished the effect of `loffset` 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 `ohlcv` 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 [220]: ts.resample('5min').ohlcv()
Out[220]:
              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 [221]: frame = pd.DataFrame(np.random.randn(2, 4),
.....:                          index=pd.date_range('1/1/2000', periods=2,
.....:                                               freq='W-WED'),
.....:                          columns=['Colorado', 'Texas', 'New York', 'Ohio'])
```

```
In [222]: frame
```

```
Out[222]:
```

	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 you are using an aggregation function with this data, there is only one value per group, and missing values result in the gaps. We use the `asfreq` method to convert to the higher frequency without any aggregation:

```
In [223]: df_daily = frame.resample('D').asfreq()
```

```
In [224]: df_daily
```

```
Out[224]:
```

	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 [225]: frame.resample('D').ffill()
```

```
Out[225]:
```

	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 [226]: frame.resample('D').ffill(limit=2)
Out[226]:
```

	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 [227]: frame.resample('W-THU').ffill()
Out[227]:
```

	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 [228]: frame = pd.DataFrame(np.random.randn(24, 4),
.....:                          index=pd.period_range('1-2000', '12-2001',
.....:                                                  freq='M'),
.....:                          columns=['Colorado', 'Texas', 'New York', 'Ohio'])
```

```
In [229]: frame[:5]
Out[229]:
```

	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 [230]: annual_frame = frame.resample('A-DEC').mean()
```

```
In [231]: annual_frame
Out[231]:
```

	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 'start' but can also be 'end':

```
# Q-DEC: Quarterly, year ending in December
In [232]: annual_frame.resample('Q-DEC').ffill()
Out[232]:
```

	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 [233]: annual_frame.resample('Q-DEC', convention='end').ffill()
Out[233]:
```

	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

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 [234]: annual_frame.resample('Q-MAR').ffill()
Out[234]:
```

	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

11.7 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. This can be useful for smoothing noisy or gappy data. 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.

Before digging in, we can load up some time series data and resample it to business day frequency:

```
In [235]: close_px_all = pd.read_csv('examples/stock_px_2.csv',
.....:                               parse_dates=True, index_col=0)

In [236]: close_px = close_px_all[['AAPL', 'MSFT', 'XOM']]

In [237]: close_px = close_px.resample('B').ffill()
```

I now 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; see [Figure 11-4](#) for the plot created):

```
In [238]: close_px.AAPL.plot()
Out[238]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f2570cf98>

In [239]: close_px.AAPL.rolling(250).mean().plot()
```

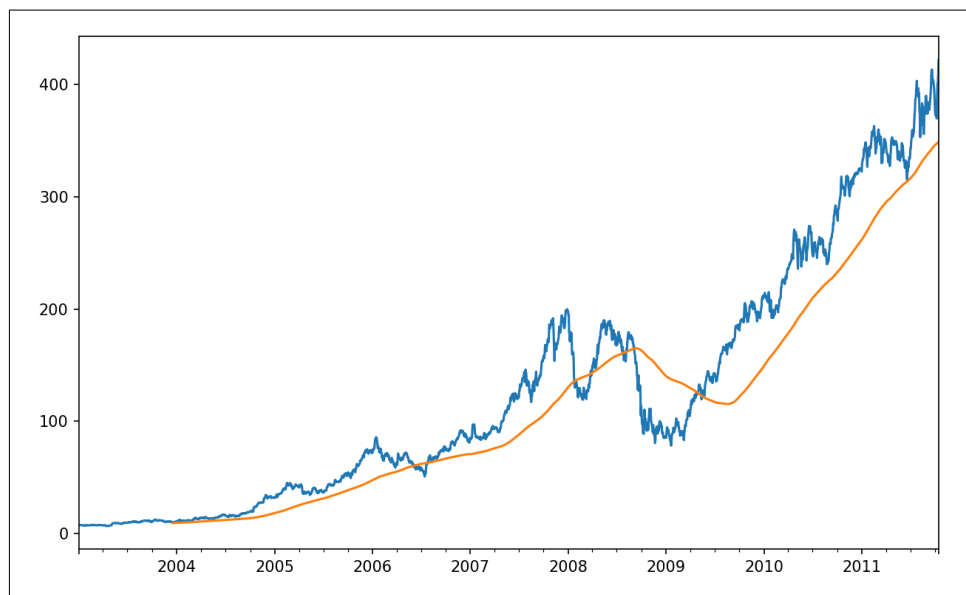


Figure 11-4. Apple Price with 250-day MA

The expression `rolling(250)` is similar in behavior to `groupby`, but instead of grouping it creates an object that enables grouping over a 250-day sliding window. So here we have the 250-day moving window average of Apple's stock price.

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 [241]: appl_std250 = close_px.AAPL.rolling(250, min_periods=10).std()
```

```
In [242]: appl_std250[5:12]
```

```
Out[242]:
```

```
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 [243]: appl_std250.plot()
```

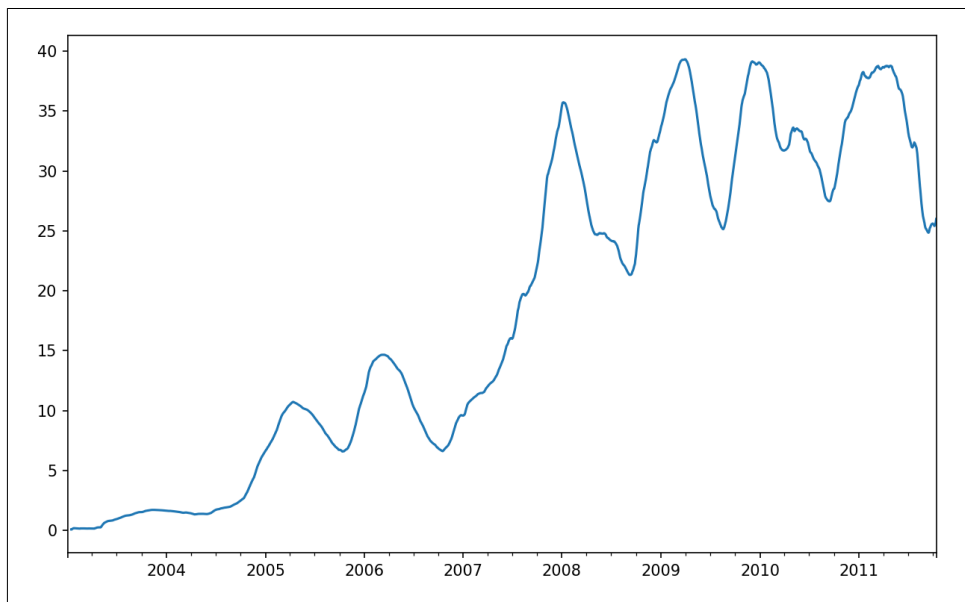


Figure 11-5. Apple 250-day daily return standard deviation

In order to compute an *expanding window mean*, use the expanding operator instead of rolling. The expanding mean starts the time window from the beginning of the time series and increases the size of the window until it encompasses the whole series. An expanding window mean on the `appl_std250` time series looks like this:

```
In [244]: expanding_mean = appl_std250.expanding().mean()
```

Calling a moving window function on a DataFrame applies the transformation to each column (see [Figure 11-6](#)):

```
In [246]: close_px.rolling(60).mean().plot(log=True)
```

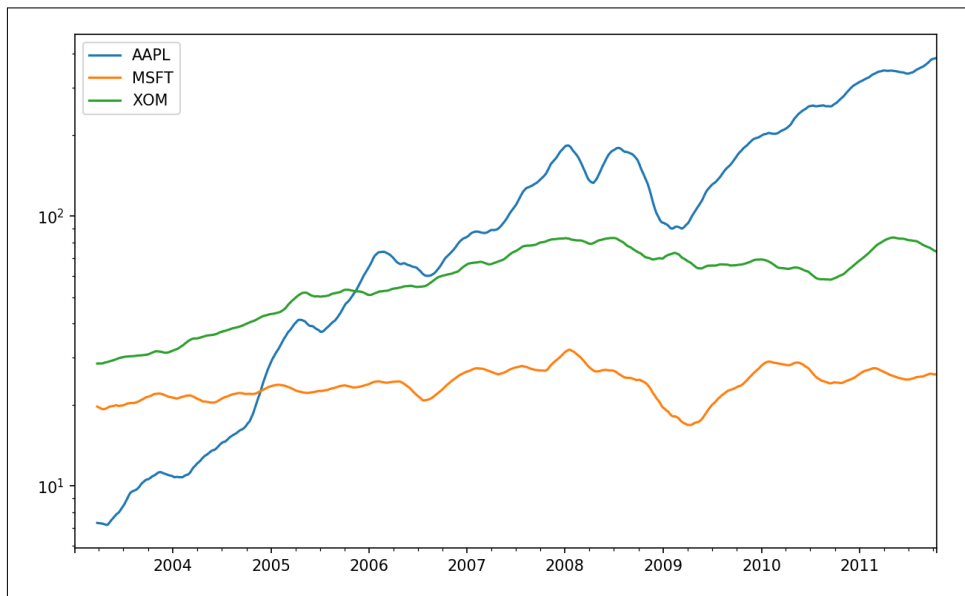


Figure 11-6. Stocks prices 60-day MA (log Y-axis)

The `rolling` function also accepts a string indicating a fixed-size time offset rather than a set number of periods. Using this notation can be useful for irregular time series. These are the same strings that you can pass to `resample`. For example, we could compute a 20-day rolling mean like so:

```
In [247]: close_px.rolling('20D').mean()
```

```
Out[247]:
```

	AAPL	MSFT	XOM
2003-01-02	7.400000	21.110000	29.220000
2003-01-03	7.425000	21.125000	29.230000
2003-01-06	7.433333	21.256667	29.473333
2003-01-07	7.432500	21.425000	29.342500
2003-01-08	7.402000	21.402000	29.240000
2003-01-09	7.391667	21.490000	29.273333
2003-01-10	7.387143	21.558571	29.238571
2003-01-13	7.378750	21.633750	29.197500
2003-01-14	7.370000	21.717778	29.194444
2003-01-15	7.355000	21.757000	29.152000
...
2011-10-03	398.002143	25.890714	72.413571
2011-10-04	396.802143	25.807857	72.427143
2011-10-05	395.751429	25.729286	72.422857

```
2011-10-06 394.099286 25.673571 72.375714
2011-10-07 392.479333 25.712000 72.454667
2011-10-10 389.351429 25.602143 72.527857
2011-10-11 388.505000 25.674286 72.835000
2011-10-12 388.531429 25.810000 73.400714
2011-10-13 388.826429 25.961429 73.905000
2011-10-14 391.038000 26.048667 74.185333
[2292 rows x 3 columns]
```

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. 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 [249]: aapl_px = close_px.AAPL['2006':'2007']

In [250]: ma60 = aapl_px.rolling(30, min_periods=20).mean()

In [251]: ewma60 = aapl_px.ewm(span=30).mean()

In [252]: ma60.plot(style='k--', label='Simple MA')
Out[252]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f252161d0>

In [253]: ewma60.plot(style='k-', label='EW MA')
Out[253]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2f252161d0>

In [254]: plt.legend()
```

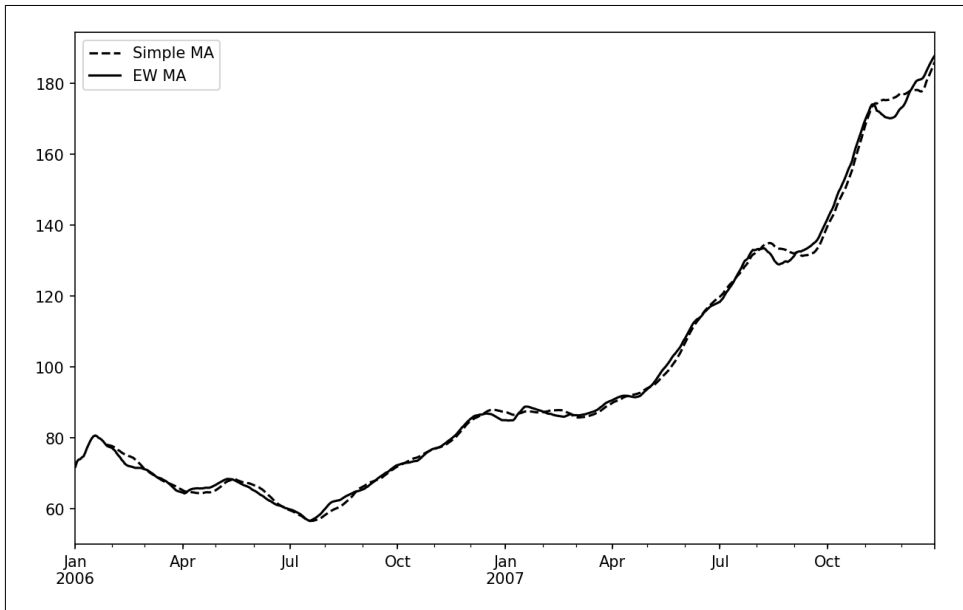



Figure 11-7. Simple moving average versus exponentially weighted

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. To have a look at this, we first compute the percent change for all of our time series of interest:

```
In [256]: spx_px = close_px_all['SPX']
```

```
In [257]: spx_rets = spx_px.pct_change()
```

```
In [258]: returns = close_px.pct_change()
```

The `corr` aggregation function after we call `rolling` can then compute the rolling correlation with `spx_rets` (see [Figure 11-8](#) for the resulting plot):

```
In [259]: corr = returns.AAPL.rolling(125, min_periods=100).corr(spx_rets)
```

```
In [260]: corr.plot()
```

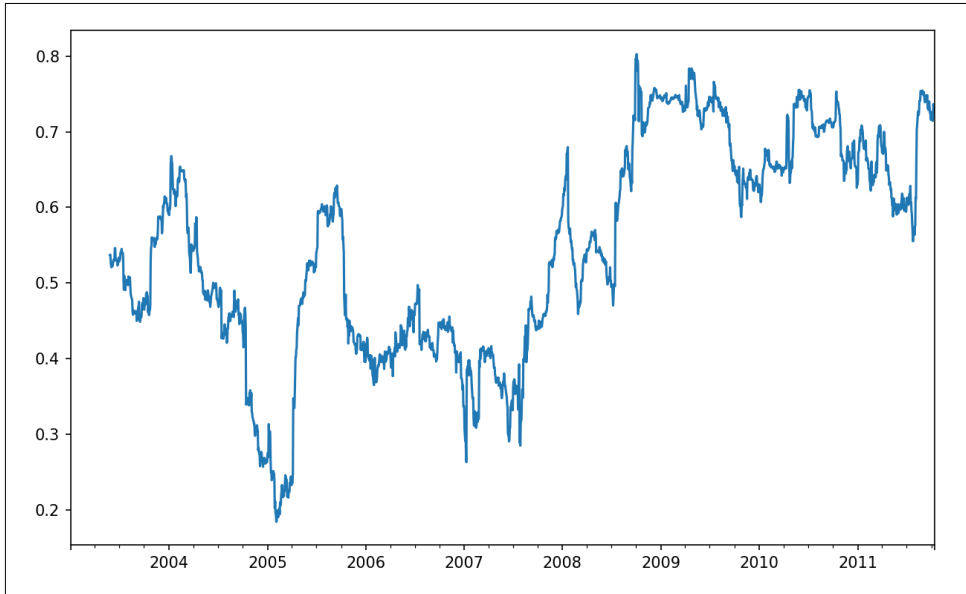


Figure 11-8. Six-month AAPL return correlation to S&P 500

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 might get repetitive, so if you pass a Series and a DataFrame, a function like `rolling_corr` will compute the correlation of the Series (`spx_rets`, in this case) with each column in the DataFrame (see Figure 11-9 for the plot of the result):

```
In [262]: corr = returns.rolling(125, min_periods=100).corr(spx_rets)
```

```
In [263]: corr.plot()
```

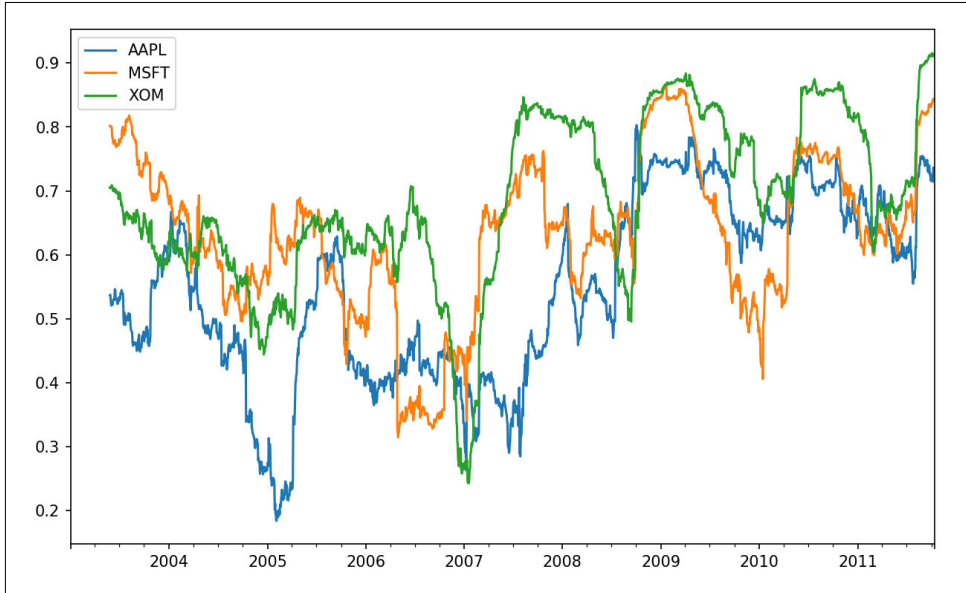


Figure 11-9. Six-month return correlations to S&P 500

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 (see Figure 11-10 for the resulting plot):

```
In [265]: from scipy.stats import percentileofscore
In [266]: score_at_2percent = lambda x: percentileofscore(x, 0.02)
In [267]: result = returns.AAPL.rolling(250).apply(score_at_2percent)
In [268]: result.plot()
```

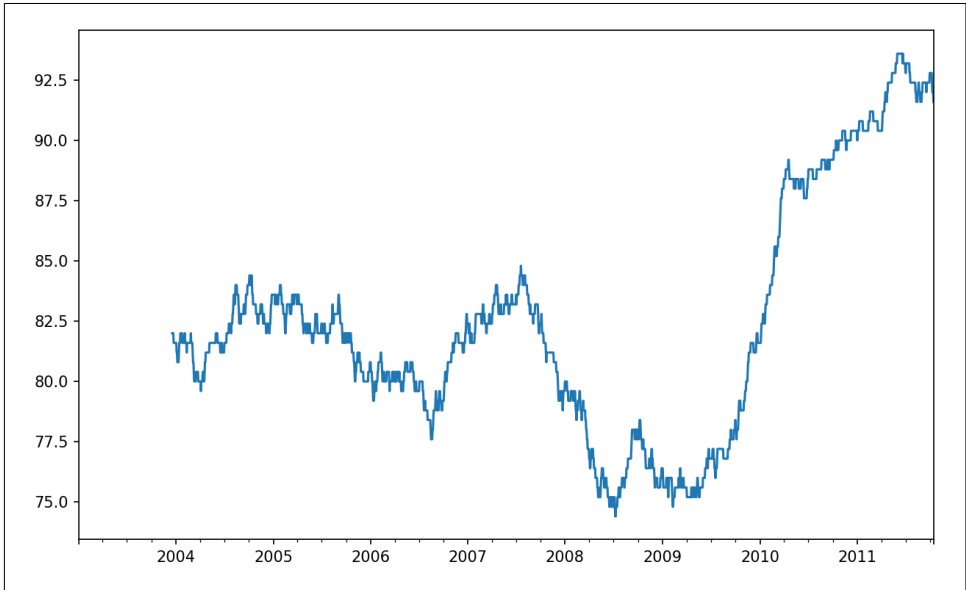


Figure 11-10. Percentile rank of 2% AAPL return over one-year window

If you don't have SciPy installed already, you can install it with conda or pip.

11.8 Conclusion

Time series data calls for different types of analysis and data transformation tools than the other types of data we have explored in previous chapters.

In the following chapters, we will move on to some advanced pandas methods and show how to start using modeling libraries like statsmodels and scikit-learn.

Advanced pandas

The preceding chapters have focused on introducing different types of data wrangling workflows and features of NumPy, pandas, and other libraries. Over time, pandas has developed a depth of features for power users. This chapter digs into a few more advanced feature areas to help you deepen your expertise as a pandas user.

12.1 Categorical Data

This section introduces the pandas `Categorical` type. I will show how you can achieve better performance and memory use in some pandas operations by using it. I also introduce some tools for using categorical data in statistics and machine learning applications.

Background and Motivation

Frequently, a column in a table may contain repeated instances of a smaller set of distinct values. We have already seen functions like `unique` and `value_counts`, which enable us to extract the distinct values from an array and compute their frequencies, respectively:

```
In [10]: import numpy as np; import pandas as pd

In [11]: values = pd.Series(['apple', 'orange', 'apple',
    ....:                    'apple'] * 2)

In [12]: values
Out[12]:
0    apple
1    orange
2    apple
3    apple
```

```
4 apple
5 orange
6 apple
7 apple
dtype: object
```

```
In [13]: pd.unique(values)
Out[13]: array(['apple', 'orange'], dtype=object)
```

```
In [14]: pd.value_counts(values)
Out[14]:
apple    6
orange   2
dtype: int64
```

Many data systems (for data warehousing, statistical computing, or other uses) have developed specialized approaches for representing data with repeated values for more efficient storage and computation. In data warehousing, a best practice is to use so-called *dimension tables* containing the distinct values and storing the primary observations as integer keys referencing the dimension table:

```
In [15]: values = pd.Series([0, 1, 0, 0] * 2)
```

```
In [16]: dim = pd.Series(['apple', 'orange'])
```

```
In [17]: values
Out[17]:
0    0
1    1
2    0
3    0
4    0
5    1
6    0
7    0
dtype: int64
```

```
In [18]: dim
Out[18]:
0    apple
1    orange
dtype: object
```

We can use the `take` method to restore the original Series of strings:

```
In [19]: dim.take(values)
Out[19]:
0    apple
1    orange
0    apple
0    apple
0    apple
1    orange
```

```
0 apple
0 apple
dtype: object
```

This representation as integers is called the *categorical* or *dictionary-encoded* representation. The array of distinct values can be called the *categories*, *dictionary*, or *levels* of the data. In this book we will use the terms *categorical* and *categories*. The integer values that reference the categories are called the *category codes* or simply *codes*.

The categorical representation can yield significant performance improvements when you are doing analytics. You can also perform transformations on the categories while leaving the codes unmodified. Some example transformations that can be made at relatively low cost are:

- Renaming categories
- Appending a new category without changing the order or position of the existing categories

Categorical Type in pandas

pandas has a special `Categorical` type for holding data that uses the integer-based categorical representation or *encoding*. Let's consider the example Series from before:

```
In [20]: fruits = ['apple', 'orange', 'apple', 'apple'] * 2
```

```
In [21]: N = len(fruits)
```

```
In [22]: df = pd.DataFrame({'fruit': fruits,
.....:                      'basket_id': np.arange(N),
.....:                      'count': np.random.randint(3, 15, size=N),
.....:                      'weight': np.random.uniform(0, 4, size=N)},
.....:                      columns=['basket_id', 'fruit', 'count', 'weight'])
```

```
In [23]: df
```

```
Out[23]:
```

	basket_id	fruit	count	weight
0	0	apple	5	3.858058
1	1	orange	8	2.612708
2	2	apple	4	2.995627
3	3	apple	7	2.614279
4	4	apple	12	2.990859
5	5	orange	8	3.845227
6	6	apple	5	0.033553
7	7	apple	4	0.425778

Here, `df['fruit']` is an array of Python string objects. We can convert it to categorical by calling:

```
In [24]: fruit_cat = df['fruit'].astype('category')
```

```
In [25]: fruit_cat
```

```
Out[25]:
```

```
0    apple
1    orange
2    apple
3    apple
4    apple
5    orange
6    apple
7    apple
```

```
Name: fruit, dtype: category
```

```
Categories (2, object): [apple, orange]
```

The values for `fruit_cat` are not a NumPy array, but an instance of `pandas.Categorical`:

```
In [26]: c = fruit_cat.values
```

```
In [27]: type(c)
```

```
Out[27]: pandas.core.categorical.Categorical
```

The `Categorical` object has `categories` and `codes` attributes:

```
In [28]: c.categories
```

```
Out[28]: Index(['apple', 'orange'], dtype='object')
```

```
In [29]: c.codes
```

```
Out[29]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

You can convert a DataFrame column to categorical by assigning the converted result:

```
In [30]: df['fruit'] = df['fruit'].astype('category')
```

```
In [31]: df.fruit
```

```
Out[31]:
```

```
0    apple
1    orange
2    apple
3    apple
4    apple
5    orange
6    apple
7    apple
```

```
Name: fruit, dtype: category
```

```
Categories (2, object): [apple, orange]
```

You can also create `pandas.Categorical` directly from other types of Python sequences:

```
In [32]: my_categories = pd.Categorical(['foo', 'bar', 'baz', 'foo', 'bar'])
```

```
In [33]: my_categories
```



```
Out[33]:  
[foo, bar, baz, foo, bar]  
Categories (3, object): [bar, baz, foo]
```

If you have obtained categorical encoded data from another source, you can use the alternative `from_codes` constructor:

```
In [34]: categories = ['foo', 'bar', 'baz']  
  
In [35]: codes = [0, 1, 2, 0, 0, 1]  
  
In [36]: my_cats_2 = pd.Categorical.from_codes(codes, categories)  
  
In [37]: my_cats_2  
Out[37]:  
[foo, bar, baz, foo, foo, bar]  
Categories (3, object): [foo, bar, baz]
```

Unless explicitly specified, categorical conversions assume no specific ordering of the categories. So the `categories` array may be in a different order depending on the ordering of the input data. When using `from_codes` or any of the other constructors, you can indicate that the categories have a meaningful ordering:

```
In [38]: ordered_cat = pd.Categorical.from_codes(codes, categories,  
.....:                                         ordered=True)  
  
In [39]: ordered_cat  
Out[39]:  
[foo, bar, baz, foo, foo, bar]  
Categories (3, object): [foo < bar < baz]
```

The output `[foo < bar < baz]` indicates that 'foo' precedes 'bar' in the ordering, and so on. An unordered categorical instance can be made ordered with `as_ordered`:

```
In [40]: my_cats_2.as_ordered()  
Out[40]:  
[foo, bar, baz, foo, foo, bar]  
Categories (3, object): [foo < bar < baz]
```

As a last note, categorical data need not be strings, even though I have only showed string examples. A categorical array can consist of any immutable value types.

Computations with Categoricals

Using `Categorical` in pandas compared with the non-encoded version (like an array of strings) generally behaves the same way. Some parts of pandas, like the `groupby` function, perform better when working with categorical. There are also some functions that can utilize the `ordered` flag.

Let's consider some random numeric data, and use the `pandas.qcut` binning function. This returns `pandas.Categorical`; we used `pandas.cut` earlier in the book but glossed over the details of how categoricals work:

```
In [41]: np.random.seed(12345)

In [42]: draws = np.random.randn(1000)

In [43]: draws[:5]
Out[43]: array([-0.2047,  0.4789, -0.5194, -0.5557,  1.9658])
```

Let's compute a quartile binning of this data and extract some statistics:

```
In [44]: bins = pd.qcut(draws, 4)

In [45]: bins
Out[45]:
[(-0.684, -0.0101], (-0.0101, 0.63], (-0.684, -0.0101], (-0.684, -0.0101], (0.63,
 3.928], ..., (-0.0101, 0.63], (-0.684, -0.0101], (-2.95, -0.684], (-0.0101, 0.63
 ], (0.63, 3.928]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.684] < (-0.684, -0.0101] < (-0.0101, 0.63] <
(0.63, 3.928]]
```

While useful, the exact sample quartiles may be less useful for producing a report than quartile names. We can achieve this with the `labels` argument to `qcut`:

```
In [46]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3', 'Q4'])

In [47]: bins
Out[47]:
[Q2, Q3, Q2, Q2, Q4, ..., Q3, Q2, Q1, Q3, Q4]
Length: 1000
Categories (4, object): [Q1 < Q2 < Q3 < Q4]

In [48]: bins.codes[:10]
Out[48]: array([1, 2, 1, 1, 3, 3, 2, 2, 3, 3], dtype=int8)
```

The labeled bins categorical does not contain information about the bin edges in the data, so we can use `groupby` to extract some summary statistics:

```
In [49]: bins = pd.Series(bins, name='quartile')

In [50]: results = (pd.Series(draws)
....:                 .groupby(bins)
....:                 .agg(['count', 'min', 'max'])
....:                 .reset_index())

In [51]: results
Out[51]:
   quartile  count      min      max
0         Q1    250 -2.949343 -0.685484
```

```
1      Q2      250 -0.683066 -0.010115
2      Q3      250 -0.010032  0.628894
3      Q4      250  0.634238  3.927528
```

The 'quartile' column in the result retains the original categorical information, including ordering, from bins:

```
In [52]: results['quartile']
Out[52]:
0      Q1
1      Q2
2      Q3
3      Q4
Name: quartile, dtype: category
Categories (4, object): [Q1 < Q2 < Q3 < Q4]
```

Better performance with categoricals

If you do a lot of analytics on a particular dataset, converting to categorical can yield substantial overall performance gains. A categorical version of a DataFrame column will often use significantly less memory, too. Let's consider some Series with 10 million elements and a small number of distinct categories:

```
In [53]: N = 10000000
In [54]: draws = pd.Series(np.random.randn(N))
In [55]: labels = pd.Series(['foo', 'bar', 'baz', 'qux'] * (N // 4))
```

Now we convert labels to categorical:

```
In [56]: categories = labels.astype('category')
```

Now we note that labels uses significantly more memory than categories:

```
In [57]: labels.memory_usage()
Out[57]: 80000080
In [58]: categories.memory_usage()
Out[58]: 10000272
```

The conversion to category is not free, of course, but it is a one-time cost:

```
In [59]: %time _ = labels.astype('category')
CPU times: user 490 ms, sys: 240 ms, total: 730 ms
Wall time: 726 ms
```

GroupBy operations can be significantly faster with categoricals because the underlying algorithms use the integer-based codes array instead of an array of strings.

Categorical Methods

Series containing categorical data have several special methods similar to the `Series.str` specialized string methods. This also provides convenient access to the categories and codes. Consider the Series:

```
In [60]: s = pd.Series(['a', 'b', 'c', 'd'] * 2)
```

```
In [61]: cat_s = s.astype('category')
```

```
In [62]: cat_s
```

```
Out[62]:
```

```
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (4, object): [a, b, c, d]
```

The special attribute `cat` provides access to categorical methods:

```
In [63]: cat_s.cat.codes
```

```
Out[63]:
```

```
0    0
1    1
2    2
3    3
4    0
5    1
6    2
7    3
dtype: int8
```

```
In [64]: cat_s.cat.categories
```

```
Out[64]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

Suppose that we know the actual set of categories for this data extends beyond the four values observed in the data. We can use the `set_categories` method to change them:

```
In [65]: actual_categories = ['a', 'b', 'c', 'd', 'e']
```

```
In [66]: cat_s2 = cat_s.cat.set_categories(actual_categories)
```

```
In [67]: cat_s2
```

```
Out[67]:
```

```
0    a
1    b
```

```
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (5, object): [a, b, c, d, e]
```

While it appears that the data is unchanged, the new categories will be reflected in operations that use them. For example, `value_counts` respects the categories, if present:

```
In [68]: cat_s.value_counts()
Out[68]:
d    2
c    2
b    2
a    2
dtype: int64
```

```
In [69]: cat_s2.value_counts()
Out[69]:
d    2
c    2
b    2
a    2
e    0
dtype: int64
```

In large datasets, categoricals are often used as a convenient tool for memory savings and better performance. After you filter a large DataFrame or Series, many of the categories may not appear in the data. To help with this, we can use the `remove_unused_categories` method to trim unobserved categories:

```
In [70]: cat_s3 = cat_s[cat_s.isin(['a', 'b'])]
```

```
In [71]: cat_s3
Out[71]:
0    a
1    b
4    a
5    b
dtype: category
Categories (4, object): [a, b, c, d]
```

```
In [72]: cat_s3.cat.remove_unused_categories()
Out[72]:
0    a
1    b
4    a
5    b
```

```
dtype: category
Categories (2, object): [a, b]
```

See [Table 12-1](#) for a listing of available categorical methods.

Table 12-1. Categorical methods for Series in pandas

Method	Description
<code>add_categories</code>	Append new (unused) categories at end of existing categories
<code>as_ordered</code>	Make categories ordered
<code>as_unordered</code>	Make categories unordered
<code>remove_categories</code>	Remove categories, setting any removed values to null
<code>remove_unused_categories</code>	Remove any category values which do not appear in the data
<code>rename_categories</code>	Replace categories with indicated set of new category names; cannot change the number of categories
<code>reorder_categories</code>	Behaves like <code>rename_categories</code> , but can also change the result to have ordered categories
<code>set_categories</code>	Replace the categories with the indicated set of new categories; can add or remove categories

Creating dummy variables for modeling

When you're using statistics or machine learning tools, you'll often transform categorical data into *dummy variables*, also known as *one-hot* encoding. This involves creating a DataFrame with a column for each distinct category; these columns contain 1s for occurrences of a given category and 0 otherwise.

Consider the previous example:

```
In [73]: cat_s = pd.Series(['a', 'b', 'c', 'd'] * 2, dtype='category')
```

As mentioned previously in [Chapter 7](#), the `pandas.get_dummies` function converts this one-dimensional categorical data into a DataFrame containing the dummy variable:

```
In [74]: pd.get_dummies(cat_s)
```

```
Out[74]:
```

	a	b	c	d
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	1	0	0	0
5	0	1	0	0
6	0	0	1	0
7	0	0	0	1

12.2 Advanced GroupBy Use

While we've already discussed using the `groupby` method for `Series` and `DataFrame` in depth in [Chapter 10](#), there are some additional techniques that you may find of use.

Group Transforms and “Unwrapped” GroupBys

In [Chapter 10](#) we looked at the `apply` method in grouped operations for performing transformations. There is another built-in method called `transform`, which is similar to `apply` but imposes more constraints on the kind of function you can use:

- It can produce a scalar value to be broadcast to the shape of the group
- It can produce an object of the same shape as the input group
- It must not mutate its input

Let's consider a simple example for illustration:

```
In [75]: df = pd.DataFrame({'key': ['a', 'b', 'c'] * 4,  
    ...:                   'value': np.arange(12.)})
```

```
In [76]: df
```

```
Out[76]:
```

	key	value
0	a	0.0
1	b	1.0
2	c	2.0
3	a	3.0
4	b	4.0
5	c	5.0
6	a	6.0
7	b	7.0
8	c	8.0
9	a	9.0
10	b	10.0
11	c	11.0

Here are the group means by key:

```
In [77]: g = df.groupby('key').value
```

```
In [78]: g.mean()
```

```
Out[78]:
```

key	
a	4.5
b	5.5
c	6.5

Name: value, dtype: float64

Suppose instead we wanted to produce a Series of the same shape as `df['value']` but with values replaced by the average grouped by 'key'. We can pass the function `lambda x: x.mean()` to transform:

```
In [79]: g.transform(lambda x: x.mean())
Out[79]:
0    4.5
1    5.5
2    6.5
3    4.5
4    5.5
5    6.5
6    4.5
7    5.5
8    6.5
9    4.5
10   5.5
11   6.5
Name: value, dtype: float64
```

For built-in aggregation functions, we can pass a string alias as with the `GroupBy` `agg` method:

```
In [80]: g.transform('mean')
Out[80]:
0    4.5
1    5.5
2    6.5
3    4.5
4    5.5
5    6.5
6    4.5
7    5.5
8    6.5
9    4.5
10   5.5
11   6.5
Name: value, dtype: float64
```

Like `apply`, `transform` works with functions that return Series, but the result must be the same size as the input. For example, we can multiply each group by 2 using a `lambda` function:

```
In [81]: g.transform(lambda x: x * 2)
Out[81]:
0    0.0
1    2.0
2    4.0
3    6.0
4    8.0
5   10.0
6   12.0
```



```
7    14.0
8    16.0
9    18.0
10   20.0
11   22.0
```

```
Name: value, dtype: float64
```

As a more complicated example, we can compute the ranks in descending order for each group:

```
In [82]: g.transform(lambda x: x.rank(ascending=False))
```

```
Out[82]:
```

```
0    4.0
1    4.0
2    4.0
3    3.0
4    3.0
5    3.0
6    2.0
7    2.0
8    2.0
9    1.0
10   1.0
11   1.0
```

```
Name: value, dtype: float64
```

Consider a group transformation function composed from simple aggregations:

```
def normalize(x):
    return (x - x.mean()) / x.std()
```

We can obtain equivalent results in this case either using transform or apply:

```
In [84]: g.transform(normalize)
```

```
Out[84]:
```

```
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
```

```
Name: value, dtype: float64
```

```
In [85]: g.apply(normalize)
```

```
Out[85]:
```

```
0    -1.161895
1    -1.161895
2    -1.161895
```

```
3 -0.387298
4 -0.387298
5 -0.387298
6 0.387298
7 0.387298
8 0.387298
9 1.161895
10 1.161895
11 1.161895
```

```
Name: value, dtype: float64
```

Built-in aggregate functions like 'mean' or 'sum' are often much faster than a general apply function. These also have a “fast path” when used with transform. This allows us to perform a so-called *unwrapped* group operation:

```
In [86]: g.transform('mean')
```

```
Out[86]:
```

```
0 4.5
1 5.5
2 6.5
3 4.5
4 5.5
5 6.5
6 4.5
7 5.5
8 6.5
9 4.5
10 5.5
11 6.5
```

```
Name: value, dtype: float64
```

```
In [87]: normalized = (df['value'] - g.transform('mean')) / g.transform('std')
```

```
In [88]: normalized
```

```
Out[88]:
```

```
0 -1.161895
1 -1.161895
2 -1.161895
3 -0.387298
4 -0.387298
5 -0.387298
6 0.387298
7 0.387298
8 0.387298
9 1.161895
10 1.161895
11 1.161895
```

```
Name: value, dtype: float64
```

While an unwrapped group operation may involve multiple group aggregations, the overall benefit of vectorized operations often outweighs this.

Grouped Time Resampling

For time series data, the `resample` method is semantically a group operation based on a time intervalization. Here's a small example table:

```
In [89]: N = 15
```

```
In [90]: times = pd.date_range('2017-05-20 00:00', freq='1min', periods=N)
```

```
In [91]: df = pd.DataFrame({'time': times,
.....:                      'value': np.arange(N)})
```

```
In [92]: df
```

```
Out[92]:
```

	time	value
0	2017-05-20 00:00:00	0
1	2017-05-20 00:01:00	1
2	2017-05-20 00:02:00	2
3	2017-05-20 00:03:00	3
4	2017-05-20 00:04:00	4
5	2017-05-20 00:05:00	5
6	2017-05-20 00:06:00	6
7	2017-05-20 00:07:00	7
8	2017-05-20 00:08:00	8
9	2017-05-20 00:09:00	9
10	2017-05-20 00:10:00	10
11	2017-05-20 00:11:00	11
12	2017-05-20 00:12:00	12
13	2017-05-20 00:13:00	13
14	2017-05-20 00:14:00	14

Here, we can index by 'time' and then resample:

```
In [93]: df.set_index('time').resample('5min').count()
```

```
Out[93]:
```

time	value
2017-05-20 00:00:00	5
2017-05-20 00:05:00	5
2017-05-20 00:10:00	5

Suppose that a DataFrame contains multiple time series, marked by an additional group key column:

```
In [94]: df2 = pd.DataFrame({'time': times.repeat(3),
.....:                      'key': np.tile(['a', 'b', 'c'], N),
.....:                      'value': np.arange(N * 3.)})
```

```
In [95]: df2[:7]
```

```
Out[95]:
```

	key	time	value
0	a	2017-05-20 00:00:00	0.0
1	b	2017-05-20 00:00:00	1.0

```

2 c 2017-05-20 00:00:00 2.0
3 a 2017-05-20 00:01:00 3.0
4 b 2017-05-20 00:01:00 4.0
5 c 2017-05-20 00:01:00 5.0
6 a 2017-05-20 00:02:00 6.0

```

To do the same resampling for each value of 'key', we introduce the `pandas.TimeGrouper` object:

```
In [96]: time_key = pd.TimeGrouper('5min')
```

We can then set the time index, group by 'key' and `time_key`, and aggregate:

```
In [97]: resampled = (df2.set_index('time')
....:                  .groupby(['key', time_key])
....:                  .sum())
```

```
In [98]: resampled
```

```
Out[98]:
```

		value
key	time	
a	2017-05-20 00:00:00	30.0
	2017-05-20 00:05:00	105.0
	2017-05-20 00:10:00	180.0
b	2017-05-20 00:00:00	35.0
	2017-05-20 00:05:00	110.0
	2017-05-20 00:10:00	185.0
c	2017-05-20 00:00:00	40.0
	2017-05-20 00:05:00	115.0
	2017-05-20 00:10:00	190.0

```
In [99]: resampled.reset_index()
```

```
Out[99]:
```

	key	time	value
0	a	2017-05-20 00:00:00	30.0
1	a	2017-05-20 00:05:00	105.0
2	a	2017-05-20 00:10:00	180.0
3	b	2017-05-20 00:00:00	35.0
4	b	2017-05-20 00:05:00	110.0
5	b	2017-05-20 00:10:00	185.0
6	c	2017-05-20 00:00:00	40.0
7	c	2017-05-20 00:05:00	115.0
8	c	2017-05-20 00:10:00	190.0

One constraint with using `TimeGrouper` is that the time must be the index of the Series or DataFrame.

12.3 Techniques for Method Chaining

When applying a sequence of transformations to a dataset, you may find yourself creating numerous temporary variables that are never used in your analysis. Consider this example, for instance:

```
df = load_data()
df2 = df[df['col2'] < 0]
df2['col1_demeaned'] = df2['col1'] - df2['col1'].mean()
result = df2.groupby('key').col1_demeaned.std()
```

While we're not using any real data here, this example highlights some new methods. First, the `DataFrame.assign` method is a *functional* alternative to column assignments of the form `df[k] = v`. Rather than modifying the object in-place, it returns a new `DataFrame` with the indicated modifications. So these statements are equivalent:

```
# Usual non-functional way
df2 = df.copy()
df2['k'] = v

# Functional assign way
df2 = df.assign(k=v)
```

Assigning in-place may execute faster than using `assign`, but `assign` enables easier method chaining:

```
result = (df2.assign(col1_demeaned=df2.col1 - df2.col2.mean())
          .groupby('key')
          .col1_demeaned.std())
```

I used the outer parentheses to make it more convenient to add line breaks.

One thing to keep in mind when doing method chaining is that you may need to refer to temporary objects. In the preceding example, we cannot refer to the result of `load_data` until it has been assigned to the temporary variable `df`. To help with this, `assign` and many other pandas functions accept function-like arguments, also known as *callables*.

To show callables in action, consider a fragment of the example from before:

```
df = load_data()
df2 = df[df['col2'] < 0]
```

This can be rewritten as:

```
df = (load_data()
      [lambda x: x['col2'] < 0])
```

Here, the result of `load_data` is not assigned to a variable, so the function passed into `[]` is then *bound* to the object at that stage of the method chain.

We can continue, then, and write the entire sequence as a single chained expression:

```
result = (load_data()
          [lambda x: x.col2 < 0]
          .assign(col1_demeaned=lambda x: x.col1 - x.col1.mean())
          .groupby('key')
          .col1_demeaned.std())
```

Whether you prefer to write code in this style is a matter of taste, and splitting up the expression into multiple steps may make your code more readable.

The pipe Method

You can accomplish a lot with built-in pandas functions and the approaches to method chaining with callables that we just looked at. However, sometimes you need to use your own functions or functions from third-party libraries. This is where the `pipe` method comes in.

Consider a sequence of function calls:

```
a = f(df, arg1=v1)
b = g(a, v2, arg3=v3)
c = h(b, arg4=v4)
```

When using functions that accept and return Series or DataFrame objects, you can rewrite this using calls to `pipe`:

```
result = (df.pipe(f, arg1=v1)
          .pipe(g, v2, arg3=v3)
          .pipe(h, arg4=v4))
```

The statement `f(df)` and `df.pipe(f)` are equivalent, but `pipe` makes chained invocation easier.

A potentially useful pattern for `pipe` is to generalize sequences of operations into reusable functions. As an example, let's consider subtracting group means from a column:

```
g = df.groupby(['key1', 'key2'])
df['col1'] = df['col1'] - g.transform('mean')
```

Suppose that you wanted to be able to demean more than one column and easily change the group keys. Additionally, you might want to perform this transformation in a method chain. Here is an example implementation:

```
def group_demean(df, by, cols):
    result = df.copy()
    g = df.groupby(by)
    for c in cols:
        result[c] = df[c] - g[c].transform('mean')
    return result
```

Then it is possible to write:

```
result = (df[df.col1 < 0]
          .pipe(group_demean, ['key1', 'key2'], ['col1']))
```

12.4 Conclusion

pandas, like many open source software projects, is still changing and acquiring new and improved functionality. As elsewhere in this book, the focus here has been on the most stable functionality that is less likely to change over the next several years.

To deepen your expertise as a pandas user, I encourage you to explore the **documentation** and read the release notes as the development team makes new open source releases. We also invite you to join in on pandas development: fixing bugs, building new features, and improving the documentation.

Introduction to Modeling Libraries in Python

In this book, I have focused on providing a programming foundation for doing data analysis in Python. Since data analysts and scientists often report spending a disproportionate amount of time with data wrangling and preparation, the book's structure reflects the importance of mastering these techniques.

Which library you use for developing models will depend on the application. Many statistical problems can be solved by simpler techniques like ordinary least squares regression, while other problems may call for more advanced machine learning methods. Fortunately, Python has become one of the languages of choice for implementing analytical methods, so there are many tools you can explore after completing this book.

In this chapter, I will review some features of pandas that may be helpful when you're crossing back and forth between data wrangling with pandas and model fitting and scoring. I will then give short introductions to two popular modeling toolkits, **statsmodels** and **scikit-learn**. Since each of these projects is large enough to warrant its own dedicated book, I make no effort to be comprehensive and instead direct you to both projects' online documentation along with some other Python-based books on data science, statistics, and machine learning.

13.1 Interfacing Between pandas and Model Code

A common workflow for model development is to use pandas for data loading and cleaning before switching over to a modeling library to build the model itself. An important part of the model development process is called *feature engineering* in machine learning. This can describe any data transformation or analytics that extract

information from a raw dataset that may be useful in a modeling context. The data aggregation and GroupBy tools we have explored in this book are used often in a feature engineering context.

While details of “good” feature engineering are out of scope for this book, I will show some methods to make switching between data manipulation with pandas and modeling as painless as possible.

The point of contact between pandas and other analysis libraries is usually NumPy arrays. To turn a DataFrame into a NumPy array, use the `.values` property:

```
In [10]: import pandas as pd
```

```
In [11]: import numpy as np
```

```
In [12]: data = pd.DataFrame({
.....:     'x0': [1, 2, 3, 4, 5],
.....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],
.....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})
```

```
In [13]: data
```

```
Out[13]:
```

	x0	x1	y
0	1	0.01	-1.5
1	2	-0.01	0.0
2	3	0.25	3.6
3	4	-4.10	1.3
4	5	0.00	-2.0

```
In [14]: data.columns
```

```
Out[14]: Index(['x0', 'x1', 'y'], dtype='object')
```

```
In [15]: data.values
```

```
Out[15]:
```

```
array([[ 1. ,  0.01, -1.5 ],
       [ 2. , -0.01,  0. ],
       [ 3. ,  0.25,  3.6 ],
       [ 4. , -4.1 ,  1.3 ],
       [ 5. ,  0. , -2.  ]])
```

To convert back to a DataFrame, as you may recall from earlier chapters, you can pass a two-dimensional ndarray with optional column names:

```
In [16]: df2 = pd.DataFrame(data.values, columns=['one', 'two', 'three'])
```

```
In [17]: df2
```

```
Out[17]:
```

	one	two	three
0	1.0	0.01	-1.5
1	2.0	-0.01	0.0
2	3.0	0.25	3.6

```
3  4.0 -4.10  1.3
4  5.0  0.00 -2.0
```



The `.values` attribute is intended to be used when your data is homogeneous—for example, all numeric types. If you have heterogeneous data, the result will be an ndarray of Python objects:

```
In [18]: df3 = data.copy()
```

```
In [19]: df3['strings'] = ['a', 'b', 'c', 'd', 'e']
```

```
In [20]: df3
```

```
Out[20]:
```

```
   x0  x1  y strings
0   1  0.01 -1.5     a
1   2 -0.01  0.0     b
2   3  0.25  3.6     c
3   4 -4.10  1.3     d
4   5  0.00 -2.0     e
```

```
In [21]: df3.values
```

```
Out[21]:
```

```
array([[1, 0.01, -1.5, 'a'],
       [2, -0.01, 0.0, 'b'],
       [3, 0.25, 3.6, 'c'],
       [4, -4.1, 1.3, 'd'],
       [5, 0.0, -2.0, 'e']], dtype=object)
```

For some models, you may only wish to use a subset of the columns. I recommend using loc indexing with values:

```
In [22]: model_cols = ['x0', 'x1']
```

```
In [23]: data.loc[:, model_cols].values
```

```
Out[23]:
```

```
array([[ 1. ,  0.01],
       [ 2. , -0.01],
       [ 3. ,  0.25],
       [ 4. , -4.1 ],
       [ 5. ,  0.  ]])
```

Some libraries have native support for pandas and do some of this work for you automatically: converting to NumPy from DataFrame and attaching model parameter names to the columns of output tables or Series. In other cases, you will have to perform this “metadata management” manually.

In [Chapter 12](#) we looked at pandas’s Categorical type and the `pandas.get_dummies` function. Suppose we had a non-numeric column in our example dataset:

```
In [24]: data['category'] = pd.Categorical(['a', 'b', 'a', 'a', 'b'],
.....:                                     categories=['a', 'b'])
```

```
In [25]: data
```

```
Out[25]:
```

	x0	x1	y	category
0	1	0.01	-1.5	a
1	2	-0.01	0.0	b
2	3	0.25	3.6	a
3	4	-4.10	1.3	a
4	5	0.00	-2.0	b

If we wanted to replace the 'category' column with dummy variables, we create dummy variables, drop the 'category' column, and then join the result:

```
In [26]: dummies = pd.get_dummies(data.category, prefix='category')
```

```
In [27]: data_with_dummies = data.drop('category', axis=1).join(dummies)
```

```
In [28]: data_with_dummies
```

```
Out[28]:
```

	x0	x1	y	category_a	category_b
0	1	0.01	-1.5	1	0
1	2	-0.01	0.0	0	1
2	3	0.25	3.6	1	0
3	4	-4.10	1.3	1	0
4	5	0.00	-2.0	0	1

There are some nuances to fitting certain statistical models with dummy variables. It may be simpler and less error-prone to use Patsy (the subject of the next section) when you have more than simple numeric columns.

13.2 Creating Model Descriptions with Patsy

Patsy is a Python library for describing statistical models (especially linear models) with a small string-based “formula syntax,” which is inspired by (but not exactly the same as) the formula syntax used by the R and S statistical programming languages.

Patsy is well supported for specifying linear models in statsmodels, so I will focus on some of the main features to help you get up and running. Patsy’s *formulas* are a special string syntax that looks like:

```
y ~ x0 + x1
```

The syntax $a + b$ does not mean to add a to b , but rather that these are *terms* in the *design matrix* created for the model. The `patsy.dmatrices` function takes a formula string along with a dataset (which can be a DataFrame or a dict of arrays) and produces design matrices for a linear model:

```
In [29]: data = pd.DataFrame({
.....:     'x0': [1, 2, 3, 4, 5],
```

```
.....: 'x1': [0.01, -0.01, 0.25, -4.1, 0.],
.....: 'y': [-1.5, 0., 3.6, 1.3, -2.]})
```

```
In [30]: data
```

```
Out[30]:
  x0  x1  y
0  1  0.01 -1.5
1  2 -0.01  0.0
2  3  0.25  3.6
3  4 -4.10  1.3
4  5  0.00 -2.0
```

```
In [31]: import patsy
```

```
In [32]: y, X = patsy.dmatrices('y ~ x0 + x1', data)
```

Now we have:

```
In [33]: y
```

```
Out[33]:
DesignMatrix with shape (5, 1)
  y
-1.5
 0.0
 3.6
 1.3
-2.0
Terms:
  'y' (column 0)
```

```
In [34]: X
```

```
Out[34]:
DesignMatrix with shape (5, 3)
  Intercept  x0  x1
  1  1  0.01
  1  2 -0.01
  1  3  0.25
  1  4 -4.10
  1  5  0.00
Terms:
  'Intercept' (column 0)
  'x0' (column 1)
  'x1' (column 2)
```

These Patsy DesignMatrix instances are NumPy ndarrays with additional metadata:

```
In [35]: np.asarray(y)
```

```
Out[35]:
array([[ -1.5],
       [  0. ],
       [  3.6],
       [  1.3],
       [ -2. ]])
```

```
In [36]: np.asarray(X)
Out[36]:
array([[ 1. ,  1. ,  0.01],
       [ 1. ,  2. , -0.01],
       [ 1. ,  3. ,  0.25],
       [ 1. ,  4. , -4.1 ],
       [ 1. ,  5. ,  0. ]])
```

You might wonder where the Intercept term came from. This is a convention for linear models like ordinary least squares (OLS) regression. You can suppress the intercept by adding the term $+ \theta$ to the model:

```
In [37]: patsy.dmatrices('y ~ x0 + x1 + \theta', data)[1]
Out[37]:
DesignMatrix with shape (5, 2)
   x0    x1
1  0.01
2 -0.01
3  0.25
4 -4.10
5  0.00
Terms:
'x0' (column 0)
'x1' (column 1)
```

The Patsy objects can be passed directly into algorithms like `numpy.linalg.lstsq`, which performs an ordinary least squares regression:

```
In [38]: coef, resid, _, _ = np.linalg.lstsq(X, y)
```

The model metadata is retained in the `design_info` attribute, so you can reattach the model column names to the fitted coefficients to obtain a Series, for example:

```
In [39]: coef
Out[39]:
array([[ 0.3129],
       [-0.0791],
       [-0.2655]])
```

```
In [40]: coef = pd.Series(coef.squeeze(), index=X.design_info.column_names)
```

```
In [41]: coef
Out[41]:
Intercept    0.312910
x0          -0.079106
x1          -0.265464
dtype: float64
```

Data Transformations in Patsy Formulas

You can mix Python code into your Patsy formulas; when evaluating the formula the library will try to find the functions you use in the enclosing scope:

```
In [42]: y, X = patsy.dmatrices('y ~ x0 + np.log(np.abs(x1) + 1)', data)
```

```
In [43]: X
```

```
Out[43]:
```

```
DesignMatrix with shape (5, 3)
Intercept  x0  np.log(np.abs(x1) + 1)
           1  1  0.00995
           1  2  0.00995
           1  3  0.22314
           1  4  1.62924
           1  5  0.00000
```

```
Terms:
```

```
'Intercept' (column 0)
'x0' (column 1)
'np.log(np.abs(x1) + 1)' (column 2)
```

Some commonly used variable transformations include standardizing (to mean 0 and variance 1) and centering (subtracting the mean). Patsy has built-in functions for this purpose:

```
In [44]: y, X = patsy.dmatrices('y ~ standardize(x0) + center(x1)', data)
```

```
In [45]: X
```

```
Out[45]:
```

```
DesignMatrix with shape (5, 3)
Intercept  standardize(x0)  center(x1)
           1  -1.41421  0.78
           1  -0.70711  0.76
           1  0.00000  1.02
           1  0.70711  -3.33
           1  1.41421  0.77
```

```
Terms:
```

```
'Intercept' (column 0)
'standardize(x0)' (column 1)
'center(x1)' (column 2)
```

As part of a modeling process, you may fit a model on one dataset, then evaluate the model based on another. This might be a *hold-out* portion or new data that is observed later. When applying transformations like `center` and `standardize`, you should be careful when using the model to form predications based on new data. These are called *stateful* transformations, because you must use statistics like the mean or standard deviation of the original dataset when transforming a new dataset.

The `patsy.build_design_matrices` function can apply transformations to new *out-of-sample* data using the saved information from the original *in-sample* dataset:

```
In [46]: new_data = pd.DataFrame({
.....:     'x0': [6, 7, 8, 9],
.....:     'x1': [3.1, -0.5, 0, 2.3],
.....:     'y': [1, 2, 3, 4]})
```

```
In [47]: new_X = patsy.build_design_matrices([X.design_info], new_data)
```

```
In [48]: new_X
```

```
Out[48]:
```

```
[DesignMatrix with shape (4, 3)
 Intercept  standardize(x0)  center(x1)
          1          2.12132      3.87
          1          2.82843      0.27
          1          3.53553      0.77
          1          4.24264      3.07
Terms:
  'Intercept' (column 0)
  'standardize(x0)' (column 1)
  'center(x1)' (column 2)]
```

Because the plus symbol (+) in the context of Patsy formulas does not mean addition, when you want to add columns from a dataset by name, you must wrap them in the special *I* function:

```
In [49]: y, X = patsy.dmatrices('y ~ I(x0 + x1)', data)
```

```
In [50]: X
```

```
Out[50]:
```

```
DesignMatrix with shape (5, 2)
 Intercept  I(x0 + x1)
          1          1.01
          1          1.99
          1          3.25
          1         -0.10
          1          5.00
Terms:
  'Intercept' (column 0)
  'I(x0 + x1)' (column 1)
```

Patsy has several other built-in transforms in the `patsy.builtins` module. See the online documentation for more.

Categorical data has a special class of transformations, which I explain next.

Categorical Data and Patsy

Non-numeric data can be transformed for a model design matrix in many different ways. A complete treatment of this topic is outside the scope of this book and would be best studied along with a course in statistics.

When you use non-numeric terms in a Patsy formula, they are converted to dummy variables by default. If there is an intercept, one of the levels will be left out to avoid collinearity:

```
In [51]: data = pd.DataFrame({
.....:     'key1': ['a', 'a', 'b', 'b', 'a', 'b', 'a', 'b'],
.....:     'key2': [0, 1, 0, 1, 0, 1, 0, 0],
.....:     'v1': [1, 2, 3, 4, 5, 6, 7, 8],
.....:     'v2': [-1, 0, 2.5, -0.5, 4.0, -1.2, 0.2, -1.7]
.....: })
```

```
In [52]: y, X = patsy.dmatrices('v2 ~ key1', data)
```

```
In [53]: X
```

```
Out[53]:
```

```
DesignMatrix with shape (8, 2)
```

Intercept	key1[T.b]
1	0
1	0
1	1
1	1
1	0
1	1
1	0
1	1

```
Terms:
```

```
'Intercept' (column 0)
'key1' (column 1)
```

If you omit the intercept from the model, then columns for each category value will be included in the model design matrix:

```
In [54]: y, X = patsy.dmatrices('v2 ~ key1 + 0', data)
```

```
In [55]: X
```

```
Out[55]:
```

```
DesignMatrix with shape (8, 2)
```

key1[a]	key1[b]
1	0
1	0
0	1
0	1
1	0
0	1
1	0
0	1

```
Terms:
```

```
'key1' (columns 0:2)
```

Numeric columns can be interpreted as categorical with the C function:

```
In [56]: y, X = patsy.dmatrices('v2 ~ C(key2)', data)
```



```

In [57]: X
Out[57]:
DesignMatrix with shape (8, 2)
  Intercept  C(key2)[T.1]
      1      0
      1      1
      1      0
      1      1
      1      0
      1      1
      1      0
      1      0

Terms:
  'Intercept' (column 0)
  'C(key2)' (column 1)

```

When you're using multiple categorical terms in a model, things can be more complicated, as you can include interaction terms of the form `key1:key2`, which can be used, for example, in analysis of variance (ANOVA) models:

```
In [58]: data['key2'] = data['key2'].map({0: 'zero', 1: 'one'})
```

```

In [59]: data
Out[59]:
  key1  key2  v1  v2
0    a  zero  1 -1.0
1    a  one   2  0.0
2    b  zero  3  2.5
3    b  one   4 -0.5
4    a  zero  5  4.0
5    b  one   6 -1.2
6    a  zero  7  0.2
7    b  zero  8 -1.7

```

```
In [60]: y, X = patsy.dmatrices('v2 ~ key1 + key2', data)
```

```

In [61]: X
Out[61]:
DesignMatrix with shape (8, 3)
  Intercept  key1[T.b]  key2[T.zero]
      1      0      1
      1      0      0
      1      1      1
      1      1      0
      1      0      1
      1      1      0
      1      0      1
      1      1      1

Terms:
  'Intercept' (column 0)
  'key1' (column 1)
  'key2' (column 2)

```

```
In [62]: y, X = patsy.dmatrices('v2 ~ key1 + key2 + key1:key2', data)
```

```
In [63]: X
```

```
Out[63]:
```

```
DesignMatrix with shape (8, 4)
```

Intercept	key1[T.b]	key2[T.zero]	key1[T.b]:key2[T.zero]
1	0	1	0
1	0	0	0
1	1	1	1
1	1	0	0
1	0	1	0
1	1	0	0
1	0	1	0
1	1	1	1

```
Terms:
```

```
'Intercept' (column 0)  
'key1' (column 1)  
'key2' (column 2)  
'key1:key2' (column 3)
```

Patsy provides for other ways to transform categorical data, including transformations for terms with a particular ordering. See the online documentation for more.

13.3 Introduction to statsmodels

statsmodels is a Python library for fitting many kinds of statistical models, performing statistical tests, and data exploration and visualization. Statsmodels contains more “classical” frequentist statistical methods, while Bayesian methods and machine learning models are found in other libraries.

Some kinds of models found in statsmodels include:

- Linear models, generalized linear models, and robust linear models
- Linear mixed effects models
- Analysis of variance (ANOVA) methods
- Time series processes and state space models
- Generalized method of moments

In the next few pages, we will use a few basic tools in statsmodels and explore how to use the modeling interfaces with Patsy formulas and pandas DataFrame objects.

Estimating Linear Models

There are several kinds of linear regression models in statsmodels, from the more basic (e.g., ordinary least squares) to more complex (e.g., iteratively reweighted least squares).

Linear models in statsmodels have two different main interfaces: array-based and formula-based. These are accessed through these API module imports:

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

To show how to use these, we generate a linear model from some random data:

```
def dnorm(mean, variance, size=1):
    if isinstance(size, int):
        size = size,
    return mean + np.sqrt(variance) * np.random.randn(*size)
```

```
# For reproducibility
np.random.seed(12345)
```

```
N = 100
X = np.c_[dnorm(0, 0.4, size=N),
          dnorm(0, 0.6, size=N),
          dnorm(0, 0.2, size=N)]
eps = dnorm(0, 0.1, size=N)
beta = [0.1, 0.3, 0.5]
```

```
y = np.dot(X, beta) + eps
```

Here, I wrote down the “true” model with known parameters beta. In this case, dnorm is a helper function for generating normally distributed data with a particular mean and variance. So now we have:

```
In [66]: X[:5]
Out[66]:
array([[ -0.1295, -1.2128,  0.5042],
       [  0.3029, -0.4357, -0.2542],
       [-0.3285, -0.0253,  0.1384],
       [-0.3515, -0.7196, -0.2582],
       [ 1.2433, -0.3738, -0.5226]])
```

```
In [67]: y[:5]
Out[67]: array([ 0.4279, -0.6735, -0.0909, -0.4895, -0.1289])
```

A linear model is generally fitted with an intercept term as we saw before with Patsy. The sm.add_constant function can add an intercept column to an existing matrix:

```
In [68]: X_model = sm.add_constant(X)
```

```
In [69]: X_model[:5]
Out[69]:
array([[ 1.    , -0.1295, -1.2128,  0.5042],
       [ 1.    ,  0.3029, -0.4357, -0.2542],
       [ 1.    , -0.3285, -0.0253,  0.1384],
       [ 1.    , -0.3515, -0.7196, -0.2582],
       [ 1.    ,  1.2433, -0.3738, -0.5226]])
```

The `sm.OLS` class can fit an ordinary least squares linear regression:

```
In [70]: model = sm.OLS(y, X)
```

The model's `fit` method returns a regression results object containing estimated model parameters and other diagnostics:

```
In [71]: results = model.fit()
```

```
In [72]: results.params
```

```
Out[72]: array([ 0.1783,  0.223 ,  0.501 ])
```

The `summary` method on `results` can print a model detailing diagnostic output of the model:

```
In [73]: print(results.summary())
```

```
OLS Regression Results
```

```
=====
Dep. Variable:          y      R-squared:          0.430
Model:                  OLS    Adj. R-squared:     0.413
Method:                 Least Squares  F-statistic:       24.42
Date:                   Mon, 25 Sep 2017  Prob (F-statistic): 7.44e-12
Time:                   14:06:15    Log-Likelihood:    -34.305
No. Observations:      100        AIC:               74.61
Df Residuals:          97         BIC:               82.42
Df Model:               3
Covariance Type:       nonrobust
=====
```

```
=====
              coef      std err          t      P>|t|      [0.025      0.975]
-----
x1            0.1783      0.053       3.364      0.001      0.073      0.283
x2            0.2230      0.046       4.818      0.000      0.131      0.315
x3            0.5010      0.080       6.237      0.000      0.342      0.660
=====
```

```
Omnibus:                4.662    Durbin-Watson:        2.201
Prob(Omnibus):           0.097    Jarque-Bera (JB):     4.098
Skew:                   0.481    Prob(JB):             0.129
Kurtosis:                3.243    Cond. No.             1.74
=====
```

```
Warnings:
```

```
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

The parameter names here have been given the generic names `x1`, `x2`, and so on. Suppose instead that all of the model parameters are in a `DataFrame`:

```
In [74]: data = pd.DataFrame(X, columns=['col0', 'col1', 'col2'])
```

```
In [75]: data['y'] = y
```

```
In [76]: data[:5]
```

```
Out[76]:
   col0  col1  col2  y
```

```
0 -0.129468 -1.212753 0.504225 0.427863
1 0.302910 -0.435742 -0.254180 -0.673480
2 -0.328522 -0.025302 0.138351 -0.090878
3 -0.351475 -0.719605 -0.258215 -0.489494
4 1.243269 -0.373799 -0.522629 -0.128941
```

Now we can use the statsmodels formula API and Patsy formula strings:

```
In [77]: results = smf.ols('y ~ col0 + col1 + col2', data=data).fit()
```

```
In [78]: results.params
```

```
Out[78]:
```

```
Intercept    0.033559
col0         0.176149
col1         0.224826
col2         0.514808
```

```
dtype: float64
```

```
In [79]: results.tvalues
```

```
Out[79]:
```

```
Intercept    0.952188
col0         3.319754
col1         4.850730
col2         6.303971
```

```
dtype: float64
```

Observe how statsmodels has returned results as Series with the DataFrame column names attached. We also do not need to use `add_constant` when using formulas and pandas objects.

Given new out-of-sample data, you can compute predicted values given the estimated model parameters:

```
In [80]: results.predict(data[:5])
```

```
Out[80]:
```

```
0 -0.002327
1 -0.141904
2 0.041226
3 -0.323070
4 -0.100535
```

```
dtype: float64
```

There are many additional tools for analysis, diagnostics, and visualization of linear model results in statsmodels that you can explore. There are also other kinds of linear models beyond ordinary least squares.

Estimating Time Series Processes

Another class of models in statsmodels are for time series analysis. Among these are autoregressive processes, Kalman filtering and other state space models, and multivariate autoregressive models.

Let's simulate some time series data with an autoregressive structure and noise:

```
init_x = 4

import random
values = [init_x, init_x]
N = 1000

b0 = 0.8
b1 = -0.4
noise = dnorm(0, 0.1, N)
for i in range(N):
    new_x = values[-1] * b0 + values[-2] * b1 + noise[i]
    values.append(new_x)
```

This data has an AR(2) structure (two *lags*) with parameters 0.8 and -0.4 . When you fit an AR model, you may not know the number of lagged terms to include, so you can fit the model with some larger number of lags:

```
In [82]: MAXLAGS = 5

In [83]: model = sm.tsa.AR(values)

In [84]: results = model.fit(MAXLAGS)
```

The estimated parameters in the results have the intercept first and the estimates for the first two lags next:

```
In [85]: results.params
Out[85]: array([-0.0062,  0.7845, -0.4085, -0.0136,  0.015 ,  0.0143])
```

Deeper details of these models and how to interpret their results is beyond what I can cover in this book, but there's plenty more to discover in the statsmodels documentation.

13.4 Introduction to scikit-learn

scikit-learn is one of the most widely used and trusted general-purpose Python machine learning toolkits. It contains a broad selection of standard supervised and unsupervised machine learning methods with tools for model selection and evaluation, data transformation, data loading, and model persistence. These models can be used for classification, clustering, prediction, and other common tasks.

There are excellent online and printed resources for learning about machine learning and how to apply libraries like scikit-learn and TensorFlow to solve real-world problems. In this section, I will give a brief flavor of the scikit-learn API style.

At the time of this writing, scikit-learn does not have deep pandas integration, though there are some add-on third-party packages that are still in development. pandas can be very useful for massaging datasets prior to model fitting, though.

As an example, I use a **now-classic dataset from a Kaggle competition** about passenger survival rates on the *Titanic*, which sank in 1912. We load the test and training dataset using pandas:

```
In [86]: train = pd.read_csv('datasets/titanic/train.csv')
```

```
In [87]: test = pd.read_csv('datasets/titanic/test.csv')
```

```
In [88]: train[:4]
```

```
Out[88]:
```

	PassengerId	Survived	Pclass	\		Name	Sex	Age	SibSp	\
0	1	0	3							
1	2	1	1							
2	3	1	3							
3	4	1	1							
0						Braund, Mr. Owen Harris	male	22.0	1	
1						Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	
2						Heikkinen, Miss. Laina	female	26.0	0	
3						Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	
	Parch	Ticket	Fare	Cabin	Embarked					
0	0	A/5 21171	7.2500	NaN	S					
1	0	PC 17599	71.2833	C85	C					
2	0	STON/O2. 3101282	7.9250	NaN	S					
3	0	113803	53.1000	C123	S					

Libraries like statsmodels and scikit-learn generally cannot be fed missing data, so we look at the columns to see if there are any that contain missing data:

```
In [89]: train.isnull().sum()
```

```
Out[89]:
```

PassengerId	0
Survived	0
Pclass	0
Name	0
Sex	0
Age	177
SibSp	0
Parch	0
Ticket	0
Fare	0
Cabin	687
Embarked	2

dtype: int64

```
In [90]: test.isnull().sum()
```

```
Out[90]:
```

PassengerId	0
Pclass	0
Name	0
Sex	0
Age	86

```
SibSp      0
Parch      0
Ticket     0
Fare       1
Cabin     327
Embarked   0
dtype: int64
```

In statistics and machine learning examples like this one, a typical task is to predict whether a passenger would survive based on features in the data. A model is fitted on a *training* dataset and then evaluated on an out-of-sample *testing* dataset.

I would like to use Age as a predictor, but it has missing data. There are a number of ways to do missing data imputation, but I will do a simple one and use the median of the training dataset to fill the nulls in both tables:

```
In [91]: impute_value = train['Age'].median()

In [92]: train['Age'] = train['Age'].fillna(impute_value)

In [93]: test['Age'] = test['Age'].fillna(impute_value)
```

Now we need to specify our models. I add a column IsFemale as an encoded version of the 'Sex' column:

```
In [94]: train['IsFemale'] = (train['Sex'] == 'female').astype(int)

In [95]: test['IsFemale'] = (test['Sex'] == 'female').astype(int)
```

Then we decide on some model variables and create NumPy arrays:

```
In [96]: predictors = ['Pclass', 'IsFemale', 'Age']

In [97]: X_train = train[predictors].values

In [98]: X_test = test[predictors].values

In [99]: y_train = train['Survived'].values
```

```
In [100]: X_train[:5]
Out[100]:
array([[ 3.,  0., 22.],
       [ 1.,  1., 38.],
       [ 3.,  1., 26.],
       [ 1.,  1., 35.],
       [ 3.,  0., 35.]])
```

```
In [101]: y_train[:5]
Out[101]: array([0, 1, 1, 1, 0])
```

I make no claims that this is a good model nor that these features are engineered properly. We use the LogisticRegression model from scikit-learn and create a model instance:


```
In [102]: from sklearn.linear_model import LogisticRegression
```

```
In [103]: model = LogisticRegression()
```

Similar to statsmodels, we can fit this model to the training data using the model's fit method:

```
In [104]: model.fit(X_train, y_train)
```

```
Out[104]:
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                    intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                    penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                    verbose=0, warm_start=False)
```

Now, we can form predictions for the test dataset using `model.predict`:

```
In [105]: y_predict = model.predict(X_test)
```

```
In [106]: y_predict[:10]
```

```
Out[106]: array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0])
```

If you had the true values for the test dataset, you could compute an accuracy percentage or some other error metric:

```
(y_true == y_predict).mean()
```

In practice, there are often many additional layers of complexity in model training. Many models have parameters that can be tuned, and there are techniques such as *cross-validation* that can be used for parameter tuning to avoid overfitting to the training data. This can often yield better predictive performance or robustness on new data.

Cross-validation works by splitting the training data to simulate out-of-sample prediction. Based on a model accuracy score like mean squared error, one can perform a grid search on model parameters. Some models, like logistic regression, have estimator classes with built-in cross-validation. For example, the `LogisticRegressionCV` class can be used with a parameter indicating how fine-grained of a grid search to do on the model regularization parameter `C`:

```
In [107]: from sklearn.linear_model import LogisticRegressionCV
```

```
In [108]: model_cv = LogisticRegressionCV(10)
```

```
In [109]: model_cv.fit(X_train, y_train)
```

```
Out[109]:
```

```
LogisticRegressionCV(Cs=10, class_weight=None, cv=None, dual=False,
                      fit_intercept=True, intercept_scaling=1.0, max_iter=100,
                      multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
                      refit=True, scoring=None, solver='lbfgs', tol=0.0001, verbose=0)
```

To do cross-validation by hand, you can use the `cross_val_score` helper function, which handles the data splitting process. For example, to cross-validate our model with four non-overlapping splits of the training data, we can do:

```
In [110]: from sklearn.model_selection import cross_val_score

In [111]: model = LogisticRegression(C=10)

In [112]: scores = cross_val_score(model, X_train, y_train, cv=4)

In [113]: scores
Out[113]: array([ 0.7723,  0.8027,  0.7703,  0.7883])
```

The default scoring metric is model-dependent, but it is possible to choose an explicit scoring function. Cross-validated models take longer to train, but can often yield better model performance.

13.5 Continuing Your Education

While I have only skimmed the surface of some Python modeling libraries, there are more and more frameworks for various kinds of statistics and machine learning either implemented in Python or with a Python user interface.

This book is focused especially on data wrangling, but there are many others dedicated to modeling and data science tools. Some excellent ones are:

- *Introduction to Machine Learning with Python* by Andreas Mueller and Sarah Guido (O'Reilly)
- *Python Data Science Handbook* by Jake VanderPlas (O'Reilly)
- *Data Science from Scratch: First Principles with Python* by Joel Grus (O'Reilly)
- *Python Machine Learning* by Sebastian Raschka (Packt Publishing)
- *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by Aurélien Géron (O'Reilly)

While books can be valuable resources for learning, they can sometimes grow out of date when the underlying open source software changes. It's a good idea to be familiar with the documentation for the various statistics or machine learning frameworks to stay up to date on the latest features and API.

Data Analysis Examples

Now that we've reached the end of this book's main chapters, we're going to take a look at a number of real-world datasets. For each dataset, we'll use the techniques presented in this book to extract meaning from the raw data. The demonstrated techniques can be applied to all manner of other datasets, including your own. This chapter contains a collection of miscellaneous example datasets that you can use for practice with the tools in this book.

The example datasets are found in the book's accompanying [GitHub repository](#).

14.1 1.USA.gov Data from Bitly

In 2011, URL shortening service [Bitly](#) partnered with the US 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. This service is shut down at the time of this writing (2017), but we preserved one of the data files for the book's examples.

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 we may see something like this:

```
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": "US", "nk": 1,  
"tz": "America\\/New_York", "gr": "MA", "g": "A6qOVH", "h": "wflQtf", "l":  
"orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r":  
"http:\\\\www.facebook.com\\/l\\/7AQEFzjSi\\/1.usa.gov\\/wflQtf", "u":  
"http:\\\\www.ncbi.nlm.nih.gov\\/pubmed\\/22415991", "t": 1331923247, "hc":  
1331822918, "cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }\\n'
```

Python has both built-in and third-party libraries for converting a JSON string into a Python dictionary object. Here we'll use the `json` module and its `loads` function invoked on each line in the sample file we 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 (KHTML, like Gecko)
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'}
```

Counting Time Zones in Pure Python

Suppose we were interested in finding the most often-occurring time zones in the dataset (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
```

At this point, we can use the **seaborn** package to make a horizontal bar plot (see **Figure 14-1** for the resulting visualization):

```
In [36]: import seaborn as sns
```

```
In [37]: subset = tz_counts[:10]
```

```
In [38]: sns.barplot(y=subset.index, x=subset.values)
```

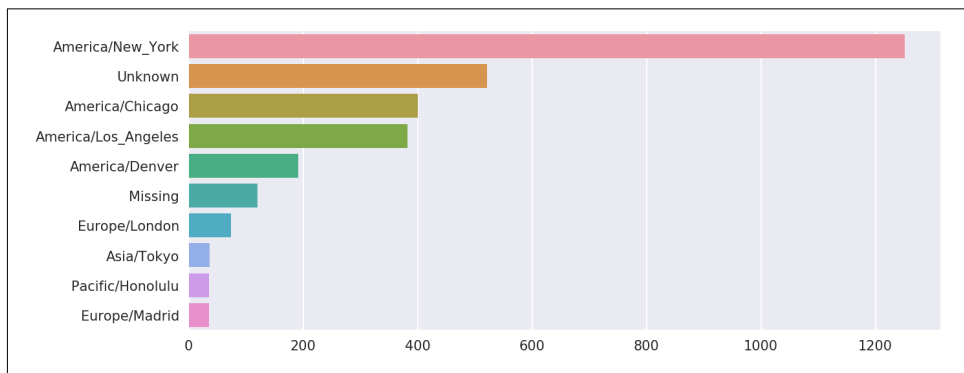


Figure 14-1. Top time zones in the 1.usa.gov sample data

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-P9'
```

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.dropna()])

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, we’ll exclude these from the data:

```
In [45]: cframe = frame[frame.a.notnull()]
```

We want to then compute a value for whether each row is Windows or not:

```
In [47]: cframe['os'] = np.where(cframe['a'].str.contains('Windows'),
....:                          '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, can be computed with `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
Africa/Cairo                245.0    276.0
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 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 convenience method called `nlargest` that 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 14-2](#)):

```
# 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_subset)
```

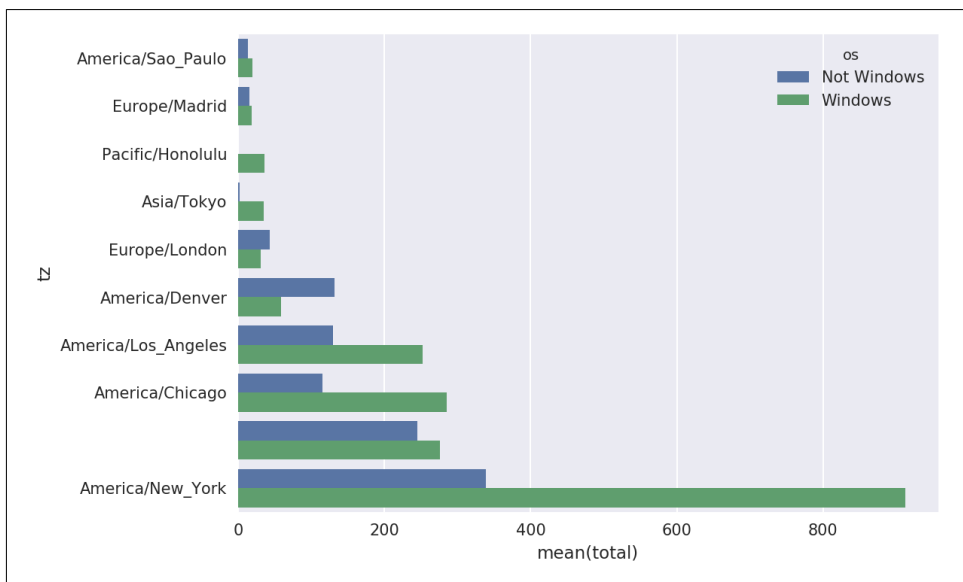


Figure 14-2. Top time zones by Windows and non-Windows users

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 14-3](#):

```
In [65]: sns.barplot(x='normed_total', y='tz', hue='os', data=results)
```

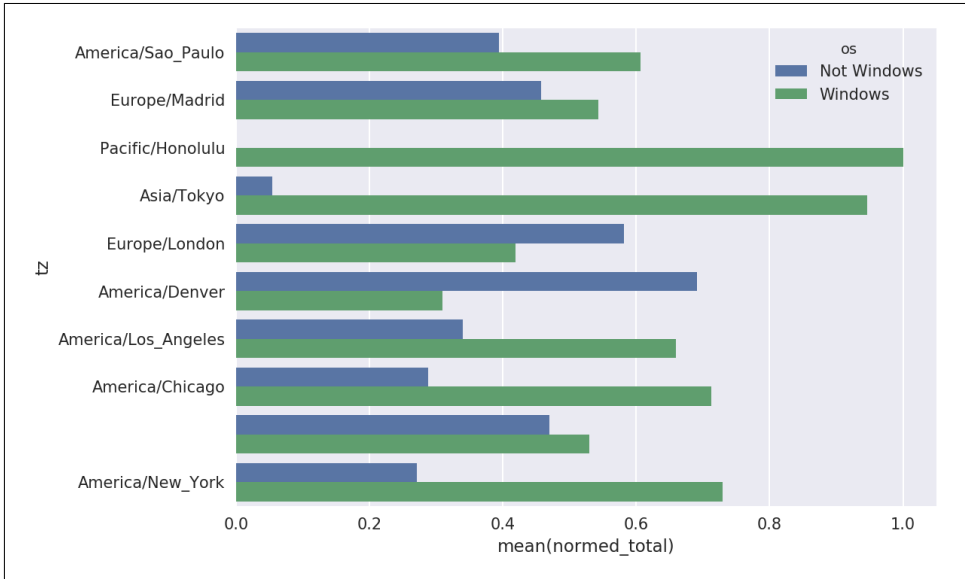


Figure 14-3. Percentage Windows and non-Windows users in top-occurring time zones

We could have computed the normalized sum more efficiently by using the `transform` method with `groupby`:

```
In [66]: g = count_subset.groupby('tz')
```

```
In [67]: results2 = count_subset.total / g.total.transform('sum')
```

14.2 MovieLens 1M Dataset

GroupLens Research 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 identification, and occupation). Such data is often of interest in the development of recommendation systems based on machine learning algorithms. While we do not explore machine learning techniques in detail in this book, I will show you how to slice and dice datasets like these into the exact form you need.

The MovieLens 1M dataset contains 1 million ratings collected from 6,000 users on 4,000 movies. It's spread across three tables: ratings, user information, and movie information. After extracting the data from the ZIP file, we can load each table 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='::',  
                      header=None, names=unames)
```

```
rnames = ['user_id', 'movie_id', 'rating', 'timestamp']  
ratings = pd.read_table('datasets/movielens/ratings.dat', sep='::',  
                        header=None, names=rnames)
```

```
mnames = ['movie_id', 'title', 'genres']  
movies = pd.read_table('datasets/movielens/movies.dat', sep='::',  
                       header=None, 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	genres
0	1	Toy Story (1995)	Animation Children's Comedy
1	2	Jumanji (1995)	Adventure Children's Fantasy
2	3	Grumpier Old Men (1995)	Comedy Romance
3	4	Waiting to Exhale (1995)	Comedy Drama
4	5	Father of the Bride Part II (1995)	Comedy

```
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    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 dataset'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 and then merge 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  occupation  zip \
0         1         1193      5  978300760      F    1          10  48067
1         2         1193      5  978298413      M   56          16  70072
2        12         1193      4  978220179      M   25          12  32793
3        15         1193      4  978199279      M   25           7  22903
4        17         1193      5  978158471      M  50           1  95350
...
1000204   5949       2198      5  958846401      M   18          17  47901
1000205   5675       2703      3  976029116      M   35          14  30030
1000206   5780       2845      1  958153068      M   18          17  92886
1000207   5851       3607      5  957756608      F   18          20  55410
1000208   5938       2909      4  957273353      M   25           1  35401
...
                                title  genres
0      One Flew Over the Cuckoo's Nest (1975)  Drama
1      One Flew Over the Cuckoo's Nest (1975)  Drama
2      One Flew Over the Cuckoo's Nest (1975)  Drama
3      One Flew Over the Cuckoo's Nest (1975)  Drama
4      One Flew Over the Cuckoo's Nest (1975)  Drama
...
                                title  genres
1000204      Modulations (1998)  Documentary
1000205      Broken Vessels (1998)  Drama
1000206      White Boys (1999)  Drama
1000207      One Little Indian (1973)  Comedy|Drama|Western
1000208  Five Wives, Three Secretaries and Me (1998)  Documentary
[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

```

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. I first filter down to movies that received at least 250 ratings (a completely arbitrary number); to do this, I then 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 (1968)',
      '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`:

```

# Select rows on the index
In [82]: mean_ratings = mean_ratings.loc[active_titles]

In [83]: mean_ratings
Out[83]:
gender                F          M
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]:
gender                F          M
title
Close Shave, A (1995)    4.644444  4.473795
Wrong Trousers, The (1993)  4.588235  4.478261
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)  4.572650  4.464589
Wallace & Gromit: The Best of Aardman Animation...  4.563107  4.385075
Schindler's List (1993)   4.562602  4.491415
Shawshank Redemption, The (1994)  4.539075  4.560625
Grand Day Out, A (1992)   4.537879  4.293255

```

To Kill a Mockingbird (1962)	4.536667	4.372611
Creature Comforts (1990)	4.513889	4.272277
Usual Suspects, The (1995)	4.513317	4.518248

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' yields the movies with the greatest rating difference so that we can see which ones 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	diff
title			
Dirty Dancing (1987)	3.790378	2.959596	-0.830782
Jumpin' Jack Flash (1986)	3.254717	2.578358	-0.676359
Grease (1978)	3.975265	3.367041	-0.608224
Little Women (1994)	3.870588	3.321739	-0.548849
Steel Magnolias (1989)	3.901734	3.365957	-0.535777
Anastasia (1997)	3.800000	3.281609	-0.518391
Rocky Horror Picture Show, The (1975)	3.673016	3.160131	-0.512885
Color Purple, The (1985)	4.158192	3.659341	-0.498851
Age of Innocence, The (1993)	3.827068	3.339506	-0.487561
Free Willy (1993)	2.921348	2.438776	-0.482573

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	diff
title			
Good, The Bad and The Ugly, The (1966)	3.494949	4.221300	0.726351
Kentucky Fried Movie, The (1977)	2.878788	3.555147	0.676359
Dumb & Dumber (1994)	2.697987	3.336595	0.638608
Longest Day, The (1962)	3.411765	4.031447	0.619682
Cable Guy, The (1996)	2.250000	2.863787	0.613787
Evil Dead II (Dead By Dawn) (1987)	3.297297	3.909283	0.611985
Hidden, The (1987)	3.137931	3.745098	0.607167
Rocky III (1982)	2.361702	2.943503	0.581801
Caddyshack (1980)	3.396135	3.969737	0.573602
For a Few Dollars More (1965)	3.409091	3.953795	0.544704

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'].std()

# Filter down to active titles
In [92]: rating_std_by_title = rating_std_by_title.loc[active_titles]

# 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.

14.3 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 dataset in illustrating data manipulation in R.

We need to do some data wrangling to load this dataset, but once we do that we will have a DataFrame that looks like this:

```
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 dataset:

- 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 whose popularity has advanced or declined the most
- 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

With the tools in this book, 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 from http://www.ssa.gov/oact/baby_names/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 an 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 already in 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 five 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 dataset 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`. So we now have a very large DataFrame containing all of the names data:

```
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 14-4](#)):

```
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')
```

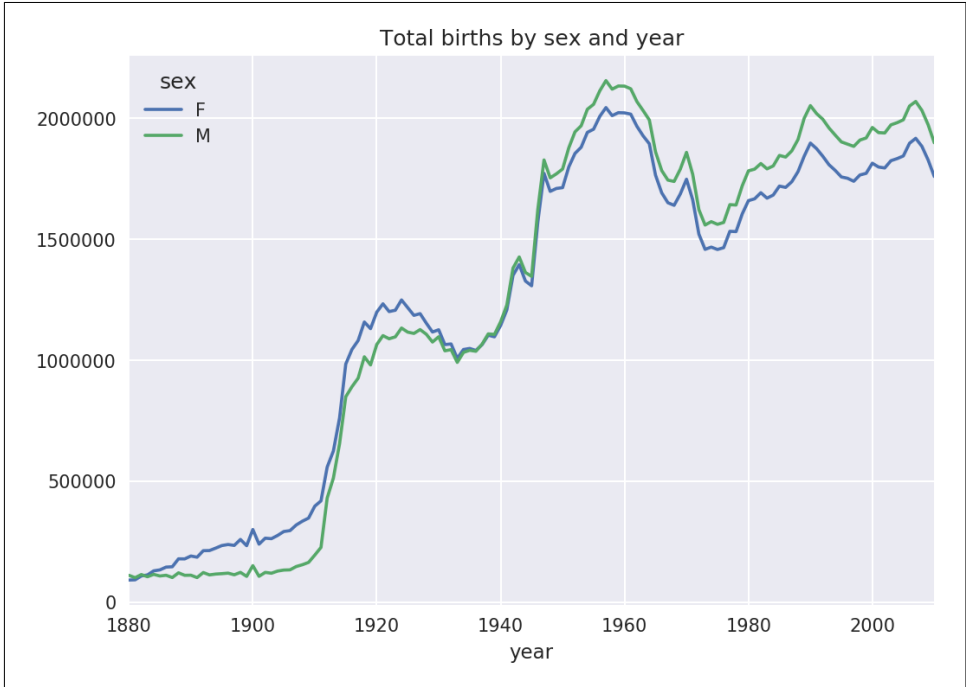


Figure 14-4. Total births by sex and year

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 were 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 dataset 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 1,000 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, try this instead:

```
pieces = []
for year, group in names.groupby(['year', 'sex']):
    pieces.append(group.sort_values(by='births', ascending=False)[:1000])
top1000 = pd.concat(pieces, ignore_index=True)
```

The resulting dataset 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 dataset in the following investigations into the data.

Analyzing Naming Trends

With the full dataset and Top 1,000 dataset 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 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='year',
.....:                                     columns='name',
.....:                                     aggfunc=sum)
```

Now, this can be plotted for a handful of names with DataFrame's plot method (Figure 14-5 shows the result):

```
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', 'Marilyn']]
```

```
In [114]: subset.plot(subplots=True, figsize=(12, 10), grid=False,
.....:                 title="Number of births per year")
```

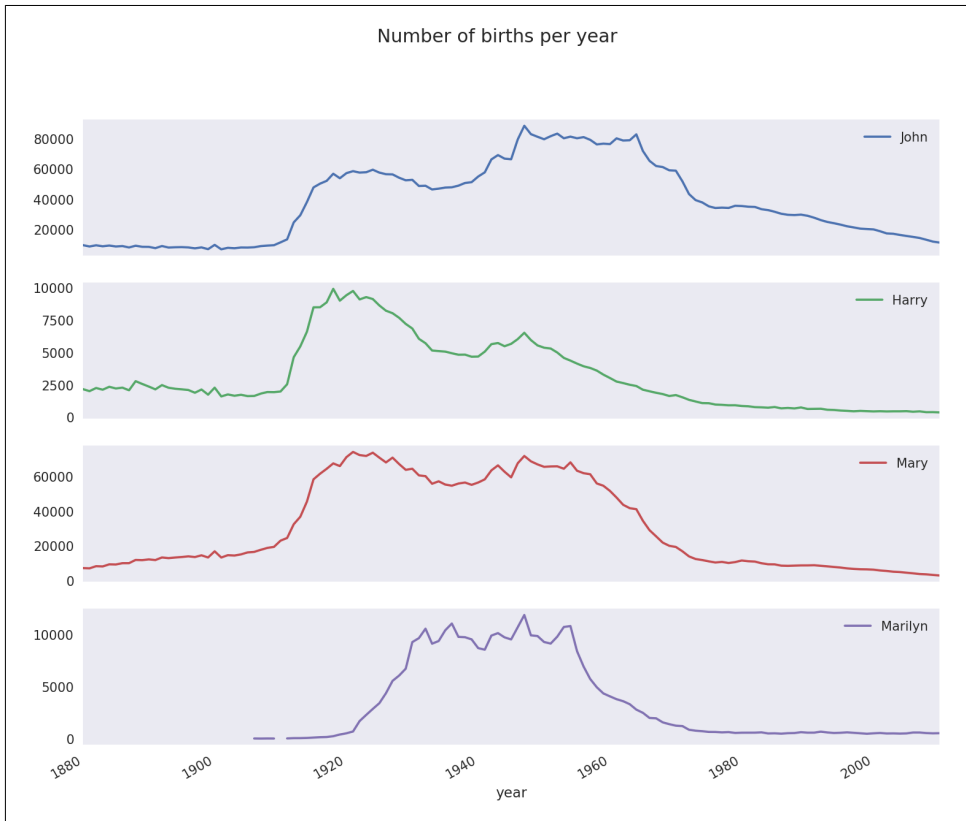


Figure 14-5. A few boy and girl names over time

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 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 1,000 most popular names, which I aggregate and plot by year and sex (Figure 14-6 shows the resulting plot):

```
In [116]: table = top1000.pivot_table('prop', index='year',
.....:                               columns='sex', aggfunc=sum)

In [117]: table.plot(title='Sum of table1000.prop by year and sex',
.....:               yticks=np.linspace(0, 1.2, 13), xticks=range(1880, 2020, 10)
.....:               )
```

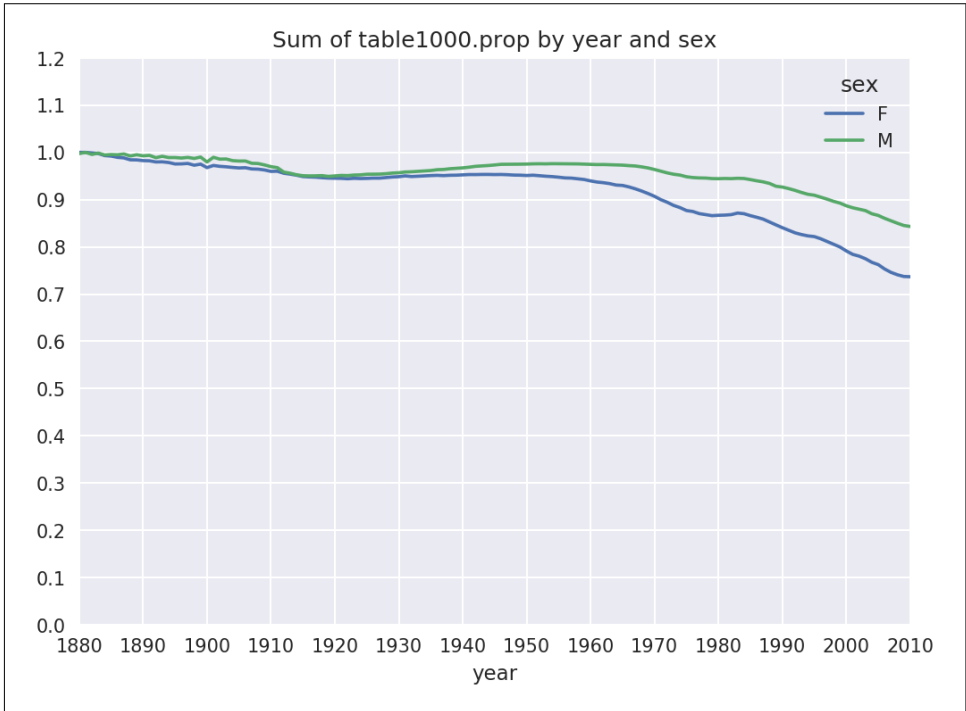


Figure 14-6. Proportion of births represented in top 1000 names by sex

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` and 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).prop.cumsum()
```

```
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).prop.cumsum()
```

```
In [125]: in1900.values.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 14-7):

```
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 50%")
```

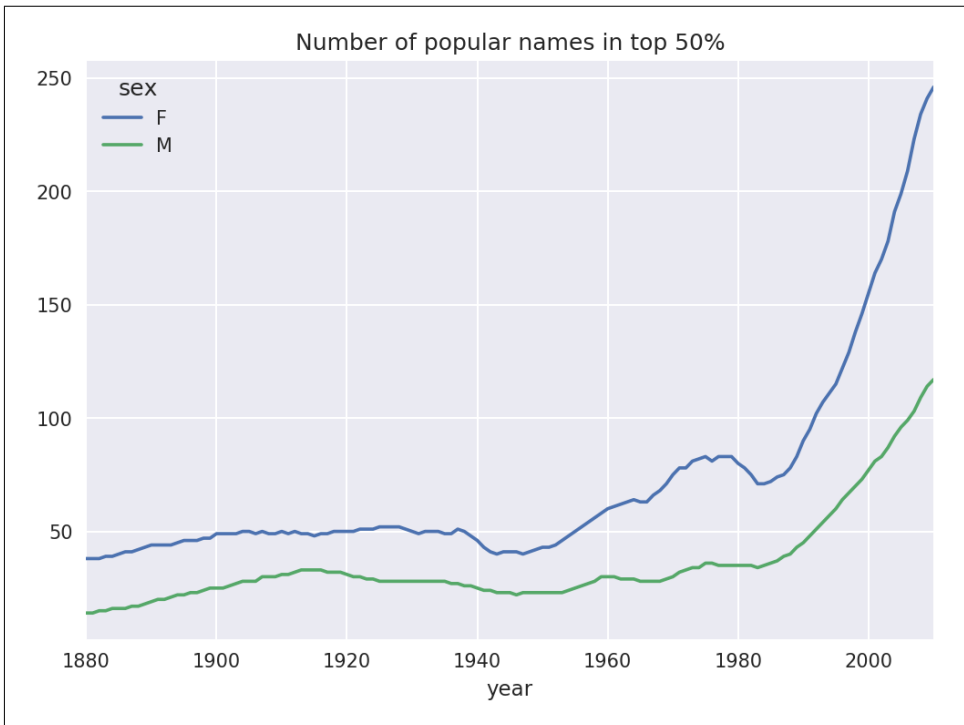


Figure 14-7. Plot of diversity metric by year

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 alternative spellings, is left to the reader.

The “last letter” revolution

In 2007, baby name researcher Laura Wattenberg pointed out on [her website](#) that the distribution of boy names by final letter has changed significantly over the last 100 years. To see this, we first aggregate all of the births in the full dataset 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 we select out three representative years spanning the history and print the first few rows:

```
In [131]: subtable = table.reindex(columns=[1910, 1960, 2010], level='year')
```

```
In [132]: subtable.head()
```

```
Out[132]:
```

sex	F			M		
year	1910	1960	2010	1910	1960	2010
last_letter						
a	108376.0	691247.0	670605.0	977.0	5204.0	28438.0
b	NaN	694.0	450.0	411.0	3912.0	38859.0
c	5.0	49.0	946.0	482.0	15476.0	23125.0
d	6750.0	3729.0	2607.0	2211.0	262112.0	44398.0
e	133569.0	435013.0	313833.0	28655.0	178823.0	129012.0

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	1910	396416.0
	1960	2022062.0
	2010	1759010.0
M	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	2010	1910	1960	2010
last_letter						
a	0.273390	0.341853	0.381240	0.005031	0.002440	0.014980

```

b          NaN    0.000343  0.000256  0.002116  0.001834  0.020470
c          0.000013  0.000024  0.000538  0.002482  0.007257  0.012181
d          0.017028  0.001844  0.001482  0.113858  0.122908  0.023387
e          0.336941  0.215133  0.178415  0.147556  0.083853  0.067959
...
v          NaN    0.000060  0.000117  0.000113  0.000037  0.001434
w          0.000020  0.000031  0.001182  0.006329  0.007711  0.016148
x          0.000015  0.000037  0.000727  0.003965  0.001851  0.008614
y          0.110972  0.152569  0.116828  0.077349  0.160987  0.058168
z          0.002439  0.000659  0.000704  0.000170  0.000184  0.001831
[26 rows x 6 columns]

```

With the letter proportions now in hand, we can make bar plots for each sex broken down by year (see [Figure 14-8](#)):

```

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_prop['F'].plot(kind='bar', rot=0, ax=axes[1], title='Female',
                      legend=False)

```

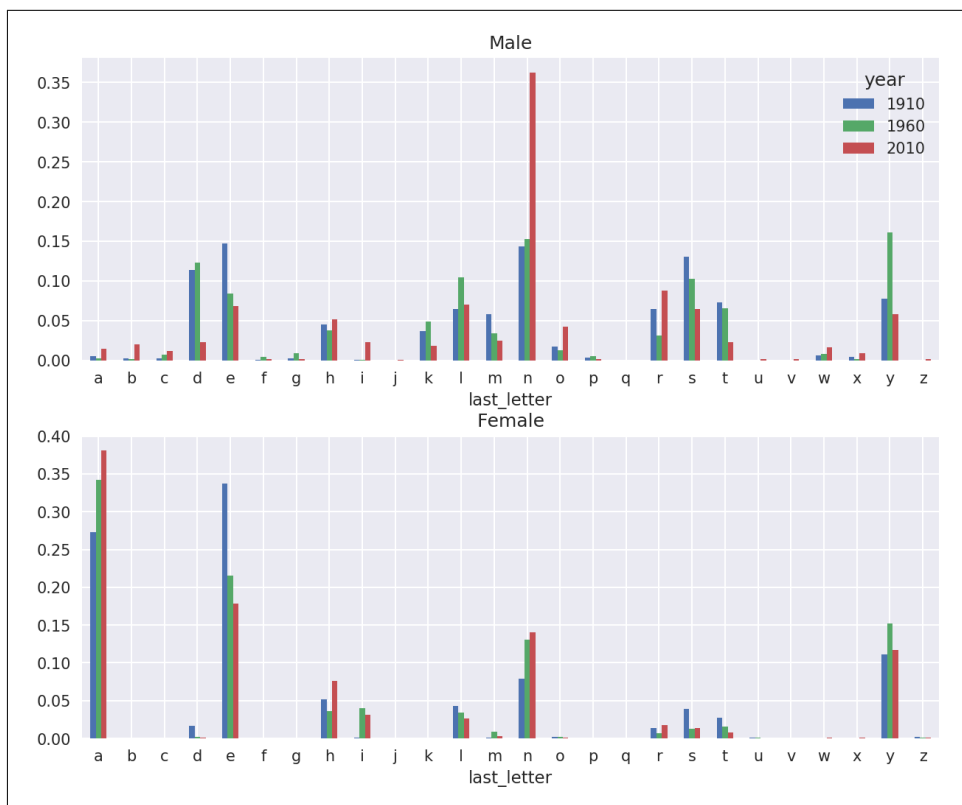


Figure 14-8. Proportion of boy and girl names ending in each letter

As you can see, boy names ending in *n* have experienced significant growth since the 1960s. Going back to the full table created before, 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 14-9](#)):

```
In [143]: dny_ts.plot()
```

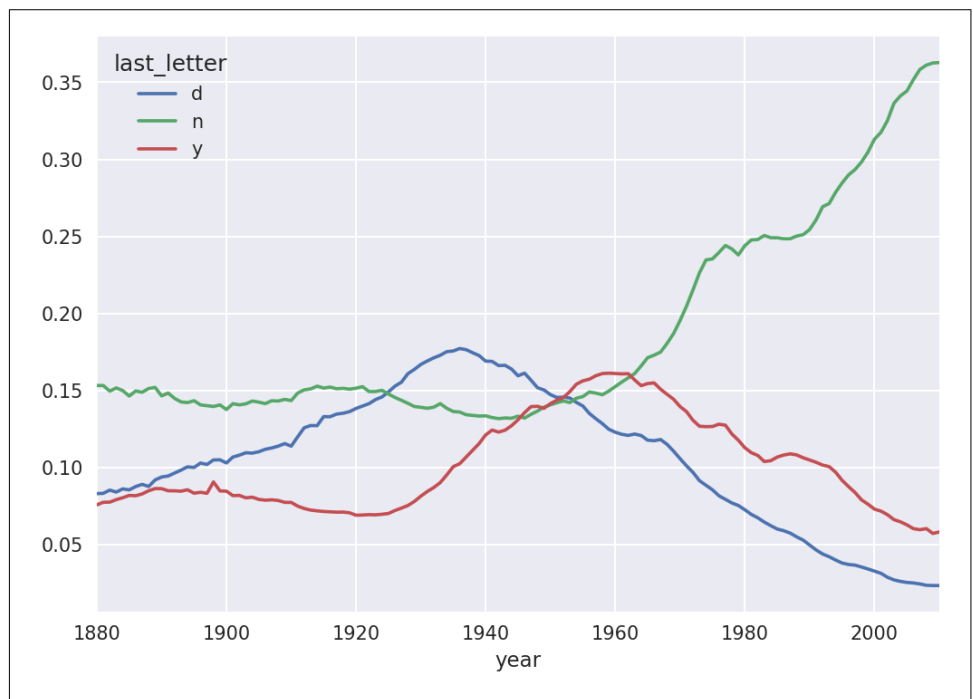


Figure 14-9. Proportion of boys born with names ending in *d/n/y* over time

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 DataFrame, 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     Leslee
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='sum')

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 14-10):

```
In [153]: table.plot(style={'M': 'k-', 'F': 'k--'})
```

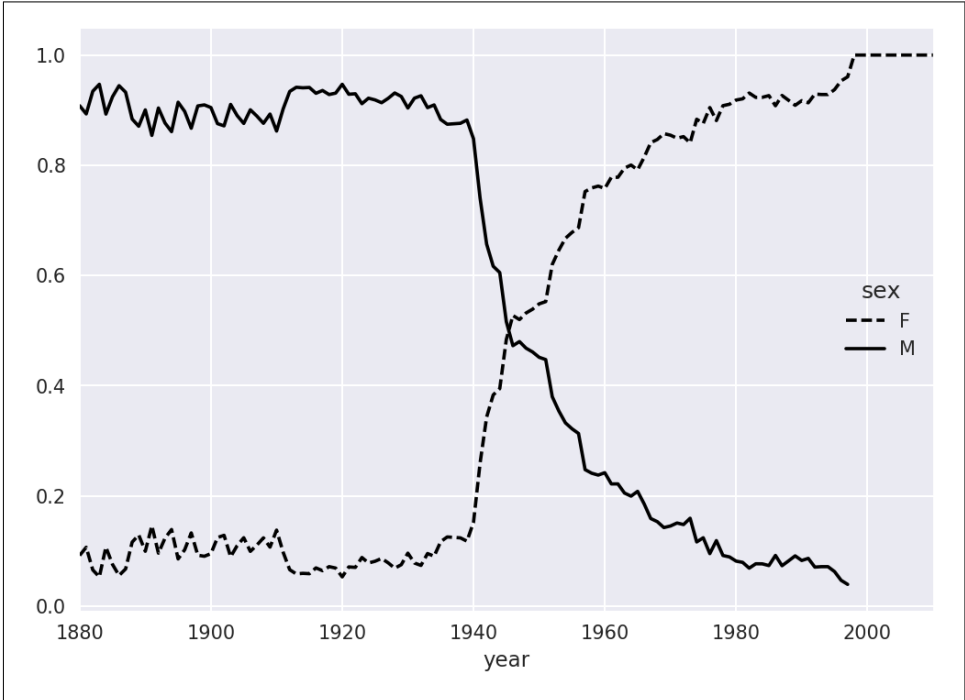


Figure 14-10. Proportion of male/female Lesley-like names over time

14.4 USDA Food Database

The US Department of Agriculture makes available a database of food nutrient information. Programmer Ashley Williams made available a version of this database in JSON format. The records look like this:

```
{  
  "id": 21441,  
  "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY,  
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. Data in this form is not particularly amenable to 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, 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', 'group', 'portions', '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	id
0	Cheese, caraway	Dairy and Egg Products	1008
1	Cheese, cheddar	Dairy and Egg Products	1009
2	Cheese, edam	Dairy and Egg Products	1018
3	Cheese, feta	Dairy and Egg Products	1019
4	Cheese, mozzarella, part skim milk manufacturer	Dairy and Egg Products	1028
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
Fast Foods	365
Legumes and Legume Products	365
Lamb, Veal, and Game Products	345
Sweets	341
Pork Products	328
Fruits and Fruit Juices	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	value	id
0	Protein	Composition	g	25.180	1008
1	Total lipid (fat)	Composition	g	29.200	1008
2	Carbohydrate, by difference	Composition	g	3.060	1008
3	Ash	Other	g	3.280	1008
4	Energy	Energy	kcal	376.000	1008
...
389350	Vitamin B-12, added	Vitamins	mcg	0.000	43546
389351	Cholesterol	Other	mg	0.000	43546
389352	Fatty acids, total saturated	Other	g	0.072	43546
389353	Fatty acids, total monounsaturated	Other	g	0.028	43546
389354	Fatty acids, total polyunsaturated	Other	g	0.041	43546

```
[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' are in both DataFrame objects, we can rename for clarity:

```
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=False)

In [175]: nutrients
Out[175]:

```

	nutrient	nutgroup	units	value	id
0	Protein	Composition	g	25.180	1008
1	Total lipid (fat)	Composition	g	29.200	1008
2	Carbohydrate, by difference	Composition	g	3.060	1008
3	Ash	Other	g	3.280	1008
4	Energy	Energy	kcal	376.000	1008
...
389350	Vitamin B-12, added	Vitamins	mcg	0.000	43546
389351	Cholesterol	Other	mg	0.000	43546
389352	Fatty acids, total saturated	Other	g	0.072	43546
389353	Fatty acids, total monounsaturated	Other	g	0.028	43546
389354	Fatty acids, total polyunsaturated	Other	g	0.041	43546

```

[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]:

```

	Glycine
nutrient	
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

```

We could now make a plot of median values by food group and nutrient type (see [Figure 14-11](#)):

```
In [180]: result = ndata.groupby(['nutrient', 'fgroup'])['value'].quantile(0.5)
```

```
In [181]: result['Zinc, Zn'].sort_values().plot(kind='barh')
```

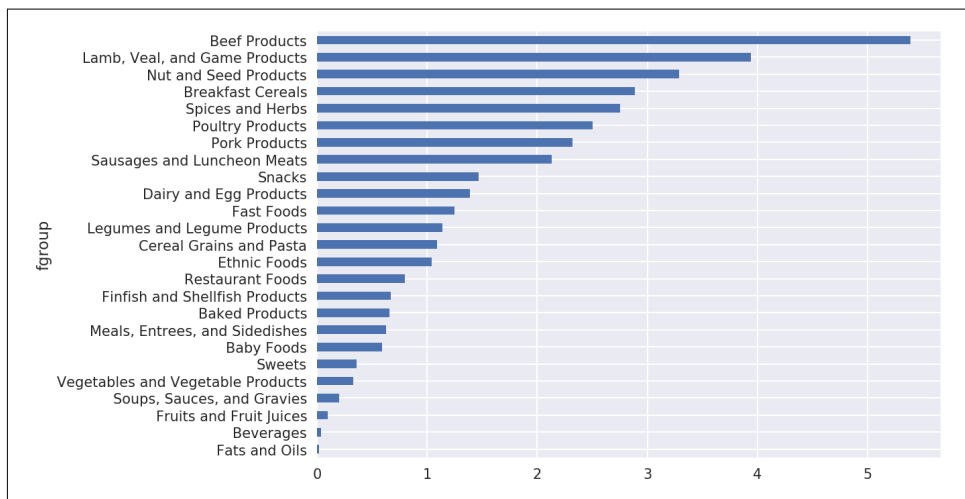


Figure 14-11. Median zinc values by nutrient group

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, low-fat
```

```
Aspartic acid   Soy protein isolate
```

```
Cystine         Seeds, cottonseed flour, low fat (glandless)
```

```
Glutamic acid   Soy protein isolate
```

```
...
```

```
Serine          Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
```

```
Threonine       Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
```

```
Tryptophan     Sea lion, Steller, meat with fat (Alaska Native)
```

```
Tyrosine        Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
```

```
Valine Soy protein isolate, PROTEIN TECHNOLOGIES INTE...
Name: food, Length: 19, dtype: object
```

14.5 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. A version of the dataset I downloaded in June 2012 is a 150 megabyte CSV file *P00000001-ALL.csv* (see the book's data repository), which can be loaded with `pd.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):
cmtc_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]:
cmtc_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`:

```
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, Herman',
      '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:¹

```
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
```

¹ This makes the simplifying assumption that Gary Johnson is a Republican even though he later became the Libertarian party candidate.

```
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 dataset to positive contributions:

```
In [196]: fec = fec[fec.contb_receipt_amt > 0]
```

Since Barack Obama and Mitt Romney were 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, Mitt'])]
```

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. The following 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_occupation',
.....:                                     columns='party', aggfunc='sum')

```

```

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 14-12):

```
In [205]: over_2mm.plot(kind='barh')
```

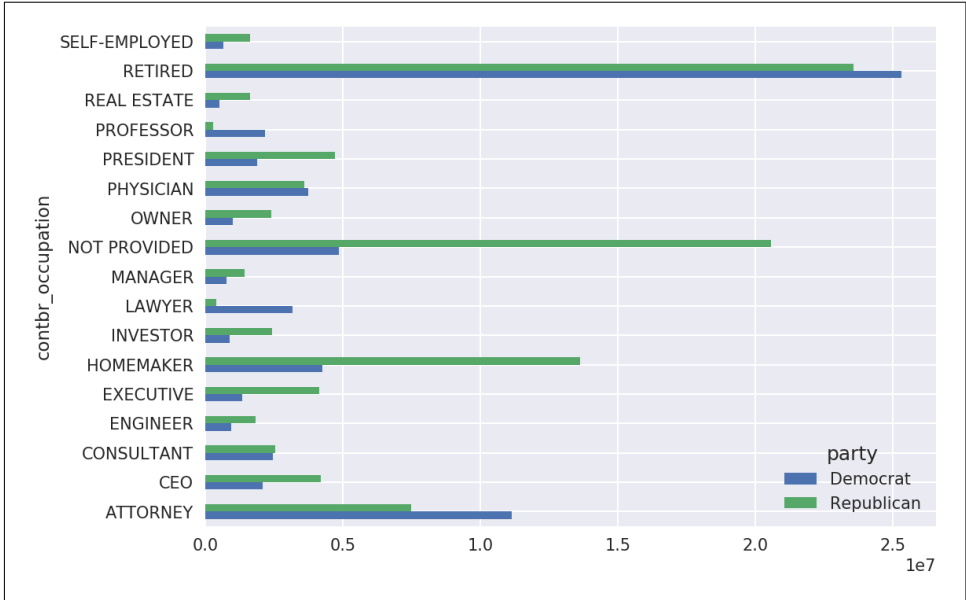


Figure 14-12. Total donations by party for top occupations

You might be interested in the top donor occupations or top companies that donated 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 aggregate by occupation and employer:

```
In [207]: grouped = fec_mrbo.groupby('cand_nm')

In [208]: grouped.apply(get_top_amounts, 'contbr_occupation', n=7)
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=10)

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, 1000000, 10000000])

```

```

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] < (100, 1000] < (1
000, 10000] <
                                     (10000, 100000] < (100000, 1000000] < (1000000,
10000000]]

```

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 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 (Figure 14-13 shows the resulting plot):

```
In [216]: bucket_sums = grouped.contb_receipt_amt.sum().unstack(0)
```

```
In [217]: normed_sums = bucket_sums.div(bucket_sums.sum(axis=1), axis=0)
```

```
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')
```

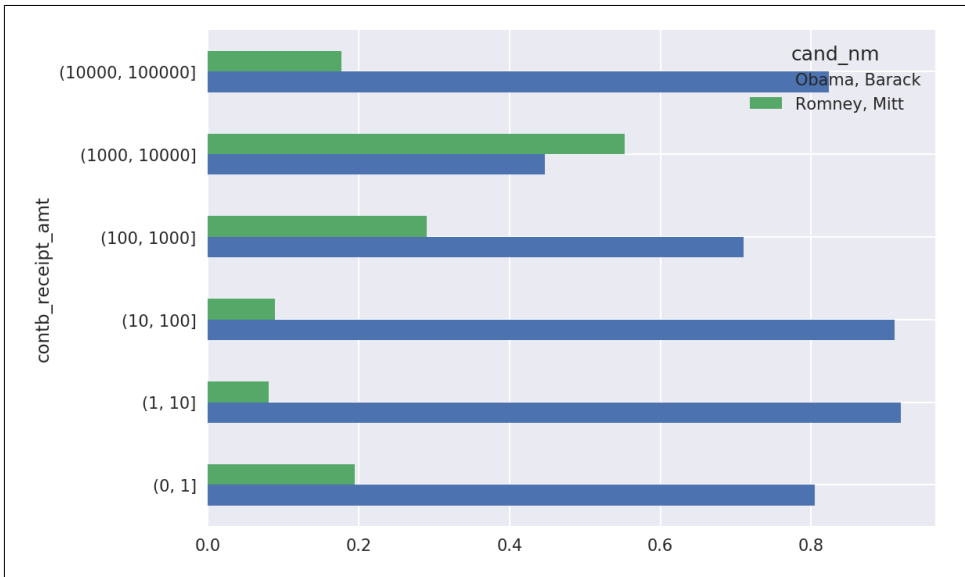


Figure 14-13. Percentage of total donations received by candidates for each donation size

I excluded the two largest bins as these are not donations by individuals.

This analysis can be refined and improved in many ways. 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 and explore the dataset 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).fillna(0)
```

```
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

14.6 Conclusion

We've reached the end of the book's main chapters. I have included some additional content you may find useful in the appendixes.

In the five years since the first edition of this book was published, Python has become a popular and widespread language for data analysis. The programming skills you have developed here will stay relevant for a long time into the future. I hope the programming tools and libraries we've explored serve you well in your work.

Advanced NumPy

In this appendix, 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.

This appendix contains miscellaneous topics and does not necessarily need to be read linearly.

A.1 ndarray Object Internals

The NumPy ndarray provides a means to interpret a block of homogeneous data (either contiguous or strided) 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 is that the ndarray is more than just a chunk of memory and a dtype; it also has “striding” information that 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

See [Figure A-1](#) for a simple mockup of the ndarray innards.

For example, a 10×5 array would have shape (10, 5):

```
In [10]: np.ones((10, 5)).shape
Out[10]: (10, 5)
```

A typical (C order) $3 \times 4 \times 5$ array of float64 (8-byte) values has strides (160, 40, 8) (knowing about the strides can be useful because, in general, the larger the strides on a particular axis, the more costly it is to perform computation along that axis):

```
In [11]: np.ones((3, 4, 5), dtype=np.float64).strides
Out[11]: (160, 40, 8)
```

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 (this would be the case, for example, in a slice like `obj[::-1]` or `obj[:, ::-1]`).

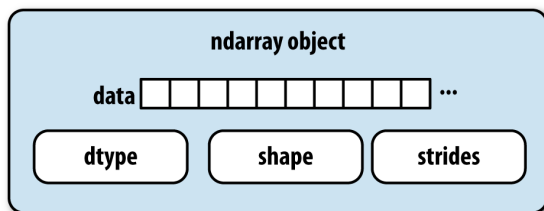


Figure A-1. The NumPy ndarray object

NumPy dtype Hierarchy

You may occasionally have code that 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 super-classes 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]
```

Therefore, we also have:

```
In [17]: np.issubdtype(ints.dtype, np.number)
Out[17]: True
```

Most NumPy users will never have to know about this, but it occasionally comes in handy. See [Figure A-2](#) for a graph of the dtype hierarchy and parent–subclass relationships.¹

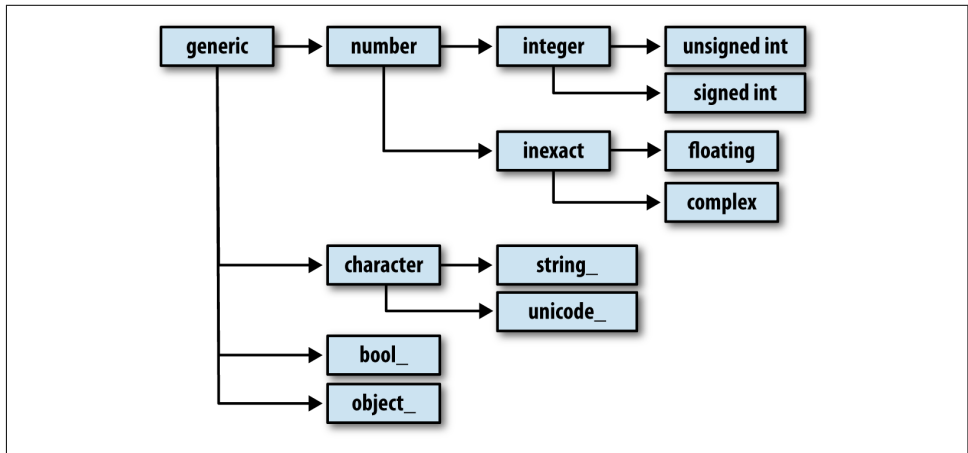


Figure A-2. The NumPy dtype class hierarchy

A.2 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.

¹ 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.

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 (the result is shown in [Figure A-3](#)):

```
In [18]: arr = np.arange(8)

In [19]: arr
Out[19]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [20]: arr.reshape((4, 2))
Out[20]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

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 A-3. Reshaping in C (row major) or Fortran (column major) order

A multidimensional array can also be reshaped:

```
In [21]: arr.reshape((4, 2)).reshape((2, 4))
Out[21]:
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 [22]: arr = np.arange(15)
```

```
In [23]: arr.reshape((5, -1))
```

```
Out[23]:
```

```
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 [24]: other_arr = np.ones((3, 5))
```

```
In [25]: other_arr.shape
```

```
Out[25]: (3, 5)
```

```
In [26]: arr.reshape(other_arr.shape)
```

```
Out[26]:
```

```
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 [27]: arr = np.arange(15).reshape((5, 3))
```

```
In [28]: arr
```

```
Out[28]:
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

```
In [29]: arr.ravel()
```

```
Out[29]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

ravel does not produce a copy of the underlying values if the values in the result were contiguous in the original array. The flatten method behaves like ravel except it always returns a copy of the data:

```
In [30]: arr.flatten()
```

```
Out[30]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

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 is usually set to 'C' or 'F' in most cases (there are also less commonly used options 'A' and 'K'; see the NumPy documentation, and refer back to [Figure A-3](#) for an illustration of these options):

```
In [31]: arr = np.arange(12).reshape((3, 4))
```

```
In [32]: arr
```

```
Out[32]:
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
In [33]: arr.ravel()
```

```
Out[33]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
In [34]: arr.ravel('F')
```

```
Out[34]: 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 (see [Figure A-3](#)). The key difference between C and Fortran order is the way 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 [35]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])
```

```
In [36]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])
```

```
In [37]: np.concatenate([arr1, arr2], axis=0)
Out[37]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
In [38]: np.concatenate([arr1, arr2], axis=1)
Out[38]:
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

There are some convenience functions, like `vstack` and `hstack`, for common kinds of concatenation. The preceding operations could have been expressed as:

```
In [39]: np.vstack((arr1, arr2))
Out[39]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
In [40]: np.hstack((arr1, arr2))
Out[40]:
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 [41]: arr = np.random.randn(5, 2)
```

```
In [42]: arr
Out[42]:
array([[ -0.2047,  0.4789],
       [-0.5194, -0.5557],
       [ 1.9658,  1.3934],
       [ 0.0929,  0.2817],
       [ 0.769 ,  1.2464]])
```

```
In [43]: first, second, third = np.split(arr, [1, 3])
```

```
In [44]: first
Out[44]: array([[ -0.2047,  0.4789]])
```

```
In [45]: second
Out[45]:
array([[ -0.5194, -0.5557],
       [ 1.9658,  1.3934]])
```

```
In [46]: third
Out[46]:
array([[ 0.0929,  0.2817],
       [ 0.769 ,  1.2464]])
```

The value `[1, 3]` passed to `np.split` indicate the indices at which to split the array into pieces.

See [Table A-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 A-1. Array concatenation functions

Function	Description
<code>concatenate</code>	Most general function, concatenates collection of arrays along one axis
<code>vstack</code> , <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> / <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 [47]: arr = np.arange(6)
```

```
In [48]: arr1 = arr.reshape((3, 2))
```

```
In [49]: arr2 = np.random.randn(3, 2)
```

```
In [50]: np.r_[arr1, arr2]
```

```
Out[50]:  
array([[ 0.    ,  1.    ],  
       [ 2.    ,  3.    ],  
       [ 4.    ,  5.    ],  
       [ 1.0072, -1.2962],  
       [ 0.275  ,  0.2289],  
       [ 1.3529,  0.8864]])
```

```
In [51]: np.c_[np.r_[arr1, arr2], arr]
```

```
Out[51]:  
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 [52]: np.c_[1:6, -10:-5]
```

```
Out[52]:
```



```
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

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 [53]: arr = np.arange(3)
```

```
In [54]: arr
Out[54]: array([0, 1, 2])
```

```
In [55]: arr.repeat(3)
Out[55]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```



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.

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 [56]: arr.repeat([2, 3, 4])
Out[56]: array([0, 0, 1, 1, 1, 2, 2, 2, 2])
```

Multidimensional arrays can have their elements repeated along a particular axis.

```
In [57]: arr = np.random.randn(2, 2)
```

```
In [58]: arr
Out[58]:
array([[ -2.0016, -0.3718],
       [ 1.669 , -0.4386]])
```

```
In [59]: arr.repeat(2, axis=0)
Out[59]:
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 [60]: arr.repeat([2, 3], axis=0)
```

```
Out[60]:
```

```
array([[ -2.0016,  -0.3718],
       [ -2.0016,  -0.3718],
       [  1.669 ,  -0.4386],
       [  1.669 ,  -0.4386],
       [  1.669 ,  -0.4386]])
```

```
In [61]: arr.repeat([2, 3], axis=1)
```

```
Out[61]:
```

```
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. Visually you can think of it as being akin to “laying down tiles”:

```
In [62]: arr
```

```
Out[62]:
```

```
array([[ -2.0016,  -0.3718],
       [  1.669 ,  -0.4386]])
```

```
In [63]: np.tile(arr, 2)
```

```
Out[63]:
```

```
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 [64]: arr
```

```
Out[64]:
```

```
array([[ -2.0016,  -0.3718],
       [  1.669 ,  -0.4386]])
```

```
In [65]: np.tile(arr, (2, 1))
```

```
Out[65]:
```

```
array([[ -2.0016,  -0.3718],
       [  1.669 ,  -0.4386],
       [ -2.0016,  -0.3718],
       [  1.669 ,  -0.4386]])
```

```
In [66]: np.tile(arr, (3, 2))
```

```
Out[66]:
```

```
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 [67]: arr = np.arange(10) * 100
```

```
In [68]: inds = [7, 1, 2, 6]
```

```
In [69]: arr[inds]
```

```
Out[69]: array([700, 100, 200, 600])
```

There are alternative ndarray methods that are useful in the special case of only making a selection on a single axis:

```
In [70]: arr.take(inds)
```

```
Out[70]: array([700, 100, 200, 600])
```

```
In [71]: arr.put(inds, 42)
```

```
In [72]: arr
```

```
Out[72]: array([  0, 42, 42, 300, 400, 500, 42, 42, 800, 900])
```

```
In [73]: arr.put(inds, [40, 41, 42, 43])
```

```
In [74]: arr
```

```
Out[74]: array([  0, 41, 42, 300, 400, 500, 43, 40, 800, 900])
```

To use `take` along other axes, you can pass the `axis` keyword:

```
In [75]: inds = [2, 0, 2, 1]
```

```
In [76]: arr = np.random.randn(2, 4)
```

```
In [77]: arr
```

```
Out[77]:
```

```
array([[ -0.5397,  0.477 ,  3.2489, -1.0212],  
       [-0.5771,  0.1241,  0.3026,  0.5238]])
```

```
In [78]: arr.take(inds, axis=1)
```

```
Out[78]:
```

```
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.

A.3 Broadcasting

Broadcasting describes how arithmetic works between arrays of different shapes. It can be a powerful feature, but one that can cause confusion, even for experienced users. The simplest example of broadcasting occurs when combining a scalar value with an array:

```
In [79]: arr = np.arange(5)

In [80]: arr
Out[80]: array([0, 1, 2, 3, 4])

In [81]: arr * 4
Out[81]: 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 [82]: arr = np.random.randn(4, 3)

In [83]: arr.mean(0)
Out[83]: array([-0.3928, -0.3824, -0.8768])

In [84]: demeaned = arr - arr.mean(0)

In [85]: demeaned
Out[85]:
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 [86]: demeaned.mean(0)
Out[86]: array([-0.,  0., -0.])
```

See [Figure A-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* (i.e., starting from the end) the axis lengths match or if either of the lengths is 1. Broadcasting is then performed over the missing or length 1 dimensions.

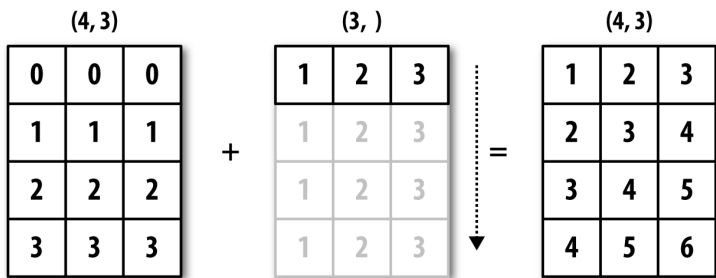


Figure A-4. Broadcasting over axis 0 with a 1D array

Even as an experienced NumPy user, I often find myself having to pause and draw a diagram as I 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 (i.e., subtract the row mean from each row), the smaller array must have shape (4, 1):

```
In [87]: arr
Out[87]:
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 [88]: row_means = arr.mean(1)
```

```
In [89]: row_means.shape
Out[89]: (4,)
```

```
In [90]: row_means.reshape((4, 1))
Out[90]:
array([[ 0.2104],
       [-1.6874],
       [-0.5222],
       [-0.2036]])
```

```
In [91]: demeaned = arr - row_means.reshape((4, 1))
```

```
In [92]: demeaned.mean(1)
```

```
Out[92]: array([ 0., -0.,  0.,  0.])
```

See [Figure A-5](#) for an illustration of this operation.

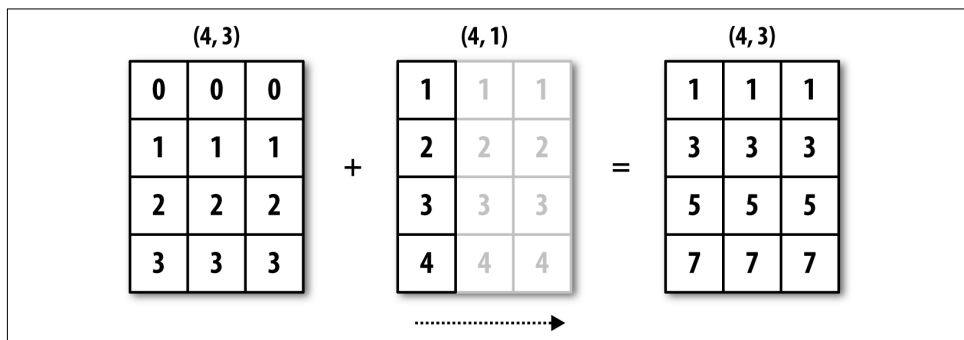


Figure A-5. Broadcasting over axis 1 of a 2D array

See [Figure A-6](#) for another illustration, this time adding a two-dimensional array to a three-dimensional one across axis 0.

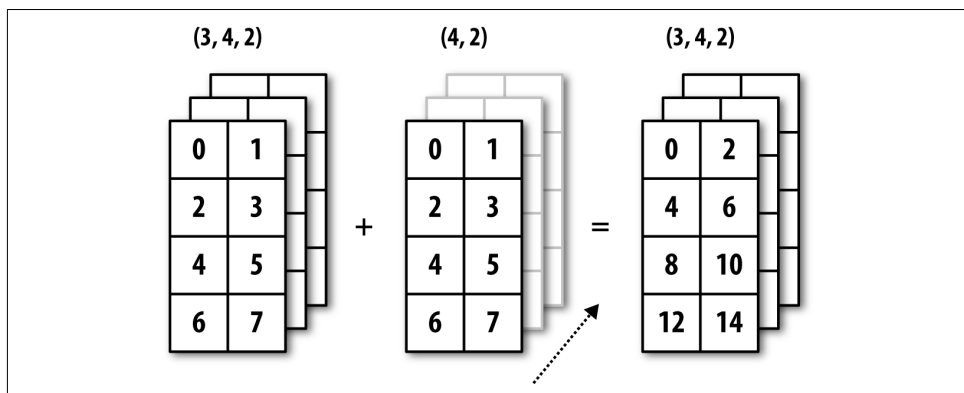


Figure A-6. Broadcasting over axis 0 of a 3D array

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 [93]: arr - arr.mean(1)
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-93-7b87b85a20b2> in <module>()  
----> 1 arr - arr.mean(1)  
ValueError: operands could not be broadcast together with shapes (4,3) (4,)
```

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 shown here, this meant reshaping the row means to be shape (4, 1) instead of (4,):

```
In [94]: arr - arr.mean(1).reshape((4, 1))  
Out[94]:  
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. [Figure A-7](#) nicely visualizes 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 [95]: arr = np.zeros((4, 4))
```

```
In [96]: arr_3d = arr[:, np.newaxis, :]
```

```
In [97]: arr_3d.shape
```

```
Out[97]: (4, 1, 4)
```

```
In [98]: arr_1d = np.random.normal(size=3)
```

```
In [99]: arr_1d[:, np.newaxis]
```

```
Out[99]:
```

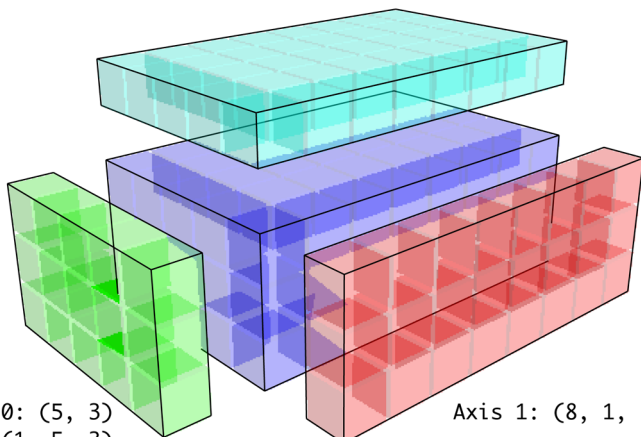
```
array([[ -2.3594],  
       [ -0.1995],  
       [ -1.542 ]])
```

```
In [100]: arr_1d[np.newaxis, :]
```

```
Out[100]: array([[ -2.3594, -0.1995, -1.542 ]])
```

Full array shape: (8, 5, 3)

Axis 2: (8, 5, 1)



Axis 0: (5, 3)
(1, 5, 3)

Axis 1: (8, 1, 3)

Figure A-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 [101]: arr = np.random.randn(3, 4, 5)
```

```
In [102]: depth_means = arr.mean(2)
```

```
In [103]: depth_means
```

```
Out[103]:
```

```
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 [104]: depth_means.shape
```

```
Out[104]: (3, 4)
```

```
In [105]: demeaned = arr - depth_means[:, :, np.newaxis]
```

```
In [106]: demeaned.mean(2)
```

```
Out[106]:
```

```
array([[ 0.,  0., -0., -0.],  
       [ 0.,  0., -0.,  0.],  
       [ 0.,  0., -0., -0.]])
```

You might be wondering 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 [107]: arr = np.zeros((4, 3))
```

```
In [108]: arr[:] = 5
```

```
In [109]: arr
Out[109]:
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 [110]: col = np.array([1.28, -0.42, 0.44, 1.6])
```

```
In [111]: arr[:] = col[:, np.newaxis]
```

```
In [112]: arr
Out[112]:
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 [113]: arr[:2] = [[-1.37], [0.509]]
```

```
In [114]: arr
Out[114]:
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 ]])
```

A.4 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 A-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 alternative way to sum elements in an array is to use `np.add.reduce`:

```
In [115]: arr = np.arange(10)
```

```
In [116]: np.add.reduce(arr)
```

```
Out[116]: 45
```

```
In [117]: arr.sum()
```

```
Out[117]: 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 [118]: np.random.seed(12346) # for reproducibility
```

```
In [119]: arr = np.random.randn(5, 5)
```

```
In [120]: arr[:, :2].sort(1) # sort a few rows
```

```
In [121]: arr[:, :-1] < arr[:, 1:]
```

```
Out[121]:
```

```
array([[ True,  True,  True,  True],
       [False,  True, False, False],
       [ True,  True,  True,  True],
       [ True, False,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
```

```
In [122]: np.logical_and.reduce(arr[:, :-1] < arr[:, 1:], axis=1)
```

```
Out[122]: 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 [123]: arr = np.arange(15).reshape((3, 5))
```

```
In [124]: np.add.accumulate(arr, axis=1)
```

```
Out[124]:  
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 [125]: arr = np.arange(3).repeat([1, 2, 2])
```

```
In [126]: arr
```

```
Out[126]: array([0, 1, 1, 2, 2])
```

```
In [127]: np.multiply.outer(arr, np.arange(5))
```

```
Out[127]:  
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 [128]: x, y = np.random.randn(3, 4), np.random.randn(5)
```

```
In [129]: result = np.subtract.outer(x, y)
```

```
In [130]: result.shape
```

```
Out[130]: (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. It accepts a sequence of “bin edges” that indicate how to split and aggregate the values:

```
In [131]: arr = np.arange(10)
```

```
In [132]: np.add.reduceat(arr, [0, 5, 8])
```

```
Out[132]: array([10, 18, 17])
```

The results are the reductions (here, sums) performed over `arr[0:5]`, `arr[5:8]`, and `arr[8:]`. As with the other methods, you can pass an axis argument:

```
In [133]: arr = np.multiply.outer(np.arange(4), np.arange(5))
```

```
In [134]: arr
```

```
Out[134]:  
array([[ 0,  0,  0,  0,  0],  
       [ 0,  1,  2,  3,  4],  
       [ 0,  2,  4,  6,  8],  
       [ 0,  3,  6,  9, 12]])
```

```
In [135]: np.add.reduceat(arr, [0, 2, 4], axis=1)
Out[135]:
array([[ 0,  0,  0],
       [ 1,  5,  4],
       [ 2, 10,  8],
       [ 3, 15, 12]])
```

See [Table A-2](#) for a partial listing of ufunc methods.

Table A-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> ; the resulting array has shape <code>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 [136]: def add_elements(x, y):
.....:     return x + y

In [137]: add_them = np.frompyfunc(add_elements, 2, 1)

In [138]: add_them(np.arange(8), np.arange(8))
Out[138]: 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 alternative (but slightly less featureful) function, `numpy.vectorize`, that allows you to specify the output type:

```
In [139]: add_them = np.vectorize(add_elements, otypes=[np.float64])

In [140]: add_them(np.arange(8), np.arange(8))
Out[140]: 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 [141]: arr = np.random.randn(10000)
```

```
In [142]: %timeit add_them(arr, arr)
```

```
4.12 ms +- 182 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [143]: %timeit np.add(arr, arr)
```

```
6.89 us +- 504 ns per loop (mean +- std. dev. of 7 runs, 100000 loops each)
```

Later in this chapter we'll show how to create fast ufuncs in Python using the [Numba project](#).

A.5 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 [144]: dtype = [('x', np.float64), ('y', np.int32)]
```

```
In [145]: sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=dtype)
```

```
In [146]: sarr
```

```
Out[146]:
```

```
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 [147]: sarr[0]
```

```
Out[147]: ( 1.5, 6)
```

```
In [148]: sarr[0]['y']
```

```
Out[148]: 6
```

The field names are stored in the dtype.names attribute. When you access a field on the structured array, a strided view on the data is returned, thus copying nothing:

```
In [149]: sarr['x']
```

```
Out[149]: 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 [150]: dtype = [('x', np.int64, 3), ('y', np.int32)]
```

```
In [151]: arr = np.zeros(4, dtype=dtype)
```

```
In [152]: arr
```

```
Out[152]:
```

```
array([(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 3 for each record:

```
In [153]: arr[0]['x']
```

```
Out[153]: 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 [154]: arr['x']
```

```
Out[154]:
```

```
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 [155]: dtype = [('x', [(('a', 'f8'), ('b', 'f4'))], ('y', np.int32)]
```

```
In [156]: data = np.array([(1, 2), 5], [(3, 4), 6], dtype=dtype)
```

```
In [157]: data['x']
```

```
Out[157]:
```

```
array([(1., 2.), (3., 4.)],  
      dtype=[('a', '<f8'), ('b', '<f4')])
```

```
In [158]: data['y']
```

```
Out[158]: array([5, 6], dtype=int32)
```

```
In [159]: data['x']['a']
```

```
Out[159]: array([1., 3.])
```

pandas DataFrame 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), transporting it over the network, and other such uses.

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 with `np.fromfile`. Specialized uses like this are beyond the scope of this book, but it's worth knowing that such things are possible.

A.6 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 [160]: arr = np.random.randn(6)
```

```
In [161]: arr.sort()
```

```
In [162]: arr
```

```
Out[162]: array([-1.082 ,  0.3759,  0.8014,  1.1397,  1.2888,  1.8413])
```

When sorting arrays in-place, remember that if the array is a view on a different `ndarray`, the original array will be modified:

```
In [163]: arr = np.random.randn(3, 5)
```

```
In [164]: arr
```

```
Out[164]:
```

```
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 [165]: arr[:, 0].sort() # Sort first column values in-place
```

```
In [166]: arr
```

```
Out[166]:
```

```
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`) as `ndarray.sort`:

```
In [167]: arr = np.random.randn(5)
```

```
In [168]: arr
```

```
Out[168]: array([-1.1181, -0.2415, -2.0051,  0.7379, -1.0614])
```

```
In [169]: np.sort(arr)
Out[169]: array([-2.0051, -1.1181, -1.0614, -0.2415,  0.7379])
```

```
In [170]: arr
Out[170]: 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 [171]: arr = np.random.randn(3, 5)
```

```
In [172]: arr
Out[172]:
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 [173]: arr.sort(axis=1)
```

```
In [174]: arr
Out[174]:
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 [175]: arr[:, ::-1]
Out[175]:
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 you may need to reorder datasets 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 of 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 [176]: values = np.array([5, 0, 1, 3, 2])
```

```
In [177]: indexer = values.argsort()
```



```
In [178]: indexer
Out[178]: array([1, 2, 4, 3, 0])
```

```
In [179]: values[indexer]
Out[179]: array([0, 1, 2, 3, 5])
```

As a more complicated example, this code reorders a two-dimensional array by its first row:

```
In [180]: arr = np.random.randn(3, 5)
```

```
In [181]: arr[0] = values
```

```
In [182]: arr
Out[182]:
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 [183]: arr[:, arr[0].argsort()]
Out[183]:
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 [184]: first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill', 'Barbara'])
```

```
In [185]: last_name = np.array(['Jones', 'Arnold', 'Arnold', 'Jones', 'Walters'])
```

```
In [186]: sorter = np.lexsort((first_name, last_name))
```

```
In [187]: sorter
Out[187]: array([1, 2, 3, 0, 4])
```

```
In [188]: zip(last_name[sorter], first_name[sorter])
Out[188]: <zip at 0x7fa203eda1c8>
```

`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`.



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).

Alternative 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 [189]: values = np.array(['2:first', '2:second', '1:first', '1:second',  
.....:                      '1:third'])
```

```
In [190]: key = np.array([2, 2, 1, 1, 1])
```

```
In [191]: indexer = key.argsort(kind='mergesort')
```

```
In [192]: indexer  
Out[192]: array([2, 3, 4, 0, 1])
```

```
In [193]: values.take(indexer)  
Out[193]:  
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 A-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 it's useful to know that it's there.

Table A-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 [194]: np.random.seed(12345)
```

```
In [195]: arr = np.random.randn(20)
```

```
In [196]: arr  
Out[196]:  
array([-0.2047,  0.4789, -0.5194, -0.5557,  1.9658,  1.3934,  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])
```

```
In [197]: np.partition(arr, 3)
```

```
Out[197]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386, -0.2047,
        0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  0.275 ,  0.2289,
        1.3529,  0.8864,  1.3934,  1.9658,  1.669 ,  1.2464])
```

After you call `partition(arr, 3)`, the first three elements in the result are the smallest three values in no particular order. `numpy.argpartition`, similar to `numpy.argsort`, returns the indices that rearrange the data into the equivalent order:

```
In [198]: indices = np.argpartition(arr, 3)
```

```
In [199]: indices
```

```
Out[199]:
```

```
array([16, 11,  3,  2, 17, 19,  0,  7,  8,  1, 10,  6, 12, 13, 14, 15,  5,
        4, 18,  9])
```

```
In [200]: arr.take(indices)
```

```
Out[200]:
```

```
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386, -0.2047,
        0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  0.275 ,  0.2289,
        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 [201]: arr = np.array([0, 1, 7, 12, 15])
```

```
In [202]: arr.searchsorted(9)
```

```
Out[202]: 3
```

You can also pass an array of values to get an array of indices back:

```
In [203]: arr.searchsorted([0, 8, 11, 16])
```

```
Out[203]: 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 [204]: arr = np.array([0, 0, 0, 1, 1, 1, 1])
```

```
In [205]: arr.searchsorted([0, 1])
```

```
Out[205]: array([0, 3])
```

```
In [206]: arr.searchsorted([0, 1], side='right')
```

```
Out[206]: 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 [207]: data = np.floor(np.random.uniform(0, 10000, size=50))

In [208]: bins = np.array([0, 100, 1000, 5000, 10000])

In [209]: data
Out[209]:
array([[ 9940.,  6768.,  7908.,  1709.,   268.,  8003.,  9037.,   246.,
    4917.,  5262.,  5963.,   519.,  8950.,  7282.,  8183.,  5002.,
    8101.,   959.,  2189.,  2587.,  4681.,  4593.,  7095.,  1780.,
    5314.,  1677.,  7688.,  9281.,  6094.,  1501.,  4896.,  3773.,
    8486.,  9110.,  3838.,  3154.,  5683.,  1878.,  1258.,  6875.,
    7996.,  5735.,  9732.,  6340.,  8884.,  4954.,  3516.,  7142.,
    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 [210]: labels = bins.searchsorted(data)

In [211]: labels
Out[211]:
array([[4, 4, 4, 3, 2, 4, 4, 2, 3, 4, 4, 2, 4, 4, 4, 4, 4, 2, 3, 3, 3, 3, 4,
    3, 4, 3, 4, 4, 4, 3, 3, 3, 4, 4, 3, 3, 4, 3, 3, 4, 4, 4, 4, 4, 3,
    3, 4, 4, 3])
```

This, combined with `pandas`’s `groupby`, can be used to bin data:

```
In [212]: pd.Series(data).groupby(labels).mean()
Out[212]:
2    498.000000
3   3064.277778
4   7389.035714
dtype: float64
```

A.7 Writing Fast NumPy Functions with Numba

Numba is an open source project that creates fast functions for NumPy-like data using CPUs, GPUs, or other hardware. It uses the **LLVM Project** 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 cannot compile arbitrary Python code, but it supports a significant subset of pure Python that is most useful for writing numerical algorithms.

Numba is a 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 previous 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](#). The next section shows an example of 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.])
```

A.8 Advanced Array Input and Output

In [Chapter 4](#), we became acquainted with `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 datasets 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 it is stored in 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 [214]: mmap = np.memmap('mymmap', dtype='float64', mode='w+',
.....:                    shape=(10000, 10000))
```

```
In [215]: mmap
```

```
Out[215]:
```

```
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 [216]: section = mmap[:5]
```

If you assign data to these, it will be buffered in memory (like a Python file object), but you can write it to disk by calling `flush`:

```
In [217]: section[:] = np.random.randn(5, 10000)
```

```
In [218]: mmap.flush()
```

```
In [219]: mmap
```

```
Out[219]:
```

```
memmap([[ 0.7584, -0.6605,  0.8626, ...,  0.6046, -0.6212,  2.0542],
         [-1.2113, -1.0375,  0.7093, ..., -1.4117, -0.1719, -0.8957],
         [-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 [220]: 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 [221]: mmap = np.memmap('mymmap', dtype='float64', shape=(10000, 10000))
```

```
In [222]: mmap
```

```
Out[222]:
```

```
memmap([[ 0.7584, -0.6605,  0.8626, ...,  0.6046, -0.6212,  2.0542],
         [-1.2113, -1.0375,  0.7093, ..., -1.4117, -0.1719, -0.8957],
         [-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. ]])
```

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 recommend reading the [pandas online documentation](#).

A.9 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. The following list briefly summarizes some things to keep in mind:

- Convert Python loops and conditional logic to array operations and boolean array operations
- Use broadcasting whenever possible
- Use arrays views (slicing) to avoid copying data
- Utilize ufuncs and ufunc methods

If you can't get the performance you require after exhausting the capabilities provided by NumPy alone, consider writing code in C, Fortran, or Cython. I use [Cython](#) frequently 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 (e.g., 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 [225]: arr_c = np.ones((1000, 1000), order='C')
```

```
In [226]: arr_f = np.ones((1000, 1000), order='F')
```

```
In [227]: arr_c.flags
```

```
Out[227]:
```

```
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

```
In [228]: arr_f.flags
```

```
Out[228]:
```

```
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

```
In [229]: arr_f.flags.f_contiguous
```

```
Out[229]: 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 [230]: %timeit arr_c.sum(1)
```

```
784 us +- 10.4 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)
```

```
In [231]: %timeit arr_f.sum(1)
```

```
934 us +- 29 us per loop (mean +- std. dev. of 7 runs, 1000 loops each)
```

When you're 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 [232]: arr_f.copy('C').flags
```

```
Out[232]:
```

```
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 [233]: arr_c[:50].flags.contiguous
Out[233]: True
```

```
In [234]: arr_c[:, :50].flags
Out[234]:
```

```
C_CONTIGUOUS : False
F_CONTIGUOUS : False
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

More on the IPython System

In [Chapter 2](#) we looked at the basics of using the IPython shell and Jupyter notebook. In this chapter, we explore some deeper functionality in the IPython system that can either be used from the console or within Jupyter.

B.1 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 and 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 the up arrow key 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 the down arrow key. 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 and 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 [24]: 2 ** 27
Out[24]: 134217728

In [25]: _
Out[25]: 134217728
```

Input variables are stored in variables named like `_iX`, where X is the input line number. For each input variable 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 they can be executed again with the Python `exec` keyword:

```
In [30]: exec(_i27)
```

Here `_i27` refers to the code input in In [27].

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.



When working with very large datasets, 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.

B.2 Interacting with the Operating System

Another feature of IPython is that it allows you to seamlessly access the filesystem 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, macOS) 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 B-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 B-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

Command	Description
%dhist	Print the history of visited directories
%env	Return the system environment variables as a dict
%matplotlib	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.

You can store the console output of a shell command in a variable by assigning the expression escaped with `!` 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/
```

You can execute multiple commands just as on the command line by separating them with semicolons:

```
In [558]: %alias test_alias (cd examples; 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.

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 you've done this, when we use the `%cd` magic, we can use any bookmarks we'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.

B.3 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 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:

```
In [3]: %debug
```

```
> /home/wesm/code/pydata-book/examples/ipython_bug.py(9)throws_an_exception()  
     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/examples/ipython_bug.py(13)calling_things()  
    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 examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()
```

```
ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/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 preceding 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/examples/ipython_bug.py(12)calling_things()
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/examples/ipython_bug.py(13)calling_things()
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/examples/ipython_bug.py(6)throws_an_exception()
5
```

```

----> 6 def throws_an_exception():
      7     a = 5

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(7)throws_an_exception()
  6 def throws_an_exception():
----> 7     a = 5
      8     b = 6

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(8)throws_an_exception()
      7     a = 5
----> 8     b = 6
      9     assert(a + b == 10)

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py(9)throws_an_exception()
      8     b = 6
----> 9     assert(a + b == 10)
     10

ipdb> !a
5
ipdb> !b
6

```

Developing proficiency with the interactive debugger is largely a matter of practice and experience. See [Table B-2](#) for a full catalog of the debugger commands. If you are accustomed to using 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 B-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 <i>path/to/file.py:number</i>	Set breakpoint at line <i>number</i> in specified file
s(step)	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):

```
from IPython.core.debugger import Pdb

def set_trace():
    Pdb(color_scheme='Linux').set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):
    pdb = Pdb(color_scheme='Linux')
    return pdb.runcall(f, *args, **kwargs)
```

The first function, `set_trace`, is very simple. You can use a `set_trace` in any part of your code that you want to temporarily stop in order to more closely examine it (e.g., right before an exception occurs):

```
In [7]: run examples/ipython_bug.py
> /home/wesm/code/pydata-book/examples/ipython_bug.py(16)calling_things()
   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 we just looked at enables you to invoke the interactive debugger easily on an arbitrary function call. Suppose we had written a function like the following and we wished to step through its logic:

```
def f(x, y, z=1):
    tmp = x + y
    return tmp / z
```

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 examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
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 examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:2
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> c
> /home/wesm/code/pydata-book/examples/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 (billionths 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 preceding 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 that does some linear algebra in a loop (computing the maximum absolute eigenvalues of a series of 100×100 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)
```

You can run this script through cProfile using the following in the command line:

```
python -m cProfile cprof_example.py
```

If you try that, you'll find that the output is 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_lite.dgeev}
  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:3178(add_newdoc)
100  0.003  0.000  0.005  0.000 linalg.py:162(_assertFinite)
...

```

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 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(run_experiment)
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_lite.dgeev}
100     0.058   0.001   0.058   0.001  {method 'randn'}
100     0.003   0.000   0.005   0.000  linalg.py:162(_assertFinite)
200     0.002   0.000   0.002   0.000  {method 'all' of 'numpy.ndarray'}
```

Similarly, calling `%run -p -s cumulative cprof_example.py` has the same effect as the command-line approach, 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?”

There are other tools available that help make profiles easier to understand when you are using IPython or Jupyter. One of these is [SnakeViz](#), which produces an interactive visualization of the profile results using d3.js.

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(function)
      1   0.036   0.036   0.046   0.046  prof_mod.py:3(add_and_sum)
      1   0.009   0.009   0.009   0.009  {method 'sum' of 'numpy.ndarray'}
      1   0.003   0.003   0.049   0.049  <string>:1(<module>)

```

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					<code>def add_and_sum(x, y):</code>
4	1	36510	36510.0	79.5	<code>added = x + y</code>
5	1	9425	9425.0	20.5	<code>summed = added.sum(axis=1)</code>
6	1	1	1.0	0.0	<code>return summed</code>

This can be much easier to interpret. In this case we profiled the same function we used in the statement. Looking at the preceding module code, 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					<code>def add_and_sum(x, y):</code>
4	1	4375	4375.0	79.2	<code>added = x + y</code>
5	1	1149	1149.0	20.8	<code>summed = added.sum(axis=1)</code>
6	1	2	2.0	0.0	<code>return summed</code>

```

File: prof_mod.py
Function: call_function at line 8
Total time: 0.121016 s

```

Line #	Hits	Time	Per Hit	% Time	Line Contents
8					<code>def call_function():</code>
9	1	57169	57169.0	47.2	<code>x = randn(1000, 1000)</code>
10	1	58304	58304.0	48.2	<code>y = randn(1000, 1000)</code>
11	1	5543	5543.0	4.6	<code>return add_and_sum(x, y)</code>

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.



The reason that you must explicitly specify the names of the functions you want to profile with `%lprun` is that the overhead of “tracing” the execution time of each line is substantial. Tracing functions that are not of interest has the potential to significantly alter the profile results.

B.4 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.

Therefore, 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 IPython code development 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.py` because of Python’s “load-once” module system. This behavior differs from some

other data analysis environments, like MATLAB, which automatically propagate code changes.¹ To cope with this, you have a couple of options. The first way is to use the `reload` function in the `importlib` module in the standard library:

```
import some_lib
import importlib

importlib.reload(some_lib)
```

This guarantees that you will get a fresh copy of `some_lib.py` 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 `some_lib.py` 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 go wrong if we were to run this program 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

¹ 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.

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.

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 headaches in general than 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.

B.5 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 preceding 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 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 (e.g., 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 alternative IPython configuration tailored for a particular application or project. Creating a new profile is as simple as typing something like the following:

```
ipython profile create secret_project
```

Once you've done this, edit the config files in the newly created *profile_secret_project* directory and then launch IPython like so:

```
$ ipython --profile=secret_project
Python 3.5.1 | packaged by conda-forge | (default, May 20 2016, 05:22:56)
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 details.
```

```
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
```

B.6 Conclusion

As you work through the code examples in this 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 that 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](#).

Index

Symbols

- ! (exclamation point), 486
- != operator, 38, 100, 108
- # (hash mark), 31
- % (percent sign), 28, 495
- %matplotlib magic function, 254
- & operator, 37, 65, 66, 101
- &= operator, 66
- () (parentheses), 32, 51
- * (asterisk), 24
- * operator, 37
- ** operator, 37
- + operator, 37, 52, 56
- operator, 37, 66
- = operator, 66
- . (period), 21
- / operator, 37
- // operator, 37, 39
- : (colon), 31
- ; (semicolon), 31
- < operator, 38, 108
- <= operator, 38, 108
- == operator, 38, 108
- > operator, 38, 108
- >= operator, 38, 108
- >>> prompt, 16
- ? (question mark), 23-24
- @ symbol, 116
- [] (square brackets), 52, 54
- \ (backslash), 41, 216
- ^ operator, 37, 66
- ^= operator, 66
- _ (underscore), 22, 54, 451, 484
- { } (curly braces), 61, 65
- | operator, 37, 65-66, 101
- |= operator, 66
- ~ operator, 101

A

- %a datetime format, 321
- %A datetime format, 321
- a(args) debugger command, 490
- abs function, 107, 121
- accumulate method, 466
- accumulations, 159
- add binary function, 107
- add method, 66, 149
- add_categories method, 372
- add_constant function, 394
- add_patch method, 266
- add_subplot method, 255
- aggfunc method, 315
- aggregate (agg) method, 297, 374
- aggregations (reductions), 111
- %alias magic function, 485-486
- all method, 113, 466
- and keyword, 21, 43, 101
- annotate function, 265
- annotating in matplotlib, 265-267
- anonymous (lambda) functions, 73
- any built-in function, 21
- any method, 113, 122, 206
- Apache Parquet format, 186
- APIs, pandas interacting with, 187
- append method, 55, 136
- append mode for files, 82
- apply method, 152, 164, 302-312, 373-376
- applymap method, 153

- arange function, 14, 90
- arccos function, 107
- arccosh function, 107
- arcsin function, 107
- arcsinh function, 107
- arctan function, 107
- arctanh function, 107
- argmax method, 112, 121, 160
- argmin method, 112, 160
- argspartition method, 474
- argsort method, 472, 475
- arithmetic operations
 - between DataFrame and Series, 149
 - between objects with different indexes, 146
 - on date and time periods, 339-347
 - with fill values, 148
 - with NumPy arrays, 93
- array function, 88, 90
- arrays (see ndarray object)
- arrow function, 265
- as keyword, 36
- asarray function, 90
- asfreq method, 340, 352
- assign method, 379
- associative arrays (see dicts)
- asterisk (*), 24
- astype method, 92
- as_ordered method, 372
- as_ordered method, 367
- as_unordered method, 372
- attributes
 - for data types, 469
 - for ndarrays, 89, 453, 463, 481
 - hidden, 22
 - in DataFrame data structure, 130
 - in Python, 35, 161
 - in Series data structure, 127
- automagic feature, 29
- %automagic magic function, 29
- average method, 156
- axes
 - broadcasting over, 462
 - concatenating along, 227, 236-241
 - renaming indexes for, 201
 - selecting indexes with duplicate labels, 157
 - swapping in arrays, 103
- AxesSubplot object, 256, 262
- axis method, 159

B

- %b datetime format, 321
- %B datetime format, 321
- b(reak) debugger command, 490
- backslash (\), 41, 216
- bang (!), 486
- bar method, 272
- bar plots, 272-277
- barh method, 272
- barplot function, 277
- base frequency, 330
- bcolz binary format, 184
- beta function, 119
- binary data formats
 - about, 183
 - binary mode for files, 82-83
 - HDF5 format, 184-186
 - Microsoft Excel files, 186-187
- binary moving window functions, 359
- binary operators and comparisons in Python, 36, 65
- binary searches of lists, 57
- binary universal functions, 106, 107
- binding, defined, 33, 236
- binning continuous data, 203
- binomial function, 119
- bisect module, 57
- Bitly dataset example, 403-413
- Blosc compression library, 184
- Bokeh tool, 285
- %bookmark magic function, 485, 487
- bookmarking directories in IPython, 487
- bool data type, 39, 43, 91
- bool function, 43
- boolean arrays, 113
- boolean indexing, 99-102
- braces {}, 61, 65
- break keyword, 47
- broadcasting, ndarrays and, 94, 457, 460-465
- bucket analysis, 305
- build_design_matrices function, 389
- builtins module, 390
- bytes data type, 39, 43

C

- %C datetime format, 321
- C order (row major order), 454, 481
- c(ontinue) debugger command, 490
- calendar module, 318

- Cartesian product, 77, 230
- casefold method, 213
- cat method, 218
- categorical data
 - basic overview, 363-372
 - facet grids and, 283
 - Patsy library and, 390-393
- Categorical object, 203, 305, 363-372
- %cd magic function, 485, 487
- ceil function, 107
- center method, 219
- chaining methods, 378-380
- chisquare function, 119
- clear method, 66
- clipboard, executing code from, 26
- close method, 80, 83
- closed attribute, 83
- !cmd command, 485
- collections module, 64
- colon (:), 31
- color selection in matplotlib, 259
- column major order (Fortran order), 454, 481
- columns method, 315
- column_stack function, 456
- combinations function, 77
- combine_first method, 227, 242
- combining data (see merging data)
- command history
 - input and output variables, 484
 - reusing, 483
 - searching, 483
 - using in IPython, 483-485
- commands
 - debugger, 490
 - magic functions, 28-29
 - updating packages, 10
- comments in Python, 31
- compile method, 214
- complex128 data type, 91
- complex256 data type, 91
- complex64 data type, 91
- concat function, 227, 235, 237-241, 300
- concatenate function, 236, 454
- concatenating
 - along an axis, 227, 236-241
 - lists, 56
 - strings, 41
- conda update command, 10
- conditional logic as array operations, 109
- configuration for IPython, 501-502
- configuring matplotlib, 268
- contains method, 218
- contiguous memory, 480-482
- continue keyword, 47
- continuing education, 401
- control flow in Python, 46-50
- coordinated universal time (UTC), 335
- copy method, 95, 132
- copysign function, 107
- corr aggregation function, 359
- corr method, 161
- correlation, 160-162, 310
- corwith method, 162
- cos function, 107
- cosh function, 107
- count method, 40, 54, 160, 212-213, 218, 296
- cov method, 161
- covariance, 160-162
- %cpaste magic function, 26, 29
- cProfile module, 494-496
- cross-tabulation, 315
- crossstab function, 316
- cross_val_score function, 401
- CSV files, 168, 175-178
- csv module, 176
- Ctrl-A keyboard shortcut, 27
- Ctrl-B keyboard shortcut, 27
- Ctrl-C keyboard shortcut, 26, 27
- Ctrl-D keyboard shortcut, 16
- Ctrl-E keyboard shortcut, 27
- Ctrl-F keyboard shortcut, 27
- Ctrl-K keyboard shortcut, 27
- Ctrl-L keyboard shortcut, 27
- Ctrl-N keyboard shortcut, 27, 484
- Ctrl-P keyboard shortcut, 27, 483
- Ctrl-R keyboard shortcut, 27, 484
- Ctrl-Shift-V keyboard shortcut, 27
- Ctrl-U keyboard shortcut, 27
- cummax method, 160
- cummin method, 160
- cumprod method, 112, 160
- cumsum method, 112, 160, 466
- curly braces {}, 61, 65
- currying, 74
- cut function, 203, 305
- c_ object, 456

D

- %d datetime format, 46, 319
- %D datetime format, 46, 320
- d(own) debugger command, 490
- data aggregation
 - about, 296
 - column-wise, 298-301
 - multiple function application, 298-301
 - returning data without row indexes, 301
- data alignment, pandas library and, 146-151
- data analysis with Python
 - about, 2, 15-16
 - glue code, 2
 - MovieLens 1M dataset example, 413-419
 - restrictions to consider, 3
 - US baby names dataset example, 419-434
 - US Federal Election Commission database example, 440-448
 - USA.gov data from Bitly example, 403-413
 - USDA food database example, 434-439
 - “two-language” problem, 3
- data cleaning and preparation (see data wrangling)
- data loading (see reading data)
- data manipulation (see data wrangling)
- data munging (see data wrangling)
- data selection
 - for axis indexes with duplicate labels, 157
 - in pandas library, 140-145
 - time series data, 323
- data structures
 - about, 51
 - dict comprehensions, 67
 - dicts, 61-65
 - for pandas library, 124-136
 - list comprehensions, 67-69
 - lists, 54-59
 - set comprehensions, 68
 - sets, 65-67
 - tuples, 51-54
- data transformation (see transforming data)
- data types
 - attributes for, 469
 - defined, 90, 449
 - for date and time data, 318
 - for ndarrays, 90-93
 - in Python, 38-46
 - nested, 469
 - NumPy hierarchy, 450
 - parent classes of, 450
- data wrangling
 - combining and merging datasets, 227-242
 - defined, 14
 - handling missing data, 191-197
 - hierarchical indexing, 221-226, 243
 - pivoting data, 246-250
 - reshaping data, 243
 - string manipulation, 211-219
 - transforming data, 197-211
 - working with delimited formats, 176-178
- databases
 - DataFrame joins, 227-232
 - pandas interacting with, 188
 - storing data in, 247
- DataFrame data structure
 - about, 4, 128-134, 470
 - database-stye joins, 227-232
 - indexing with columns, 225
 - JSON data and, 180
 - operations between Series and, 149
 - optional function arguments, 168
 - plot method arguments, 271
 - possible data inputs to, 134
 - ranking data in, 155
 - sorting considerations, 153, 473
 - summary statistics methods for, 161
- DataOffset object, 338
- datasets
 - combining and merging, 227-242
 - MovieLens 1M example, 413-419
 - US baby names example, 419-434
 - US Federal Election Commission database example, 440-448
 - USA.gov data from Bitly example, 403-413
 - USDA food database example, 434-439
- date data type, 44, 319
- date offsets, 330, 333-334
- date ranges, generating, 328-330
- dates and times
 - about, 44
 - converting between strings and datetime, 319-321
 - data types and tools, 318
 - formatting specifications, 319, 321
 - generating date ranges, 328-330
 - period arithmetic and, 339-347
- datetime data type
 - about, 44, 318-319

- converting between strings and, 319-321
- format specification for, 319
- datetime module, 44, 318
- datetime64 data type, 322
- DatetimeIndex class, 322, 328, 337
- dateutil package, 320
- date_range function, 328-330
- daylight saving time (DST), 335
- debug function, 491
- %debug magic function, 80, 488
- debugger, IPython, 488-492
- decode method, 42
- def keyword, 69, 74
- default values for dicts, 63
- defaultdict class, 64
- del keyword, 62, 132
- del method, 132
- delete method, 136
- delimited formats, working with, 176-178
- dense method, 156
- density plots, 277-279
- deque (double-ended queue), 55
- describe method, 160, 297
- design matrix, 386
- det function, 117
- development tools for IPython (see software development tools for IPython)
- %dhist magic function, 486
- diag function, 117
- Dialect class, 177
- dict comprehensions, 67
- dict function, 63
- dictionary-encoded representation, 365
- dicts (data structures)
 - about, 61
 - creating from sequences, 63
 - DataFrame data structure as, 129
 - default values, 63
 - grouping with, 294
 - Series data structure as, 125
 - valid key types, 64
- diff method, 160
- difference method, 66, 136
- difference_update method, 66
- dimension tables, 364
- directories, bookmarking in IPython, 487
- %dirs magic function, 485
- discretization, 203
- distplot method, 279

- div method, 149
- divide function, 107
- divmod function, 106
- dmatrices function, 386
- dnorm function, 394
- dot function, 104, 116-117
- downsampling, 348, 349-351
- dreload function, 499
- drop method, 136, 138
- dropna method, 192-193, 306, 315
- drop_duplicates method, 197
- DST (daylight saving time), 335
- dstack function, 456
- dtype (see data types)
- dtype attribute, 88, 92
- duck typing, 35
- dummy variables, 208-211, 372, 386, 391
- dumps function, 179
- duplicate data
 - axis indexes with duplicate labels, 157
 - removing, 197
 - time series with duplicate indexes, 326
- uplicated method, 197
- dynamic references in Python, 33

E

- edit-compile-run workflow, 6
- education, continuing, 401
- eig function, 118
- elif statement, 46
- else statement, 46
- empty function, 89-90
- empty namespace, 25
- empty_like function, 90
- encode method, 42
- end-of-line (EOL) markers, 80
- endswith method, 213, 218
- enumerate function, 59
- %env magic function, 486
- EOL (end-of-line) markers, 80
- equal function, 108
- error handling in Python, 77-80
- escape characters, 41
- ewm function, 358
- Excel files (Microsoft), 186-187
- ExcelFile class, 186
- exception handling in Python, 77-80
- exclamation point (!), 486
- execute-explore workflow, 6

- exit command, 16
- exp function, 107
- expanding function, 356
- exponentially-weighted functions, 358
- extend method, 56
- extract method, 218
- eye function, 90

F

- %F datetime format, 46, 320
- fabs function, 107
- facet grids, 283
- FacetGrid class, 285
- factorplot built-in function, 283
- fancy indexing, 102, 459
- FDIC bank failures list, 180
- Feather binary file format, 168, 184
- feature engineering, 383
- Federal Election Commission database example, 440-448
- Figure object, 255
- file management
 - binary data formats, 183-187
 - commonly used file methods, 82
 - design tips, 500
 - file input and output with arrays, 115
 - JSON data, 178-180
 - memory-mapped files, 478
 - opening files, 80
 - Python file modes, 82
 - reading and writing data in text format, 167-176
 - saving plots to files, 267
 - Web scraping, 180-183
 - working with delimited formats, 176-178
- filling in data
 - arithmetic methods with fill values, 148
 - filling in missing data, 195-197, 200
 - with group-specific values, 306
- fillna method, 192, 195-197, 200, 306, 352
- fill_value method, 315
- filtering
 - in pandas library, 140-145
 - missing data, 193
 - outliers, 205
- find method, 212-213
- findall method, 214, 216, 218
- finditer method, 216
- first method, 156, 296

- fit method, 395, 400
- fixed frequency, 317
- flags attribute, 481
- flatten method, 453
- float data type, 39, 43
- float function, 43
- float128 data type, 91
- float16 data type, 91
- float32 data type, 91
- float64 data type, 91
- floor function, 107
- floordiv method, 149
- floor_divide function, 107
- flow control in Python, 46-50
- flush method, 83, 479
- fmax function, 107
- fmin function, 107
- for loops, 47, 68
- format method, 41
- formatting
 - dates and times, 319, 321
 - strings, 41
- Fortran order (column major order), 454, 481
- frequencies
 - base, 330
 - basic for time series, 329
 - converting between, 327, 348-354
 - date offsets and, 330
 - fixed, 317
 - period conversion, 340
 - quarterly period frequencies, 342
- fromfile function, 471
- frompyfunc function, 468
- from_codes method, 367
- full function, 90
- full_like function, 90
- functions, 69
 - (see also universal functions)
 - about, 69
 - accessing variables, 70
 - anonymous, 73
 - as objects, 72-73
 - currying, 74
 - errors and exception handling, 77
 - exponentially-weighted, 358
 - generators and, 75-80
 - grouping with, 295
 - in Python, 32
 - lambda, 73

- magic, 28-29
- namespaces and, 70
- object introspection, 23
- partial argument application, 74
- profiling line by line, 496-498
- returning multiple values, 71
- sequence, 59-61
- transforming data using, 198
- type inference in, 168
- writing fast NumPy functions with Numba, 476-478

functools module, 74

G

- gamma function, 119
- generators
 - about, 75
 - generator expressions for, 76
 - itertools module and, 76
- get method, 63, 218
- GET request (HTTP), 187
- getattr function, 35
- getroot method, 182
- get_chunk method, 175
- get_dummies function, 208, 372, 385
- get_indexer method, 164
- get_value method, 145
- GIL (global interpreter lock), 3
- global keyword, 71
- glue for code, Python as, 2
- greater function, 108
- greater_equal function, 108
- Greenwich Mean Time, 335
- group keys, suppressing, 304
- group operations
 - about, 287, 373
 - cross-tabulation, 315
 - data aggregation, 296-302
 - GroupBy mechanics, 288-296
 - pivot tables, 287, 313-316
 - split-apply-combine, 288, 302-312
 - unwrapped, 376
- group weighted average, 310
- groupby function, 77
- groupby method, 368, 476
- GroupBy object
 - about, 288-291
 - grouping by index level, 295
 - grouping with dicts, 294

- grouping with functions, 295
- grouping with Series, 294
- iterating over groups, 291
- optimized methods, 296
- selecting columns, 293
- selecting subset of columns, 293

groups method, 215

H

- %H datetime format, 46, 319
- h(elp) debugger command, 490
- hasattr function, 35
- hash function, 64
- hash maps (see dicts)
- hash mark (#), 31
- hashability, 64
- HDF5 (hierarchical data format 5), 184-186, 480
- HDFStore class, 184
- head method, 129
- heapsort method, 474
- hierarchical data format (HDF5), 480
- hierarchical indexing
 - about, 221-224
 - in pandas, 170
 - reordering and sorting levels, 224
 - reshaping data with, 243
 - summary statistics by level, 225
 - with DataFrame columns, 225
- %hist magic function, 29
- hist method, 277
- histograms, 277-279
- hsplit function, 456
- hstack function, 455
- HTML files, 180-183
- HTTP requests, 187
- Huginin, Jim, 86
- Hunter, John D., 5, 253

I

- %I datetime format, 46, 319
- identity function, 90
- IDEs (Integrated Development Environments), 11
- idxmax method, 160
- idxmin method, 160
- if statement, 46
- iloc operator, 143, 207
- immutable objects, 38, 367

- import conventions
 - for matplotlib, 253
 - for modules, 14, 36
 - for Python, 14, 36, 88
- importlib module, 499
- imshow function, 109
- in keyword, 56, 212
- in-place sorts, 57, 471
- in1d method, 114, 115
- indentation in Python, 30
- index method, 212-213, 315
- Index objects, 134-136
- indexes and indexing
 - axis indexes with duplicate labels, 157
 - boolean indexing, 99-102
 - fancy indexing, 102, 459
 - for ndarrays, 94-98
 - for pandas library, 140-145, 157
 - grouping by index level, 295
 - hierarchical indexing, 170, 221-226, 243
 - Index objects, 134-136
 - integer indexing, 145
 - merging on index, 232-235
 - renaming axis indexes, 201
 - time series data, 323
 - time series with duplicate indexes, 326
 - timedeltas and, 318
- indexing operator, 58
- indicator variables, 208-211
- indirect sorts, 472
- inner join type, 229
- input variables, 484
- insert method, 55, 136
- insort function, 57
- int data type, 39, 43
- int function, 43
- int16 data type, 91
- int32 data type, 91
- int64 data type, 91
- int8 data type, 91
- integer arrays, indexing, 102, 459
- integer indexing, 145
- Integrated Development Environments (IDEs), 11
- interactive debugger, 488-492
- interpreted languages, 2, 16
- interrupting running code, 26
- intersect1d method, 115
- intersection method, 65-66, 136
- intersection_update method, 66
- intervals of time, 317
- inv function, 118
- .ipynb file extension, 20
- IPython
 - %run command and, 17
 - %run command in, 25-26
 - about, 6
 - advanced features, 500-502
 - bookmarking directories, 487
 - code development tips, 498-500
 - command history in, 483-485
 - exception handling in, 79
 - executing code from clipboard, 26
 - figures and subplots, 255
 - interacting with operating system, 485-487
 - keyboard shortcuts for, 27
 - magic commands in, 28-29
 - matplotlib integration, 29
 - object introspection, 23-24
 - running Jupyter notebook, 18-20
 - running shell, 17-18
 - shell commands in, 486
 - software development tools, 487-498
 - tab completion in, 21-23
- ipython command, 17-18
- is keyword, 38
- is not keyword, 38
- isalnum method, 218
- isalpha method, 218
- isdecimal method, 218
- isdigit method, 218
- isdisjoint method, 66
- isfinite function, 107
- isin method, 136, 163
- isinf function, 107
- isinstance function, 34
- islower method, 218
- isnan function, 107
- isnull method, 126, 192
- isnumeric method, 218
- issubdtype function, 450
- issubset method, 66
- issuperset method, 66
- isupper method, 218
- is_monotonic property, 136
- is_unique property, 136, 157, 326
- iter function, 35
- __iter__ magic method, 35

iterator protocol, 35, 75-77
itertools module, 76

J

jit function, 477
join method, 212-213, 218, 235
join operations, 227-232
JSON (JavaScript Object Notation), 178-180, 403
json method, 187
Jupyter notebook
 %load magic function, 25
 about, 6
 plotting nuances, 256
 running, 18-20
jupyter notebook command, 19

K

KDE (kernel density estimate) plots, 278
kernels, defined, 6, 18
key-value pairs, 61
keyboard shortcuts for IPython, 27
KeyboardInterrupt exception, 26
KeyError exception, 66
keys method, 62
keyword arguments, 32, 70
kurt method, 160

L

l(ist) debugger command, 490
labels
 axis indexes with duplicate labels, 157
 selecting in matplotlib, 261-263
lagging data, 332
lambda (anonymous) functions, 73
language semantics for Python
 about, 30
 attributes, 35
 binary operators and comparisons, 36, 65
 comments, 31
 duck typing, 35
 function and object method calls, 32
 import conventions, 36
 indentation not braces, 30
 methods, 35
 mutable and immutable objects, 38
 object model, 31
 references, 32-34

strongly typed language, 33
variables and argument passing, 32

last method, 296
leading data, 332
left join type, 229
legend method, 264
legend selection in matplotlib, 261-265
len function, 295
len method, 218
less function, 108
less_equal function, 108
level keyword, 296
level method, 159
levels
 grouping by index levels, 295
 sorting, 224
 summary statistics by, 225
lexsort method, 473
libraries (see specific libraries)
line plots, 269-271
line style selection in matplotlib, 260
linear algebra, 116-118
linear regression, 312, 393-396
Linux, setting up Python on, 9
list comprehensions, 67-69
list function, 37, 54
lists (data structures)
 about, 54
 adding and removing elements, 55
 combining, 56
 concatenating, 56
 maintaining sorted lists, 57
 slicing, 58
 sorting, 57
lists (data structures) binary searches, 57
ljust method, 213
load function, 115, 478
%load magic function, 25
loads function, 179
loc operator, 130, 143, 265, 385
local namespace, 70, 123
localizing data to time zones, 335
log function, 107
log10 function, 107
log1p function, 107
log2 function, 107
logical_and function, 108, 466
logical_not function, 107
logical_or function, 108

logical_xor function, 108
LogisticRegression class, 399
LogisticRegressionCV class, 400
long format, 246
lower method, 199, 213, 218
%lprun magic function, 496
lstrip method, 213, 219
lstsq function, 118
lxml library, 180-183

M

%m datetime format, 46, 319
%M datetime format, 46, 319
mad method, 160
magic functions, 28-29
(see also specific magic functions)
%debug magic function, 29
%magic magic function, 29
many-to-many merge, 229
many-to-one join, 228
map built-in function, 68, 73
map method, 153, 199, 202
mapping
 transforming data using, 198
 universal functions, 151-156
margins method, 315
margins, defined, 313
marker selection in matplotlib, 260
match method, 164, 214, 216, 219
Math Kernel Library (MKL), 117
matplotlib library
 about, 5, 253
 annotations in, 265-267
 color selection in, 259
 configuring, 268
 creating image plots, 109
 figures in, 255-259
 import convention, 253
 integration with IPython, 29
 label selection in, 261-263
 legend selection in, 261-265
 line style selection in, 260
 marker selection in, 260
 saving plots to files, 267
 subplots in, 255-259, 265-267
 tick mark selection in, 261-263
%matplotlib magic function, 30, 486
matrix operations in NumPy, 104, 116
max method, 112, 156, 160, 296

maximum function, 107
mean method, 112, 160, 289, 296
median method, 160, 296
melt method, 249
mmap object, 478
memory management
 C versus Fortran order, 454
 contiguous memory, 480-482
 NumPy-based algorithms and, 87
memory-mapped files, 478
merge function, 227-232
mergesort method, 474
merging data
 combining data with overlap, 241
 concatenating along an axis, 236-241
 database-stye DataFrame joins, 227-232
 merging on index, 232-235
meshgrid function, 108
methods
 categorical, 370-372
 chaining, 378-380
 defined, 32
 for boolean arrays, 113
 for strings, 211-213
 for summary statistics, 162-165
 for tuples, 54
 hidden, 22
 in Python, 32, 35
 object introspection, 23
 optimized for GroupBy, 296
 statistical, 111-112
 ufunc instance methods, 466-468
 vectorized string methods in pandas,
 216-219
Microsoft Excel files, 186-187
min method, 112, 156, 160, 296
minimum function, 107
missing data
 about, 191
 filling in, 195-197, 200
 filling with group-specific values, 306
 filtering out, 193
 marked by sentinel values, 171, 191
 sorting considerations, 154
mixture-of-normals estimate, 278
MKL (Math Kernel Library), 117
mod function, 107
modf function, 106-107
modules

- import conventions for, 14, 36
- reloading dependencies, 498
- MovieLens 1M dataset example, 413-419
- moving window functions
 - about, 354-357
 - binary, 359
 - exponentially-weighted functions, 358
 - user-defined, 361
- mro method, 450
- MSFT attribute, 161
- mul method, 149
- multiply function, 107
- munging (see data wrangling)
- mutable objects, 38

N

- n(ext) debugger command, 490
- NA data type, 192
- name attribute, 127, 130
- names attribute, 100, 469
- namespaces
 - empty, 25
 - functions and, 70
 - in Python, 34
 - NumPy, 88
- NaN (Not a Number), 107, 126, 191
- NaT (Not a Time), 321
- ndarray object
 - about, 85, 87-88
 - advanced input and output, 478-480
 - arithmetic with, 93
 - array-oriented programming, 108-115
 - as structured arrays, 469-471
 - attributes for, 89, 453, 463, 481
 - boolean indexing, 99-102
 - broadcasting and, 94, 457, 460-465
 - C versus Fortran order, 454
 - C versus Fortran order, 481
 - concatenating arrays, 454
 - creating, 88-90
 - creating PeriodIndex from arrays, 345
 - data types for, 90-93
 - fancy indexing, 102, 459
 - file input and output, 115
 - finding elements in sorted arrays, 475
 - indexes for, 94-98
 - internals overview, 449-451
 - linear algebra and, 116-118
 - partially sorting arrays, 474

- pseudorandom number generation, 118-119
- random walks example, 119-122
- repeating elements in, 457
- reshaping arrays, 103, 452
- slicing arrays, 94-98
- sorting considerations, 113, 471
- splitting arrays, 455
- storage options, 480
- swapping axes in, 103
- transposing arrays, 103
- ndim attribute, 89
- nested code, 500
- nested data types, 469
- nested list comprehensions, 68-69
- nested tuples, 53
- New York MTA (Metropolitan Transportation Authority), 181
- newaxis attribute, 463
- “no-op” statement, 48
- None data type, 39, 44, 192
- normal function, 119
- not keyword, 56
- notfull method, 192
- notnull method, 126
- not_equal function, 108
- .npy file extension, 115
- .npz file extension, 115
- null value, 39, 44, 178
- Numba
 - creating custom ufunc objects with, 478
 - writing fast NumPy functions with, 476-478
- numeric data types, 39
- NumPy library
 - about, 4, 85-87
 - advanced array input and output, 478-480
 - advanced array manipulation, 451-459
 - advanced ufunc usage, 466-469
 - array-oriented programming, 108-115
 - arrays and broadcasting, 460-465
 - file input and output with arrays, 115
 - linear algebra and, 116-118
 - ndarray object internals, 449-451
 - ndarray object overview, 87-105
 - performance tips, 480-482
 - pseudorandom number generation, 118-119
 - random walks example, 119-122
 - sorting considerations, 113, 471-476
 - structured and record arrays, 469-471
 - ufunc overview, 105-108

O

- object data type, 91
- object introspection, 23-24
- object model, 31
- objectify function, 181-183
- objects (see Python objects)
- OHLC (Open-High-Low-Close) resampling, 351
- ohlcv aggregate function, 351
- Oliphant, Travis, 86
- OLS (ordinary least squares) regression, 312, 388
- OLS class, 395
- Olson database, 335
- ones function, 89-90
- ones_like function, 90
- open built-in function, 80, 83
- openpyxl package, 186
- operating system, IPython interacting with, 485-487
- or keyword, 43, 101
- OS X, setting up Python on, 9
- outer method, 467
- outliers, detecting and filtering, 205
- output join type, 229
- output variables, 484

P

- %p datetime format, 321
- packages, installing or updating, 10
- pad method, 219
- %page magic function, 29
- pairplot function, 281
- pairs plot, 281
- pandas library, 4
 - (see also data wrangling)
 - about, 4, 123
 - arithmetic and data alignment, 146-151
 - as time zone naive, 335
 - binary data formats, 183-187
 - categorical data and, 363-372
 - data structures for, 124-136
 - drop method, 138
 - filtering in, 140-145
 - function application and mapping, 151
 - group operations and, 373-378
 - indexes in, 140-145, 157
 - integer indexing, 145
 - interacting with databases, 188
 - interacting with Web APIs, 187
 - interfacing with model code, 383
 - JSON data, 178-180
 - method chaining, 378-380
 - nested data types and, 470
 - plotting with, 268-285
 - ranking data in, 153-156
 - reading and writing data in text format, 167-176
 - reductions in, 158-165
 - reindex method, 136-138
 - selecting data in, 140-145
 - sorting considerations, 153-156, 473, 476
 - summary statistics in, 158-165
 - vectorized string methods in, 216-219
 - Web scraping, 180-183
 - working with delimited formats, 176-178
- pandas-datareader package, 160
- parentheses (), 32, 51
- parse method, 186, 320
- partial argument application, 74
- partial function, 74
- partition method, 474
- pass statement, 48
- %paste magic function, 26, 29
- patches, defined, 266
- Patsy library
 - about, 386
 - categorical data and, 390-393
 - creating model descriptions with, 386-388
 - data transformations in Patsy formulas, 389
- pct_change method, 160, 311
- %pdb magic function, 29, 80, 489
- percent sign (%), 28, 495
- percentileofscore function, 361
- Pérez, Fernando, 6
- period (.), 21
- Period class, 339
- PeriodIndex class, 340, 345
- periods of dates and times
 - about, 339
 - converting frequencies, 340
 - converting timestamps to/from, 344
 - creating PeriodIndex from arrays, 345
 - fixed periods, 317
 - quarterly period frequencies, 342
 - resampling with, 353

- period_range function, 340, 343
- PerktoId, Josef, 8
- permutation function, 119, 206
- permutations function, 77
- pickle module, 183
- pinv function, 118
- pip tool, 10, 180
- pipe method, 380
- pivot method, 247
- pivot tables, 287, 313-316
- pivoting data, 246-250
- pivot_table method, 313
- plot function, 259
- plot method, 269-271
- Plotly tool, 285
- plotting
 - with matplotlib, 253-268
 - with pandas and seaborn, 268-285
- point plots, 280
- pop method, 55, 62-63, 66
- %popd magic function, 485
- positional arguments, 32, 70
- pound sign (#), 31
- pow method, 149
- power function, 107
- pprint module, 500
- predict method, 400
- preparation, data (see data wrangling)
- private attributes, 22
- private methods, 22
- prod method, 160, 296
- product function, 77
- profiles for IPython, 501-502
- profiling code in IPython, 494-496
- profiling functions line by line, 496-498
- %prun magic function, 29, 495-496
- pseudocode, 14, 30
- pseudorandom number generation, 118-119
- %pushd magic function, 485
- put method, 459
- %pwd magic function, 485
- .py file extension, 16, 36
- pyplot module, 261
- Python
 - community and conferences, 12
 - control flow, 46-50
 - data analysis with, 2-3, 15-16
 - essential libraries, 4-8
 - historical background, 11

- import conventions, 14, 36, 88
- installation and setup, 8-12
- interpreter for, 16
- language semantics, 30-38
- scalar types, 38-46
- python command, 16
- Python objects
 - attributes and methods, 35
 - converting to strings, 40
 - defined, 31
 - formatting, 18
 - functions as, 72-73
 - key-value pairs, 61
- pytz library, 335

Q

- q(uit) debugger command, 490
- qcut function, 204, 305, 368
- qr function, 118
- quantile analysis, 305
- quantile method, 160, 296
- quarterly period frequencies, 342
- question mark (?), 23-24
- %quickref magic function, 29
- quicksort method, 474
- quotation marks in strings, 39

R

- r character prefacing quotes, 41
- R language, 5, 8, 192
- radd method, 149
- rand function, 119
- randint function, 119
- randn function, 99, 119
- random module, 118-122
- random number generation, 118-119
- random sampling and permutation, 308
- random walks example, 119-122
- RandomState class, 119
- range function, 48, 90
- rank method, 155
- ranking data in pandas library, 153-156
- ravel method, 453
- rc method, 268
- rdiv method, 149
- re module, 72, 213
- read method, 81-82
- read-and-write mode for files, 82
- read-only mode for files, 82

- reading data
 - in Microsoft Excel files, 186-187
 - in text format, 167-175
- readline functionality, 484
- readlines method, 82
- read_clipboard function, 167
- read_csv function, 80, 167, 172, 274, 298
- read_excel function, 167, 186
- read_feather function, 168
- read_fwf function, 167
- read_hdf function, 167, 185
- read_html function, 167, 180-183
- read_json function, 167, 179
- read_msgpack function, 167
- read_pickle function, 167, 183
- read_sas function, 168
- read_sql function, 168, 190
- read_stata function, 168
- read_table function, 167, 172, 176
- reduce method, 466
- reducat method, 467
- reductions (aggregations), 111
- references in Python, 32-34
- regplot method, 281
- regress function, 312
- regular expressions
 - passes as delimiters, 171
 - string manipulation and, 213-216
- reindex method, 136-138, 145, 157, 352
- reload function, 499
- remove method, 56, 66
- remove_categories method, 372
- remove_unused_categories method, 372
- rename method, 202
- rename_categories method, 372
- reorder_categories method, 372
- repeat function, 457
- repeat method, 219
- replace method, 200, 212-213, 219
- requests package, 187
- resample method, 327, 348-351, 377
- resampling
 - defined, 348
 - downsampling and, 348-351
 - OHLC, 351
 - upsampling and, 348, 352
 - with periods, 353
- %reset magic function, 29, 485
- reset_index method, 250, 302

- reshape method, 103, 452
- *rest syntax, 54
- return statement, 69
- reusing command history, 483
- reversed function, 61
- rfind method, 213
- rfloordiv method, 149
- right join type, 229
- rint function, 107
- rjust method, 213
- rmul method, 149
- rollback method, 333
- rollforward method, 333
- rolling function, 355, 357
- rolling_corr function, 360
- row major order (C order), 454, 481
- row_stack function, 456
- rpow method, 149
- rstrip method, 213, 219
- rsub method, 149
- %run magic function
 - about, 29
 - exceptions and, 79
 - interactive debugger and, 489, 492
 - IPython and, 17, 25-26
 - reusing command history with, 483
- r_ object, 456

S

- %S datetime format, 46, 319
- s(tep) debugger command, 490
- sample method, 207, 308
- save function, 115, 478
- savefig method, 267
- savez function, 115
- savez_compressed function, 116
- scalar types in Python, 38-46, 93
- scatter plot matrix, 281
- scatter plots, 280
- scikit-learn library, 7, 397-401
- SciPy library, 6
- scope of functions, 70
- scripting languages, 2
- Seabold, Skipper, 8
- seaborn library, 269
- search method, 214, 216
- searching
 - binary searches of lists, 57
 - command history, 483

- searchsorted method, 475
- seed function, 119
- seek method, 81, 83-84
- semantics, language (see language semantics for Python)
- semicolon (;), 31
- sentinel value, 171, 191
- sequence functions, 59-61
- serialization (see storing data)
- Series data structure
 - about, 4, 124-128
 - duplicate indexes example, 157
 - grouping with, 294
 - JSON data and, 180
 - operations between DataFrame and, 149
 - plot method arguments, 271
 - ranking data in, 155
 - sorting considerations, 154, 473
 - summary statistics methods for, 161
- set comprehensions, 68
- set function, 65, 277
- set literals, 65
- set operations, 65-67, 114
- setattr function, 35
- setdefault method, 64
- setdiff1d method, 115
- sets (data structures), 65-67
- setxor1d method, 115
- set_categories method, 372
- set_index method, 248
- set_title method, 263, 266
- set_trace function, 491
- set_value method, 145
- set_xlabel method, 263
- set_xlim method, 266
- set_xticklabels method, 262
- set_xticks method, 262
- set_ylim method, 266
- shape attribute, 88-89, 453
- shell commands in IPython, 486
- shift method, 332, 351
- shifting time series data, 332-334
- shuffle function, 119
- side effects, 38
- sign function, 107, 206
- sin function, 107
- sinh function, 107
- size method, 291
- skew method, 160
- skipna method, 159
- slice method, 219
- slice notation, 58
- slicing
 - lists, 58
 - ndarrays, 94-98
 - strings, 41
- Smith, Nathaniel, 8
- Social Security Administration (SSA), 419
- software development tools for IPython
 - about, 487
 - basic profiling, 494-496
 - interactive debugger, 488-492
 - profiling functions line by line, 496-498
 - timing code, 492-493
- solve function, 118
- sort method, 57, 60, 74, 113
- sorted function, 57, 60
- sorting considerations
 - finding elements in sorted arrays, 475
 - hierarchical indexing, 224
 - in-place sorts, 57, 71
 - indirect sorts, 472
 - missing data, 154
 - NumPy library, 113, 471-476
 - pandas library, 153-156, 473, 476
 - partially sorting arrays, 474
 - stable sorting, 474
- sort_index method, 153
- sort_values method, 154, 473
- spaces, structuring code with, 30
- split concatenation function, 456
- split function, 455
- split method, 178, 211, 213-214, 216, 219
- split-apply-combine
 - about, 288
 - applying, 302-312
 - filling missing values with group-specific values, 306
 - group weighted average and correlation, 310
 - group-wise linear regression, 312
 - quantile and bucket analysis, 305
 - random sampling and permutation, 308
 - suppressing group keys, 304
- SQL (structured query language), 287
- SQLAlchemy project, 190
- sqlite3 module, 188
- sqrt function, 107
- square brackets [], 52, 54

- square function, 107
- SSA (Social Security Administration), 419
- stable sorting, 474
- stack method, 243
- stacked format, 246
- stacking operation, 227, 236
- start index, 58
- startswith method, 213, 218
- Stata file format, 168
- statistical methods, 111-112
- statsmodels library
 - about, 8, 393
 - estimating linear models, 393-396
 - estimating time series processes, 396
 - OLS regression and, 312
- std method, 112, 160, 296
- step index, 59
- stop index, 58
- storing data
 - in binary format, 183-187
 - in databases, 247
 - ndarray object, 480
- str data type, 39, 43
- str function, 40, 43, 319
- strftime method, 45, 319
- strides/strided view, 449
- strings
 - concatenating, 41
 - converting between datetime and, 319-321
 - converting Python objects to, 40
 - data types for, 39-42
 - formatting, 41
 - manipulating, 211-219
 - methods for, 211-213
 - regular expressions and, 213-216
 - slicing, 41
 - vectorized methods in pandas, 216-219
- string_data type, 91
- strip method, 211, 213, 219
- strongly typed language, 33
- strptime function, 45, 320
- structured arrays, 469-471
- structured data, 1
- sub method, 149, 215, 216
- subn method, 216
- subplots
 - about, 255-259
 - drawing on, 265-267
 - subplots method, 257

- subplots_adjust method, 258
- subsetting time series data, 323
- subtract function, 107
- sum method, 112, 158, 160, 296, 466
- summary method, 395
- summary statistics
 - about, 158-160
 - by level, 225
 - correlation and covariance, 160-162
 - methods for, 162-165
- svd function, 118
- swapaxes method, 105
- swapping axes in arrays, 103
- symmetric_difference method, 66
- symmetric_difference_update method, 66
- syntactic sugar, 14
- sys module, 81, 175

T

- T attribute, 103
- tab completion in IPython, 21-23
- tabs, structuring code with, 30
- take method, 207, 364, 459
- tan function, 107
- tanh function, 107
- Taylor, Jonathan, 8
- tell method, 81, 83
- ternary expressions, 49
- text editors, 11
- text files
 - reading, 167-175
 - text mode for files, 82-83
 - writing to, 167-176
- text function, 265
- TextParser class, 174
- tick mark selection in matplotlib, 261-263
- tile function, 457
- time data type, 44, 319
- %time magic function, 29, 492
- time module, 318
- time series data
 - about, 317
 - basics overview, 322-323
 - date offsets and, 330, 333-334
 - estimating time series processes, 396
 - frequencies and, 329
 - frequencies and, 330, 348-354
 - indexing and, 323
 - moving window functions, 354-362

- periods in, 339-347
- resampling, 348-354
- selecting, 323
- shifting, 332-334
- subsetting, 323
- time zone handling, 335-339
- with duplicate indexes, 326

time zones

- about, 335
- converting data to, 336
- localizing data to, 335
- operations between different, 339
- operations with timestamp objects, 338
- USA.gov dataset example, 404-413

time, programmer versus CPU, 3

timedelta data type, 318-319

TimeGrouper object, 378

%timeit magic function, 29, 481, 492

Timestamp object, 322, 333, 338

timestamps

- converting periods to/from, 344
- defined, 317
- operations with time-zone-aware objects, 338

timezone method, 335

timing code, 492-493

top function, 303

to_csv method, 175

to_datetime method, 320

to_excel method, 187

to_json method, 180

to_period method, 344

to_pickle method, 183

to_timestamp method, 345

trace function, 117

transform method, 373-376

transforming data

- about, 197
- computing indicator/dummy variables, 208-211
- detecting and filtering outliers, 205
- discretization and binning, 203
- in Patsy formulas, 389
- permutation and random sampling, 206
- removing duplicates, 197
- renaming axis indexes, 201
- replacing values, 200
- using functions or mapping, 198

transpose method, 103

- transposing arrays, 103
- truncate method, 325
- try/except blocks, 77-79
- tuples (data structures)
 - about, 51
 - methods for, 54
 - nested, 53
 - unpacking, 53
- “two-language” problem, 3
- type casting, 43
- type inference in functions, 168
- TypeError exception, 78
- tzinfo data type, 319
- tz_convert method, 336

U

- %U datetime format, 46, 320
- u(p) debugger command, 490
- ufuncs (see universal functions)
- uint16 data type, 91
- uint32 data type, 91
- uint64 data type, 91
- uint8 data type, 91
- unary universal functions, 106, 107
- underscore (`_`), 22, 54, 451
- underscore (`_`), 484
- Unicode standard, 40, 42, 83
- unicode_ data type, 91
- uniform function, 119
- union method, 65-66, 136
- unionId method, 115
- unique method, 114-115, 136, 162, 164, 363
- universal functions
 - applying and mapping, 151
 - comprehensive overview, 105-108
 - creating custom objects with Numba, 478
 - instance methods, 466-468
 - writing in Python, 468
- unpacking tuples, 53
- unstack method, 243
- unwrapped group operation, 376
- update method, 63, 66
- updating packages, 10
- upper method, 213, 218
- upsampling, 348, 352
- US baby names dataset example, 419-434
- US Federal Election Commission database example, 440-448
- USA.gov dataset example, 403-413

USDA food database example, [434-439](#)
UTC (coordinated universal time), [335](#)
UTF-8 encoding, [83](#)

V

ValueError exception, [77, 92](#)
values attribute, [133](#)
values method, [62, 315](#)
values property, [384](#)
value_count method, [203](#)
value_counts method, [162, 274, 363](#)
var method, [112, 160, 296](#)
variables
 dummy, [208-211, 372, 386, 391](#)
 function scope and, [70](#)
 in Python, [32-34](#)
 indicator, [208-211](#)
 input, [484](#)
 output, [484](#)
 shell commands and, [486](#)
vectorization, [93](#)
vectorize function, [468, 478](#)
vectorized string methods in pandas, [216-219](#)
visualization tools, [285](#)
vsplit function, [456](#)
vstack function, [455](#)

W

%w datetime format, [46, 319](#)
%W datetime format, [46, 320](#)
w(here) debugger command, [490](#)
Waskom, Michael, [269](#)
Wattenberg, Laura, [430](#)
Web APIs, pandas interacting with, [187](#)
Web scraping, [180-183](#)
where function, [109, 241](#)
while loops, [48](#)
whitespace
 regular expression describing, [214](#)

 structuring code with, [30](#)
 trimming around figures, [267](#)
%who magic function, [29](#)
%whos magic function, [29](#)
%who_ls magic function, [29](#)
Wickham, Hadley, [184, 288, 419](#)
wildcard expressions, [24](#)
Williams, Ashley, [434](#)
Windows, setting up Python on, [9](#)
 with statement, [81](#)
wrangling (see data wrangling)
write method, [82](#)
write-only mode for files, [82](#)
writelines method, [82-83](#)
writing data in text format, [167-176](#)

X

%x datetime format, [321](#)
%X datetime format, [321](#)
%xdel magic function, [29, 485](#)
xlim method, [262](#)
xlrd package, [186](#)
XLS files, [186](#)
XLSX files, [186](#)
XML files, [180-183](#)
%xmode magic function, [79](#)

Y

%Y datetime format, [45, 319](#)
%y datetime format, [45, 319](#)
yield keyword, [75](#)

Z

%z datetime format, [46, 320](#)
"zero-copy" array views, [450](#)
zeros function, [89-90](#)
zeros_like function, [90](#)
zip function, [60](#)