



Python Data Science Handbook

ESSENTIAL TOOLS FOR WORKING WITH DATA

powered by



Jake VanderPlas

Python Data Science Handbook

For many researchers, Python is a first-class tool mainly because of its libraries for storing, manipulating, and gaining insight from data. Several resources exist for individual pieces of this data science stack, but only with the *Python Data Science Handbook* do you get them all—IPython, NumPy, Pandas, Matplotlib, Scikit-Learn, and other related tools.

Working scientists and data crunchers familiar with reading and writing Python code will find this comprehensive desk reference ideal for tackling day-to-day issues: manipulating, transforming, and cleaning data; visualizing different types of data; and using data to build statistical or machine learning models. Quite simply, this is the must-have reference for scientific computing in Python.

With this handbook, you'll learn how to use:

- **IPython and Jupyter:** provide computational environments for data scientists using Python
- **NumPy:** includes the *ndarray* for efficient storage and manipulation of dense data arrays in Python
- **Pandas:** features the *DataFrame* for efficient storage and manipulation of labeled/columnar data in Python
- **Matplotlib:** includes capabilities for a flexible range of data visualizations in Python
- **Scikit-Learn:** for efficient and clean Python implementations of the most important and established machine learning algorithms

“If you want to learn data science with Python, this book is a fantastic starting point. I've used it with great success to teach computer science and statistics majors. Jake goes far beyond the basics of open source tools; he also explains the underlying concepts, patterns, and abstractions of data science using clear language and approachable explanations.”

—**Brian Granger**

Associate Professor of Physics,
Cal Poly; cofounder of Project Jupyter

Jake VanderPlas, a long-time user and developer of the Python scientific stack, currently works as an interdisciplinary research director at the University of Washington. He conducts his own astronomy research, and spends time advising and consulting with local scientists from a wide range of fields.

PYTHON / DATA

US \$59.99

CAN \$68.99

ISBN: 978-1-491-91205-8



9 781491 912058

5 5 9 9 9



Twitter: @oreillymedia
facebook.com/oreilly

Python Data Science Handbook

Essential Tools for Working with Data

Jake VanderPlas

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Python Data Science Handbook

by Jake VanderPlas

Copyright © 2017 Jake VanderPlas. All rights reserved.

Printed in the United States of America.

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

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

Editor: Dawn Schanafelt

Indexer: WordCo Indexing Services, Inc.

Production Editor: Kristen Brown

Interior Designer: David Futato

Copyeditor: Jasmine Kwityn

Cover Designer: Karen Montgomery

Proofreader: Rachel Monaghan

Illustrator: Rebecca Demarest

December 2016: First Edition

Revision History for the First Edition

2016-11-17: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491912058> for release details.

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

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

978-1-491-91205-8

[LSI]

Table of Contents

Preface.....	xı
1. IPython: Beyond Normal Python.....	1
Shell or Notebook?	2
Launching the IPython Shell	2
Launching the Jupyter Notebook	2
Help and Documentation in IPython	3
Accessing Documentation with ?	3
Accessing Source Code with ??	5
Exploring Modules with Tab Completion	6
Keyboard Shortcuts in the IPython Shell	8
Navigation Shortcuts	8
Text Entry Shortcuts	9
Command History Shortcuts	9
Miscellaneous Shortcuts	10
IPython Magic Commands	10
Pasting Code Blocks: %paste and %cpaste	11
Running External Code: %run	12
Timing Code Execution: %timeit	12
Help on Magic Functions: ?, %magic, and %lsmagic	13
Input and Output History	13
IPython's In and Out Objects	13
Underscore Shortcuts and Previous Outputs	15
Suppressing Output	15
Related Magic Commands	16
IPython and Shell Commands	16
Quick Introduction to the Shell	16
Shell Commands in IPython	18

Passing Values to and from the Shell	18
Shell-Related Magic Commands	19
Errors and Debugging	20
Controlling Exceptions: %xmode	20
Debugging: When Reading Tracebacks Is Not Enough	22
Profiling and Timing Code	25
Timing Code Snippets: %timeit and %time	25
Profiling Full Scripts: %prun	27
Line-by-Line Profiling with %lprun	28
Profiling Memory Use: %memit and %mprun	29
More IPython Resources	30
Web Resources	30
Books	31
2. Introduction to NumPy.....	33
Understanding Data Types in Python	34
A Python Integer Is More Than Just an Integer	35
A Python List Is More Than Just a List	37
Fixed-Type Arrays in Python	38
Creating Arrays from Python Lists	39
Creating Arrays from Scratch	39
NumPy Standard Data Types	41
The Basics of NumPy Arrays	42
NumPy Array Attributes	42
Array Indexing: Accessing Single Elements	43
Array Slicing: Accessing Subarrays	44
Reshaping of Arrays	47
Array Concatenation and Splitting	48
Computation on NumPy Arrays: Universal Functions	50
The Slowness of Loops	50
Introducing UFuncs	51
Exploring NumPy's UFuncs	52
Advanced Ufunc Features	56
Ufuncs: Learning More	58
Aggregations: Min, Max, and Everything in Between	58
Summing the Values in an Array	59
Minimum and Maximum	59
Example: What Is the Average Height of US Presidents?	61
Computation on Arrays: Broadcasting	63
Introducing Broadcasting	63
Rules of Broadcasting	65
Broadcasting in Practice	68

Comparisons, Masks, and Boolean Logic	70
Example: Counting Rainy Days	70
Comparison Operators as ufuncs	71
Working with Boolean Arrays	73
Boolean Arrays as Masks	75
Fancy Indexing	78
Exploring Fancy Indexing	79
Combined Indexing	80
Example: Selecting Random Points	81
Modifying Values with Fancy Indexing	82
Example: Binning Data	83
Sorting Arrays	85
Fast Sorting in NumPy: np.sort and np.argsort	86
Partial Sorts: Partitioning	88
Example: k-Nearest Neighbors	88
Structured Data: NumPy's Structured Arrays	92
Creating Structured Arrays	94
More Advanced Compound Types	95
RecordArrays: Structured Arrays with a Twist	96
On to Pandas	96
3. Data Manipulation with Pandas.....	97
Installing and Using Pandas	97
Introducing Pandas Objects	98
The Pandas Series Object	99
The Pandas DataFrame Object	102
The Pandas Index Object	105
Data Indexing and Selection	107
Data Selection in Series	107
Data Selection in DataFrame	110
Operating on Data in Pandas	115
Ufuncs: Index Preservation	115
UFuncs: Index Alignment	116
Ufuncs: Operations Between DataFrame and Series	118
Handling Missing Data	119
Trade-Offs in Missing Data Conventions	120
Missing Data in Pandas	120
Operating on Null Values	124
Hierarchical Indexing	128
A Multiply Indexed Series	128
Methods of MultiIndex Creation	131
Indexing and Slicing a MultiIndex	134

Rearranging Multi-Indices	137
Data Aggregations on Multi-Indices	140
Combining Datasets: Concat and Append	141
Recall: Concatenation of NumPy Arrays	142
Simple Concatenation with pd.concat	142
Combining Datasets: Merge and Join	146
Relational Algebra	146
Categories of Joins	147
Specification of the Merge Key	149
Specifying Set Arithmetic for Joins	152
Overlapping Column Names: The suffixes Keyword	153
Example: US States Data	154
Aggregation and Grouping	158
Planets Data	159
Simple Aggregation in Pandas	159
GroupBy: Split, Apply, Combine	161
Pivot Tables	170
Motivating Pivot Tables	170
Pivot Tables by Hand	171
Pivot Table Syntax	171
Example: Birthrate Data	174
Vectorized String Operations	178
Introducing Pandas String Operations	178
Tables of Pandas String Methods	180
Example: Recipe Database	184
Working with Time Series	188
Dates and Times in Python	188
Pandas Time Series: Indexing by Time	192
Pandas Time Series Data Structures	192
Frequencies and Offsets	195
Resampling, Shifting, and Windowing	196
Where to Learn More	202
Example: Visualizing Seattle Bicycle Counts	202
High-Performance Pandas: eval() and query()	208
Motivating query() and eval(): Compound Expressions	209
pandas.eval() for Efficient Operations	210
DataFrame.eval() for Column-Wise Operations	211
DataFrame.query() Method	213
Performance: When to Use These Functions	214
Further Resources	215

4. Visualization with Matplotlib.....	217
General Matplotlib Tips	218
Importing matplotlib	218
Setting Styles	218
show() or No show()? How to Display Your Plots	218
Saving Figures to File	221
Two Interfaces for the Price of One	222
Simple Line Plots	224
Adjusting the Plot: Line Colors and Styles	226
Adjusting the Plot: Axes Limits	228
Labeling Plots	230
Simple Scatter Plots	233
Scatter Plots with plt.plot	233
Scatter Plots with plt.scatter	235
plot Versus scatter: A Note on Efficiency	237
Visualizing Errors	237
Basic Errorbars	238
Continuous Errors	239
Density and Contour Plots	241
Visualizing a Three-Dimensional Function	241
Histograms, Binnings, and Density	245
Two-Dimensional Histograms and Binnings	247
Customizing Plot Legends	249
Choosing Elements for the Legend	251
Legend for Size of Points	252
Multiple Legends	254
Customizing Colorbars	255
Customizing Colorbars	256
Example: Handwritten Digits	261
Multiple Subplots	262
plt.axes: Subplots by Hand	263
plt.subplot: Simple Grids of Subplots	264
plt.subplots: The Whole Grid in One Go	265
plt.GridSpec: More Complicated Arrangements	266
Text and Annotation	268
Example: Effect of Holidays on US Births	269
Transforms and Text Position	270
Arrows and Annotation	272
Customizing Ticks	275
Major and Minor Ticks	276
Hiding Ticks or Labels	277
Reducing or Increasing the Number of Ticks	278

Fancy Tick Formats	279
Summary of Formatters and Locators	281
Customizing Matplotlib: Configurations and Stylesheets	282
Plot Customization by Hand	282
Changing the Defaults: rcParams	284
Stylesheets	285
Three-Dimensional Plotting in Matplotlib	290
Three-Dimensional Points and Lines	291
Three-Dimensional Contour Plots	292
Wireframes and Surface Plots	293
Surface Triangulations	295
Geographic Data with Basemap	298
Map Projections	300
Drawing a Map Background	304
Plotting Data on Maps	307
Example: California Cities	308
Example: Surface Temperature Data	309
Visualization with Seaborn	311
Seaborn Versus Matplotlib	312
Exploring Seaborn Plots	313
Example: Exploring Marathon Finishing Times	322
Further Resources	329
Matplotlib Resources	329
Other Python Graphics Libraries	330
5. Machine Learning	331
What Is Machine Learning?	332
Categories of Machine Learning	332
Qualitative Examples of Machine Learning Applications	333
Summary	342
Introducing Scikit-Learn	343
Data Representation in Scikit-Learn	343
Scikit-Learn’s Estimator API	346
Application: Exploring Handwritten Digits	354
Summary	359
Hyperparameters and Model Validation	359
Thinking About Model Validation	359
Selecting the Best Model	363
Learning Curves	370
Validation in Practice: Grid Search	373
Summary	375
Feature Engineering	375

Categorical Features	376
Text Features	377
Image Features	378
Derived Features	378
Imputation of Missing Data	381
Feature Pipelines	381
In Depth: Naive Bayes Classification	382
Bayesian Classification	383
Gaussian Naive Bayes	383
Multinomial Naive Bayes	386
When to Use Naive Bayes	389
In Depth: Linear Regression	390
Simple Linear Regression	390
Basis Function Regression	392
Regularization	396
Example: Predicting Bicycle Traffic	400
In-Depth: Support Vector Machines	405
Motivating Support Vector Machines	405
Support Vector Machines: Maximizing the Margin	407
Example: Face Recognition	416
Support Vector Machine Summary	420
In-Depth: Decision Trees and Random Forests	421
Motivating Random Forests: Decision Trees	421
Ensembles of Estimators: Random Forests	426
Random Forest Regression	428
Example: Random Forest for Classifying Digits	430
Summary of Random Forests	432
In Depth: Principal Component Analysis	433
Introducing Principal Component Analysis	433
PCA as Noise Filtering	440
Example: Eigenfaces	442
Principal Component Analysis Summary	445
In-Depth: Manifold Learning	445
Manifold Learning: “HELLO”	446
Multidimensional Scaling (MDS)	447
MDS as Manifold Learning	450
Nonlinear Embeddings: Where MDS Fails	452
Nonlinear Manifolds: Locally Linear Embedding	453
Some Thoughts on Manifold Methods	455
Example: Isomap on Faces	456
Example: Visualizing Structure in Digits	460
In Depth: k-Means Clustering	462

Introducing k-Means	463
k-Means Algorithm: Expectation–Maximization	465
Examples	470
In Depth: Gaussian Mixture Models	476
Motivating GMM: Weaknesses of k-Means	477
Generalizing E–M: Gaussian Mixture Models	480
GMM as Density Estimation	484
Example: GMM for Generating New Data	488
In-Depth: Kernel Density Estimation	491
Motivating KDE: Histograms	491
Kernel Density Estimation in Practice	496
Example: KDE on a Sphere	498
Example: Not-So-Naive Bayes	501
Application: A Face Detection Pipeline	506
HOG Features	506
HOG in Action: A Simple Face Detector	507
Caveats and Improvements	512
Further Machine Learning Resources	514
Machine Learning in Python	514
General Machine Learning	515
Index.....	517

Preface

What Is Data Science?

This is a book about doing data science with Python, which immediately begs the question: what is *data science*? It's a surprisingly hard definition to nail down, especially given how ubiquitous the term has become. Vocal critics have variously dismissed the term as a superfluous label (after all, what science doesn't involve data?) or a simple buzzword that only exists to salt résumés and catch the eye of overzealous tech recruiters.

In my mind, these critiques miss something important. Data science, despite its hypeladen veneer, is perhaps the best label we have for the cross-disciplinary set of skills that are becoming increasingly important in many applications across industry and academia. This cross-disciplinary piece is key: in my mind, the best existing definition of data science is illustrated by Drew Conway's Data Science Venn Diagram, first published on his blog in September 2010 (see [Figure P-1](#)).

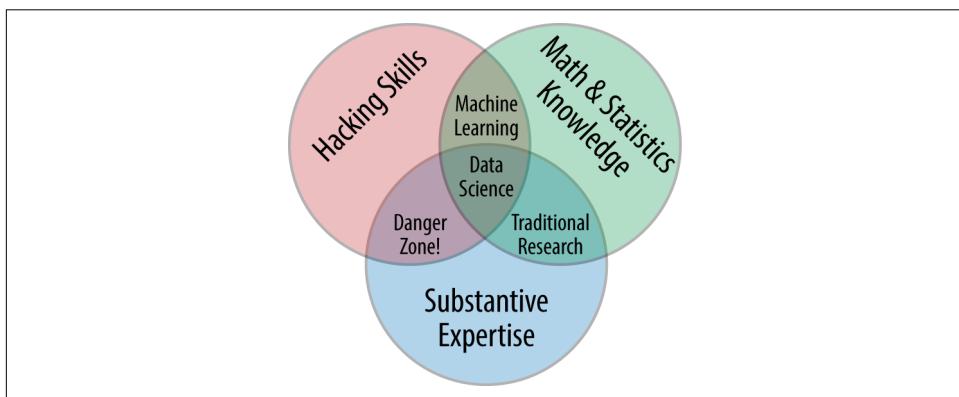


Figure P-1. Drew Conway's Data Science Venn Diagram

While some of the intersection labels are a bit tongue-in-cheek, this diagram captures the essence of what I think people mean when they say “data science”: it is fundamentally an *interdisciplinary* subject. Data science comprises three distinct and overlapping areas: the skills of a *statistician* who knows how to model and summarize datasets (which are growing ever larger); the skills of a *computer scientist* who can design and use algorithms to efficiently store, process, and visualize this data; and the *domain expertise*—what we might think of as “classical” training in a subject—necessary both to formulate the right questions and to put their answers in context.

With this in mind, I would encourage you to think of data science not as a new domain of knowledge to learn, but as a new set of skills that you can apply within your current area of expertise. Whether you are reporting election results, forecasting stock returns, optimizing online ad clicks, identifying microorganisms in microscope photos, seeking new classes of astronomical objects, or working with data in any other field, the goal of this book is to give you the ability to ask and answer new questions about your chosen subject area.

Who Is This Book For?

In my teaching both at the University of Washington and at various tech-focused conferences and meetups, one of the most common questions I have heard is this: “how should I learn Python?” The people asking are generally technically minded students, developers, or researchers, often with an already strong background in writing code and using computational and numerical tools. Most of these folks don’t want to learn Python per se, but want to learn the language with the aim of using it as a tool for data-intensive and computational science. While a large patchwork of videos, blog posts, and tutorials for this audience is available online, I’ve long been frustrated by the lack of a single good answer to this question; that is what inspired this book.

The book is not meant to be an introduction to Python or to programming in general; I assume the reader has familiarity with the Python language, including defining functions, assigning variables, calling methods of objects, controlling the flow of a program, and other basic tasks. Instead, it is meant to help Python users learn to use Python’s data science stack—libraries such as IPython, NumPy, Pandas, Matplotlib, Scikit-Learn, and related tools—to effectively store, manipulate, and gain insight from data.

Why Python?

Python has emerged over the last couple decades as a first-class tool for scientific computing tasks, including the analysis and visualization of large datasets. This may have come as a surprise to early proponents of the Python language: the language itself was not specifically designed with data analysis or scientific computing in mind.

The usefulness of Python for data science stems primarily from the large and active ecosystem of third-party packages: NumPy for manipulation of homogeneous array-based data, Pandas for manipulation of heterogeneous and labeled data, SciPy for common scientific computing tasks, Matplotlib for publication-quality visualizations, IPython for interactive execution and sharing of code, Scikit-Learn for machine learning, and many more tools that will be mentioned in the following pages.

If you are looking for a guide to the Python language itself, I would suggest the sister project to this book, *A Whirlwind Tour of the Python Language*. This short report provides a tour of the essential features of the Python language, aimed at data scientists who already are familiar with one or more other programming languages.

Python 2 Versus Python 3

This book uses the syntax of Python 3, which contains language enhancements that are not compatible with the 2.x series of Python. Though Python 3.0 was first released in 2008, adoption has been relatively slow, particularly in the scientific and web development communities. This is primarily because it took some time for many of the essential third-party packages and toolkits to be made compatible with the new language internals. Since early 2014, however, stable releases of the most important tools in the data science ecosystem have been fully compatible with both Python 2 and 3, and so this book will use the newer Python 3 syntax. However, the vast majority of code snippets in this book will also work without modification in Python 2: in cases where a Py2-incompatible syntax is used, I will make every effort to note it explicitly.

Outline of This Book

Each chapter of this book focuses on a particular package or tool that contributes a fundamental piece of the Python data science story.

IPython and Jupyter ([Chapter 1](#))

These packages provide the computational environment in which many Python-using data scientists work.

NumPy ([Chapter 2](#))

This library provides the `ndarray` object for efficient storage and manipulation of dense data arrays in Python.

Pandas ([Chapter 3](#))

This library provides the `DataFrame` object for efficient storage and manipulation of labeled/columnar data in Python.

Matplotlib ([Chapter 4](#))

This library provides capabilities for a flexible range of data visualizations in Python.

Scikit-Learn ([Chapter 5](#))

This library provides efficient and clean Python implementations of the most important and established machine learning algorithms.

The PyData world is certainly much larger than these five packages, and is growing every day. With this in mind, I make every attempt through these pages to provide references to other interesting efforts, projects, and packages that are pushing the boundaries of what can be done in Python. Nevertheless, these five are currently fundamental to much of the work being done in the Python data science space, and I expect they will remain important even as the ecosystem continues growing around them.

Using Code Examples

Supplemental material (code examples, figures, etc.) is available for download at <https://github.com/jakevdp/PythonDataScienceHandbook>. 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 Data Science Handbook* by Jake VanderPlas (O'Reilly). Copyright 2017 Jake VanderPlas, 978-1-491-91205-8.”

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.

Installation Considerations

Installing Python and the suite of libraries that enable scientific computing is straightforward. This section will outline some of the considerations to keep in mind when setting up your computer.

Though there are various ways to install Python, the one I would suggest for use in data science is the Anaconda distribution, which works similarly whether you use Windows, Linux, or Mac OS X. The Anaconda distribution comes in two flavors:

- **Miniconda** gives you the Python interpreter itself, along with a command-line tool called *conda* that operates as a cross-platform package manager geared

toward Python packages, similar in spirit to the apt or yum tools that Linux users might be familiar with.

- **Anaconda** includes both Python and conda, and additionally bundles a suite of other preinstalled packages geared toward scientific computing. Because of the size of this bundle, expect the installation to consume several gigabytes of disk space.

Any of the packages included with Anaconda can also be installed manually on top of Miniconda; for this reason I suggest starting with Miniconda.

To get started, download and install the Miniconda package (make sure to choose a version with Python 3), and then install the core packages used in this book:

```
[~]$ conda install numpy pandas scikit-learn matplotlib seaborn ipython-notebook
```

Throughout the text, we will also make use of other, more specialized tools in Python’s scientific ecosystem; installation is usually as easy as typing **conda install packagename**. For more information on conda, including information about creating and using conda environments (which I would *highly* recommend), refer to [conda’s online documentation](#).

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.

O’Reilly Safari



Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit <http://oreilly.com/safari>.

How to Contact Us

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

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <http://bit.ly/python-data-sci-handbook>.

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

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

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

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

IPython: Beyond Normal Python

There are many options for development environments for Python, and I'm often asked which one I use in my own work. My answer sometimes surprises people: my preferred environment is IPython plus a text editor (in my case, Emacs or Atom depending on my mood). IPython (short for *Interactive Python*) was started in 2001 by Fernando Perez as an enhanced Python interpreter, and has since grown into a project aiming to provide, in Perez's words, "Tools for the entire lifecycle of research computing." If Python is the engine of our data science task, you might think of IPython as the interactive control panel.

As well as being a useful interactive interface to Python, IPython also provides a number of useful syntactic additions to the language; we'll cover the most useful of these additions here. In addition, IPython is closely tied with the Jupyter project, which provides a browser-based notebook that is useful for development, collaboration, sharing, and even publication of data science results. The IPython notebook is actually a special case of the broader Jupyter notebook structure, which encompasses notebooks for Julia, R, and other programming languages. As an example of the usefulness of the notebook format, look no further than the page you are reading: the entire manuscript for this book was composed as a set of IPython notebooks.

IPython is about using Python effectively for interactive scientific and data-intensive computing. This chapter will start by stepping through some of the IPython features that are useful to the practice of data science, focusing especially on the syntax it offers beyond the standard features of Python. Next, we will go into a bit more depth on some of the more useful "magic commands" that can speed up common tasks in creating and using data science code. Finally, we will touch on some of the features of the notebook that make it useful in understanding data and sharing results.

Shell or Notebook?

There are two primary means of using IPython that we'll discuss in this chapter: the IPython shell and the IPython notebook. The bulk of the material in this chapter is relevant to both, and the examples will switch between them depending on what is most convenient. In the few sections that are relevant to just one or the other, I will explicitly state that fact. Before we start, some words on how to launch the IPython shell and IPython notebook.

Launching the IPython Shell

This chapter, like most of this book, is not designed to be absorbed passively. I recommend that as you read through it, you follow along and experiment with the tools and syntax we cover: the muscle-memory you build through doing this will be far more useful than the simple act of reading about it. Start by launching the IPython interpreter by typing `ipython` on the command line; alternatively, if you've installed a distribution like Anaconda or EPD, there may be a launcher specific to your system (we'll discuss this more fully in “[Help and Documentation in IPython](#)” on page 3).

Once you do this, you should see a prompt like the following:

```
IPython 4.0.1 -- 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]:
```

With that, you're ready to follow along.

Launching the Jupyter Notebook

The Jupyter notebook is a browser-based graphical interface to the IPython shell, and builds on it a rich set of dynamic display capabilities. As well as executing Python/IPython statements, the notebook allows the user to include formatted text, static and dynamic visualizations, mathematical equations, JavaScript widgets, and much more. Furthermore, these documents can be saved in a way that lets other people open them and execute the code on their own systems.

Though the IPython notebook is viewed and edited through your web browser window, it must connect to a running Python process in order to execute code. To start this process (known as a “kernel”), run the following command in your system shell:

```
$ jupyter notebook
```

This command will launch a local web server that will be visible to your browser. It immediately spits out a log showing what it is doing; that log will look something like this:

```
$ jupyter notebook
[NotebookApp] Serving notebooks from local directory: /Users/jakevdp/...
[NotebookApp] 0 active kernels
[NotebookApp] The IPython Notebook is running at: http://localhost:8888/
[NotebookApp] Use Control-C to stop this server and shut down all kernels...
```

Upon issuing the command, your default browser should automatically open and navigate to the listed local URL; the exact address will depend on your system. If the browser does not open automatically, you can open a window and manually open this address (<http://localhost:8888/> in this example).

Help and Documentation in IPython

If you read no other section in this chapter, read this one: I find the tools discussed here to be the most transformative contributions of IPython to my daily workflow.

When a technologically minded person is asked to help a friend, family member, or colleague with a computer problem, most of the time it's less a matter of knowing the answer as much as knowing how to quickly find an unknown answer. In data science it's the same: searchable web resources such as online documentation, mailing-list threads, and Stack Overflow answers contain a wealth of information, even (especially?) if it is a topic you've found yourself searching before. Being an effective practitioner of data science is less about memorizing the tool or command you should use for every possible situation, and more about learning to effectively find the information you don't know, whether through a web search engine or another means.

One of the most useful functions of IPython/Jupyter is to shorten the gap between the user and the type of documentation and search that will help them do their work effectively. While web searches still play a role in answering complicated questions, an amazing amount of information can be found through IPython alone. Some examples of the questions IPython can help answer in a few keystrokes:

- How do I call this function? What arguments and options does it have?
- What does the source code of this Python object look like?
- What is in this package I imported? What attributes or methods does this object have?

Here we'll discuss IPython's tools to quickly access this information, namely the `?` character to explore documentation, the `??` characters to explore source code, and the Tab key for autocompletion.

Accessing Documentation with `?`

The Python language and its data science ecosystem are built with the user in mind, and one big part of that is access to documentation. Every Python object contains the

reference to a string, known as a *docstring*, which in most cases will contain a concise summary of the object and how to use it. Python has a built-in `help()` function that can access this information and print the results. For example, to see the documentation of the built-in `len` function, you can do the following:

```
In [1]: help(len)
Help on built-in function len in module builtins:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or mapping.
```

Depending on your interpreter, this information may be displayed as inline text, or in some separate pop-up window.

Because finding help on an object is so common and useful, IPython introduces the `?` character as a shorthand for accessing this documentation and other relevant information:

```
In [2]: len?
Type:       builtin_function_or_method
String form: <built-in function len>
Namespace:  Python builtin
Docstring:
len(object) -> integer

    Return the number of items of a sequence or mapping.
```

This notation works for just about anything, including object methods:

```
In [3]: L = [1, 2, 3]
In [4]: L.insert?
Type:       builtin_function_or_method
String form: <built-in method insert of list object at 0x1024b8ea8>
Docstring:  L.insert(index, object) -- insert object before index
```

or even objects themselves, with the documentation from their type:

```
In [5]: L?
Type:       list
String form: [1, 2, 3]
Length:     3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

Importantly, this will even work for functions or other objects you create yourself! Here we'll define a small function with a docstring:

```
In [6]: def square(a):
....:     """Return the square of a."""

```

```
....:     return a ** 2
....:
```

Note that to create a docstring for our function, we simply placed a string literal in the first line. Because docstrings are usually multiple lines, by convention we used Python's triple-quote notation for multiline strings.

Now we'll use the ? mark to find this docstring:

```
In [7]: square?
Type:      function
String form: <function square at 0x103713cb0>
Definition: square(a)
Docstring:  Return the square of a.
```

This quick access to documentation via docstrings is one reason you should get in the habit of always adding such inline documentation to the code you write!

Accessing Source Code with ??

Because the Python language is so easily readable, you can usually gain another level of insight by reading the source code of the object you're curious about. IPython provides a shortcut to the source code with the double question mark (??):

```
In [8]: square??
Type:      function
String form: <function square at 0x103713cb0>
Definition: square(a)
Source:
def square(a):
    "Return the square of a"
    return a ** 2
```

For simple functions like this, the double question mark can give quick insight into the under-the-hood details.

If you play with this much, you'll notice that sometimes the ?? suffix doesn't display any source code: this is generally because the object in question is not implemented in Python, but in C or some other compiled extension language. If this is the case, the ?? suffix gives the same output as the ? suffix. You'll find this particularly with many of Python's built-in objects and types, for example `len` from above:

```
In [9]: len??
Type:      builtin_function_or_method
String form: <built-in function len>
Namespace:  Python builtin
Docstring:
len(object) -> integer

Return the number of items of a sequence or mapping.
```

Using `? and/or ??` gives a powerful and quick interface for finding information about what any Python function or module does.

Exploring Modules with Tab Completion

IPython's other useful interface is the use of the Tab key for autocomplete and exploration of the contents of objects, modules, and namespaces. In the examples that follow, we'll use `<TAB>` to indicate when the Tab key should be pressed.

Tab completion of object contents

Every Python object has various attributes and methods associated with it. Like with the `help` function discussed before, Python has a built-in `dir` function that returns a list of these, but the tab-completion interface is much easier to use in practice. To see a list of all available attributes of an object, you can type the name of the object followed by a period (.) character and the Tab key:

```
In [10]: L.<TAB>
L.append  L.copy    L.extend  L.insert  L.remove  L.sort
L.clear   L.count   L.index   L.pop     L.reverse
```

To narrow down the list, you can type the first character or several characters of the name, and the Tab key will find the matching attributes and methods:

```
In [10]: L.c<TAB>
L.clear  L.copy   L.count

In [10]: L.co<TAB>
L.copy  L.count
```

If there is only a single option, pressing the Tab key will complete the line for you. For example, the following will instantly be replaced with `L.count`:

```
In [10]: L.cou<TAB>
```

Though Python has no strictly enforced distinction between public/external attributes and private/internal attributes, by convention a preceding underscore is used to denote such methods. For clarity, these private methods and special methods are omitted from the list by default, but it's possible to list them by explicitly typing the underscore:

```
In [10]: L._<TAB>
L.__add__      L.__gt__      L.__reduce__
L.__class__   L.__hash__   L.__reduce_ex__
```

For brevity, we've only shown the first couple lines of the output. Most of these are Python's special double-underscore methods (often nicknamed "dunder" methods).

Tab completion when importing

Tab completion is also useful when importing objects from packages. Here we'll use it to find all possible imports in the `itertools` package that start with `co`:

```
In [10]: from itertools import co<TAB>
combinations           compress
combinations_with_replacement  count
```

Similarly, you can use tab completion to see which imports are available on your system (this will change depending on which third-party scripts and modules are visible to your Python session):

```
In [10]: import <TAB>
Display all 399 possibilities? (y or n)
Crypto          dis          py_compile
Cython          distutils    pyclbr
...
difflib         pwd          zmq
In [10]: import h<TAB>
hashlib         hmac         http
heapq           html         husl
```

(Note that for brevity, I did not print here all 399 importable packages and modules on my system.)

Beyond tab completion: Wildcard matching

Tab completion is useful if you know the first few characters of the object or attribute you're looking for, but is little help if you'd like to match characters at the middle or end of the word. For this use case, IPython provides a means of wildcard matching for names using the `*` character.

For example, we can use this to list every object in the namespace that ends with `Warning`:

```
In [10]: *Warning?
BytesWarning      RuntimeWarning
DeprecationWarning SyntaxWarning
FutureWarning    UnicodeWarning
ImportWarning    UserWarning
PendingDeprecationWarning Warning
ResourceWarning
```

Notice that the `*` character matches any string, including the empty string.

Similarly, suppose we are looking for a string method that contains the word `find` somewhere in its name. We can search for it this way:

```
In [10]: str.*find*?  
str.find  
str.rfind
```

I find this type of flexible wildcard search can be very useful for finding a particular command when I'm getting to know a new package or reacquainting myself with a familiar one.

Keyboard Shortcuts in the IPython Shell

If you spend any amount of time on the computer, you've probably found a use for keyboard shortcuts in your workflow. Most familiar perhaps are Cmd-C and Cmd-V (or Ctrl-C and Ctrl-V) for copying and pasting in a wide variety of programs and systems. Power users tend to go even further: popular text editors like Emacs, Vim, and others provide users an incredible range of operations through intricate combinations of keystrokes.

The IPython shell doesn't go this far, but does provide a number of keyboard shortcuts for fast navigation while you're typing commands. These shortcuts are not in fact provided by IPython itself, but through its dependency on the GNU Readline library: thus, some of the following shortcuts may differ depending on your system configuration. Also, while some of these shortcuts do work in the browser-based notebook, this section is primarily about shortcuts in the IPython shell.

Once you get accustomed to these, they can be very useful for quickly performing certain commands without moving your hands from the "home" keyboard position. If you're an Emacs user or if you have experience with Linux-style shells, the following will be very familiar. We'll group these shortcuts into a few categories: *navigation shortcuts*, *text entry shortcuts*, *command history shortcuts*, and *miscellaneous shortcuts*.

Navigation Shortcuts

While the use of the left and right arrow keys to move backward and forward in the line is quite obvious, there are other options that don't require moving your hands from the "home" keyboard position:

Keystroke	Action
Ctrl-a	Move cursor to the beginning of the line
Ctrl-e	Move cursor to the end of the line
Ctrl-b (or the left arrow key)	Move cursor back one character
Ctrl-f (or the right arrow key)	Move cursor forward one character

Text Entry Shortcuts

While everyone is familiar with using the Backspace key to delete the previous character, reaching for the key often requires some minor finger gymnastics, and it only deletes a single character at a time. In IPython there are several shortcuts for removing some portion of the text you're typing. The most immediately useful of these are the commands to delete entire lines of text. You'll know these have become second nature if you find yourself using a combination of Ctrl-b and Ctrl-d instead of reaching for the Backspace key to delete the previous character!

Keystroke	Action
Backspace key	Delete previous character in line
Ctrl-d	Delete next character in line
Ctrl-k	Cut text from cursor to end of line
Ctrl-u	Cut text from beginning fo line to cursor
Ctrl-y	Yank (i.e., paste) text that was previously cut
Ctrl-t	Transpose (i.e., switch) previous two characters

Command History Shortcuts

Perhaps the most impactful shortcuts discussed here are the ones IPython provides for navigating the command history. This command history goes beyond your current IPython session: your entire command history is stored in a SQLite database in your IPython profile directory. The most straightforward way to access these is with the up and down arrow keys to step through the history, but other options exist as well:

Keystroke	Action
Ctrl-p (or the up arrow key)	Access previous command in history
Ctrl-n (or the down arrow key)	Access next command in history
Ctrl-r	Reverse-search through command history

The reverse-search can be particularly useful. Recall that in the previous section we defined a function called `square`. Let's reverse-search our Python history from a new IPython shell and find this definition again. When you press Ctrl-r in the IPython terminal, you'll see the following prompt:

```
In [1]:  
(reverse-i-search)`':
```

If you start typing characters at this prompt, IPython will auto-fill the most recent command, if any, that matches those characters:

```
In [1]:  
(reverse-i-search)`sqa': square??
```

At any point, you can add more characters to refine the search, or press Ctrl-r again to search further for another command that matches the query. If you followed along in the previous section, pressing Ctrl-r twice more gives:

```
In [1]:  
(reverse-i-search)`sqa': def square(a):  
    """Return the square of a"""  
    return a ** 2
```

Once you have found the command you're looking for, press Return and the search will end. We can then use the retrieved command, and carry on with our session:

```
In [1]: def square(a):  
    """Return the square of a"""  
    return a ** 2
```

```
In [2]: square(2)  
Out[2]: 4
```

Note that you can also use Ctrl-p/Ctrl-n or the up/down arrow keys to search through history, but only by matching characters at the beginning of the line. That is, if you type `def` and then press Ctrl-p, it would find the most recent command (if any) in your history that begins with the characters `def`.

Miscellaneous Shortcuts

Finally, there are a few miscellaneous shortcuts that don't fit into any of the preceding categories, but are nevertheless useful to know:

Keystroke	Action
Ctrl-l	Clear terminal screen
Ctrl-c	Interrupt current Python command
Ctrl-d	Exit IPython session

The Ctrl-c shortcut in particular can be useful when you inadvertently start a very long-running job.

While some of the shortcuts discussed here may seem a bit tedious at first, they quickly become automatic with practice. Once you develop that muscle memory, I suspect you will even find yourself wishing they were available in other contexts.

IPython Magic Commands

The previous two sections showed how IPython lets you use and explore Python efficiently and interactively. Here we'll begin discussing some of the enhancements that

IPython adds on top of the normal Python syntax. These are known in IPython as *magic commands*, and are prefixed by the % character. These magic commands are designed to succinctly solve various common problems in standard data analysis. Magic commands come in two flavors: *line magics*, which are denoted by a single % prefix and operate on a single line of input, and *cell magics*, which are denoted by a double %% prefix and operate on multiple lines of input. We'll demonstrate and discuss a few brief examples here, and come back to more focused discussion of several useful magic commands later in the chapter.

Pasting Code Blocks: %paste and %cpaste

When you're working in the IPython interpreter, one common gotcha is that pasting multiline code blocks can lead to unexpected errors, especially when indentation and interpreter markers are involved. A common case is that you find some example code on a website and want to paste it into your interpreter. Consider the following simple function:

```
>>> def donothing(x):
...     return x
```

The code is formatted as it would appear in the Python interpreter, and if you copy and paste this directly into IPython you get an error:

```
In [2]: >>> def donothing(x):
...     ...     return x
...     ...
File "<ipython-input-20-5a66c8964687>", line 2
...     return x
          ^
SyntaxError: invalid syntax
```

In the direct paste, the interpreter is confused by the additional prompt characters. But never fear—IPython's `%paste` magic function is designed to handle this exact type of multiline, marked-up input:

```
In [3]: %paste
>>> def donothing(x):
...     return x

## -- End pasted text --
```

The `%paste` command both enters and executes the code, so now the function is ready to be used:

```
In [4]: donothing(10)
Out[4]: 10
```

A command with a similar intent is `%cpaste`, which opens up an interactive multiline prompt in which you can paste one or more chunks of code to be executed in a batch:

```
In [5]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:>>> def donothing(x):
:...     return x
:--
```

These magic commands, like others we'll see, make available functionality that would be difficult or impossible in a standard Python interpreter.

Running External Code: %run

As you begin developing more extensive code, you will likely find yourself working in both IPython for interactive exploration, as well as a text editor to store code that you want to reuse. Rather than running this code in a new window, it can be convenient to run it within your IPython session. This can be done with the `%run` magic.

For example, imagine you've created a `myscript.py` file with the following contents:

```
#-----
# file: myscript.py

def square(x):
    """square a number"""
    return x ** 2

for N in range(1, 4):
    print(N, "squared is", square(N))
```

You can execute this from your IPython session as follows:

```
In [6]: %run myscript.py
1 squared is 1
2 squared is 4
3 squared is 9
```

Note also that after you've run this script, any functions defined within it are available for use in your IPython session:

```
In [7]: square(5)
Out[7]: 25
```

There are several options to fine-tune how your code is run; you can see the documentation in the normal way, by typing `%run?` in the IPython interpreter.

Timing Code Execution: %timeit

Another example of a useful magic function is `%timeit`, which will automatically determine the execution time of the single-line Python statement that follows it. For example, we may want to check the performance of a list comprehension:

```
In [8]: %timeit L = [n ** 2 for n in range(1000)]
1000 loops, best of 3: 325 µs per loop
```

The benefit of `%timeit` is that for short commands it will automatically perform multiple runs in order to attain more robust results. For multiline statements, adding a second % sign will turn this into a cell magic that can handle multiple lines of input. For example, here's the equivalent construction with a `for` loop:

```
In [9]: %%timeit
....: L = []
....: for n in range(1000):
....:     L.append(n ** 2)
....:
1000 loops, best of 3: 373 µs per loop
```

We can immediately see that list comprehensions are about 10% faster than the equivalent `for` loop construction in this case. We'll explore `%timeit` and other approaches to timing and profiling code in “[Profiling and Timing Code](#)” on page 25.

Help on Magic Functions: ?, %magic, and %lsmagic

Like normal Python functions, IPython magic functions have docstrings, and this useful documentation can be accessed in the standard manner. So, for example, to read the documentation of the `%timeit` magic, simply type this:

```
In [10]: %timeit?
```

Documentation for other functions can be accessed similarly. To access a general description of available magic functions, including some examples, you can type this:

```
In [11]: %magic
```

For a quick and simple list of all available magic functions, type this:

```
In [12]: %lsmagic
```

Finally, I'll mention that it is quite straightforward to define your own magic functions if you wish. We won't discuss it here, but if you are interested, see the references listed in “[More IPython Resources](#)” on page 30.

Input and Output History

Previously we saw that the IPython shell allows you to access previous commands with the up and down arrow keys, or equivalently the Ctrl-p/Ctrl-n shortcuts. Additionally, in both the shell and the notebook, IPython exposes several ways to obtain the output of previous commands, as well as string versions of the commands themselves. We'll explore those here.

IPython's In and Out Objects

By now I imagine you're quite familiar with the `In[1]:/Out[1]:` style prompts used by IPython. But it turns out that these are not just pretty decoration: they give a clue

as to how you can access previous inputs and outputs in your current session. Imagine you start a session that looks like this:

```
In [1]: import math  
  
In [2]: math.sin(2)  
Out[2]: 0.9092974268256817  
  
In [3]: math.cos(2)  
Out[3]: -0.4161468365471424
```

We've imported the built-in `math` package, then computed the sine and the cosine of the number 2. These inputs and outputs are displayed in the shell with `In`/`Out` labels, but there's more—IPython actually creates some Python variables called `In` and `Out` that are automatically updated to reflect this history:

```
In [4]: print(In)  
[], 'import math', 'math.sin(2)', 'math.cos(2)', 'print(In)'  
  
In [5]: Out  
Out[5]: {2: 0.9092974268256817, 3: -0.4161468365471424}
```

The `In` object is a list, which keeps track of the commands in order (the first item in the list is a placeholder so that `In[1]` can refer to the first command):

```
In [6]: print(In[1])  
import math
```

The `Out` object is not a list but a dictionary mapping input numbers to their outputs (if any):

```
In [7]: print(Out[2])  
0.9092974268256817
```

Note that not all operations have outputs: for example, `import` statements and `print` statements don't affect the output. The latter may be surprising, but makes sense if you consider that `print` is a function that returns `None`; for brevity, any command that returns `None` is not added to `Out`.

Where this can be useful is if you want to interact with past results. For example, let's check the sum of `sin(2) ** 2` and `cos(2) ** 2` using the previously computed results:

```
In [8]: Out[2] ** 2 + Out[3] ** 2  
Out[8]: 1.0
```

The result is `1.0` as we'd expect from the well-known trigonometric identity. In this case, using these previous results probably is not necessary, but it can become very handy if you execute a very expensive computation and want to reuse the result!

Underscore Shortcuts and Previous Outputs

The standard Python shell contains just one simple shortcut for accessing previous output; the variable `_` (i.e., a single underscore) is kept updated with the previous output; this works in IPython as well:

```
In [9]: print(_)
1.0
```

But IPython takes this a bit further—you can use a double underscore to access the second-to-last output, and a triple underscore to access the third-to-last output (skipping any commands with no output):

```
In [10]: print(__)
-0.4161468365471424
```

```
In [11]: print(__)
0.9092974268256817
```

IPython stops there: more than three underscores starts to get a bit hard to count, and at that point it's easier to refer to the output by line number.

There is one more shortcut we should mention, however—a shorthand for `Out[X]` is `_X` (i.e., a single underscore followed by the line number):

```
In [12]: Out[2]
Out[12]: 0.9092974268256817
```

```
In [13]: _2
Out[13]: 0.9092974268256817
```

Suppressing Output

Sometimes you might wish to suppress the output of a statement (this is perhaps most common with the plotting commands that we'll explore in [Chapter 4](#)). Or maybe the command you're executing produces a result that you'd prefer not to store in your output history, perhaps so that it can be deallocated when other references are removed. The easiest way to suppress the output of a command is to add a semicolon to the end of the line:

```
In [14]: math.sin(2) + math.cos(2);
```

Note that the result is computed silently, and the output is neither displayed on the screen or stored in the `Out` dictionary:

```
In [15]: 14 in Out
Out[15]: False
```

Related Magic Commands

For accessing a batch of previous inputs at once, the `%history` magic command is very helpful. Here is how you can print the first four inputs:

```
In [16]: %history -n 1-4
1: import math
2: math.sin(2)
3: math.cos(2)
4: print(In)
```

As usual, you can type `%history?` for more information and a description of options available. Other similar magic commands are `%rerun` (which will re-execute some portion of the command history) and `%save` (which saves some set of the command history to a file). For more information, I suggest exploring these using the `?` help functionality discussed in “[Help and Documentation in IPython](#)” on page 3.

IPython and Shell Commands

When working interactively with the standard Python interpreter, one of the frustrations you’ll face is the need to switch between multiple windows to access Python tools and system command-line tools. IPython bridges this gap, and gives you a syntax for executing shell commands directly from within the IPython terminal. The magic happens with the exclamation point: anything appearing after `!` on a line will be executed not by the Python kernel, but by the system command line.

The following assumes you’re on a Unix-like system, such as Linux or Mac OS X. Some of the examples that follow will fail on Windows, which uses a different type of shell by default (though with the 2016 announcement of native Bash shells on Windows, soon this may no longer be an issue!). If you’re unfamiliar with shell commands, I’d suggest reviewing the [Shell Tutorial](#) put together by the always excellent Software Carpentry Foundation.

Quick Introduction to the Shell

A full intro to using the shell/terminal/command line is well beyond the scope of this chapter, but for the uninitiated we will offer a quick introduction here. The shell is a way to interact textually with your computer. Ever since the mid-1980s, when Microsoft and Apple introduced the first versions of their now ubiquitous graphical operating systems, most computer users have interacted with their operating system through familiar clicking of menus and drag-and-drop movements. But operating systems existed long before these graphical user interfaces, and were primarily controlled through sequences of text input: at the prompt, the user would type a command, and the computer would do what the user told it to. Those early prompt

systems are the precursors of the shells and terminals that most active data scientists still use today.

Someone unfamiliar with the shell might ask why you would bother with this, when you can accomplish many results by simply clicking on icons and menus. A shell user might reply with another question: why hunt icons and click menus when you can accomplish things much more easily by typing? While it might sound like a typical tech preference impasse, when moving beyond basic tasks it quickly becomes clear that the shell offers much more control of advanced tasks, though admittedly the learning curve can intimidate the average computer user.

As an example, here is a sample of a Linux/OS X shell session where a user explores, creates, and modifies directories and files on their system (osx:~ \$ is the prompt, and everything after the \$ sign is the typed command; text that is preceded by a # is meant just as description, rather than something you would actually type in):

```
osx:~ $ echo "hello world"          # echo is like Python's print function
hello world

osx:~ $ pwd                         # pwd = print working directory
/home/jake                           # this is the "path" that we're in

osx:~ $ ls                           # ls = list working directory contents
notebooks  projects

osx:~ $ cd projects/                 # cd = change directory

osx:projects $ pwd                  # pwd = print working directory
/home/jake/projects

osx:projects $ ls                   # ls = list working directory contents
datasci_book  mpld3  myproject.txt

osx:projects $ mkdir myproject      # mkdir = make new directory

osx:projects $ cd myproject/
osx:myproject $ mv .. /myproject.txt ./  # mv = move file. Here we're moving the
                                         # file myproject.txt from one directory
                                         # up (..) to the current directory (./)
osx:myproject $ ls
myproject.txt
```

Notice that all of this is just a compact way to do familiar operations (navigating a directory structure, creating a directory, moving a file, etc.) by typing commands rather than clicking icons and menus. Note that with just a few commands (`pwd`, `ls`, `cd`, `mkdir`, and `cp`) you can do many of the most common file operations. It's when you go beyond these basics that the shell approach becomes really powerful.

Shell Commands in IPython

You can use any command that works at the command line in IPython by prefixing it with the ! character. For example, the ls, pwd, and echo commands can be run as follows:

```
In [1]: !ls  
myproject.txt  
  
In [2]: !pwd  
/home/jake/projects/myproject  
  
In [3]: !echo "printing from the shell"  
printing from the shell
```

Passing Values to and from the Shell

Shell commands can not only be called from IPython, but can also be made to interact with the IPython namespace. For example, you can save the output of any shell command to a Python list using the assignment operator:

```
In [4]: contents = !ls  
  
In [5]: print(contents)  
['myproject.txt']  
  
In [6]: directory = !pwd  
  
In [7]: print(directory)  
['/Users/jakevdp/notebooks/tmp/myproject']
```

Note that these results are not returned as lists, but as a special shell return type defined in IPython:

```
In [8]: type(directory)  
IPython.utils.text.SList
```

This looks and acts a lot like a Python list, but has additional functionality, such as the grep and fields methods and the s, n, and p properties that allow you to search, filter, and display the results in convenient ways. For more information on these, you can use IPython's built-in help features.

Communication in the other direction—passing Python variables into the shell—is possible through the {varname} syntax:

```
In [9]: message = "hello from Python"  
  
In [10]: !echo {message}  
hello from Python
```

The curly braces contain the variable name, which is replaced by the variable's contents in the shell command.

Shell-Related Magic Commands

If you play with IPython's shell commands for a while, you might notice that you cannot use `!cd` to navigate the filesystem:

```
In [11]: !pwd  
/home/jake/projects/myproject
```

```
In [12]: !cd ..
```

```
In [13]: !pwd  
/home/jake/projects/myproject
```

The reason is that shell commands in the notebook are executed in a temporary subshell. If you'd like to change the working directory in a more enduring way, you can use the `%cd` magic command:

```
In [14]: %cd ..  
/home/jake/projects
```

In fact, by default you can even use this without the `%` sign:

```
In [15]: cd myproject  
/home/jake/projects/myproject
```

This is known as an `automagic` function, and this behavior can be toggled with the `%automagic` magic function.

Besides `%cd`, other available shell-like magic functions are `%cat`, `%cp`, `%env`, `%ls`, `%man`, `%mkdir`, `%more`, `%mv`, `%pwd`, `%rm`, and `%rmdir`, any of which can be used without the `%` sign if `automagic` is on. This makes it so that you can almost treat the IPython prompt as if it's a normal shell:

```
In [16]: mkdir tmp
```

```
In [17]: ls  
myproject.txt tmp/
```

```
In [18]: cp myproject.txt tmp/
```

```
In [19]: ls tmp  
myproject.txt
```

```
In [20]: rm -r tmp
```

This access to the shell from within the same terminal window as your Python session means that there is a lot less switching back and forth between interpreter and shell as you write your Python code.

Errors and Debugging

Code development and data analysis always require a bit of trial and error, and IPython contains tools to streamline this process. This section will briefly cover some options for controlling Python's exception reporting, followed by exploring tools for debugging errors in code.

Controlling Exceptions: %xmode

Most of the time when a Python script fails, it will raise an exception. When the interpreter hits one of these exceptions, information about the cause of the error can be found in the *traceback*, which can be accessed from within Python. With the `%xmode` magic function, IPython allows you to control the amount of information printed when the exception is raised. Consider the following code:

```
In[1]: def func1(a, b):
         return a / b

def func2(x):
    a = x
    b = x - 1
    return func1(a, b)

In[2]: func2(1)

-----
ZeroDivisionError                                Traceback (most recent call last)

<ipython-input-2-b2e110f6fc8f>; in <module>()
----> 1 func2(1)

<ipython-input-1-d849e34d61fb> in func2(x)
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)

<ipython-input-1-d849e34d61fb> in func1(a, b)
      1 def func1(a, b):
----> 2     return a / b
      3
      4 def func2(x):
      5     a = x

ZeroDivisionError: division by zero
```

Calling `func2` results in an error, and reading the printed trace lets us see exactly what happened. By default, this trace includes several lines showing the context of each

step that led to the error. Using the `%xmode` magic function (short for *exception mode*), we can change what information is printed.

`%xmode` takes a single argument, the mode, and there are three possibilities: Plain, Context, and Verbose. The default is Context, and gives output like that just shown. Plain is more compact and gives less information:

```
In[3]: %xmode Plain
Exception reporting mode: Plain

In[4]: func2(1)
-----
Traceback (most recent call last):

File "<ipython-input-4-b2e110f6fc8f>", line 1, in <module>
  func2(1)

File "<ipython-input-1-d849e34d61fb>", line 7, in func2
    return func1(a, b)

File "<ipython-input-1-d849e34d61fb>", line 2, in func1
    return a / b

ZeroDivisionError: division by zero
```

The Verbose mode adds some extra information, including the arguments to any functions that are called:

```
In[5]: %xmode Verbose
Exception reporting mode: Verbose

In[6]: func2(1)
-----
ZeroDivisionError                               Traceback (most recent call last)

<ipython-input-6-b2e110f6fc8f> in <module>()
      1 func2(1)
      2     global func2 = <function func2 at 0x103729320>

<ipython-input-1-d849e34d61fb> in func2(x=1)
      5     a = x
      6     b = x - 1
      7     return func1(a, b)
      8     global func1 = <function func1 at 0x1037294d0>
      9     a = 1
     10     b = 0
```

```
<ipython-input-1-d849e34d61fb> in func1(a=1, b=0)
      1 def func1(a, b):
----> 2     return a / b
          a = 1
          b = 0
          3
          4 def func2(x):
          5     a = x

ZeroDivisionError: division by zero
```

This extra information can help you narrow in on why the exception is being raised. So why not use the `Verbose` mode all the time? As code gets complicated, this kind of traceback can get extremely long. Depending on the context, sometimes the brevity of `Default` mode is easier to work with.

Debugging: When Reading Tracebacks Is Not Enough

The standard Python tool for interactive debugging is `pdb`, the Python debugger. This debugger lets the user step through the code line by line in order to see what might be causing a more difficult error. The IPython-enhanced version of this is `ipdb`, the IPython debugger.

There are many ways to launch and use both these debuggers; we won't cover them fully here. Refer to the online documentation of these two utilities to learn more.

In IPython, perhaps the most convenient interface to debugging is the `%debug` magic command. If you call it after hitting an exception, it will automatically open an interactive debugging prompt at the point of the exception. The `ipdb` prompt lets you explore the current state of the stack, explore the available variables, and even run Python commands!

Let's look at the most recent exception, then do some basic tasks—print the values of `a` and `b`, and type `quit` to quit the debugging session:

```
In[7]: %debug
> <ipython-input-1-d849e34d61fb>(2)func1()
      1 def func1(a, b):
----> 2     return a / b
          3

ipdb> print(a)
1
ipdb> print(b)
0
ipdb> quit
```

The interactive debugger allows much more than this, though—we can even step up and down through the stack and explore the values of variables there:

```
In[8]: %debug
> <ipython-input-1-d849e34d61fb>(2)func1()
    1 def func1(a, b):
----> 2     return a / b
      3

ipdb> up
> <ipython-input-1-d849e34d61fb>(7)func2()
    5     a = x
    6     b = x - 1
----> 7     return func1(a, b)

ipdb> print(x)
1
ipdb> up
> <ipython-input-6-b2e110f6fc8f>(1)<module>()
----> 1 func2(1)

ipdb> down
> <ipython-input-1-d849e34d61fb>(7)func2()
    5     a = x
    6     b = x - 1
----> 7     return func1(a, b)

ipdb> quit
```

This allows you to quickly find out not only what caused the error, but also what function calls led up to the error.

If you'd like the debugger to launch automatically whenever an exception is raised, you can use the %pdb magic function to turn on this automatic behavior:

```
In[9]: %xmode Plain
        %pdb on
        func2(1)

Exception reporting mode: Plain
Automatic pdb calling has been turned ON

Traceback (most recent call last):

File "<ipython-input-9-569a67d2d312>", line 3, in <module>
  func2(1)

File "<ipython-input-1-d849e34d61fb>", line 7, in func2
  return func1(a, b)
```

```
File "<ipython-input-1-d849e34d61fb>", line 2, in func1
    return a / b

ZeroDivisionError: division by zero

> <ipython-input-1-d849e34d61fb>(2)func1()
  1 def func1(a, b):
----> 2     return a / b
      3

ipdb> print(b)
0
ipdb> quit
```

Finally, if you have a script that you'd like to run from the beginning in interactive mode, you can run it with the command `%run -d`, and use the `next` command to step through the lines of code interactively.

Partial list of debugging commands

There are many more available commands for interactive debugging than we've listed here; the following table contains a description of some of the more common and useful ones:

Command	Description
<code>list</code>	Show the current location in the file
<code>h(elp)</code>	Show a list of commands, or find help on a specific command
<code>q(uit)</code>	Quit the debugger and the program
<code>c(ontinue)</code>	Quit the debugger; continue in the program
<code>n(ext)</code>	Go to the next step of the program
<code><enter></code>	Repeat the previous command
<code>print</code>	Print variables
<code>s(tep)</code>	Step into a subroutine
<code>return</code>	Return out of a subroutine

For more information, use the `help` command in the debugger, or take a look at [ipdb's online documentation](#).

Profiling and Timing Code

In the process of developing code and creating data processing pipelines, there are often trade-offs you can make between various implementations. Early in developing your algorithm, it can be counterproductive to worry about such things. As Donald Knuth famously quipped, “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.”

But once you have your code working, it can be useful to dig into its efficiency a bit. Sometimes it’s useful to check the execution time of a given command or set of commands; other times it’s useful to dig into a multiline process and determine where the bottleneck lies in some complicated series of operations. IPython provides access to a wide array of functionality for this kind of timing and profiling of code. Here we’ll discuss the following IPython magic commands:

`%time`

Time the execution of a single statement

`%timeit`

Time repeated execution of a single statement for more accuracy

`%prun`

Run code with the profiler

`%lprun`

Run code with the line-by-line profiler

`%memit`

Measure the memory use of a single statement

`%mprun`

Run code with the line-by-line memory profiler

The last four commands are not bundled with IPython—you’ll need to install the `line_profiler` and `memory_profiler` extensions, which we will discuss in the following sections.

Timing Code Snippets: `%timeit` and `%time`

We saw the `%timeit` line magic and `%%timeit` cell magic in the introduction to magic functions in “[IPython Magic Commands](#)” on page 10; `%%timeit` can be used to time the repeated execution of snippets of code:

```
In[1]: %timeit sum(range(100))  
100000 loops, best of 3: 1.54 µs per loop
```

Note that because this operation is so fast, `%timeit` automatically does a large number of repetitions. For slower commands, `%timeit` will automatically adjust and perform fewer repetitions:

```
In[2]: %%timeit
    total = 0
    for i in range(1000):
        for j in range(1000):
            total += i * (-1) ** j

1 loops, best of 3: 407 ms per loop
```

Sometimes repeating an operation is not the best option. For example, if we have a list that we'd like to sort, we might be misled by a repeated operation. Sorting a pre-sorted list is much faster than sorting an unsorted list, so the repetition will skew the result:

```
In[3]: import random
L = [random.random() for i in range(100000)]
%timeit L.sort()

100 loops, best of 3: 1.9 ms per loop
```

For this, the `%time` magic function may be a better choice. It also is a good choice for longer-running commands, when short, system-related delays are unlikely to affect the result. Let's time the sorting of an unsorted and a presorted list:

```
In[4]: import random
L = [random.random() for i in range(100000)]
print("sorting an unsorted list:")
%time L.sort()

sorting an unsorted list:
CPU times: user 40.6 ms, sys: 896 µs, total: 41.5 ms
Wall time: 41.5 ms

In[5]: print("sorting an already sorted list:")
%time L.sort()

sorting an already sorted list:
CPU times: user 8.18 ms, sys: 10 µs, total: 8.19 ms
Wall time: 8.24 ms
```

Notice how much faster the presorted list is to sort, but notice also how much longer the timing takes with `%time` versus `%timeit`, even for the presorted list! This is a result of the fact that `%timeit` does some clever things under the hood to prevent system calls from interfering with the timing. For example, it prevents cleanup of unused Python objects (known as *garbage collection*) that might otherwise affect the timing. For this reason, `%timeit` results are usually noticeably faster than `%time` results.

For `%time` as with `%timeit`, using the double-percent-sign cell-magic syntax allows timing of multiline scripts:

```
In[6]: %%time
    total = 0
    for i in range(1000):
        for j in range(1000):
            total += i * (-1) ** j

CPU times: user 504 ms, sys: 979 µs, total: 505 ms
Wall time: 505 ms
```

For more information on `%time` and `%timeit`, as well as their available options, use the IPython help functionality (i.e., type `%time?` at the IPython prompt).

Profiling Full Scripts: `%prun`

A program is made of many single statements, and sometimes timing these statements in context is more important than timing them on their own. Python contains a built-in code profiler (which you can read about in the Python documentation), but IPython offers a much more convenient way to use this profiler, in the form of the magic function `%prun`.

By way of example, we'll define a simple function that does some calculations:

```
In[7]: def sum_of_lists(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in range(N)]
        total += sum(L)
    return total
```

Now we can call `%prun` with a function call to see the profiled results:

```
In[8]: %prun sum_of_lists(1000000)
```

In the notebook, the output is printed to the pager, and looks something like this:

```
14 function calls in 0.714 seconds

Ordered by: internal time

      ncalls  tottime  percall  cumtime  percall filename:lineno(function)
          5    0.599    0.120    0.599    0.120 <ipython-input-19>:4(<listcomp>)
          5    0.064    0.013    0.064    0.013 {built-in method sum}
          1    0.036    0.036    0.699    0.699 <ipython-input-19>:1(sum_of_lists)
          1    0.014    0.014    0.714    0.714 <string>:1(<module>)
          1    0.000    0.000    0.714    0.714 {built-in method exec}
```

The result is a table that indicates, in order of total time on each function call, where the execution is spending the most time. In this case, the bulk of execution time is in the list comprehension inside `sum_of_lists`. From here, we could start thinking about what changes we might make to improve the performance in the algorithm.

For more information on `%prun`, as well as its available options, use the IPython help functionality (i.e., type `%prun?` at the IPython prompt).

Line-by-Line Profiling with `%lprun`

The function-by-function profiling of `%prun` is useful, but sometimes it's more convenient to have a line-by-line profile report. This is not built into Python or IPython, but there is a `line_profiler` package available for installation that can do this. Start by using Python's packaging tool, `pip`, to install the `line_profiler` package:

```
$ pip install line_profiler
```

Next, you can use IPython to load the `line_profiler` IPython extension, offered as part of this package:

```
In[9]: %load_ext line_profiler
```

Now the `%lprun` command will do a line-by-line profiling of any function—in this case, we need to tell it explicitly which functions we're interested in profiling:

```
In[10]: %lprun -f sum_of_lists sum_of_lists(5000)
```

As before, the notebook sends the result to the pager, but it looks something like this:

```
Timer unit: 1e-06 s

Total time: 0.009382 s
File: <ipython-input-19-fa2be176cc3e>
Function: sum_of_lists at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
=====
 1           1           2     2.0     0.0    def sum_of_lists(N):
 2           1           8     1.3     0.1        total = 0
 3           6          95.9   15.9     95.9    for i in range(5):
 4           5          1800.2  360.0    95.9        L = [j ^ (j >> i) ...
 5           5          371     74.2    4.0        total += sum(L)
 6           1           0     0.0     0.0    return total
```

The information at the top gives us the key to reading the results: the time is reported in microseconds and we can see where the program is spending the most time. At this point, we may be able to use this information to modify aspects of the script and make it perform better for our desired use case.

For more information on `%lprun`, as well as its available options, use the IPython help functionality (i.e., type `%lprun?` at the IPython prompt).

Profiling Memory Use: %memit and %mprun

Another aspect of profiling is the amount of memory an operation uses. This can be evaluated with another IPython extension, the `memory_profiler`. As with the `line_profiler`, we start by pip-installing the extension:

```
$ pip install memory_profiler
```

Then we can use IPython to load the extension:

```
In[12]: %load_ext memory_profiler
```

The memory profiler extension contains two useful magic functions: the `%memit` magic (which offers a memory-measuring equivalent of `%timeit`) and the `%mprun` function (which offers a memory-measuring equivalent of `%lprun`). The `%memit` function can be used rather simply:

```
In[13]: %memit sum_of_lists(1000000)
```

```
peak memory: 100.08 MiB, increment: 61.36 MiB
```

We see that this function uses about 100 MB of memory.

For a line-by-line description of memory use, we can use the `%mprun` magic. Unfortunately, this magic works only for functions defined in separate modules rather than the notebook itself, so we'll start by using the `%%file` magic to create a simple module called `mprun_demo.py`, which contains our `sum_of_lists` function, with one addition that will make our memory profiling results more clear:

```
In[14]: %%file mprun_demo.py
def sum_of_lists(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in range(N)]
        total += sum(L)
        del L # remove reference to L
    return total
```

```
Overwriting mprun_demo.py
```

We can now import the new version of this function and run the memory line profiler:

```
In[15]: from mprun_demo import sum_of_lists
%mprun -f sum_of_lists sum_of_lists(1000000)
```

The result, printed to the pager, gives us a summary of the memory use of the function, and looks something like this:

```
Filename: ./mprun_demo.py

Line #    Mem usage    Increment  Line Contents
=====
4        71.9 MiB      0.0 MiB      L = [j ^ (j >> i) for j in range(N)]
```

```
Filename: ./mprun_demo.py

Line #    Mem usage    Increment  Line Contents
=====
1        39.0 MiB      0.0 MiB      def sum_of_lists(N):
2        39.0 MiB      0.0 MiB      total = 0
3        46.5 MiB      7.5 MiB      for i in range(5):
4        71.9 MiB      25.4 MiB      L = [j ^ (j >> i) for j in range(N)]
5        71.9 MiB      0.0 MiB      total += sum(L)
6        46.5 MiB     -25.4 MiB      del L # remove reference to L
7        39.1 MiB     -7.4 MiB      return total
```

Here the `Increment` column tells us how much each line affects the total memory budget: observe that when we create and delete the list `L`, we are adding about 25 MB of memory usage. This is on top of the background memory usage from the Python interpreter itself.

For more information on `%memit` and `%mprun`, as well as their available options, use the IPython help functionality (i.e., type `%memit?` at the IPython prompt).

More IPython Resources

In this chapter, we've just scratched the surface of using IPython to enable data science tasks. Much more information is available both in print and on the Web, and here we'll list some other resources that you may find helpful.

Web Resources

The IPython website

The IPython website links to documentation, examples, tutorials, and a variety of other resources.

The nbviewer website

This site shows static renderings of any IPython notebook available on the Internet. The front page features some example notebooks that you can browse to see what other folks are using IPython for!

A Gallery of Interesting IPython Notebooks

This ever-growing list of notebooks, powered by nbviewer, shows the depth and breadth of numerical analysis you can do with IPython. It includes everything from short examples and tutorials to full-blown courses and books composed in the notebook format!

Video tutorials

Searching the Internet, you will find many video-recorded tutorials on IPython. I'd especially recommend seeking tutorials from the PyCon, SciPy, and PyData conferences by Fernando Perez and Brian Granger, two of the primary creators and maintainers of IPython and Jupyter.

Books

Python for Data Analysis

Wes McKinney's book includes a chapter that covers using IPython as a data scientist. Although much of the material overlaps what we've discussed here, another perspective is always helpful.

Learning IPython for Interactive Computing and Data Visualization

This short book by Cyrille Rossant offers a good introduction to using IPython for data analysis.

IPython Interactive Computing and Visualization Cookbook

Also by Cyrille Rossant, this book is a longer and more advanced treatment of using IPython for data science. Despite its name, it's not just about IPython—it also goes into some depth on a broad range of data science topics.

Finally, a reminder that you can find help on your own: IPython's `?-based help functionality` (discussed in “[Help and Documentation in IPython](#)” on page 3) can be very useful if you use it well and use it often. As you go through the examples here and elsewhere, you can use it to familiarize yourself with all the tools that IPython has to offer.

CHAPTER 2

Introduction to NumPy

This chapter, along with [Chapter 3](#), outlines techniques for effectively loading, storing, and manipulating in-memory data in Python. The topic is very broad: datasets can come from a wide range of sources and a wide range of formats, including collections of documents, collections of images, collections of sound clips, collections of numerical measurements, or nearly anything else. Despite this apparent heterogeneity, it will help us to think of all data fundamentally as arrays of numbers.

For example, images—particularly digital images—can be thought of as simply two-dimensional arrays of numbers representing pixel brightness across the area. Sound clips can be thought of as one-dimensional arrays of intensity versus time. Text can be converted in various ways into numerical representations, perhaps binary digits representing the frequency of certain words or pairs of words. No matter what the data are, the first step in making them analyzable will be to transform them into arrays of numbers. (We will discuss some specific examples of this process later in “[Feature Engineering](#)” on page 375.)

For this reason, efficient storage and manipulation of numerical arrays is absolutely fundamental to the process of doing data science. We’ll now take a look at the specialized tools that Python has for handling such numerical arrays: the NumPy package and the Pandas package (discussed in [Chapter 3](#).)

This chapter will cover NumPy in detail. NumPy (short for *Numerical Python*) provides an efficient interface to store and operate on dense data buffers. In some ways, NumPy arrays are like Python’s built-in `list` type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size. NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you.

If you followed the advice outlined in the preface and installed the Anaconda stack, you already have NumPy installed and ready to go. If you’re more the do-it-yourself type, you can go to [the NumPy website](#) and follow the installation instructions found there. Once you do, you can import NumPy and double-check the version:

```
In[1]: import numpy  
       numpy.__version__  
  
Out[1]: '1.11.1'
```

For the pieces of the package discussed here, I’d recommend NumPy version 1.8 or later. By convention, you’ll find that most people in the SciPy/PyData world will import NumPy using `np` as an alias:

```
In[2]: import numpy as np
```

Throughout this chapter, and indeed the rest of the book, you’ll find that this is the way we will import and use NumPy.

Reminder About Built-In Documentation

As you read through this chapter, don’t forget that IPython gives you the ability to quickly explore the contents of a package (by using the tab-completion feature) as well as the documentation of various functions (using the `?` character). Refer back to “[Help and Documentation in IPython](#)” on page 3 if you need a refresher on this.

For example, to display all the contents of the `numpy` namespace, you can type this:

```
In [3]: np.<TAB>
```

And to display NumPy’s built-in documentation, you can use this:

```
In [4]: np?
```

More detailed documentation, along with tutorials and other resources, can be found at <http://www.numpy.org>.

Understanding Data Types in Python

Effective data-driven science and computation requires understanding how data is stored and manipulated. This section outlines and contrasts how arrays of data are handled in the Python language itself, and how NumPy improves on this. Understanding this difference is fundamental to understanding much of the material throughout the rest of the book.

Users of Python are often drawn in by its ease of use, one piece of which is dynamic typing. While a statically typed language like C or Java requires each variable to be

explicitly declared, a dynamically typed language like Python skips this specification. For example, in C you might specify a particular operation as follows:

```
/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

While in Python the equivalent operation could be written this way:

```
# Python code
result = 0
for i in range(100):
    result += i
```

Notice the main difference: in C, the data types of each variable are explicitly declared, while in Python the types are dynamically inferred. This means, for example, that we can assign any kind of data to any variable:

```
# Python code
x = 4
x = "four"
```

Here we've switched the contents of `x` from an integer to a string. The same thing in C would lead (depending on compiler settings) to a compilation error or other unintended consequences:

```
/* C code */
int x = 4;
x = "four"; // FAILS
```

This sort of flexibility is one piece that makes Python and other dynamically typed languages convenient and easy to use. Understanding *how* this works is an important piece of learning to analyze data efficiently and effectively with Python. But what this type flexibility also points to is the fact that Python variables are more than just their value; they also contain extra information about the type of the value. We'll explore this more in the sections that follow.

A Python Integer Is More Than Just an Integer

The standard Python implementation is written in C. This means that every Python object is simply a cleverly disguised C structure, which contains not only its value, but other information as well. For example, when we define an integer in Python, such as `x = 10000`, `x` is not just a “raw” integer. It’s actually a pointer to a compound C structure, which contains several values. Looking through the Python 3.4 source code, we find that the integer (long) type definition effectively looks like this (once the C macros are expanded):

```

struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};

```

A single integer in Python 3.4 actually contains four pieces:

- `ob_refcnt`, a reference count that helps Python silently handle memory allocation and deallocation
- `ob_type`, which encodes the type of the variable
- `ob_size`, which specifies the size of the following data members
- `ob_digit`, which contains the actual integer value that we expect the Python variable to represent

This means that there is some overhead in storing an integer in Python as compared to an integer in a compiled language like C, as illustrated in [Figure 2-1](#).

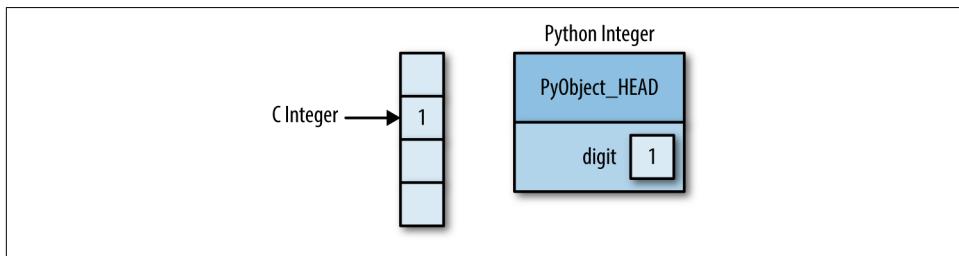


Figure 2-1. The difference between C and Python integers

Here `PyObject_HEAD` is the part of the structure containing the reference count, type code, and other pieces mentioned before.

Notice the difference here: a C integer is essentially a label for a position in memory whose bytes encode an integer value. A Python integer is a pointer to a position in memory containing all the Python object information, including the bytes that contain the integer value. This extra information in the Python integer structure is what allows Python to be coded so freely and dynamically. All this additional information in Python types comes at a cost, however, which becomes especially apparent in structures that combine many of these objects.

A Python List Is More Than Just a List

Let's consider now what happens when we use a Python data structure that holds many Python objects. The standard mutable multielement container in Python is the list. We can create a list of integers as follows:

```
In[1]: L = list(range(10))
L

Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In[2]: type(L[0])

Out[2]: int
```

Or, similarly, a list of strings:

```
In[3]: L2 = [str(c) for c in L]
L2

Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

In[4]: type(L2[0])

Out[4]: str
```

Because of Python's dynamic typing, we can even create heterogeneous lists:

```
In[5]: L3 = [True, "2", 3.0, 4]
[type(item) for item in L3]

Out[5]: [bool, str, float, int]
```

But this flexibility comes at a cost: to allow these flexible types, each item in the list must contain its own type info, reference count, and other information—that is, each item is a complete Python object. In the special case that all variables are of the same type, much of this information is redundant: it can be much more efficient to store data in a fixed-type array. The difference between a dynamic-type list and a fixed-type (NumPy-style) array is illustrated in [Figure 2-2](#).

At the implementation level, the array essentially contains a single pointer to one contiguous block of data. The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object like the Python integer we saw earlier. Again, the advantage of the list is flexibility: because each list element is a full structure containing both data and type information, the list can be filled with data of any desired type. Fixed-type NumPy-style arrays lack this flexibility, but are much more efficient for storing and manipulating data.



Figure 2-2. The difference between C and Python lists

Fixed-Type Arrays in Python

Python offers several different options for storing data in efficient, fixed-type data buffers. The built-in `array` module (available since Python 3.3) can be used to create dense arrays of a uniform type:

```
In[6]: import array
L = list(range(10))
A = array.array('i', L)
A
```

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

Here '`i`' is a type code indicating the contents are integers.

Much more useful, however, is the `ndarray` object of the NumPy package. While Python's `array` object provides efficient storage of array-based data, NumPy adds to this efficient *operations* on that data. We will explore these operations in later sections; here we'll demonstrate several ways of creating a NumPy array.

We'll start with the standard NumPy import, under the alias `np`:

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

Creating Arrays from Python Lists

First, we can use `np.array` to create arrays from Python lists:

```
In[8]: # integer array:  
np.array([1, 4, 2, 5, 3])  
  
Out[8]: array([1, 4, 2, 5, 3])
```

Remember that unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are upcast to floating point):

```
In[9]: np.array([3.14, 4, 2, 3])  
  
Out[9]: array([ 3.14,  4. ,  2. ,  3. ])
```

If we want to explicitly set the data type of the resulting array, we can use the `dtype` keyword:

```
In[10]: np.array([1, 2, 3, 4], dtype='float32')  
  
Out[10]: array([ 1.,  2.,  3.,  4.], dtype=float32)
```

Finally, unlike Python lists, NumPy arrays can explicitly be multidimensional; here's one way of initializing a multidimensional array using a list of lists:

```
In[11]: # nested lists result in multidimensional arrays  
np.array([range(i, i + 3) for i in [2, 4, 6]])  
  
Out[11]: array([[2, 3, 4],  
                [4, 5, 6],  
                [6, 7, 8]])
```

The inner lists are treated as rows of the resulting two-dimensional array.

Creating Arrays from Scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy. Here are several examples:

```
In[12]: # Create a length-10 integer array filled with zeros  
np.zeros(10, dtype=int)  
  
Out[12]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])  
  
In[13]: # Create a 3x5 floating-point array filled with 1s  
np.ones((3, 5), dtype=float)  
  
Out[13]: array([[ 1.,  1.,  1.,  1.,  1.],  
                [ 1.,  1.,  1.,  1.,  1.],  
                [ 1.,  1.,  1.,  1.,  1.]])  
  
In[14]: # Create a 3x5 array filled with 3.14  
np.full((3, 5), 3.14)
```

```
Out[14]: array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
   [ 3.14,  3.14,  3.14,  3.14,  3.14],
   [ 3.14,  3.14,  3.14,  3.14,  3.14]])
```

```
In[15]: # Create an array filled with a linear sequence
      # Starting at 0, ending at 20, stepping by 2
      # (this is similar to the built-in range() function)
      np.arange(0, 20, 2)
```

```
Out[15]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
In[16]: # Create an array of five values evenly spaced between 0 and 1
      np.linspace(0, 1, 5)
```

```
Out[16]: array([ 0. ,  0.25,  0.5 ,  0.75,  1. ])
```

```
In[17]: # Create a 3x3 array of uniformly distributed
      # random values between 0 and 1
      np.random.random((3, 3))
```

```
Out[17]: array([[ 0.99844933,  0.52183819,  0.22421193],
   [ 0.08007488,  0.45429293,  0.20941444],
   [ 0.14360941,  0.96910973,  0.946117 ]])
```

```
In[18]: # Create a 3x3 array of normally distributed random values
      # with mean 0 and standard deviation 1
      np.random.normal(0, 1, (3, 3))
```

```
Out[18]: array([[ 1.51772646,  0.39614948, -0.10634696],
   [ 0.25671348,  0.00732722,  0.37783601],
   [ 0.68446945,  0.15926039, -0.70744073]])
```

```
In[19]: # Create a 3x3 array of random integers in the interval [0, 10)
      np.random.randint(0, 10, (3, 3))
```

```
Out[19]: array([[2, 3, 4],
   [5, 7, 8],
   [0, 5, 0]])
```

```
In[20]: # Create a 3x3 identity matrix
      np.eye(3)
```

```
Out[20]: array([[ 1.,  0.,  0.],
   [ 0.,  1.,  0.],
   [ 0.,  0.,  1.]])
```

```
In[21]: # Create an uninitialized array of three integers
      # The values will be whatever happens to already exist at that
      # memory location
      np.empty(3)
```

```
Out[21]: array([ 1.,  1.,  1.])
```

NumPy Standard Data Types

NumPy arrays contain values of a single type, so it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.

The standard NumPy data types are listed in [Table 2-1](#). Note that when constructing an array, you can specify them using a string:

```
np.zeros(10, dtype='int16')
```

Or using the associated NumPy object:

```
np.zeros(10, dtype=np.int16)
```

Table 2-1. Standard NumPy data types

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64
float16	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128
complex64	Complex number, represented by two 32-bit floats
complex128	Complex number, represented by two 64-bit floats

More advanced type specification is possible, such as specifying big or little endian numbers; for more information, refer to the [NumPy documentation](#). NumPy also supports compound data types, which will be covered in “[Structured Data: NumPy’s Structured Arrays](#)” on page 92.

The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas ([Chapter 3](#)) are built around the NumPy array. This section will present several examples using NumPy array manipulation to access data and subarrays, and to split, reshape, and join the arrays. While the types of operations shown here may seem a bit dry and pedantic, they comprise the building blocks of many other examples used throughout the book. Get to know them well!

We'll cover a few categories of basic array manipulations here:

Attributes of arrays

Determining the size, shape, memory consumption, and data types of arrays

Indexing of arrays

Getting and setting the value of individual array elements

Slicing of arrays

Getting and setting smaller subarrays within a larger array

Reshaping of arrays

Changing the shape of a given array

Joining and splitting of arrays

Combining multiple arrays into one, and splitting one array into many

NumPy Array Attributes

First let's discuss some useful array attributes. We'll start by defining three random arrays: a one-dimensional, two-dimensional, and three-dimensional array. We'll use NumPy's random number generator, which we will *seed* with a set value in order to ensure that the same random arrays are generated each time this code is run:

```
In[1]: import numpy as np
        np.random.seed(0) # seed for reproducibility

        x1 = np.random.randint(10, size=6) # One-dimensional array
        x2 = np.random.randint(10, size=(3, 4)) # Two-dimensional array
        x3 = np.random.randint(10, size=(3, 4, 5)) # Three-dimensional array
```

Each array has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array):

```
In[2]: print("x3 ndim: ", x3.ndim)
        print("x3 shape:", x3.shape)
        print("x3 size: ", x3.size)

x3 ndim: 3
x3 shape: (3, 4, 5)
x3 size: 60
```

Another useful attribute is the `dtype`, the data type of the array (which we discussed previously in “[Understanding Data Types in Python](#)” on page 34):

```
In[3]: print("dtype:", x3.dtype)
dtype: int64
```

Other attributes include `itemsize`, which lists the size (in bytes) of each array element, and `nbytes`, which lists the total size (in bytes) of the array:

```
In[4]: print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")

itemsize: 8 bytes
nbytes: 480 bytes
```

In general, we expect that `nbytes` is equal to `itemsize` times `size`.

Array Indexing: Accessing Single Elements

If you are familiar with Python’s standard list indexing, indexing in NumPy will feel quite familiar. In a one-dimensional array, you can access the i^{th} value (counting from zero) by specifying the desired index in square brackets, just as with Python lists:

```
In[5]: x1
Out[5]: array([5, 0, 3, 3, 7, 9])
In[6]: x1[0]
Out[6]: 5
In[7]: x1[4]
Out[7]: 7
```

To index from the end of the array, you can use negative indices:

```
In[8]: x1[-1]
Out[8]: 9
In[9]: x1[-2]
Out[9]: 7
```

In a multidimensional array, you access items using a comma-separated tuple of indices:

```
In[10]: x2
Out[10]: array([[3, 5, 2, 4],
               [7, 6, 8, 8],
               [1, 6, 7, 7]])
In[11]: x2[0, 0]
Out[11]: 3
```

```
In[12]: x2[2, 0]
```

```
Out[12]: 1
```

```
In[13]: x2[2, -1]
```

```
Out[13]: 7
```

You can also modify values using any of the above index notation:

```
In[14]: x2[0, 0] = 12
x2
```

```
Out[14]: array([[12,  5,  2,  4],
                [ 7,  6,  8,  8],
                [ 1,  6,  7,  7]])
```

Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated. Don't be caught unaware by this behavior!

```
In[15]: x1[0] = 3.14159 # this will be truncated!
x1
```

```
Out[15]: array([3, 0, 3, 3, 7, 9])
```

Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array *x*, use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values *start=0*, *stop=size of dimension*, *step=1*. We'll take a look at accessing subarrays in one dimension and in multiple dimensions.

One-dimensional subarrays

```
In[16]: x = np.arange(10)
x
```

```
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In[17]: x[:5] # first five elements
```

```
Out[17]: array([0, 1, 2, 3, 4])
```

```
In[18]: x[5:] # elements after index 5
```

```
Out[18]: array([5, 6, 7, 8, 9])
```

```
In[19]: x[4:7] # middle subarray
```

```
Out[19]: array([4, 5, 6])
```

```
In[20]: x[::2] # every other element
Out[20]: array([0, 2, 4, 6, 8])
In[21]: x[1::2] # every other element, starting at index 1
Out[21]: array([1, 3, 5, 7, 9])
```

A potentially confusing case is when the `step` value is negative. In this case, the defaults for `start` and `stop` are swapped. This becomes a convenient way to reverse an array:

```
In[22]: x[::-1] # all elements, reversed
Out[22]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
In[23]: x[5::-2] # reversed every other from index 5
Out[23]: array([5, 3, 1])
```

Multidimensional subarrays

Multidimensional slices work in the same way, with multiple slices separated by commas. For example:

```
In[24]: x2
Out[24]: array([[12, 5, 2, 4],
               [7, 6, 8, 8],
               [1, 6, 7, 7]])
In[25]: x2[:2, :3] # two rows, three columns
Out[25]: array([[12, 5, 2],
               [7, 6, 8]])
In[26]: x2[:, ::2] # all rows, every other column
Out[26]: array([[12, 2],
               [7, 8],
               [1, 7]])
```

Finally, subarray dimensions can even be reversed together:

```
In[27]: x2[::-1, ::-1]
Out[27]: array([[ 7,  7,  6,  1],
               [ 8,  8,  6,  7],
               [ 4,  2,  5, 12]])
```

Accessing array rows and columns. One commonly needed routine is accessing single rows or columns of an array. You can do this by combining indexing and slicing, using an empty slice marked by a single colon (`:`):

```
In[28]: print(x2[:, 0]) # first column of x2
[12 7 1]
```

```
In[29]: print(x2[0, :]) # first row of x2  
[12 5 2 4]
```

In the case of row access, the empty slice can be omitted for a more compact syntax:

```
In[30]: print(x2[0]) # equivalent to x2[0, :]  
[12 5 2 4]
```

Subarrays as no-copy views

One important—and extremely useful—thing to know about array slices is that they return *views* rather than *copies* of the array data. This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies. Consider our two-dimensional array from before:

```
In[31]: print(x2)  
[[12 5 2 4]  
 [ 7 6 8 8]  
 [ 1 6 7 7]]
```

Let's extract a 2×2 subarray from this:

```
In[32]: x2_sub = x2[:2, :2]  
print(x2_sub)  
  
[[12 5]  
 [ 7 6]]
```

Now if we modify this subarray, we'll see that the original array is changed! Observe:

```
In[33]: x2_sub[0, 0] = 99  
print(x2_sub)  
  
[[99 5]  
 [ 7 6]]  
  
In[34]: print(x2)  
[[99 5 2 4]  
 [ 7 6 8 8]  
 [ 1 6 7 7]]
```

This default behavior is actually quite useful: it means that when we work with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

Creating copies of arrays

Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method:

```
In[35]: x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)

[[99  5]
 [ 7  6]]
```

If we now modify this subarray, the original array is not touched:

```
In[36]: x2_sub_copy[0, 0] = 42
print(x2_sub_copy)

[[42  5]
 [ 7  6]]

In[37]: print(x2)

[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape()` method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

```
In[38]: grid = np.arange(1, 10).reshape((3, 3))
print(grid)

[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Note that for this to work, the size of the initial array must match the size of the reshaped array. Where possible, the `reshape` method will use a no-copy view of the initial array, but with noncontiguous memory buffers this is not always the case.

Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix. You can do this with the `reshape` method, or more easily by making use of the `newaxis` keyword within a slice operation:

```
In[39]: x = np.array([1, 2, 3])

# row vector via reshape
x.reshape((1, 3))

Out[39]: array([[1, 2, 3]])

In[40]: # row vector via newaxis
x[np.newaxis, :]

Out[40]: array([[1, 2, 3]])
```

```
In[41]: # column vector via reshape  
x.reshape((3, 1))  
  
Out[41]: array([[1],  
                 [2],  
                 [3]])  
  
In[42]: # column vector via newaxis  
x[:, np.newaxis]  
  
Out[42]: array([[1],  
                 [2],  
                 [3]])
```

We will see this type of transformation often throughout the remainder of the book.

Array Concatenation and Splitting

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished through the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

```
In[43]: x = np.array([1, 2, 3])  
y = np.array([3, 2, 1])  
np.concatenate([x, y])  
  
Out[43]: array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

```
In[44]: z = [99, 99, 99]  
print(np.concatenate([x, y, z]))  
  
[ 1  2  3  3  2  1 99 99 99]
```

`np.concatenate` can also be used for two-dimensional arrays:

```
In[45]: grid = np.array([[1, 2, 3],  
                      [4, 5, 6]])  
  
In[46]: # concatenate along the first axis  
np.concatenate([grid, grid])  
  
Out[46]: array([[1, 2, 3],  
                 [4, 5, 6],  
                 [1, 2, 3],  
                 [4, 5, 6]])  
  
In[47]: # concatenate along the second axis (zero-indexed)  
np.concatenate([grid, grid], axis=1)
```

```
Out[47]: array([[1, 2, 3, 1, 2, 3],  
                 [4, 5, 6, 4, 5, 6]])
```

For working with arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions:

```
In[48]: x = np.array([1, 2, 3])  
grid = np.array([[9, 8, 7],  
                [6, 5, 4]])
```

```
# vertically stack the arrays  
np.vstack([x, grid])
```

```
Out[48]: array([[1, 2, 3],  
                 [9, 8, 7],  
                 [6, 5, 4]])
```

```
In[49]: # horizontally stack the arrays  
y = np.array([[99],  
             [99]])  
np.hstack([grid, y])
```

```
Out[49]: array([[ 9,  8,  7, 99],  
                 [ 6,  5,  4, 99]])
```

Similarly, `np.dstack` will stack arrays along the third axis.

Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points:

```
In[50]: x = [1, 2, 3, 99, 99, 3, 2, 1]  
x1, x2, x3 = np.split(x, [3, 5])  
print(x1, x2, x3)
```



```
[1 2 3] [99 99] [3 2 1]
```

Notice that N split points lead to $N + 1$ subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

```
In[51]: grid = np.arange(16).reshape((4, 4))  
grid
```

```
Out[51]: array([[ 0,  1,  2,  3],  
                 [ 4,  5,  6,  7],  
                 [ 8,  9, 10, 11],  
                 [12, 13, 14, 15]])
```

```
In[52]: upper, lower = np.vsplit(grid, [2])  
print(upper)  
print(lower)
```

```
[[0 1 2 3]  
 [4 5 6 7]]
```

```
[[ 8  9 10 11]
 [12 13 14 15]]
In[53]: left, right = np.hsplit(grid, [2])
          print(left)
          print(right)

[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Similarly, `np.dsplit` will split arrays along the third axis.

Computation on NumPy Arrays: Universal Functions

Up until now, we have been discussing some of the basic nuts and bolts of NumPy; in the next few sections, we will dive into the reasons that NumPy is so important in the Python data science world. Namely, it provides an easy and flexible interface to optimized computation with arrays of data.

Computation on NumPy arrays can be very fast, or it can be very slow. The key to making it fast is to use *vectorized* operations, generally implemented through NumPy's *universal functions* (ufuncs). This section motivates the need for NumPy's ufuncs, which can be used to make repeated calculations on array elements much more efficient. It then introduces many of the most common and useful arithmetic ufuncs available in the NumPy package.

The Slowness of Loops

Python's default implementation (known as CPython) does some operations very slowly. This is in part due to the dynamic, interpreted nature of the language: the fact that types are flexible, so that sequences of operations cannot be compiled down to efficient machine code as in languages like C and Fortran. Recently there have been various attempts to address this weakness: well-known examples are the [PyPy project](#), a just-in-time compiled implementation of Python; the [Cython project](#), which converts Python code to compilable C code; and the [Numba project](#), which converts snippets of Python code to fast LLVM bytecode. Each of these has its strengths and weaknesses, but it is safe to say that none of the three approaches has yet surpassed the reach and popularity of the standard CPython engine.

The relative sluggishness of Python generally manifests itself in situations where many small operations are being repeated—for instance, looping over arrays to oper-

ate on each element. For example, imagine we have an array of values and we'd like to compute the reciprocal of each. A straightforward approach might look like this:

```
In[1]: import numpy as np
np.random.seed(0)

def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1.0 / values[i]
    return output

values = np.random.randint(1, 10, size=5)
compute_reciprocals(values)

Out[1]: array([ 0.16666667,  1.          ,  0.25        ,  0.25        ,  0.125        ])
```

This implementation probably feels fairly natural to someone from, say, a C or Java background. But if we measure the execution time of this code for a large input, we see that this operation is very slow, perhaps surprisingly so! We'll benchmark this with IPython's `%timeit` magic (discussed in “[Profiling and Timing Code](#)” on page 25):

```
In[2]: big_array = np.random.randint(1, 100, size=1000000)
%timeit compute_reciprocals(big_array)

1 loop, best of 3: 2.91 s per loop
```

It takes several seconds to compute these million operations and to store the result! When even cell phones have processing speeds measured in Giga-FLOPS (i.e., billions of numerical operations per second), this seems almost absurdly slow. It turns out that the bottleneck here is not the operations themselves, but the type-checking and function dispatches that CPython must do at each cycle of the loop. Each time the reciprocal is computed, Python first examines the object's type and does a dynamic lookup of the correct function to use for that type. If we were working in compiled code instead, this type specification would be known before the code executes and the result could be computed much more efficiently.

Introducing UFuncs

For many types of operations, NumPy provides a convenient interface into just this kind of statically typed, compiled routine. This is known as a *vectorized* operation. You can accomplish this by simply performing an operation on the array, which will then be applied to each element. This vectorized approach is designed to push the loop into the compiled layer that underlies NumPy, leading to much faster execution.

Compare the results of the following two:

```
In[3]: print(compute_reciprocals(values))
print(1.0 / values)
```

```
[ 0.16666667  1.          0.25          0.25          0.125        ]
[ 0.16666667  1.          0.25          0.25          0.125        ]
```

Looking at the execution time for our big array, we see that it completes orders of magnitude faster than the Python loop:

```
In[4]: %timeit (1.0 / big_array)
100 loops, best of 3: 4.6 ms per loop
```

Vectorized operations in NumPy are implemented via *ufuncs*, whose main purpose is to quickly execute repeated operations on values in NumPy arrays. Ufuncs are extremely flexible—before we saw an operation between a scalar and an array, but we can also operate between two arrays:

```
In[5]: np.arange(5) / np.arange(1, 6)
Out[5]: array([ 0.          ,  0.5         ,  0.66666667,  0.75         ,  0.8         ])
```

And ufunc operations are not limited to one-dimensional arrays—they can act on multidimensional arrays as well:

```
In[6]: x = np.arange(9).reshape((3, 3))
2 ** x
Out[6]: array([[ 1,   2,   4],
               [ 8,  16,  32],
               [ 64, 128, 256]])
```

Computations using vectorization through ufuncs are nearly always more efficient than their counterpart implemented through Python loops, especially as the arrays grow in size. Any time you see such a loop in a Python script, you should consider whether it can be replaced with a vectorized expression.

Exploring NumPy's UFuncs

Ufuncs exist in two flavors: *unary ufuncs*, which operate on a single input, and *binary ufuncs*, which operate on two inputs. We'll see examples of both these types of functions here.

Array arithmetic

NumPy's ufuncs feel very natural to use because they make use of Python's native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used:

```
In[7]: x = np.arange(4)
print("x      =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
```

```

print("x / 2 =", x / 2)
print("x // 2 =", x // 2) # floor division

x      = [0 1 2 3]
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [ 0.   0.5  1.   1.5]
x // 2 = [0 0 1 1]

```

There is also a unary ufunc for negation, a `**` operator for exponentiation, and a `%` operator for modulus:

```

In[8]: print("-x      = ", -x)
        print("x ** 2 = ", x ** 2)
        print("x % 2  = ", x % 2)

-x      = [ 0 -1 -2 -3]
x ** 2 = [0 1 4 9]
x % 2  = [0 1 0 1]

```

In addition, these can be strung together however you wish, and the standard order of operations is respected:

```

In[9]: -(0.5*x + 1) ** 2
Out[9]: array([-1. , -2.25, -4. , -6.25])

```

All of these arithmetic operations are simply convenient wrappers around specific functions built into NumPy; for example, the `+` operator is a wrapper for the `add` function:

```

In[10]: np.add(x, 2)
Out[10]: array([2, 3, 4, 5])

```

Table 2-2 lists the arithmetic operators implemented in NumPy.

Table 2-2. Arithmetic operators implemented in NumPy

Operator	Equivalent ufunc	Description
<code>+</code>	<code>np.add</code>	Addition (e.g., $1 + 1 = 2$)
<code>-</code>	<code>np.subtract</code>	Subtraction (e.g., $3 - 2 = 1$)
<code>-</code>	<code>np.negative</code>	Unary negation (e.g., -2)
<code>*</code>	<code>np.multiply</code>	Multiplication (e.g., $2 * 3 = 6$)
<code>/</code>	<code>np.divide</code>	Division (e.g., $3 / 2 = 1.5$)
<code>//</code>	<code>np.floor_divide</code>	Floor division (e.g., $3 // 2 = 1$)
<code>**</code>	<code>np.power</code>	Exponentiation (e.g., $2 ** 3 = 8$)
<code>%</code>	<code>np.mod</code>	Modulus/remainder (e.g., $9 \% 4 = 1$)

Additionally there are Boolean/bitwise operators; we will explore these in “[Comparisons, Masks, and Boolean Logic](#)” on page 70.

Absolute value

Just as NumPy understands Python’s built-in arithmetic operators, it also understands Python’s built-in absolute value function:

```
In[11]: x = np.array([-2, -1, 0, 1, 2])
abs(x)

Out[11]: array([2, 1, 0, 1, 2])
```

The corresponding NumPy ufunc is `np.absolute`, which is also available under the alias `np.abs`:

```
In[12]: np.absolute(x)

Out[12]: array([2, 1, 0, 1, 2])

In[13]: np.abs(x)

Out[13]: array([2, 1, 0, 1, 2])
```

This ufunc can also handle complex data, in which the absolute value returns the magnitude:

```
In[14]: x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
np.abs(x)

Out[14]: array([ 5.,  5.,  2.,  1.])
```

Trigonometric functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. We’ll start by defining an array of angles:

```
In[15]: theta = np.linspace(0, np.pi, 3)
```

Now we can compute some trigonometric functions on these values:

```
In[16]: print("theta      = ", theta)
        print("sin(theta) = ", np.sin(theta))
        print("cos(theta) = ", np.cos(theta))
        print("tan(theta) = ", np.tan(theta))

theta      = [ 0.           1.57079633  3.14159265]
sin(theta) = [ 0.00000000e+00  1.00000000e+00  1.22464680e-16]
cos(theta) = [ 1.00000000e+00  6.12323400e-17 -1.00000000e+00]
tan(theta) = [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

The values are computed to within machine precision, which is why values that should be zero do not always hit exactly zero. Inverse trigonometric functions are also available:

```
In[17]: x = [-1, 0, 1]
    print("x      =", x)
    print("arcsin(x) =", np.arcsin(x))
    print("arccos(x) =", np.arccos(x))
    print("arctan(x) =", np.arctan(x))

x      = [-1, 0, 1]
arcsin(x) = [-1.57079633  0.           1.57079633]
arccos(x) = [ 3.14159265  1.57079633  0.           ]
arctan(x) = [-0.78539816  0.           0.78539816]
```

Exponents and logarithms

Another common type of operation available in a NumPy ufunc are the exponentials:

```
In[18]: x = [1, 2, 3]
    print("x      =", x)
    print("e^x    =", np.exp(x))
    print("2^x    =", np.exp2(x))
    print("3^x    =", np.power(3, x))

x      = [1, 2, 3]
e^x    = [ 2.71828183  7.3890561   20.08553692]
2^x    = [ 2.        4.        8.]
3^x    = [ 3         9         27]
```

The inverse of the exponentials, the logarithms, are also available. The basic `np.log` gives the natural logarithm; if you prefer to compute the base-2 logarithm or the base-10 logarithm, these are available as well:

```
In[19]: x = [1, 2, 4, 10]
    print("x      =", x)
    print("ln(x)  =", np.log(x))
    print("log2(x) =", np.log2(x))
    print("log10(x) =", np.log10(x))

x      = [1, 2, 4, 10]
ln(x)  = [ 0.          0.69314718  1.38629436  2.30258509]
log2(x) = [ 0.          1.          2.          3.32192809]
log10(x) = [ 0.          0.30103    0.60205999  1.          ]
```

There are also some specialized versions that are useful for maintaining precision with very small input:

```
In[20]: x = [0, 0.001, 0.01, 0.1]
    print("exp(x) - 1 =", np.expm1(x))
    print("log(1 + x) =", np.log1p(x))

exp(x) - 1 = [ 0.          0.0010005  0.01005017  0.10517092]
log(1 + x) = [ 0.          0.0009995  0.00995033  0.09531018]
```

When `x` is very small, these functions give more precise values than if the raw `np.log` or `np.exp` were used.

Specialized ufuncs

NumPy has many more ufuncs available, including hyperbolic trig functions, bitwise arithmetic, comparison operators, conversions from radians to degrees, rounding and remainders, and much more. A look through the NumPy documentation reveals a lot of interesting functionality.

Another excellent source for more specialized and obscure ufuncs is the submodule `scipy.special`. If you want to compute some obscure mathematical function on your data, chances are it is implemented in `scipy.special`. There are far too many functions to list them all, but the following snippet shows a couple that might come up in a statistics context:

```
In[21]: from scipy import special

In[22]: # Gamma functions (generalized factorials) and related functions
          x = [1, 5, 10]
          print("gamma(x)      =", special.gamma(x))
          print("ln|gamma(x)| =", special.gammaln(x))
          print("beta(x, 2)   =", special.beta(x, 2))

gamma(x)      = [ 1.00000000e+00  2.40000000e+01  3.62880000e+05]
ln|gamma(x)| = [ 0.           3.17805383  12.80182748]
beta(x, 2)   = [ 0.5          0.03333333  0.00909091]

In[23]: # Error function (integral of Gaussian)
          # its complement, and its inverse
          x = np.array([0, 0.3, 0.7, 1.0])
          print("erf(x)      =", special.erf(x))
          print("erfc(x)     =", special.erfc(x))
          print("erfinv(x)   =", special.erfinv(x))

erf(x)      = [ 0.          0.32862676  0.67780119  0.84270079]
erfc(x)     = [ 1.          0.67137324  0.32219881  0.15729921]
erfinv(x)   = [ 0.          0.27246271  0.73286908         inf]
```

There are many, many more ufuncs available in both NumPy and `scipy.special`. Because the documentation of these packages is available online, a web search along the lines of “gamma function python” will generally find the relevant information.

Advanced Ufunc Features

Many NumPy users make use of ufuncs without ever learning their full set of features. We'll outline a few specialized features of ufuncs here.

Specifying output

For large calculations, it is sometimes useful to be able to specify the array where the result of the calculation will be stored. Rather than creating a temporary array, you can use this to write computation results directly to the memory location where you'd

like them to be. For all ufuncs, you can do this using the `out` argument of the function:

```
In[24]: x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)

[ 0.  10.  20.  30.  40.]
```

This can even be used with array views. For example, we can write the results of a computation to every other element of a specified array:

```
In[25]: y = np.zeros(10)
np.power(2, x, out=y[::2])
print(y)

[ 1.  0.  2.  0.  4.  0.  8.  0.  16.  0.]
```

If we had instead written $y[::2] = 2^{**} x$, this would have resulted in the creation of a temporary array to hold the results of $2^{**} x$, followed by a second operation copying those values into the `y` array. This doesn't make much of a difference for such a small computation, but for very large arrays the memory savings from careful use of the `out` argument can be significant.

Aggregates

For binary ufuncs, there are some interesting aggregates that can be computed directly from the object. For example, if we'd like to *reduce* an array with a particular operation, we can use the `reduce` method of any ufunc. A `reduce` repeatedly applies a given operation to the elements of an array until only a single result remains.

For example, calling `reduce` on the `add` ufunc returns the sum of all elements in the array:

```
In[26]: x = np.arange(1, 6)
np.add.reduce(x)
```

```
Out[26]: 15
```

Similarly, calling `reduce` on the `multiply` ufunc results in the product of all array elements:

```
In[27]: np.multiply.reduce(x)
```

```
Out[27]: 120
```

If we'd like to store all the intermediate results of the computation, we can instead use `accumulate`:

```
In[28]: np.add.accumulate(x)
```

```
Out[28]: array([ 1,  3,  6, 10, 15])
```

```
In[29]: np.multiply.accumulate(x)
Out[29]: array([ 1,  2,  6, 24, 120])
```

Note that for these particular cases, there are dedicated NumPy functions to compute the results (`np.sum`, `np.prod`, `np.cumsum`, `np.cumprod`), which we'll explore in “[Aggregations: Min, Max, and Everything in Between](#)” on page 58.

Outer products

Finally, any ufunc can compute the output of all pairs of two different inputs using the `outer` method. This allows you, in one line, to do things like create a multiplication table:

```
In[30]: x = np.arange(1, 6)
        np.multiply.outer(x, x)

Out[30]: array([[ 1,  2,  3,  4,  5],
                [ 2,  4,  6,  8, 10],
                [ 3,  6,  9, 12, 15],
                [ 4,  8, 12, 16, 20],
                [ 5, 10, 15, 20, 25]])
```

The `ufunc.at` and `ufunc.reduceat` methods, which we'll explore in “[Fancy Indexing](#)” on page 78, are very helpful as well.

Another extremely useful feature of ufuncs is the ability to operate between arrays of different sizes and shapes, a set of operations known as *broadcasting*. This subject is important enough that we will devote a whole section to it (see “[Computation on Arrays: Broadcasting](#)” on page 63).

Ufuncs: Learning More

More information on universal functions (including the full list of available functions) can be found on the [NumPy](#) and [SciPy](#) documentation websites.

Recall that you can also access information directly from within IPython by importing the packages and using IPython's tab-completion and help (?) functionality, as described in “[Help and Documentation in IPython](#)” on page 3.

Aggregations: Min, Max, and Everything in Between

Often when you are faced with a large amount of data, a first step is to compute summary statistics for the data in question. Perhaps the most common summary statistics are the mean and standard deviation, which allow you to summarize the “typical” values in a dataset, but other aggregates are useful as well (the sum, product, median, minimum and maximum, quantiles, etc.).

NumPy has fast built-in aggregation functions for working on arrays; we'll discuss and demonstrate some of them here.

Summing the Values in an Array

As a quick example, consider computing the sum of all values in an array. Python itself can do this using the built-in `sum` function:

```
In[1]: import numpy as np  
In[2]: L = np.random.random(100)  
       sum(L)  
Out[2]: 55.61209116604941
```

The syntax is quite similar to that of NumPy's `sum` function, and the result is the same in the simplest case:

```
In[3]: np.sum(L)  
Out[3]: 55.612091166049424
```

However, because it executes the operation in compiled code, NumPy's version of the operation is computed much more quickly:

```
In[4]: big_array = np.random.rand(1000000)  
       %timeit sum(big_array)  
       %timeit np.sum(big_array)  
  
10 loops, best of 3: 104 ms per loop  
1000 loops, best of 3: 442 µs per loop
```

Be careful, though: the `sum` function and the `np.sum` function are not identical, which can sometimes lead to confusion! In particular, their optional arguments have different meanings, and `np.sum` is aware of multiple array dimensions, as we will see in the following section.

Minimum and Maximum

Similarly, Python has built-in `min` and `max` functions, used to find the minimum value and maximum value of any given array:

```
In[5]: min(big_array), max(big_array)  
Out[5]: (1.1717128136634614e-06, 0.9999976784968716)
```

NumPy's corresponding functions have similar syntax, and again operate much more quickly:

```
In[6]: np.min(big_array), np.max(big_array)  
Out[6]: (1.1717128136634614e-06, 0.9999976784968716)
```

```
In[7]: %timeit min(big_array)
%timeit np.min(big_array)

10 loops, best of 3: 82.3 ms per loop
1000 loops, best of 3: 497 µs per loop
```

For `min`, `max`, `sum`, and several other NumPy aggregates, a shorter syntax is to use methods of the array object itself:

```
In[8]: print(big_array.min(), big_array.max(), big_array.sum())

1.17171281366e-06 0.999997678497 499911.628197
```

Whenever possible, make sure that you are using the NumPy version of these aggregates when operating on NumPy arrays!

Multidimensional aggregates

One common type of aggregation operation is an aggregate along a row or column. Say you have some data stored in a two-dimensional array:

```
In[9]: M = np.random.random((3, 4))
print(M)

[[ 0.8967576   0.03783739  0.75952519  0.06682827]
 [ 0.8354065   0.99196818  0.19544769  0.43447084]
 [ 0.66859307  0.15038721  0.37911423  0.6687194 ]]
```

By default, each NumPy aggregation function will return the aggregate over the entire array:

```
In[10]: M.sum()

Out[10]: 6.0850555667307118
```

Aggregation functions take an additional argument specifying the *axis* along which the aggregate is computed. For example, we can find the minimum value within each column by specifying `axis=0`:

```
In[11]: M.min(axis=0)

Out[11]: array([ 0.66859307,  0.03783739,  0.19544769,  0.06682827])
```

The function returns four values, corresponding to the four columns of numbers.

Similarly, we can find the maximum value within each row:

```
In[12]: M.max(axis=1)

Out[12]: array([ 0.8967576 ,  0.99196818,  0.6687194 ])
```

The way the axis is specified here can be confusing to users coming from other languages. The `axis` keyword specifies the *dimension of the array that will be collapsed*, rather than the dimension that will be returned. So specifying `axis=0` means that the

first axis will be collapsed: for two-dimensional arrays, this means that values within each column will be aggregated.

Other aggregation functions

NumPy provides many other aggregation functions, but we won't discuss them in detail here. Additionally, most aggregates have a NaN-safe counterpart that computes the result while ignoring missing values, which are marked by the special IEEE floating-point NaN value (for a fuller discussion of missing data, see “[Handling Missing Data](#)” on page 119). Some of these NaN-safe functions were not added until NumPy 1.8, so they will not be available in older NumPy versions.

[Table 2-3](#) provides a list of useful aggregation functions available in NumPy.

Table 2-3. Aggregation functions available in NumPy

Function Name	Nan-safe Version	Description
np.sum	np.nansum	Compute sum of elements
np.prod	np.nanprod	Compute product of elements
np.mean	np.nanmean	Compute median of elements
np.std	np.nanstd	Compute standard deviation
np.var	np.nanvar	Compute variance
np.min	np.nanmin	Find minimum value
np.max	np.nanmax	Find maximum value
np.argmax	np.nanargmin	Find index of minimum value
np.argmax	np.nanargmax	Find index of maximum value
np.median	np.nanmedian	Compute median of elements
np.percentile	np.nanpercentile	Compute rank-based statistics of elements
np.any	N/A	Evaluate whether any elements are true
np.all	N/A	Evaluate whether all elements are true

We will see these aggregates often throughout the rest of the book.

Example: What Is the Average Height of US Presidents?

Aggregates available in NumPy can be extremely useful for summarizing a set of values. As a simple example, let's consider the heights of all US presidents. This data is available in the file *president_heights.csv*, which is a simple comma-separated list of labels and values:

```
In[13]: !head -4 data/president_heights.csv
order,name,height(cm)
1,George Washington,189
```

```
2,John Adams,170  
3,Thomas Jefferson,189
```

We'll use the Pandas package, which we'll explore more fully in [Chapter 3](#), to read the file and extract this information (note that the heights are measured in centimeters):

```
In[14]: import pandas as pd  
data = pd.read_csv('data/president_heights.csv')  
heights = np.array(data['height(cm)'])  
print(heights)  
  
[189 170 189 163 183 171 185 168 173 183 173 173 175 178 183 193 178 173  
174 183 183 168 170 178 182 180 183 178 182 188 175 179 183 193 182 183  
177 185 188 188 182 185]
```

Now that we have this data array, we can compute a variety of summary statistics:

```
In[15]: print("Mean height:      ", heights.mean())  
print("Standard deviation:", heights.std())  
print("Minimum height:    ", heights.min())  
print("Maximum height:    ", heights.max())  
  
Mean height:      179.738095238  
Standard deviation: 6.93184344275  
Minimum height:    163  
Maximum height:    193
```

Note that in each case, the aggregation operation reduced the entire array to a single summarizing value, which gives us information about the distribution of values. We may also wish to compute quantiles:

```
In[16]: print("25th percentile:   ", np.percentile(heights, 25))  
print("Median:           ", np.median(heights))  
print("75th percentile:   ", np.percentile(heights, 75))  
  
25th percentile:   174.25  
Median:           182.0  
75th percentile:   183.0
```

We see that the median height of US presidents is 182 cm, or just shy of six feet.

Of course, sometimes it's more useful to see a visual representation of this data, which we can accomplish using tools in Matplotlib (we'll discuss Matplotlib more fully in [Chapter 4](#)). For example, this code generates the chart shown in [Figure 2-3](#):

```
In[17]: %matplotlib inline  
import matplotlib.pyplot as plt  
import seaborn; seaborn.set() # set plot style  
  
In[18]: plt.hist(heights)  
plt.title('Height Distribution of US Presidents')  
plt.xlabel('height (cm)')  
plt.ylabel('number');
```



Figure 2-3. Histogram of presidential heights

These aggregates are some of the fundamental pieces of exploratory data analysis that we'll explore in more depth in later chapters of the book.

Computation on Arrays: Broadcasting

We saw in the previous section how NumPy's universal functions can be used to *vectorize* operations and thereby remove slow Python loops. Another means of vectorizing operations is to use NumPy's *broadcasting* functionality. Broadcasting is simply a set of rules for applying binary ufuncs (addition, subtraction, multiplication, etc.) on arrays of different sizes.

Introducing Broadcasting

Recall that for arrays of the same size, binary operations are performed on an element-by-element basis:

```
In[1]: import numpy as np
In[2]: a = np.array([0, 1, 2])
        b = np.array([5, 5, 5])
        a + b
Out[2]: array([5, 6, 7])
```

Broadcasting allows these types of binary operations to be performed on arrays of different sizes—for example, we can just as easily add a scalar (think of it as a zero-dimensional array) to an array:

```
In[3]: a + 5  
Out[3]: array([5, 6, 7])
```

We can think of this as an operation that stretches or duplicates the value 5 into the array [5, 5, 5], and adds the results. The advantage of NumPy's broadcasting is that this duplication of values does not actually take place, but it is a useful mental model as we think about broadcasting.

We can similarly extend this to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensional array:

```
In[4]: M = np.ones((3, 3))  
M  
  
Out[4]: array([[ 1.,  1.,  1.],  
               [ 1.,  1.,  1.],  
               [ 1.,  1.,  1.]])  
  
In[5]: M + a  
  
Out[5]: array([[ 1.,  2.,  3.],  
               [ 1.,  2.,  3.],  
               [ 1.,  2.,  3.]])
```

Here the one-dimensional array `a` is stretched, or broadcast, across the second dimension in order to match the shape of `M`.

While these examples are relatively easy to understand, more complicated cases can involve broadcasting of both arrays. Consider the following example:

```
In[6]: a = np.arange(3)  
       b = np.arange(3)[:, np.newaxis]  
  
       print(a)  
       print(b)  
  
[0 1 2]  
[[0]  
 [1]  
 [2]]  
  
In[7]: a + b  
  
Out[7]: array([[0, 1, 2],  
               [1, 2, 3],  
               [2, 3, 4]])
```

Just as before we stretched or broadcasted one value to match the shape of the other, here we've stretched *both* `a` and `b` to match a common shape, and the result is a two-dimensional array! The geometry of these examples is visualized in [Figure 2-4](#).¹

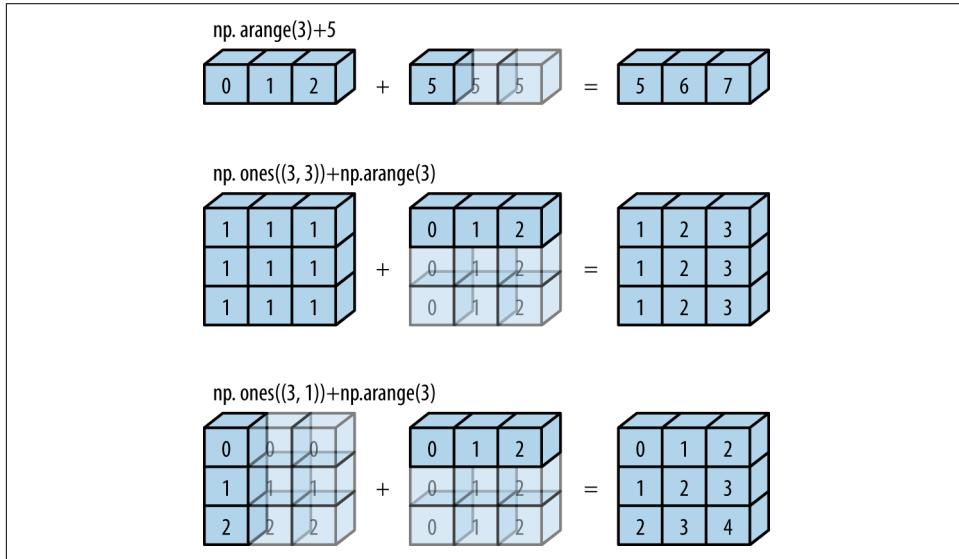


Figure 2-4. Visualization of NumPy broadcasting

The light boxes represent the broadcasted values: again, this extra memory is not actually allocated in the course of the operation, but it can be useful conceptually to imagine that it is.

Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

- Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.
- Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

¹ Code to produce this plot can be found in the online appendix, and is adapted from source published in the [astroML documentation](#). Used with permission.

To make these rules clear, let's consider a few examples in detail.

Broadcasting example 1

Let's look at adding a two-dimensional array to a one-dimensional array:

```
In[8]: M = np.ones((2, 3))
a = np.arange(3)
```

Let's consider an operation on these two arrays. The shapes of the arrays are:

```
M.shape = (2, 3)
a.shape = (3,)
```

We see by rule 1 that the array `a` has fewer dimensions, so we pad it on the left with ones:

```
M.shape -> (2, 3)
a.shape -> (1, 3)
```

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

```
M.shape -> (2, 3)
a.shape -> (2, 3)
```

The shapes match, and we see that the final shape will be $(2, 3)$:

```
In[9]: M + a
Out[9]: array([[ 1.,  2.,  3.],
 [ 1.,  2.,  3.]])
```

Broadcasting example 2

Let's take a look at an example where both arrays need to be broadcast:

```
In[10]: a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

Again, we'll start by writing out the shape of the arrays:

```
a.shape = (3, 1)
b.shape = (3,)
```

Rule 1 says we must pad the shape of `b` with ones:

```
a.shape -> (3, 1)
b.shape -> (1, 3)
```

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

```
a.shape -> (3, 3)
b.shape -> (3, 3)
```

Because the result matches, these shapes are compatible. We can see this here:

```
In[11]: a + b
Out[11]: array([[0, 1, 2],
 [1, 2, 3],
 [2, 3, 4]])
```

Broadcasting example 3

Now let's take a look at an example in which the two arrays are not compatible:

```
In[12]: M = np.ones((3, 2))
a = np.arange(3)
```

This is just a slightly different situation than in the first example: the matrix `M` is transposed. How does this affect the calculation? The shapes of the arrays are:

```
M.shape = (3, 2)
a.shape = (3,)
```

Again, rule 1 tells us that we must pad the shape of `a` with ones:

```
M.shape -> (3, 2)
a.shape -> (1, 3)
```

By rule 2, the first dimension of `a` is stretched to match that of `M`:

```
M.shape -> (3, 2)
a.shape -> (3, 3)
```

Now we hit rule 3—the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

```
In[13]: M + a
```

```
-----
```

```
ValueError                                     Traceback (most recent call last)

<ipython-input-13-9e16e9f98da6> in <module>()
      1 M + a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Note the potential confusion here: you could imagine making `a` and `M` compatible by, say, padding `a`'s shape with ones on the right rather than the left. But this is not how the broadcasting rules work! That sort of flexibility might be useful in some cases, but it would lead to potential areas of ambiguity. If right-side padding is what you'd like, you can do this explicitly by reshaping the array (we'll use the `np.newaxis` keyword introduced in “[The Basics of NumPy Arrays](#)” on page 42):

```
In[14]: a[:, np.newaxis].shape
```

```
Out[14]: (3, 1)
```

```
In[15]: M + a[:, np.newaxis]
```

```
Out[15]: array([[ 1.,  1.],
                [ 2.,  2.],
                [ 3.,  3.]])
```

Also note that while we've been focusing on the `+` operator here, these broadcasting rules apply to *any* binary ufunc. For example, here is the `logaddexp(a, b)` function, which computes `log(exp(a) + exp(b))` with more precision than the naive approach:

```
In[16]: np.logaddexp(M, a[:, np.newaxis])
```

```
Out[16]: array([[ 1.31326169,  1.31326169],
                [ 1.69314718,  1.69314718],
                [ 2.31326169,  2.31326169]])
```

For more information on the many available universal functions, refer to “[Computation on NumPy Arrays: Universal Functions](#)” on page 50.

Broadcasting in Practice

Broadcasting operations form the core of many examples we'll see throughout this book. We'll now take a look at a couple simple examples of where they can be useful.

Centering an array

In the previous section, we saw that ufuncs allow a NumPy user to remove the need to explicitly write slow Python loops. Broadcasting extends this ability. One com-

monly seen example is centering an array of data. Imagine you have an array of 10 observations, each of which consists of 3 values. Using the standard convention (see “[Data Representation in Scikit-Learn](#)” on page 343), we’ll store this in a 10×3 array:

```
In[17]: X = np.random.random((10, 3))
```

We can compute the mean of each feature using the `mean` aggregate across the first dimension:

```
In[18]: Xmean = X.mean(0)
Xmean
```

```
Out[18]: array([ 0.53514715,  0.66567217,  0.44385899])
```

And now we can center the `X` array by subtracting the mean (this is a broadcasting operation):

```
In[19]: X_centered = X - Xmean
```

To double-check that we’ve done this correctly, we can check that the centered array has near zero mean:

```
In[20]: X_centered.mean(0)
```

```
Out[20]: array([ 2.22044605e-17, -7.77156117e-17, -1.66533454e-17])
```

To within-machine precision, the mean is now zero.

Plotting a two-dimensional function

One place that broadcasting is very useful is in displaying images based on two-dimensional functions. If we want to define a function $z = f(x, y)$, broadcasting can be used to compute the function across the grid:

```
In[21]: # x and y have 50 steps from 0 to 5
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50)[:, np.newaxis]

z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

We’ll use Matplotlib to plot this two-dimensional array (these tools will be discussed in full in “[Density and Contour Plots](#)” on page 241):

```
In[22]: %matplotlib inline
import matplotlib.pyplot as plt

In[23]: plt.imshow(z, origin='lower', extent=[0, 5, 0, 5],
cmap='viridis')
plt.colorbar();
```

The result, shown in [Figure 2-5](#), is a compelling visualization of the two-dimensional function.



Figure 2-5. Visualization of a 2D array

Comparisons, Masks, and Boolean Logic

This section covers the use of Boolean masks to examine and manipulate values within NumPy arrays. Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers that are above some threshold. In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.

Example: Counting Rainy Days

Imagine you have a series of data that represents the amount of precipitation each day for a year in a given city. For example, here we'll load the daily rainfall statistics for the city of Seattle in 2014, using Pandas (which is covered in more detail in [Chapter 3](#)):

```
In[1]: import numpy as np
       import pandas as pd

       # use Pandas to extract rainfall inches as a NumPy array
       rainfall = pd.read_csv('data/Seattle2014.csv')['PRCP'].values
       inches = rainfall / 254 # 1/10mm -> inches
       inches.shape

Out[1]: (365,)
```

The array contains 365 values, giving daily rainfall in inches from January 1 to December 31, 2014.

As a first quick visualization, let's look at the histogram of rainy days shown in [Figure 2-6](#), which was generated using Matplotlib (we will explore this tool more fully in [Chapter 4](#)):

```
In[2]: %matplotlib inline  
import matplotlib.pyplot as plt  
import seaborn; seaborn.set() # set plot styles  
  
In[3]: plt.hist(inches, 40);
```



Figure 2-6. Histogram of 2014 rainfall in Seattle

This histogram gives us a general idea of what the data looks like: despite its reputation, the vast majority of days in Seattle saw near zero measured rainfall in 2014. But this doesn't do a good job of conveying some information we'd like to see: for example, how many rainy days were there in the year? What is the average precipitation on those rainy days? How many days were there with more than half an inch of rain?

Digging into the data

One approach to this would be to answer these questions by hand: loop through the data, incrementing a counter each time we see values in some desired range. For reasons discussed throughout this chapter, such an approach is very inefficient, both from the standpoint of time writing code and time computing the result. We saw in “[Computation on NumPy Arrays: Universal Functions](#)” on page 50 that NumPy's ufuncs can be used in place of loops to do fast element-wise arithmetic operations on arrays; in the same way, we can use other ufuncs to do element-wise *comparisons* over arrays, and we can then manipulate the results to answer the questions we have. We'll leave the data aside for right now, and discuss some general tools in NumPy to use *masking* to quickly answer these types of questions.

Comparison Operators as ufuncs

In “[Computation on NumPy Arrays: Universal Functions](#)” on page 50 we introduced ufuncs, and focused in particular on arithmetic operators. We saw that using `+`, `-`, `*`, `/`,

and others on arrays leads to element-wise operations. NumPy also implements comparison operators such as `<` (less than) and `>` (greater than) as element-wise ufuncs. The result of these comparison operators is always an array with a Boolean data type. All six of the standard comparison operations are available:

```
In[4]: x = np.array([1, 2, 3, 4, 5])
In[5]: x < 3 # less than
Out[5]: array([ True,  True, False, False, False], dtype=bool)
In[6]: x > 3 # greater than
Out[6]: array([False, False, False,  True,  True], dtype=bool)
In[7]: x <= 3 # less than or equal
Out[7]: array([ True,  True,  True, False, False], dtype=bool)
In[8]: x >= 3 # greater than or equal
Out[8]: array([False, False,  True,  True,  True], dtype=bool)
In[9]: x != 3 # not equal
Out[9]: array([ True,  True, False,  True,  True], dtype=bool)
In[10]: x == 3 # equal
Out[10]: array([False, False,  True, False, False], dtype=bool)
```

It is also possible to do an element-by-element comparison of two arrays, and to include compound expressions:

```
In[11]: (2 * x) == (x ** 2)
Out[11]: array([False,  True, False, False, False], dtype=bool)
```

As in the case of arithmetic operators, the comparison operators are implemented as ufuncs in NumPy; for example, when you write `x < 3`, internally NumPy uses `np.less(x, 3)`. A summary of the comparison operators and their equivalent ufunc is shown here:

Operator	Equivalent ufunc
<code>==</code>	<code>np.equal</code>
<code>!=</code>	<code>np.not_equal</code>
<code><</code>	<code>np.less</code>
<code><=</code>	<code>np.less_equal</code>
<code>></code>	<code>np.greater</code>
<code>>=</code>	<code>np.greater_equal</code>

Just as in the case of arithmetic ufuncs, these will work on arrays of any size and shape. Here is a two-dimensional example:

```
In[12]: rng = np.random.RandomState(0)
         x = rng.randint(10, size=(3, 4))
         x

Out[12]: array([[5, 0, 3, 3],
               [7, 9, 3, 5],
               [2, 4, 7, 6]])

In[13]: x < 6

Out[13]: array([[ True,  True,  True,  True],
                [False, False,  True,  True],
                [ True,  True, False, False]], dtype=bool)
```

In each case, the result is a Boolean array, and NumPy provides a number of straightforward patterns for working with these Boolean results.

Working with Boolean Arrays

Given a Boolean array, there are a host of useful operations you can do. We'll work with `x`, the two-dimensional array we created earlier:

```
In[14]: print(x)

[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
```

Counting entries

To count the number of `True` entries in a Boolean array, `np.count_nonzero` is useful:

```
In[15]: # how many values less than 6?
         np.count_nonzero(x < 6)

Out[15]: 8
```

We see that there are eight array entries that are less than 6. Another way to get at this information is to use `np.sum`; in this case, `False` is interpreted as 0, and `True` is interpreted as 1:

```
In[16]: np.sum(x < 6)

Out[16]: 8
```

The benefit of `sum()` is that like with other NumPy aggregation functions, this summation can be done along rows or columns as well:

```
In[17]: # how many values less than 6 in each row?
         np.sum(x < 6, axis=1)

Out[17]: array([4, 2, 2])
```

This counts the number of values less than 6 in each row of the matrix.

If we're interested in quickly checking whether any or all the values are true, we can use (you guessed it) `np.any()` or `np.all()`:

```
In[18]: # are there any values greater than 8?  
np.any(x > 8)  
  
Out[18]: True  
  
In[19]: # are there any values less than zero?  
np.any(x < 0)  
  
Out[19]: False  
  
In[20]: # are all values less than 10?  
np.all(x < 10)  
  
Out[20]: True  
  
In[21]: # are all values equal to 6?  
np.all(x == 6)  
  
Out[21]: False
```

`np.all()` and `np.any()` can be used along particular axes as well. For example:

```
In[22]: # are all values in each row less than 8?  
np.all(x < 8, axis=1)  
  
Out[22]: array([ True, False,  True], dtype=bool)
```

Here all the elements in the first and third rows are less than 8, while this is not the case for the second row.

Finally, a quick warning: as mentioned in “[Aggregations: Min, Max, and Everything in Between](#)” on page 58, Python has built-in `sum()`, `any()`, and `all()` functions. These have a different syntax than the NumPy versions, and in particular will fail or produce unintended results when used on multidimensional arrays. Be sure that you are using `np.sum()`, `np.any()`, and `np.all()` for these examples!

Boolean operators

We've already seen how we might count, say, all days with rain less than four inches, or all days with rain greater than two inches. But what if we want to know about all days with rain less than four inches *and* greater than one inch? This is accomplished through Python's *bitwise logic operators*, `&`, `|`, `^`, and `~`. Like with the standard arithmetic operators, NumPy overloads these as ufuncs that work element-wise on (usually Boolean) arrays.

For example, we can address this sort of compound question as follows:

```
In[23]: np.sum((inches > 0.5) & (inches < 1))  
  
Out[23]: 29
```

So we see that there are 29 days with rainfall between 0.5 and 1.0 inches.

Note that the parentheses here are important—because of operator precedence rules, with parentheses removed this expression would be evaluated as follows, which results in an error:

```
inches > (0.5 & inches) < 1
```

Using the equivalence of $A \text{ AND } B$ and $\text{NOT}(A \text{ OR } B)$ (which you may remember if you've taken an introductory logic course), we can compute the same result in a different manner:

```
In[24]: np.sum(~( (inches <= 0.5) | (inches >= 1) ))
```

```
Out[24]: 29
```

Combining comparison operators and Boolean operators on arrays can lead to a wide range of efficient logical operations.

The following table summarizes the bitwise Boolean operators and their equivalent ufuncs:

Operator	Equivalent ufunc
&	np.bitwise_and
	np.bitwise_or
^	np.bitwise_xor
~	np.bitwise_not

Using these tools, we might start to answer the types of questions we have about our weather data. Here are some examples of results we can compute when combining masking with aggregations:

```
In[25]: print("Number days without rain:      ", np.sum(inches == 0))
          print("Number days with rain:        ", np.sum(inches != 0))
          print("Days with more than 0.5 inches:", np.sum(inches > 0.5))
          print("Rainy days with < 0.1 inches : ", np.sum((inches > 0) &
                                         (inches < 0.2)))
```



```
Number days without rain:      215
Number days with rain:        150
Days with more than 0.5 inches: 37
Rainy days with < 0.1 inches : 75
```

Boolean Arrays as Masks

In the preceding section, we looked at aggregates computed directly on Boolean arrays. A more powerful pattern is to use Boolean arrays as masks, to select particular subsets of the data themselves. Returning to our `x` array from before, suppose we want an array of all values in the array that are less than, say, 5:

```
In[26]: x  
Out[26]: array([[5, 0, 3, 3],  
                 [7, 9, 3, 5],  
                 [2, 4, 7, 6]])
```

We can obtain a Boolean array for this condition easily, as we've already seen:

```
In[27]: x < 5  
Out[27]: array([[False, True, True, True],  
                 [False, False, True, False],  
                 [ True, True, False, False]]], dtype=bool)
```

Now to *select* these values from the array, we can simply index on this Boolean array; this is known as a *masking* operation:

```
In[28]: x[x < 5]  
Out[28]: array([0, 3, 3, 3, 2, 4])
```

What is returned is a one-dimensional array filled with all the values that meet this condition; in other words, all the values in positions at which the mask array is `True`.

We are then free to operate on these values as we wish. For example, we can compute some relevant statistics on our Seattle rain data:

```
In[29]:  
# construct a mask of all rainy days  
rainy = (inches > 0)  
  
# construct a mask of all summer days (June 21st is the 172nd day)  
summer = (np.arange(365) - 172 < 90) & (np.arange(365) - 172 > 0)  
  
print("Median precip on rainy days in 2014 (inches): ",  
     np.median(inches[rainy]))  
print("Median precip on summer days in 2014 (inches): ",  
     np.median(inches[summer]))  
print("Maximum precip on summer days in 2014 (inches): ",  
     np.max(inches[summer]))  
print("Median precip on non-summer rainy days (inches):",  
     np.median(inches[rainy & ~summer]))  
  
Median precip on rainy days in 2014 (inches): 0.194881889764  
Median precip on summer days in 2014 (inches): 0.0  
Maximum precip on summer days in 2014 (inches): 0.850393700787  
Median precip on non-summer rainy days (inches): 0.200787401575
```

By combining Boolean operations, masking operations, and aggregates, we can very quickly answer these sorts of questions for our dataset.

Using the Keywords and/or Versus the Operators &/|

One common point of confusion is the difference between the keywords `and` and `or` on one hand, and the operators `&` and `|` on the other hand. When would you use one versus the other?

The difference is this: `and` and `or` gauge the truth or falsehood of *entire object*, while `&` and `|` refer to *bits within each object*.

When you use `and` or `or`, it's equivalent to asking Python to treat the object as a single Boolean entity. In Python, all nonzero integers will evaluate as `True`. Thus:

```
In[30]: bool(42), bool(0)  
Out[30]: (True, False)  
  
In[31]: bool(42 and 0)  
Out[31]: False  
  
In[32]: bool(42 or 0)  
Out[32]: True
```

When you use `&` and `|` on integers, the expression operates on the bits of the element, applying the `and` or the `or` to the individual bits making up the number:

```
In[33]: bin(42)  
Out[33]: '0b101010'  
  
In[34]: bin(59)  
Out[34]: '0b111011'  
  
In[35]: bin(42 & 59)  
Out[35]: '0b101010'  
  
In[36]: bin(42 | 59)  
Out[36]: '0b111011'
```

Notice that the corresponding bits of the binary representation are compared in order to yield the result.

When you have an array of Boolean values in NumPy, this can be thought of as a string of bits where `1 = True` and `0 = False`, and the result of `&` and `|` operates in a similar manner as before:

```
In[37]: A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)  
B = np.array([1, 1, 0, 1, 1, 1], dtype=bool)  
A | B  
  
Out[37]: array([ True,  True,  True, False,  True,  True], dtype=bool)
```

Using `or` on these arrays will try to evaluate the truth or falsehood of the entire array object, which is not a well-defined value:

```
In[38]: A or B
```

```
-----
```

```
ValueError
```

```
Traceback (most recent call last)
```

```
<ipython-input-38-5d8e4f2e21c0> in <module>()
----> 1 A or B
```

```
ValueError: The truth value of an array with more than one element is...
```

Similarly, when doing a Boolean expression on a given array, you should use `|` or `&` rather than `or` or `and`:

```
In[39]: x = np.arange(10)
(x > 4) & (x < 8)
```

```
Out[39]: array([False, False, ..., True, True, False, False], dtype=bool)
```

Trying to evaluate the truth or falsehood of the entire array will give the same `ValueError` we saw previously:

```
In[40]: (x > 4) and (x < 8)
```

```
-----
```

```
ValueError
```

```
Traceback (most recent call last)
```

```
<ipython-input-40-3d24f1ffd63d> in <module>()
----> 1 (x > 4) and (x < 8)
```

```
ValueError: The truth value of an array with more than one element is...
```

So remember this: `and` and `or` perform a single Boolean evaluation on an entire object, while `&` and `|` perform multiple Boolean evaluations on the content (the individual bits or bytes) of an object. For Boolean NumPy arrays, the latter is nearly always the desired operation.

Fancy Indexing

In the previous sections, we saw how to access and modify portions of arrays using simple indices (e.g., `arr[0]`), slices (e.g., `arr[:5]`), and Boolean masks (e.g., `arr[arr > 0]`). In this section, we'll look at another style of array indexing, known as *fancy indexing*. Fancy indexing is like the simple indexing we've already seen, but we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

Exploring Fancy Indexing

Fancy indexing is conceptually simple: it means passing an array of indices to access multiple array elements at once. For example, consider the following array:

```
In[1]: import numpy as np  
rand = np.random.RandomState(42)  
  
x = rand.randint(100, size=10)  
print(x)  
  
[51 92 14 71 60 20 82 86 74 74]
```

Suppose we want to access three different elements. We could do it like this:

```
In[2]: [x[3], x[7], x[2]]  
Out[2]: [71, 86, 14]
```

Alternatively, we can pass a single list or array of indices to obtain the same result:

```
In[3]: ind = [3, 7, 4]  
x[ind]  
Out[3]: array([71, 86, 60])
```

With fancy indexing, the shape of the result reflects the shape of the *index arrays* rather than the shape of the *array being indexed*:

```
In[4]: ind = np.array([[3, 7],  
                     [4, 5]])  
x[ind]  
Out[4]: array([[71, 86],  
              [60, 20]])
```

Fancy indexing also works in multiple dimensions. Consider the following array:

```
In[5]: X = np.arange(12).reshape((3, 4))  
X  
Out[5]: array([[ 0,  1,  2,  3],  
              [ 4,  5,  6,  7],  
              [ 8,  9, 10, 11]])
```

Like with standard indexing, the first index refers to the row, and the second to the column:

```
In[6]: row = np.array([0, 1, 2])  
col = np.array([2, 1, 3])  
X[row, col]  
Out[6]: array([ 2,  5, 11])
```

Notice that the first value in the result is $X[0, 2]$, the second is $X[1, 1]$, and the third is $X[2, 3]$. The pairing of indices in fancy indexing follows all the broadcasting rules that were mentioned in “Computation on Arrays: Broadcasting” on page 63. So,

for example, if we combine a column vector and a row vector within the indices, we get a two-dimensional result:

```
In[7]: X[row[:, np.newaxis], col]
```

```
Out[7]: array([[ 2,  1,  3],
               [ 6,  5,  7],
               [10,  9, 11]])
```

Here, each row value is matched with each column vector, exactly as we saw in broadcasting of arithmetic operations. For example:

```
In[8]: row[:, np.newaxis] * col
```

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

It is always important to remember with fancy indexing that the return value reflects the *broadcasted shape of the indices*, rather than the shape of the array being indexed.

Combined Indexing

For even more powerful operations, fancy indexing can be combined with the other indexing schemes we've seen:

```
In[9]: print(X)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

We can combine fancy and simple indices:

```
In[10]: X[2, [2, 0, 1]]
```

```
Out[10]: array([10,  8,  9])
```

We can also combine fancy indexing with slicing:

```
In[11]: X[1:, [2, 0, 1]]
```

```
Out[11]: array([[ 6,  4,  5],
                [10,  8,  9]])
```

And we can combine fancy indexing with masking:

```
In[12]: mask = np.array([1, 0, 1, 0], dtype=bool)
          X[row[:, np.newaxis], mask]
```

```
Out[12]: array([[ 0,  2],
                 [ 4,  6],
                 [ 8, 10]])
```

All of these indexing options combined lead to a very flexible set of operations for accessing and modifying array values.

Example: Selecting Random Points

One common use of fancy indexing is the selection of subsets of rows from a matrix. For example, we might have an N by D matrix representing N points in D dimensions, such as the following points drawn from a two-dimensional normal distribution:

```
In[13]: mean = [0, 0]
cov = [[1, 2],
       [2, 5]]
X = rand.multivariate_normal(mean, cov, 100)
X.shape

Out[13]: (100, 2)
```

Using the plotting tools we will discuss in [Chapter 4](#), we can visualize these points as a scatter plot ([Figure 2-7](#)):

```
In[14]: %matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # for plot styling

plt.scatter(X[:, 0], X[:, 1]);
```



Figure 2-7. Normally distributed points

Let's use fancy indexing to select 20 random points. We'll do this by first choosing 20 random indices with no repeats, and use these indices to select a portion of the original array:

```
In[15]: indices = np.random.choice(X.shape[0], 20, replace=False)
indices

Out[15]: array([93, 45, 73, 81, 50, 10, 98, 94, 4, 64, 65, 89, 47, 84, 82,
   80, 25, 90, 63, 20])
```

```
In[16]: selection = X[:, 0] # fancy indexing here  
selection.shape
```

```
Out[16]: (20, 2)
```

Now to see which points were selected, let's over-plot large circles at the locations of the selected points (Figure 2-8):

```
In[17]: plt.scatter(X[:, 0], X[:, 1], alpha=0.3)  
plt.scatter(selection[:, 0], selection[:, 1],  
            facecolor='none', s=200);
```

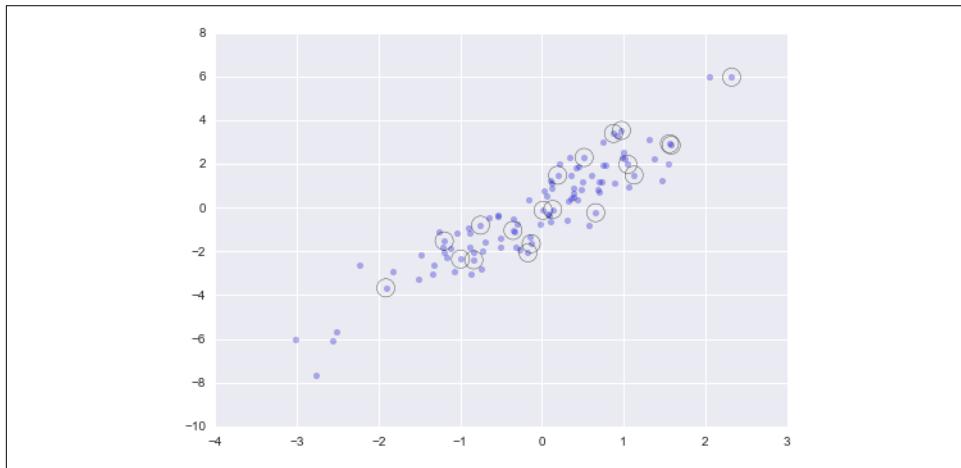


Figure 2-8. Random selection among points

This sort of strategy is often used to quickly partition datasets, as is often needed in train/test splitting for validation of statistical models (see “[Hyperparameters and Model Validation](#)” on page 359), and in sampling approaches to answering statistical questions.

Modifying Values with Fancy Indexing

Just as fancy indexing can be used to access parts of an array, it can also be used to modify parts of an array. For example, imagine we have an array of indices and we'd like to set the corresponding items in an array to some value:

```
In[18]: x = np.arange(10)  
i = np.array([2, 1, 8, 4])  
x[i] = 99  
print(x)
```

```
[ 0 99 99  3 99  5  6  7 99  9]
```

We can use any assignment-type operator for this. For example:

```
In[19]: x[i] -= 10
print(x)

[ 0 89 89  3 89  5  6  7 89  9]
```

Notice, though, that repeated indices with these operations can cause some potentially unexpected results. Consider the following:

```
In[20]: x = np.zeros(10)
x[[0, 0]] = [4, 6]
print(x)

[ 6.  0.  0.  0.  0.  0.  0.  0.  0.]
```

Where did the 4 go? The result of this operation is to first assign $x[0] = 4$, followed by $x[0] = 6$. The result, of course, is that $x[0]$ contains the value 6.

Fair enough, but consider this operation:

```
In[21]: i = [2, 3, 3, 4, 4, 4]
x[i] += 1
x

Out[21]: array([ 6.,  0.,  1.,  1.,  0.,  0.,  0.,  0.,  0.])
```

You might expect that $x[3]$ would contain the value 2, and $x[4]$ would contain the value 3, as this is how many times each index is repeated. Why is this not the case? Conceptually, this is because $x[i] += 1$ is meant as a shorthand of $x[i] = x[i] + 1$. $x[i] + 1$ is evaluated, and then the result is assigned to the indices in x . With this in mind, it is not the augmentation that happens multiple times, but the assignment, which leads to the rather nonintuitive results.

So what if you want the other behavior where the operation is repeated? For this, you can use the `at()` method of ufuncs (available since NumPy 1.8), and do the following:

```
In[22]: x = np.zeros(10)
np.add.at(x, i, 1)
print(x)

[ 0.  0.  1.  2.  3.  0.  0.  0.  0.]
```

The `at()` method does an in-place application of the given operator at the specified indices (here, i) with the specified value (here, 1). Another method that is similar in spirit is the `reduceat()` method of ufuncs, which you can read about in the NumPy documentation.

Example: Binning Data

You can use these ideas to efficiently bin data to create a histogram by hand. For example, imagine we have 1,000 values and would like to quickly find where they fall within an array of bins. We could compute it using `ufunc.at` like this:

```
In[23]: np.random.seed(42)
x = np.random.randn(100)

# compute a histogram by hand
bins = np.linspace(-5, 5, 20)
counts = np.zeros_like(bins)

# find the appropriate bin for each x
i = np.searchsorted(bins, x)

# add 1 to each of these bins
np.add.at(counts, i, 1)
```

The counts now reflect the number of points within each bin—in other words, a histogram (Figure 2-9):

```
In[24]: # plot the results
plt.plot(bins, counts, linestyle='steps');
```

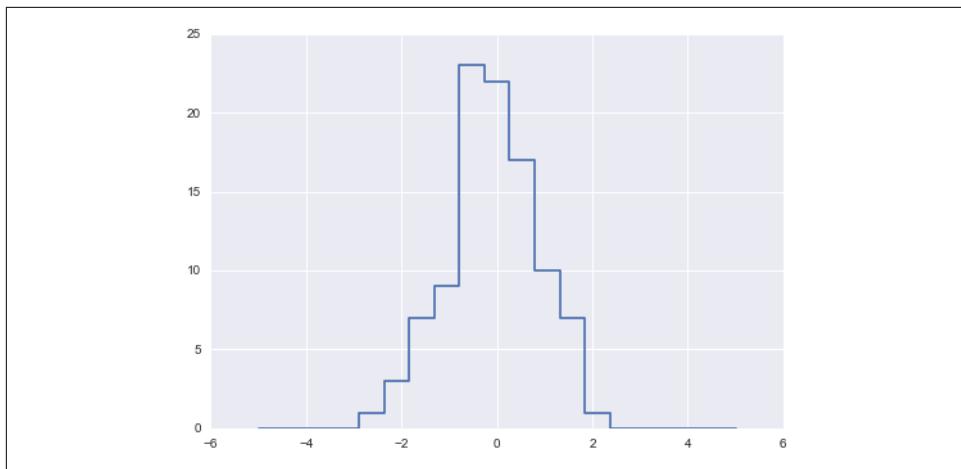


Figure 2-9. A histogram computed by hand

Of course, it would be silly to have to do this each time you want to plot a histogram. This is why Matplotlib provides the `plt.hist()` routine, which does the same in a single line:

```
plt.hist(x, bins, histtype='step');
```

This function will create a nearly identical plot to the one seen here. To compute the binning, Matplotlib uses the `np.histogram` function, which does a very similar computation to what we did before. Let's compare the two here:

```
In[25]: print("NumPy routine:")
%timeit counts, edges = np.histogram(x, bins)
```

```
print("Custom routine:")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)

NumPy routine:
10000 loops, best of 3: 97.6 µs per loop
Custom routine:
10000 loops, best of 3: 19.5 µs per loop
```

Our own one-line algorithm is several times faster than the optimized algorithm in NumPy! How can this be? If you dig into the `np.histogram` source code (you can do this in IPython by typing `np.histogram??`), you'll see that it's quite a bit more involved than the simple search-and-count that we've done; this is because NumPy's algorithm is more flexible, and particularly is designed for better performance when the number of data points becomes large:

```
In[26]: x = np.random.randn(1000000)
print("NumPy routine:")
%timeit counts, edges = np.histogram(x, bins)

print("Custom routine:")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)

NumPy routine:
10 loops, best of 3: 68.7 ms per loop
Custom routine:
10 loops, best of 3: 135 ms per loop
```

What this comparison shows is that algorithmic efficiency is almost never a simple question. An algorithm efficient for large datasets will not always be the best choice for small datasets, and vice versa (see “[Big-O Notation](#)” on page 92). But the advantage of coding this algorithm yourself is that with an understanding of these basic methods, you could use these building blocks to extend this to do some very interesting custom behaviors. The key to efficiently using Python in data-intensive applications is knowing about general convenience routines like `np.histogram` and when they're appropriate, but also knowing how to make use of lower-level functionality when you need more pointed behavior.

Sorting Arrays

Up to this point we have been concerned mainly with tools to access and operate on array data with NumPy. This section covers algorithms related to sorting values in NumPy arrays. These algorithms are a favorite topic in introductory computer science courses: if you've ever taken one, you probably have had dreams (or, depending on your temperament, nightmares) about *insertion sorts*, *selection sorts*, *merge sorts*, *quick sorts*, *bubble sorts*, and many, many more. All are means of accomplishing a similar task: sorting the values in a list or array.

For example, a simple *selection sort* repeatedly finds the minimum value from a list, and makes swaps until the list is sorted. We can code this in just a few lines of Python:

```
In[1]: import numpy as np

def selection_sort(x):
    for i in range(len(x)):
        swap = i + np.argmin(x[i:])
        (x[i], x[swap]) = (x[swap], x[i])
    return x

In[2]: x = np.array([2, 1, 4, 3, 5])
selection_sort(x)

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

As any first-year computer science major will tell you, the selection sort is useful for its simplicity, but is much too slow to be useful for larger arrays. For a list of N values, it requires N loops, each of which does on the order of $\sim N$ comparisons to find the swap value. In terms of the “big-O” notation often used to characterize these algorithms (see “[Big-O Notation](#)” on page 92), selection sort averages $\mathcal{O}[N^2]$: if you double the number of items in the list, the execution time will go up by about a factor of four.

Even selection sort, though, is much better than my all-time favorite sorting algorithms, the *bogosort*:

```
In[3]: def bogosort(x):
    while np.any(x[:-1] > x[1:]):
        np.random.shuffle(x)
    return x

In[4]: x = np.array([2, 1, 4, 3, 5])
bogosort(x)

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

This silly sorting method relies on pure chance: it repeatedly applies a random shuffling of the array until the result happens to be sorted. With an average scaling of $\mathcal{O}[N \times N!]$ (that’s N times N factorial), this should—quite obviously—never be used for any real computation.

Fortunately, Python contains built-in sorting algorithms that are *much* more efficient than either of the simplistic algorithms just shown. We’ll start by looking at the Python built-ins, and then take a look at the routines included in NumPy and optimized for NumPy arrays.

Fast Sorting in NumPy: `np.sort` and `np.argsort`

Although Python has built-in `sort` and `sorted` functions to work with lists, we won’t discuss them here because NumPy’s `np.sort` function turns out to be much more

efficient and useful for our purposes. By default `np.sort` uses an $\mathcal{O}[N \log N]$, *quicksort* algorithm, though *mergesort* and *heapsort* are also available. For most applications, the default quicksort is more than sufficient.

To return a sorted version of the array without modifying the input, you can use `np.sort`:

```
In[5]: x = np.array([2, 1, 4, 3, 5])
         np.sort(x)

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

If you prefer to sort the array in-place, you can instead use the `sort` method of arrays:

```
In[6]: x.sort()
         print(x)

[1 2 3 4 5]
```

A related function is `argsort`, which instead returns the *indices* of the sorted elements:

```
In[7]: x = np.array([2, 1, 4, 3, 5])
         i = np.argsort(x)
         print(i)

[1 0 3 2 4]
```

The first element of this result gives the index of the smallest element, the second value gives the index of the second smallest, and so on. These indices can then be used (via fancy indexing) to construct the sorted array if desired:

```
In[8]: x[i]
         Out[8]: array([1, 2, 3, 4, 5])
```

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the `axis` argument. For example:

```
In[9]: rand = np.random.RandomState(42)
         X = rand.randint(0, 10, (4, 6))
         print(X)

[[6 3 7 4 6 9]
 [2 6 7 4 3 7]
 [7 2 5 4 1 7]
 [5 1 4 0 9 5]]

In[10]: # sort each column of X
         np.sort(X, axis=0)

Out[10]: array([[2, 1, 4, 0, 1, 5],
                 [5, 2, 5, 4, 3, 7],
```

```
[6, 3, 7, 4, 6, 7],  
[7, 6, 7, 4, 9, 9]])  
  
In[11]: # sort each row of X  
np.sort(X, axis=1)  
  
Out[11]: array([[3, 4, 6, 6, 7, 9],  
                 [2, 3, 4, 6, 7, 7],  
                 [1, 2, 4, 5, 7, 7],  
                 [0, 1, 4, 5, 5, 9]])
```

Keep in mind that this treats each row or column as an independent array, and any relationships between the row or column values will be lost!

Partial Sorts: Partitioning

Sometimes we're not interested in sorting the entire array, but simply want to find the K smallest values in the array. NumPy provides this in the `np.partition` function. `np.partition` takes an array and a number K ; the result is a new array with the smallest K values to the left of the partition, and the remaining values to the right, in arbitrary order:

```
In[12]: x = np.array([7, 2, 3, 1, 6, 5, 4])  
np.partition(x, 3)  
  
Out[12]: array([2, 1, 3, 4, 6, 5, 7])
```

Note that the first three values in the resulting array are the three smallest in the array, and the remaining array positions contain the remaining values. Within the two partitions, the elements have arbitrary order.

Similarly to sorting, we can partition along an arbitrary axis of a multidimensional array:

```
In[13]: np.partition(X, 2, axis=1)  
  
Out[13]: array([[3, 4, 6, 7, 6, 9],  
                 [2, 3, 4, 7, 6, 7],  
                 [1, 2, 4, 5, 7, 7],  
                 [0, 1, 4, 5, 9, 5]])
```

The result is an array where the first two slots in each row contain the smallest values from that row, with the remaining values filling the remaining slots.

Finally, just as there is a `np.argsort` that computes indices of the sort, there is a `np.argpartition` that computes indices of the partition. We'll see this in action in the following section.

Example: k-Nearest Neighbors

Let's quickly see how we might use this `argsort` function along multiple axes to find the nearest neighbors of each point in a set. We'll start by creating a random set of 10

points on a two-dimensional plane. Using the standard convention, we'll arrange these in a 10×2 array:

```
In[14]: X = rand.rand(10, 2)
```

To get an idea of how these points look, let's quickly scatter plot them (Figure 2-10):

```
In[15]: %matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # Plot styling
plt.scatter(X[:, 0], X[:, 1], s=100);
```



Figure 2-10. Visualization of points in the k-neighbors example

Now we'll compute the distance between each pair of points. Recall that the squared-distance between two points is the sum of the squared differences in each dimension; using the efficient broadcasting (“Computation on Arrays: Broadcasting” on page 63) and aggregation (“Aggregations: Min, Max, and Everything in Between” on page 58) routines provided by NumPy, we can compute the matrix of square distances in a single line of code:

```
In[16]: dist_sq = np.sum((X[:,np.newaxis,:,:] - X[np.newaxis,:,:,:]) ** 2, axis=-1)
```

This operation has a lot packed into it, and it might be a bit confusing if you're unfamiliar with NumPy's broadcasting rules. When you come across code like this, it can be useful to break it down into its component steps:

```
In[17]: # for each pair of points, compute differences in their coordinates
differences = X[:, np.newaxis, :] - X[np.newaxis, :, :]
differences.shape
```

```
Out[17]: (10, 10, 2)
```

```
In[18]: # square the coordinate differences
sq_differences = differences ** 2
sq_differences.shape

Out[18]: (10, 10, 2)

In[19]: # sum the coordinate differences to get the squared distance
dist_sq = sq_differences.sum(-1)
dist_sq.shape

Out[19]: (10, 10)
```

Just to double-check what we are doing, we should see that the diagonal of this matrix (i.e., the set of distances between each point and itself) is all zero:

```
In[20]: dist_sq.diagonal()

Out[20]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

It checks out! With the pairwise square-distances converted, we can now use `np.argsort` to sort along each row. The leftmost columns will then give the indices of the nearest neighbors:

```
In[21]: nearest = np.argsort(dist_sq, axis=1)
print(nearest)

[[0 3 9 7 1 4 2 5 6 8]
 [1 4 7 9 3 6 8 5 0 2]
 [2 1 4 6 3 0 8 9 7 5]
 [3 9 7 0 1 4 5 8 6 2]
 [4 1 8 5 6 7 9 3 0 2]
 [5 8 6 4 1 7 9 3 2 0]
 [6 8 5 4 1 7 9 3 2 0]
 [7 9 3 1 4 0 5 8 6 2]
 [8 5 6 4 1 7 9 3 2 0]
 [9 7 3 0 1 4 5 8 6 2]]
```

Notice that the first column gives the numbers 0 through 9 in order: this is due to the fact that each point's closest neighbor is itself, as we would expect.

By using a full sort here, we've actually done more work than we need to in this case. If we're simply interested in the nearest k neighbors, all we need is to partition each row so that the smallest $k + 1$ squared distances come first, with larger distances filling the remaining positions of the array. We can do this with the `np.argpartition` function:

```
In[22]: K = 2
nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
```

In order to visualize this network of neighbors, let's quickly plot the points along with lines representing the connections from each point to its two nearest neighbors (Figure 2-11):

```
In[23]: plt.scatter(X[:, 0], X[:, 1], s=100)

# draw lines from each point to its two nearest neighbors
K = 2

for i in range(X.shape[0]):
    for j in nearest_partition[i, :K+1]:
        # plot a line from X[i] to X[j]
        # use some zip magic to make it happen:
        plt.plot(*zip(X[j], X[i]), color='black')
```

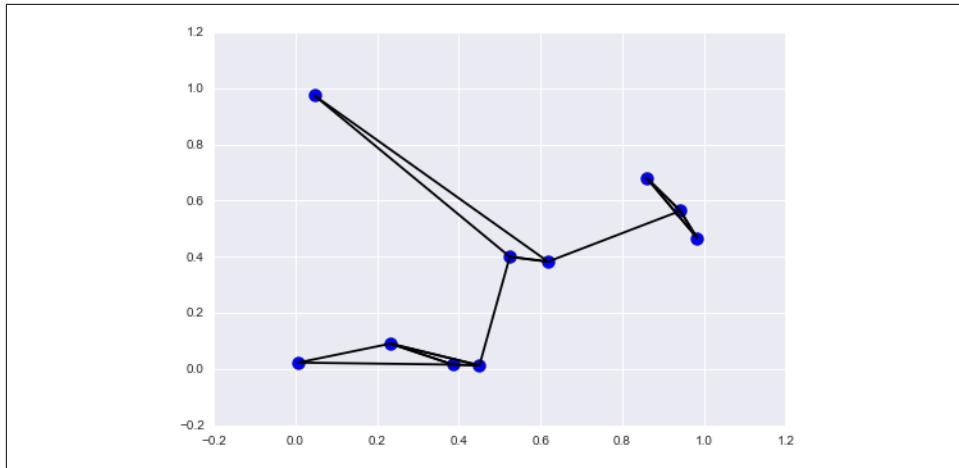


Figure 2-11. Visualization of the neighbors of each point

Each point in the plot has lines drawn to its two nearest neighbors. At first glance, it might seem strange that some of the points have more than two lines coming out of them: this is due to the fact that if point A is one of the two nearest neighbors of point B, this does not necessarily imply that point B is one of the two nearest neighbors of point A.

Although the broadcasting and row-wise sorting of this approach might seem less straightforward than writing a loop, it turns out to be a very efficient way of operating on this data in Python. You might be tempted to do the same type of operation by manually looping through the data and sorting each set of neighbors individually, but this would almost certainly lead to a slower algorithm than the vectorized version we used. The beauty of this approach is that it's written in a way that's agnostic to the size of the input data: we could just as easily compute the neighbors among 100 or 1,000,000 points in any number of dimensions, and the code would look the same.

Finally, I'll note that when doing very large nearest-neighbor searches, there are tree-based and/or approximate algorithms that can scale as $\mathcal{O}[N \log N]$ or better rather

than the $\mathcal{O}[N^2]$ of the brute-force algorithm. One example of this is the KD-Tree, [implemented in Scikit-Learn](#).

Big-O Notation

Big-O notation is a means of describing how the number of operations required for an algorithm scales as the input grows in size. To use it correctly is to dive deeply into the realm of computer science theory, and to carefully distinguish it from the related small-o notation, big- θ notation, big- Ω notation, and probably many mutant hybrids thereof. While these distinctions add precision to statements about algorithmic scaling, outside computer science theory exams and the remarks of pedantic blog commenters, you'll rarely see such distinctions made in practice. Far more common in the data science world is a less rigid use of big-O notation: as a general (if imprecise) description of the scaling of an algorithm. With apologies to theorists and pedants, this is the interpretation we'll use throughout this book.

Big-O notation, in this loose sense, tells you how much time your algorithm will take as you increase the amount of data. If you have an $\mathcal{O}[N]$ (read “order N ”) algorithm that takes 1 second to operate on a list of length $N=1,000$, then you should expect it to take roughly 5 seconds for a list of length $N=5,000$. If you have an $\mathcal{O}[N^2]$ (read “order N squared”) algorithm that takes 1 second for $N=1,000$, then you should expect it to take about 25 seconds for $N=5,000$.

For our purposes, the N will usually indicate some aspect of the size of the dataset (the number of points, the number of dimensions, etc.). When trying to analyze billions or trillions of samples, the difference between $\mathcal{O}[N]$ and $\mathcal{O}[N^2]$ can be far from trivial!

Notice that the big-O notation by itself tells you nothing about the actual wall-clock time of a computation, but only about its scaling as you change N . Generally, for example, an $\mathcal{O}[N]$ algorithm is considered to have better scaling than an $\mathcal{O}[N^2]$ algorithm, and for good reason. But for small datasets in particular, the algorithm with better scaling might not be faster. For example, in a given problem an $\mathcal{O}[N^2]$ algorithm might take 0.01 seconds, while a “better” $\mathcal{O}[N]$ algorithm might take 1 second. Scale up N by a factor of 1,000, though, and the $\mathcal{O}[N]$ algorithm will win out.

Even this loose version of Big-O notation can be very useful for comparing the performance of algorithms, and we'll use this notation throughout the book when talking about how algorithms scale.

Structured Data: NumPy's Structured Arrays

While often our data can be well represented by a homogeneous array of values, sometimes this is not the case. This section demonstrates the use of NumPy's *structured arrays* and *record arrays*, which provide efficient storage for compound, hetero-

geneous data. While the patterns shown here are useful for simple operations, scenarios like this often lend themselves to the use of Pandas DataFrames, which we'll explore in [Chapter 3](#).

Imagine that we have several categories of data on a number of people (say, name, age, and weight), and we'd like to store these values for use in a Python program. It would be possible to store these in three separate arrays:

```
In[2]: name = ['Alice', 'Bob', 'Cathy', 'Doug']
      age = [25, 45, 37, 19]
      weight = [55.0, 85.5, 68.0, 61.5]
```

But this is a bit clumsy. There's nothing here that tells us that the three arrays are related; it would be more natural if we could use a single structure to store all of this data. NumPy can handle this through structured arrays, which are arrays with compound data types.

Recall that previously we created a simple array using an expression like this:

```
In[3]: x = np.zeros(4, dtype=int)
```

We can similarly create a structured array using a compound data type specification:

```
In[4]: # Use a compound data type for structured arrays
      data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                                'formats':('U10', 'i4', 'f8')})
      print(data.dtype)
[(('name', 'U10'), ('age', 'i4'), ('weight', 'f8'))]
```

Here 'U10' translates to “Unicode string of maximum length 10,” 'i4' translates to “4-byte (i.e., 32 bit) integer,” and 'f8' translates to “8-byte (i.e., 64 bit) float.” We’ll discuss other options for these type codes in the following section.

Now that we’ve created an empty container array, we can fill the array with our lists of values:

```
In[5]: data['name'] = name
      data['age'] = age
      data['weight'] = weight
      print(data)
[('Alice', 25, 55.0) ('Bob', 45, 85.5) ('Cathy', 37, 68.0)
 ('Doug', 19, 61.5)]
```

As we had hoped, the data is now arranged together in one convenient block of memory.

The handy thing with structured arrays is that you can now refer to values either by index or by name:

```
In[6]: # Get all names
      data['name']
```

```
Out[6]: array(['Alice', 'Bob', 'Cathy', 'Doug'],
              dtype='<U10')

In[7]: # Get first row of data
        data[0]

Out[7]: ('Alice', 25, 55.0)

In[8]: # Get the name from the last row
        data[-1]['name']

Out[8]: 'Doug'
```

Using Boolean masking, this even allows you to do some more sophisticated operations such as filtering on age:

```
In[9]: # Get names where age is under 30
        data[data['age'] < 30]['name']

Out[9]: array(['Alice', 'Doug'],
              dtype='<U10')
```

Note that if you'd like to do any operations that are any more complicated than these, you should probably consider the Pandas package, covered in the next chapter. As we'll see, Pandas provides a `DataFrame` object, which is a structure built on NumPy arrays that offers a variety of useful data manipulation functionality similar to what we've shown here, as well as much, much more.

Creating Structured Arrays

Structured array data types can be specified in a number of ways. Earlier, we saw the `dictionary` method:

```
In[10]: np.dtype({'names':('name', 'age', 'weight'),
                  'formats':('U10', 'i4', 'f8')})

Out[10]: dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

For clarity, numerical types can be specified with Python types or NumPy `dtypes` instead:

```
In[11]: np.dtype({'names':('name', 'age', 'weight'),
                  'formats':((np.str_, 10), int, np.float32)})

Out[11]: dtype([('name', '<U10'), ('age', '<i8'), ('weight', '<f4')])
```

A compound type can also be specified as a list of tuples:

```
In[12]: np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])

Out[12]: dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

If the names of the types do not matter to you, you can specify the types alone in a comma-separated string:

```
In[13]: np.dtype('S10,i4,f8')
```

```
Out[13]: dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

The shortened string format codes may seem confusing, but they are built on simple principles. The first (optional) character is `<` or `>`, which means “little endian” or “big endian,” respectively, and specifies the ordering convention for significant bits. The next character specifies the type of data: characters, bytes, ints, floating points, and so on (see [Table 2-4](#)). The last character or characters represents the size of the object in bytes.

Table 2-4. NumPy data types

Character	Description	Example
'b'	Byte	<code>np.dtype('b')</code>
'i'	Signed integer	<code>np.dtype('i4') == np.int32</code>
'u'	Unsigned integer	<code>np.dtype('u1') == np.uint8</code>
'f'	Floating point	<code>np.dtype('f8') == np.int64</code>
'c'	Complex floating point	<code>np.dtype('c16') == np.complex128</code>
'S', 'a'	string	<code>np.dtype('S5')</code>
'U'	Unicode string	<code>np.dtype('U') == np.str_</code>
'V'	Raw data (void)	<code>np.dtype('V') == np.void</code>

More Advanced Compound Types

It is possible to define even more advanced compound types. For example, you can create a type where each element contains an array or matrix of values. Here, we’ll create a data type with a `mat` component consisting of a 3×3 floating-point matrix:

```
In[14]: tp = np.dtype([(id', 'i8'), ('mat', 'f8', (3, 3))])
X = np.zeros(1, dtype=tp)
print(X[0])
print(X['mat'][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.]]]
```

Now each element in the `X` array consists of an `id` and a 3×3 matrix. Why would you use this rather than a simple multidimensional array, or perhaps a Python dictionary? The reason is that this NumPy `dtype` directly maps onto a C structure definition, so the buffer containing the array content can be accessed directly within an appropriately written C program. If you find yourself writing a Python interface to a legacy C or Fortran library that manipulates structured data, you’ll probably find structured arrays quite useful!

RecordArrays: Structured Arrays with a Twist

NumPy also provides the `np.recarray` class, which is almost identical to the structured arrays just described, but with one additional feature: fields can be accessed as attributes rather than as dictionary keys. Recall that we previously accessed the ages by writing:

```
In[15]: data['age']  
Out[15]: array([25, 45, 37, 19], dtype=int32)
```

If we view our data as a record array instead, we can access this with slightly fewer keystrokes:

```
In[16]: data_rec = data.view(np.recarray)  
        data_rec.age  
Out[16]: array([25, 45, 37, 19], dtype=int32)
```

The downside is that for record arrays, there is some extra overhead involved in accessing the fields, even when using the same syntax. We can see this here:

```
In[17]: %timeit data['age']  
        %timeit data_rec['age']  
        %timeit data_rec.age  
  
1000000 loops, best of 3: 241 ns per loop  
100000 loops, best of 3: 4.61 µs per loop  
100000 loops, best of 3: 7.27 µs per loop
```

Whether the more convenient notation is worth the additional overhead will depend on your own application.

On to Pandas

This section on structured and record arrays is purposely at the end of this chapter, because it leads so well into the next package we will cover: Pandas. Structured arrays like the ones discussed here are good to know about for certain situations, especially in case you're using NumPy arrays to map onto binary data formats in C, Fortran, or another language. For day-to-day use of structured data, the Pandas package is a much better choice, and we'll dive into a full discussion of it in the next chapter.

Data Manipulation with Pandas

In the previous chapter, we dove into detail on NumPy and its `ndarray` object, which provides efficient storage and manipulation of dense typed arrays in Python. Here we'll build on this knowledge by looking in detail at the data structures provided by the Pandas library. Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a `DataFrame`. `DataFrames` are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data. As well as offering a convenient storage interface for labeled data, Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

As we saw, NumPy's `ndarray` data structure provides essential features for the type of clean, well-organized data typically seen in numerical computing tasks. While it serves this purpose very well, its limitations become clear when we need more flexibility (attaching labels to data, working with missing data, etc.) and when attempting operations that do not map well to element-wise broadcasting (groupings, pivots, etc.), each of which is an important piece of analyzing the less structured data available in many forms in the world around us. Pandas, and in particular its `Series` and `DataFrame` objects, builds on the NumPy array structure and provides efficient access to these sorts of “data munging” tasks that occupy much of a data scientist's time.

In this chapter, we will focus on the mechanics of using `Series`, `DataFrame`, and related structures effectively. We will use examples drawn from real datasets where appropriate, but these examples are not necessarily the focus.

Installing and Using Pandas

Installing Pandas on your system requires NumPy to be installed, and if you're building the library from source, requires the appropriate tools to compile the C and

Cython sources on which Pandas is built. Details on this installation can be found in [the Pandas documentation](#). If you followed the advice outlined in the preface and used the Anaconda stack, you already have Pandas installed.

Once Pandas is installed, you can import it and check the version:

```
In[1]: import pandas  
       pandas.__version__  
  
Out[1]: '0.18.1'
```

Just as we generally import NumPy under the alias np, we will import Pandas under the alias pd:

```
In[2]: import pandas as pd
```

This import convention will be used throughout the remainder of this book.

Reminder About Built-In Documentation

As you read through this chapter, don't forget that IPython gives you the ability to quickly explore the contents of a package (by using the tab-completion feature) as well as the documentation of various functions (using the ? character). (Refer back to [“Help and Documentation in IPython” on page 3](#) if you need a refresher on this.)

For example, to display all the contents of the pandas namespace, you can type this:

```
In [3]: pd.<TAB>
```

And to display the built-in Pandas documentation, you can use this:

```
In [4]: pd?
```

More detailed documentation, along with tutorials and other resources, can be found at <http://pandas.pydata.org/>.

Introducing Pandas Objects

At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices. As we will see during the course of this chapter, Pandas provides a host of useful tools, methods, and functionality on top of the basic data structures, but nearly everything that follows will require an understanding of what these structures are. Thus, before we go any further, let's introduce these three fundamental Pandas data structures: the Series, DataFrame, and Index.

We will start our code sessions with the standard NumPy and Pandas imports:

```
In[1]: import numpy as np  
       import pandas as pd
```

The Pandas Series Object

A Pandas `Series` is a one-dimensional array of indexed data. It can be created from a list or array as follows:

```
In[2]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
Out[2]: 0    0.25
         1    0.50
         2    0.75
         3    1.00
        dtype: float64
```

As we see in the preceding output, the `Series` wraps both a sequence of values and a sequence of indices, which we can access with the `values` and `index` attributes. The `values` are simply a familiar NumPy array:

```
In[3]: data.values
Out[3]: array([ 0.25,  0.5 ,  0.75,  1. ])
```

The `index` is an array-like object of type `pd.Index`, which we'll discuss in more detail momentarily:

```
In[4]: data.index
Out[4]: RangeIndex(start=0, stop=4, step=1)
```

Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket notation:

```
In[5]: data[1]
Out[5]: 0.5
In[6]: data[1:3]
Out[6]: 1    0.50
         2    0.75
        dtype: float64
```

As we will see, though, the Pandas `Series` is much more general and flexible than the one-dimensional NumPy array that it emulates.

Series as generalized NumPy array

From what we've seen so far, it may look like the `Series` object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the NumPy array has an *implicitly defined* integer index used to access the values, the Pandas `Series` has an *explicitly defined* index associated with the values.

This explicit index definition gives the `Series` object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

```
In[7]: data = pd.Series([0.25, 0.5, 0.75, 1.0],  
                      index=['a', 'b', 'c', 'd'])  
       data  
  
Out[7]: a    0.25  
        b    0.50  
        c    0.75  
        d    1.00  
       dtype: float64
```

And the item access works as expected:

```
In[8]: data['b']  
Out[8]: 0.5
```

We can even use noncontiguous or nonsequential indices:

```
In[9]: data = pd.Series([0.25, 0.5, 0.75, 1.0],  
                      index=[2, 5, 3, 7])  
       data  
  
Out[9]: 2    0.25  
        5    0.50  
        3    0.75  
        7    1.00  
       dtype: float64  
  
In[10]: data[5]  
Out[10]: 0.5
```

Series as specialized dictionary

In this way, you can think of a Pandas `Series` a bit like a specialization of a Python dictionary. A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a `Series` is a structure that maps typed keys to a set of typed values. This typing is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas `Series` makes it much more efficient than Python dictionaries for certain operations.

We can make the `Series`-as-dictionary analogy even more clear by constructing a `Series` object directly from a Python dictionary:

```
In[11]: population_dict = {'California': 38332521,
                           'Texas': 26448193,
                           'New York': 19651127,
                           'Florida': 19552860,
                           'Illinois': 12882135}
population = pd.Series(population_dict)
population
```

```
Out[11]: California    38332521
          Florida       19552860
          Illinois      12882135
          New York      19651127
          Texas         26448193
          dtype: int64
```

By default, a `Series` will be created where the index is drawn from the sorted keys. From here, typical dictionary-style item access can be performed:

```
In[12]: population['California']
Out[12]: 38332521
```

Unlike a dictionary, though, the `Series` also supports array-style operations such as slicing:

```
In[13]: population['California':'Illinois']
Out[13]: California    38332521
          Florida       19552860
          Illinois      12882135
          dtype: int64
```

We'll discuss some of the quirks of Pandas indexing and slicing in “[Data Indexing and Selection](#)” on page 107.

Constructing Series objects

We've already seen a few ways of constructing a Pandas `Series` from scratch; all of them are some version of the following:

```
>>> pd.Series(data, index=index)
```

where `index` is an optional argument, and `data` can be one of many entities.

For example, `data` can be a list or NumPy array, in which case `index` defaults to an integer sequence:

```
In[14]: pd.Series([2, 4, 6])
Out[14]: 0    2
          1    4
          2    6
          dtype: int64
```

`data` can be a scalar, which is repeated to fill the specified index:

```
In[15]: pd.Series(5, index=[100, 200, 300])  
Out[15]: 100    5  
         200    5  
         300    5  
        dtype: int64
```

`data` can be a dictionary, in which `index` defaults to the sorted dictionary keys:

```
In[16]: pd.Series({2:'a', 1:'b', 3:'c'})  
Out[16]: 1    b  
         2    a  
         3    c  
        dtype: object
```

In each case, the index can be explicitly set if a different result is preferred:

```
In[17]: pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])  
Out[17]: 3    c  
         2    a  
        dtype: object
```

Notice that in this case, the `Series` is populated only with the explicitly identified keys.

The Pandas DataFrame Object

The next fundamental structure in Pandas is the `DataFrame`. Like the `Series` object discussed in the previous section, the `DataFrame` can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary. We'll now take a look at each of these perspectives.

DataFrame as a generalized NumPy array

If a `Series` is an analog of a one-dimensional array with flexible indices, a `DataFrame` is an analog of a two-dimensional array with both flexible row indices and flexible column names. Just as you might think of a two-dimensional array as an ordered sequence of aligned one-dimensional columns, you can think of a `DataFrame` as a sequence of aligned `Series` objects. Here, by “aligned” we mean that they share the same index.

To demonstrate this, let's first construct a new `Series` listing the area of each of the five states discussed in the previous section:

```
In[18]:  
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,  
            'Florida': 170312, 'Illinois': 149995}
```

```
area = pd.Series(area_dict)
area

Out[18]: California    423967
          Florida      170312
          Illinois     149995
          New York     141297
          Texas        695662
          dtype: int64
```

Now that we have this along with the population Series from before, we can use a dictionary to construct a single two-dimensional object containing this information:

```
In[19]: states = pd.DataFrame({'population': population,
                               'area': area})
states

Out[19]:      area  population
California  423967   38332521
Florida     170312   19552860
Illinois    149995   12882135
New York    141297   19651127
Texas       695662   26448193
```

Like the Series object, the DataFrame has an `index` attribute that gives access to the index labels:

```
In[20]: states.index

Out[20]:
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'], dtype='object')
```

Additionally, the DataFrame has a `columns` attribute, which is an Index object holding the column labels:

```
In[21]: states.columns

Out[21]: Index(['area', 'population'], dtype='object')
```

Thus the DataFrame can be thought of as a generalization of a two-dimensional NumPy array, where both the rows and columns have a generalized index for accessing the data.

DataFrame as specialized dictionary

Similarly, we can also think of a DataFrame as a specialization of a dictionary. Where a dictionary maps a key to a value, a DataFrame maps a column name to a Series of column data. For example, asking for the 'area' attribute returns the Series object containing the areas we saw earlier:

```
In[22]: states['area']

Out[22]: California    423967
          Florida      170312
```

```
Illinois      149995
New York     141297
Texas        695662
Name: area, dtype: int64
```

Notice the potential point of confusion here: in a two-dimensional NumPy array, `data[0]` will return the first *row*. For a DataFrame, `data['col0']` will return the first *column*. Because of this, it is probably better to think about DataFrames as generalized dictionaries rather than generalized arrays, though both ways of looking at the situation can be useful. We'll explore more flexible means of indexing DataFrames in “[Data Indexing and Selection](#)” on page 107.

Constructing DataFrame objects

A Pandas DataFrame can be constructed in a variety of ways. Here we'll give several examples.

From a single Series object. A DataFrame is a collection of Series objects, and a single-column DataFrame can be constructed from a single Series:

```
In[23]: pd.DataFrame(population, columns=['population'])

Out[23]:           population
California    38332521
Florida       19552860
Illinois      12882135
New York      19651127
Texas         26448193
```

From a list of dicts. Any list of dictionaries can be made into a DataFrame. We'll use a simple list comprehension to create some data:

```
In[24]: data = [{'a': i, 'b': 2 * i}
               for i in range(3)]
pd.DataFrame(data)

Out[24]:   a  b
0  0  0
1  1  2
2  2  4
```

Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e., “not a number”) values:

```
In[25]: pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])

Out[25]:   a  b  c
0  1.0  2  NaN
1  NaN  3  4.0
```

From a dictionary of Series objects. As we saw before, a `DataFrame` can be constructed from a dictionary of `Series` objects as well:

```
In[26]: pd.DataFrame({'population': population,
                     'area': area})
```

```
Out[26]:      area    population
California   423967   38332521
Florida      170312   19552860
Illinois     149995   12882135
New York     141297   19651127
Texas        695662   26448193
```

From a two-dimensional NumPy array. Given a two-dimensional array of data, we can create a `DataFrame` with any specified column and index names. If omitted, an integer index will be used for each:

```
In[27]: pd.DataFrame(np.random.rand(3, 2),
                     columns=['foo', 'bar'],
                     index=['a', 'b', 'c'])
```

```
Out[27]:    foo      bar
a  0.865257  0.213169
b  0.442759  0.108267
c  0.047110  0.905718
```

From a NumPy structured array. We covered structured arrays in “[Structured Data: NumPy’s Structured Arrays](#)” on page 92. A Pandas `DataFrame` operates much like a structured array, and can be created directly from one:

```
In[28]: A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
A
```

```
Out[28]: array([(0, 0.0), (0, 0.0), (0, 0.0)],
               dtype=[('A', '<i8'), ('B', '<f8')])
```

```
In[29]: pd.DataFrame(A)
```

```
Out[29]:   A   B
0  0  0.0
1  0  0.0
2  0  0.0
```

The Pandas Index Object

We have seen here that both the `Series` and `DataFrame` objects contain an explicit `index` that lets you reference and modify data. This `Index` object is an interesting structure in itself, and it can be thought of either as an *immutable array* or as an *ordered set* (technically a multiset, as `Index` objects may contain repeated values). Those views have some interesting consequences in the operations available on `Index` objects. As a simple example, let’s construct an `Index` from a list of integers:

```
In[30]: ind = pd.Index([2, 3, 5, 7, 11])
ind
Out[30]: Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Index as immutable array

The Index object in many ways operates like an array. For example, we can use standard Python indexing notation to retrieve values or slices:

```
In[31]: ind[1]
Out[31]: 3
In[32]: ind[::-2]
Out[32]: Int64Index([2, 5, 11], dtype='int64')
```

Index objects also have many of the attributes familiar from NumPy arrays:

```
In[33]: print(ind.size, ind.shape, ind.ndim, ind.dtype)
5 (5,) 1 int64
```

One difference between Index objects and NumPy arrays is that indices are immutable—that is, they cannot be modified via the normal means:

```
In[34]: ind[1] = 0
-----
TypeError                                         Traceback (most recent call last)

<ipython-input-34-40e631c82e8a> in <module>()
----> 1 ind[1] = 0

/Users/jakevdp/anaconda/lib/python3.5/site-packages/pandas/indexes/base.py ...
 1243
 1244     def __setitem__(self, key, value):
-> 1245         raise TypeError("Index does not support mutable operations")
 1246
 1247     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

This immutability makes it safer to share indices between multiple `DataFrames` and arrays, without the potential for side effects from inadvertent index modification.

Index as ordered set

Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic. The Index object follows many of

the conventions used by Python’s built-in `set` data structure, so that unions, intersections, differences, and other combinations can be computed in a familiar way:

```
In[35]: indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])

In[36]: indA & indB # intersection
Out[36]: Int64Index([3, 5, 7], dtype='int64')

In[37]: indA | indB # union
Out[37]: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

In[38]: indA ^ indB # symmetric difference
Out[38]: Int64Index([1, 2, 9, 11], dtype='int64')
```

These operations may also be accessed via object methods—for example, `indA.intersection(indB)`.

Data Indexing and Selection

In [Chapter 2](#), we looked in detail at methods and tools to access, set, and modify values in NumPy arrays. These included indexing (e.g., `arr[2, 1]`), slicing (e.g., `arr[:, 1:5]`), masking (e.g., `arr[arr > 0]`), fancy indexing (e.g., `arr[0, [1, 5]]`), and combinations thereof (e.g., `arr[:, [1, 5]]`). Here we’ll look at similar means of accessing and modifying values in Pandas `Series` and `DataFrame` objects. If you have used the NumPy patterns, the corresponding patterns in Pandas will feel very familiar, though there are a few quirks to be aware of.

We’ll start with the simple case of the one-dimensional `Series` object, and then move on to the more complicated two-dimensional `DataFrame` object.

Data Selection in Series

As we saw in the previous section, a `Series` object acts in many ways like a one-dimensional NumPy array, and in many ways like a standard Python dictionary. If we keep these two overlapping analogies in mind, it will help us to understand the patterns of data indexing and selection in these arrays.

Series as dictionary

Like a dictionary, the `Series` object provides a mapping from a collection of keys to a collection of values:

```
In[1]: import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

```
Out[1]: a    0.25
        b    0.50
        c    0.75
        d    1.00
       dtype: float64

In[2]: data['b']

Out[2]: 0.5
```

We can also use dictionary-like Python expressions and methods to examine the keys/indices and values:

```
In[3]: 'a' in data

Out[3]: True

In[4]: data.keys()

Out[4]: Index(['a', 'b', 'c', 'd'], dtype='object')

In[5]: list(data.items())

Out[5]: [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

Series objects can even be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a Series by assigning to a new index value:

```
In[6]: data['e'] = 1.25
        data

Out[6]: a    0.25
        b    0.50
        c    0.75
        d    1.00
        e    1.25
       dtype: float64
```

This easy mutability of the objects is a convenient feature: under the hood, Pandas is making decisions about memory layout and data copying that might need to take place; the user generally does not need to worry about these issues.

Series as one-dimensional array

A Series builds on this dictionary-like interface and provides array-style item selection via the same basic mechanisms as NumPy arrays—that is, *slices*, *masking*, and *fancy indexing*. Examples of these are as follows:

```
In[7]: # slicing by explicit index
        data['a':'c']

Out[7]: a    0.25
        b    0.50
        c    0.75
       dtype: float64
```

```
In[8]: # slicing by implicit integer index
       data[0:2]

Out[8]: a    0.25
         b    0.50
         dtype: float64

In[9]: # masking
       data[(data > 0.3) & (data < 0.8)]

Out[9]: b    0.50
         c    0.75
         dtype: float64

In[10]: # fancy indexing
        data[['a', 'e']]

Out[10]: a    0.25
          e    1.25
          dtype: float64
```

Among these, slicing may be the source of the most confusion. Notice that when you are slicing with an explicit index (i.e., `data['a':'c']`), the final index is *included* in the slice, while when you're slicing with an implicit index (i.e., `data[0:2]`), the final index is *excluded* from the slice.

Indexers: loc, iloc, and ix

These slicing and indexing conventions can be a source of confusion. For example, if your `Series` has an explicit integer index, an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style index.

```
In[11]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
       data

Out[11]: 1    a
         3    b
         5    c
         dtype: object

In[12]: # explicit index when indexing
       data[1]

Out[12]: 'a'

In[13]: # implicit index when slicing
       data[1:3]

Out[13]: 3    b
         5    c
         dtype: object
```

Because of this potential confusion in the case of integer indexes, Pandas provides some special `indexer` attributes that explicitly expose certain indexing schemes. These

are not functional methods, but attributes that expose a particular slicing interface to the data in the `Series`.

First, the `loc` attribute allows indexing and slicing that always references the explicit index:

```
In[14]: data.loc[1]
Out[14]: 'a'

In[15]: data.loc[1:3]
Out[15]: 1    a
          3    b
          dtype: object
```

The `iloc` attribute allows indexing and slicing that always references the implicit Python-style index:

```
In[16]: data.iloc[1]
Out[16]: 'b'

In[17]: data.iloc[1:3]
Out[17]: 3    b
          5    c
          dtype: object
```

A third indexing attribute, `ix`, is a hybrid of the two, and for `Series` objects is equivalent to standard `[]`-based indexing. The purpose of the `ix` indexer will become more apparent in the context of `DataFrame` objects, which we will discuss in a moment.

One guiding principle of Python code is that “explicit is better than implicit.” The explicit nature of `loc` and `iloc` make them very useful in maintaining clean and readable code; especially in the case of integer indexes, I recommend using these both to make code easier to read and understand, and to prevent subtle bugs due to the mixed indexing/slicing convention.

Data Selection in DataFrame

Recall that a `DataFrame` acts in many ways like a two-dimensional or structured array, and in other ways like a dictionary of `Series` structures sharing the same index. These analogies can be helpful to keep in mind as we explore data selection within this structure.

DataFrame as a dictionary

The first analogy we will consider is the `DataFrame` as a dictionary of related `Series` objects. Let’s return to our example of areas and populations of states:

```
In[18]: area = pd.Series({'California': 423967, 'Texas': 695662,
                           'New York': 141297, 'Florida': 170312,
                           'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                  'New York': 19651127, 'Florida': 19552860,
                  'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```



```
Out[18]:      area    pop
California  423967  38332521
Florida     170312  19552860
Illinois    149995  12882135
New York    141297  19651127
Texas       695662  26448193
```

The individual `Series` that make up the columns of the `DataFrame` can be accessed via dictionary-style indexing of the column name:

```
In[19]: data['area']
```



```
Out[19]: California    423967
          Florida      170312
          Illinois     149995
          New York     141297
          Texas        695662
          Name: area, dtype: int64
```

Equivalently, we can use attribute-style access with column names that are strings:

```
In[20]: data.area
```



```
Out[20]: California    423967
          Florida      170312
          Illinois     149995
          New York     141297
          Texas        695662
          Name: area, dtype: int64
```

This attribute-style column access actually accesses the exact same object as the dictionary-style access:

```
In[21]: data.area is data['area']
```



```
Out[21]: True
```

Though this is a useful shorthand, keep in mind that it does not work for all cases! For example, if the column names are not strings, or if the column names conflict with methods of the `DataFrame`, this attribute-style access is not possible. For example, the `DataFrame` has a `pop()` method, so `data.pop` will point to this rather than the "pop" column:

```
In[22]: data.pop is data['pop']
```



```
Out[22]: False
```

In particular, you should avoid the temptation to try column assignment via attribute (i.e., use `data['pop'] = z` rather than `data.pop = z`).

Like with the `Series` objects discussed earlier, this dictionary-style syntax can also be used to modify the object, in this case to add a new column:

```
In[23]: data['density'] = data['pop'] / data['area']
       data
```

```
Out[23]:    area      pop      density
California  423967  38332521  90.413926
Florida     170312  19552860  114.806121
Illinois    149995  12882135  85.883763
New York    141297  19651127  139.076746
Texas       695662  26448193  38.018740
```

This shows a preview of the straightforward syntax of element-by-element arithmetic between `Series` objects; we'll dig into this further in “[Operating on Data in Pandas](#)” on page 115.

DataFrame as two-dimensional array

As mentioned previously, we can also view the `DataFrame` as an enhanced two-dimensional array. We can examine the raw underlying data array using the `values` attribute:

```
In[24]: data.values
```

```
Out[24]: array([[ 4.23967000e+05,   3.83325210e+07,   9.04139261e+01],
                 [ 1.70312000e+05,   1.95528600e+07,   1.14806121e+02],
                 [ 1.49995000e+05,   1.28821350e+07,   8.58837628e+01],
                 [ 1.41297000e+05,   1.96511270e+07,   1.39076746e+02],
                 [ 6.95662000e+05,   2.64481930e+07,   3.80187404e+01]])
```

With this picture in mind, we can do many familiar array-like observations on the `DataFrame` itself. For example, we can transpose the full `DataFrame` to swap rows and columns:

```
In[25]: data.T
```

```
Out[25]:    California    Florida    Illinois    New York    Texas
area        4.239670e+05  1.703120e+05  1.499950e+05  1.412970e+05  6.956620e+05
pop         3.833252e+07  1.955286e+07  1.288214e+07  1.965113e+07  2.644819e+07
density    9.041393e+01  1.148061e+02  8.588376e+01  1.390767e+02  3.801874e+01
```

When it comes to indexing of `DataFrame` objects, however, it is clear that the dictionary-style indexing of columns precludes our ability to simply treat it as a NumPy array. In particular, passing a single index to an array accesses a row:

```
In[26]: data.values[0]
```

```
Out[26]: array([ 4.23967000e+05,   3.83325210e+07,   9.04139261e+01])
```

and passing a single “index” to a `DataFrame` accesses a column:

```
In[27]: data['area']

Out[27]: California    423967
          Florida      170312
          Illinois     149995
          New York     141297
          Texas        695662
          Name: area, dtype: int64
```

Thus for array-style indexing, we need another convention. Here Pandas again uses the `loc`, `iloc`, and `ix` indexers mentioned earlier. Using the `iloc` indexer, we can index the underlying array as if it is a simple NumPy array (using the implicit Python-style index), but the `DataFrame` index and column labels are maintained in the result:

```
In[28]: data.iloc[:3, :2]

Out[28]:      area    pop
California  423967  38332521
Florida     170312  19552860
Illinois    149995  12882135

In[29]: data.loc[:, 'Illinois', : 'pop']

Out[29]:      area    pop
California  423967  38332521
Florida     170312  19552860
Illinois    149995  12882135
```

The `ix` indexer allows a hybrid of these two approaches:

```
In[30]: data.ix[:3, : 'pop']

Out[30]:      area    pop
California  423967  38332521
Florida     170312  19552860
Illinois    149995  12882135
```

Keep in mind that for integer indices, the `ix` indexer is subject to the same potential sources of confusion as discussed for integer-indexed `Series` objects.

Any of the familiar NumPy-style data access patterns can be used within these indexers. For example, in the `loc` indexer we can combine masking and fancy indexing as in the following:

```
In[31]: data.loc[data.density > 100, [ 'pop', 'density']]

Out[31]:      pop    density
Florida  19552860  114.806121
New York 19651127  139.076746
```

Any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be accustomed to from working with NumPy:

```
In[32]: data.iloc[0, 2] = 90
data

Out[32]:      area    pop    density
California  423967  38332521  90.000000
Florida     170312  19552860  114.806121
Illinois    149995  12882135  85.883763
New York   141297  19651127  139.076746
Texas       695662  26448193  38.018740
```

To build up your fluency in Pandas data manipulation, I suggest spending some time with a simple `DataFrame` and exploring the types of indexing, slicing, masking, and fancy indexing that are allowed by these various indexing approaches.

Additional indexing conventions

There are a couple extra indexing conventions that might seem at odds with the preceding discussion, but nevertheless can be very useful in practice. First, while *indexing* refers to columns, *slicing* refers to rows:

```
In[33]: data['Florida':'Illinois']

Out[33]:      area    pop    density
Florida     170312  19552860  114.806121
Illinois    149995  12882135  85.883763
```

Such slices can also refer to rows by number rather than by index:

```
In[34]: data[1:3]

Out[34]:      area    pop    density
Florida     170312  19552860  114.806121
Illinois    149995  12882135  85.883763
```

Similarly, direct masking operations are also interpreted row-wise rather than column-wise:

```
In[35]: data[data.density > 100]

Out[35]:      area    pop    density
Florida     170312  19552860  114.806121
New York   141297  19651127  139.076746
```

These two conventions are syntactically similar to those on a NumPy array, and while these may not precisely fit the mold of the Pandas conventions, they are nevertheless quite useful in practice.

Operating on Data in Pandas

One of the essential pieces of NumPy is the ability to perform quick element-wise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more sophisticated operations (trigonometric functions, exponential and logarithmic functions, etc.). Pandas inherits much of this functionality from NumPy, and the ufuncs that we introduced in “[Computation on NumPy Arrays: Universal Functions](#)” on page 50 are key to this.

Pandas includes a couple useful twists, however: for unary operations like negation and trigonometric functions, these ufuncs will *preserve index and column labels* in the output, and for binary operations such as addition and multiplication, Pandas will automatically *align indices* when passing the objects to the ufunc. This means that keeping the context of data and combining data from different sources—both potentially error-prone tasks with raw NumPy arrays—become essentially foolproof ones with Pandas. We will additionally see that there are well-defined operations between one-dimensional Series structures and two-dimensional DataFrame structures.

Ufuncs: Index Preservation

Because Pandas is designed to work with NumPy, any NumPy ufunc will work on Pandas Series and DataFrame objects. Let’s start by defining a simple Series and DataFrame on which to demonstrate this:

```
In[1]: import pandas as pd
        import numpy as np

In[2]: rng = np.random.RandomState(42)
        ser = pd.Series(rng.randint(0, 10, 4))
        ser

Out[2]: 0    6
        1    3
        2    7
        3    4
        dtype: int64

In[3]: df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                        columns=['A', 'B', 'C', 'D'])
        df

Out[3]:   A  B  C  D
0   6  9  2  6
1   7  4  3  7
2   7  2  5  4
```

If we apply a NumPy ufunc on either of these objects, the result will be another Pandas object *with the indices preserved*:

```
In[4]: np.exp(ser)
```

```
Out[4]: 0      403.428793
       1      20.085537
       2     1096.633158
       3      54.598150
      dtype: float64
```

Or, for a slightly more complex calculation:

```
In[5]: np.sin(df * np.pi / 4)

Out[5]:          A           B           C           D
0 -1.000000  7.071068e-01  1.000000 -1.000000e+00
1 -0.707107  1.224647e-16  0.707107 -7.071068e-01
2 -0.707107  1.000000e+00 -0.707107  1.224647e-16
```

Any of the ufuncs discussed in “Computation on NumPy Arrays: Universal Functions” on page 50 can be used in a similar manner.

UFuncs: Index Alignment

For binary operations on two `Series` or `DataFrame` objects, Pandas will align indices in the process of performing the operation. This is very convenient when you are working with incomplete data, as we’ll see in some of the examples that follow.

Index alignment in Series

As an example, suppose we are combining two different data sources, and find only the top three US states by *area* and the top three US states by *population*:

```
In[6]: area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                       'California': 423967}, name='area')
population = pd.Series({'California': 38332521, 'Texas': 26448193,
                       'New York': 19651127}, name='population')
```

Let’s see what happens when we divide these to compute the population density:

```
In[7]: population / area

Out[7]: Alaska        NaN
         California   90.413926
         New York      NaN
         Texas        38.018740
        dtype: float64
```

The resulting array contains the *union* of indices of the two input arrays, which we could determine using standard Python set arithmetic on these indices:

```
In[8]: area.index | population.index

Out[8]: Index(['Alaska', 'California', 'New York', 'Texas'], dtype='object')
```

Any item for which one or the other does not have an entry is marked with `NaN`, or “Not a Number,” which is how Pandas marks missing data (see further discussion of missing data in “Handling Missing Data” on page 119). This index matching is imple-

mented this way for any of Python's built-in arithmetic expressions; any missing values are filled in with NaN by default:

```
In[9]: A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])
A + B

Out[9]: 0    NaN
         1    5.0
         2    9.0
         3    NaN
dtype: float64
```

If using NaN values is not the desired behavior, we can modify the fill value using appropriate object methods in place of the operators. For example, calling `A.add(B)` is equivalent to calling `A + B`, but allows optional explicit specification of the fill value for any elements in A or B that might be missing:

```
In[10]: A.add(B, fill_value=0)

Out[10]: 0    2.0
          1    5.0
          2    9.0
          3    5.0
dtype: float64
```

Index alignment in DataFrame

A similar type of alignment takes place for *both* columns and indices when you are performing operations on `DataFrames`:

```
In[11]: A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
                        columns=list('AB'))
A

Out[11]:   A   B
0   1  11
1   5   1

In[12]: B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                        columns=list('BAC'))
B

Out[12]:   B   A   C
0   4   0   9
1   5   8   0
2   9   2   6

In[13]: A + B

Out[13]:      A      B      C
0  1.0  15.0  NaN
1 13.0   6.0  NaN
2  NaN   NaN  NaN
```

Notice that indices are aligned correctly irrespective of their order in the two objects, and indices in the result are sorted. As was the case with `Series`, we can use the associated object's arithmetic method and pass any desired `fill_value` to be used in place of missing entries. Here we'll fill with the mean of all values in `A` (which we compute by first stacking the rows of `A`):

```
In[14]: fill = A.stack().mean()
A.add(B, fill_value=fill)

Out[14]:      A      B      C
0    1.0   15.0  13.5
1   13.0    6.0   4.5
2    6.5   13.5  10.5
```

Table 3-1 lists Python operators and their equivalent Pandas object methods.

Table 3-1. Mapping between Python operators and Pandas methods

Python operator	Pandas method(s)
+	<code>add()</code>
-	<code>sub()</code> , <code>subtract()</code>
*	<code>mul()</code> , <code>multiply()</code>
/	<code>truediv()</code> , <code>div()</code> , <code>divide()</code>
//	<code>floordiv()</code>
%	<code>mod()</code>
**	<code>pow()</code>

Ufuncs: Operations Between DataFrame and Series

When you are performing operations between a `DataFrame` and a `Series`, the index and column alignment is similarly maintained. Operations between a `DataFrame` and a `Series` are similar to operations between a two-dimensional and one-dimensional NumPy array. Consider one common operation, where we find the difference of a two-dimensional array and one of its rows:

```
In[15]: A = rng.randint(10, size=(3, 4))
A

Out[15]: array([[3, 8, 2, 4],
 [2, 6, 4, 8],
 [6, 1, 3, 8]])

In[16]: A - A[0]

Out[16]: array([[ 0,  0,  0,  0],
 [-1, -2,  2,  4],
 [ 3, -7,  1,  4]])
```

According to NumPy’s broadcasting rules (see “[Computation on Arrays: Broadcasting](#)” on page 63), subtraction between a two-dimensional array and one of its rows is applied row-wise.

In Pandas, the convention similarly operates row-wise by default:

```
In[17]: df = pd.DataFrame(A, columns=list('QRST'))
df - df.iloc[0]

Out[17]:   Q  R  S  T
0  0  0  0  0
1 -1 -2  2  4
2  3 -7  1  4
```

If you would instead like to operate column-wise, you can use the object methods mentioned earlier, while specifying the `axis` keyword:

```
In[18]: df.subtract(df['R'], axis=0)

Out[18]:   Q  R  S  T
0 -5  0 -6 -4
1 -4  0 -2  2
2  5  0  2  7
```

Note that these `DataFrame/Series` operations, like the operations discussed before, will automatically align indices between the two elements:

```
In[19]: halfrow = df.iloc[0, ::2]
halfrow

Out[19]: Q    3
          S    2
          Name: 0, dtype: int64

In[20]: df - halfrow

Out[20]:      Q   R     S   T
0  0.0  0.0  NaN  0.0  NaN
1 -1.0  NaN  2.0  NaN
2  3.0  NaN  1.0  NaN
```

This preservation and alignment of indices and columns means that operations on data in Pandas will always maintain the data context, which prevents the types of silly errors that might come up when you are working with heterogeneous and/or misaligned data in raw NumPy arrays.

Handling Missing Data

The difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous. In particular, many interesting datasets will have some amount of data missing. To make matters even more complicated, different data sources may indicate missing data in different ways.

In this section, we will discuss some general considerations for missing data, discuss how Pandas chooses to represent it, and demonstrate some built-in Pandas tools for handling missing data in Python. Here and throughout the book, we'll refer to missing data in general as *null*, *NAN*, or *NA* values.

Trade-Offs in Missing Data Conventions

A number of schemes have been developed to indicate the presence of missing data in a table or `DataFrame`. Generally, they revolve around one of two strategies: using a *mask* that globally indicates missing values, or choosing a *sentinel value* that indicates a missing entry.

In the masking approach, the mask might be an entirely separate Boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with `-9999` or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with `NaN` (Not a Number), a special value which is part of the IEEE floating-point specification.

None of these approaches is without trade-offs: use of a separate mask array requires allocation of an additional Boolean array, which adds overhead in both storage and computation. A sentinel value reduces the range of valid values that can be represented, and may require extra (often non-optimized) logic in CPU and GPU arithmetic. Common special values like `NaN` are not available for all data types.

As in most cases where no universally optimal choice exists, different languages and systems use different conventions. For example, the R language uses reserved bit patterns within each data type as sentinel values indicating missing data, while the SciDB system uses an extra byte attached to every cell to indicate a NA state.

Missing Data in Pandas

The way in which Pandas handles missing values is constrained by its reliance on the NumPy package, which does not have a built-in notion of NA values for non-floating-point data types.

Pandas could have followed R's lead in specifying bit patterns for each individual data type to indicate nullness, but this approach turns out to be rather unwieldy. While R contains four basic data types, NumPy supports *far* more than this: for example, while R has a single integer type, NumPy supports *fourteen* basic integer types once you account for available precisions, signedness, and endianness of the encoding. Reserving a specific bit pattern in all available NumPy types would lead to an unwieldy amount of overhead in special-casing various operations for various types,

likely even requiring a new fork of the NumPy package. Further, for the smaller data types (such as 8-bit integers), sacrificing a bit to use as a mask will significantly reduce the range of values it can represent.

NumPy does have support for masked arrays—that is, arrays that have a separate Boolean mask array attached for marking data as “good” or “bad.” Pandas could have derived from this, but the overhead in both storage, computation, and code maintenance makes that an unattractive choice.

With these constraints in mind, Pandas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floating-point `NaN` value, and the Python `None` object. This choice has some side effects, as we will see, but in practice ends up being a good compromise in most cases of interest.

None: Pythonic missing data

The first sentinel value used by Pandas is `None`, a Python singleton object that is often used for missing data in Python code. Because `None` is a Python object, it cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type ‘`object`’ (i.e., arrays of Python objects):

```
In[1]: import numpy as np  
       import pandas as pd  
  
In[2]: vals1 = np.array([1, None, 3, 4])  
       vals1  
  
Out[2]: array([1, None, 3, 4], dtype=object)
```

This `dtype=object` means that the best common type representation NumPy could infer for the contents of the array is that they are Python objects. While this kind of object array is useful for some purposes, any operations on the data will be done at the Python level, with much more overhead than the typically fast operations seen for arrays with native types:

```
In[3]: for dtype in ['object', 'int']:  
        print("dtype =", dtype)  
        %timeit np.arange(1E6, dtype=dtype).sum()  
        print()  
  
dtype = object  
10 loops, best of 3: 78.2 ms per loop  
  
dtype = int  
100 loops, best of 3: 3.06 ms per loop
```

The use of Python objects in an array also means that if you perform aggregations like `sum()` or `min()` across an array with a `None` value, you will generally get an error:

```
In[4]: vals1.sum()
TypeError                                 Traceback (most recent call last)
<ipython-input-4-749fd8ae6030> in <module>()
      1 vals1.sum()

/Users/jakevdp/anaconda/lib/python3.5/site-packages/numpy/core/_methods.py ...
    30
    31 def _sum(a, axis=None, dtype=None, out=None, keepdims=False):
--> 32     return umr_sum(a, axis, dtype, out, keepdims)
    33
    34 def _prod(a, axis=None, dtype=None, out=None, keepdims=False):

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

This reflects the fact that addition between an integer and None is undefined.

NaN: Missing numerical data

The other missing data representation, NaN (acronym for *Not a Number*), is different; it is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation:

```
In[5]: vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype

Out[5]: dtype('float64')
```

Notice that NumPy chose a native floating-point type for this array: this means that unlike the object array from before, this array supports fast operations pushed into compiled code. You should be aware that NaN is a bit like a data virus—it infects any other object it touches. Regardless of the operation, the result of arithmetic with NaN will be another NaN:

```
In[6]: 1 + np.nan

Out[6]: nan

In[7]: 0 * np.nan

Out[7]: nan
```

Note that this means that aggregates over the values are well defined (i.e., they don't result in an error) but not always useful:

```
In[8]: vals2.sum(), vals2.min(), vals2.max()

Out[8]: (nan, nan, nan)
```

NumPy does provide some special aggregations that will ignore these missing values:

```
In[9]: np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
Out[9]: (8.0, 1.0, 4.0)
```

Keep in mind that `NaN` is specifically a floating-point value; there is no equivalent `NaN` value for integers, strings, or other types.

NaN and None in Pandas

`NaN` and `None` both have their place, and Pandas is built to handle the two of them nearly interchangeably, converting between them where appropriate:

```
In[10]: pd.Series([1, np.nan, 2, None])
Out[10]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
dtype: float64
```

For types that don't have an available sentinel value, Pandas automatically type-casts when NA values are present. For example, if we set a value in an integer array to `np.nan`, it will automatically be upcast to a floating-point type to accommodate the NA:

```
In[11]: x = pd.Series(range(2), dtype=int)
          x
Out[11]: 0    0
         1    1
dtype: int64
In[12]: x[0] = None
          x
Out[12]: 0    NaN
         1    1.0
dtype: float64
```

Notice that in addition to casting the integer array to floating point, Pandas automatically converts the `None` to a `NaN` value. (Be aware that there is a proposal to add a native integer NA to Pandas in the future; as of this writing, it has not been included.)

While this type of magic may feel a bit hackish compared to the more unified approach to NA values in domain-specific languages like R, the Pandas sentinel/casting approach works quite well in practice and in my experience only rarely causes issues.

Table 3-2 lists the upcasting conventions in Pandas when NA values are introduced.

Table 3-2. Pandas handling of NAs by type

Typeclass	Conversion when storing NAs	NA sentinel value
floating	No change	np.nan
object	No change	None or np.nan
integer	Cast to float64	np.nan
boolean	Cast to object	None or np.nan

Keep in mind that in Pandas, string data is always stored with an `object` dtype.

Operating on Null Values

As we have seen, Pandas treats `None` and `NaN` as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:

`isnull()`

Generate a Boolean mask indicating missing values

`notnull()`

Opposite of `isnull()`

`dropna()`

Return a filtered version of the data

`fillna()`

Return a copy of the data with missing values filled or imputed

We will conclude this section with a brief exploration and demonstration of these routines.

Detecting null values

Pandas data structures have two useful methods for detecting null data: `isnull()` and `notnull()`. Either one will return a Boolean mask over the data. For example:

```
In[13]: data = pd.Series([1, np.nan, 'hello', None])  
In[14]: data.isnull()  
Out[14]: 0    False  
        1    True  
        2    False  
        3    True  
       dtype: bool
```

As mentioned in “[Data Indexing and Selection](#)” on page 107, Boolean masks can be used directly as a `Series` or `DataFrame` index:

```
In[15]: data[data.notnull()]  
Out[15]: 0      1  
         2    hello  
        dtype: object
```

The `isnull()` and `notnull()` methods produce similar Boolean results for DataFrames.

Dropping null values

In addition to the masking used before, there are the convenience methods, `dropna()` (which removes NA values) and `fillna()` (which fills in NA values). For a Series, the result is straightforward:

```
In[16]: data.dropna()  
Out[16]: 0      1  
         2    hello  
        dtype: object
```

For a DataFrame, there are more options. Consider the following DataFrame:

```
In[17]: df = pd.DataFrame([[1, np.nan, 2],  
                           [2, 3, 5],  
                           [np.nan, 4, 6]])  
df  
Out[17]:   0  1  2  
0  1.0  NaN  2  
1  2.0  3.0  5  
2  NaN  4.0  6
```

We cannot drop single values from a DataFrame; we can only drop full rows or full columns. Depending on the application, you might want one or the other, so `dropna()` gives a number of options for a DataFrame.

By default, `dropna()` will drop all rows in which *any* null value is present:

```
In[18]: df.dropna()  
Out[18]:   0  1  2  
1  2.0  3.0  5
```

Alternatively, you can drop NA values along a different axis; `axis=1` drops all columns containing a null value:

```
In[19]: df.dropna(axis='columns')  
Out[19]: 2  
0 2  
1 5  
2 6
```

But this drops some good data as well; you might rather be interested in dropping rows or columns with *all* NA values, or a majority of NA values. This can be specified through the `how` or `thresh` parameters, which allow fine control of the number of nulls to allow through.

The default is `how='any'`, such that any row or column (depending on the `axis` keyword) containing a null value will be dropped. You can also specify `how='all'`, which will only drop rows/columns that are *all* null values:

```
In[20]: df[3] = np.nan  
df  
  
Out[20]:      0    1    2    3  
0  1.0  NaN  2  NaN  
1  2.0  3.0  5  NaN  
2  NaN  4.0  6  NaN  
  
In[21]: df.dropna(axis='columns', how='all')  
  
Out[21]:      0    1    2  
0  1.0  NaN  2  
1  2.0  3.0  5  
2  NaN  4.0  6
```

For finer-grained control, the `thresh` parameter lets you specify a minimum number of non-null values for the row/column to be kept:

```
In[22]: df.dropna(axis='rows', thresh=3)  
  
Out[22]:      0    1    2    3  
1  2.0  3.0  5  NaN
```

Here the first and last row have been dropped, because they contain only two non-null values.

Filling null values

Sometimes rather than dropping NA values, you'd rather replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. You could do this in-place using the `isnull()` method as a mask, but because it is such a common operation Pandas provides the `fillna()` method, which returns a copy of the array with the null values replaced.

Consider the following Series:

```
In[23]: data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))  
data  
  
Out[23]: a    1.0  
         b    NaN  
         c    2.0  
         d    NaN
```

```
e    3.0  
dtype: float64
```

We can fill NA entries with a single value, such as zero:

```
In[24]: data.fillna(0)  
  
Out[24]: a    1.0  
         b    0.0  
         c    2.0  
         d    0.0  
         e    3.0  
         dtype: float64
```

We can specify a forward-fill to propagate the previous value forward:

```
In[25]: # forward-fill  
        data.fillna(method='ffill')  
  
Out[25]: a    1.0  
         b    1.0  
         c    2.0  
         d    2.0  
         e    3.0  
         dtype: float64
```

Or we can specify a back-fill to propagate the next values backward:

```
In[26]: # back-fill  
        data.fillna(method='bfill')  
  
Out[26]: a    1.0  
         b    2.0  
         c    2.0  
         d    3.0  
         e    3.0  
         dtype: float64
```

For DataFrames, the options are similar, but we can also specify an `axis` along which the fills take place:

```
In[27]: df  
  
Out[27]:      0    1    2    3  
0  1.0  NaN  2  NaN  
1  2.0  3.0  5  NaN  
2  NaN  4.0  6  NaN  
  
In[28]: df.fillna(method='ffill', axis=1)  
  
Out[28]:      0    1    2    3  
0  1.0  1.0  2.0  2.0  
1  2.0  3.0  5.0  5.0  
2  NaN  4.0  6.0  6.0
```

Notice that if a previous value is not available during a forward fill, the NA value remains.

Hierarchical Indexing

Up to this point we've been focused primarily on one-dimensional and two-dimensional data, stored in Pandas `Series` and `DataFrame` objects, respectively. Often it is useful to go beyond this and store higher-dimensional data—that is, data indexed by more than one or two keys. While Pandas does provide `Panel` and `Panel4D` objects that natively handle three-dimensional and four-dimensional data (see “[Panel Data](#)” on page 141), a far more common pattern in practice is to make use of *hierarchical indexing* (also known as *multi-indexing*) to incorporate multiple index *levels* within a single index. In this way, higher-dimensional data can be compactly represented within the familiar one-dimensional `Series` and two-dimensional `DataFrame` objects.

In this section, we'll explore the direct creation of `MultiIndex` objects; considerations around indexing, slicing, and computing statistics across multiply indexed data; and useful routines for converting between simple and hierarchically indexed representations of your data.

We begin with the standard imports:

```
In[1]: import pandas as pd  
       import numpy as np
```

A Multiply Indexed Series

Let's start by considering how we might represent two-dimensional data within a one-dimensional `Series`. For concreteness, we will consider a series of data where each point has a character and numerical key.

The bad way

Suppose you would like to track data about states from two different years. Using the Pandas tools we've already covered, you might be tempted to simply use Python tuples as keys:

```
In[2]: index = [('California', 2000), ('California', 2010),  
              ('New York', 2000), ('New York', 2010),  
              ('Texas', 2000), ('Texas', 2010)]  
populations = [33871648, 37253956,  
                18976457, 19378102,  
                20851820, 25145561]  
pop = pd.Series(populations, index=index)  
pop
```



```
Out[2]: (California, 2000)    33871648  
        (California, 2010)    37253956  
        (New York, 2000)     18976457  
        (New York, 2010)     19378102  
        (Texas, 2000)        20851820
```

```
(Texas, 2010)      25145561  
dtype: int64
```

With this indexing scheme, you can straightforwardly index or slice the series based on this multiple index:

```
In[3]: pop[('California', 2010):('Texas', 2000)]  
  
Out[3]: (California, 2010)      37253956  
          (New York, 2000)      18976457  
          (New York, 2010)      19378102  
          (Texas, 2000)        20851820  
dtype: int64
```

But the convenience ends there. For example, if you need to select all values from 2010, you'll need to do some messy (and potentially slow) munging to make it happen:

```
In[4]: pop[[i for i in pop.index if i[1] == 2010]]  
  
Out[4]: (California, 2010)      37253956  
          (New York, 2010)      19378102  
          (Texas, 2010)        25145561  
dtype: int64
```

This produces the desired result, but is not as clean (or as efficient for large datasets) as the slicing syntax we've grown to love in Pandas.

The better way: Pandas MultiIndex

Fortunately, Pandas provides a better way. Our tuple-based indexing is essentially a rudimentary multi-index, and the Pandas `MultiIndex` type gives us the type of operations we wish to have. We can create a multi-index from the tuples as follows:

```
In[5]: index = pd.MultiIndex.from_tuples(index)  
index  
  
Out[5]: MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],  
                   labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

Notice that the `MultiIndex` contains multiple *levels* of indexing—in this case, the state names and the years, as well as multiple *labels* for each data point which encode these levels.

If we reindex our series with this `MultiIndex`, we see the hierarchical representation of the data:

```
In[6]: pop = pop.reindex(index)  
pop  
  
Out[6]: California  2000      33871648  
                  2010      37253956  
New York       2000      18976457  
                  2010      19378102
```

```
Texas      2000    20851820  
          2010    25145561  
dtype: int64
```

Here the first two columns of the `Series` representation show the multiple index values, while the third column shows the data. Notice that some entries are missing in the first column: in this multi-index representation, any blank entry indicates the same value as the line above it.

Now to access all data for which the second index is 2010, we can simply use the Pandas slicing notation:

```
In[7]: pop[:, 2010]  
Out[7]: California    37253956  
        New York     19378102  
        Texas       25145561  
dtype: int64
```

The result is a singly indexed array with just the keys we're interested in. This syntax is much more convenient (and the operation is much more efficient!) than the home-spun tuple-based multi-indexing solution that we started with. We'll now further discuss this sort of indexing operation on hierarchically indexed data.

MultIndex as extra dimension

You might notice something else here: we could easily have stored the same data using a simple `DataFrame` with index and column labels. In fact, Pandas is built with this equivalence in mind. The `unstack()` method will quickly convert a multiply-indexed `Series` into a conventionally indexed `DataFrame`:

```
In[8]: pop_df = pop.unstack()  
pop_df  
Out[8]:           2000      2010  
California  33871648  37253956  
New York    18976457  19378102  
Texas       20851820  25145561
```

Naturally, the `stack()` method provides the opposite operation:

```
In[9]: pop_df.stack()  
Out[9]:  California  2000    33871648  
                  2010    37253956  
        New York   2000    18976457  
                  2010    19378102  
        Texas     2000    20851820  
                  2010    25145561  
dtype: int64
```

Seeing this, you might wonder why would we would bother with hierarchical indexing at all. The reason is simple: just as we were able to use multi-indexing to represent

two-dimensional data within a one-dimensional `Series`, we can also use it to represent data of three or more dimensions in a `Series` or `DataFrame`. Each extra level in a multi-index represents an extra dimension of data; taking advantage of this property gives us much more flexibility in the types of data we can represent. Concretely, we might want to add another column of demographic data for each state at each year (say, population under 18); with a `MultiIndex` this is as easy as adding another column to the `DataFrame`:

```
In[10]: pop_df = pd.DataFrame({'total': pop,
                               'under18': [9267089, 9284094,
                                           4687374, 4318033,
                                           5906301, 6879014]})

pop_df
```

```
Out[10]:          total  under18
California  2000    33871648   9267089
              2010    37253956   9284094
New York    2000    18976457   4687374
              2010    19378102   4318033
Texas       2000    20851820   5906301
              2010    25145561   6879014
```

In addition, all the ufuncs and other functionality discussed in “[Operating on Data in Pandas](#)” on page 115 work with hierarchical indices as well. Here we compute the fraction of people under 18 by year, given the above data:

```
In[11]: f_u18 = pop_df['under18'] / pop_df['total']
f_u18.unstack()
```

```
Out[11]:        2000      2010
California  0.273594  0.249211
New York    0.247010  0.222831
Texas       0.283251  0.273568
```

This allows us to easily and quickly manipulate and explore even high-dimensional data.

Methods of MultiIndex Creation

The most straightforward way to construct a multiply indexed `Series` or `DataFrame` is to simply pass a list of two or more index arrays to the constructor. For example:

```
In[12]: df = pd.DataFrame(np.random.rand(4, 2),
                        index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                        columns=['data1', 'data2'])

df
```

```
Out[12]:        data1     data2
a 1  0.554233  0.356072
      2  0.925244  0.219474
b 1  0.441759  0.610054
      2  0.171495  0.886688
```

The work of creating the `MultiIndex` is done in the background.

Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a `MultiIndex` by default:

```
In[13]: data = {('California', 2000): 33871648,
                ('California', 2010): 37253956,
                ('Texas', 2000): 20851820,
                ('Texas', 2010): 25145561,
                ('New York', 2000): 18976457,
                ('New York', 2010): 19378102}
pd.Series(data)

Out[13]: California  2000    33871648
                  2010    37253956
           New York   2000    18976457
                  2010    19378102
           Texas      2000    20851820
                  2010    25145561
           dtype: int64
```

Nevertheless, it is sometimes useful to explicitly create a `MultiIndex`; we'll see a couple of these methods here.

Explicit MultiIndex constructors

For more flexibility in how the index is constructed, you can instead use the class method constructors available in the `pd.MultiIndex`. For example, as we did before, you can construct the `MultiIndex` from a simple list of arrays, giving the index values within each level:

```
In[14]: pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])

Out[14]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                     labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can construct it from a list of tuples, giving the multiple index values of each point:

```
In[15]: pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])

Out[15]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                     labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can even construct it from a Cartesian product of single indices:

```
In[16]: pd.MultiIndex.from_product([('a', 'b'), [1, 2]])

Out[16]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                     labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Similarly, you can construct the `MultiIndex` directly using its internal encoding by passing `levels` (a list of lists containing available index values for each level) and `labels` (a list of lists that reference these labels):

```
In[17]: pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
                      labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

Out[17]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                      labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can pass any of these objects as the `index` argument when creating a `Series` or `DataFrame`, or to the `reindex` method of an existing `Series` or `DataFrame`.

MultilIndex level names

Sometimes it is convenient to name the levels of the `MultiIndex`. You can accomplish this by passing the `names` argument to any of the above `MultiIndex` constructors, or by setting the `names` attribute of the index after the fact:

```
In[18]: pop.index.names = ['state', 'year']
pop

Out[18]: state      year
          California  2000    33871648
                     2010    37253956
          New York   2000    18976457
                     2010    19378102
          Texas      2000    20851820
                     2010    25145561
dtype: int64
```

With more involved datasets, this can be a useful way to keep track of the meaning of various index values.

MultilIndex for columns

In a `DataFrame`, the rows and columns are completely symmetric, and just as the rows can have multiple levels of indices, the columns can have multiple levels as well. Consider the following, which is a mock-up of some (somewhat realistic) medical data:

```
In[19]:
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                    names=['year', 'visit'])
columns = pd.MultiIndex.from_product([('Bob', 'Guido', 'Sue'), ['HR', 'Temp']],
                                    names=['subject', 'type'])

# mock some data
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37

# create the DataFrame
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
```

```
Out[19]: subject      Bob        Guido        Sue
          type         HR  Temp       HR  Temp       HR  Temp
          year visit
2013 1      31.0  38.7  32.0  36.7  35.0  37.2
            2      44.0  37.7  50.0  35.0  29.0  36.7
2014 1      30.0  37.4  39.0  37.8  61.0  36.9
            2      47.0  37.8  48.0  37.3  51.0  36.5
```

Here we see where the multi-indexing for both rows and columns can come in *very* handy. This is fundamentally four-dimensional data, where the dimensions are the subject, the measurement type, the year, and the visit number. With this in place we can, for example, index the top-level column by the person's name and get a full `DataFrame` containing just that person's information:

```
In[20]: health_data['Guido']

Out[20]: type      HR  Temp
          year visit
2013 1      32.0  36.7
            2      50.0  35.0
2014 1      39.0  37.8
            2      48.0  37.3
```

For complicated records containing multiple labeled measurements across multiple times for many subjects (people, countries, cities, etc.), use of hierarchical rows and columns can be extremely convenient!

Indexing and Slicing a MultiIndex

Indexing and slicing on a `MultiIndex` is designed to be intuitive, and it helps if you think about the indices as added dimensions. We'll first look at indexing multiply indexed `Series`, and then multiply indexed `DataFrames`.

Multiply indexed Series

Consider the multiply indexed `Series` of state populations we saw earlier:

```
In[21]: pop

Out[21]: state      year
          California  2000    33871648
                      2010    37253956
          New York    2000    18976457
                      2010    19378102
          Texas       2000    20851820
                      2010    25145561
dtype: int64
```

We can access single elements by indexing with multiple terms:

```
In[22]: pop['California', 2000]

Out[22]: 33871648
```

The MultiIndex also supports *partial indexing*, or indexing just one of the levels in the index. The result is another Series, with the lower-level indices maintained:

```
In[23]: pop['California']
```

```
Out[23]: year
          2000    33871648
          2010    37253956
          dtype: int64
```

Partial slicing is available as well, as long as the MultiIndex is sorted (see discussion in “[Sorted and unsorted indices](#)” on page 137):

```
In[24]: pop.loc['California':'New York']
```

```
Out[24]: state      year
          California  2000    33871648
                      2010    37253956
          New York    2000    18976457
                      2010    19378102
          dtype: int64
```

With sorted indices, we can perform partial indexing on lower levels by passing an empty slice in the first index:

```
In[25]: pop[:, 2000]
```

```
Out[25]: state
          California    33871648
          New York     18976457
          Texas        20851820
          dtype: int64
```

Other types of indexing and selection (discussed in “[Data Indexing and Selection](#)” on page 107) work as well; for example, selection based on Boolean masks:

```
In[26]: pop[pop > 22000000]
```

```
Out[26]: state      year
          California  2000    33871648
                      2010    37253956
          Texas       2010    25145561
          dtype: int64
```

Selection based on fancy indexing also works:

```
In[27]: pop[['California', 'Texas']]
```

```
Out[27]: state      year
          California  2000    33871648
                      2010    37253956
          Texas       2000    20851820
                      2010    25145561
          dtype: int64
```

Multiply indexed DataFrames

A multiply indexed DataFrame behaves in a similar manner. Consider our toy medical DataFrame from before:

```
In[28]: health_data
```

```
Out[28]: subject      Bob        Guido        Sue
          type         HR  Temp       HR  Temp       HR  Temp
          year visit
2013   1      31.0  38.7  32.0  36.7  35.0  37.2
        2      44.0  37.7  50.0  35.0  29.0  36.7
2014   1      30.0  37.4  39.0  37.8  61.0  36.9
        2      47.0  37.8  48.0  37.3  51.0  36.5
```

Remember that columns are primary in a DataFrame, and the syntax used for multiply indexed Series applies to the columns. For example, we can recover Guido's heart rate data with a simple operation:

```
In[29]: health_data['Guido', 'HR']
```

```
Out[29]: year  visit
2013  1      32.0
      2      50.0
2014  1      39.0
      2      48.0
Name: (Guido, HR), dtype: float64
```

Also, as with the single-index case, we can use the `loc`, `iloc`, and `ix` indexers introduced in “[Data Indexing and Selection](#)” on page 107. For example:

```
In[30]: health_data.iloc[:2, :2]
```

```
Out[30]: subject      Bob
          type         HR  Temp
          year visit
2013  1      31.0  38.7
      2      44.0  37.7
```

These indexers provide an array-like view of the underlying two-dimensional data, but each individual index in `loc` or `iloc` can be passed a tuple of multiple indices. For example:

```
In[31]: health_data.loc[:, ('Bob', 'HR')]
```

```
Out[31]: year  visit
2013  1      31.0
      2      44.0
2014  1      30.0
      2      47.0
Name: (Bob, HR), dtype: float64
```

Working with slices within these index tuples is not especially convenient; trying to create a slice within a tuple will lead to a syntax error:

```
In[32]: health_data.loc[:, 1], :, 'HR')]  
File "<ipython-input-32-8e3cc151e316>", line 1  
    health_data.loc[:, 1], :, 'HR')]  
          ^  
SyntaxError: invalid syntax
```

You could get around this by building the desired slice explicitly using Python's built-in `slice()` function, but a better way in this context is to use an `IndexSlice` object, which Pandas provides for precisely this situation. For example:

```
In[33]: idx = pd.IndexSlice  
        health_data.loc[idx[:, 1], idx[:, 'HR']]  
  
Out[33]: subject      Bob Guido     Sue  
         type          HR     HR     HR  
         year visit  
         2013 1      31.0  32.0  35.0  
         2014 1      30.0  39.0  61.0
```

There are so many ways to interact with data in multiply indexed `Series` and `DataFrames`, and as with many tools in this book the best way to become familiar with them is to try them out!

Rearranging Multi-Indices

One of the keys to working with multiply indexed data is knowing how to effectively transform the data. There are a number of operations that will preserve all the information in the dataset, but rearrange it for the purposes of various computations. We saw a brief example of this in the `stack()` and `unstack()` methods, but there are many more ways to finely control the rearrangement of data between hierarchical indices and columns, and we'll explore them here.

Sorted and unsorted indices

Earlier, we briefly mentioned a caveat, but we should emphasize it more here. *Many of the MultiIndex slicing operations will fail if the index is not sorted.* Let's take a look at this here.

We'll start by creating some simple multiply indexed data where the indices are *not lexographically sorted*:

```
In[34]: index = pd.MultiIndex.from_product([[['a', 'c', 'b'], [1, 2]]])  
        data = pd.Series(np.random.rand(6), index=index)  
        data.index.names = ['char', 'int']  
        data  
  
Out[34]: char  int  
         a      1    0.003001  
                 2    0.164974  
         c      1    0.741650
```

```
      2      0.569264
b      1      0.001693
      2      0.526226
dtype: float64
```

If we try to take a partial slice of this index, it will result in an error:

```
In[35]: try:
    data['a':'b']
except KeyError as e:
    print(type(e))
    print(e)

<class 'KeyError'>
'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

Although it is not entirely clear from the error message, this is the result of the MultiIndex not being sorted. For various reasons, partial slices and other similar operations require the levels in the MultiIndex to be in sorted (i.e., lexicographical) order. Pandas provides a number of convenience routines to perform this type of sorting; examples are the `sort_index()` and `sortlevel()` methods of the DataFrame. We'll use the simplest, `sort_index()`, here:

```
In[36]: data = data.sort_index()
data

Out[36]: char  int
      a      1      0.003001
              2      0.164974
      b      1      0.001693
              2      0.526226
      c      1      0.741650
              2      0.569264
dtype: float64
```

With the index sorted in this way, partial slicing will work as expected:

```
In[37]: data['a':'b']

Out[37]: char  int
      a      1      0.003001
              2      0.164974
      b      1      0.001693
              2      0.526226
dtype: float64
```

Stacking and unstacking indices

As we saw briefly before, it is possible to convert a dataset from a stacked multi-index to a simple two-dimensional representation, optionally specifying the level to use:

```
In[38]: pop.unstack(level=0)

Out[38]: state    California     New York      Texas
year
2000        33871648   18976457  20851820
2010        37253956   19378102  25145561

In[39]: pop.unstack(level=1)

Out[39]: year          2000          2010
state
California  33871648  37253956
New York    18976457  19378102
Texas       20851820  25145561
```

The opposite of `unstack()` is `stack()`, which here can be used to recover the original series:

```
In[40]: pop.unstack().stack()

Out[40]: state      year
California  2000    33871648
              2010    37253956
New York    2000    18976457
              2010    19378102
Texas       2000    20851820
              2010    25145561
dtype: int64
```

Index setting and resetting

Another way to rearrange hierarchical data is to turn the index labels into columns; this can be accomplished with the `reset_index` method. Calling this on the population dictionary will result in a DataFrame with a `state` and `year` column holding the information that was formerly in the index. For clarity, we can optionally specify the name of the data for the column representation:

```
In[41]: pop_flat = pop.reset_index(name='population')
pop_flat

Out[41]:      state  year  population
0  California  2000    33871648
1  California  2010    37253956
2    New York  2000    18976457
3    New York  2010    19378102
4      Texas  2000    20851820
5      Texas  2010    25145561
```

Often when you are working with data in the real world, the raw input data looks like this and it's useful to build a `MultiIndex` from the column values. This can be done with the `set_index` method of the DataFrame, which returns a multiply indexed DataFrame:

```
In[42]: pop_flat.set_index(['state', 'year'])
```

```
Out[42]:           population
      state      year
      California 2000    33871648
                  2010    37253956
      New York   2000    18976457
                  2010    19378102
      Texas      2000    20851820
                  2010    25145561
```

In practice, I find this type of reindexing to be one of the more useful patterns when I encounter real-world datasets.

Data Aggregations on Multi-Indices

We've previously seen that Pandas has built-in data aggregation methods, such as `mean()`, `sum()`, and `max()`. For hierarchically indexed data, these can be passed a `level` parameter that controls which subset of the data the aggregate is computed on.

For example, let's return to our health data:

```
In[43]: health_data
```

```
Out[43]:   subject      Bob        Guido        Sue
          type        HR  Temp        HR  Temp        HR  Temp
          year visit
          2013 1      31.0  38.7  32.0  36.7  35.0  37.2
                  2      44.0  37.7  50.0  35.0  29.0  36.7
          2014 1      30.0  37.4  39.0  37.8  61.0  36.9
                  2      47.0  37.8  48.0  37.3  51.0  36.5
```

Perhaps we'd like to average out the measurements in the two visits each year. We can do this by naming the index level we'd like to explore, in this case the `year`:

```
In[44]: data_mean = health_data.mean(level='year')
         data_mean
```

```
Out[44]:   subject      Bob        Guido        Sue
          type        HR  Temp        HR  Temp        HR  Temp
          year
          2013      37.5  38.2  41.0  35.85  32.0  36.95
          2014      38.5  37.6  43.5  37.55  56.0  36.70
```

By further making use of the `axis` keyword, we can take the mean among levels on the columns as well:

```
In[45]: data_mean.mean(axis=1, level='type')
```

```
Out[45]:   type        HR        Temp
          year
          2013  36.833333  37.000000
          2014  46.000000  37.283333
```

Thus in two lines, we've been able to find the average heart rate and temperature measured among all subjects in all visits each year. This syntax is actually a shortcut to the `GroupBy` functionality, which we will discuss in “[Aggregation and Grouping](#)” on [page 158](#). While this is a toy example, many real-world datasets have similar hierarchical structure.

Panel Data

Pandas has a few other fundamental data structures that we have not yet discussed, namely the `pd.Panel` and `pd.Panel4D` objects. These can be thought of, respectively, as three-dimensional and four-dimensional generalizations of the (one-dimensional) `Series` and (two-dimensional) `DataFrame` structures. Once you are familiar with indexing and manipulation of data in a `Series` and `DataFrame`, `Panel` and `Panel4D` are relatively straightforward to use. In particular, the `ix`, `loc`, and `iloc` indexers discussed in “[Data Indexing and Selection](#)” on [page 107](#) extend readily to these higher-dimensional structures.

We won't cover these panel structures further in this text, as I've found in the majority of cases that multi-indexing is a more useful and conceptually simpler representation for higher-dimensional data. Additionally, panel data is fundamentally a dense data representation, while multi-indexing is fundamentally a sparse data representation. As the number of dimensions increases, the dense representation can become very inefficient for the majority of real-world datasets. For the occasional specialized application, however, these structures can be useful. If you'd like to read more about the `Panel` and `Panel4D` structures, see the references listed in “[Further Resources](#)” on [page 215](#).

Combining Datasets: Concat and Append

Some of the most interesting studies of data come from combining different data sources. These operations can involve anything from very straightforward concatenation of two different datasets, to more complicated database-style joins and merges that correctly handle any overlaps between the datasets. `Series` and `DataFrames` are built with this type of operation in mind, and Pandas includes functions and methods that make this sort of data wrangling fast and straightforward.

Here we'll take a look at simple concatenation of `Series` and `DataFrames` with the `pd.concat` function; later we'll dive into more sophisticated in-memory merges and joins implemented in Pandas.

We begin with the standard imports:

```
In[1]: import pandas as pd  
       import numpy as np
```

For convenience, we'll define this function, which creates a `DataFrame` of a particular form that will be useful below:

```
In[2]: def make_df(cols, ind):
    """Quickly make a DataFrame"""
    data = {c: [str(c) + str(i) for i in ind]
            for c in cols}
    return pd.DataFrame(data, ind)

# example DataFrame
make_df('ABC', range(3))

Out[2]:   A   B   C
0  A0  B0  C0
1  A1  B1  C1
2  A2  B2  C2
```

Recall: Concatenation of NumPy Arrays

Concatenation of `Series` and `DataFrame` objects is very similar to concatenation of NumPy arrays, which can be done via the `np.concatenate` function as discussed in “[The Basics of NumPy Arrays](#)” on page 42. Recall that with it, you can combine the contents of two or more arrays into a single array:

```
In[4]: x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]
np.concatenate([x, y, z])

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

The first argument is a list or tuple of arrays to concatenate. Additionally, it takes an `axis` keyword that allows you to specify the axis along which the result will be concatenated:

```
In[5]: x = [[1, 2],
           [3, 4]]
np.concatenate([x, x], axis=1)

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

Simple Concatenation with `pd.concat`

Pandas has a function, `pd.concat()`, which has a similar syntax to `np.concatenate` but contains a number of options that we'll discuss momentarily:

```
# Signature in Pandas v0.18
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False,
          copy=True)
```

`pd.concat()` can be used for a simple concatenation of Series or DataFrame objects, just as `np.concatenate()` can be used for simple concatenations of arrays:

```
In[6]: ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
       ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
       pd.concat([ser1, ser2])

Out[6]: 1    A
        2    B
        3    C
        4    D
        5    E
        6    F
       dtype: object
```

It also works to concatenate higher-dimensional objects, such as DataFrames:

```
In[7]: df1 = make_df('AB', [1, 2])
       df2 = make_df('AB', [3, 4])
       print(df1); print(df2); print(pd.concat([df1, df2]))

df1           df2           pd.concat([df1, df2])
      A   B      A   B      A   B
1  A1  B1    3  A3  B3    1  A1  B1
2  A2  B2    4  A4  B4    2  A2  B2
                           3  A3  B3
                           4  A4  B4
```

By default, the concatenation takes place row-wise within the DataFrame (i.e., `axis=0`). Like `np.concatenate`, `pd.concat` allows specification of an axis along which concatenation will take place. Consider the following example:

```
In[8]: df3 = make_df('AB', [0, 1])
       df4 = make_df('CD', [0, 1])
       print(df3); print(df4); print(pd.concat([df3, df4], axis='col'))

df3           df4           pd.concat([df3, df4], axis='col')
      A   B      C   D      A   B   C   D
0  A0  B0    0  C0  D0    0  A0  B0  C0  D0
1  A1  B1    1  C1  D1    1  A1  B1  C1  D1
```

We could have equivalently specified `axis=1`; here we've used the more intuitive `axis='col'`.

Duplicate indices

One important difference between `np.concatenate` and `pd.concat` is that Pandas concatenation *preserves indices*, even if the result will have duplicate indices! Consider this simple example:

```
In[9]: x = make_df('AB', [0, 1])
       y = make_df('AB', [2, 3])
```

```

y.index = x.index # make duplicate indices!
print(x); print(y); print(pd.concat([x, y]))

x           y           pd.concat([x, y])
   A   B       A   B       A   B
0  A0  B0     0  A2  B2     0  A0  B0
1  A1  B1     1  A3  B3     1  A1  B1
                           0  A2  B2
                           1  A3  B3

```

Notice the repeated indices in the result. While this is valid within `DataFrames`, the outcome is often undesirable. `pd.concat()` gives us a few ways to handle it.

Catching the repeats as an error. If you'd like to simply verify that the indices in the result of `pd.concat()` do not overlap, you can specify the `verify_integrity` flag. With this set to `True`, the concatenation will raise an exception if there are duplicate indices. Here is an example, where for clarity we'll catch and print the error message:

```

In[10]: try:
            pd.concat([x, y], verify_integrity=True)
        except ValueError as e:
            print("ValueError:", e)

ValueError: Indexes have overlapping values: [0, 1]

```

Ignoring the index. Sometimes the index itself does not matter, and you would prefer it to simply be ignored. You can specify this option using the `ignore_index` flag. With this set to `True`, the concatenation will create a new integer index for the resulting `Series`:

```

In[11]: print(x); print(y); print(pd.concat([x, y], ignore_index=True))

x           y           pd.concat([x, y], ignore_index=True)
   A   B       A   B       A   B
0  A0  B0     0  A2  B2     0  A0  B0
1  A1  B1     1  A3  B3     1  A1  B1
                           2  A2  B2
                           3  A3  B3

```

Adding MultiIndex keys. Another alternative is to use the `keys` option to specify a label for the data sources; the result will be a hierarchically indexed series containing the data:

```

In[12]: print(x); print(y); print(pd.concat([x, y], keys=['x', 'y']))

x           y           pd.concat([x, y], keys=['x', 'y'])
   A   B       A   B           A   B
0  A0  B0     0  A2  B2     x  0  A0  B0
1  A1  B1     1  A3  B3           1  A1  B1
                           y  0  A2  B2
                           1  A3  B3

```

The result is a multiply indexed DataFrame, and we can use the tools discussed in “Hierarchical Indexing” on page 128 to transform this data into the representation we’re interested in.

Concatenation with joins

In the simple examples we just looked at, we were mainly concatenating DataFrames with shared column names. In practice, data from different sources might have different sets of column names, and pd.concat offers several options in this case. Consider the concatenation of the following two DataFrames, which have some (but not all!) columns in common:

```
In[13]: df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
print(df5); print(df6); print(pd.concat([df5, df6]))
```

	df5			df6			pd.concat([df5, df6])					
	A	B	C	B	C	D	A	B	C	D		
1	A1	B1	C1	3	B3	C3	D3	1	A1	B1	C1	NaN
2	A2	B2	C2	4	B4	C4	D4	2	A2	B2	C2	NaN
							3	NaN	B3	C3	D3	
							4	NaN	B4	C4	D4	

By default, the entries for which no data is available are filled with NA values. To change this, we can specify one of several options for the join and join_axes parameters of the concatenate function. By default, the join is a union of the input columns (join='outer'), but we can change this to an intersection of the columns using join='inner':

```
In[14]: print(df5); print(df6);
print(pd.concat([df5, df6], join='inner'))
```

	df5			df6			pd.concat([df5, df6], join='inner')				
	A	B	C	B	C	D	B	C			
1	A1	B1	C1	3	B3	C3	D3	1	B1	C1	
2	A2	B2	C2	4	B4	C4	D4	2	B2	C2	
							3	B3	C3		
							4	B4	C4		

Another option is to directly specify the index of the remaining columns using the join_axes argument, which takes a list of index objects. Here we’ll specify that the returned columns should be the same as those of the first input:

```
In[15]: print(df5); print(df6);
print(pd.concat([df5, df6], join_axes=[df5.columns]))
```

	df5			df6			pd.concat([df5, df6], join_axes=[df5.columns])				
	A	B	C	B	C	D	A	B	C		
1	A1	B1	C1	3	B3	C3	D3	1	A1	B1	C1
2	A2	B2	C2	4	B4	C4	D4	2	A2	B2	C2

```
3  NaN  B3  C3
4  NaN  B4  C4
```

The combination of options of the `pd.concat` function allows a wide range of possible behaviors when you are joining two datasets; keep these in mind as you use these tools for your own data.

The `append()` method

Because direct array concatenation is so common, `Series` and `DataFrame` objects have an `append` method that can accomplish the same thing in fewer keystrokes. For example, rather than calling `pd.concat([df1, df2])`, you can simply call `df1.append(df2)`:

```
In[16]: print(df1); print(df2); print(df1.append(df2))

df1              df2              df1.append(df2)
   A    B      A    B      A    B
1  A1  B1    3  A3  B3    1  A1  B1
2  A2  B2    4  A4  B4    2  A2  B2
                           3  A3  B3
                           4  A4  B4
```

Keep in mind that unlike the `append()` and `extend()` methods of Python lists, the `append()` method in Pandas does not modify the original object—instead, it creates a new object with the combined data. It also is not a very efficient method, because it involves creation of a new index *and* data buffer. Thus, if you plan to do multiple `append` operations, it is generally better to build a list of `DataFrames` and pass them all at once to the `concat()` function.

In the next section, we'll look at another more powerful approach to combining data from multiple sources, the database-style merges/joins implemented in `pd.merge`. For more information on `concat()`, `append()`, and related functionality, see the “[Merge, Join, and Concatenate](#)” section of the Pandas documentation.

Combining Datasets: Merge and Join

One essential feature offered by Pandas is its high-performance, in-memory join and merge operations. If you have ever worked with databases, you should be familiar with this type of data interaction. The main interface for this is the `pd.merge` function, and we'll see a few examples of how this can work in practice.

Relational Algebra

The behavior implemented in `pd.merge()` is a subset of what is known as *relational algebra*, which is a formal set of rules for manipulating relational data, and forms the conceptual foundation of operations available in most databases. The strength of the

relational algebra approach is that it proposes several primitive operations, which become the building blocks of more complicated operations on any dataset. With this lexicon of fundamental operations implemented efficiently in a database or other program, a wide range of fairly complicated composite operations can be performed.

Pandas implements several of these fundamental building blocks in the `pd.merge()` function and the related `join()` method of `Series` and `DataFrames`. As we will see, these let you efficiently link data from different sources.

Categories of Joins

The `pd.merge()` function implements a number of types of joins: the *one-to-one*, *many-to-one*, and *many-to-many* joins. All three types of joins are accessed via an identical call to the `pd.merge()` interface; the type of join performed depends on the form of the input data. Here we will show simple examples of the three types of merges, and discuss detailed options further below.

One-to-one joins

Perhaps the simplest type of merge expression is the one-to-one join, which is in many ways very similar to the column-wise concatenation seen in “[Combining Datasets: Concat and Append](#)” on page 141. As a concrete example, consider the following two `DataFrames`, which contain information on several employees in a company:

```
In[2]:  
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],  
                   'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})  
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],  
                   'hire_date': [2004, 2008, 2012, 2014]})  
print(df1); print(df2)  
  
df1                df2  
employee      group   employee  hire_date  
0    Bob  Accounting     0    Lisa    2004  
1    Jake  Engineering    1    Bob    2008  
2    Lisa  Engineering    2    Jake    2012  
3    Sue        HR         3    Sue    2014
```

To combine this information into a single `DataFrame`, we can use the `pd.merge()` function:

```
In[3]: df3 = pd.merge(df1, df2)  
df3  
  
Out[3]:   employee      group  hire_date  
0    Bob  Accounting    2008  
1    Jake  Engineering  2012  
2    Lisa  Engineering  2004  
3    Sue        HR      2014
```

The `pd.merge()` function recognizes that each `DataFrame` has an “employee” column, and automatically joins using this column as a key. The result of the merge is a new `DataFrame` that combines the information from the two inputs. Notice that the order of entries in each column is not necessarily maintained: in this case, the order of the “employee” column differs between `df1` and `df2`, and the `pd.merge()` function correctly accounts for this. Additionally, keep in mind that the merge in general discards the index, except in the special case of merges by index (see “[The `left_index` and `right_index` keywords](#)” on page 151).

Many-to-one joins

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting `DataFrame` will preserve those duplicate entries as appropriate. Consider the following example of a many-to-one join:

```
In[4]: df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                           'supervisor': ['Carly', 'Guido', 'Steve']})
        print(df3); print(df4); print(pd.merge(df3, df4))

df3                               df4
employee      group  hire_date   group supervisor
0    Bob  Accounting     2008      0  Accounting    Carly
1   Jake  Engineering     2012      1  Engineering    Guido
2   Lisa  Engineering     2004      2          HR    Steve
3    Sue           HR     2014

pd.merge(df3, df4)
   employee      group  hire_date supervisor
0    Bob  Accounting     2008      Carly
1   Jake  Engineering     2012      Guido
2   Lisa  Engineering     2004      Guido
3    Sue           HR     2014      Steve
```

The resulting `DataFrame` has an additional column with the “supervisor” information, where the information is repeated in one or more locations as required by the inputs.

Many-to-many joins

Many-to-many joins are a bit confusing conceptually, but are nevertheless well defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete example. Consider the following, where we have a `DataFrame` showing one or more skills associated with a particular group.

By performing a many-to-many join, we can recover the skills associated with any individual person:

```
In[5]: df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                                         'Engineering', 'Engineering', 'HR', 'HR'],
                           'skill': ['Python', 'R', 'Python', 'R', 'SQL', 'Excel']})
```

```

'skills': ['math', 'spreadsheets', 'coding', 'linux',
           'spreadsheets', 'organization']})
print(df1); print(df5); print(pd.merge(df1, df5))

df1                  df5
employee      group      group      skills
0    Bob  Accounting    0  Accounting      math
1   Jake  Engineering    1  Accounting  spreadsheets
2   Lisa  Engineering    2  Engineering      coding
3    Sue        HR        3  Engineering      linux
                           4        HR  spreadsheets
                           5        HR  organization

pd.merge(df1, df5)
  employee      group      skills
0    Bob  Accounting      math
1    Bob  Accounting  spreadsheets
2   Jake  Engineering      coding
3   Jake  Engineering      linux
4   Lisa  Engineering      coding
5   Lisa  Engineering      linux
6    Sue        HR  spreadsheets
7    Sue        HR  organization

```

These three types of joins can be used with other Pandas tools to implement a wide array of functionality. But in practice, datasets are rarely as clean as the one we're working with here. In the following section, we'll consider some of the options provided by `pd.merge()` that enable you to tune how the join operations work.

Specification of the Merge Key

We've already seen the default behavior of `pd.merge()`: it looks for one or more matching column names between the two inputs, and uses this as the key. However, often the column names will not match so nicely, and `pd.merge()` provides a variety of options for handling this.

The `on` keyword

Most simply, you can explicitly specify the name of the key column using the `on` keyword, which takes a column name or a list of column names:

```
In[6]: print(df1); print(df2); print(pd.merge(df1, df2, on='employee'))

df1                  df2
employee      group      employee  hire_date
0    Bob  Accounting        0    Lisa     2004
1   Jake  Engineering        1    Bob     2008
2   Lisa  Engineering        2   Jake     2012
3    Sue        HR          3    Sue     2014
```

```
pd.merge(df1, df2, on='employee')
   employee      group  hire_date
0      Bob  Accounting    2008
1     Jake  Engineering   2012
2     Lisa  Engineering   2004
3      Sue        HR     2014
```

This option works only if both the left and right `DataFrames` have the specified column name.

The `left_on` and `right_on` keywords

At times you may wish to merge two datasets with different column names; for example, we may have a dataset in which the employee name is labeled as “name” rather than “employee”. In this case, we can use the `left_on` and `right_on` keywords to specify the two column names:

```
In[7]:
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})

print(df1); print(df3);
print(pd.merge(df1, df3, left_on="employee", right_on="name"))

df1                  df3
   employee      group           name  salary
0      Bob  Accounting       0    Bob  70000
1     Jake  Engineering     1   Jake  80000
2     Lisa  Engineering     2   Lisa 120000
3      Sue        HR        3   Sue  90000

pd.merge(df1, df3, left_on="employee", right_on="name")
   employee      group  name  salary
0      Bob  Accounting  Bob  70000
1     Jake  Engineering  Jake  80000
2     Lisa  Engineering  Lisa 120000
3      Sue        HR     Sue  90000
```

The result has a redundant column that we can drop if desired—for example, by using the `drop()` method of `DataFrames`:

```
In[8]:
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)

Out[8]:   employee      group  salary
0      Bob  Accounting  70000
1     Jake  Engineering  80000
2     Lisa  Engineering 120000
3      Sue        HR    90000
```

The left_index and right_index keywords

Sometimes, rather than merging on a column, you would instead like to merge on an index. For example, your data might look like this:

```
In[9]: df1a = df1.set_index('employee')
        df2a = df2.set_index('employee')
        print(df1a); print(df2a)

df1a                               df2a
employee      group      employee      hire_date
Bob           Accounting   Lisa          2004
Jake          Engineering  Bob           2008
Lisa          Engineering  Jake          2012
Sue            HR           Sue           2014
```

You can use the index as the key for merging by specifying the `left_index` and/or `right_index` flags in `pd.merge()`:

```
In[10]: 
print(df1a); print(df2a);
print(pd.merge(df1a, df2a, left_index=True, right_index=True))

df1a                               df2a
employee      group      employee      hire_date
Bob           Accounting   Lisa          2004
Jake          Engineering  Bob           2008
Lisa          Engineering  Jake          2012
Sue            HR           Sue           2014

pd.merge(df1a, df2a, left_index=True, right_index=True)
                    group  hire_date
employee
Lisa          Engineering    2004
Bob           Accounting    2008
Jake          Engineering    2012
Sue            HR           2014
```

For convenience, `DataFrames` implement the `join()` method, which performs a merge that defaults to joining on indices:

```
In[11]: print(df1a); print(df2a); print(df1a.join(df2a))

df1a                               df2a
employee      group      employee      hire_date
Bob           Accounting   Lisa          2004
Jake          Engineering  Bob           2008
Lisa          Engineering  Jake          2012
Sue            HR           Sue           2014
```

```
df1a.join(df2a)
      group  hire_date
employee
Bob      Accounting    2008
Jake     Engineering   2012
Lisa     Engineering   2004
Sue       HR           2014
```

If you'd like to mix indices and columns, you can combine `left_index` with `right_on` or `left_on` with `right_index` to get the desired behavior:

```
In[12]:  
print(df1a); print(df3);  
print(pd.merge(df1a, df3, left_index=True, right_on='name'))  
  
df1a                      df3  
      group  
employee          name  salary  
Bob      Accounting    0  Bob    70000  
Jake     Engineering   1  Jake   80000  
Lisa     Engineering   2  Lisa  120000  
Sue       HR           3  Sue    90000  
  
pd.merge(df1a, df3, left_index=True, right_on='name')  
      group  name  salary  
0  Accounting  Bob   70000  
1  Engineering Jake   80000  
2  Engineering Lisa  120000  
3        HR     Sue   90000
```

All of these options also work with multiple indices and/or multiple columns; the interface for this behavior is very intuitive. For more information on this, see the “Merge, Join, and Concatenate” section of the Pandas documentation.

Specifying Set Arithmetic for Joins

In all the preceding examples we have glossed over one important consideration in performing a join: the type of set arithmetic used in the join. This comes up when a value appears in one key column but not the other. Consider this example:

```
In[13]: df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                           'food': ['fish', 'beans', 'bread']},
                           columns=['name', 'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                     'drink': ['wine', 'beer']},
                     columns=['name', 'drink'])
print(df6); print(df7); print(pd.merge(df6, df7))
```

```

df6           df7           pd.merge(df6, df7)
   name  food      name drink      name  food  drink
0  Peter  fish      0  Mary  wine     0  Mary  bread  wine
1  Paul  beans      1 Joseph  beer    1 Joseph  NaN   beer
2  Mary  bread

```

Here we have merged two datasets that have only a single “name” entry in common: Mary. By default, the result contains the *intersection* of the two sets of inputs; this is what is known as an *inner join*. We can specify this explicitly using the `how` keyword, which defaults to ‘`inner`’:

```
In[14]: pd.merge(df6, df7, how='inner')
```

```
Out[14]:   name  food  drink
            0  Mary  bread  wine
```

Other options for the `how` keyword are ‘`outer`’, ‘`left`’, and ‘`right`’. An *outer join* returns a join over the union of the input columns, and fills in all missing values with NAs:

```
In[15]: print(df6); print(df7); print(pd.merge(df6, df7, how='outer'))
```

```

df6           df7           pd.merge(df6, df7, how='outer')
   name  food      name drink      name  food  drink
0  Peter  fish      0  Mary  wine     0  Peter  fish  NaN
1  Paul  beans      1 Joseph  beer    1  Paul  beans  NaN
2  Mary  bread
                           2  Mary  bread  wine
                           3 Joseph  NaN  beer

```

The *left join* and *right join* return join over the left entries and right entries, respectively. For example:

```
In[16]: print(df6); print(df7); print(pd.merge(df6, df7, how='left'))
```

```

df6           df7           pd.merge(df6, df7, how='left')
   name  food      name drink      name  food  drink
0  Peter  fish      0  Mary  wine     0  Peter  fish  NaN
1  Paul  beans      1 Joseph  beer    1  Paul  beans  NaN
2  Mary  bread
                           2  Mary  bread  wine

```

The output rows now correspond to the entries in the left input. Using `how='right'` works in a similar manner.

All of these options can be applied straightforwardly to any of the preceding join types.

Overlapping Column Names: The `suffixes` Keyword

Finally, you may end up in a case where your two input `DataFrames` have conflicting column names. Consider this example:

```
In[17]: df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                           'rank': [1, 2, 3, 4]})
```

```

df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [3, 1, 4, 2]})
print(df8); print(df9); print(pd.merge(df8, df9, on="name"))

df8           df9           pd.merge(df8, df9, on="name")
   name  rank      name  rank      name  rank_x  rank_y
0  Bob     1      0  Bob     3      0  Bob       1       3
1  Jake    2      1  Jake    1      1  Jake       2       1
2  Lisa    3      2  Lisa    4      2  Lisa       3       4
3  Sue     4      3  Sue     2      3  Sue       4       2

```

Because the output would have two conflicting column names, the merge function automatically appends a suffix `_x` or `_y` to make the output columns unique. If these defaults are inappropriate, it is possible to specify a custom suffix using the `suffixes` keyword:

```

In[18]:
print(df8); print(df9);
print(pd.merge(df8, df9, on="name", suffixes=["_L", "_R"]))

df8           df9
   name  rank      name  rank
0  Bob     1      0  Bob     3
1  Jake    2      1  Jake    1
2  Lisa    3      2  Lisa    4
3  Sue     4      3  Sue     2

pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
   name  rank_L  rank_R
0  Bob       1       3
1  Jake      2       1
2  Lisa      3       4
3  Sue      4       2

```

These suffixes work in any of the possible join patterns, and work also if there are multiple overlapping columns.

For more information on these patterns, see “[Aggregation and Grouping](#)” on page 158, where we dive a bit deeper into relational algebra. Also see the “[Merge, Join, and Concatenate](#)” section of the Pandas documentation for further discussion of these topics.

Example: US States Data

Merge and join operations come up most often when one is combining data from different sources. Here we will consider an example of some data about US states and their populations. The data files can be found at <http://github.com/jakevdp/data-USstates/>:

```

In[19]:
# Following are shell commands to download the data

```

```
# !curl -O https://raw.githubusercontent.com/jakevdp/
#       data-USstates/master/state-population.csv
# !curl -O https://raw.githubusercontent.com/jakevdp/
#       data-USstates/master/state-areas.csv
# !curl -O https://raw.githubusercontent.com/jakevdp/
#       data-USstates/master/state-abbrevs.csv
```

Let's take a look at the three datasets, using the Pandas `read_csv()` function:

```
In[20]: pop = pd.read_csv('state-population.csv')
areas = pd.read_csv('state-areas.csv')
abbrvs = pd.read_csv('state-abbrevs.csv')

print(pop.head()); print(areas.head()); print(abbrvs.head())

pop.head()                                     areas.head()
   state/region    ages   year  population           state   area (sq. mi)
0        AL  under18  2012     1117489.0        0  Alabama      52423
1        AL     total  2012     4817528.0        1  Alaska      656425
2        AL  under18  2010     1130966.0        2  Arizona     114006
3        AL     total  2010     4785570.0        3  Arkansas     53182
4        AL  under18  2011     1125763.0        3  Arkansas     53182
                                         4  California     163707

abbrvs.head()
   state abbreviation
0  Alabama          AL
1  Alaska           AK
2  Arizona          AZ
3  Arkansas          AR
4  California        CA
```

Given this information, say we want to compute a relatively straightforward result: rank US states and territories by their 2010 population density. We clearly have the data here to find this result, but we'll have to combine the datasets to get it.

We'll start with a many-to-one merge that will give us the full state name within the population DataFrame. We want to merge based on the `state/region` column of `pop`, and the `abbreviation` column of `abbrvs`. We'll use `how='outer'` to make sure no data is thrown away due to mismatched labels.

```
In[21]: merged = pd.merge(pop, abbrvs, how='outer',
                        left_on='state/region', right_on='abbreviation')
merged = merged.drop('abbreviation', 1) # drop duplicate info
merged.head()

Out[21]:   state/region    ages   year  population      state
0        AL  under18  2012     1117489.0  Alabama
1        AL     total  2012     4817528.0  Alabama
2        AL  under18  2010     1130966.0  Alabama
3        AL     total  2010     4785570.0  Alabama
4        AL  under18  2011     1125763.0  Alabama
```

Let's double-check whether there were any mismatches here, which we can do by looking for rows with nulls:

```
In[22]: merged.isnull().any()
```

```
Out[22]: state/region    False
          ages        False
          year        False
          population   True
          state        True
          dtype: bool
```

Some of the population info is null; let's figure out which these are!

```
In[23]: merged[merged['population'].isnull()].head()
```

```
Out[23]:      state/region    ages  year  population  state
2448           PR  under18  1990       NaN  NaN
2449           PR    total  1990       NaN  NaN
2450           PR    total  1991       NaN  NaN
2451           PR  under18  1991       NaN  NaN
2452           PR    total  1993       NaN  NaN
```

It appears that all the null population values are from Puerto Rico prior to the year 2000; this is likely due to this data not being available from the original source.

More importantly, we see also that some of the new state entries are also null, which means that there was no corresponding entry in the abbrevs key! Let's figure out which regions lack this match:

```
In[24]: merged.loc[merged['state'].isnull(), 'state/region'].unique()
```

```
Out[24]: array(['PR', 'USA'], dtype=object)
```

We can quickly infer the issue: our population data includes entries for Puerto Rico (PR) and the United States as a whole (USA), while these entries do not appear in the state abbreviation key. We can fix these quickly by filling in appropriate entries:

```
In[25]: merged.loc[merged['state/region'] == 'PR', 'state'] = 'Puerto Rico'
merged.loc[merged['state/region'] == 'USA', 'state'] = 'United States'
merged.isnull().any()
```

```
Out[25]: state/region    False
          ages        False
          year        False
          population   True
          state        False
          dtype: bool
```

No more nulls in the state column: we're all set!

Now we can merge the result with the area data using a similar procedure. Examining our results, we will want to join on the state column in both:

```
In[26]: final = pd.merge(merged, areas, on='state', how='left')
final.head()

Out[26]: state/region    ages   year  population    state   area (sq. mi)
          0        AL  under18  2012  1117489.0  Alabama  52423.0
          1        AL    total  2012  4817528.0  Alabama  52423.0
          2        AL  under18  2010  1130966.0  Alabama  52423.0
          3        AL    total  2010  4785570.0  Alabama  52423.0
          4        AL  under18  2011  1125763.0  Alabama  52423.0
```

Again, let's check for nulls to see if there were any mismatches:

```
In[27]: final.isnull().any()

Out[27]: state/region      False
          ages        False
          year        False
          population     True
          state        False
          area (sq. mi)  True
          dtype: bool
```

There are nulls in the `area` column; we can take a look to see which regions were ignored here:

```
In[28]: final['state'][final['area (sq. mi)'].isnull()].unique()

Out[28]: array(['United States'], dtype=object)
```

We see that our `areas` DataFrame does not contain the area of the United States as a whole. We could insert the appropriate value (using the sum of all state areas, for instance), but in this case we'll just drop the null values because the population density of the entire United States is not relevant to our current discussion:

```
In[29]: final.dropna(inplace=True)
final.head()

Out[29]: state/region    ages   year  population    state   area (sq. mi)
          0        AL  under18  2012  1117489.0  Alabama  52423.0
          1        AL    total  2012  4817528.0  Alabama  52423.0
          2        AL  under18  2010  1130966.0  Alabama  52423.0
          3        AL    total  2010  4785570.0  Alabama  52423.0
          4        AL  under18  2011  1125763.0  Alabama  52423.0
```

Now we have all the data we need. To answer the question of interest, let's first select the portion of the data corresponding with the year 2000, and the total population. We'll use the `query()` function to do this quickly (this requires the `numexpr` package to be installed; see “[High-Performance Pandas: eval\(\) and query\(\)](#)” on page 208):

```
In[30]: data2010 = final.query("year == 2010 & ages == 'total'")
data2010.head()

Out[30]: state/region    ages   year  population    state   area (sq. mi)
          3        AL    total  2010  4785570.0  Alabama  52423.0
         91        AK    total  2010  713868.0  Alaska   656425.0
```

```
101          AZ total 2010 6408790.0    Arizona      114006.0
189          AR total 2010 2922280.0    Arkansas      53182.0
197          CA total 2010 37333601.0 California     163707.0
```

Now let's compute the population density and display it in order. We'll start by reindexing our data on the state, and then compute the result:

```
In[31]: data2010.set_index('state', inplace=True)
density = data2010['population'] / data2010['area (sq. mi)']

In[32]: density.sort_values(ascending=False, inplace=True)
density.head()

Out[32]: state
          District of Columbia    8898.897059
          Puerto Rico            1058.665149
          New Jersey             1009.253268
          Rhode Island           681.339159
          Connecticut            645.600649
          dtype: float64
```

The result is a ranking of US states plus Washington, DC, and Puerto Rico in order of their 2010 population density, in residents per square mile. We can see that by far the densest region in this dataset is Washington, DC (i.e., the District of Columbia); among states, the densest is New Jersey.

We can also check the end of the list:

```
In[33]: density.tail()

Out[33]: state
          South Dakota     10.583512
          North Dakota    9.537565
          Montana        6.736171
          Wyoming        5.768079
          Alaska         1.087509
          dtype: float64
```

We see that the least dense state, by far, is Alaska, averaging slightly over one resident per square mile.

This type of messy data merging is a common task when one is trying to answer questions using real-world data sources. I hope that this example has given you an idea of the ways you can combine tools we've covered in order to gain insight from your data!

Aggregation and Grouping

An essential piece of analysis of large data is efficient summarization: computing aggregations like `sum()`, `mean()`, `median()`, `min()`, and `max()`, in which a single number gives insight into the nature of a potentially large dataset. In this section, we'll

explore aggregations in Pandas, from simple operations akin to what we've seen on NumPy arrays, to more sophisticated operations based on the concept of a `groupby`.

Planets Data

Here we will use the Planets dataset, available via the [Seaborn package](#) (see “[Visualization with Seaborn](#)” on page 311). It gives information on planets that astronomers have discovered around other stars (known as *extrasolar planets* or *exoplanets* for short). It can be downloaded with a simple Seaborn command:

```
In[2]: import seaborn as sns  
planets = sns.load_dataset('planets')  
planets.shape  
  
Out[2]: (1035, 6)  
  
In[3]: planets.head()  
  
Out[3]:   method      number  orbital_period  mass  distance  year  
0  Radial Velocity    1        269.300    7.10    77.40  2006  
1  Radial Velocity    1        874.774    2.21    56.95  2008  
2  Radial Velocity    1        763.000    2.60    19.84  2011  
3  Radial Velocity    1        326.030   19.40   110.62  2007  
4  Radial Velocity    1        516.220   10.50   119.47  2009
```

This has some details on the 1,000+ exoplanets discovered up to 2014.

Simple Aggregation in Pandas

Earlier we explored some of the data aggregations available for NumPy arrays (“[Aggregations: Min, Max, and Everything in Between](#)” on page 58). As with a one-dimensional NumPy array, for a Pandas Series the aggregates return a single value:

```
In[4]: rng = np.random.RandomState(42)  
ser = pd.Series(rng.rand(5))  
ser  
  
Out[4]: 0    0.374540  
1    0.950714  
2    0.731994  
3    0.598658  
4    0.156019  
dtype: float64  
  
In[5]: ser.sum()  
  
Out[5]: 2.8119254917081569  
  
In[6]: ser.mean()  
  
Out[6]: 0.56238509834163142
```

For a DataFrame, by default the aggregates return results within each column:

```
In[7]: df = pd.DataFrame({'A': rng.rand(5),
                           'B': rng.rand(5)})
df
Out[7]:          A         B
0  0.155995  0.020584
1  0.058084  0.969910
2  0.866176  0.832443
3  0.601115  0.212339
4  0.708073  0.181825

In[8]: df.mean()
Out[8]: A    0.477888
        B    0.443420
       dtype: float64
```

By specifying the `axis` argument, you can instead aggregate within each row:

```
In[9]: df.mean(axis='columns')
Out[9]: 0    0.088290
        1    0.513997
        2    0.849309
        3    0.406727
        4    0.444949
       dtype: float64
```

Pandas Series and DataFrames include all of the common aggregates mentioned in “[Aggregations: Min, Max, and Everything in Between](#)” on page 58; in addition, there is a convenience method `describe()` that computes several common aggregates for each column and returns the result. Let’s use this on the Planets data, for now dropping rows with missing values:

```
In[10]: planets.dropna().describe()
Out[10]:      number  orbital_period      mass  distance      year
count    498.00000   498.000000  498.000000  498.000000  498.000000
mean     1.73494   835.778671   2.509320  52.068213  2007.377510
std      1.17572  1469.128259   3.636274  46.596041   4.167284
min      1.00000   1.328300   0.003600  1.350000  1989.000000
25%     1.00000   38.272250   0.212500  24.497500  2005.000000
50%     1.00000  357.000000   1.245000  39.940000  2009.000000
75%     2.00000  999.600000   2.867500  59.332500  2011.000000
max     6.00000  17337.500000  25.000000 354.000000  2014.000000
```

This can be a useful way to begin understanding the overall properties of a dataset. For example, we see in the `year` column that although exoplanets were discovered as far back as 1989, half of all known exoplanets were not discovered until 2010 or after. This is largely thanks to the *Kepler* mission, which is a space-based telescope specifically designed for finding eclipsing planets around other stars.

[Table 3-3](#) summarizes some other built-in Pandas aggregations.

Table 3-3. Listing of Pandas aggregation methods

Aggregation	Description
count()	Total number of items
first(), last()	First and last item
mean(), median()	Mean and median
min(), max()	Minimum and maximum
std(), var()	Standard deviation and variance
mad()	Mean absolute deviation
prod()	Product of all items
sum()	Sum of all items

These are all methods of `DataFrame` and `Series` objects.

To go deeper into the data, however, simple aggregates are often not enough. The next level of data summarization is the `groupby` operation, which allows you to quickly and efficiently compute aggregates on subsets of data.

GroupBy: Split, Apply, Combine

Simple aggregations can give you a flavor of your dataset, but often we would prefer to aggregate conditionally on some label or index: this is implemented in the so-called `groupby` operation. The name “group by” comes from a command in the SQL database language, but it is perhaps more illuminative to think of it in the terms first coined by Hadley Wickham of Rstats fame: *split, apply, combine*.

Split, apply, combine

A canonical example of this split-apply-combine operation, where the “apply” is a summation aggregation, is illustrated in [Figure 3-1](#).

[Figure 3-1](#) makes clear what the `GroupBy` accomplishes:

- The *split* step involves breaking up and grouping a `DataFrame` depending on the value of the specified key.
- The *apply* step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.
- The *combine* step merges the results of these operations into an output array.



Figure 3-1. A visual representation of a groupby operation

While we could certainly do this manually using some combination of the masking, aggregation, and merging commands covered earlier, it's important to realize that *the intermediate splits do not need to be explicitly instantiated*. Rather, the `GroupBy` can (often) do this in a single pass over the data, updating the sum, mean, count, min, or other aggregate for each group along the way. The power of the `GroupBy` is that it abstracts away these steps: the user need not think about *how* the computation is done under the hood, but rather thinks about the *operation as a whole*.

As a concrete example, let's take a look at using Pandas for the computation shown in Figure 3-1. We'll start by creating the input DataFrame:

```
In[11]: df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                           'data': range(6)}, columns=['key', 'data'])

Out[11]:   key  data
0     A     0
1     B     1
2     C     2
3     A     3
4     B     4
5     C     5
```

We can compute the most basic split-apply-combine operation with the `groupby()` method of `DataFrames`, passing the name of the desired key column:

```
In[12]: df.groupby('key')

Out[12]: <pandas.core.groupby.DataFrameGroupBy object at 0x117272160>
```

Notice that what is returned is not a set of `DataFrames`, but a `DataFrameGroupBy` object. This object is where the magic is: you can think of it as a special view of the `DataFrame`, which is poised to dig into the groups but does no actual computation until the aggregation is applied. This “lazy evaluation” approach means that common aggregates can be implemented very efficiently in a way that is almost transparent to the user.

To produce a result, we can apply an aggregate to this `DataFrameGroupBy` object, which will perform the appropriate `apply/combine` steps to produce the desired result:

```
In[13]: df.groupby('key').sum()  
Out[13]:      data  
    key  
    A      3  
    B      5  
    C      7
```

The `sum()` method is just one possibility here; you can apply virtually any common Pandas or NumPy aggregation function, as well as virtually any valid `DataFrame` operation, as we will see in the following discussion.

The GroupBy object

The `GroupBy` object is a very flexible abstraction. In many ways, you can simply treat it as if it’s a collection of `DataFrames`, and it does the difficult things under the hood. Let’s see some examples using the Planets data.

Perhaps the most important operations made available by a `GroupBy` are *aggregate*, *filter*, *transform*, and *apply*. We’ll discuss each of these more fully in “[Aggregate, filter, transform, apply](#)” on page 165, but before that let’s introduce some of the other functionality that can be used with the basic `GroupBy` operation.

Column indexing. The `GroupBy` object supports column indexing in the same way as the `DataFrame`, and returns a modified `GroupBy` object. For example:

```
In[14]: planets.groupby('method')  
Out[14]: <pandas.core.groupby.DataFrameGroupBy object at 0x1172727b8>  
In[15]: planets.groupby('method')['orbital_period']  
Out[15]: <pandas.core.groupby.SeriesGroupBy object at 0x117272da0>
```

Here we’ve selected a particular `Series` group from the original `DataFrame` group by reference to its column name. As with the `GroupBy` object, no computation is done until we call some aggregate on the object:

```
In[16]: planets.groupby('method')['orbital_period'].median()
```

```
Out[16]: method
          Astrometry           631.180000
          Eclipse Timing Variations 4343.500000
          Imaging            27500.000000
          Microlensing         3300.000000
          Orbital Brightness Modulation 0.342887
          Pulsar Timing        66.541900
          Pulsation Timing Variations 1170.000000
          Radial Velocity      360.200000
          Transit              5.714932
          Transit Timing Variations 57.011000
          Name: orbital_period, dtype: float64
```

This gives an idea of the general scale of orbital periods (in days) that each method is sensitive to.

Iteration over groups. The `GroupBy` object supports direct iteration over the groups, returning each group as a `Series` or `DataFrame`:

```
In[17]: for (method, group) in planets.groupby('method'):
    print("{0}: {1} shape={2}".format(method, group.shape))

Astrometry           shape=(2, 6)
Eclipse Timing Variations  shape=(9, 6)
Imaging             shape=(38, 6)
Microlensing         shape=(23, 6)
Orbital Brightness Modulation  shape=(3, 6)
Pulsar Timing        shape=(5, 6)
Pulsation Timing Variations  shape=(1, 6)
Radial Velocity      shape=(553, 6)
Transit              shape=(397, 6)
Transit Timing Variations  shape=(4, 6)
```

This can be useful for doing certain things manually, though it is often much faster to use the built-in `apply` functionality, which we will discuss momentarily.

Dispatch methods. Through some Python class magic, any method not explicitly implemented by the `GroupBy` object will be passed through and called on the groups, whether they are `DataFrame` or `Series` objects. For example, you can use the `describe()` method of `DataFrames` to perform a set of aggregations that describe each group in the data:

```
In[18]: planets.groupby('method')['year'].describe().unstack()

Out[18]:
               count      mean       std      min     25%  \\
method
Astrometry      2.0  2011.500000  2.121320  2010.0  2010.75
Eclipse Timing Variations  9.0  2010.000000  1.414214  2008.0  2009.00
Imaging         38.0  2009.131579  2.781901  2004.0  2008.00
Microlensing    23.0  2009.782609  2.859697  2004.0  2008.00
Orbital Brightness Modulation  3.0  2011.666667  1.154701  2011.0  2011.00
```

Pulsar Timing	5.0	1998.400000	8.384510	1992.0	1992.00
Pulsation Timing Variations	1.0	2007.000000	NaN	2007.0	2007.00
Radial Velocity	553.0	2007.518987	4.249052	1989.0	2005.00
Transit	397.0	2011.236776	2.077867	2002.0	2010.00
Transit Timing Variations	4.0	2012.500000	1.290994	2011.0	2011.75
				50%	75%
method					max
Astrometry	2011.5	2012.25	2013.0		
Eclipse Timing Variations	2010.0	2011.00	2012.0		
Imaging	2009.0	2011.00	2013.0		
Microlensing	2010.0	2012.00	2013.0		
Orbital Brightness Modulation	2011.0	2012.00	2013.0		
Pulsar Timing	1994.0	2003.00	2011.0		
Pulsation Timing Variations	2007.0	2007.00	2007.0		
Radial Velocity	2009.0	2011.00	2014.0		
Transit	2012.0	2013.00	2014.0		
Transit Timing Variations	2012.5	2013.25	2014.0		

Looking at this table helps us to better understand the data: for example, the vast majority of planets have been discovered by the Radial Velocity and Transit methods, though the latter only became common (due to new, more accurate telescopes) in the last decade. The newest methods seem to be Transit Timing Variation and Orbital Brightness Modulation, which were not used to discover a new planet until 2011.

This is just one example of the utility of dispatch methods. Notice that they are applied *to each individual group*, and the results are then combined within `GroupBy` and returned. Again, any valid `DataFrame/Series` method can be used on the corresponding `GroupBy` object, which allows for some very flexible and powerful operations!

Aggregate, filter, transform, apply

The preceding discussion focused on aggregation for the combine operation, but there are more options available. In particular, `GroupBy` objects have `aggregate()`, `filter()`, `transform()`, and `apply()` methods that efficiently implement a variety of useful operations before combining the grouped data.

For the purpose of the following subsections, we'll use this `DataFrame`:

```
In[19]: rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6),
                   'data2': rng.randint(0, 10, 6)},
                  columns = ['key', 'data1', 'data2'])
df

Out[19]:   key  data1  data2
0    A      0      5
1    B      1      0
2    C      2      3
```

```

3   A      3      3
4   B      4      7
5   C      5      9

```

Aggregation. We're now familiar with `GroupBy` aggregations with `sum()`, `median()`, and the like, but the `aggregate()` method allows for even more flexibility. It can take a string, a function, or a list thereof, and compute all the aggregates at once. Here is a quick example combining all these:

```
In[20]: df.groupby('key').aggregate(['min', np.median, max])
Out[20]:
          data1           data2
          min median max    min median max
key
A      0    1.5   3     3    4.0   5
B      1    2.5   4     0    3.5   7
C      2    3.5   5     3    6.0   9
```

Another useful pattern is to pass a dictionary mapping column names to operations to be applied on that column:

```
In[21]: df.groupby('key').aggregate({'data1': 'min',
                                    'data2': 'max'})
Out[21]:
          data1  data2
          key
A      0      5
B      1      7
C      2      9
```

Filtering. A filtering operation allows you to drop data based on the group properties. For example, we might want to keep all groups in which the standard deviation is larger than some critical value:

```
In[22]:
def filter_func(x):
    return x['data2'].std() > 4

print(df); print(df.groupby('key').std());
print(df.groupby('key').filter(filter_func))

df
          data1  data2
          key
0      0      5
1      1      0
2      2      3
3      3      3
4      4      7
5      5      9

df.groupby('key').filter(filter_func)
          data1  data2
          key
1      1      0
```

```
2   C      2      3
4   B      4      7
5   C      5      9
```

The `filter()` function should return a Boolean value specifying whether the group passes the filtering. Here because group A does not have a standard deviation greater than 4, it is dropped from the result.

Transformation. While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. For such a transformation, the output is the same shape as the input. A common example is to center the data by subtracting the group-wise mean:

```
In[23]: df.groupby('key').transform(lambda x: x - x.mean())
Out[23]:   data1  data2
0     -1.5    1.0
1     -1.5   -3.5
2     -1.5   -3.0
3      1.5   -1.0
4      1.5    3.5
5      1.5    3.0
```

The apply() method. The `apply()` method lets you apply an arbitrary function to the group results. The function should take a `DataFrame`, and return either a Pandas object (e.g., `DataFrame`, `Series`) or a scalar; the combine operation will be tailored to the type of output returned.

For example, here is an `apply()` that normalizes the first column by the sum of the second:

```
In[24]: def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x

print(df); print(df.groupby('key').apply(norm_by_data2))

df                               df.groupby('key').apply(norm_by_data2)
  key  data1  data2      key      data1  data2
0   A      0      5      0   A  0.000000      5
1   B      1      0      1   B  0.142857      0
2   C      2      3      2   C  0.166667      3
3   A      3      3      3   A  0.375000      3
4   B      4      7      4   B  0.571429      7
5   C      5      9      5   C  0.416667      9
```

`apply()` within a `GroupBy` is quite flexible: the only criterion is that the function takes a `DataFrame` and returns a Pandas object or scalar; what you do in the middle is up to you!

Specifying the split key

In the simple examples presented before, we split the DataFrame on a single column name. This is just one of many options by which the groups can be defined, and we'll go through some other options for group specification here.

A list, array, series, or index providing the grouping keys. The key can be any series or list with a length matching that of the DataFrame. For example:

```
In[25]: L = [0, 1, 0, 1, 2, 0]
print(df); print(df.groupby(L).sum())

df                               df.groupby(L).sum()
  key    data1   data2           data1   data2
0   A      0      5      0      7     17
1   B      1      0      1      4      3
2   C      2      3      2      4      7
3   A      3      3
4   B      4      7
5   C      5      9
```

Of course, this means there's another, more verbose way of accomplishing the `df.groupby('key')` from before:

```
In[26]: print(df); print(df.groupby(df['key']).sum())

df                               df.groupby(df['key']).sum()
  key    data1   data2           data1   data2
0   A      0      5      A      3     8
1   B      1      0      B      5     7
2   C      2      3      C      7    12
3   A      3      3
4   B      4      7
5   C      5      9
```

A dictionary or series mapping index to group. Another method is to provide a dictionary that maps index values to the group keys:

```
In[27]: df2 = df.set_index('key')
mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
print(df2); print(df2.groupby(mapping).sum())

df2                               df2.groupby(mapping).sum()
  key    data1   data2           data1   data2
A      0      5      consonant     12     19
B      1      0      vowel        3      8
C      2      3
A      3      3
B      4      7
C      5      9
```

Any Python function. Similar to mapping, you can pass any Python function that will input the index value and output the group:

```
In[28]: print(df2); print(df2.groupby(str.lower).mean())
```

```
df2                                     df2.groupby(str.lower).mean()
key  data1  data2                      data1  data2
A      0     5          a    1.5    4.0
B      1     0          b    2.5    3.5
C      2     3          c    3.5    6.0
A      3     3
B      4     7
C      5     9
```

A list of valid keys. Further, any of the preceding key choices can be combined to group on a multi-index:

```
In[29]: df2.groupby([str.lower, mapping]).mean()
```

```
Out[29]:           data1  data2
a vowel        1.5    4.0
b consonant    2.5    3.5
c consonant    3.5    6.0
```

Grouping example

As an example of this, in a couple lines of Python code we can put all these together and count discovered planets by method and by decade:

```
In[30]: decade = 10 * (planets['year'] // 10)
decade = decade.astype(str) + 's'
decade.name = 'decade'
planets.groupby(['method', decade])['number'].sum().unstack().fillna(0)
```

```
Out[30]: decade           1980s  1990s  2000s  2010s
method
Astrometry            0.0    0.0    0.0    2.0
Eclipse Timing Variations  0.0    0.0    5.0   10.0
Imaging                0.0    0.0   29.0   21.0
Microlensing             0.0    0.0   12.0   15.0
Orbital Brightness Modulation  0.0    0.0    0.0    5.0
Pulsar Timing              0.0    9.0    1.0    1.0
Pulsation Timing Variations  0.0    0.0    1.0    0.0
Radial Velocity            1.0   52.0  475.0  424.0
Transit                  0.0    0.0   64.0  712.0
Transit Timing Variations  0.0    0.0    0.0    9.0
```

This shows the power of combining many of the operations we've discussed up to this point when looking at realistic datasets. We immediately gain a coarse understanding of when and how planets have been discovered over the past several decades!

Here I would suggest digging into these few lines of code, and evaluating the individual steps to make sure you understand exactly what they are doing to the result. It's certainly a somewhat complicated example, but understanding these pieces will give you the means to similarly explore your own data.

Pivot Tables

We have seen how the `GroupBy` abstraction lets us explore relationships within a dataset. A *pivot table* is a similar operation that is commonly seen in spreadsheets and other programs that operate on tabular data. The pivot table takes simple column-wise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data. The difference between pivot tables and `GroupBy` can sometimes cause confusion; it helps me to think of pivot tables as essentially a *multidimensional* version of `GroupBy` aggregation. That is, you split-apply-combine, but both the split and the combine happen across not a one-dimensional index, but across a two-dimensional grid.

Motivating Pivot Tables

For the examples in this section, we'll use the database of passengers on the *Titanic*, available through the Seaborn library (see “[Visualization with Seaborn](#)” on page 311):

```
In[1]: import numpy as np
        import pandas as pd
        import seaborn as sns
        titanic = sns.load_dataset('titanic')

In[2]: titanic.head()

Out[2]:
   survived  pclass     sex   age  sibsp  parch     fare embarked class  \\
0         0       3    male  22.0      1      0    7.2500        S  Third
1         1       1  female  38.0      1      0   71.2833        C  First
2         1       1  female  26.0      0      0    7.9250        S  Third
3         1       1  female  35.0      1      0   53.1000        S  First
4         0       3    male  35.0      0      0    8.0500        S  Third

          who adult_male deck embark_town alive  alone
0    man        True    NaN  Southampton  no  False
1  woman       False     C  Cherbourg  yes  False
2  woman       False    NaN  Southampton  yes   True
3  woman       False     C  Southampton  yes  False
4    man        True    NaN  Southampton  no   True
```

This contains a wealth of information on each passenger of that ill-fated voyage, including gender, age, class, fare paid, and much more.

Pivot Tables by Hand

To start learning more about this data, we might begin by grouping it according to gender, survival status, or some combination thereof. If you have read the previous section, you might be tempted to apply a `GroupBy` operation—for example, let's look at survival rate by gender:

```
In[3]: titanic.groupby('sex')[['survived']].mean()  
Out[3]:      survived  
             sex  
             female  0.742038  
             male   0.188908
```

This immediately gives us some insight: overall, three of every four females on board survived, while only one in five males survived!

This is useful, but we might like to go one step deeper and look at survival by both sex and, say, class. Using the vocabulary of `GroupBy`, we might proceed using something like this: we *group by* class and gender, *select* survival, *apply* a mean aggregate, *combine* the resulting groups, and then *unstack* the hierarchical index to reveal the hidden multidimensionality. In code:

```
In[4]: titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()  
Out[4]: class      First      Second      Third  
         sex  
         female  0.968085  0.921053  0.500000  
         male   0.368852  0.157407  0.135447
```

This gives us a better idea of how both gender and class affected survival, but the code is starting to look a bit garbled. While each step of this pipeline makes sense in light of the tools we've previously discussed, the long string of code is not particularly easy to read or use. This two-dimensional `GroupBy` is common enough that Pandas includes a convenience routine, `pivot_table`, which succinctly handles this type of multidimensional aggregation.

Pivot Table Syntax

Here is the equivalent to the preceding operation using the `pivot_table` method of `DataFrames`:

```
In[5]: titanic.pivot_table('survived', index='sex', columns='class')  
Out[5]: class      First      Second      Third  
         sex  
         female  0.968085  0.921053  0.500000  
         male   0.368852  0.157407  0.135447
```

This is eminently more readable than the `GroupBy` approach, and produces the same result. As you might expect of an early 20th-century transatlantic cruise, the survival

gradient favors both women and higher classes. First-class women survived with near certainty (hi, Rose!), while only one in ten third-class men survived (sorry, Jack!).

Multilevel pivot tables

Just as in the `GroupBy`, the grouping in pivot tables can be specified with multiple levels, and via a number of options. For example, we might be interested in looking at age as a third dimension. We'll bin the age using the `pd.cut` function:

```
In[6]: age = pd.cut(titanic['age'], [0, 18, 80])
titanic.pivot_table('survived', ['sex', 'age'], 'class')

Out[6]:   class            First    Second    Third
          sex      age
          female (0, 18]  0.909091  1.000000  0.511628
                     (18, 80]  0.972973  0.900000  0.423729
          male   (0, 18]  0.800000  0.600000  0.215686
                     (18, 80]  0.375000  0.071429  0.133663
```

We can apply this same strategy when working with the columns as well; let's add info on the fare paid using `pd.qcut` to automatically compute quantiles:

```
In[7]: fare = pd.qcut(titanic['fare'], 2)
titanic.pivot_table('survived', ['sex', 'age'], [fare, 'class'])

Out[7]:
fare            [0, 14.454]
class           First    Second    Third    \\
sex      age
female (0, 18]      NaN  1.000000  0.714286
          (18, 80]      NaN  0.880000  0.444444
          male (0, 18]      NaN  0.000000  0.260870
          (18, 80]      0.0  0.098039  0.125000

fare            (14.454, 512.329]
class           First    Second    Third
sex      age
female (0, 18]  0.909091  1.000000  0.318182
          (18, 80]  0.972973  0.914286  0.391304
          male (0, 18]  0.800000  0.818182  0.178571
          (18, 80]  0.391304  0.030303  0.192308
```

The result is a four-dimensional aggregation with hierarchical indices (see “[Hierarchical Indexing](#)” on page 128), shown in a grid demonstrating the relationship between the values.

Additional pivot table options

The full call signature of the `pivot_table` method of `DataFrames` is as follows:

```
# call signature as of Pandas 0.18
DataFrame.pivot_table(data, values=None, index=None, columns=None,
                      aggfunc='mean', fill_value=None, margins=False,
                      dropna=True, margins_name='All')
```

We've already seen examples of the first three arguments; here we'll take a quick look at the remaining ones. Two of the options, `fill_value` and `dropna`, have to do with missing data and are fairly straightforward; we will not show examples of them here.

The `aggfunc` keyword controls what type of aggregation is applied, which is a mean by default. As in the `GroupBy`, the aggregation specification can be a string representing one of several common choices ('`sum`', '`mean`', '`count`', '`min`', '`max`', etc.) or a function that implements an aggregation (`np.sum()`, `min()`, `sum()`, etc.). Additionally, it can be specified as a dictionary mapping a column to any of the above desired options:

```
In[8]: titanic.pivot_table(index='sex', columns='class',
                           aggfunc={'survived':sum, 'fare':'mean'})

Out[8]:      fare          survived
            class First Second Third First Second Third
            sex
            female 106.125798 21.970121 16.118810 91.0 70.0 72.0
            male   67.226127 19.741782 12.661633 45.0 17.0 47.0
```

Notice also here that we've omitted the `values` keyword; when you're specifying a mapping for `aggfunc`, this is determined automatically.

At times it's useful to compute totals along each grouping. This can be done via the `margins` keyword:

```
In[9]: titanic.pivot_table('survived', index='sex', columns='class', margins=True)

Out[9]: class      First     Second     Third      All
        sex
        female  0.968085  0.921053  0.500000  0.742038
        male    0.368852  0.157407  0.135447  0.188908
        All     0.629630  0.472826  0.242363  0.383838
```

Here this automatically gives us information about the class-agnostic survival rate by gender, the gender-agnostic survival rate by class, and the overall survival rate of 38%. The margin label can be specified with the `margins_name` keyword, which defaults to "All".

Example: Birthrate Data

As a more interesting example, let's take a look at the freely available data on births in the United States, provided by the Centers for Disease Control (CDC). This data can be found at <https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv> (this dataset has been analyzed rather extensively by Andrew Gelman and his group; see, for example, [this blog post](#)):

```
In[10]:  
# shell command to download the data:  
# !curl -O https://raw.githubusercontent.com/jakevdp/data-CDCbirths/  
# master/births.csv  
  
In[11]: births = pd.read_csv('births.csv')
```

Taking a look at the data, we see that it's relatively simple—it contains the number of births grouped by date and gender:

```
In[12]: births.head()  
  
Out[12]:   year  month  day  gender  births  
0    1969      1     1      F     4046  
1    1969      1     1      M     4440  
2    1969      1     2      F     4454  
3    1969      1     2      M     4548  
4    1969      1     3      F     4548
```

We can start to understand this data a bit more by using a pivot table. Let's add a decade column, and take a look at male and female births as a function of decade:

```
In[13]:  
births['decade'] = 10 * (births['year'] // 10)  
births.pivot_table('births', index='decade', columns='gender', aggfunc='sum')  
  
Out[13]: gender          F          M  
decade  
1960    1753634    1846572  
1970    16263075    17121550  
1980    18310351    19243452  
1990    19479454    20420553  
2000    18229309    19106428
```

We immediately see that male births outnumber female births in every decade. To see this trend a bit more clearly, we can use the built-in plotting tools in Pandas to visualize the total number of births by year (Figure 3-2; see Chapter 4 for a discussion of plotting with Matplotlib):

```
In[14]:  
%matplotlib inline  
import matplotlib.pyplot as plt  
sns.set() # use Seaborn styles  
births.pivot_table('births', index='year', columns='gender', aggfunc='sum').plot()  
plt.ylabel('total births per year');
```

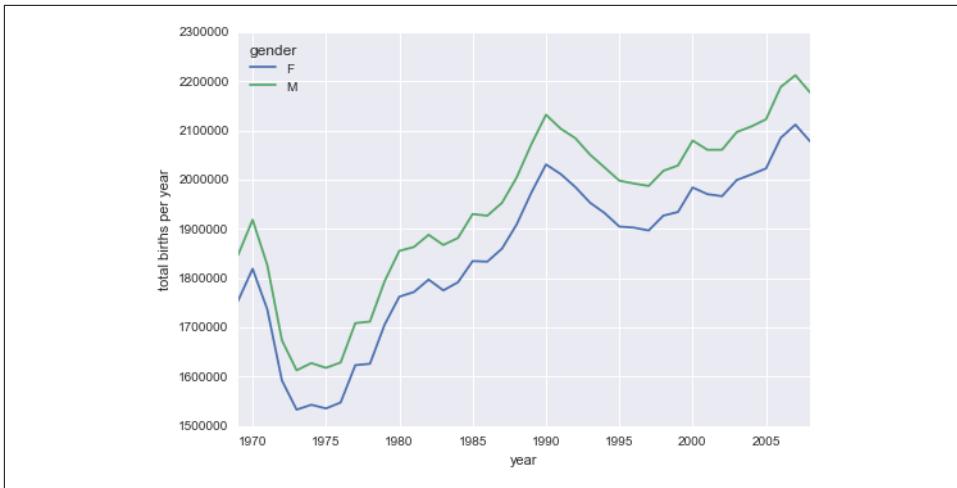


Figure 3-2. Total number of US births by year and gender

With a simple pivot table and `plot()` method, we can immediately see the annual trend in births by gender. By eye, it appears that over the past 50 years male births have outnumbered female births by around 5%.

Further data exploration

Though this doesn't necessarily relate to the pivot table, there are a few more interesting features we can pull out of this dataset using the Pandas tools covered up to this point. We must start by cleaning the data a bit, removing outliers caused by mistyped dates (e.g., June 31st) or missing values (e.g., June 99th). One easy way to remove these all at once is to cut outliers; we'll do this via a robust sigma-clipping operation:¹

```
In[15]: quartiles = np.percentile(births['births'], [25, 50, 75])
mu = quartiles[1]
sig = 0.74 * (quartiles[2] - quartiles[0])
```

This final line is a robust estimate of the sample mean, where the 0.74 comes from the interquartile range of a Gaussian distribution. With this we can use the `query()` method (discussed further in “[High-Performance Pandas: eval\(\) and query\(\)](#)” on [page 208](#)) to filter out rows with births outside these values:

```
In[16]:
births = births.query('(births > @mu - 5 * @sig) & (births < @mu + 5 * @sig)')
```

¹ You can learn more about sigma-clipping operations in a book I coauthored with Željko Ivezić, Andrew J. Connolly, and Alexander Gray: *Statistics, Data Mining, and Machine Learning in Astronomy: A Practical Python Guide for the Analysis of Survey Data* (Princeton University Press, 2014).

Next we set the day column to integers; previously it had been a string because some columns in the dataset contained the value 'null':

```
In[17]: # set 'day' column to integer; it originally was a string due to nulls
births['day'] = births['day'].astype(int)
```

Finally, we can combine the day, month, and year to create a Date index (see “[Working with Time Series](#)” on page 188). This allows us to quickly compute the weekday corresponding to each row:

```
In[18]: # create a datetime index from the year, month, day
births.index = pd.to_datetime(10000 * births.year +
                             100 * births.month +
                             births.day, format='%Y%m%d')

births['dayofweek'] = births.index.dayofweek
```

Using this we can plot births by weekday for several decades ([Figure 3-3](#)):

```
In[19]:
import matplotlib.pyplot as plt
import matplotlib as mpl

births.pivot_table('births', index='dayofweek',
                   columns='decade', aggfunc='mean').plot()
plt.gca().set_xticklabels(['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun'])
plt.ylabel('mean births by day');
```



Figure 3-3. Average daily births by day of week and decade

Apparently births are slightly less common on weekends than on weekdays! Note that the 1990s and 2000s are missing because the CDC data contains only the month of birth starting in 1989.

Another interesting view is to plot the mean number of births by the day of the *year*. Let's first group the data by month and day separately:

```
In[20]:  
births_by_date = births.pivot_table('births',  
                                     [births.index.month, births.index.day])  
births_by_date.head()  
  
Out[20]: 1    1    4009.225  
         2    4247.400  
         3    4500.900  
         4    4571.350  
         5    4603.625  
Name: births, dtype: float64
```

The result is a multi-index over months and days. To make this easily plottable, let's turn these months and days into a date by associating them with a dummy year variable (making sure to choose a leap year so February 29th is correctly handled!)

```
In[21]: births_by_date.index = [pd.datetime(2012, month, day)  
                               for (month, day) in births_by_date.index]  
births_by_date.head()  
  
Out[21]: 2012-01-01    4009.225  
2012-01-02    4247.400  
2012-01-03    4500.900  
2012-01-04    4571.350  
2012-01-05    4603.625  
Name: births, dtype: float64
```

Focusing on the month and day only, we now have a time series reflecting the average number of births by date of the year. From this, we can use the `plot` method to plot the data ([Figure 3-4](#)). It reveals some interesting trends:

```
In[22]: # Plot the results  
fig, ax = plt.subplots(figsize=(12, 4))  
births_by_date.plot(ax=ax);
```



Figure 3-4. Average daily births by date

In particular, the striking feature of this graph is the dip in birthrate on US holidays (e.g., Independence Day, Labor Day, Thanksgiving, Christmas, New Year's Day) although this likely reflects trends in scheduled/induced births rather than some deep psychosomatic effect on natural births. For more discussion on this trend, see the analysis and links in [Andrew Gelman's blog post](#) on the subject. We'll return to this figure in “[Example: Effect of Holidays on US Births](#)” on page 269, where we will use Matplotlib's tools to annotate this plot.

Looking at this short example, you can see that many of the Python and Pandas tools we've seen to this point can be combined and used to gain insight from a variety of datasets. We will see some more sophisticated applications of these data manipulations in future sections!

Vectorized String Operations

One strength of Python is its relative ease in handling and manipulating string data. Pandas builds on this and provides a comprehensive set of *vectorized string operations* that become an essential piece of the type of munging required when one is working with (read: cleaning up) real-world data. In this section, we'll walk through some of the Pandas string operations, and then take a look at using them to partially clean up a very messy dataset of recipes collected from the Internet.

Introducing Pandas String Operations

We saw in previous sections how tools like NumPy and Pandas generalize arithmetic operations so that we can easily and quickly perform the same operation on many array elements. For example:

```
In[1]: import numpy as np  
x = np.array([2, 3, 5, 7, 11, 13])  
x * 2  
  
Out[1]: array([ 4,  6, 10, 14, 22, 26])
```

This *vectorization* of operations simplifies the syntax of operating on arrays of data: we no longer have to worry about the size or shape of the array, but just about what operation we want done. For arrays of strings, NumPy does not provide such simple access, and thus you're stuck using a more verbose loop syntax:

```
In[2]: data = ['peter', 'Paul', 'MARY', 'gUIDO']  
[s.capitalize() for s in data]  
  
Out[2]: ['Peter', 'Paul', 'Mary', 'Guido']
```

This is perhaps sufficient to work with some data, but it will break if there are any missing values. For example:

```
In[3]: data = ['peter', 'Paul', None, 'MARY', 'gUIDO']  
[s.capitalize() for s in data]
```

```
-----
-----
AttributeError                                Traceback (most recent call last)

<ipython-input-3-fc1d891ab539> in <module>()
      1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
----> 2 [s.capitalize() for s in data]

<ipython-input-3-fc1d891ab539> in <listcomp>(.0)
      1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
----> 2 [s.capitalize() for s in data]

AttributeError: 'NoneType' object has no attribute 'capitalize'
```

Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data via the `str` attribute of Pandas Series and Index objects containing strings. So, for example, suppose we create a Pandas Series with this data:

```
In[4]: import pandas as pd
        names = pd.Series(data)
        names

Out[4]: 0    peter
        1    Paul
        2    None
        3    MARY
        4    gUIDO
       dtype: object
```

We can now call a single method that will capitalize all the entries, while skipping over any missing values:

```
In[5]: names.str.capitalize()

Out[5]: 0    Peter
        1    Paul
        2    None
        3    Mary
        4    Guido
       dtype: object
```

Using tab completion on this `str` attribute will list all the vectorized string methods available to Pandas.

Tables of Pandas String Methods

If you have a good understanding of string manipulation in Python, most of Pandas' string syntax is intuitive enough that it's probably sufficient to just list a table of available methods; we will start with that here, before diving deeper into a few of the subtleties. The examples in this section use the following series of names:

```
In[6]: monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                           'Eric Idle', 'Terry Jones', 'Michael Palin'])
```

Methods similar to Python string methods

Nearly all Python's built-in string methods are mirrored by a Pandas vectorized string method. Here is a list of Pandas `str` methods that mirror Python string methods:

<code>len()</code>	<code>lower()</code>	<code>translate()</code>	<code>islower()</code>
<code>ljust()</code>	<code>upper()</code>	<code>startswith()</code>	<code>isupper()</code>
<code>rjust()</code>	<code>find()</code>	<code>endswith()</code>	<code>isnumeric()</code>
<code>center()</code>	<code>rfind()</code>	<code>isalnum()</code>	<code>isdecimal()</code>
<code>zfill()</code>	<code>index()</code>	<code>isalpha()</code>	<code>split()</code>
<code>strip()</code>	<code>rindex()</code>	<code>isdigit()</code>	<code>rsplit()</code>
<code>rstrip()</code>	<code>capitalize()</code>	<code>isspace()</code>	<code>partition()</code>
<code>lstrip()</code>	<code>swapcase()</code>	<code>istitle()</code>	<code>rpartition()</code>

Notice that these have various return values. Some, like `lower()`, return a series of strings:

```
In[7]: monte.str.lower()

Out[7]: 0    graham chapman
        1    john cleese
        2    terry gilliam
        3    eric idle
        4    terry jones
        5    michael palin
       dtype: object
```

But some others return numbers:

```
In[8]: monte.str.len()

Out[8]: 0    14
        1    11
        2    13
        3     9
        4    11
        5    13
       dtype: int64
```

Or Boolean values:

```
In[9]: monte.str.startswith('T')
```

```
Out[9]: 0    False
        1    False
        2     True
        3    False
        4     True
        5    False
       dtype: bool
```

Still others return lists or other compound values for each element:

```
In[10]: monte.str.split()
```

```
Out[10]: 0    [Graham, Chapman]
        1    [John, Cleese]
        2    [Terry, Gilliam]
        3    [Eric, Idle]
        4    [Terry, Jones]
        5    [Michael, Palin]
       dtype: object
```

We'll see further manipulations of this kind of series-of-lists object as we continue our discussion.

Methods using regular expressions

In addition, there are several methods that accept regular expressions to examine the content of each string element, and follow some of the API conventions of Python's built-in `re` module (see [Table 3-4](#)).

Table 3-4. Mapping between Pandas methods and functions in Python's re module

Method	Description
<code>match()</code>	Call <code>re.match()</code> on each element, returning a Boolean.
<code>extract()</code>	Call <code>re.match()</code> on each element, returning matched groups as strings.
<code>findall()</code>	Call <code>re.findall()</code> on each element.
<code>replace()</code>	Replace occurrences of pattern with some other string.
<code>contains()</code>	Call <code>re.search()</code> on each element, returning a Boolean.
<code>count()</code>	Count occurrences of pattern.
<code>split()</code>	Equivalent to <code>str.split()</code> , but accepts regexps.
<code>rsplit()</code>	Equivalent to <code>str.rsplit()</code> , but accepts regexps.

With these, you can do a wide range of interesting operations. For example, we can extract the first name from each by asking for a contiguous group of characters at the beginning of each element:

```
In[11]: monte.str.extract('([A-Za-z]+)')

Out[11]: 0      Graham
          1      John
          2      Terry
          3      Eric
          4      Terry
          5    Michael
dtype: object
```

Or we can do something more complicated, like finding all names that start and end with a consonant, making use of the start-of-string (^) and end-of-string (\$) regular expression characters:

```
In[12]: monte.str.findall(r'^[^AEIOU].*[^aeiou]$')

Out[12]: 0    [Graham Chapman]
          1        []
          2    [Terry Gilliam]
          3        []
          4    [Terry Jones]
          5    [Michael Palin]
dtype: object
```

The ability to concisely apply regular expressions across Series or DataFrame entries opens up many possibilities for analysis and cleaning of data.

Miscellaneous methods

Finally, there are some miscellaneous methods that enable other convenient operations (see [Table 3-5](#)).

Table 3-5. Other Pandas string methods

Method	Description
get()	Index each element
slice()	Slice each element
slice_replace()	Replace slice in each element with passed value
cat()	Concatenate strings
repeat()	Repeat values
normalize()	Return Unicode form of string
pad()	Add whitespace to left, right, or both sides of strings
wrap()	Split long strings into lines with length less than a given width
join()	Join strings in each element of the Series with passed separator
get_dummies()	Extract dummy variables as a DataFrame