

# Preliminaries

## What Is This Book About?

This book is concerned with the nuts and bolts of manipulating, processing, cleaning, and crunching data in Python. It is also a practical, modern introduction to scientific computing in Python, tailored for data-intensive applications. This is a book about the parts of the Python language and libraries you'll need to effectively solve a broad set of data analysis problems. This book is *not* an exposition on analytical methods using Python as the implementation language.

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

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

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

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

# Why Python for Data Analysis?

For many people (myself among them), the Python language is easy to fall in love with. Since its first appearance in 1991, Python has become one of the most popular dynamic, programming languages, along with Perl, Ruby, and others. Python and Ruby have become especially popular in recent years for building websites using their numerous web frameworks, like Rails (Ruby) and Django (Python). Such languages are often called *scripting* languages as they can be used to write quick-and-dirty small programs, or *scripts*. I don't like the term "scripting language" as it carries a connotation that they cannot be used for building mission-critical software. Among interpreted languages Python is distinguished by its large and active *scientific computing* community. Adoption of Python for scientific computing in both industry applications and academic research has increased significantly since the early 2000s.

For data analysis and interactive, exploratory computing and data visualization, Python will inevitably draw comparisons with the many other domain-specific open source and commercial programming languages and tools in wide use, such as R, MATLAB, SAS, Stata, and others. In recent years, Python's improved library support (primarily pandas) has made it a strong alternative for data manipulation tasks. Combined with Python's strength in general purpose programming, it is an excellent choice as a single language for building data-centric applications.

## Python as Glue

Part of Python's success as a scientific computing platform is the ease of integrating C, C++, and FORTRAN code. Most modern computing environments share a similar set of legacy FORTRAN and C libraries for doing linear algebra, optimization, integration, fast fourier transforms, and other such algorithms. The same story has held true for many companies and national labs that have used Python to glue together 30 years' worth of legacy software.

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

In the last few years, the Cython project (<http://cython.org>) has become one of the preferred ways of both creating fast compiled extensions for Python and also interfacing with C and C++ code.

## Solving the "Two-Language" Problem

In many organizations, it is common to research, prototype, and test new ideas using a more domain-specific computing language like MATLAB or R then later port those

ideas to be part of a larger production system written in, say, Java, C#, or C++. What people are increasingly finding is that Python is a suitable language not only for doing research and prototyping but also building the production systems, too. I believe that more and more companies will go down this path as there are often significant organizational benefits to having both scientists and technologists using the same set of programmatic tools.

## Why Not Python?

While Python is an excellent environment for building computationally-intensive scientific applications and building most kinds of general purpose systems, there are a number of uses for which Python may be less suitable.

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

Python is not an ideal language for highly concurrent, multithreaded applications, particularly applications with many CPU-bound threads. The reason for this is that it has what is known as the *global interpreter lock* (GIL), a mechanism which prevents the interpreter from executing more than one Python bytecode instruction at a time. The technical reasons for why the GIL exists are beyond the scope of this book, but as of this writing it does not seem likely that the GIL will disappear anytime soon. While it is true that in many big data processing applications, a cluster of computers may be required to process a data set in a reasonable amount of time, there are still situations where a single-process, multithreaded system is desirable.

This is not to say that Python cannot execute truly multithreaded, parallel code; that code just cannot be executed in a single Python process. As an example, the Cython project features easy integration with OpenMP, a C framework for parallel computing, in order to parallelize loops and thus significantly speed up numerical algorithms.

## Essential Python Libraries

For those who are less familiar with the scientific Python ecosystem and the libraries used throughout the book, I present the following overview of each library.

## NumPy

NumPy, short for Numerical Python, is the foundational package for scientific computing in Python. The majority of this book will be based on NumPy and libraries built on top of NumPy. It provides, among other things:

- A fast and efficient multidimensional array object *ndarray*
- Functions for performing element-wise computations with arrays or mathematical operations between arrays
- Tools for reading and writing array-based data sets to disk
- Linear algebra operations, Fourier transform, and random number generation
- Tools for integrating connecting C, C++, and Fortran code to Python

Beyond the fast array-processing capabilities that NumPy adds to Python, one of its primary purposes with regards to data analysis is as the primary container for data to be passed between algorithms. For numerical data, NumPy arrays are a much more efficient way of storing and manipulating data than the other built-in Python data structures. Also, libraries written in a lower-level language, such as C or Fortran, can operate on the data stored in a NumPy array without copying any data.

## pandas

pandas provides rich data structures and functions designed to make working with structured data fast, easy, and expressive. It is, as you will see, one of the critical ingredients enabling Python to be a powerful and productive data analysis environment. The primary object in pandas that will be used in this book is the **DataFrame**, a two-dimensional tabular, column-oriented data structure with both row and column labels:

```
>>> frame
      total_bill  tip   sex   smoker  day    time    size
  1  16.99     1.01 Female  No  Sun Dinner    2
  2  10.34     1.66  Male   No  Sun Dinner    3
  3  21.01     3.50  Male   No  Sun Dinner    3
  4  23.68     3.31  Male   No  Sun Dinner    2
  5  24.59     3.61 Female  No  Sun Dinner    4
  6  25.29     4.71  Male   No  Sun Dinner    4
  7   8.77      2.00  Male   No  Sun Dinner    2
  8  26.88     3.12  Male   No  Sun Dinner    4
  9  15.04     1.96  Male   No  Sun Dinner    2
 10  14.78     3.23  Male   No  Sun Dinner    2
```

pandas combines the high performance array-computing features of NumPy with the flexible data manipulation capabilities of spreadsheets and relational databases (such as SQL). It provides sophisticated indexing functionality to make it easy to reshape, slice and dice, perform aggregations, and select subsets of data. pandas is the primary tool that we will use in this book.

For financial users, pandas features rich, high-performance time series functionality and tools well-suited for working with financial data. In fact, I initially designed pandas as an ideal tool for financial data analysis applications.

For users of the R language for statistical computing, the `DataFrame` name will be familiar, as the object was named after the similar R `data.frame` object. They are not the same, however; the functionality provided by `data.frame` in R is essentially a strict subset of that provided by the pandas `DataFrame`. While this is a book about Python, I will occasionally draw comparisons with R as it is one of the most widely-used open source data analysis environments and will be familiar to many readers.

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

## matplotlib

`matplotlib` is the most popular Python library for producing plots and other 2D data visualizations. It was originally created by John D. Hunter (JDH) and is now maintained by a large team of developers. It is well-suited for creating plots suitable for publication. It integrates well with IPython (see below), thus providing a comfortable interactive environment for plotting and exploring data. The plots are also *interactive*; you can zoom in on a section of the plot and pan around the plot using the toolbar in the plot window.

## IPython

IPython is the component in the standard scientific Python toolset that ties everything together. It provides a robust and productive environment for interactive and exploratory computing. It is an enhanced Python shell designed to accelerate the writing, testing, and debugging of Python code. It is particularly useful for interactively working with data and visualizing data with `matplotlib`. IPython is usually involved with the majority of my Python work, including running, debugging, and testing code.

Aside from the standard terminal-based IPython shell, the project also provides

- A Mathematica-like HTML notebook for connecting to IPython through a web browser (more on this later).
- A Qt framework-based GUI console with inline plotting, multiline editing, and syntax highlighting
- An infrastructure for interactive parallel and distributed computing

I will devote a chapter to IPython and how to get the most out of its features. I strongly recommend using it while working through this book.

## SciPy

SciPy is a collection of packages addressing a number of different standard problem domains in scientific computing. Here is a sampling of the packages included:

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

Together NumPy and SciPy form a reasonably complete computational replacement for much of MATLAB along with some of its add-on toolboxes.

## Installation and Setup

Since everyone uses Python for different applications, there is no single solution for setting up Python and required add-on packages. Many readers will not have a complete scientific Python environment suitable for following along with this book, so here I will give detailed instructions to get set up on each operating system. I recommend using one of the following base Python distributions:

- Enthought Python Distribution: a scientific-oriented Python distribution from Enthought (<http://www.enthought.com>). This includes EPDFree, a free base scientific distribution (with NumPy, SciPy, matplotlib, Chaco, and IPython) and EPD Full, a comprehensive suite of more than 100 scientific packages across many domains. EPD Full is free for academic use but has an annual subscription for non-academic users.
- Python(x,y) (<http://pythonxy.googlecode.com>): A free scientific-oriented Python distribution for Windows.

I will be using EPDFree for the installation guides, though you are welcome to take another approach depending on your needs. At the time of this writing, EPD includes Python 2.7, though this might change at some point in the future. After installing, you will have the following packages installed and importable:

- Scientific Python base: NumPy, SciPy, matplotlib, and IPython. These are all included in EPDFree.
- IPython Notebook dependencies: tornado and pyzmq. These are included in EPDFree.
- pandas (version 0.8.2 or higher).

At some point while reading you may wish to install one or more of the following packages: statsmodels, PyTables, PyQt (or equivalently, PySide), xlrd, lxml, basemap, pymongo, and requests. These are used in various examples. Installing these optional libraries is not necessary, and I would suggest waiting until you need them. For example, installing PyQt or PyTables from source on OS X or Linux can be rather arduous. For now, it's most important to get up and running with the bare minimum: EPDFree and pandas.

For information on each Python package and links to binary installers or other help, see the Python Package Index (PyPI, <http://pypi.python.org>). This is also an excellent resource for finding new Python packages.



To avoid confusion and to keep things simple, I am avoiding discussion of more complex environment management tools like pip and virtualenv. There are many excellent guides available for these tools on the Internet.



Some users may be interested in alternate Python implementations, such as IronPython, Jython, or PyPy. To make use of the tools presented in this book, it is (currently) necessary to use the standard C-based Python interpreter, known as CPython.

## Windows

To get started on Windows, download the EPDFree installer from <http://www.enthought.com>, which should be an MSI installer named like `epd_free-7.3-1-win-x86.msi`. Run the installer and accept the default installation location `C:\Python27`. If you had previously installed Python in this location, you may want to delete it manually first (or using Add/Remove Programs).

Next, you need to verify that Python has been successfully added to the system path and that there are no conflicts with any prior-installed Python versions. First, open a command prompt by going to the Start Menu and starting the Command Prompt application, also known as `cmd.exe`. Try starting the Python interpreter by typing `python`. You should see a message that matches the version of EPDFree you installed:

```
C:\Users\Wes>python
Python 2.7.3 |EPD_free 7.3-1 (32-bit)| (default, Apr 12 2012, 14:30:37) on win32
Type "credits", "demo" or "enthought" for more information.
>>>
```

If you see a message for a different version of EPD or it doesn't work at all, you will need to clean up your Windows environment variables. On Windows 7 you can start typing "environment variables" in the programs search field and select **Edit environment variables for your account**. On Windows XP, you will have to go to **Control Panel > System > Advanced > Environment Variables**. On the window that pops up, you are looking for the **Path** variable. It needs to contain the following two directory paths, separated by semicolons:

```
C:\Python27;C:\Python27\Scripts
```

If you installed other versions of Python, be sure to delete any other Python-related directories from both the system and user **Path** variables. After making a path alteration, you have to restart the command prompt for the changes to take effect.

Once you can launch Python successfully from the command prompt, you need to install pandas. The easiest way is to download the appropriate binary installer from <http://pypi.python.org/pypi/pandas>. For EPDFree, this should be **pandas-0.9.0.win32-py2.7.exe**. After you run this, let's launch IPython and check that things are installed correctly by importing pandas and making a simple matplotlib plot:

```
C:\Users\Wes>ipython --pylab
Python 2.7.3 |EPD_free 7.3-1 (32-bit)|
Type "copyright", "credits" or "license" for more information.

IPython 0.12.1 -- 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.

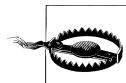
Welcome to pylab, a matplotlib-based Python environment [backend: WXAgg].
For more information, type 'help(pylab)'.

In [1]: import pandas

In [2]: plot(arange(10))
```

If successful, there should be no error messages and a plot window will appear. You can also check that the IPython HTML notebook can be successfully run by typing:

```
$ ipython notebook --pylab=inline
```



If you use the IPython notebook application on Windows and normally use Internet Explorer, you will likely need to install and run Mozilla Firefox or Google Chrome instead.

EPDFree on Windows contains only 32-bit executables. If you want or need a 64-bit setup on Windows, using EPD Full is the most painless way to accomplish that. If you would rather install from scratch and not pay for an EPD subscription, Christoph Gohlke at the University of California, Irvine, publishes unofficial binary installers for

all of the book's necessary packages (<http://www.lfd.uci.edu/~gohlke/pythonlibs/>) for 32- and 64-bit Windows.

## Apple OS X

To get started on OS X, you must first install Xcode, which includes Apple's suite of software development tools. The necessary component for our purposes is the gcc C and C++ compiler suite. The Xcode installer can be found on the OS X install DVD that came with your computer or downloaded from Apple directly.

Once you've installed Xcode, launch the terminal (Terminal.app) by navigating to **Applications > Utilities**. Type `gcc` and press enter. You should hopefully see something like:

```
$ gcc  
i686-apple-darwin10-gcc-4.2.1: no input files
```

Now you need to install EPDFree. Download the installer which should be a disk image named something like `epd_free-7.3-1-macosx-i386.dmg`. Double-click the `.dmg` file to mount it, then double-click the `.mpkg` file inside to run the installer.

When the installer runs, it automatically appends the EPDFree executable path to your `.bash_profile` file. This is located at `/Users/your_uname/.bash_profile`:

```
# Setting PATH for EPD_free-7.3-1  
PATH="/Library/Frameworks/Python.framework/Versions/Current/bin:${PATH}"  
export PATH
```

Should you encounter any problems in the following steps, you'll want to inspect your `.bash_profile` and potentially add the above directory to your path.

Now, it's time to install pandas. Execute this command in the terminal:

```
$ sudo easy_install pandas  
Searching for pandas  
Reading http://pypi.python.org/simple/pandas/  
Reading http://pandas.pydata.org  
Reading http://pandas.sourceforge.net  
Best match: pandas 0.9.0  
Downloading http://pypi.python.org/packages/source/p/pandas/pandas-0.9.0.zip  
Processing pandas-0.9.0.zip  
Writing /tmp/easy_install-H5mIX6/pandas-0.9.0/setup.cfg  
Running pandas-0.9.0/setup.py -q bdist_egg --dist-dir /tmp/easy_install-H5mIX6/  
pandas-0.9.0/egg-dist-tmp-RhLG0z  
Adding pandas 0.9.0 to easy-install.pth file  
  
Installed /Library/Frameworks/Python.framework/Versions/7.3/lib/python2.7/  
site-packages/pandas-0.9.0-py2.7-macosx-10.5-i386.egg  
Processing dependencies for pandas  
Finished processing dependencies for pandas
```

To verify everything is working, launch IPython in Pylab mode and test importing pandas then making a plot interactively:

```
$ ipython --pylab
22:29 ~/VirtualBox VMs/WindowsXP $ ipython
Python 2.7.3 |EPD_free 7.3-1 (32-bit)| (default, Apr 12 2012, 11:28:34)
Type "copyright", "credits" or "license" for more information.

IPython 0.12.1 -- 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.

Welcome to pylab, a matplotlib-based Python environment [backend: WXAgg].
For more information, type 'help(pylab)'.

In [1]: import pandas

In [2]: plot(arange(10))
```

If this succeeds, a plot window with a straight line should pop up.

## GNU/Linux



Some, but not all, Linux distributions include sufficiently up-to-date versions of all the required Python packages and can be installed using the built-in package management tool like `apt`. I detail setup using EPDFree as it's easily reproducible across distributions.

Linux details will vary a bit depending on your Linux flavor, but here I give details for Debian-based GNU/Linux systems like Ubuntu and Mint. Setup is similar to OS X with the exception of how EPDFree is installed. The installer is a shell script that must be executed in the terminal. Depending on whether you have a 32-bit or 64-bit system, you will either need to install the `x86` (32-bit) or `x86_64` (64-bit) installer. You will then have a file named something similar to `epd_free-7.3-1-rh5-x86_64.sh`. To install it, execute this script with bash:

```
$ bash epd_free-7.3-1-rh5-x86_64.sh
```

After accepting the license, you will be presented with a choice of where to put the EPDFree files. I recommend installing the files in your home directory, say `/home/wesm/epd` (substituting your own username for `wesm`).

Once the installer has finished, you need to add EPDFree's `bin` directory to your `$PATH` variable. If you are using the bash shell (the default in Ubuntu, for example), this means adding the following path addition in your `.bashrc`:

```
export PATH=/home/wesm/epd/bin:$PATH
```

Obviously, substitute the installation directory you used for `/home/wesm/epd/`. After doing this you can either start a new terminal process or execute your `.bashrc` again with `source ~/.bashrc`.

You need a C compiler such as gcc to move forward; many Linux distributions include gcc, but others may not. On Debian systems, you can install gcc by executing:

```
sudo apt-get install gcc
```

If you type `gcc` on the command line it should say something like:

```
$ gcc  
gcc: no input files
```

Now, time to install pandas:

```
$ easy_install pandas
```

If you installed EPDFree as root, you may need to add `sudo` to the command and enter the sudo or root password. To verify things are working, perform the same checks as in the OS X section.

## Python 2 and Python 3

The Python community is currently undergoing a drawn-out transition from the Python 2 series of interpreters to the Python 3 series. Until the appearance of Python 3.0, all Python code was backwards compatible. The community decided that in order to move the language forward, certain backwards incompatible changes were necessary.

I am writing this book with Python 2.7 as its basis, as the majority of the scientific Python community has not yet transitioned to Python 3. The good news is that, with a few exceptions, you should have no trouble following along with the book if you happen to be using Python 3.2.

## Integrated Development Environments (IDEs)

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

However, some will still prefer to work in an IDE instead of a text editor. They do provide many nice “code intelligence” features like completion or quickly pulling up the documentation associated with functions and classes. Here are some that you can explore:

- Eclipse with PyDev Plugin
- Python Tools for Visual Studio (for Windows users)
- PyCharm
- Spyder
- Komodo IDE

## Community and Conferences

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

- pydata: a Google Group list for questions related to Python for data analysis and pandas
- pystatsmodels: for statsmodels or pandas-related questions
- numpy-discussion: for NumPy-related questions
- scipy-user: for general SciPy or scientific Python questions

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

Each year many conferences are held all over the world for Python programmers. PyCon and EuroPython are the two main general Python conferences in the United States and Europe, respectively. SciPy and EuroSciPy are scientific-oriented Python conferences where you will likely find many “birds of a feather” if you become more involved with using Python for data analysis after reading this book.

## Navigating This Book

If you have never programmed in Python before, you may actually want to start at the *end* of the book, where I have placed a condensed tutorial on Python syntax, language features, and built-in data structures like tuples, lists, and dicts. These things are considered prerequisite knowledge for the remainder of the book.

The book starts by introducing you to the IPython environment. Next, I give a short introduction to the key features of NumPy, leaving more advanced NumPy use for another chapter at the end of the book. Then, I introduce pandas and devote the rest of the book to data analysis topics applying pandas, NumPy, and matplotlib (for visualization). I have structured the material in the most incremental way possible, though there is occasionally some minor cross-over between chapters.

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

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

I encourage you to download the data and use it to replicate the book’s code examples and experiment with the tools presented in each chapter. I will happily accept contributions, scripts, IPython notebooks, or any other materials you wish to contribute to the book’s repository for all to enjoy.

## Code Examples

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

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

At times, for clarity, multiple code examples will be shown side by side. These should be read left to right and executed separately.

```
In [5]: code           In [6]: code2  
Out[5]: output        Out[6]: output2
```

## Data for Examples

Data sets for the examples in each chapter are hosted in a repository on GitHub: <http://github.com/pydata/pydata-book>. You can download this data either by using the git revision control command-line program or by downloading a zip file of the repository from the website.

I have made every effort to ensure that it contains everything necessary to reproduce the examples, but I may have made some mistakes or omissions. If so, please send me an e-mail: [wesmckinn@gmail.com](mailto:wesmckinn@gmail.com).

## Import Conventions

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

```
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt
```

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

## Jargon

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

### *Munge/Munging/Wrangling*

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

### *Pseudocode*

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

### *Syntactic sugar*

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

## Acknowledgements

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

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

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

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

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

I'd also like to thank a number of people from my formative years. First, my former AQR colleagues who've cheered me on in my pandas work over the years: Alex Reyfman, Michael Wong, Tim Sargent, Oktay Kurbanov, Matthew Tschantz, Roni Israelov, Michael Katz, Chris Uga, Prasad Ramanan, Ted Square, and Hoon Kim. Lastly, my academic advisors Haynes Miller (MIT) and Mike West (Duke).

On the personal side, Casey Dinkin provided invaluable day-to-day support during the writing process, tolerating my highs and lows as I hacked together the final draft on

top of an already overcommitted schedule. Lastly, my parents, Bill and Kim, taught me to always follow my dreams and to never settle for less.



# Introductory Examples

This book teaches you the Python tools to work productively with data. While readers may have many different end goals for their work, the tasks required generally fall into a number of different broad groups:

### *Interacting with the outside world*

Reading and writing with a variety of file formats and databases.

### *Preparation*

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

### *Transformation*

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

### *Modeling and computation*

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

### *Presentation*

Creating interactive or static graphical visualizations or textual summaries

In this chapter I will show you a few data sets and some things we can do with them. These examples are just intended to pique your interest and thus will only be explained at a high level. Don't worry if you have no experience with any of these tools; they will be discussed in great detail throughout the rest of the book. In the code examples you'll see input and output prompts like `In [15]:`; these are from the IPython shell.

## 1.usa.gov data from bit.ly

In 2011, URL shortening service bit.ly partnered with the United States government website `usa.gov` to provide a feed of anonymous data gathered from users who shorten links ending with `.gov` or `.mil`. As of this writing, in addition to providing a live feed, hourly snapshots are available as downloadable text files.<sup>1</sup>

In the case of the hourly snapshots, each line in each file contains a common form of web data known as JSON, which stands for JavaScript Object Notation. For example, if we read just the first line of a file you may see something like

```
In [15]: path = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'

In [16]: open(path).readline()
Out[16]: '{ "a": "Mozilla\\/5.0 (Windows NT 6.1; WOW64) AppleWebKit\\/535.11
(KHTML, like Gecko) Chrome\\/17.0.963.78 Safari\\/535.11", "c": "US", "nk": 1,
"tz": "America\\/New_York", "gr": "MA", "g": "A6qOVH", "h": "wfLQtf", "l":
"orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r":
"http://www.facebook.com/1/7AQEFzjSi/1.usa.gov/wfLQtf", "u":
"http://www.ncbi.nlm.nih.gov/pubmed/22415991", "t": 1331923247, "hc":
1331822918, "cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }\\n'
```

Python has numerous built-in and 3rd party modules for converting a JSON string into a Python dictionary object. Here I'll use the `json` module and its `loads` function invoked on each line in the sample file I downloaded:

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

If you've never programmed in Python before, the last expression here is called a *list comprehension*, which is a concise way of applying an operation (like `json.loads`) to a collection of strings or other objects. Conveniently, iterating over an open file handle gives you a sequence of its lines. The resulting object `records` is now a list of Python dicts:

```
In [18]: records[0]
Out[18]:
{u'a': u'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like
Gecko) Chrome/17.0.963.78 Safari/535.11',
 u'al': u'en-US,en;q=0.8',
 u'c': u'US',
 u'cy': u'Danvers',
 u'g': u'A6qOVH',
 u'gr': u'MA',
 u'h': u>wfLQtf',
 u'hc': 1331822918,
 u'hh': u'1.usa.gov',
 u'l': u'orofrog',
 u'll': [42.576698, -70.954903],
 u'nk': 1,
 u'r': u'http://www.facebook.com/1/7AQEFzjSi/1.usa.gov/wfLQtf',
 u't': 1331923247,
 u'tz': u'America/New_York',
 u'u': u'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

1. <http://www.usa.gov/About/developer-resources/usagov.shtml>

Note that Python indices start at 0 and not 1 like some other languages (like R). It's now easy to access individual values within records by passing a string for the key you wish to access:

```
In [19]: records[0]['tz']
Out[19]: u'America/New_York'
```

The `u` here in front of the quotation stands for *unicode*, a standard form of string encoding. Note that IPython shows the time zone string object *representation* here rather than its print equivalent:

```
In [20]: print records[0]['tz']
America/New_York
```

## Counting Time Zones in Pure Python

Suppose we were interested in the most often-occurring time zones in the data set (the `tz` field). There are many ways we could do this. First, let's extract a list of time zones again using a list comprehension:

```
In [25]: time_zones = [rec['tz'] for rec in records]
-----
KeyError                                                 Traceback (most recent call last)
/home/wesm/book_scripts/whetting/<ipython> in <module>()
----> 1 time_zones = [rec['tz'] for rec in records]

KeyError: 'tz'
```

Oops! Turns out that not all of the records have a time zone field. This is easy to handle as we can add the check `if 'tz' in rec` at the end of the list comprehension:

```
In [26]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]

In [27]: time_zones[:10]
Out[27]:
[u'America/New_York',
 u'America/Denver',
 u'America/New_York',
 u'America/Sao_Paulo',
 u'America/New_York',
 u'America/New_York',
 u'Europe/Warsaw',
 u '',
 u '',
 u '']
```

Just looking at the first 10 time zones we see that some of them are unknown (empty). You can filter these out also but I'll leave them in for now. Now, to produce counts by time zone I'll show two approaches: the harder way (using just the Python standard library) and the easier way (using pandas). One way to do the counting is to use a dict to store counts while we iterate through the time zones:

```
def get_counts(sequence):
    counts = {}
```

```

for x in sequence:
    if x in counts:
        counts[x] += 1
    else:
        counts[x] = 1
return counts

```

If you know a bit more about the Python standard library, you might prefer to write the same thing more briefly:

```

from collections import defaultdict

def get_counts2(sequence):
    counts = defaultdict(int) # values will initialize to 0
    for x in sequence:
        counts[x] += 1
    return counts

```

I put this logic in a function just to make it more reusable. To use it on the time zones, just pass the `time_zones` list:

```

In [31]: counts = get_counts(time_zones)

In [32]: counts['America/New_York']
Out[32]: 1251

In [33]: len(time_zones)
Out[33]: 3440

```

If we wanted the top 10 time zones and their counts, we have to do a little bit of dictionary acrobatics:

```

def top_counts(count_dict, n=10):
    value_key_pairs = [(count, tz) for tz, count in count_dict.items()]
    value_key_pairs.sort()
    return value_key_pairs[-n:]

```

We have then:

```

In [35]: top_counts(counts)
Out[35]:
[(33, u'America/Sao_Paulo'),
 (35, u'Europe/Madrid'),
 (36, u'Pacific/Honolulu'),
 (37, u'Asia/Tokyo'),
 (74, u'Europe/London'),
 (191, u'America/Denver'),
 (382, u'America/Los_Angeles'),
 (400, u'America/Chicago'),
 (521, u''),
 (1251, u'America/New_York')]

```

If you search the Python standard library, you may find the `collections.Counter` class, which makes this task a lot easier:

```
In [49]: from collections import Counter
```

```
In [50]: counts = Counter(time_zones)
```

```
In [51]: counts.most_common(10)
```

```
Out[51]:
```

```
[('America/New_York', 1251),  
 ('', 521),  
 ('America/Chicago', 400),  
 ('America/Los_Angeles', 382),  
 ('America/Denver', 191),  
 ('Europe/London', 74),  
 ('Asia/Tokyo', 37),  
 ('Pacific/Honolulu', 36),  
 ('Europe/Madrid', 35),  
 ('America/Sao_Paulo', 33)]
```

## Counting Time Zones with pandas

The main pandas data structure is the `DataFrame`, which you can think of as representing a table or spreadsheet of data. Creating a DataFrame from the original set of records is simple:

```
In [289]: from pandas import DataFrame, Series
```

```
In [290]: import pandas as pd
```

```
In [291]: frame = DataFrame(records)
```

```
In [292]: frame
```

```
Out[292]:
```

```
<class 'pandas.core.frame.DataFrame'>  
Int64Index: 3560 entries, 0 to 3559  
Data columns:  
 _heartbeat_    120 non-null values  
 a            3440 non-null values  
 al           3094 non-null values  
 c            2919 non-null values  
 cy           2919 non-null values  
 g            3440 non-null values  
 gr          2919 non-null values  
 h            3440 non-null values  
 hc          3440 non-null values  
 hh          3440 non-null values  
 kw           93 non-null values  
 l            3440 non-null values  
 ll          2919 non-null values  
 nk          3440 non-null values  
 r            3440 non-null values  
 t            3440 non-null values  
 tz          3440 non-null values
```

```
u           3440 non-null values
dtypes: float64(4), object(14)

In [293]: frame['tz'][:10]
Out[293]:
0    America/New_York
1    America/Denver
2    America/New_York
3    America/Sao_Paulo
4    America/New_York
5    America/New_York
6    Europe/Warsaw
7
8
9
Name: tz
```

The output shown for the `frame` is the *summary view*, shown for large DataFrame objects. The Series object returned by `frame['tz']` has a method `value_counts` that gives us what we're looking for:

```
In [294]: tz_counts = frame['tz'].value_counts()

In [295]: tz_counts[:10]
Out[295]:
America/New_York      1251
                     521
America/Chicago        400
America/Los_Angeles   382
America/Denver         191
Europe/London          74
Asia/Tokyo             37
Pacific/Honolulu       36
Europe/Madrid          35
America/Sao_Paulo      33
```

Then, we might want to make a plot of this data using plotting library, matplotlib. You can do a bit of munging to fill in a substitute value for unknown and missing time zone data in the records. The `fillna` function can replace missing (NA) values and unknown (empty strings) values can be replaced by boolean array indexing:

```
In [296]: clean_tz = frame['tz'].fillna('Missing')

In [297]: clean_tz[clean_tz == ''] = 'Unknown'

In [298]: tz_counts = clean_tz.value_counts()

In [299]: tz_counts[:10]
Out[299]:
America/New_York      1251
Unknown                521
America/Chicago        400
America/Los_Angeles   382
America/Denver         191
Missing                120
```

Europe/London	74
Asia/Tokyo	37
Pacific/Honolulu	36
Europe/Madrid	35

Making a horizontal bar plot can be accomplished using the `plot` method on the `counts` objects:

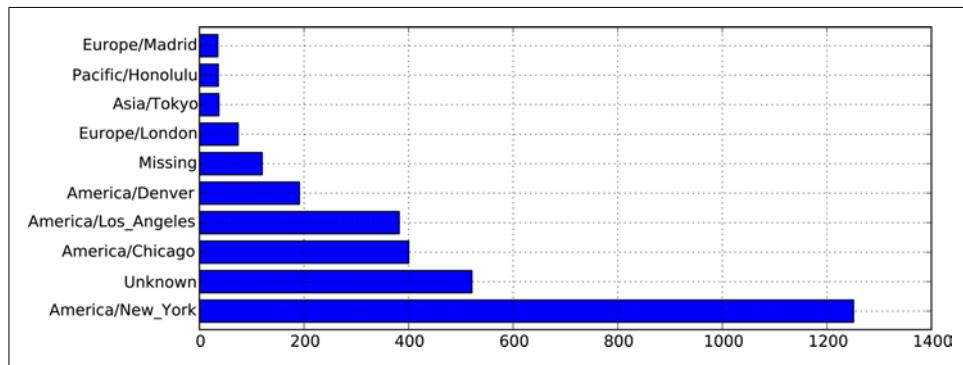
```
In [301]: tz_counts[:10].plot(kind='barh', rot=0)
```

See [Figure 2-1](#) for the resulting figure. We'll explore more tools for working with this kind of data. For example, the `a` field contains information about the browser, device, or application used to perform the URL shortening:

```
In [302]: frame['a'][1]
Out[302]: u'GoogleMaps/RochesterNY'
```

```
In [303]: frame['a'][50]
Out[303]: u'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Firefox/10.0.2'
```

```
In [304]: frame['a'][51]
Out[304]: u'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P925/V10e Build/FRG83G) AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0.2 Safari/533.1'
```



*Figure 2-1. Top time zones in the 1.usa.gov sample data*

Parsing all of the interesting information in these “agent” strings may seem like a daunting task. Luckily, once you have mastered Python’s built-in string functions and regular expression capabilities, it is really not so bad. For example, we could split off the first token in the string (corresponding roughly to the browser capability) and make another summary of the user behavior:

```
In [305]: results = Series([x.split()[0] for x in frame.a.dropna()])
```

```
In [306]: results[:5]
Out[306]:
0      Mozilla/5.0
1  GoogleMaps/RochesterNY
2      Mozilla/4.0
3      Mozilla/5.0
4      Mozilla/5.0
```

```
In [307]: results.value_counts()[:8]
Out[307]:
Mozilla/5.0           2594
Mozilla/4.0            601
GoogleMaps/RochesterNY 121
Opera/9.80              34
TEST_INTERNET_AGENT      24
GoogleProducer            21
Mozilla/6.0                5
BlackBerry8520/5.0.0.681    4
```

Now, suppose you wanted to decompose the top time zones into Windows and non-Windows users. As a simplification, let's say that a user is on Windows if the string 'Windows' is in the agent string. Since some of the agents are missing, I'll exclude these from the data:

```
In [308]: cframe = frame[frame.a.notnull()]
```

We want to then compute a value whether each row is Windows or not:

```
In [309]: operating_system = np.where(cframe['a'].str.contains('Windows'),
.....                           'Windows', 'Not Windows')
```

```
In [310]: operating_system[:5]
Out[310]:
0      Windows
1    Not Windows
2      Windows
3    Not Windows
4      Windows
Name: a
```

Then, you can group the data by its time zone column and this new list of operating systems:

```
In [311]: by_tz_os = cframe.groupby(['tz', operating_system])
```

The group counts, analogous to the `value_counts` function above, can be computed using `size`. This result is then reshaped into a table with `unstack`:

```
In [312]: agg_counts = by_tz_os.size().unstack().fillna(0)
```

```
In [313]: agg_counts[:10]
Out[313]:
a                  Not Windows  Windows
tz
Africa/Cairo          0         3
Africa/Casablanca      0         1
Africa/Ceuta            0         2
Africa/Johannesburg     0         1
Africa/Lusaka            0         1
America/Anchorage        4         1
America/Argentina/Buenos_Aires 1         0
```

America/Argentina/Cordoba	0	1
America/Argentina/Mendoza	0	1

Finally, let's select the top overall time zones. To do so, I construct an indirect index array from the row counts in `agg_counts`:

```
# Use to sort in ascending order
In [314]: indexer = agg_counts.sum(1).argsort()

In [315]: indexer[:10]
Out[315]:
tz
24
Africa/Cairo      20
Africa/Casablanca 21
Africa/Ceuta       92
Africa/Johannesburg 87
Africa/Lusaka      53
America/Anchorage 54
America/Argentina/Buenos_Aires 57
America/Argentina/Cordoba 26
America/Argentina/Mendoza 55
```

I then use `take` to select the rows in that order, then slice off the last 10 rows:

```
In [316]: count_subset = agg_counts.take(indexer)[-10:]

In [317]: count_subset
Out[317]:
a           Not Windows  Windows
tz
America/Sao_Paulo      13      20
Europe/Madrid          16      19
Pacific/Honolulu        0      36
Asia/Tokyo              2      35
Europe/London          43      31
America/Denver          132     59
America/Los_Angeles    130     252
America/Chicago          115     285
                           245     276
America/New_York         339     912
```

Then, as shown in the preceding code block, this can be plotted in a bar plot; I'll make it a stacked bar plot by passing `stacked=True` (see [Figure 2-2](#)) :

```
In [319]: count_subset.plot(kind='barh', stacked=True)
```

The plot doesn't make it easy to see the relative percentage of Windows users in the smaller groups, but the rows can easily be normalized to sum to 1 then plotted again (see [Figure 2-3](#)):

```
In [321]: normed_subset = count_subset.div(count_subset.sum(1), axis=0)
```

```
In [322]: normed_subset.plot(kind='barh', stacked=True)
```

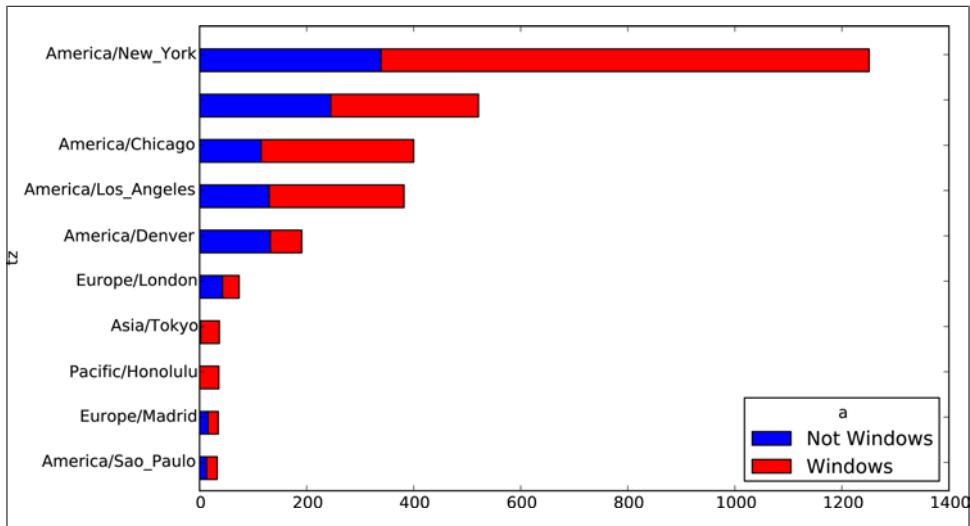


Figure 2-2. Top time zones by Windows and non-Windows users

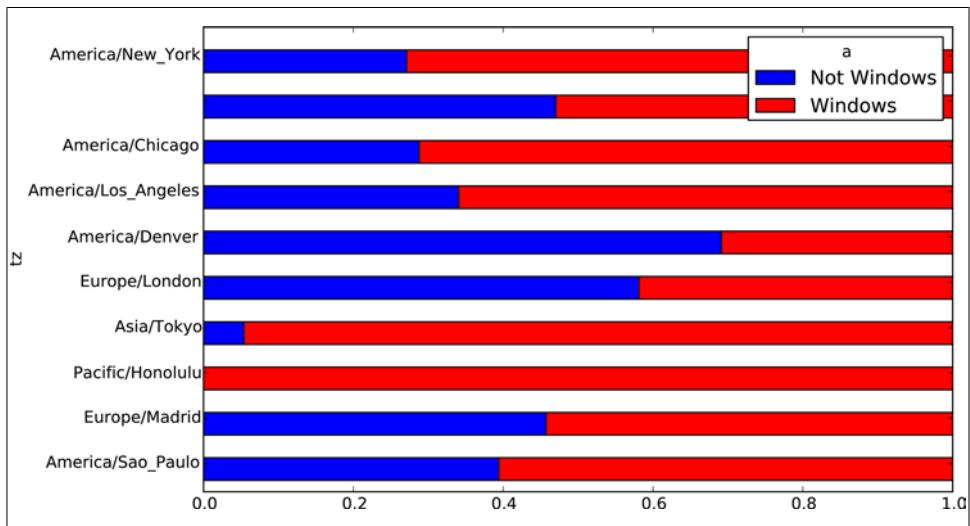


Figure 2-3. Percentage Windows and non-Windows users in top-occurring time zones

All of the methods employed here will be examined in great detail throughout the rest of the book.

## MovieLens 1M Data Set

GroupLens Research (<http://www.grouplens.org/node/73>) provides a number of collections of movie ratings data collected from users of MovieLens in the late 1990s and

early 2000s. The data provide movie ratings, movie metadata (genres and year), and demographic data about the users (age, zip code, gender, and occupation). Such data is often of interest in the development of recommendation systems based on machine learning algorithms. While I will not be exploring machine learning techniques in great detail in this book, I will show you how to slice and dice data sets like these into the exact form you need.

The MovieLens 1M data set contains 1 million ratings collected from 6000 users on 4000 movies. It's spread across 3 tables: ratings, user information, and movie information. After extracting the data from the zip file, each table can be loaded into a pandas DataFrame object using `pandas.read_table`:

```
import pandas as pd

unames = ['user_id', 'gender', 'age', 'occupation', 'zip']
users = pd.read_table('ml-1m/users.dat', sep='::', header=None,
                      names=unames)

rnames = ['user_id', 'movie_id', 'rating', 'timestamp']
ratings = pd.read_table('ml-1m/ratings.dat', sep='::', header=None,
                        names=rnames)

mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('ml-1m/movies.dat', sep='::', header=None,
                       names=mnames)
```

You can verify that everything succeeded by looking at the first few rows of each DataFrame with Python's slice syntax:

```
In [334]: users[:5]
Out[334]:
   user_id  gender  age  occupation    zip
0         1      F    1           10  48067
1         2      M   56           16  70072
2         3      M   25           15  55117
3         4      M   45            7  02460
4         5      M   25           20  55455

In [335]: ratings[:5]
Out[335]:
   user_id  movie_id  rating  timestamp
0         1       1193      5  978300760
1         1        661      3  978302109
2         1        914      3  978301968
3         1       3408      4  978300275
4         1       2355      5  978824291

In [336]: movies[:5]
Out[336]:
   movie_id          title                genres
0         1  Toy Story (1995)  Animation|Children's|Comedy
1         2     Jumanji (1995)  Adventure|Children's|Fantasy
2         3  Grumpier Old Men (1995)  Comedy|Romance
3         4  Waiting to Exhale (1995)  Comedy|Drama
```

```
In [337]: ratings
Out[337]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000209 entries, 0 to 1000208
Data columns:
user_id      1000209 non-null values
movie_id     1000209 non-null values
rating       1000209 non-null values
timestamp    1000209 non-null values
dtypes: int64(4)
```

Note that ages and occupations are coded as integers indicating groups described in the data set's README file. Analyzing the data spread across three tables is not a simple task; for example, suppose you wanted to compute mean ratings for a particular movie by sex and age. As you will see, this is much easier to do with all of the data merged together into a single table. Using pandas's `merge` function, we first merge `ratings` with `users` then merging that result with the `movies` data. pandas infers which columns to use as the merge (or *join*) keys based on overlapping names:

```
In [338]: data = pd.merge(pd.merge(ratings, users), movies)
```

```
In [339]: data
Out[339]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000209 entries, 0 to 1000208
Data columns:
user_id      1000209 non-null values
movie_id     1000209 non-null values
rating       1000209 non-null values
timestamp    1000209 non-null values
gender       1000209 non-null values
age          1000209 non-null values
occupation   1000209 non-null values
zip          1000209 non-null values
title        1000209 non-null values
genres       1000209 non-null values
dtypes: int64(6), object(4)
```

```
In [340]: data.ix[0]
Out[340]:
user_id              1
movie_id             1
rating               5
timestamp            978824268
gender               F
age                  1
occupation           10
zip                 48067
title                Toy Story (1995)
genres               Animation|Children's|Comedy
Name: 0
```

In this form, aggregating the ratings grouped by one or more user or movie attributes is straightforward once you build some familiarity with pandas. To get mean movie ratings for each film grouped by gender, we can use the `pivot_table` method:

```
In [341]: mean_ratings = data.pivot_table('rating', rows='title',
.....:                                         cols='gender', aggfunc='mean')

In [342]: mean_ratings[:5]
Out[342]:
   gender          F      M
   title
$1,000,000 Duck (1971)  3.375000  2.761905
'Night Mother (1986)    3.388889  3.352941
'Til There Was You (1997) 2.675676  2.733333
'burbs, The (1989)     2.793478  2.962085
...And Justice for All (1979) 3.828571  3.689024
```

This produced another DataFrame containing mean ratings with movie totals as row labels and gender as column labels. First, I'm going to filter down to movies that received at least 250 ratings (a completely arbitrary number); to do this, I group the data by title and use `size()` to get a Series of group sizes for each title:

```
In [343]: ratings_by_title = data.groupby('title').size()

In [344]: ratings_by_title[:10]
Out[344]:
   title
$1,000,000 Duck (1971)      37
'Night Mother (1986)        70
'Til There Was You (1997)    52
'burbs, The (1989)         303
...And Justice for All (1979) 199
1-900 (1994)                 2
10 Things I Hate About You (1999) 700
101 Dalmatians (1961)       565
101 Dalmatians (1996)       364
12 Angry Men (1957)         616

In [345]: active_titles = ratings_by_title.index[ratings_by_title >= 250]

In [346]: active_titles
Out[346]:
Index(['burbs, The (1989), 10 Things I Hate About You (1999),
       101 Dalmatians (1961), ..., Young Sherlock Holmes (1985),
       Zero Effect (1998), eXistenZ (1999)], dtype=object)
```

The index of titles receiving at least 250 ratings can then be used to select rows from `mean_ratings` above:

```
In [347]: mean_ratings = mean_ratings.ix[active_titles]

In [348]: mean_ratings
Out[348]:
<class 'pandas.core.frame.DataFrame'>
Index: 1216 entries, 'burbs, The (1989) to eXistenZ (1999)
```

```
Data columns:  
F    1216 non-null values  
M    1216 non-null values  
dtypes: float64(2)
```

To see the top films among female viewers, we can sort by the F column in descending order:

```
In [350]: top_female_ratings = mean_ratings.sort_index(by='F', ascending=False)
```

```
In [351]: top_female_ratings[:10]
```

```
Out[351]:
```

gender	F	M
Close Shave, A (1995)	4.644444	4.473795
Wrong Trousers, The (1993)	4.588235	4.478261
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)	4.572650	4.464589
Wallace & Gromit: The Best of Aardman Animation (1996)	4.563107	4.385075
Schindler's List (1993)	4.562602	4.491415
Shawshank Redemption, The (1994)	4.539075	4.560625
Grand Day Out, A (1992)	4.537879	4.293255
To Kill a Mockingbird (1962)	4.536667	4.372611
Creature Comforts (1990)	4.513889	4.272277
Usual Suspects, The (1995)	4.513317	4.518248

## Measuring rating disagreement

Suppose you wanted to find the movies that are most divisive between male and female viewers. One way is to add a column to `mean_ratings` containing the difference in means, then sort by that:

```
In [352]: mean_ratings['diff'] = mean_ratings['M'] - mean_ratings['F']
```

Sorting by 'diff' gives us the movies with the greatest rating difference and which were preferred by women:

```
In [353]: sorted_by_diff = mean_ratings.sort_index(by='diff')
```

```
In [354]: sorted_by_diff[:15]
```

```
Out[354]:
```

gender	F	M	diff
Dirty Dancing (1987)	3.790378	2.959596	-0.830782
Jumpin' Jack Flash (1986)	3.254717	2.578358	-0.676359
Grease (1978)	3.975265	3.367041	-0.608224
Little Women (1994)	3.870588	3.321739	-0.548849
Steel Magnolias (1989)	3.901734	3.365957	-0.535777
Anastasia (1997)	3.800000	3.281609	-0.518391
Rocky Horror Picture Show, The (1975)	3.673016	3.160131	-0.512885
Color Purple, The (1985)	4.158192	3.659341	-0.498851
Age of Innocence, The (1993)	3.827068	3.339506	-0.487561
Free Willy (1993)	2.921348	2.438776	-0.482573
French Kiss (1995)	3.535714	3.056962	-0.478752
Little Shop of Horrors, The (1960)	3.650000	3.179688	-0.470312
Guys and Dolls (1955)	4.051724	3.583333	-0.468391
Mary Poppins (1964)	4.197740	3.730594	-0.467147
Patch Adams (1998)	3.473282	3.008746	-0.464536

# Time Series

Time series data is an important form of structured data in many different fields, such as finance, economics, ecology, neuroscience, or physics. Anything that is observed or measured at many points in time forms a time series. Many time series are *fixed frequency*, which is to say that data points occur at regular intervals according to some rule, such as every 15 seconds, every 5 minutes, or once per month. Time series can also be *irregular* without a fixed unit or time or offset between units. How you mark and refer to time series data depends on the application and you may have one of the following:

- *Timestamps*, specific instants in time
- Fixed *periods*, such as the month January 2007 or the full year 2010
- *Intervals* of time, indicated by a start and end timestamp. Periods can be thought of as special cases of intervals
- *Elapsed time*; each timestamp is a measure of time relative to a particular start time. For example, the diameter of a cookie baking each second since being placed in the oven

In this chapter, I am mainly concerned with time series in the first 3 categories, though many of the techniques can be applied to experimental time series where the index may be an integer or floating point number indicating elapsed time from the start of the experiment. The simplest and most widely used kind of time series are those indexed by timestamp.

pandas provides a standard set of time series tools and data algorithms. With this, you can efficiently work with very large time series and easily slice and dice, aggregate, and resample irregular and fixed frequency time series. As you might guess, many of these tools are especially useful for financial and economics applications, but you could certainly use them to analyze server log data, too.



Some of the features and code, in particular period logic, presented in this chapter were derived from the now defunct `scikits.timeseries` library.

## Date and Time Data Types and Tools

The Python standard library includes data types for date and time data, as well as calendar-related functionality. The `datetime`, `time`, and `calendar` modules are the main places to start. The `datetime.datetime` type, or simply `datetime`, is widely used:

```
In [317]: from datetime import datetime  
  
In [318]: now = datetime.now()  
  
In [319]: now  
Out[319]: datetime.datetime(2012, 8, 4, 17, 9, 21, 832092)  
  
In [320]: now.year, now.month, now.day  
Out[320]: (2012, 8, 4)
```

`datetime` stores both the date and time down to the microsecond. `datetime.time` `delta` represents the temporal difference between two `datetime` objects:

```
In [321]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)  
  
In [322]: delta  
Out[322]: datetime.timedelta(926, 56700)  
  
In [323]: delta.days           In [324]: delta.seconds  
Out[323]: 926                 Out[324]: 56700
```

You can add (or subtract) a `timedelta` or multiple thereof to a `datetime` object to yield a new shifted object:

```
In [325]: from datetime import timedelta  
  
In [326]: start = datetime(2011, 1, 7)  
  
In [327]: start + timedelta(12)  
Out[327]: datetime.datetime(2011, 1, 19, 0, 0)  
  
In [328]: start - 2 * timedelta(12)  
Out[328]: datetime.datetime(2010, 12, 14, 0, 0)
```

The data types in the `datetime` module are summarized in [Table 10-1](#). While this chapter is mainly concerned with the data types in pandas and higher level time series manipulation, you will undoubtedly encounter the `datetime`-based types in many other places in Python the wild.

Table 10-1. Types in `datetime` module

Type	Description
<code>date</code>	Store calendar date (year, month, day) using the Gregorian calendar.
<code>time</code>	Store time of day as hours, minutes, seconds, and microseconds
<code>datetime</code>	Stores both date and time
<code>timedelta</code>	Represents the difference between two datetime values (as days, seconds, and microseconds)

## Converting between string and `datetime`

`datetime` objects and pandas `Timestamp` objects, which I'll introduce later, can be formatted as strings using `str` or the `strftime` method, passing a format specification:

```
In [329]: stamp = datetime(2011, 1, 3)

In [330]: str(stamp)
Out[330]: '2011-01-03 00:00:00'

In [331]: stamp.strftime('%Y-%m-%d')
Out[331]: '2011-01-03'
```

See [Table 10-2](#) for a complete list of the format codes. These same format codes can be used to convert strings to dates using `datetime.strptime`:

```
In [332]: value = '2011-01-03'

In [333]: datetime.strptime(value, '%Y-%m-%d')
Out[333]: datetime.datetime(2011, 1, 3, 0, 0)

In [334]: datestrs = ['7/6/2011', '8/6/2011']

In [335]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]
Out[335]: [datetime.datetime(2011, 7, 6, 0, 0), datetime.datetime(2011, 8, 6, 0, 0)]
```

`datetime.strptime` is the best way to parse a date with a known format. However, it can be a bit annoying to have to write a format spec each time, especially for common date formats. In this case, you can use the `parser.parse` method in the third party `dateutil` package:

```
In [336]: from dateutil.parser import parse

In [337]: parse('2011-01-03')
Out[337]: datetime.datetime(2011, 1, 3, 0, 0)
```

`dateutil` is capable of parsing almost any human-intelligible date representation:

```
In [338]: parse('Jan 31, 1997 10:45 PM')
Out[338]: datetime.datetime(1997, 1, 31, 22, 45)
```

In international locales, day appearing before month is very common, so you can pass `dayfirst=True` to indicate this:

```
In [339]: parse('6/12/2011', dayfirst=True)
Out[339]: datetime.datetime(2011, 12, 6, 0, 0)
```

pandas is generally oriented toward working with arrays of dates, whether used as an axis index or a column in a DataFrame. The `to_datetime` method parses many different kinds of date representations. Standard date formats like ISO8601 can be parsed very quickly.

```
In [340]: datestrs  
Out[340]: ['7/6/2011', '8/6/2011']  
  
In [341]: pd.to_datetime(datestrs)  
Out[341]:  
<class 'pandas.tseries.index.DatetimeIndex'>  
[2011-07-06 00:00:00, 2011-08-06 00:00:00]  
Length: 2, Freq: None, Timezone: None
```

It also handles values that should be considered missing (`None`, empty string, etc.):

```
In [342]: idx = pd.to_datetime(datestrs + [None])  
  
In [343]: idx  
Out[343]:  
<class 'pandas.tseries.index.DatetimeIndex'>  
[2011-07-06 00:00:00, ..., NaT]  
Length: 3, Freq: None, Timezone: None  
  
In [344]: idx[2]  
Out[344]: NaT  
  
In [345]: pd.isnull(idx)  
Out[345]: array([False, False, True], dtype=bool)
```

`NaT` (Not a Time) is pandas's NA value for timestamp data.



`dateutil.parser` is a useful, but not perfect tool. Notably, it will recognize some strings as dates that you might prefer that it didn't, like '`42`' will be parsed as the year `2042` with today's calendar date.

Table 10-2. Datetime format specification (ISO C89 compatible)

Type	Description
%Y	4-digit year
%y	2-digit year
%m	2-digit month [01, 12]
%d	2-digit day [01, 31]
%H	Hour (24-hour clock) [00, 23]
%I	Hour (12-hour clock) [01, 12]
%M	2-digit minute [00, 59]
%S	Second [00, 61] (seconds 60, 61 account for leap seconds)
%w	Weekday as integer [0 (Sunday), 6]

Type	Description
%U	Week number of the year [00, 53]. Sunday is considered the first day of the week, and days before the first Sunday of the year are “week 0”.
%W	Week number of the year [00, 53]. Monday is considered the first day of the week, and days before the first Monday of the year are “week 0”.
%z	UTC time zone offset as +HHMM or -HHMM, empty if time zone naive
%F	Shortcut for %Y-%m-%d, for example 2012-4-18
%D	Shortcut for %m/%d/%y, for example 04/18/12

`datetime` objects also have a number of locale-specific formatting options for systems in other countries or languages. For example, the abbreviated month names will be different on German or French systems compared with English systems.

Table 10-3. Locale-specific date formatting

Type	Description
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Full date and time, for example ‘Tue 01 May 2012 04:20:57 PM’
%p	Locale equivalent of AM or PM
%x	Locale-appropriate formatted date; e.g. in US May 1, 2012 yields ‘05/01/2012’
%X	Locale-appropriate time, e.g. ‘04:24:12 PM’

## Time Series Basics

The most basic kind of time series object in pandas is a Series indexed by timestamps, which is often represented external to pandas as Python strings or `datetime` objects:

```
In [346]: from datetime import datetime
In [347]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5), datetime(2011, 1, 7),
.....:             datetime(2011, 1, 8), datetime(2011, 1, 10), datetime(2011, 1, 12)]
In [348]: ts = Series(np.random.randn(6), index=dates)

In [349]: ts
Out[349]:
2011-01-02    0.690002
2011-01-05    1.001543
2011-01-07   -0.503087
2011-01-08   -0.622274
```

```
2011-01-10 -0.921169  
2011-01-12 -0.726213
```

Under the hood, these `datetime` objects have been put in a `DatetimeIndex`, and the variable `ts` is now of type `TimeSeries`:

```
In [350]: type(ts)  
Out[350]: pandas.core.series.TimeSeries  
  
In [351]: ts.index  
Out[351]:  
<class 'pandas.tseries.index.DatetimeIndex'>  
[2011-01-02 00:00:00, ..., 2011-01-12 00:00:00]  
Length: 6, Freq: None, Timezone: None
```



It's not necessary to use the `TimeSeries` constructor explicitly; when creating a Series with a `DatetimeIndex`, pandas knows that the object is a time series.

Like other Series, arithmetic operations between differently-indexed time series automatically align on the dates:

```
In [352]: ts + ts[::2]  
Out[352]:  
2011-01-02    1.380004  
2011-01-05      NaN  
2011-01-07   -1.006175  
2011-01-08      NaN  
2011-01-10   -1.842337  
2011-01-12      NaN
```

pandas stores timestamps using NumPy's `datetime64` data type at the nanosecond resolution:

```
In [353]: ts.index.dtype  
Out[353]: dtype('datetime64[ns]')
```

Scalar values from a `DatetimeIndex` are pandas `Timestamp` objects

```
In [354]: stamp = ts.index[0]  
  
In [355]: stamp  
Out[355]: <Timestamp: 2011-01-02 00:00:00>
```

A `Timestamp` can be substituted anywhere you would use a `datetime` object. Additionally, it can store frequency information (if any) and understands how to do time zone conversions and other kinds of manipulations. More on both of these things later.

## Indexing, Selection, Subsetting

`TimeSeries` is a subclass of `Series` and thus behaves in the same way with regard to indexing and selecting data based on label:

```
In [356]: stamp = ts.index[2]
```

```
In [357]: ts[stamp]
```

```
Out[357]: -0.50308739136034464
```

As a convenience, you can also pass a string that is interpretable as a date:

```
In [358]: ts['1/10/2011']
```

```
Out[358]: -0.92116860801301081
```

```
In [359]: ts['20110110']
```

```
Out[359]: -0.92116860801301081
```

For longer time series, a year or only a year and month can be passed to easily select slices of data:

```
In [360]: longer_ts = Series(np.random.randn(1000),  
.....: index=pd.date_range('1/1/2000', periods=1000))
```

```
In [361]: longer_ts
```

```
Out[361]:  
2000-01-01    0.222896  
2000-01-02    0.051316  
2000-01-03   -1.157719  
2000-01-04    0.816707  
...  
2002-09-23   -0.395813  
2002-09-24   -0.180737  
2002-09-25    1.337508  
2002-09-26   -0.416584  
Freq: D, Length: 1000
```

```
In [362]: longer_ts['2001']
```

```
Out[362]:  
2001-01-01   -1.499503  
2001-01-02    0.545154  
2001-01-03    0.400823  
2001-01-04   -1.946230  
...  
2001-12-28   -1.568139  
2001-12-29   -0.900887  
2001-12-30    0.652346  
2001-12-31    0.871600  
Freq: D, Length: 365
```

```
In [363]: longer_ts['2001-05']
```

```
Out[363]:  
2001-05-01    1.662014  
2001-05-02   -1.189203  
2001-05-03    0.093597  
2001-05-04   -0.539164  
...  
2001-05-28   -0.683066  
2001-05-29   -0.950313  
2001-05-30    0.400710  
2001-05-31   -0.126072  
Freq: D, Length: 31
```

Slicing with dates works just like with a regular Series:

```
In [364]: ts[datetime(2011, 1, 7):]
```

```
Out[364]:  
2011-01-07   -0.503087  
2011-01-08   -0.622274  
2011-01-10   -0.921169  
2011-01-12   -0.726213
```

Because most time series data is ordered chronologically, you can slice with timestamps not contained in a time series to perform a range query:

```
In [365]: ts
```

```
Out[365]:  
2011-01-02    0.690002
```

```
In [366]: ts['1/6/2011':'1/11/2011']
```

```
Out[366]:  
2011-01-07   -0.503087
```

```
2011-01-05    1.001543      2011-01-08   -0.622274
2011-01-07   -0.503087      2011-01-10   -0.921169
2011-01-08   -0.622274
2011-01-10   -0.921169
2011-01-12   -0.726213
```

As before you can pass either a string date, datetime, or Timestamp. Remember that slicing in this manner produces views on the source time series just like slicing NumPy arrays. There is an equivalent instance method `truncate` which slices a TimeSeries between two dates:

```
In [367]: ts.truncate(after='1/9/2011')
Out[367]:
2011-01-02    0.690002
2011-01-05    1.001543
2011-01-07   -0.503087
2011-01-08   -0.622274
```

All of the above holds true for DataFrame as well, indexing on its rows:

```
In [368]: dates = pd.date_range('1/1/2000', periods=100, freq='W-WED')

In [369]: long_df = DataFrame(np.random.randn(100, 4),
.....:                      index=dates,
.....:                      columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [370]: long_df.ix['5-2001']
Out[370]:
Colorado      Texas     New York      Ohio
2001-05-02  0.943479 -0.349366  0.530412 -0.508724
2001-05-09  0.230643 -0.065569 -0.248717 -0.587136
2001-05-16 -1.022324  1.060661  0.954768 -0.511824
2001-05-23 -1.387680  0.767902 -1.164490  1.527070
2001-05-30  0.287542  0.715359 -0.345805  0.470886
```

## Time Series with Duplicate Indices

In some applications, there may be multiple data observations falling on a particular timestamp. Here is an example:

```
In [371]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000', '1/2/2000',
.....:                   '1/3/2000'])

In [372]: dup_ts = Series(np.arange(5), index=dates)

In [373]: dup_ts
Out[373]:
2000-01-01    0
2000-01-02    1
2000-01-02    2
2000-01-02    3
2000-01-03    4
```

We can tell that the index is not unique by checking its `is_unique` property:

```
In [374]: dup_ts.index.is_unique  
Out[374]: False
```

Indexing into this time series will now either produce scalar values or slices depending on whether a timestamp is duplicated:

```
In [375]: dup_ts['1/3/2000'] # not duplicated  
Out[375]: 4
```

```
In [376]: dup_ts['1/2/2000'] # duplicated  
Out[376]:  
2000-01-02    1  
2000-01-02    2  
2000-01-02    3
```

Suppose you wanted to aggregate the data having non-unique timestamps. One way to do this is to use `groupby` and pass `level=0` (the only level of indexing!):

```
In [377]: grouped = dup_ts.groupby(level=0)  
  
In [378]: grouped.mean()      In [379]: grouped.count()  
Out[378]:                      Out[379]:  
2000-01-01    0              2000-01-01    1  
2000-01-02    2              2000-01-02    3  
2000-01-03    4              2000-01-03    1
```

## Date Ranges, Frequencies, and Shifting

Generic time series in pandas are assumed to be irregular; that is, they have no fixed frequency. For many applications this is sufficient. However, it's often desirable to work relative to a fixed frequency, such as daily, monthly, or every 15 minutes, even if that means introducing missing values into a time series. Fortunately pandas has a full suite of standard time series frequencies and tools for resampling, inferring frequencies, and generating fixed frequency date ranges. For example, in the example time series, converting it to be fixed daily frequency can be accomplished by calling `resample`:

```
In [380]: ts                  In [381]: ts.resample('D')  
Out[380]:                      Out[381]:  
2011-01-02    0.690002      2011-01-02    0.690002  
2011-01-05    1.001543      2011-01-03    NaN  
2011-01-07    -0.503087     2011-01-04    NaN  
2011-01-08    -0.622274     2011-01-05    1.001543  
2011-01-10    -0.921169     2011-01-06    NaN  
2011-01-12    -0.726213     2011-01-07    -0.503087  
                           2011-01-08    -0.622274  
                           2011-01-09    NaN  
                           2011-01-10    -0.921169  
                           2011-01-11    NaN  
                           2011-01-12    -0.726213  
Freq: D
```

Conversion between frequencies or *resampling* is a big enough topic to have its own section later. Here I'll show you how to use the base frequencies and multiples thereof.

## Generating Date Ranges

While I used it previously without explanation, you may have guessed that `pandas.date_range` is responsible for generating a `DatetimeIndex` with an indicated length according to a particular frequency:

```
In [382]: index = pd.date_range('4/1/2012', '6/1/2012')

In [383]: index
Out[383]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-04-01 00:00:00, ..., 2012-06-01 00:00:00]
Length: 62, Freq: D, Timezone: None
```

By default, `date_range` generates daily timestamps. If you pass only a start or end date, you must pass a number of periods to generate:

```
In [384]: pd.date_range(start='4/1/2012', periods=20)
Out[384]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-04-01 00:00:00, ..., 2012-04-20 00:00:00]
Length: 20, Freq: D, Timezone: None

In [385]: pd.date_range(end='6/1/2012', periods=20)
Out[385]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-05-13 00:00:00, ..., 2012-06-01 00:00:00]
Length: 20, Freq: D, Timezone: None
```

The start and end dates define strict boundaries for the generated date index. For example, if you wanted a date index containing the last business day of each month, you would pass the '`BM`' frequency (business end of month) and only dates falling on or inside the date interval will be included:

```
In [386]: pd.date_range('1/1/2000', '12/1/2000', freq='BM')
Out[386]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-31 00:00:00, ..., 2000-11-30 00:00:00]
Length: 11, Freq: BM, Timezone: None
```

`date_range` by default preserves the time (if any) of the start or end timestamp:

```
In [387]: pd.date_range('5/2/2012 12:56:31', periods=5)
Out[387]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-05-02 12:56:31, ..., 2012-05-06 12:56:31]
Length: 5, Freq: D, Timezone: None
```

Sometimes you will have start or end dates with time information but want to generate a set of timestamps *normalized* to midnight as a convention. To do this, there is a `normalize` option:

```
In [388]: pd.date_range('5/2/2012 12:56:31', periods=5, normalize=True)
Out[388]:
<class 'pandas.tseries.index.DatetimeIndex'>
```

```
[2012-05-02 00:00:00, ..., 2012-05-06 00:00:00]
Length: 5, Freq: D, Timezone: None
```

## Frequencies and Date Offsets

Frequencies in pandas are composed of a *base frequency* and a multiplier. Base frequencies are typically referred to by a string alias, like 'M' for monthly or 'H' for hourly. For each base frequency, there is an object defined generally referred to as a *date offset*. For example, hourly frequency can be represented with the `Hour` class:

```
In [389]: from pandas.tseries.offsets import Hour, Minute

In [390]: hour = Hour()

In [391]: hour
Out[391]: <1 Hour>
```

You can define a multiple of an offset by passing an integer:

```
In [392]: four_hours = Hour(4)

In [393]: four_hours
Out[393]: <4 Hours>
```

In most applications, you would never need to explicitly create one of these objects, instead using a string alias like 'H' or '4H'. Putting an integer before the base frequency creates a multiple:

```
In [394]: pd.date_range('1/1/2000', '1/3/2000 23:59', freq='4H')
Out[394]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-01-03 20:00:00]
Length: 18, Freq: 4H, Timezone: None
```

Many offsets can be combined together by addition:

```
In [395]: Hour(2) + Minute(30)
Out[395]: <150 Minutes>
```

Similarly, you can pass frequency strings like '2h30min' which will effectively be parsed to the same expression:

```
In [396]: pd.date_range('1/1/2000', periods=10, freq='1h30min')
Out[396]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-01-01 13:30:00]
Length: 10, Freq: 90T, Timezone: None
```

Some frequencies describe points in time that are not evenly spaced. For example, 'M' (calendar month end) and 'BM' (last business/weekday of month) depend on the number of days in a month and, in the latter case, whether the month ends on a weekend or not. For lack of a better term, I call these *anchored* offsets.

See [Table 10-4](#) for a listing of frequency codes and date offset classes available in pandas.



Users can define their own custom frequency classes to provide date logic not available in pandas, though the full details of that are outside the scope of this book.

Table 10-4. Base Time Series Frequencies

Alias	Offset Type	Description
D	Day	Calendar daily
B	BusinessDay	Business daily
H	Hour	Hourly
T or min	Minute	Minutely
S	Second	Secondly
L or ms	Milli	Millisecond (1/1000th of 1 second)
U	Micro	Microsecond (1/1000000th of 1 second)
M	MonthEnd	Last calendar day of month
BM	BusinessMonthEnd	Last business day (weekday) of month
MS	MonthBegin	First calendar day of month
BMS	BusinessMonthBegin	First weekday of month
W-MON, W-TUE, ...	Week	Weekly on given day of week: MON, TUE, WED, THU, FRI, SAT, or SUN.
WOM-1MON, WOM-2MON, ...	WeekOfMonth	Generate weekly dates in the first, second, third, or fourth week of the month. For example, WOM-3FRI for the 3rd Friday of each month.
Q-JAN, Q-FEB, ...	QuarterEnd	Quarterly dates anchored on last calendar day of each month, for year ending in indicated month: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC.
BQ-JAN, BQ-FEB, ...	BusinessQuarterEnd	Quarterly dates anchored on last weekday day of each month, for year ending in indicated month
QS-JAN, QS-FEB, ...	QuarterBegin	Quarterly dates anchored on first calendar day of each month, for year ending in indicated month
BQS-JAN, BQS-FEB, ...	BusinessQuarterBegin	Quarterly dates anchored on first weekday day of each month, for year ending in indicated month
A-JAN, A-FEB, ...	YearEnd	Annual dates anchored on last calendar day of given month: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC
BA-JAN, BA-FEB, ...	BusinessYearEnd	Annual dates anchored on last weekday of given month
AS-JAN, AS-FEB, ...	YearBegin	Annual dates anchored on first day of given month
BAS-JAN, BAS-FEB, ...	BusinessYearBegin	Annual dates anchored on first weekday of given month

## Week of month dates

One useful frequency class is “week of month”, starting with `WOM`. This enables you to get dates like the third Friday of each month:

```
In [397]: rng = pd.date_range('1/1/2012', '9/1/2012', freq='WOM-3FRI')

In [398]: list(rng)
Out[398]:
[<Timestamp: 2012-01-20 00:00:00>,
 <Timestamp: 2012-02-17 00:00:00>,
 <Timestamp: 2012-03-16 00:00:00>,
 <Timestamp: 2012-04-20 00:00:00>,
 <Timestamp: 2012-05-18 00:00:00>,
 <Timestamp: 2012-06-15 00:00:00>,
 <Timestamp: 2012-07-20 00:00:00>,
 <Timestamp: 2012-08-17 00:00:00>]
```

Traders of US equity options will recognize these dates as the standard dates of monthly expiry.

## Shifting (Leading and Lagging) Data

“Shifting” refers to moving data backward and forward through time. Both Series and DataFrame have a `shift` method for doing naive shifts forward or backward, leaving the index unmodified:

```
In [399]: ts = Series(np.random.randn(4),
.....           index=pd.date_range('1/1/2000', periods=4, freq='M'))

In [400]: ts
Out[400]:
2000-01-31    0.575283
2000-02-29    0.304205
2000-03-31    1.814582
2000-04-30    1.634858
Freq: M

In [401]: ts.shift(2)
Out[401]:
2000-01-31      NaN
2000-02-29      NaN
2000-03-31    0.575283
2000-04-30    0.304205
Freq: M

In [402]: ts.shift(-2)
Out[402]:
2000-01-31    1.814582
2000-02-29    1.634858
2000-03-31      NaN
2000-04-30      NaN
Freq: M
```

A common use of `shift` is computing percent changes in a time series or multiple time series as DataFrame columns. This is expressed as

```
ts / ts.shift(1) - 1
```

Because naive shifts leave the index unmodified, some data is discarded. Thus if the frequency is known, it can be passed to `shift` to advance the timestamps instead of simply the data:

```
In [403]: ts.shift(2, freq='M')
Out[403]:
2000-03-31    0.575283
2000-04-30    0.304205
2000-05-31    1.814582
2000-06-30    1.634858
Freq: M
```

Other frequencies can be passed, too, giving you a lot of flexibility in how to lead and lag the data:

```
In [404]: ts.shift(3, freq='D')
Out[404]:
2000-02-03    0.575283
2000-03-03    0.304205
2000-04-03    1.814582
2000-05-03    1.634858
```

```
In [405]: ts.shift(1, freq='3D')
Out[405]:
2000-02-03    0.575283
2000-03-03    0.304205
2000-04-03    1.814582
2000-05-03    1.634858
```

```
In [406]: ts.shift(1, freq='90T')
Out[406]:
2000-01-31 01:30:00    0.575283
2000-02-29 01:30:00    0.304205
2000-03-31 01:30:00    1.814582
2000-04-30 01:30:00    1.634858
```

## Shifting dates with offsets

The pandas date offsets can also be used with `datetime` or `Timestamp` objects:

```
In [407]: from pandas.tseries.offsets import Day, MonthEnd
In [408]: now = datetime(2011, 11, 17)
In [409]: now + 3 * Day()
Out[409]: datetime.datetime(2011, 11, 20, 0, 0)
```

If you add an anchored offset like `MonthEnd`, the first increment will `roll forward` a date to the next date according to the frequency rule:

```
In [410]: now + MonthEnd()
Out[410]: datetime.datetime(2011, 11, 30, 0, 0)
In [411]: now + MonthEnd(2)
Out[411]: datetime.datetime(2011, 12, 31, 0, 0)
```

Anchored offsets can explicitly “roll” dates forward or backward using their `rollforward` and `rollback` methods, respectively:

```
In [412]: offset = MonthEnd()
In [413]: offset.rollforward(now)
Out[413]: datetime.datetime(2011, 11, 30, 0, 0)
In [414]: offset.rollback(now)
Out[414]: datetime.datetime(2011, 10, 31, 0, 0)
```

A clever use of date offsets is to use these methods with `groupby`:

```
In [415]: ts = Series(np.random.randn(20),
.....:                 index=pd.date_range('1/15/2000', periods=20, freq='4d'))
In [416]: ts.groupby(offset.rollforward).mean()
Out[416]:
2000-01-31    -0.448874
```

```
2000-02-29 -0.683663
2000-03-31  0.251920
```

Of course, an easier and faster way to do this is using `resample` (much more on this later):

```
In [417]: ts.resample('M', how='mean')
Out[417]:
2000-01-31 -0.448874
2000-02-29 -0.683663
2000-03-31  0.251920
Freq: M
```

## Time Zone Handling

Working with time zones is generally considered one of the most unpleasant parts of time series manipulation. In particular, daylight savings time (DST) transitions are a common source of complication. As such, many time series users choose to work with time series in *coordinated universal time* or *UTC*, which is the successor to Greenwich Mean Time and is the current international standard. Time zones are expressed as offsets from UTC; for example, New York is four hours behind UTC during daylight savings time and 5 hours the rest of the year.

In Python, time zone information comes from the 3rd party `pytz` library, which exposes the *Olson database*, a compilation of world time zone information. This is especially important for historical data because the DST transition dates (and even UTC offsets) have been changed numerous times depending on the whims of local governments. In the United States, the DST transition times have been changed many times since 1900!

For detailed information about `pytz` library, you'll need to look at that library's documentation. As far as this book is concerned, pandas wraps `pytz`'s functionality so you can ignore its API outside of the time zone names. Time zone names can be found interactively and in the docs:

```
In [418]: import pytz

In [419]: pytz.common_timezones[-5:]
Out[419]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']
```

To get a time zone object from `pytz`, use `pytz.timezone`:

```
In [420]: tz = pytz.timezone('US/Eastern')

In [421]: tz
Out[421]: <DstTzInfo 'US/Eastern' EST-1 day, 19:00:00 STD>
```

Methods in pandas will accept either time zone names or these objects. I recommend just using the names.

## Localization and Conversion

By default, time series in pandas are *time zone naive*. Consider the following time series:

```
rng = pd.date_range('3/9/2012 9:30', periods=6, freq='D')
ts = Series(np.random.randn(len(rng)), index=rng)
```

The index's `tz` field is `None`:

```
In [423]: print(ts.index.tz)
None
```

Date ranges can be generated with a time zone set:

```
In [424]: pd.date_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC')
Out[424]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-09 09:30:00, ..., 2012-03-18 09:30:00]
Length: 10, Freq: D, Timezone: UTC
```

Conversion from *naive* to *localized* is handled by the `tz_localize` method:

```
In [425]: ts_utc = ts.tz_localize('UTC')
```

```
In [426]: ts_utc
Out[426]:
2012-03-09 09:30:00+00:00    0.414615
2012-03-10 09:30:00+00:00    0.427185
2012-03-11 09:30:00+00:00    1.172557
2012-03-12 09:30:00+00:00   -0.351572
2012-03-13 09:30:00+00:00    1.454593
2012-03-14 09:30:00+00:00    2.043319
Freq: D
```

```
In [427]: ts_utc.index
Out[427]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-09 09:30:00, ..., 2012-03-14 09:30:00]
Length: 6, Freq: D, Timezone: UTC
```

Once a time series has been localized to a particular time zone, it can be converted to another time zone using `tz_convert`:

```
In [428]: ts_utc.tz_convert('US/Eastern')
Out[428]:
2012-03-09 04:30:00-05:00    0.414615
2012-03-10 04:30:00-05:00    0.427185
2012-03-11 05:30:00-04:00    1.172557
2012-03-12 05:30:00-04:00   -0.351572
2012-03-13 05:30:00-04:00    1.454593
2012-03-14 05:30:00-04:00    2.043319
Freq: D
```

In the case of the above time series, which straddles a DST transition in the US/Eastern time zone, we could localize to EST and convert to, say, UTC or Berlin time:

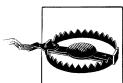
```
In [429]: ts_eastern = ts.tz_localize('US/Eastern')
```

```
In [430]: ts_eastern.tz_convert('UTC')
Out[430]:
2012-03-09 14:30:00+00:00    0.414615
2012-03-10 14:30:00+00:00    0.427185
2012-03-11 13:30:00+00:00    1.172557
2012-03-12 13:30:00+00:00   -0.351572
2012-03-13 13:30:00+00:00    1.454593
2012-03-14 13:30:00+00:00    2.043319
Freq: D

In [431]: ts_eastern.tz_convert('Europe/Berlin')
Out[431]:
2012-03-09 15:30:00+01:00    0.414615
2012-03-10 15:30:00+01:00    0.427185
2012-03-11 14:30:00+01:00    1.172557
2012-03-12 14:30:00+01:00   -0.351572
2012-03-13 14:30:00+01:00    1.454593
2012-03-14 14:30:00+01:00    2.043319
Freq: D
```

`tz_localize` and `tz_convert` are also instance methods on `DatetimeIndex`:

```
In [432]: ts.index.tz_localize('Asia/Shanghai')
Out[432]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-09 09:30:00, ..., 2012-03-14 09:30:00]
Length: 6, Freq: D, Timezone: Asia/Shanghai
```



Localizing naive timestamps also checks for ambiguous or non-existent times around daylight savings time transitions.

## Operations with Time Zone-aware Timestamp Objects

Similar to time series and date ranges, individual `Timestamp` objects similarly can be localized from naive to time zone-aware and converted from one time zone to another:

```
In [433]: stamp = pd.Timestamp('2011-03-12 04:00')

In [434]: stamp_utc = stamp.tz_localize('utc')

In [435]: stamp_utc.tz_convert('US/Eastern')
Out[435]: <Timestamp: 2011-03-11 23:00:00-0500 EST, tz=US/Eastern>
```

You can also pass a time zone when creating the `Timestamp`:

```
In [436]: stamp_moscow = pd.Timestamp('2011-03-12 04:00', tz='Europe/Moscow')

In [437]: stamp_moscow
Out[437]: <Timestamp: 2011-03-12 04:00:00+0300 MSK, tz=Europe/Moscow>
```

Time zone-aware `Timestamp` objects internally store a UTC timestamp value as nanoseconds since the UNIX epoch (January 1, 1970); this UTC value is invariant between time zone conversions:

```
In [438]: stamp_utc.value  
Out[438]: 1299902400000000000
```

```
In [439]: stamp_utc.tz_convert('US/Eastern').value  
Out[439]: 1299902400000000000
```

When performing time arithmetic using pandas's `DateOffset` objects, daylight savings time transitions are respected where possible:

```
# 30 minutes before DST transition  
In [440]: from pandas.tseries.offsets import Hour  
  
In [441]: stamp = pd.Timestamp('2012-03-12 01:30', tz='US/Eastern')  
  
In [442]: stamp  
Out[442]: <Timestamp: 2012-03-12 01:30:00-0400 EDT, tz=US/Eastern>  
  
In [443]: stamp + Hour()  
Out[443]: <Timestamp: 2012-03-12 02:30:00-0400 EDT, tz=US/Eastern>  
  
# 90 minutes before DST transition  
In [444]: stamp = pd.Timestamp('2012-11-04 00:30', tz='US/Eastern')  
  
In [445]: stamp  
Out[445]: <Timestamp: 2012-11-04 00:30:00-0400 EDT, tz=US/Eastern>  
  
In [446]: stamp + 2 * Hour()  
Out[446]: <Timestamp: 2012-11-04 01:30:00-0500 EST, tz=US/Eastern>
```

## Operations between Different Time Zones

If two time series with different time zones are combined, the result will be UTC. Since the timestamps are stored under the hood in UTC, this is a straightforward operation and requires no conversion to happen:

```
In [447]: rng = pd.date_range('3/7/2012 9:30', periods=10, freq='B')  
  
In [448]: ts = Series(np.random.randn(len(rng)), index=rng)  
  
In [449]: ts  
Out[449]:  
2012-03-07 09:30:00    -1.749309  
2012-03-08 09:30:00    -0.387235  
2012-03-09 09:30:00    -0.208074  
2012-03-12 09:30:00    -1.221957  
2012-03-13 09:30:00    -0.067460  
2012-03-14 09:30:00     0.229005  
2012-03-15 09:30:00    -0.576234  
2012-03-16 09:30:00     0.816895  
2012-03-19 09:30:00    -0.772192  
2012-03-20 09:30:00    -1.333576  
Freq: B  
  
In [450]: ts1 = ts[:7].tz_localize('Europe/London')
```

```
In [451]: ts2 = ts1[2: ].tz_convert('Europe/Moscow')

In [452]: result = ts1 + ts2

In [453]: result.index
Out[453]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-07 09:30:00, ..., 2012-03-15 09:30:00]
Length: 7, Freq: B, Timezone: UTC
```

## Periods and Period Arithmetic

*Periods* represent time spans, like days, months, quarters, or years. The `Period` class represents this data type, requiring a string or integer and a frequency from the above table:

```
In [454]: p = pd.Period(2007, freq='A-DEC')

In [455]: p
Out[455]: Period('2007', 'A-DEC')
```

In this case, the `Period` object represents the full timespan from January 1, 2007 to December 31, 2007, inclusive. Conveniently, adding and subtracting integers from periods has the effect of shifting by their frequency:

```
In [456]: p + 5
Out[456]: Period('2012', 'A-DEC')           In [457]: p - 2
Out[457]: Period('2005', 'A-DEC')
```

If two periods have the same frequency, their difference is the number of units between them:

```
In [458]: pd.Period('2014', freq='A-DEC') - p
Out[458]: 7
```

Regular ranges of periods can be constructed using the `period_range` function:

```
In [459]: rng = pd.period_range('1/1/2000', '6/30/2000', freq='M')

In [460]: rng
Out[460]:
<class 'pandas.tseries.period.PeriodIndex'>
freq: M
[2000-01, ..., 2000-06]
length: 6
```

The `PeriodIndex` class stores a sequence of periods and can serve as an axis index in any pandas data structure:

```
In [461]: Series(np.random.randn(6), index=rng)
Out[461]:
2000-01    -0.309119
2000-02     0.028558
2000-03     1.129605
2000-04    -0.374173
2000-05    -0.011401
```

```
2000-06    0.272924
Freq: M
```

If you have an array of strings, you can also appeal to the `PeriodIndex` class itself:

```
In [462]: values = ['2001Q3', '2002Q2', '2003Q1']

In [463]: index = pd.PeriodIndex(values, freq='Q-DEC')

In [464]: index
Out[464]:
<class 'pandas.tseries.period.PeriodIndex'>
freq: Q-DEC
[2001Q3, ..., 2003Q1]
length: 3
```

## Period Frequency Conversion

Periods and `PeriodIndex` objects can be converted to another frequency using their `asfreq` method. As an example, suppose we had an annual period and wanted to convert it into a monthly period either at the start or end of the year. This is fairly straightforward:

```
In [465]: p = pd.Period('2007', freq='A-DEC')

In [466]: p.asfreq('M', how='start')      In [467]: p.asfreq('M', how='end')
Out[466]: Period('2007-01', 'M')        Out[467]: Period('2007-12', 'M')
```

You can think of `Period('2007', 'A-DEC')` as being a cursor pointing to a span of time, subdivided by monthly periods. See [Figure 10-1](#) for an illustration of this. For a *fiscal year* ending on a month other than December, the monthly subperiods belonging are different:

```
In [468]: p = pd.Period('2007', freq='A-JUN')

In [469]: p.asfreq('M', 'start')      In [470]: p.asfreq('M', 'end')
Out[469]: Period('2006-07', 'M')    Out[470]: Period('2007-07', 'M')
```

When converting from high to low frequency, the superperiod will be determined depending on where the subperiod “belongs”. For example, in `A-JUN` frequency, the month `Aug-2007` is actually part of the `2008` period:

```
In [471]: p = pd.Period('2007-08', 'M')

In [472]: p.asfreq('A-JUN')
Out[472]: Period('2008', 'A-JUN')
```

Whole `PeriodIndex` objects or `TimeSeries` can be similarly converted with the same semantics:

```
In [473]: rng = pd.period_range('2006', '2009', freq='A-DEC')

In [474]: ts = Series(np.random.randn(len(rng)), index=rng)

In [475]: ts
```

```

Out[475]:
2006    -0.601544
2007     0.574265
2008    -0.194115
2009     0.202225
Freq: A-DEC

In [476]: ts.asfreq('M', how='start')
Out[476]:
2006-01   -0.601544
2007-01    0.574265
2008-01   -0.194115
2009-01    0.202225
Freq: M

In [477]: ts.asfreq('B', how='end')
Out[477]:
2006-12-29   -0.601544
2007-12-31    0.574265
2008-12-31   -0.194115
2009-12-31    0.202225
Freq: B

```

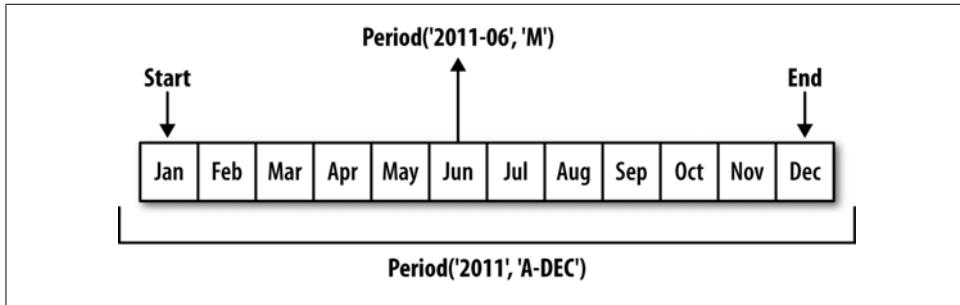


Figure 10-1. Period frequency conversion illustration

## Quarterly Period Frequencies

Quarterly data is standard in accounting, finance, and other fields. Much quarterly data is reported relative to a *fiscal year end*, typically the last calendar or business day of one of the 12 months of the year. As such, the period 2012Q4 has a different meaning depending on fiscal year end. pandas supports all 12 possible quarterly frequencies as Q-JAN through Q-DEC:

```

In [478]: p = pd.Period('2012Q4', freq='Q-JAN')

In [479]: p
Out[479]: Period('2012Q4', 'Q-JAN')

```

In the case of fiscal year ending in January, 2012Q4 runs from November through January, which you can check by converting to daily frequency. See [Figure 10-2](#) for an illustration:

```

In [480]: p.asfreq('D', 'start')
Out[480]: Period('2011-11-01', 'D')           In [481]: p.asfreq('D', 'end')
                                                Out[481]: Period('2012-01-31', 'D')

```

Thus, it's possible to do period arithmetic very easily; for example, to get the timestamp at 4PM on the 2nd to last business day of the quarter, you could do:

```
In [482]: p4pm = (p.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60
```

```
In [483]: p4pm
```

```
Out[483]: Period('2012-01-30 16:00', 'T')
```

```
In [484]: p4pm.to_timestamp()
```

```
Out[484]: <Timestamp: 2012-01-30 16:00:00>
```

Year 2012												
M	JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
<b>Q-DEC</b>	2012Q1		2012Q2			2012Q3			2012Q4			
<b>Q-SEP</b>	2012Q2		2012Q3			2012Q4			2013Q1			
<b>Q-FEB</b>	2012Q4	2013Q1		2013Q2			2013Q3		Q4			

Figure 10-2. Different quarterly frequency conventions

Generating quarterly ranges works as you would expect using `period_range`. Arithmetic is identical, too:

```
In [485]: rng = pd.period_range('2011Q3', '2012Q4', freq='Q-JAN')
```

```
In [486]: ts = Series(np.arange(len(rng)), index=rng)
```

```
In [487]: ts
```

```
Out[487]:
```

```
2011Q3    0
```

```
2011Q4    1
```

```
2012Q1    2
```

```
2012Q2    3
```

```
2012Q3    4
```

```
2012Q4    5
```

```
Freq: Q-JAN
```

```
In [488]: new_rng = (rng.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60
```

```
In [489]: ts.index = new_rng.to_timestamp()
```

```
In [490]: ts
```

```
Out[490]:
```

```
2010-10-28 16:00:00    0
```

```
2011-01-28 16:00:00    1
```

```
2011-04-28 16:00:00    2
```

```
2011-07-28 16:00:00    3
```

```
2011-10-28 16:00:00    4
```

```
2012-01-30 16:00:00    5
```

## Converting Timestamps to Periods (and Back)

Series and DataFrame objects indexed by timestamps can be converted to periods using the `to_period` method:

```
In [491]: rng = pd.date_range('1/1/2000', periods=3, freq='M')

In [492]: ts = Series(randn(3), index=rng)

In [493]: pts = ts.to_period()

In [494]: ts
Out[494]:
2000-01-31    -0.505124
2000-02-29     2.954439
2000-03-31    -2.630247
Freq: M

In [495]: pts
Out[495]:
2000-01    -0.505124
2000-02     2.954439
2000-03    -2.630247
Freq: M
```

Since periods always refer to non-overlapping timespans, a timestamp can only belong to a single period for a given frequency. While the frequency of the new `PeriodIndex` is inferred from the timestamps by default, you can specify any frequency you want. There is also no problem with having duplicate periods in the result:

```
In [496]: rng = pd.date_range('1/29/2000', periods=6, freq='D')

In [497]: ts2 = Series(randn(6), index=rng)

In [498]: ts2.to_period('M')
Out[498]:
2000-01    -0.352453
2000-01    -0.477808
2000-01     0.161594
2000-02     1.686833
2000-02     0.821965
2000-02    -0.667406
Freq: M
```

To convert back to timestamps, use `to_timestamp`:

```
In [499]: pts = ts.to_period()

In [500]: pts
Out[500]:
2000-01    -0.505124
2000-02     2.954439
2000-03    -2.630247
Freq: M

In [501]: pts.to_timestamp(how='end')
Out[501]:
2000-01-31    -0.505124
2000-02-29     2.954439
2000-03-31    -2.630247
Freq: M
```

## Creating a PeriodIndex from Arrays

Fixed frequency data sets are sometimes stored with timespan information spread across multiple columns. For example, in this macroeconomic data set, the year and quarter are in different columns:

```
In [502]: data = pd.read_csv('ch08/macrodta.csv')

In [503]: data.year          In [504]: data.quarter
Out[503]:                         Out[504]:
0    1959                      0    1
1    1959                      1    2
2    1959                      2    3
3    1959                      3    4
...
199   2008                      ...
200   2009                      199   4
201   2009                      200   1
202   2009                      201   2
203   2009                      202   3
Name: year, Length: 203        Name: quarter, Length: 203
```

By passing these arrays to `PeriodIndex` with a frequency, they can be combined to form an index for the DataFrame:

```
In [505]: index = pd.PeriodIndex(year=data.year, quarter=data.quarter, freq='Q-DEC')

In [506]: index
Out[506]:
<class 'pandas.tseries.period.PeriodIndex'>
freq: Q-DEC
[1959Q1, ..., 2009Q3]
length: 203

In [507]: data.index = index

In [508]: data.infl
Out[508]:
1959Q1    0.00
1959Q2    2.34
1959Q3    2.74
1959Q4    0.27
...
2008Q4   -8.79
2009Q1    0.94
2009Q2    3.37
2009Q3    3.56
Freq: Q-DEC, Name: infl, Length: 203
```

## Resampling and Frequency Conversion

*Resampling* refers to the process of converting a time series from one frequency to another. Aggregating higher frequency data to lower frequency is called *downsampling*, while converting lower frequency to higher frequency is called *upsampling*. Not

all resampling falls into either of these categories; for example, converting W-WED (weekly on Wednesday) to W-FRI is neither upsampling nor downstamping.

pandas objects are equipped with a `resample` method, which is the workhorse function for all frequency conversion:

```
In [509]: rng = pd.date_range('1/1/2000', periods=100, freq='D')
```

```
In [510]: ts = Series(randn(len(rng)), index=rng)
```

```
In [511]: ts.resample('M', how='mean')
```

```
Out[511]:
```

```
2000-01-31    0.170876  
2000-02-29    0.165020  
2000-03-31    0.095451  
2000-04-30    0.363566
```

```
Freq: M
```

```
In [512]: ts.resample('M', how='mean', kind='period')
```

```
Out[512]:
```

```
2000-01    0.170876  
2000-02    0.165020  
2000-03    0.095451  
2000-04    0.363566
```

```
Freq: M
```

`resample` is a flexible and high-performance method that can be used to process very large time series. I'll illustrate its semantics and use through a series of examples.

*Table 10-5. Resample method arguments*

Argument	Description
<code>freq</code>	String or DateOffset indicating desired resampled frequency, e.g. 'M', '5min', or <code>Second(15)</code>
<code>how='mean'</code>	Function name or array function producing aggregated value, for example 'mean', 'ohlc', <code>np.max</code> . Defaults to 'mean'. Other common values: 'first', 'last', 'median', 'ohlc', 'max', 'min'.
<code>axis=0</code>	Axis to resample on, default axis=0
<code>fill_method=None</code>	How to interpolate when upsampling, as in 'ffill' or 'bfill'. By default does no interpolation.
<code>closed='right'</code>	In downsampling, which end of each interval is closed (inclusive), 'right' or 'left'. Defaults to 'right'
<code>label='right'</code>	In downsampling, how to label the aggregated result, with the 'right' or 'left' bin edge. For example, the 9:30 to 9:35 5-minute interval could be labeled 9:30 or 9:35. Defaults to 'right' (or 9:35, in this example).
<code>loffset=None</code>	Time adjustment to the bin labels, such as '-1s' / <code>Second(-1)</code> to shift the aggregate labels one second earlier
<code>limit=None</code>	When forward or backward filling, the maximum number of periods to fill

Argument	Description
kind=None	Aggregate to periods ('period') or timestamps ('timestamp'); defaults to kind of index the time series has
convention=None	When resampling periods, the convention ('start' or 'end') for converting the low frequency period to high frequency. Defaults to 'end'

## Downsampling

Aggregating data to a regular, lower frequency is a pretty normal time series task. The data you're aggregating doesn't need to be fixed frequently; the desired frequency defines *bin edges* that are used to slice the time series into pieces to aggregate. For example, to convert to monthly, 'M' or 'BM', the data need to be chopped up into one month intervals. Each interval is said to be *half-open*; a data point can only belong to one interval, and the union of the intervals must make up the whole time frame. There are a couple things to think about when using `resample` to downsample data:

- Which side of each interval is *closed*
- How to label each aggregated bin, either with the start of the interval or the end

To illustrate, let's look at some one-minute data:

```
In [513]: rng = pd.date_range('1/1/2000', periods=12, freq='T')
```

```
In [514]: ts = Series(np.arange(12), index=rng)
```

```
In [515]: ts
Out[515]:
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
2000-01-01 00:09:00    9
2000-01-01 00:10:00   10
2000-01-01 00:11:00   11
Freq: T
```

Suppose you wanted to aggregate this data into five-minute chunks or *bars* by taking the sum of each group:

```
In [516]: ts.resample('5min', how='sum')
Out[516]:
2000-01-01 00:00:00    0
2000-01-01 00:05:00   15
2000-01-01 00:10:00   40
2000-01-01 00:15:00   11
Freq: 5T
```

The frequency you pass defines bin edges in five-minute increments. By default, the *right* bin edge is inclusive, so the 00:05 value is included in the 00:00 to 00:05 interval.<sup>1</sup> Passing `closed='left'` changes the interval to be closed on the left:

```
In [517]: ts.resample('5min', how='sum', closed='left')
Out[517]:
2000-01-01 00:05:00    10
2000-01-01 00:10:00    35
2000-01-01 00:15:00    21
Freq: 5T
```

As you can see, the resulting time series is labeled by the timestamps from the right side of each bin. By passing `label='left'` you can label them with the left bin edge:

```
In [518]: ts.resample('5min', how='sum', closed='left', label='left')
Out[518]:
2000-01-01 00:00:00    10
2000-01-01 00:05:00    35
2000-01-01 00:10:00    21
Freq: 5T
```

See [Figure 10-3](#) for an illustration of minutely data being resampled to five-minute.

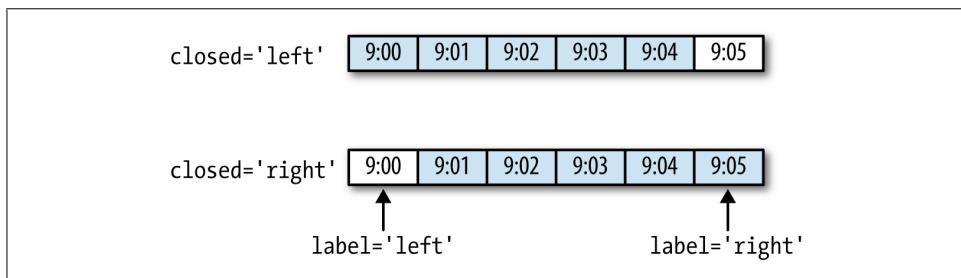


Figure 10-3. 5-minute resampling illustration of closed, label conventions

Lastly, you might want to shift the result index by some amount, say subtracting one second from the right edge to make it more clear which interval the timestamp refers to. To do this, pass a string or date offset to `loffset`:

```
In [519]: ts.resample('5min', how='sum', loffset='-1s')
Out[519]:
1999-12-31 23:59:59    0
2000-01-01 00:04:59    15
2000-01-01 00:09:59    40
2000-01-01 00:14:59    11
Freq: 5T
```

1. The choice of `closed='right'`, `label='right'` as the default might seem a bit odd to some users. In practice the choice is somewhat arbitrary; for some target frequencies, `closed='left'` is preferable, while for others `closed='right'` makes more sense. The important thing is that you keep in mind exactly how you are segmenting the data.

This also could have been accomplished by calling the `shift` method on the result without the `loffset`.

### Open-High-Low-Close (OHLC) resampling

In finance, an ubiquitous way to aggregate a time series is to compute four values for each bucket: the first (open), last (close), maximum (high), and minimal (low) values. By passing `how='ohlc'` you will obtain a DataFrame having columns containing these four aggregates, which are efficiently computed in a single sweep of the data:

```
In [520]: ts.resample('5min', how='ohlc')
Out[520]:
          open  high  low  close
2000-01-01 00:00:00    0    0    0    0
2000-01-01 00:05:00    1    5    1    5
2000-01-01 00:10:00    6   10    6   10
2000-01-01 00:15:00   11   11   11   11
```

### Resampling with GroupBy

An alternate way to downsample is to use pandas's `groupby` functionality. For example, you can group by month or weekday by passing a function that accesses those fields on the time series's index:

```
In [521]: rng = pd.date_range('1/1/2000', periods=100, freq='D')
In [522]: ts = Series(np.arange(100), index=rng)
In [523]: ts.groupby(lambda x: x.month).mean()
Out[523]:
1    15
2    45
3    75
4    95

In [524]: ts.groupby(lambda x: x.weekday).mean()
Out[524]:
0    47.5
1    48.5
2    49.5
3    50.5
4    51.5
5    49.0
6    50.0
```

## Upsampling and Interpolation

When converting from a low frequency to a higher frequency, no aggregation is needed. Let's consider a DataFrame with some weekly data:

```
In [525]: frame = DataFrame(np.random.randn(2, 4),
.....:                         index=pd.date_range('1/1/2000', periods=2, freq='W-WED'),
.....:                         columns=['Colorado', 'Texas', 'New York', 'Ohio'])
```

```
In [526]: frame[:5]
Out[526]:
          Colorado      Texas  New York      Ohio
2000-01-05 -0.609657 -0.268837  0.195592  0.85979
2000-01-12 -0.263206  1.141350 -0.101937 -0.07666
```

When resampling this to daily frequency, by default missing values are introduced:

```
In [527]: df_daily = frame.resample('D')
In [528]: df_daily
Out[528]:
          Colorado      Texas  New York      Ohio
2000-01-05 -0.609657 -0.268837  0.195592  0.85979
2000-01-06      NaN       NaN       NaN       NaN
2000-01-07      NaN       NaN       NaN       NaN
2000-01-08      NaN       NaN       NaN       NaN
2000-01-09      NaN       NaN       NaN       NaN
2000-01-10      NaN       NaN       NaN       NaN
2000-01-11      NaN       NaN       NaN       NaN
2000-01-12 -0.263206  1.141350 -0.101937 -0.07666
```

Suppose you wanted to fill forward each weekly value on the non-Wednesdays. The same filling or interpolation methods available in the `fillna` and `reindex` methods are available for resampling:

```
In [529]: frame.resample('D', fill_method='ffill')
Out[529]:
          Colorado      Texas  New York      Ohio
2000-01-05 -0.609657 -0.268837  0.195592  0.85979
2000-01-06 -0.609657 -0.268837  0.195592  0.85979
2000-01-07 -0.609657 -0.268837  0.195592  0.85979
2000-01-08 -0.609657 -0.268837  0.195592  0.85979
2000-01-09 -0.609657 -0.268837  0.195592  0.85979
2000-01-10 -0.609657 -0.268837  0.195592  0.85979
2000-01-11 -0.609657 -0.268837  0.195592  0.85979
2000-01-12 -0.263206  1.141350 -0.101937 -0.07666
```

You can similarly choose to only fill a certain number of periods forward to limit how far to continue using an observed value:

```
In [530]: frame.resample('D', fill_method='ffill', limit=2)
Out[530]:
          Colorado      Texas  New York      Ohio
2000-01-05 -0.609657 -0.268837  0.195592  0.85979
2000-01-06 -0.609657 -0.268837  0.195592  0.85979
2000-01-07 -0.609657 -0.268837  0.195592  0.85979
2000-01-08      NaN       NaN       NaN       NaN
2000-01-09      NaN       NaN       NaN       NaN
2000-01-10      NaN       NaN       NaN       NaN
2000-01-11      NaN       NaN       NaN       NaN
2000-01-12 -0.263206  1.141350 -0.101937 -0.07666
```

Notably, the new date index need not overlap with the old one at all:

```
In [531]: frame.resample('W-THU', fill_method='ffill')
Out[531]:
          Colorado      Texas  New York      Ohio
2000-01-06 -0.609657 -0.268837  0.195592  0.85979
2000-01-13 -0.263206  1.141350 -0.101937 -0.07666
```

## Resampling with Periods

Resampling data indexed by periods is reasonably straightforward and works as you would hope:

```
In [532]: frame = DataFrame(np.random.randn(24, 4),
.....:                      index=pd.period_range('1-2000', '12-2001', freq='M'),
.....:                      columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [533]: frame[:5]
Out[533]:
          Colorado      Texas  New York      Ohio
2000-01  0.120837  1.076607  0.434200  0.056432
2000-02 -0.378890  0.047831  0.341626  1.567920
2000-03 -0.047619 -0.821825 -0.179330 -0.166675
2000-04  0.333219 -0.544615 -0.653635 -2.311026
2000-05  1.612270 -0.806614  0.557884  0.580201

In [534]: annual_frame = frame.resample('A-DEC', how='mean')

In [535]: annual_frame
Out[535]:
          Colorado      Texas  New York      Ohio
2000  0.352070 -0.553642  0.196642 -0.094099
2001  0.158207  0.042967 -0.360755  0.184687
```

Upsampling is more nuanced as you must make a decision about which end of the timespan in the new frequency to place the values before resampling, just like the `asfreq` method. The `convention` argument defaults to `'end'` but can also be `'start'`:

```
# Q-DEC: Quarterly, year ending in December
In [536]: annual_frame.resample('Q-DEC', fill_method='ffill')
Out[536]:
          Colorado      Texas  New York      Ohio
2000Q4  0.352070 -0.553642  0.196642 -0.094099
2001Q1  0.352070 -0.553642  0.196642 -0.094099
2001Q2  0.352070 -0.553642  0.196642 -0.094099
2001Q3  0.352070 -0.553642  0.196642 -0.094099
2001Q4  0.158207  0.042967 -0.360755  0.184687

In [537]: annual_frame.resample('Q-DEC', fill_method='ffill', convention='start')
Out[537]:
          Colorado      Texas  New York      Ohio
2000Q1  0.352070 -0.553642  0.196642 -0.094099
2000Q2  0.352070 -0.553642  0.196642 -0.094099
2000Q3  0.352070 -0.553642  0.196642 -0.094099
2000Q4  0.352070 -0.553642  0.196642 -0.094099
2001Q1  0.158207  0.042967 -0.360755  0.184687
```

Since periods refer to timespans, the rules about upsampling and downsampling are more rigid:

- In downsampling, the target frequency must be a *subperiod* of the source frequency.
- In upsampling, the target frequency must be a *superperiod* of the source frequency.

If these rules are not satisfied, an exception will be raised. This mainly affects the quarterly, annual, and weekly frequencies; for example, the timespans defined by Q-MAR only line up with A-MAR, A-JUN, A-SEP, and A-DEC:

```
In [538]: annual_frame.resample('Q-MAR', fill_method='ffill')
Out[538]:
   Colorado      Texas  New York      Ohio
2001Q3  0.352070 -0.553642  0.196642 -0.094099
2001Q4  0.352070 -0.553642  0.196642 -0.094099
2002Q1  0.352070 -0.553642  0.196642 -0.094099
2002Q2  0.352070 -0.553642  0.196642 -0.094099
2002Q3  0.158207  0.042967 -0.360755  0.184687
```

## Time Series Plotting

Plots with pandas time series have improved date formatting compared with matplotlib out of the box. As an example, I downloaded some stock price data on a few common US stock from Yahoo! Finance:

```
In [539]: close_px_all = pd.read_csv('ch09/stock_px.csv', parse_dates=True, index_col=0)

In [540]: close_px = close_px_all[['AAPL', 'MSFT', 'XOM']]

In [541]: close_px = close_px.resample('B', fill_method='ffill')

In [542]: close_px
Out[542]:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2292 entries, 2003-01-02 00:00:00 to 2011-10-14 00:00:00
Freq: B
Data columns:
AAPL    2292 non-null values
MSFT    2292 non-null values
XOM    2292 non-null values
dtypes: float64(3)
```

Calling `plot` on one of the columns generates a simple plot, seen in [Figure 10-4](#).

```
In [544]: close_px['AAPL'].plot()
```

When called on a DataFrame, as you would expect, all of the time series are drawn on a single subplot with a legend indicating which is which. I'll plot only the year 2009 data so you can see how both months and years are formatted on the X axis; see [Figure 10-5](#).

```
In [546]: close_px.ix['2009'].plot()
```

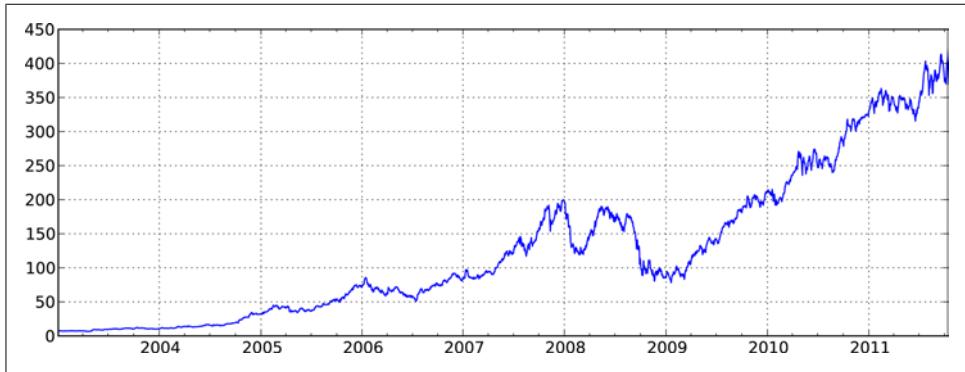


Figure 10-4. AAPL Daily Price

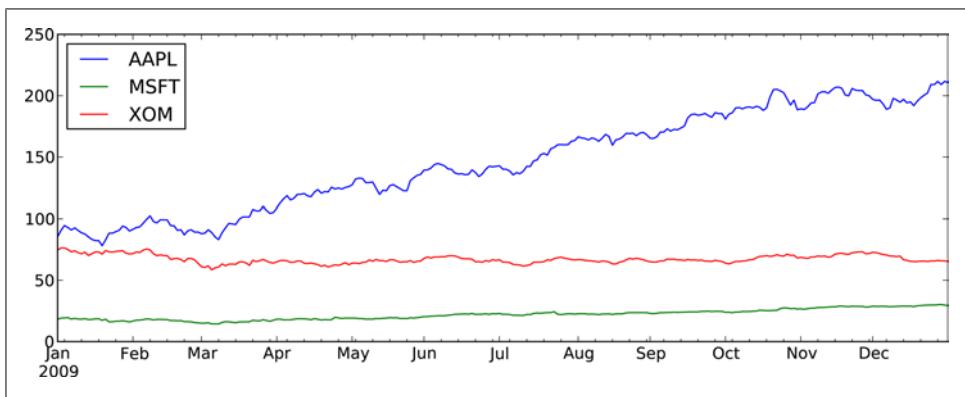


Figure 10-5. Stock Prices in 2009

```
In [548]: close_px['AAPL'].ix['01-2011':'03-2011'].plot()
```

Quarterly frequency data is also more nicely formatted with quarterly markers, something that would be quite a bit more work to do by hand. See [Figure 10-7](#).

```
In [550]: appl_q = close_px['AAPL'].resample('Q-DEC', fill_method='ffill')
```

```
In [551]: appl_q.ix['2009':].plot()
```

A last feature of time series plotting in pandas is that by right-clicking and dragging to zoom in and out, the dates will be dynamically expanded or contracted and reformatting depending on the timespan contained in the plot view. This is of course only true when using matplotlib in interactive mode.

## Moving Window Functions

A common class of array transformations intended for time series operations are statistics and other functions evaluated over a sliding window or with exponentially de-

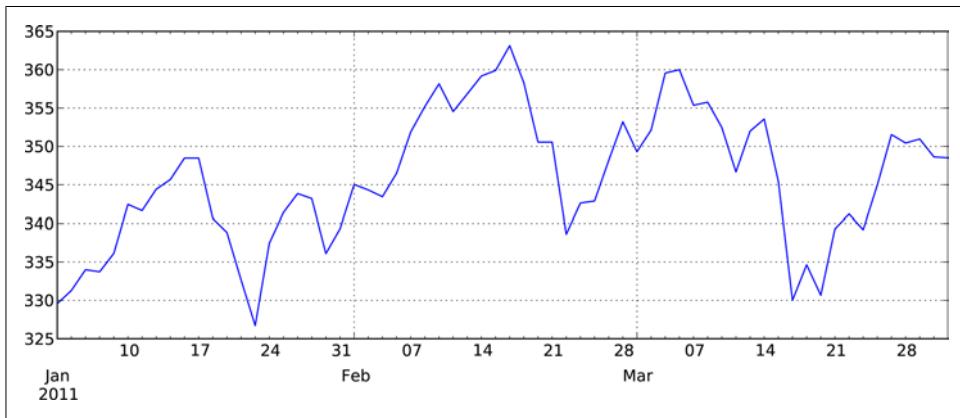


Figure 10-6. Apple Daily Price in 1/2011-3/2011

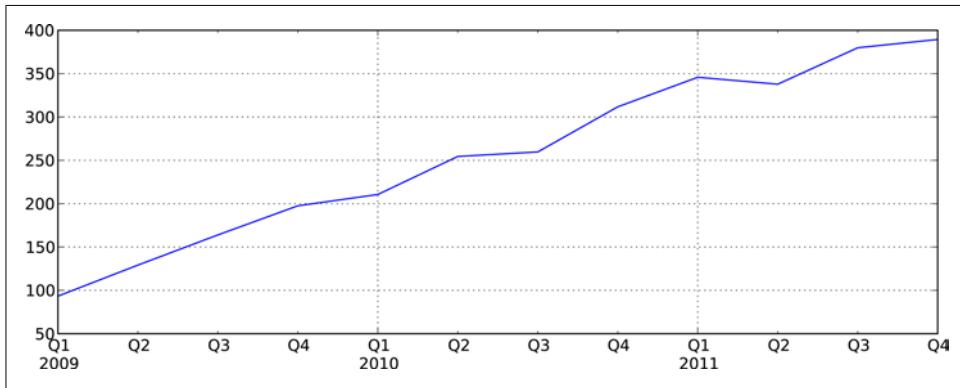


Figure 10-7. Apple Quarterly Price 2009-2011

caying weights. I call these *moving window functions*, even though it includes functions without a fixed-length window like exponentially-weighted moving average. Like other statistical functions, these also automatically exclude missing data.

`rolling_mean` is one of the simplest such functions. It takes a TimeSeries or DataFrame along with a `window` (expressed as a number of periods):

```
In [555]: close_px.AAPL.plot()
Out[555]: <matplotlib.axes.AxesSubplot at 0x1099b3990>
```

```
In [556]: pd.rolling_mean(close_px.AAPL, 250).plot()
```

See Figure 10-8 for the plot. By default functions like `rolling_mean` require the indicated number of non-NA observations. This behavior can be changed to account for missing data and, in particular, the fact that you will have fewer than `window` periods of data at the beginning of the time series (see Figure 10-9):

```
In [558]: appl_std250 = pd.rolling_std(close_px.AAPL, 250, min_periods=10)
```

```
In [559]: appl_std250[5:12]
```

```
Out[559]:
```

2003-01-09	NaN
2003-01-10	NaN
2003-01-13	NaN
2003-01-14	NaN
2003-01-15	0.077496
2003-01-16	0.074760
2003-01-17	0.112368

```
Freq: B
```

```
In [560]: appl_std250.plot()
```

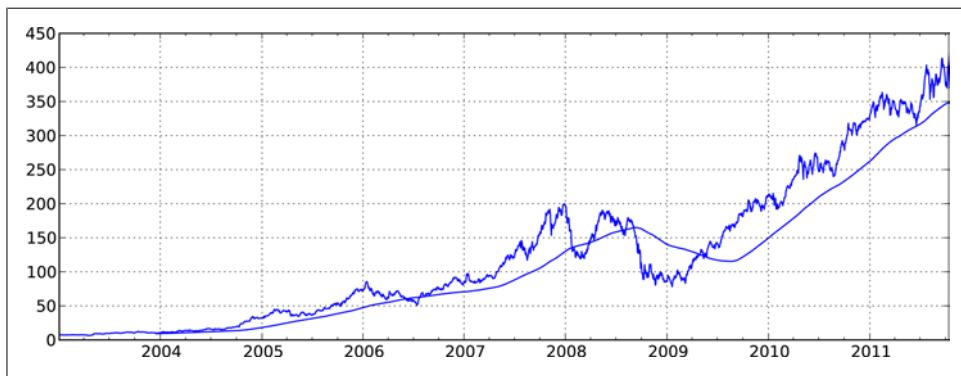


Figure 10-8. Apple Price with 250-day MA

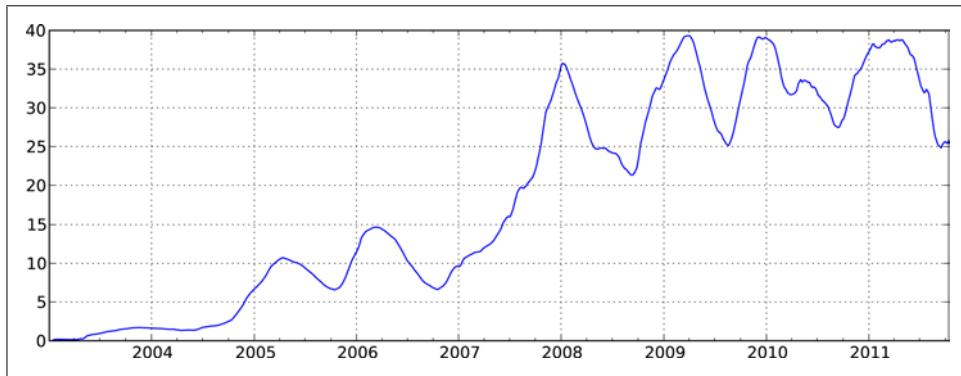


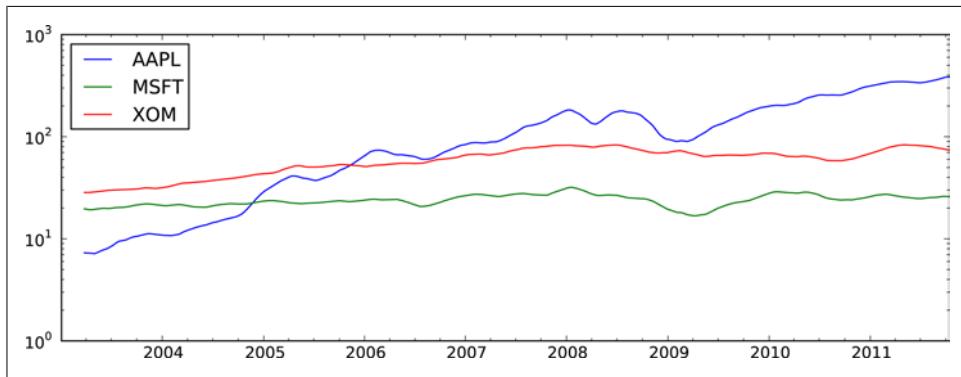
Figure 10-9. Apple 250-day daily return standard deviation

To compute an *expanding window mean*, you can see that an expanding window is just a special case where the window is the length of the time series, but only one or more periods is required to compute a value:

```
# Define expanding mean in terms of rolling_mean
In [561]: expanding_mean = lambda x: rolling_mean(x, len(x), min_periods=1)
```

Calling `rolling_mean` and friends on a DataFrame applies the transformation to each column (see [Figure 10-10](#)):

```
In [563]: pd.rolling_mean(close_px, 60).plot(logy=True)
```



*Figure 10-10. Stocks Prices 60-day MA (log Y-axis)*

See [Table 10-6](#) for a listing of related functions in pandas.

*Table 10-6. Moving window and exponentially-weighted functions*

Function	Description
<code>rolling_count</code>	Returns number of non-NA observations in each trailing window.
<code>rolling_sum</code>	Moving window sum.
<code>rolling_mean</code>	Moving window mean.
<code>rolling_median</code>	Moving window median.
<code>rolling_var</code> , <code>rolling_std</code>	Moving window variance and standard deviation, respectively. Uses n - 1 denominator.
<code>rolling_skew</code> , <code>rolling_kurt</code>	Moving window skewness (3rd moment) and kurtosis (4th moment), respectively.
<code>rolling_min</code> , <code>rolling_max</code>	Moving window minimum and maximum.
<code>rolling_quantile</code>	Moving window score at percentile/sample quantile.
<code>rolling_corr</code> , <code>rolling_cov</code>	Moving window correlation and covariance.
<code>rolling_apply</code>	Apply generic array function over a moving window.
<code>ewma</code>	Exponentially-weighted moving average.
<code>ewmvar</code> , <code>ewmstd</code>	Exponentially-weighted moving variance and standard deviation.
<code>ewmcov</code>	Exponentially-weighted moving correlation and covariance.



bottleneck, a Python library by Keith Goodman, provides an alternate implementation of `NaN`-friendly moving window functions and may be worth looking at depending on your application.

## Exponentially-weighted functions

An alternative to using a static window size with equally-weighted observations is to specify a constant *decay factor* to give more weight to more recent observations. In mathematical terms, if  $ma_t$  is the moving average result at time  $t$  and  $x$  is the time series in question, each value in the result is computed as  $ma_t = a * ma_{t-1} + (a - 1) * x_{t-t}$ , where  $a$  is the decay factor. There are a couple of ways to specify the decay factor, a popular one is using a *span*, which makes the result comparable to a simple moving window function with window size equal to the span.

Since an exponentially-weighted statistic places more weight on more recent observations, it “adapts” faster to changes compared with the equal-weighted version. Here’s an example comparing a 60-day moving average of Apple’s stock price with an EW moving average with `span=60` (see [Figure 10-11](#)):

```
fig, axes = plt.subplots(nrows=2, ncols=1, sharex=True, sharey=True,
                       figsize=(12, 7))

aapl_px = close_px.AAPL['2005':'2009']

ma60 = pd.rolling_mean(aapl_px, 60, min_periods=50)
ewma60 = pd.ewma(aapl_px, span=60)

aapl_px.plot(style='k-', ax=axes[0])
ma60.plot(style='k--', ax=axes[0])
aapl_px.plot(style='k-', ax=axes[1])
ewma60.plot(style='k--', ax=axes[1])
axes[0].set_title('Simple MA')
axes[1].set_title('Exponentially-weighted MA')
```

## Binary Moving Window Functions

Some statistical operators, like correlation and covariance, need to operate on two time series. As an example, financial analysts are often interested in a stock’s correlation to a benchmark index like the S&P 500. We can compute that by computing the percent changes and using `rolling_corr` (see [Figure 10-12](#)):

```
In [570]: spx_rets = spx_px / spx_px.shift(1) - 1
In [571]: returns = close_px.pct_change()
In [572]: corr = pd.rolling_corr(returns.AAPL, spx_rets, 125, min_periods=100)
In [573]: corr.plot()
```

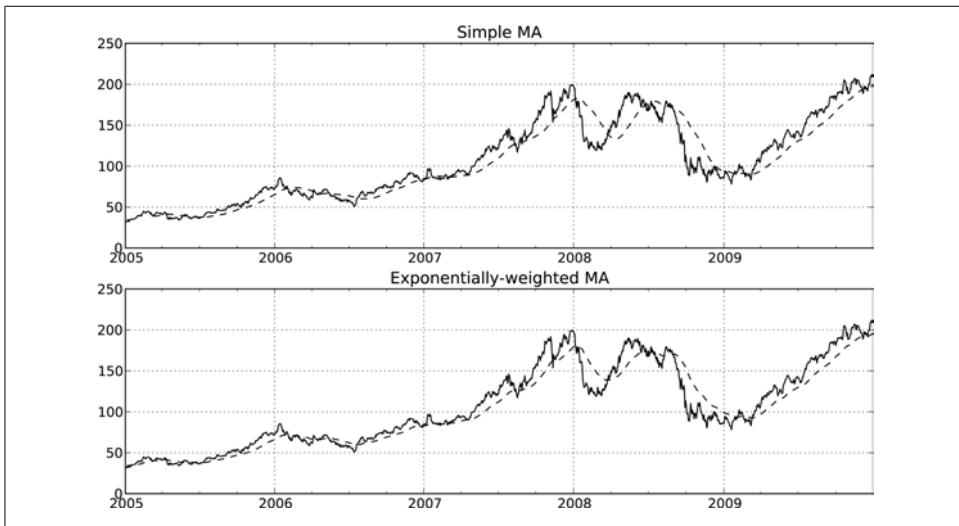


Figure 10-11. Simple moving average versus exponentially-weighted

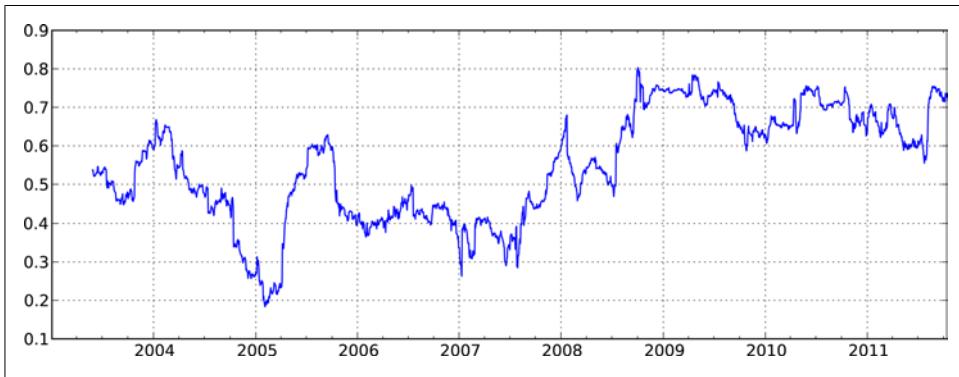


Figure 10-12. Six-month AAPL return correlation to S&P 500

Suppose you wanted to compute the correlation of the S&P 500 index with many stocks at once. Writing a loop and creating a new DataFrame would be easy but maybe get repetitive, so if you pass a TimeSeries and a DataFrame, a function like `rolling_corr` will compute the correlation of the TimeSeries (`spx_rets` in this case) with each column in the DataFrame. See Figure 10-13 for the plot of the result:

```
In [575]: corr = pd.rolling_corr(returns, spx_rets, 125, min_periods=100)
```

```
In [576]: corr.plot()
```

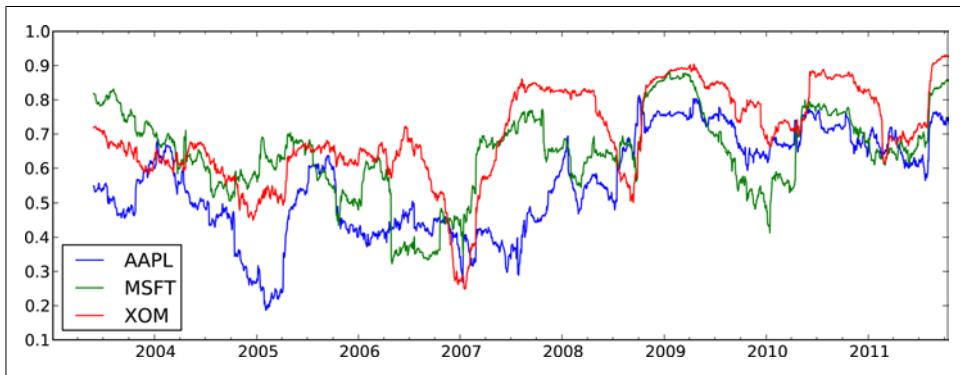


Figure 10-13. Six-month return correlations to S&P 500

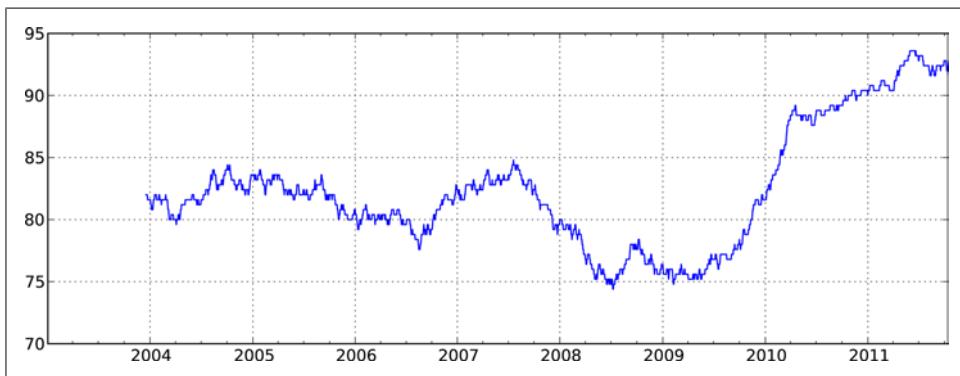


Figure 10-14. Percentile rank of 2% AAPL return over 1 year window

## User-Defined Moving Window Functions

The `rolling_apply` function provides a means to apply an array function of your own devising over a moving window. The only requirement is that the function produce a single value (a reduction) from each piece of the array. For example, while we can compute sample quantiles using `rolling_quantile`, we might be interested in the percentile rank of a particular value over the sample. The `scipy.stats.percentileofscore` function does just this:

```
In [578]: from scipy.stats import percentileofscore
In [579]: score_at_2percent = lambda x: percentileofscore(x, 0.02)
In [580]: result = pd.rolling_apply(returns.AAPL, 250, score_at_2percent)
In [581]: result.plot()
```

## Performance and Memory Usage Notes

Timestamps and periods are represented as 64-bit integers using NumPy's `datetime64` dtype. This means that for each data point, there is an associated 8 bytes of memory per timestamp. Thus, a time series with 1 million `float64` data points has a memory footprint of approximately 16 megabytes. Since pandas makes every effort to share indexes among time series, creating views on existing time series do not cause any more memory to be used. Additionally, indexes for lower frequencies (daily and up) are stored in a central cache, so that any fixed-frequency index is a view on the date cache. Thus, if you have a large collection of low-frequency time series, the memory footprint of the indexes will not be as significant.

Performance-wise, pandas has been highly optimized for data alignment operations (the behind-the-scenes work of differently indexed `ts1 + ts2`) and resampling. Here is an example of aggregating 10MM data points to OHLC:

```
In [582]: rng = pd.date_range('1/1/2000', periods=10000000, freq='10ms')

In [583]: ts = Series(np.random.randn(len(rng)), index=rng)

In [584]: ts
Out[584]:
2000-01-01 00:00:00      -1.402235
2000-01-01 00:00:00.010000    2.424667
2000-01-01 00:00:00.020000   -1.956042
2000-01-01 00:00:00.030000   -0.897339
...
2000-01-02 03:46:39.960000    0.495530
2000-01-02 03:46:39.970000    0.574766
2000-01-02 03:46:39.980000    1.348374
2000-01-02 03:46:39.990000    0.665034
Freq: 10L, Length: 10000000

In [585]: ts.resample('15min', how='ohlc')
Out[585]:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 113 entries, 2000-01-01 00:00:00 to 2000-01-02 04:00:00
Freq: 15T
Data columns:
open     113 non-null values
high     113 non-null values
low     113 non-null values
close    113 non-null values
dtypes: float64(4)

In [586]: %timeit ts.resample('15min', how='ohlc')
10 loops, best of 3: 61.1 ms per loop
```

The runtime may depend slightly on the relative size of the aggregated result; higher frequency aggregates unsurprisingly take longer to compute:

```
In [587]: rng = pd.date_range('1/1/2000', periods=10000000, freq='1s')
```

```
In [588]: ts = Series(np.random.randn(len(rng)), index=rng)
```

```
In [589]: %timeit ts.resample('15s', how='ohlc')
1 loops, best of 3: 88.2 ms per loop
```

It's possible that by the time you read this, the performance of these algorithms may be even further improved. As an example, there are currently no optimizations for conversions between regular frequencies, but that would be fairly straightforward to do.

# **Financial and Economic Data Applications**

The use of Python in the financial industry has been increasing rapidly since 2005, led largely by the maturation of libraries (like NumPy and pandas) and the availability of skilled Python programmers. Institutions have found that Python is well-suited both as an interactive analysis environment as well as enabling robust systems to be developed often in a fraction of the time it would have taken in Java or C++. Python is also an ideal glue layer; it is easy to build Python interfaces to legacy libraries built in C or C++.

While the field of financial analysis is broad enough to fill an entire book, I hope to show you how the tools in this book can be applied to a number of specific problems in finance. As with other research and analysis domains, too much programming effort is often spent wrangling data rather than solving the core modeling and research problems. I personally got started building pandas in 2008 while grappling with inadequate data tools.

In these examples, I'll use the term *cross-section* to refer to data at a fixed point in time. For example, the closing prices of all the stocks in the S&P 500 index on a particular date form a cross-section. Cross-sectional data at multiple points in time over multiple data items (for example, prices together with volume) form a *panel*. Panel data can either be represented as a hierarchically-indexed DataFrame or using the three-dimensional Panel pandas object.

## **Data Munging Topics**

Many helpful data munging tools for financial applications are spread across the earlier chapters. Here I'll highlight a number of topics as they relate to this problem domain.

# Data Wrangling: Clean, Transform, Merge, Reshape

Much of the programming work in data analysis and modeling is spent on data preparation: loading, cleaning, transforming, and rearranging. Sometimes the way that data is stored in files or databases is not the way you need it for a data processing application. Many people choose to do ad hoc processing of data from one form to another using a general purpose programming, like Python, Perl, R, or Java, or UNIX text processing tools like sed or awk. Fortunately, pandas along with the Python standard library provide you with a high-level, flexible, and high-performance set of core manipulations and algorithms to enable you to wrangle data into the right form without much trouble.

If you identify a type of data manipulation that isn't anywhere in this book or elsewhere in the pandas library, feel free to suggest it on the mailing list or GitHub site. Indeed, much of the design and implementation of pandas has been driven by the needs of real world applications.

## Combining and Merging Data Sets

Data contained in pandas objects can be combined together in a number of built-in ways:

- `pandas.merge` connects rows in DataFrames based on one or more keys. This will be familiar to users of SQL or other relational databases, as it implements database *join* operations.
- `pandas.concat` glues or stacks together objects along an axis.
- `combine_first` instance method enables splicing together overlapping data to fill in missing values in one object with values from another.

I will address each of these and give a number of examples. They'll be utilized in examples throughout the rest of the book.

## Database-style DataFrame Merges

*Merge* or *join* operations combine data sets by linking rows using one or more *keys*. These operations are central to relational databases. The `merge` function in pandas is the main entry point for using these algorithms on your data.

Let's start with a simple example:

```
In [15]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                  'data1': range(7)})

In [16]: df2 = DataFrame({'key': ['a', 'b', 'd'],
   ....:                  'data2': range(3)})

In [17]: df1           In [18]: df2
Out[17]:           Out[18]:
   data1  key           data2  key
   0      0   b           0      0   a
   1      1   b           1      1   b
   2      2   a           2      2   d
   3      3   c
   4      4   a
   5      5   a
   6      6   b
```

This is an example of a *many-to-one* merge situation; the data in `df1` has multiple rows labeled `a` and `b`, whereas `df2` has only one row for each value in the `key` column. Calling `merge` with these objects we obtain:

```
In [19]: pd.merge(df1, df2)
Out[19]:
   data1  key  data2
   0      2   a      0
   1      4   a      0
   2      5   a      0
   3      0   b      1
   4      1   b      1
   5      6   b      1
```

Note that I didn't specify which column to join on. If not specified, `merge` uses the overlapping column names as the keys. It's a good practice to specify explicitly, though:

```
In [20]: pd.merge(df1, df2, on='key')
Out[20]:
   data1  key  data2
   0      2   a      0
   1      4   a      0
   2      5   a      0
   3      0   b      1
   4      1   b      1
   5      6   b      1
```

If the column names are different in each object, you can specify them separately:

```
In [21]: df3 = DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                  'data1': range(7)})
```

```
In [22]: df4 = DataFrame({'rkey': ['a', 'b', 'd'],
   ....:                  'data2': range(3)})

In [23]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[23]:
   data1  lkey  data2  rkey
0      2     a      0     a
1      4     a      0     a
2      5     a      0     a
3      0     b      1     b
4      1     b      1     b
5      6     b      1     b
```

You probably noticed that the 'c' and 'd' values and associated data are missing from the result. By default `merge` does an '`inner`' join; the keys in the result are the intersection. Other possible options are '`left`', '`right`', and '`outer`'. The outer join takes the union of the keys, combining the effect of applying both left and right joins:

```
In [24]: pd.merge(df1, df2, how='outer')
Out[24]:
   data1  key  data2
0      2     a      0
1      4     a      0
2      5     a      0
3      0     b      1
4      1     b      1
5      6     b      1
6      3     c    NaN
7     NaN     d      2
```

*Many-to-many* merges have well-defined though not necessarily intuitive behavior. Here's an example:

```
In [25]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   ....:                  'data1': range(6)})

In [26]: df2 = DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
   ....:                  'data2': range(5)})
```

```
In [27]: df1
Out[27]:
   data1  key
0      0     b
1      1     b
2      2     a
3      3     c
4      4     a
5      5     b
```

```
In [28]: df2
Out[28]:
   data2  key
0      0     a
1      1     b
2      2     a
3      3     b
4      4     d
```

```
In [29]: pd.merge(df1, df2, on='key', how='left')
Out[29]:
   data1  key  data2
0      2     a      0
1      2     a      2
```

```

2      4    a    0
3      4    a    2
4      0    b    1
5      0    b    3
6      1    b    1
7      1    b    3
8      5    b    1
9      5    b    3
10     3    c    NaN

```

Many-to-many joins form the Cartesian product of the rows. Since there were 3 'b' rows in the left DataFrame and 2 in the right one, there are 6 'b' rows in the result. The join method only affects the distinct key values appearing in the result:

```

In [30]: pd.merge(df1, df2, how='inner')
Out[30]:
   data1  key  data2
0      2    a    0
1      2    a    2
2      4    a    0
3      4    a    2
4      0    b    1
5      0    b    3
6      1    b    1
7      1    b    3
8      5    b    1
9      5    b    3

```

To merge with multiple keys, pass a list of column names:

```

In [31]: left = DataFrame({'key1': ['foo', 'foo', 'bar'],
....:                   'key2': ['one', 'two', 'one'],
....:                   'lval': [1, 2, 3]})

In [32]: right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
....:                   'key2': ['one', 'one', 'one', 'two'],
....:                   'rval': [4, 5, 6, 7]})

In [33]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
Out[33]:
   key1  key2  lval  rval
0  bar   one     3     6
1  bar   two    NaN     7
2  foo   one     1     4
3  foo   one     1     5
4  foo   two     2    NaN

```

To determine which key combinations will appear in the result depending on the choice of merge method, think of the multiple keys as forming an array of tuples to be used as a single join key (even though it's not actually implemented that way).



When joining columns-on-columns, the indexes on the passed DataFrame objects are discarded.

A last issue to consider in merge operations is the treatment of overlapping column names. While you can address the overlap manually (see the later section on renaming axis labels), `merge` has a `suffixes` option for specifying strings to append to overlapping names in the left and right DataFrame objects:

```
In [34]: pd.merge(left, right, on='key1')
Out[34]:
   key1  key2_x  lval  key2_y  rval
0  bar      one     3    one     6
1  bar      one     3   two     7
2  foo      one     1    one     4
3  foo      one     1    one     5
4  foo     two     2    one     4
5  foo     two     2    one     5

In [35]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
Out[35]:
   key1  key2_left  lval  key2_right  rval
0  bar      one     3        one     6
1  bar      one     3       two     7
2  foo      one     1        one     4
3  foo      one     1        one     5
4  foo     two     2        one     4
5  foo     two     2        one     5
```

See [Table 7-1](#) for an argument reference on `merge`. Joining on index is the subject of the next section.

*Table 7-1. merge function arguments*

Argument	Description
left	DataFrame to be merged on the left side
right	DataFrame to be merged on the right side
how	One of 'inner', 'outer', 'left' or 'right'. 'inner' by default
on	Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names in <code>left</code> and <code>right</code> as the join keys
left_on	Columns in <code>left</code> DataFrame to use as join keys
right_on	Analogous to <code>left_on</code> for <code>left</code> DataFrame
left_index	Use row index in <code>left</code> as its join key (or keys, if a MultiIndex)
right_index	Analogous to <code>left_index</code>
sort	Sort merged data lexicographically by join keys; True by default. Disable to get better performance in some cases on large datasets
suffixes	Tuple of string values to append to column names in case of overlap; defaults to ('_x', '_y'). For example, if 'data' in both DataFrame objects, would appear as 'data_x' and 'data_y' in result
copy	If False, avoid copying data into resulting data structure in some exceptional cases. By default always copies

## Merging on Index

In some cases, the merge key or keys in a DataFrame will be found in its index. In this case, you can pass `left_index=True` or `right_index=True` (or both) to indicate that the index should be used as the merge key:

```
In [36]: left1 = DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
....:                      'value': range(6)})  
  
In [37]: right1 = DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])  
  
In [38]: left1           In [39]: right1
Out[38]:                Out[39]:
   key  value            group_val
0    a     0              a      3.5
1    b     1              b      7.0
2    a     2
3    a     3
4    b     4
5    c     5  
  
In [40]: pd.merge(left1, right1, left_on='key', right_index=True)
Out[40]:
   key  value  group_val
0    a     0      3.5
1    a     2      3.5
2    a     3      3.5
3    b     1      7.0
4    b     4      7.0
```

Since the default merge method is to intersect the join keys, you can instead form the union of them with an outer join:

```
In [41]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
Out[41]:
   key  value  group_val
0    a     0      3.5
1    a     2      3.5
2    a     3      3.5
3    b     1      7.0
4    b     4      7.0
5    c     5      NaN
```

With hierarchically-indexed data, things are a bit more complicated:

```
In [42]: lefth = DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
....:                      'key2': [2000, 2001, 2002, 2001, 2002],
....:                      'data': np.arange(5.)})  
  
In [43]: righth = DataFrame(np.arange(12).reshape((6, 2)),
....:                        index=[['Nevada', 'Nevada', 'Ohio', 'Ohio', 'Ohio', 'Ohio'],
....:                               [2001, 2000, 2000, 2000, 2001, 2002]],
....:                        columns=['event1', 'event2'])  
  
In [44]: lefth           In [45]: righth
Out[44]:                Out[45]:
```

	data	key1	key2		event1	event2
0	0	Ohio	2000	Nevada	2001	0
1	1	Ohio	2001		2000	1
2	2	Ohio	2002	Ohio	2000	2
3	3	Nevada	2001		2000	3
4	4	Nevada	2002		2001	4
					2002	5
						6
						7
						8
						9
						10
						11

In this case, you have to indicate multiple columns to merge on as a list (pay attention to the handling of duplicate index values):

```
In [46]: pd.merge(left, right, left_on=['key1', 'key2'], right_index=True)
```

```
Out[46]:
```

	data	key1	key2	event1	event2
3	3	Nevada	2001	0	1
0	0	Ohio	2000	4	5
0	0	Ohio	2000	6	7
1	1	Ohio	2001	8	9
2	2	Ohio	2002	10	11

```
In [47]: pd.merge(left, right, left_on=['key1', 'key2'],
....:                 right_index=True, how='outer')
```

```
Out[47]:
```

	data	key1	key2	event1	event2
4	NaN	Nevada	2000	2	3
3	3	Nevada	2001	0	1
4	4	Nevada	2002	NaN	NaN
0	0	Ohio	2000	4	5
0	0	Ohio	2000	6	7
1	1	Ohio	2001	8	9
2	2	Ohio	2002	10	11

Using the indexes of both sides of the merge is also not an issue:

```
In [48]: left2 = DataFrame([[1., 2.], [3., 4.], [5., 6.]], index=['a', 'c', 'e'],
....:                      columns=['Ohio', 'Nevada'])
```

```
In [49]: right2 = DataFrame([[7., 8.], [9., 10.], [11., 12.], [13., 14.]],
....:                      index=['b', 'c', 'd', 'e'], columns=['Missouri', 'Alabama'])
```

```
In [50]: left2
```

```
Out[50]:
```

	Ohio	Nevada	Missouri	Alabama
a	1	2	b	7
c	3	4	c	9
e	5	6	d	11
			e	13
				14

```
In [51]: right2
```

```
Out[51]:
```

```
In [52]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
```

```
Out[52]:
```

	Ohio	Nevada	Missouri	Alabama
a	1	2	NaN	NaN
b	NaN	NaN	7	8
c	3	4	9	10
d	NaN	NaN	11	12
e	5	6	13	14

DataFrame has a more convenient `join` instance for merging by index. It can also be used to combine together many DataFrame objects having the same or similar indexes but non-overlapping columns. In the prior example, we could have written:

```
In [53]: left2.join(right2, how='outer')
Out[53]:
    Ohio  Nevada  Missouri  Alabama
a      1        2       NaN      NaN
b     NaN      NaN        7       8
c      3        4       9       10
d     NaN      NaN       11      12
e      5        6      13      14
```

In part for legacy reasons (much earlier versions of pandas), DataFrame's `join` method performs a left join on the join keys. It also supports joining the index of the passed DataFrame on one of the columns of the calling DataFrame:

```
In [54]: left1.join(right1, on='key')
Out[54]:
   key  value  group_val
0   a      0      3.5
1   b      1      7.0
2   a      2      3.5
3   a      3      3.5
4   b      4      7.0
5   c      5      NaN
```

Lastly, for simple index-on-index merges, you can pass a list of DataFrames to `join` as an alternative to using the more general `concat` function described below:

```
In [55]: another = DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
....:                         index=['a', 'c', 'e', 'f'], columns=['New York', 'Oregon'])
```

```
In [56]: left2.join([right2, another])
Out[56]:
    Ohio  Nevada  Missouri  Alabama  New York  Oregon
a      1        2       NaN      NaN       7       8
c      3        4       9       10       9      10
e      5        6      13      14      11      12
```

```
In [57]: left2.join([right2, another], how='outer')
Out[57]:
    Ohio  Nevada  Missouri  Alabama  New York  Oregon
a      1        2       NaN      NaN       7       8
b     NaN      NaN        7       8      NaN      NaN
c      3        4       9       10       9      10
d     NaN      NaN       11      12      NaN      NaN
e      5        6      13      14      11      12
f     NaN      NaN      NaN      NaN      16      17
```

## Concatenating Along an Axis

Another kind of data combination operation is alternatively referred to as concatenation, binding, or stacking. NumPy has a `concatenate` function for doing this with raw NumPy arrays:

```
In [58]: arr = np.arange(12).reshape((3, 4))
```

```
In [59]: arr
```

```
Out[59]:
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
In [60]: np.concatenate([arr, arr], axis=1)
```

```
Out[60]:
```

```
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

In the context of pandas objects such as Series and DataFrame, having labeled axes enable you to further generalize array concatenation. In particular, you have a number of additional things to think about:

- If the objects are indexed differently on the other axes, should the collection of axes be unioned or intersected?
- Do the groups need to be identifiable in the resulting object?
- Does the concatenation axis matter at all?

The `concat` function in pandas provides a consistent way to address each of these concerns. I'll give a number of examples to illustrate how it works. Suppose we have three Series with no index overlap:

```
In [61]: s1 = Series([0, 1], index=['a', 'b'])
```

```
In [62]: s2 = Series([2, 3, 4], index=['c', 'd', 'e'])
```

```
In [63]: s3 = Series([5, 6], index=['f', 'g'])
```

Calling `concat` with these object in a list glues together the values and indexes:

```
In [64]: pd.concat([s1, s2, s3])
```

```
Out[64]:
```

```
a    0
b    1
c    2
d    3
e    4
f    5
g    6
```

By default `concat` works along `axis=0`, producing another Series. If you pass `axis=1`, the result will instead be a DataFrame (`axis=1` is the columns):

```
In [65]: pd.concat([s1, s2, s3], axis=1)
Out[65]:
   0   1   2
a  0  NaN  NaN
b  1  NaN  NaN
c  NaN  2  NaN
d  NaN  3  NaN
e  NaN  4  NaN
f  NaN  NaN  5
g  NaN  NaN  6
```

In this case there is no overlap on the other axis, which as you can see is the sorted union (the '`outer`' join) of the indexes. You can instead intersect them by passing `join='inner'`:

```
In [66]: s4 = pd.concat([s1 * 5, s3])
```

```
In [67]: pd.concat([s1, s4], axis=1)
Out[67]:
   0   1
a  0  0
b  1  5
f  NaN  5
g  NaN  6
```

```
In [68]: pd.concat([s1, s4], axis=1, join='inner')
Out[68]:
   0   1
a  0  0
b  1  5
```

You can even specify the axes to be used on the other axes with `join_axes`:

```
In [69]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
Out[69]:
   0   1
a  0  0
c  NaN  NaN
b  1  5
e  NaN  NaN
```

One issue is that the concatenated pieces are not identifiable in the result. Suppose instead you wanted to create a hierarchical index on the concatenation axis. To do this, use the `keys` argument:

```
In [70]: result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])
```

```
In [71]: result
Out[71]:
one    a    0
      b    1
two    a    0
      b    1
three   f    5
        g    6

# Much more on the unstack function later
In [72]: result.unstack()
Out[72]:
```

```
      a   b   f   g
one    0   1  NaN  NaN
two    0   1  NaN  NaN
three  NaN  NaN   5   6
```

In the case of combining Series along `axis=1`, the keys become the DataFrame column headers:

```
In [73]: pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
Out[73]:
      one   two   three
a      0   NaN   NaN
b      1   NaN   NaN
c     NaN   2   NaN
d     NaN   3   NaN
e     NaN   4   NaN
f     NaN  NaN   5
g     NaN  NaN   6
```

The same logic extends to DataFrame objects:

```
In [74]: df1 = DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],
....:                   columns=['one', 'two'])

In [75]: df2 = DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
....:                   columns=['three', 'four'])

In [76]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
Out[76]:
      level1      level2
      one   two   three   four
a      0   1      5      6
b      2   3     NaN  NaN
c      4   5      7      8
```

If you pass a dict of objects instead of a list, the dict's keys will be used for the `keys` option:

```
In [77]: pd.concat({'level1': df1, 'level2': df2}, axis=1)
Out[77]:
      level1      level2
      one   two   three   four
a      0   1      5      6
b      2   3     NaN  NaN
c      4   5      7      8
```

There are a couple of additional arguments governing how the hierarchical index is created (see [Table 7-2](#)):

```
In [78]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
....:                 names=['upper', 'lower'])
Out[78]:
      upper  level1      level2
      lower   one   two   three   four
a          0   1      5      6
b          2   3     NaN  NaN
c          4   5      7      8
```

A last consideration concerns DataFrames in which the row index is not meaningful in the context of the analysis:

```
In [79]: df1 = DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])
```

```
In [80]: df2 = DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])
```

```
In [81]: df1
```

```
Out[81]:
```

	a	b	c	d
0	-0.204708	0.478943	-0.519439	-0.555730
1	1.965781	1.393406	0.092908	0.281746
2	0.769023	1.246435	1.007189	-1.296221

```
In [82]: df2
```

```
Out[82]:
```

	b	d	a
0	0.274992	0.228913	1.352917
1	0.886429	-2.001637	-0.371843

In this case, you can pass `ignore_index=True`:

```
In [83]: pd.concat([df1, df2], ignore_index=True)
```

```
Out[83]:
```

	a	b	c	d
0	-0.204708	0.478943	-0.519439	-0.555730
1	1.965781	1.393406	0.092908	0.281746
2	0.769023	1.246435	1.007189	-1.296221
3	1.352917	0.274992	NaN	0.228913
4	-0.371843	0.886429	NaN	-2.001637

Table 7-2. concat function arguments

Argument	Description
objs	List or dict of pandas objects to be concatenated. The only required argument
axis	Axis to concatenate along; defaults to 0
join	One of 'inner', 'outer', defaulting to 'outer'; whether to intersection (inner) or union (outer) together indexes along the other axes
join_axes	Specific indexes to use for the other n-1 axes instead of performing union/intersection logic
keys	Values to associate with objects being concatenated, forming a hierarchical index along the concatenation axis. Can either be a list or array of arbitrary values, an array of tuples, or a list of arrays (if multiple level arrays passed in levels)
levels	Specific indexes to use as hierarchical index level or levels if keys passed
names	Names for created hierarchical levels if keys and / or levels passed
verify_integrity	Check new axis in concatenated object for duplicates and raise exception if so. By default (False) allows duplicates
ignore_index	Do not preserve indexes along concatenation axis, instead producing a new range(total_length) index

## Combining Data with Overlap

Another data combination situation can't be expressed as either a merge or concatenation operation. You may have two datasets whose indexes overlap in full or part. As a motivating example, consider NumPy's `where` function, which expressed a vectorized if-else:

```
In [84]: a = Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],
....:                  index=['f', 'e', 'd', 'c', 'b', 'a'])

In [85]: b = Series(np.arange(len(a), dtype=np.float64),
....:                  index=['f', 'e', 'd', 'c', 'b', 'a'])

In [86]: b[-1] = np.nan

In [87]: a      In [88]: b      In [89]: np.where(pd.isnull(a), b, a)
Out[87]:      Out[88]:      Out[89]:
f    NaN        f    0          f    0.0
e    2.5        e    1          e    2.5
d    NaN        d    2          d    2.0
c    3.5        c    3          c    3.5
b    4.5        b    4          b    4.5
a    NaN        a    NaN       a    NaN
```

Series has a `combine_first` method, which performs the equivalent of this operation plus data alignment:

```
In [90]: b[:-2].combine_first(a[2:])
Out[90]:
a    NaN
b    4.5
c    3.0
d    2.0
e    1.0
f    0.0
```

With DataFrames, `combine_first` naturally does the same thing column by column, so you can think of it as “patching” missing data in the calling object with data from the object you pass:

```
In [91]: df1 = DataFrame({'a': [1., np.nan, 5., np.nan],
....:                      'b': [np.nan, 2., np.nan, 6.],
....:                      'c': range(2, 18, 4)})

In [92]: df2 = DataFrame({'a': [5., 4., np.nan, 3., 7.],
....:                      'b': [np.nan, 3., 4., 6., 8.]})

In [93]: df1.combine_first(df2)
Out[93]:
   a   b   c
0  1  NaN  2
1  4   2   6
2  5   4  10
3  3   6  14
4  7   8  NaN
```

## Reshaping and Pivoting

There are a number of fundamental operations for rearranging tabular data. These are alternately referred to as *reshape* or *pivot* operations.

## Reshaping with Hierarchical Indexing

Hierarchical indexing provides a consistent way to rearrange data in a DataFrame. There are two primary actions:

- `stack`: this “rotates” or pivots from the columns in the data to the rows
- `unstack`: this pivots from the rows into the columns

I'll illustrate these operations through a series of examples. Consider a small DataFrame with string arrays as row and column indexes:

```
In [94]: data = DataFrame(np.arange(6).reshape((2, 3)),  
....:                      index=pd.Index(['Ohio', 'Colorado'], name='state'),  
....:                      columns=pd.Index(['one', 'two', 'three'], name='number'))  
  
In [95]: data  
Out[95]:  
number    one   two   three  
state  
Ohio      0     1     2  
Colorado  3     4     5
```

Using the `stack` method on this data pivots the columns into the rows, producing a Series:

```
In [96]: result = data.stack()  
  
In [97]: result  
Out[97]:  
state    number  
Ohio      one      0  
          two      1  
          three    2  
Colorado  one      3  
          two      4  
          three    5
```

From a hierarchically-indexed Series, you can rearrange the data back into a DataFrame with `unstack`:

```
In [98]: result.unstack()  
Out[98]:  
number    one   two   three  
state  
Ohio      0     1     2  
Colorado  3     4     5
```

By default the innermost level is unstacked (same with `stack`). You can unstack a different level by passing a level number or name:

```
In [99]: result.unstack(0)           In [100]: result.unstack('state')  
Out[99]:  
state  Ohio  Colorado  
number  
one      0       3  
                    state  Ohio  Colorado  
                    number  
                    one      0       3
```

```

two      1      4      two      1      4
three    2      5      three    2      5

```

Unstacking might introduce missing data if all of the values in the level aren't found in each of the subgroups:

```
In [101]: s1 = Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
```

```
In [102]: s2 = Series([4, 5, 6], index=['c', 'd', 'e'])
```

```
In [103]: data2 = pd.concat([s1, s2], keys=['one', 'two'])
```

```
In [104]: data2.unstack()
```

```
Out[104]:
```

	a	b	c	d	e
one	0	1	2	3	NaN
two	NaN	NaN	4	5	6

Stacking filters out missing data by default, so the operation is easily invertible:

```
In [105]: data2.unstack().stack()
```

```
Out[105]:
```

	a	b	c	d	e
one	0	1	2	3	3
two	4	5	6	5	6

```
In [106]: data2.unstack().stack(dropna=False)
```

```
Out[106]:
```

	a	b	c	d	e
one	0	1	2	3	NaN
two	a	NaN	b	NaN	c
			d	5	4
			e	6	5

When unstacking in a DataFrame, the level unstacked becomes the lowest level in the result:

```
In [107]: df = DataFrame({'left': result, 'right': result + 5},
.....:                      columns=pd.Index(['left', 'right'], name='side'))
```

```
In [108]: df
```

```
Out[108]:
```

side	left	right	
state	number		
Ohio	one	0	5
	two	1	6
	three	2	7
Colorado	one	3	8
	two	4	9
	three	5	10

```
In [109]: df.unstack('state')
```

```
Out[109]:
```

side	left	right		
state	Ohio	Colorado	Ohio	Colorado
number				
one	0	3	5	8
two	1	4	6	9

```
In [110]: df.unstack('state').stack('side')
```

```
Out[110]:
```

state	Ohio	Colorado	
number	side		
one	left	0	3
	right	5	8
two	left	1	4

three	2	5	7	10		right	6	9
					three	left	2	5
						right	7	10

## Pivoting “long” to “wide” Format

A common way to store multiple time series in databases and CSV is in so-called *long* or *stacked* format:

```
In [116]: ldata[:10]
Out[116]:
      date      item    value
0 1959-03-31 00:00:00  realgdp  2710.349
1 1959-03-31 00:00:00      infl     0.000
2 1959-03-31 00:00:00     unemp    5.800
3 1959-06-30 00:00:00  realgdp  2778.801
4 1959-06-30 00:00:00      infl    2.340
5 1959-06-30 00:00:00     unemp    5.100
6 1959-09-30 00:00:00  realgdp  2775.488
7 1959-09-30 00:00:00      infl    2.740
8 1959-09-30 00:00:00     unemp    5.300
9 1959-12-31 00:00:00  realgdp  2785.204
```

Data is frequently stored this way in relational databases like MySQL as a fixed schema (column names and data types) allows the number of distinct values in the `item` column to increase or decrease as data is added or deleted in the table. In the above example `date` and `item` would usually be the primary keys (in relational database parlance), offering both relational integrity and easier joins and programmatic queries in many cases. The downside, of course, is that the data may not be easy to work with in long format; you might prefer to have a DataFrame containing one column per distinct `item` value indexed by timestamps in the `date` column. DataFrame’s `pivot` method performs exactly this transformation:

```
In [117]: pivoted = ldata.pivot('date', 'item', 'value')

In [118]: pivoted.head()
Out[118]:
      item      infl    realgdp   unemp
      date
1959-03-31  0.00  2710.349    5.8
1959-06-30  2.34  2778.801    5.1
1959-09-30  2.74  2775.488    5.3
1959-12-31  0.27  2785.204    5.6
1960-03-31  2.31  2847.699    5.2
```

The first two values passed are the columns to be used as the row and column index, and finally an optional value column to fill the DataFrame. Suppose you had two value columns that you wanted to reshape simultaneously:

```
In [119]: ldata['value2'] = np.random.randn(len(ldata))

In [120]: ldata[:10]
Out[120]:
```

```

          date      item    value   value2
0 1959-03-31 00:00:00  realgdp  2710.349  1.669025
1 1959-03-31 00:00:00     infl     0.000 -0.438570
2 1959-03-31 00:00:00    unemp     5.800 -0.539741
3 1959-06-30 00:00:00  realgdp  2778.801  0.476985
4 1959-06-30 00:00:00     infl     2.340  3.248944
5 1959-06-30 00:00:00    unemp     5.100 -1.021228
6 1959-09-30 00:00:00  realgdp  2775.488 -0.577087
7 1959-09-30 00:00:00     infl     2.740  0.124121
8 1959-09-30 00:00:00    unemp     5.300  0.302614
9 1959-12-31 00:00:00  realgdp  2785.204  0.523772

```

By omitting the last argument, you obtain a DataFrame with hierarchical columns:

```
In [121]: pivoted = ldata.pivot('date', 'item')
```

```
In [122]: pivoted[:5]
```

```
Out[122]:
```

item	value			value2		
	infl	realgdp	unemp	infl	realgdp	unemp
date						
1959-03-31	0.00	2710.349	5.8	-0.438570	1.669025	-0.539741
1959-06-30	2.34	2778.801	5.1	3.248944	0.476985	-1.021228
1959-09-30	2.74	2775.488	5.3	0.124121	-0.577087	0.302614
1959-12-31	0.27	2785.204	5.6	0.000940	0.523772	1.343810
1960-03-31	2.31	2847.699	5.2	-0.831154	-0.713544	-2.370232

```
In [123]: pivoted['value'][:5]
```

```
Out[123]:
```

item	infl	realgdp	unemp
date			
1959-03-31	0.00	2710.349	5.8
1959-06-30	2.34	2778.801	5.1
1959-09-30	2.74	2775.488	5.3
1959-12-31	0.27	2785.204	5.6
1960-03-31	2.31	2847.699	5.2

Note that `pivot` is just a shortcut for creating a hierarchical index using `set_index` and reshaping with `unstack`:

```
In [124]: unstacked = ldata.set_index(['date', 'item']).unstack('item')
```

```
In [125]: unstacked[:7]
```

```
Out[125]:
```

item	value			value2		
	infl	realgdp	unemp	infl	realgdp	unemp
date						
1959-03-31	0.00	2710.349	5.8	-0.438570	1.669025	-0.539741
1959-06-30	2.34	2778.801	5.1	3.248944	0.476985	-1.021228
1959-09-30	2.74	2775.488	5.3	0.124121	-0.577087	0.302614
1959-12-31	0.27	2785.204	5.6	0.000940	0.523772	1.343810
1960-03-31	2.31	2847.699	5.2	-0.831154	-0.713544	-2.370232
1960-06-30	0.14	2834.390	5.2	-0.860757	-1.860761	0.560145
1960-09-30	2.70	2839.022	5.6	0.119827	-1.265934	-1.063512

# Data Transformation

So far in this chapter we've been concerned with rearranging data. Filtering, cleaning, and other transformations are another class of important operations.

## Removing Duplicates

Duplicate rows may be found in a DataFrame for any number of reasons. Here is an example:

```
In [126]: data = DataFrame({'k1': ['one'] * 3 + ['two'] * 4,
.....:                  'k2': [1, 1, 2, 3, 3, 4, 4]})

In [127]: data
Out[127]:
   k1  k2
0  one  1
1  one  1
2  one  2
3  two  3
4  two  3
5  two  4
6  two  4
```

The DataFrame method `duplicated` returns a boolean Series indicating whether each row is a duplicate or not:

```
In [128]: data.duplicated()
Out[128]:
0    False
1     True
2    False
3    False
4     True
5    False
6     True
```

Relatedly, `drop_duplicates` returns a DataFrame where the `duplicated` array is `True`:

```
In [129]: data.drop_duplicates()
Out[129]:
   k1  k2
0  one  1
2  one  2
3  two  3
5  two  4
```

Both of these methods by default consider all of the columns; alternatively you can specify any subset of them to detect duplicates. Suppose we had an additional column of values and wanted to filter duplicates only based on the `'k1'` column:

```
In [130]: data['v1'] = range(7)

In [131]: data.drop_duplicates(['k1'])
```

```
Out[131]:  
      k1  k2  v1  
0   one    1    0  
3   two    3    3
```

duplicated and drop\_duplicates by default keep the first observed value combination. Passing take\_last=True will return the last one:

```
In [132]: data.drop_duplicates(['k1', 'k2'], take_last=True)  
Out[132]:  
      k1  k2  v1  
1   one    1    1  
2   one    2    2  
4   two    3    4  
6   two    4    6
```

## Transforming Data Using a Function or Mapping

For many data sets, you may wish to perform some transformation based on the values in an array, Series, or column in a DataFrame. Consider the following hypothetical data collected about some kinds of meat:

```
In [133]: data = DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'Pastrami',  
.....:                               'corned beef', 'Bacon', 'pastrami', 'honey ham',  
.....:                               'nova lox'],  
.....:                               'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})  
  
In [134]: data  
Out[134]:  
      food  ounces  
0     bacon    4.0  
1  pulled pork    3.0  
2     bacon   12.0  
3    Pastrami    6.0  
4  corned beef    7.5  
5      Bacon    8.0  
6    pastrami    3.0  
7  honey ham    5.0  
8   nova lox    6.0
```

Suppose you wanted to add a column indicating the type of animal that each food came from. Let's write down a mapping of each distinct meat type to the kind of animal:

```
meat_to_animal = {  
    'bacon': 'pig',  
    'pulled pork': 'pig',  
    'pastrami': 'cow',  
    'corned beef': 'cow',  
    'honey ham': 'pig',  
    'nova lox': 'salmon'  
}
```

The `map` method on a Series accepts a function or dict-like object containing a mapping, but here we have a small problem in that some of the meats above are capitalized and others are not. Thus, we also need to convert each value to lower case:

```
In [136]: data['animal'] = data['food'].map(str.lower).map(meat_to_animal)
```

```
In [137]: data
```

```
Out[137]:
```

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	Pastrami	6.0	cow
4	corned beef	7.5	cow
5	Bacon	8.0	pig
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

We could also have passed a function that does all the work:

```
In [138]: data['food'].map(lambda x: meat_to_animal[x.lower()])
```

```
Out[138]:
```

0	pig
1	pig
2	pig
3	cow
4	cow
5	pig
6	cow
7	pig
8	salmon

```
Name: food
```

Using `map` is a convenient way to perform element-wise transformations and other data cleaning-related operations.

## Replacing Values

Filling in missing data with the `fillna` method can be thought of as a special case of more general value replacement. While `map`, as you've seen above, can be used to modify a subset of values in an object, `replace` provides a simpler and more flexible way to do so. Let's consider this Series:

```
In [139]: data = Series([1., -999., 2., -999., -1000., 3.])
```

```
In [140]: data
```

```
Out[140]:
```

0	1
1	-999
2	2
3	-999
4	-1000
5	3

The `-999` values might be sentinel values for missing data. To replace these with NA values that pandas understands, we can use `replace`, producing a new Series:

```
In [141]: data.replace(-999, np.nan)
Out[141]:
0      1
1    NaN
2      2
3    NaN
4   -1000
5      3
```

If you want to replace multiple values at once, you instead pass a list then the substitute value:

```
In [142]: data.replace([-999, -1000], np.nan)
Out[142]:
0      1
1    NaN
2      2
3    NaN
4    NaN
5      3
```

To use a different replacement for each value, pass a list of substitutes:

```
In [143]: data.replace([-999, -1000], [np.nan, 0])
Out[143]:
0      1
1    NaN
2      2
3    NaN
4      0
5      3
```

The argument passed can also be a dict:

```
In [144]: data.replace({-999: np.nan, -1000: 0})
Out[144]:
0      1
1    NaN
2      2
3    NaN
4      0
5      3
```

## Renaming Axis Indexes

Like values in a Series, axis labels can be similarly transformed by a function or mapping of some form to produce new, differently labeled objects. The axes can also be modified in place without creating a new data structure. Here's a simple example:

```
In [145]: data = DataFrame(np.arange(12).reshape((3, 4)),
.....:                   index=['Ohio', 'Colorado', 'New York'],
.....:                   columns=['one', 'two', 'three', 'four'])
```

Like a Series, the axis indexes have a `map` method:

```
In [146]: data.index.map(str.upper)
Out[146]: array([OHIO, COLORADO, NEW YORK], dtype=object)
```

You can assign to `index`, modifying the DataFrame in place:

```
In [147]: data.index = data.index.map(str.upper)
```

```
In [148]: data
Out[148]:
   one  two  three  four
OHIO     0    1      2    3
COLORADO 4    5      6    7
NEW YORK 8    9     10    11
```

If you want to create a transformed version of a data set without modifying the original, a useful method is `rename`:

```
In [149]: data.rename(index=str.title, columns=str.upper)
Out[149]:
   ONE  TWO  THREE  FOUR
Ohio     0    1      2    3
Colorado 4    5      6    7
New York 8    9     10    11
```

Notably, `rename` can be used in conjunction with a dict-like object providing new values for a subset of the axis labels:

```
In [150]: data.rename(index={'OHIO': 'INDIANA'},
.....:           columns={'three': 'peekaboo'})
Out[150]:
   one  two  peekaboo  four
INDIANA  0    1        2    3
COLORADO 4    5        6    7
NEW YORK 8    9       10    11
```

`rename` saves having to copy the DataFrame manually and assign to its `index` and `columns` attributes. Should you wish to modify a data set in place, pass `inplace=True`:

```
# Always returns a reference to a DataFrame
In [151]: _ = data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

In [152]: data
Out[152]:
   one  two  three  four
INDIANA  0    1      2    3
COLORADO 4    5      6    7
NEW YORK 8    9     10    11
```

## Discretization and Binning

Continuous data is often discretized or otherwise separated into “bins” for analysis. Suppose you have data about a group of people in a study, and you want to group them into discrete age buckets:

```
In [153]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Let’s divide these into bins of 18 to 25, 26 to 35, 35 to 60, and finally 60 and older. To do so, you have to use `cut`, a function in pandas:

```
In [154]: bins = [18, 25, 35, 60, 100]
```

```
In [155]: cats = pd.cut(ages, bins)
```

```
In [156]: cats
```

```
Out[156]:
```

```
Categorical:
```

```
array([(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], (18, 25],
       (35, 60], (25, 35], (60, 100], (35, 60], (35, 60], (25, 35]], dtype=object)
Levels (4): Index([(18, 25], (25, 35], (35, 60], (60, 100]], dtype=object)
```

The object pandas returns is a special `Categorical` object. You can treat it like an array of strings indicating the bin name; internally it contains a `levels` array indicating the distinct category names along with a labeling for the `ages` data in the `labels` attribute:

```
In [157]: cats.labels
```

```
Out[157]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1])
```

```
In [158]: cats.levels
```

```
Out[158]: Index([(18, 25], (25, 35], (35, 60], (60, 100]], dtype=object)
```

```
In [159]: pd.value_counts(cats)
```

```
Out[159]:
```

```
(18, 25]      5
(35, 60]      3
(25, 35]      3
(60, 100]     1
```

Consistent with mathematical notation for intervals, a parenthesis means that the side is *open* while the square bracket means it is *closed* (inclusive). Which side is closed can be changed by passing `right=False`:

```
In [160]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)
```

```
Out[160]:
```

```
Categorical:
```

```
array([(18, 26), [18, 26), [18, 26), [26, 36), [18, 26), [18, 26),
       [36, 61), [26, 36), [61, 100], [36, 61), [36, 61), [26, 36]], dtype=object)
Levels (4): Index([(18, 26), [26, 36), [36, 61), [61, 100]], dtype=object)
```

You can also pass your own bin names by passing a list or array to the `labels` option:

```
In [161]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
```

```
In [162]: pd.cut(ages, bins, labels=group_names)
```

```
Out[162]:
```

```
Categorical:  
array([Youth, Youth, Youth, YoungAdult, Youth, Youth, MiddleAged,  
      YoungAdult, Senior, MiddleAged, MiddleAged, YoungAdult], dtype=object)  
Levels (4): Index([Youth, YoungAdult, MiddleAged, Senior], dtype=object)
```

If you pass `cut` a integer number of bins instead of explicit bin edges, it will compute equal-length bins based on the minimum and maximum values in the data. Consider the case of some uniformly distributed data chopped into fourths:

```
In [163]: data = np.random.rand(20)  
  
In [164]: pd.cut(data, 4, precision=2)  
Out[164]:  
Categorical:  
array([(0.45, 0.67], (0.23, 0.45], (0.0037, 0.23], (0.45, 0.67],  
      (0.67, 0.9], (0.45, 0.67], (0.67, 0.9], (0.23, 0.45], (0.23, 0.45],  
      (0.67, 0.9], (0.67, 0.9], (0.67, 0.9], (0.23, 0.45], (0.23, 0.45],  
      (0.23, 0.45], (0.67, 0.9], (0.0037, 0.23], (0.0037, 0.23],  
      (0.23, 0.45], (0.23, 0.45]], dtype=object)  
Levels (4): Index([(0.0037, 0.23], (0.23, 0.45], (0.45, 0.67],  
                  (0.67, 0.9]], dtype=object)
```

A closely related function, `qcut`, bins the data based on sample quantiles. Depending on the distribution of the data, using `cut` will not usually result in each bin having the same number of data points. Since `qcut` uses sample quantiles instead, by definition you will obtain roughly equal-size bins:

```
In [165]: data = np.random.randn(1000) # Normally distributed  
  
In [166]: cats = pd.qcut(data, 4) # Cut into quartiles  
  
In [167]: cats  
Out[167]:  
Categorical:  
array([[-0.022, 0.641], [-3.745, -0.635], (0.641, 3.26], ...,  
      (-0.635, -0.022], (0.641, 3.26], (-0.635, -0.022]], dtype=object)  
Levels (4): Index([-3.745, -0.635], (-0.635, -0.022], (-0.022, 0.641],  
                  (0.641, 3.26]], dtype=object)  
  
In [168]: pd.value_counts(cats)  
Out[168]:  
[-3.745, -0.635]    250  
(0.641, 3.26]      250  
(-0.635, -0.022]   250  
(-0.022, 0.641]    250
```

Similar to `cut` you can pass your own quantiles (numbers between 0 and 1, inclusive):

```
In [169]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])  
Out[169]:  
Categorical:  
array([[-0.022, 1.302], (-1.266, -0.022], (-0.022, 1.302], ...,  
      (-1.266, -0.022], (-0.022, 1.302], (-1.266, -0.022]], dtype=object)  
Levels (4): Index([-3.745, -1.266], (-1.266, -0.022], (-0.022, 1.302],  
                  (1.302, 3.26]], dtype=object)
```

We'll return to `cut` and `qcut` later in the chapter on aggregation and group operations, as these discretization functions are especially useful for quantile and group analysis.

## Detecting and Filtering Outliers

Filtering or transforming outliers is largely a matter of applying array operations. Consider a DataFrame with some normally distributed data:

```
In [170]: np.random.seed(12345)
```

```
In [171]: data = DataFrame(np.random.randn(1000, 4))
```

```
In [172]: data.describe()
```

```
Out[172]:
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.067684	0.067924	0.025598	-0.002298
std	0.998035	0.992106	1.006835	0.996794
min	-3.428254	-3.548824	-3.184377	-3.745356
25%	-0.774890	-0.591841	-0.641675	-0.644144
50%	-0.116401	0.101143	0.002073	-0.013611
75%	0.616366	0.780282	0.680391	0.654328
max	3.366626	2.653656	3.260383	3.927528

Suppose you wanted to find values in one of the columns exceeding three in magnitude:

```
In [173]: col = data[3]
```

```
In [174]: col[np.abs(col) > 3]
```

```
Out[174]:
```

97	3.927528
305	-3.399312
400	-3.745356
Name:	3

To select all rows having a value exceeding 3 or -3, you can use the `any` method on a boolean DataFrame:

```
In [175]: data[(np.abs(data) > 3).any(1)]
```

```
Out[175]:
```

	0	1	2	3
5	-0.539741	0.476985	3.248944	-1.021228
97	-0.774363	0.552936	0.106061	3.927528
102	-0.655054	-0.565230	3.176873	0.959533
305	-2.315555	0.457246	-0.025907	-3.399312
324	0.050188	1.951312	3.260383	0.963301
400	0.146326	0.508391	-0.196713	-3.745356
499	-0.293333	-0.242459	-3.056990	1.918403
523	-3.428254	-0.296336	-0.439938	-0.867165
586	0.275144	1.179227	-3.184377	1.369891
808	-0.362528	-3.548824	1.553205	-2.186301
900	3.366626	-2.372214	0.851010	1.332846

Values can just as easily be set based on these criteria. Here is code to cap values outside the interval -3 to 3:

```
In [176]: data[np.abs(data) > 3] = np.sign(data) * 3
```

```
In [177]: data.describe()
```

```
Out[177]:
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.067623	0.068473	0.025153	-0.002081
std	0.995485	0.990253	1.003977	0.989736
min	-3.000000	-3.000000	-3.000000	-3.000000
25%	-0.774890	-0.591841	-0.641675	-0.644144
50%	-0.116401	0.101143	0.002073	-0.013611
75%	0.616366	0.780282	0.680391	0.654328
max	3.000000	2.653656	3.000000	3.000000

The ufunc `np.sign` returns an array of 1 and -1 depending on the sign of the values.

## Permutation and Random Sampling

Permuting (randomly reordering) a Series or the rows in a DataFrame is easy to do using the `numpy.random.permutation` function. Calling `permutation` with the length of the axis you want to permute produces an array of integers indicating the new ordering:

```
In [178]: df = DataFrame(np.arange(5 * 4).reshape(5, 4))
```

```
In [179]: sampler = np.random.permutation(5)
```

```
In [180]: sampler
```

```
Out[180]: array([1, 0, 2, 3, 4])
```

That array can then be used in `ix`-based indexing or the `take` function:

```
In [181]: df
```

```
Out[181]:
```

```
   0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9   10  11
3  12  13  14  15
4  16  17  18  19
```

```
In [182]: df.take(sampler)
```

```
Out[182]:
```

```
   0   1   2   3
1   4   5   6   7
0   0   1   2   3
2   8   9   10  11
3  12  13  14  15
4  16  17  18  19
```

To select a random subset without replacement, one way is to slice off the first  $k$  elements of the array returned by `permutation`, where  $k$  is the desired subset size. There are much more efficient sampling-without-replacement algorithms, but this is an easy strategy that uses readily available tools:

```
In [183]: df.take(np.random.permutation(len(df))[:3])
```

```
Out[183]:
```

```
   0   1   2   3
1   4   5   6   7
3  12  13  14  15
4  16  17  18  19
```

To generate a sample *with* replacement, the fastest way is to use `np.random.randint` to draw random integers:

```
In [184]: bag = np.array([5, 7, -1, 6, 4])
In [185]: sampler = np.random.randint(0, len(bag), size=10)
In [186]: sampler
Out[186]: array([4, 4, 2, 2, 2, 0, 3, 0, 4, 1])
In [187]: draws = bag.take(sampler)
In [188]: draws
Out[188]: array([ 4,  4, -1, -1,  5,  6,  5,  4,  7])
```

## Computing Indicator/Dummy Variables

Another type of transformation for statistical modeling or machine learning applications is converting a categorical variable into a “dummy” or “indicator” matrix. If a column in a DataFrame has k distinct values, you would derive a matrix or DataFrame containing k columns containing all 1’s and 0’s. pandas has a `get_dummies` function for doing this, though devising one yourself is not difficult. Let’s return to an earlier example DataFrame:

```
In [189]: df = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
.....:                 'data1': range(6)})
In [190]: pd.get_dummies(df['key'])
Out[190]:
   a   b   c
0  0   1   0
1  0   1   0
2  1   0   0
3  0   0   1
4  1   0   0
5  0   1   0
```

In some cases, you may want to add a prefix to the columns in the indicator DataFrame, which can then be merged with the other data. `get_dummies` has a `prefix` argument for doing just this:

```
In [191]: dummies = pd.get_dummies(df['key'], prefix='key')
In [192]: df_with_dummy = df[['data1']].join(dummies)

In [193]: df_with_dummy
Out[193]:
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

If a row in a DataFrame belongs to multiple categories, things are a bit more complicated. Let's return to the MovieLens 1M dataset from earlier in the book:

```
In [194]: mnames = ['movie_id', 'title', 'genres']

In [195]: movies = pd.read_table('ch07/movies.dat', sep='::', header=None,
.....:             names=mnames)

In [196]: movies[:10]
Out[196]:
   movie_id          title           genres
0        1      Toy Story (1995) Animation|Children's|Comedy
1        2       Jumanji (1995) Adventure|Children's|Fantasy
2        3  Grumpier Old Men (1995)            Comedy|Romance
3        4    Waiting to Exhale (1995)            Comedy|Drama
4        5 Father of the Bride Part II (1995)            Comedy
5        6              Heat (1995) Action|Crime|Thriller
6        7            Sabrina (1995)            Comedy|Romance
7        8        Tom and Huck (1995) Adventure|Children's
8        9      Sudden Death (1995)            Action
9       10      GoldenEye (1995) Action|Adventure|Thriller
```

Adding indicator variables for each genre requires a little bit of wrangling. First, we extract the list of unique genres in the dataset (using a nice `set.union` trick):

```
In [197]: genre_iter = (set(x.split('|')) for x in movies.genres)

In [198]: genres = sorted(set.union(*genre_iter))
```

Now, one way to construct the indicator DataFrame is to start with a DataFrame of all zeros:

```
In [199]: dummies = DataFrame(np.zeros((len(movies), len(genres))), columns=genres)
```

Now, iterate through each movie and set entries in each row of `dummies` to 1:

```
In [200]: for i, gen in enumerate(movies.genres):
.....:     dummies.ix[i, gen.split('|')] = 1
```

Then, as above, you can combine this with `movies`:

```
In [201]: movies_windic = movies.join(dummies.add_prefix('Genre_'))

In [202]: movies_windic.ix[0]
Out[202]:
   movie_id          title           1
   genres           Animation|Children's|Comedy
   Genre_Action      0
   Genre_Adventure   0
   Genre_Animation    1
   Genre_Children's   1
   Genre_Comedy       1
   Genre_Crime        0
   Genre_Documentary  0
   Genre_Drama         0
   Genre_Fantasy       0
```

```
Genre_Film-Noir          0
Genre_Horror              0
Genre_Musical             0
Genre_Mystery             0
Genre_Romance             0
Genre_Sci-Fi              0
Genre_Thriller            0
Genre_War                 0
Genre_Western              0
Name: 0
```



For much larger data, this method of constructing indicator variables with multiple membership is not especially speedy. A lower-level function leveraging the internals of the DataFrame could certainly be written.

A useful recipe for statistical applications is to combine `get_dummies` with a discretization function like `cut`:

```
In [204]: values = np.random.rand(10)

In [205]: values
Out[205]:
array([ 0.9296,  0.3164,  0.1839,  0.2046,  0.5677,  0.5955,  0.9645,
       0.6532,  0.7489,  0.6536])

In [206]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [207]: pd.get_dummies(pd.cut(values, bins))
Out[207]:
  (0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1]
0          0          0          0          0          1
1          0          1          0          0          0
2          1          0          0          0          0
3          0          1          0          0          0
4          0          0          1          0          0
5          0          0          1          0          0
6          0          0          0          0          1
7          0          0          0          1          0
8          0          0          0          1          0
9          0          0          0          1          0
```

## String Manipulation

Python has long been a popular data munging language in part due to its ease-of-use for string and text processing. Most text operations are made simple with the string object's built-in methods. For more complex pattern matching and text manipulations, regular expressions may be needed. pandas adds to the mix by enabling you to apply string and regular expressions concisely on whole arrays of data, additionally handling the annoyance of missing data.

## String Object Methods

In many string munging and scripting applications, built-in string methods are sufficient. As an example, a comma-separated string can be broken into pieces with `split`:

```
In [208]: val = 'a,b, guido'
```

```
In [209]: val.split(',')
Out[209]: ['a', 'b', ' guido']
```

`split` is often combined with `strip` to trim whitespace (including newlines):

```
In [210]: pieces = [x.strip() for x in val.split(',')]
```

```
In [211]: pieces
Out[211]: ['a', 'b', 'guido']
```

These substrings could be concatenated together with a two-colon delimiter using addition:

```
In [212]: first, second, third = pieces
```

```
In [213]: first + '::' + second + '::' + third
Out[213]: 'a::b::guido'
```

But, this isn't a practical generic method. A faster and more Pythonic way is to pass a list or tuple to the `join` method on the string `:::`:

```
In [214]: '::'.join(pieces)
Out[214]: 'a::b::guido'
```

Other methods are concerned with locating substrings. Using Python's `in` keyword is the best way to detect a substring, though `index` and `find` can also be used:

```
In [215]: 'guido' in val
Out[215]: True
```

```
In [216]: val.index(',')
Out[216]: 1
```

```
In [217]: val.find(':')
Out[217]: -1
```

Note the difference between `find` and `index` is that `index` raises an exception if the string isn't found (versus returning `-1`):

```
In [218]: val.index(':')
-----
ValueError                                Traceback (most recent call last)
<ipython-input-218-280f8b2856ce> in <module>()
      1 val.index(':')
ValueError: substring not found
```

Relatedly, `count` returns the number of occurrences of a particular substring:

```
In [219]: val.count(',')
Out[219]: 2
```

`replace` will substitute occurrences of one pattern for another. This is commonly used to delete patterns, too, by passing an empty string:

```
In [220]: val.replace(',', '::')
Out[220]: 'a::b:: guido'
```

```
In [221]: val.replace(',', '')
Out[221]: 'ab guido'
```

Regular expressions can also be used with many of these operations as you'll see below.

Table 7-3. Python built-in string methods

Argument	Description
count	Return the number of non-overlapping occurrences of substring in the string.
endswith, startswith	Returns True if string ends with suffix (starts with prefix).
join	Use string as delimiter for concatenating a sequence of other strings.
index	Return position of first character in substring if found in the string. Raises ValueError if not found.
find	Return position of first character of <i>first</i> occurrence of substring in the string. Like index, but returns -1 if not found.
rfind	Return position of first character of <i>last</i> occurrence of substring in the string. Returns -1 if not found.
replace	Replace occurrences of string with another string.
strip, rstrip, lstrip	Trim whitespace, including newlines; equivalent to x.strip() (and rstrip, lstrip, respectively) for each element.
split	Break string into list of substrings using passed delimiter.
lower, upper	Convert alphabet characters to lowercase or uppercase, respectively.
ljust, rjust	Left justify or right justify, respectively. Pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width.

## Regular expressions

Regular expressions provide a flexible way to search or match string patterns in text. A single expression, commonly called a *regex*, is a string formed according to the regular expression language. Python's built-in `re` module is responsible for applying regular expressions to strings; I'll give a number of examples of its use here.



The art of writing regular expressions could be a chapter of its own and thus is outside the book's scope. There are many excellent tutorials and references on the internet, such as Zed Shaw's *Learn Regex The Hard Way* (<http://regex.learncodethehardway.org/book/>).

The `re` module functions fall into three categories: pattern matching, substitution, and splitting. Naturally these are all related; a regex describes a pattern to locate in the text, which can then be used for many purposes. Let's look at a simple example: suppose I wanted to split a string with a variable number of whitespace characters (tabs, spaces, and newlines). The regex describing one or more whitespace characters is `\s+`:

```
In [222]: import re  
  
In [223]: text = "foo    bar\t baz  \tqux"  
  
In [224]: re.split('\s+', text)  
Out[224]: ['foo', 'bar', 'baz', 'qux']
```

When you call `re.split('`s+', text), the regular expression is first *compiled*, then its `split` method is called on the passed text. You can compile the regex yourself with `re.compile`, forming a reusable regex object:

```
In [225]: regex = re.compile('\s+')  
  
In [226]: regex.split(text)  
Out[226]: ['foo', 'bar', 'baz', 'qux']
```

If, instead, you wanted to get a list of all patterns matching the regex, you can use the `findall` method:

```
In [227]: regex.findall(text)  
Out[227]: [' ', '\t ', '\t']
```



To avoid unwanted escaping with \ in a regular expression, use *raw* string literals like `r'C:\x'` instead of the equivalent '`C:\\x`'.

Creating a regex object with `re.compile` is highly recommended if you intend to apply the same expression to many strings; doing so will save CPU cycles.

`match` and `search` are closely related to `findall`. While `findall` returns all matches in a string, `search` returns only the first match. More rigidly, `match` *only* matches at the beginning of the string. As a less trivial example, let's consider a block of text and a regular expression capable of identifying most email addresses:

```
text = """Dave dave@google.com  
Steve steve@gmail.com  
Rob rob@gmail.com  
Ryan ryan@yahoo.com  
"""\n\npattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'\n\n# re.IGNORECASE makes the regex case-insensitive\nregex = re.compile(pattern, flags=re.IGNORECASE)
```

Using `findall` on the text produces a list of the e-mail addresses:

```
In [229]: regex.findall(text)  
Out[229]: ['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
```

`search` returns a special match object for the first email address in the text. For the above regex, the match object can only tell us the start and end position of the pattern in the string:

```
In [230]: m = regex.search(text)

In [231]: m
Out[231]: <_sre.SRE_Match at 0x10a05de00>

In [232]: text[m.start():m.end()]
Out[232]: 'dave@google.com'
```

`regex.match` returns `None`, as it only will match if the pattern occurs at the start of the string:

```
In [233]: print regex.match(text)
None
```

Relatedly, `sub` will return a new string with occurrences of the pattern replaced by the a new string:

```
In [234]: print regex.sub('REDACTED', text)
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

Suppose you wanted to find email addresses and simultaneously segment each address into its 3 components: username, domain name, and domain suffix. To do this, put parentheses around the parts of the pattern to segment:

```
In [235]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [236]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

A match object produced by this modified regex returns a tuple of the pattern components with its `groups` method:

```
In [237]: m = regex.match('wesm@bright.net')

In [238]: m.groups()
Out[238]: ('wesm', 'bright', 'net')
```

`findall` returns a list of tuples when the pattern has groups:

```
In [239]: regex.findall(text)
Out[239]:
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

`sub` also has access to groups in each match using special symbols like `\1`, `\2`, etc.:

```
In [240]: print regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text)
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

There is much more to regular expressions in Python, most of which is outside the book's scope. To give you a flavor, one variation on the above email regex gives names to the match groups:

```
regex = re.compile(r"""
    (?P<username>[A-Z0-9._%+-]+)
    @@
    (?P<domain>[A-Z0-9.-]+)
    \.
    (?P<suffix>[A-Z]{2,4})""", flags=re.IGNORECASE|re.VERBOSE)
```

The match object produced by such a regex can produce a handy dict with the specified group names:

```
In [242]: m = regex.match('wesm@bright.net')

In [243]: m.groupdict()
Out[243]: {'domain': 'bright', 'suffix': 'net', 'username': 'wesm'}
```

Table 7-4. Regular expression methods

Argument	Description
findall, finditer	Return all non-overlapping matching patterns in a string. <code>.findall</code> returns a list of all patterns while <code>finditer</code> returns them one by one from an iterator.
match	Match pattern at start of string and optionally segment pattern components into groups. If the pattern matches, returns a match object, otherwise None.
search	Scan string for match to pattern; returning a match object if so. Unlike <code>match</code> , the match can be anywhere in the string as opposed to only at the beginning.
split	Break string into pieces at each occurrence of pattern.
sub, subn	Replace all (sub) or first n occurrences (subn) of pattern in string with replacement expression. Use symbols \1, \2, ... to refer to match group elements in the replacement string.

## Vectorized string functions in pandas

Cleaning up a messy data set for analysis often requires a lot of string munging and regularization. To complicate matters, a column containing strings will sometimes have missing data:

```
In [244]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
.....:             'Rob': 'rob@gmail.com', 'Wes': np.nan}
```

```
In [245]: data = Series(data)
```

```
In [246]: data
Out[246]:
Dave      dave@google.com
Rob       rob@gmail.com
Steve     steve@gmail.com
Wes        NaN
```

```
In [247]: data.isnull()
Out[247]:
Dave      False
Rob      False
Steve     False
Wes       True
```

String and regular expression methods can be applied (passing a `lambda` or other function) to each value using `data.map`, but it will fail on the NA. To cope with this, Series has concise methods for string operations that skip NA values. These are accessed through Series's `str` attribute; for example, we could check whether each email address has 'gmail' in it with `str.contains`:

```
In [248]: data.str.contains('gmail')
Out[248]:
Dave      False
Rob       True
Steve     True
Wes      NaN
```

Regular expressions can be used, too, along with any `re` options like `IGNORECASE`:

```
In [249]: pattern
Out[249]: '([A-Zo-9._%+-]+)@([A-Zo-9.-]+)\\.([A-Z]{2,4})'

In [250]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[250]:
Dave      [('dave', 'google', 'com')]
Rob      [('rob', 'gmail', 'com')]
Steve    [('steve', 'gmail', 'com')]
Wes      NaN
```

There are a couple of ways to do vectorized element retrieval. Either use `str.get` or index into the `str` attribute:

```
In [251]: matches = data.str.match(pattern, flags=re.IGNORECASE)
```

```
In [252]: matches
Out[252]:
Dave      ('dave', 'google', 'com')
Rob      ('rob', 'gmail', 'com')
Steve    ('steve', 'gmail', 'com')
Wes      NaN
```

```
In [253]: matches.str.get(1)      In [254]: matches.str[0]
Out[253]:                               Out[254]:
Dave      google                  Dave      dave
Rob      gmail                   Rob      rob
Steve    gmail                   Steve    steve
Wes      NaN                     Wes      NaN
```

You can similarly slice strings using this syntax:

```
In [255]: data.str[:5]
Out[255]:
Dave      dave@
Rob      rob@g
Steve    steve
Wes      NaN
```

Table 7-5. Vectorized string methods

Method	Description
cat	Concatenate strings element-wise with optional delimiter
contains	Return boolean array if each string contains pattern/regex
count	Count occurrences of pattern
endswith, startswith	Equivalent to <code>x.endswith(pattern)</code> or <code>x.startswith(pattern)</code> for each element.
findall	Compute list of all occurrences of pattern/regex for each string
get	Index into each element (retrieve i-th element)
join	Join strings in each element of the Series with passed separator
len	Compute length of each string
lower, upper	Convert cases; equivalent to <code>x.lower()</code> or <code>x.upper()</code> for each element.
match	Use <code>re.match</code> with the passed regular expression on each element, returning matched groups as list.
pad	Add whitespace to left, right, or both sides of strings
center	Equivalent to <code>pad(side='both')</code>
repeat	Duplicate values; for example <code>s.str.repeat(3)</code> equivalent to <code>x * 3</code> for each string.
replace	Replace occurrences of pattern/regex with some other string
slice	Slice each string in the Series.
split	Split strings on delimiter or regular expression
strip, rstrip, lstrip	Trim whitespace, including newlines; equivalent to <code>x.strip()</code> (and <code>rstrip</code> , <code>lstrip</code> , respectively) for each element.

## Example: USDA Food Database

The US Department of Agriculture makes available a database of food nutrient information. Ashley Williams, an English hacker, has made available a version of this database in JSON format (<http://ashleyw.co.uk/project/food-nutrient-database>). The records look like this:

```
{
  "id": 21441,
  "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY,
  Wing, meat and skin with breading",
  "tags": ["KFC"],
  "manufacturer": "Kentucky Fried Chicken",
  "group": "Fast Foods",
  "portions": [
    {
      "amount": 1,
      "unit": "wing, with skin",
      "grams": 68.0
    },
  ]}
```

```

        ...
    ],
    "nutrients": [
        {
            "value": 20.8,
            "units": "g",
            "description": "Protein",
            "group": "Composition"
        },
        ...
    ]
}

```

Each food has a number of identifying attributes along with two lists of nutrients and portion sizes. Having the data in this form is not particularly amenable for analysis, so we need to do some work to wrangle the data into a better form.

After downloading and extracting the data from the link above, you can load it into Python with any JSON library of your choosing. I'll use the built-in Python `json` module:

```

In [256]: import json

In [257]: db = json.load(open('ch07/foods-2011-10-03.json'))

In [258]: len(db)
Out[258]: 6636

```

Each entry in `db` is a dict containing all the data for a single food. The '`nutrients`' field is a list of dicts, one for each nutrient:

<pre> In [259]: db[0].keys() Out[259]: [u'portions',  u'description',  u'tags',  u'nutrients',  u'group',  u'id',  u'manufacturer'] </pre>	<pre> In [260]: db[0]['nutrients'][0] Out[260]: {u'description': u'Protein',  u'group': u'Composition',  u'units': u'g',  u'value': 25.18} </pre>
<pre> In [261]: nutrients = DataFrame(db[0]['nutrients'])  In [262]: nutrients[:7] Out[262]:       description      group units   value 0          Protein  Composition     g  25.18 1  Total lipid (fat)  Composition     g  29.20 2  Carbohydrate, by difference  Composition     g   3.06 3             Ash       Other     g   3.28 4           Energy      Energy kcal 376.00 5           Water  Composition     g   39.28 6           Energy      Energy  kJ 1573.00 </pre>	

When converting a list of dicts to a DataFrame, we can specify a list of fields to extract. We'll take the food names, group, id, and manufacturer:

```
In [263]: info_keys = ['description', 'group', 'id', 'manufacturer']
```

```
In [264]: info = DataFrame(db, columns=info_keys)
```

```
In [265]: info[:5]
```

```
Out[265]:
```

	description	group	id	manufacturer
0	Cheese, caraway	Dairy and Egg Products	1008	
1	Cheese, cheddar	Dairy and Egg Products	1009	
2	Cheese, edam	Dairy and Egg Products	1018	
3	Cheese, feta	Dairy and Egg Products	1019	
4	Cheese, mozzarella, part skim milk	Dairy and Egg Products	1028	

```
In [266]: info
```

```
Out[266]:
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6636 entries, 0 to 6635
Data columns:
description    6636 non-null values
group          6636 non-null values
id             6636 non-null values
manufacturer   5195 non-null values
dtypes: int64(1), object(3)
```

You can see the distribution of food groups with `value_counts`:

```
In [267]: pd.value_counts(info.group)[:10]
```

```
Out[267]:
```

Vegetables and Vegetable Products	812
Beef Products	618
Baked Products	496
Breakfast Cereals	403
Legumes and Legume Products	365
Fast Foods	365
Lamb, Veal, and Game Products	345
Sweets	341
Pork Products	328
Fruits and Fruit Juices	328

Now, to do some analysis on all of the nutrient data, it's easiest to assemble the nutrients for each food into a single large table. To do so, we need to take several steps. First, I'll convert each list of food nutrients to a DataFrame, add a column for the food `id`, and append the DataFrame to a list. Then, these can be concatenated together with `concat`:

```
nutrients = []

for rec in db:
    fnuts = DataFrame(rec['nutrients'])
    fnuts['id'] = rec['id']
    nutrients.append(fnuts)

nutrients = pd.concat(nutrients, ignore_index=True)
```

If all goes well, `nutrients` should look like this:

```
In [269]: nutrients
Out[269]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 389355 entries, 0 to 389354
Data columns:
description    389355 non-null values
group         389355 non-null values
units          389355 non-null values
value          389355 non-null values
id             389355 non-null values
dtypes: float64(1), int64(1), object(3)
```

I noticed that, for whatever reason, there are duplicates in this DataFrame, so it makes things easier to drop them:

```
In [270]: nutrients.duplicated().sum()
Out[270]: 14179
```

```
In [271]: nutrients = nutrients.drop_duplicates()
```

Since `'group'` and `'description'` is in both DataFrame objects, we can rename them to make it clear what is what:

```
In [272]: col_mapping = {'description' : 'food',
.....:           'group'       : 'fgroup'}
```

```
In [273]: info = info.rename(columns=col_mapping, copy=False)
```

```
In [274]: info
Out[274]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6636 entries, 0 to 6635
Data columns:
food        6636 non-null values
fgroup      6636 non-null values
id          6636 non-null values
manufacturer 5195 non-null values
dtypes: int64(1), object(3)
```

```
In [275]: col_mapping = {'description' : 'nutrient',
.....:           'group'       : 'nutgroup'}
```

```
In [276]: nutrients = nutrients.rename(columns=col_mapping, copy=False)
```

```
In [277]: nutrients
Out[277]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 389354
Data columns:
nutrient     375176 non-null values
nutgroup     375176 non-null values
units        375176 non-null values
value        375176 non-null values
```

```
id           375176 non-null values  
dtypes: float64(1), int64(1), object(3)
```

With all of this done, we're ready to merge `info` with `nutrients`:

```
In [278]: ndata = pd.merge(nutrients, info, on='id', how='outer')
```

```
In [279]: ndata  
Out[279]:  
<class 'pandas.core.frame.DataFrame'>  
Int64Index: 375176 entries, 0 to 375175  
Data columns:  
nutrient      375176 non-null values  
nutgroup      375176 non-null values  
units          375176 non-null values  
value          375176 non-null values  
id            375176 non-null values  
food          375176 non-null values  
fgroup        375176 non-null values  
manufacturer  293054 non-null values  
dtypes: float64(1), int64(1), object(6)
```

```
In [280]: ndata.ix[30000]  
Out[280]:  
nutrient                  Folic acid  
nutgroup                  Vitamins  
units                      mcg  
value                      0  
id                        5658  
food          Ostrich, top loin, cooked  
fgroup        Poultry Products  
manufacturer  
Name: 30000
```

The tools that you need to slice and dice, aggregate, and visualize this dataset will be explored in detail in the next two chapters, so after you get a handle on those methods you might return to this dataset. For example, we could a plot of median values by food group and nutrient type (see [Figure 7-1](#)):

```
In [281]: result = ndata.groupby(['nutrient', 'fgroup'])['value'].quantile(0.5)
```

```
In [282]: result['Zinc, Zn'].order().plot(kind='barh')
```

With a little cleverness, you can find which food is most dense in each nutrient:

```
by_nutrient = ndata.groupby(['nutgroup', 'nutrient'])  
  
get_maximum = lambda x: x.xs(x.value.idxmax())  
get_minimum = lambda x: x.xs(x.value.idxmin())  
  
max_foods = by_nutrient.apply(get_maximum)[['value', 'food']]  
  
# make the food a little smaller  
max_foods.food = max_foods.food.str[:50]
```

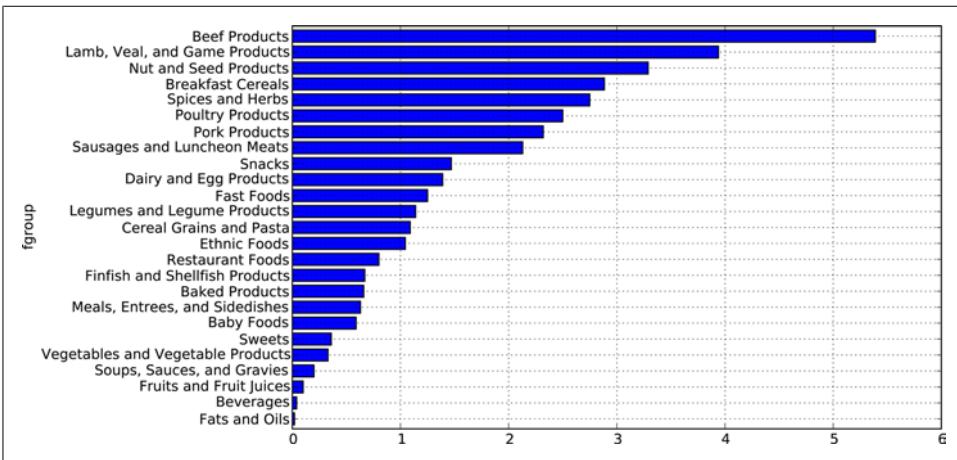


Figure 7-1. Median Zinc values by nutrient group

The resulting DataFrame is a bit too large to display in the book; here is just the 'Amino Acids' nutrient group:

```
In [284]: max_foods.ix['Amino Acids']['food']
Out[284]:
nutrient
Alanine           Gelatins, dry powder, unsweetened
Arginine          Seeds, sesame flour, low-fat
Aspartic acid     Soy protein isolate
Cystine           Seeds, cottonseed flour, low fat (glandless)
Glutamic acid    Soy protein isolate
Glycine           Gelatins, dry powder, unsweetened
Histidine         Whale, beluga, meat, dried (Alaska Native)
Hydroxyproline   KENTUCKY FRIED CHICKEN, Fried Chicken, ORIGINAL R
Isoleucine        Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Leucine           Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Lysine            Seal, bearded (Oogruk), meat, dried (Alaska Nativ
Methionine       Fish, cod, Atlantic, dried and salted
Phenylalanine    Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Proline           Gelatins, dry powder, unsweetened
Serine            Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Threonine         Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Tryptophan        Sea lion, Steller, meat with fat (Alaska Native)
Tyrosine          Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Valine            Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Name: food
```



# Plotting and Visualization

Making plots and static or interactive visualizations is one of the most important tasks in data analysis. It may be a part of the exploratory process; for example, helping identify outliers, needed data transformations, or coming up with ideas for models. For others, building an interactive visualization for the web using a toolkit like d3.js (<http://d3js.org/>) may be the end goal. Python has many visualization tools (see the end of this chapter), but I'll be mainly focused on matplotlib (<http://matplotlib.sourceforge.net>).

matplotlib is a (primarily 2D) desktop plotting package designed for creating publication-quality plots. The project was started by John Hunter in 2002 to enable a MATLAB-like plotting interface in Python. He, Fernando Pérez (of IPython), and others have collaborated for many years since then to make IPython combined with matplotlib a very functional and productive environment for scientific computing. When used in tandem with a GUI toolkit (for example, within IPython), matplotlib has interactive features like zooming and panning. It supports many different GUI backends on all operating systems and additionally can export graphics to all of the common vector and raster graphics formats: PDF, SVG, JPG, PNG, BMP, GIF, etc. I have used it to produce almost all of the graphics outside of diagrams in this book.

matplotlib has a number of add-on toolkits, such as `mplot3d` for 3D plots and `basemap` for mapping and projections. I will give an example using `basemap` to plot data on a map and to read `shapefiles` at the end of the chapter.

To follow along with the code examples in the chapter, make sure you have started IPython in Pylab mode (`ipython --pylab`) or enabled GUI event loop integration with the `%gui` magic.

## A Brief matplotlib API Primer

There are several ways to interact with matplotlib. The most common is through *pylab mode* in IPython by running `ipython --pylab`. This launches IPython configured to be able to support the matplotlib GUI backend of your choice (Tk, wxPython, PyQt, Mac

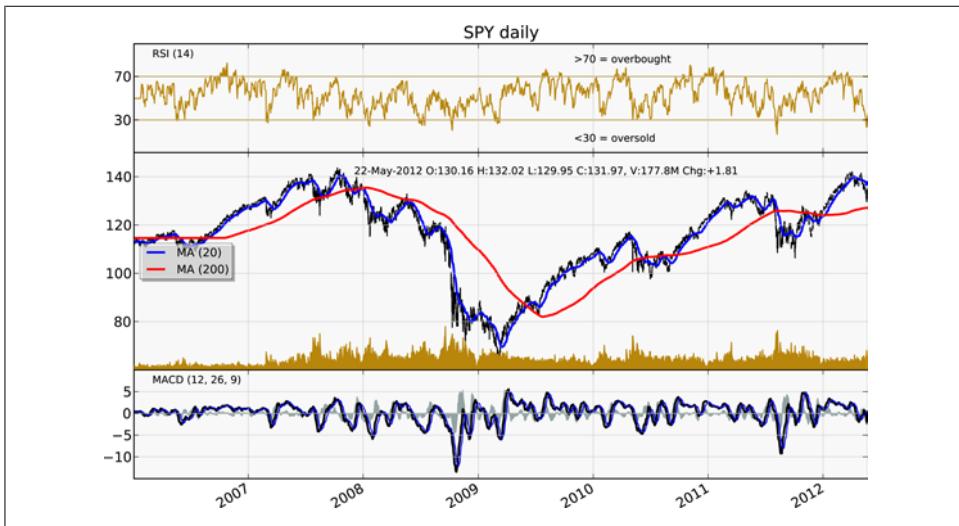


Figure 8-1. A more complex matplotlib financial plot

OS X native, GTK). For most users, the default backend will be sufficient. Pylab mode also imports a large set of modules and functions into IPython to provide a more MATLAB-like interface. You can test that everything is working by making a simple plot:

```
plot(np.arange(10))
```

If everything is set up right, a new window should pop up with a line plot. You can close it by using the mouse or entering `close()`. Matplotlib API functions like `plot` and `close` are all in the `matplotlib.pyplot` module, which is typically imported by convention as:

```
import matplotlib.pyplot as plt
```

While the pandas plotting functions described later deal with many of the mundane details of making plots, should you wish to customize them beyond the function options provided you will need to learn a bit about the matplotlib API.



There is not enough room in the book to give a comprehensive treatment to the breadth and depth of functionality in matplotlib. It should be enough to teach you the ropes to get up and running. The matplotlib gallery and documentation are the best resource for becoming a plotting guru and using advanced features.

## Figures and Subplots

Plots in matplotlib reside within a `Figure` object. You can create a new figure with `plt.figure`:

```
In [13]: fig = plt.figure()
```

If you are in pylab mode in IPython, a new empty window should pop up. `plt.figure` has a number of options, notably `figsize` will guarantee the figure has a certain size and aspect ratio if saved to disk. Figures in matplotlib also support a numbering scheme (for example, `plt.figure(2)`) that mimics MATLAB. You can get a reference to the active figure using `plt.gcf()`.

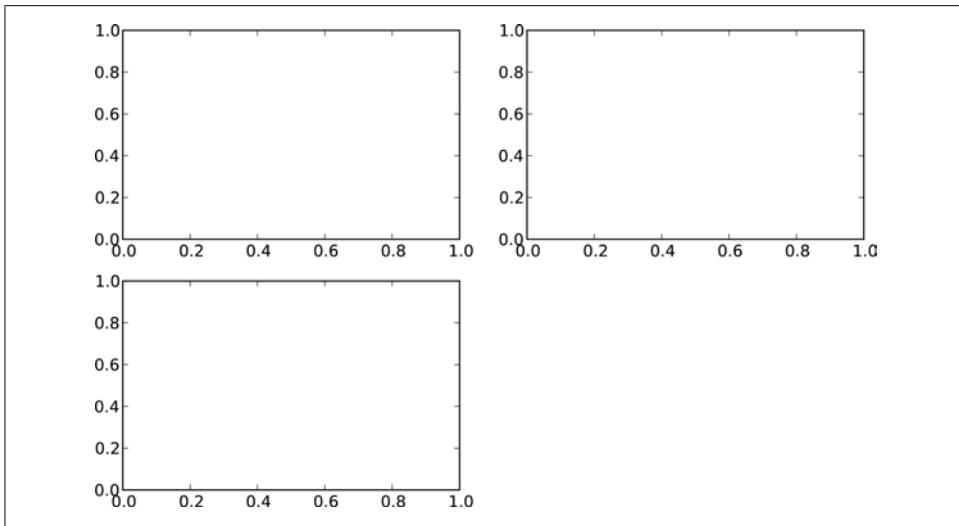
You can't make a plot with a blank figure. You have to create one or more `subplots` using `add_subplot`:

```
In [14]: ax1 = fig.add_subplot(2, 2, 1)
```

This means that the figure should be  $2 \times 2$ , and we're selecting the first of 4 subplots (numbered from 1). If you create the next two subplots, you'll end up with a figure that looks like [Figure 8-2](#).

```
In [15]: ax2 = fig.add_subplot(2, 2, 2)
```

```
In [16]: ax3 = fig.add_subplot(2, 2, 3)
```



*Figure 8-2. An empty matplotlib Figure with 3 subplots*

When you issue a plotting command like `plt.plot([1.5, 3.5, -2, 1.6])`, matplotlib draws on the last figure and subplot used (creating one if necessary), thus hiding the figure and subplot creation. Thus, if we run the following command, you'll get something like [Figure 8-3](#):

```
In [17]: from numpy.random import randn
```

```
In [18]: plt.plot(randn(50).cumsum(), 'k--')
```

The '`k--'`' is a *style* option instructing matplotlib to plot a black dashed line. The objects returned by `fig.add_subplot` above are `AxesSubplot` objects, on which you can directly plot on the other empty subplots by calling each one's instance methods, see [Figure 8-4](#):

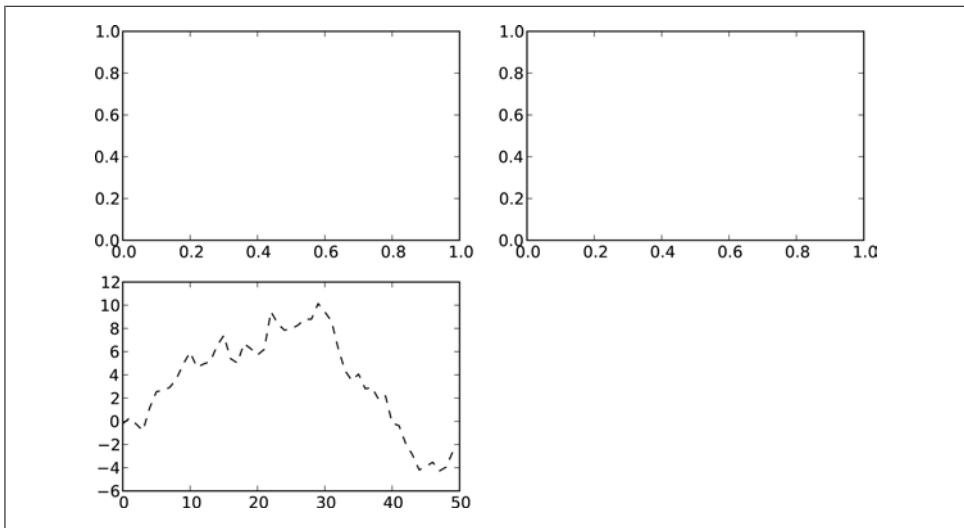


Figure 8-3. Figure after single plot

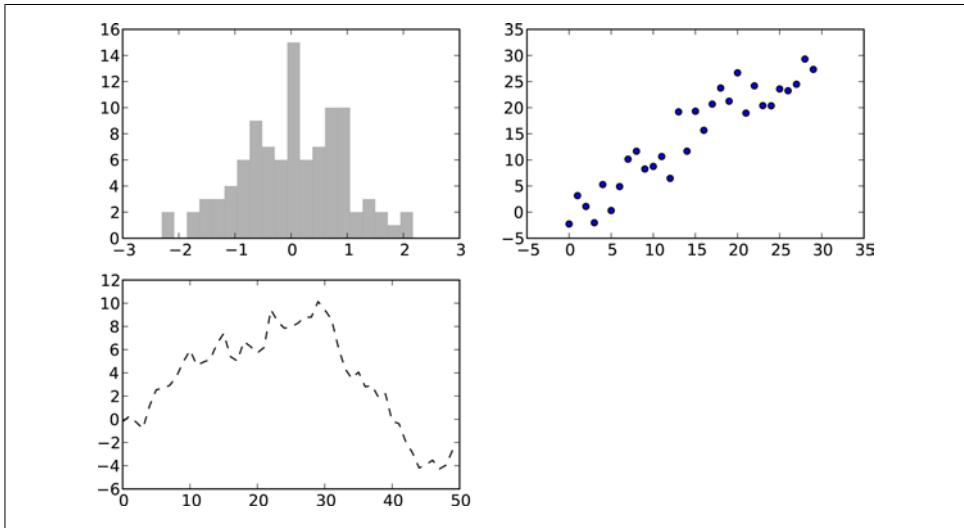


Figure 8-4. Figure after additional plots

```
In [19]: _ = ax1.hist(randn(100), bins=20, color='k', alpha=0.3)
```

```
In [20]: ax2.scatter(np.arange(30), np.arange(30) + 3 * randn(30))
```

You can find a comprehensive catalogue of plot types in the matplotlib documentation.

Since creating a figure with multiple subplots according to a particular layout is such a common task, there is a convenience method, `plt.subplots`, that creates a new figure and returns a NumPy array containing the created subplot objects:

```
In [22]: fig, axes = plt.subplots(2, 3)

In [23]: axes
Out[23]:
array([[Axes(0.125,0.536364;0.227941x0.363636),
       Axes(0.398529,0.536364;0.227941x0.363636),
       Axes(0.672059,0.536364;0.227941x0.363636)],
      [Axes(0.125,0.1;0.227941x0.363636),
       Axes(0.398529,0.1;0.227941x0.363636),
       Axes(0.672059,0.1;0.227941x0.363636)]], dtype=object)
```

This is very useful as the `axes` array can be easily indexed like a two-dimensional array; for example, `axes[0, 1]`. You can also indicate that subplots should have the same X or Y axis using `sharex` and `sharey`, respectively. This is especially useful when comparing data on the same scale; otherwise, matplotlib auto-scales plot limits independently. See [Table 8-1](#) for more on this method.

*Table 8-1. pyplot.subplots options*

Argument	Description
<code>nrows</code>	Number of rows of subplots
<code>ncols</code>	Number of columns of subplots
<code>sharex</code>	All subplots should use the same X-axis ticks (adjusting the <code>xlim</code> will affect all subplots)
<code>sharey</code>	All subplots should use the same Y-axis ticks (adjusting the <code>ylim</code> will affect all subplots)
<code>subplot_kw</code>	Dict of keywords for creating the
<code>**fig_kw</code>	Additional keywords to subplots are used when creating the figure, such as <code>plt.subplots(2, 2, figsize=(8, 6))</code>

### Adjusting the spacing around subplots

By default matplotlib leaves a certain amount of padding around the outside of the subplots and spacing between subplots. This spacing is all specified relative to the height and width of the plot, so that if you resize the plot either programmatically or manually using the GUI window, the plot will dynamically adjust itself. The spacing can be most easily changed using the `subplots_adjust` Figure method, also available as a top-level function:

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)
```

`wspace` and `hspace` controls the percent of the figure width and figure height, respectively, to use as spacing between subplots. Here is a small example where I shrink the spacing all the way to zero (see [Figure 8-5](#)):

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
    for j in range(2):
        axes[i, j].hist(randn(500), bins=50, color='k', alpha=0.5)
plt.subplots_adjust(wspace=0, hspace=0)
```

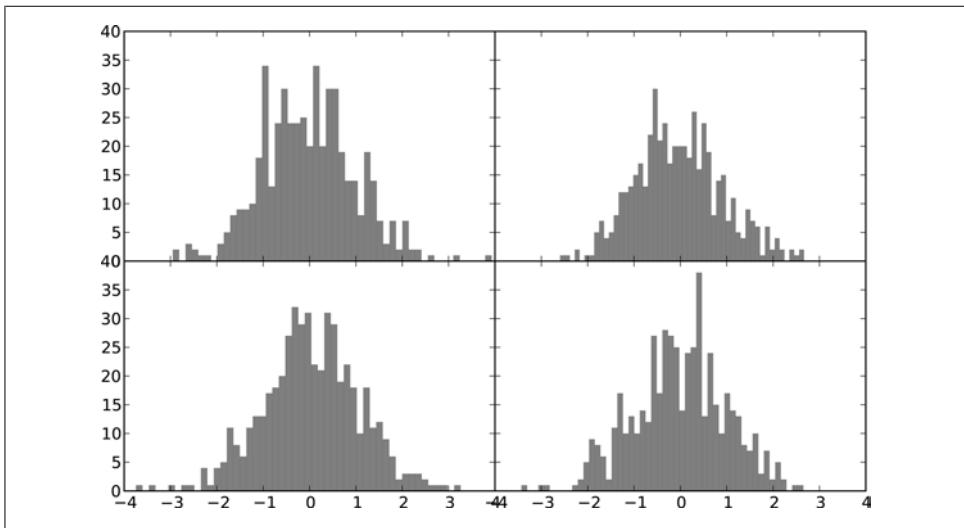


Figure 8-5. Figure with no inter-subplot spacing

You may notice that the axis labels overlap. matplotlib doesn't check whether the labels overlap, so in a case like this you would need to fix the labels yourself by specifying explicit tick locations and tick labels. More on this in the coming sections.

## Colors, Markers, and Line Styles

Matplotlib's main `plot` function accepts arrays of X and Y coordinates and optionally a string abbreviation indicating color and line style. For example, to plot `x` versus `y` with green dashes, you would execute:

```
ax.plot(x, y, 'g--')
```

This way of specifying both color and linestyle in a string is provided as a convenience; in practice if you were creating plots programmatically you might prefer not to have to munge strings together to create plots with the desired style. The same plot could also have been expressed more explicitly as:

```
ax.plot(x, y, linestyle='--', color='g')
```

There are a number of color abbreviations provided for commonly-used colors, but any color on the spectrum can be used by specifying its RGB value (for example, '`#CECECE`'). You can see the full set of linestyles by looking at the docstring for `plot`.

Line plots can additionally have *markers* to highlight the actual data points. Since matplotlib creates a continuous line plot, interpolating between points, it can occasionally be unclear where the points lie. The marker can be part of the style string, which must have color followed by marker type and line style (see [Figure 8-6](#)):

```
In [28]: plt.plot(randn(30).cumsum(), 'ko--')
```

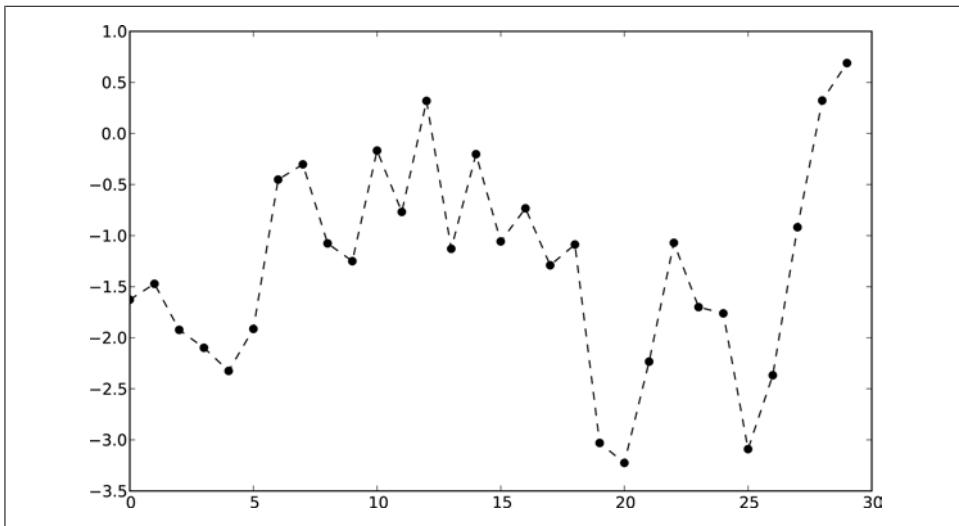


Figure 8-6. Line plot with markers example

This could also have been written more explicitly as:

```
plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')
```

For line plots, you will notice that subsequent points are linearly interpolated by default. This can be altered with the `drawstyle` option:

```
In [30]: data = randn(30).cumsum()
```

```
In [31]: plt.plot(data, 'k--', label='Default')
Out[31]: [

```

```
In [32]: plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')
Out[32]: [

```

```
In [33]: plt.legend(loc='best')
```

## Ticks, Labels, and Legends

For most kinds of plot decorations, there are two main ways to do things: using the procedural `pyplot` interface (which will be very familiar to MATLAB users) and the more object-oriented native `matplotlib` API.

The `pyplot` interface, designed for interactive use, consists of methods like `xlim`, `xticks`, and `xticklabels`. These control the plot range, tick locations, and tick labels, respectively. They can be used in two ways:

- Called with no arguments returns the current parameter value. For example `plt.xlim()` returns the current X axis plotting range

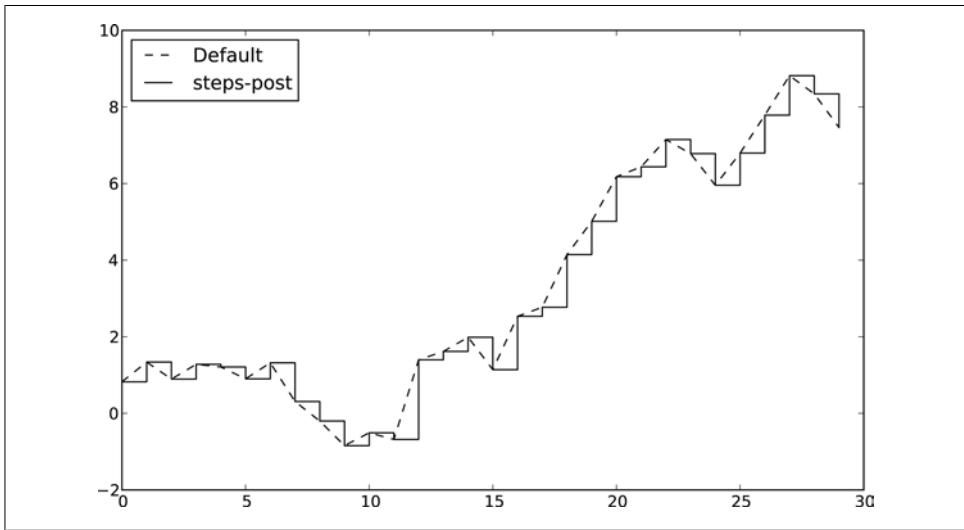


Figure 8-7. Line plot with different drawstyle options

- Called with parameters sets the parameter value. So `plt.xlim([0, 10])`, sets the X axis range to 0 to 10

All such methods act on the active or most recently-created `AxesSubplot`. Each of them corresponds to two methods on the subplot object itself; in the case of `xlim` these are `ax.get_xlim` and `ax.set_xlim`. I prefer to use the subplot instance methods myself in the interest of being explicit (and especially when working with multiple subplots), but you can certainly use whichever you find more convenient.

### Setting the title, axis labels, ticks, and ticklabels

To illustrate customizing the axes, I'll create a simple figure and plot of a random walk (see [Figure 8-8](#)):

```
In [34]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)
```

```
In [35]: ax.plot(randn(1000).cumsum())
```

To change the X axis ticks, it's easiest to use `set_xticks` and `set_xticklabels`. The former instructs matplotlib where to place the ticks along the data range; by default these locations will also be the labels. But we can set any other values as the labels using `set_xticklabels`:

```
In [36]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])
```

```
In [37]: labels = ax.set_xticklabels(['one', 'two', 'three', 'four', 'five'],
...:                                     rotation=30, fontsize='small')
```

Lastly, `set_xlabel` gives a name to the X axis and `set_title` the subplot title:

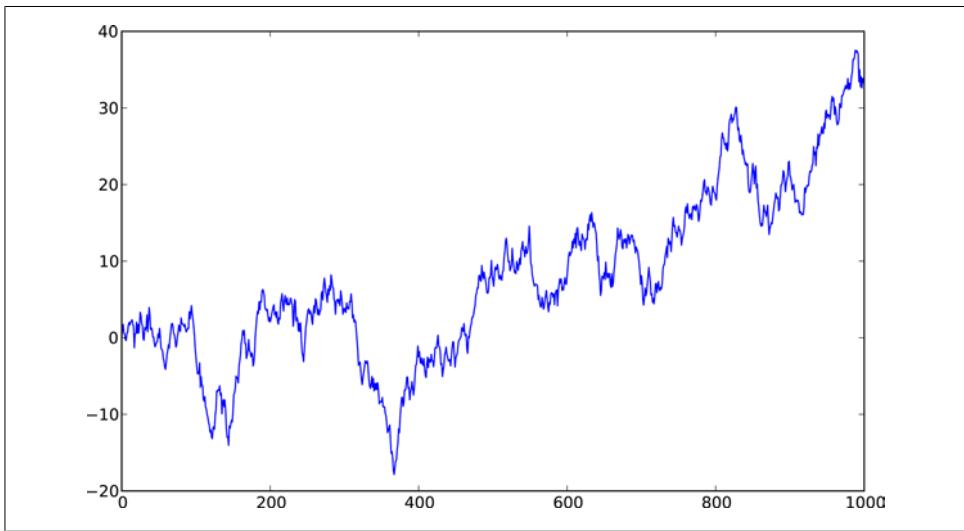


Figure 8-8. Simple plot for illustrating xticks

```
In [38]: ax.set_title('My first matplotlib plot')
Out[38]: <matplotlib.text.Text at 0x7f910912850>
```

```
In [39]: ax.set_xlabel('Stages')
```

See [Figure 8-9](#) for the resulting figure. Modifying the Y axis consists of the same process, substituting `y` for `x` in the above.

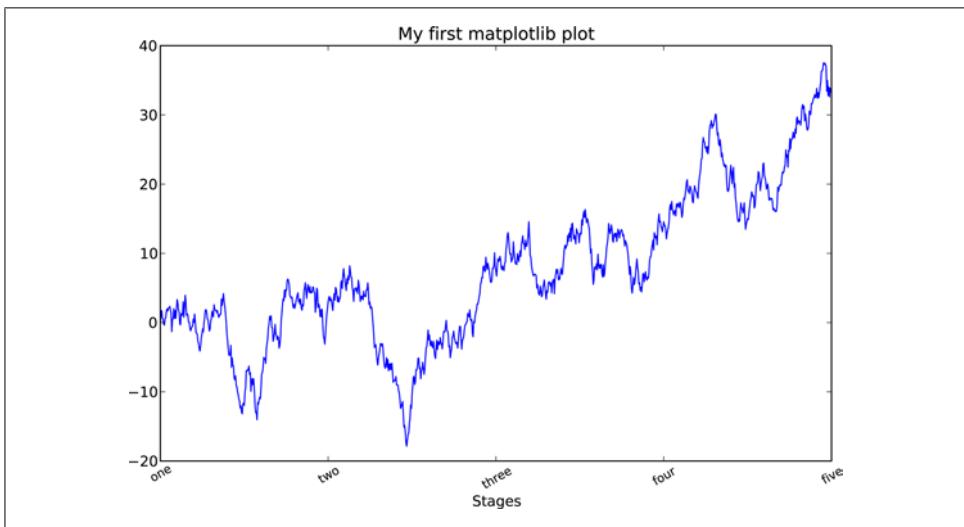


Figure 8-9. Simple plot for illustrating xticks

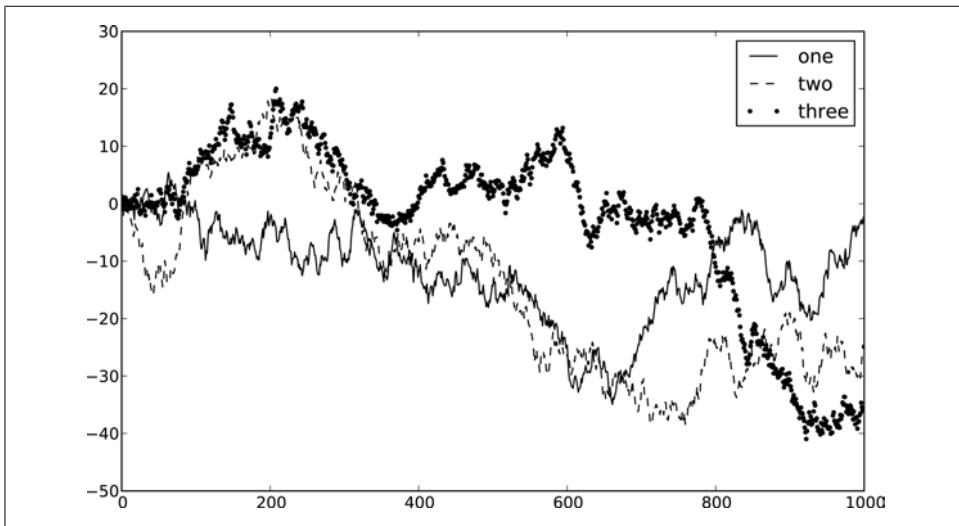


Figure 8-10. Simple plot with 3 lines and legend

### Adding legends

Legends are another critical element for identifying plot elements. There are a couple of ways to add one. The easiest is to pass the `label` argument when adding each piece of the plot:

```
In [40]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)
In [41]: ax.plot(randn(1000).cumsum(), 'k', label='one')
Out[41]: [<matplotlib.lines.Line2D at 0x4720a90>]
In [42]: ax.plot(randn(1000).cumsum(), 'k--', label='two')
Out[42]: [<matplotlib.lines.Line2D at 0x4720f90>]
In [43]: ax.plot(randn(1000).cumsum(), 'k.', label='three')
Out[43]: [<matplotlib.lines.Line2D at 0x4723550>]
```

Once you've done this, you can either call `ax.legend()` or `plt.legend()` to automatically create a legend:

```
In [44]: ax.legend(loc='best')
```

See [Figure 8-10](#). The `loc` tells matplotlib where to place the plot. If you aren't picky '`best`' is a good option, as it will choose a location that is most out of the way. To exclude one or more elements from the legend, pass no label or `label='nolegend'`.

## Annotations and Drawing on a Subplot

In addition to the standard plot types, you may wish to draw your own plot annotations, which could consist of text, arrows, or other shapes.

Annotations and text can be added using the `text`, `arrow`, and `annotate` functions. `text` draws text at given coordinates (`x`, `y`) on the plot with optional custom styling:

```
ax.text(x, y, 'Hello world!',  
        family='monospace', fontsize=10)
```

Annotations can draw both text and arrows arranged appropriately. As an example, let's plot the closing S&P 500 index price since 2007 (obtained from Yahoo! Finance) and annotate it with some of the important dates from the 2008-2009 financial crisis. See [Figure 8-11](#) for the result:

```
from datetime import datetime  
  
fig = plt.figure()  
ax = fig.add_subplot(1, 1, 1)  
  
data = pd.read_csv('ch08/spx.csv', index_col=0, parse_dates=True)  
spx = data['SPX']  
  
spx.plot(ax=ax, style='k-')  
  
crisis_data = [  
    (datetime(2007, 10, 11), 'Peak of bull market'),  
    (datetime(2008, 3, 12), 'Bear Stearns Fails'),  
    (datetime(2008, 9, 15), 'Lehman Bankruptcy')  
]  
  
for date, label in crisis_data:  
    ax.annotate(label, xy=(date, spx.asof(date) + 50),  
               xytext=(date, spx.asof(date) + 200),  
               arrowprops=dict(facecolor='black'),  
               horizontalalignment='left', verticalalignment='top')  
  
# Zoom in on 2007-2010  
ax.set_xlim(['1/1/2007', '1/1/2011'])  
ax.set_ylim([600, 1800])  
  
ax.set_title('Important dates in 2008-2009 financial crisis')
```

See the online matplotlib gallery for many more annotation examples to learn from.

Drawing shapes requires some more care. matplotlib has objects that represent many common shapes, referred to as *patches*. Some of these, like `Rectangle` and `Circle` are found in `matplotlib.pyplot`, but the full set is located in `matplotlib.patches`.

To add a shape to a plot, you create the patch object `shp` and add it to a subplot by calling `ax.add_patch(shp)` (see [Figure 8-12](#)):

```
fig = plt.figure()  
ax = fig.add_subplot(1, 1, 1)  
  
rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)  
circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)  
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],  
                  color='g', alpha=0.5)
```

```
ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)
```

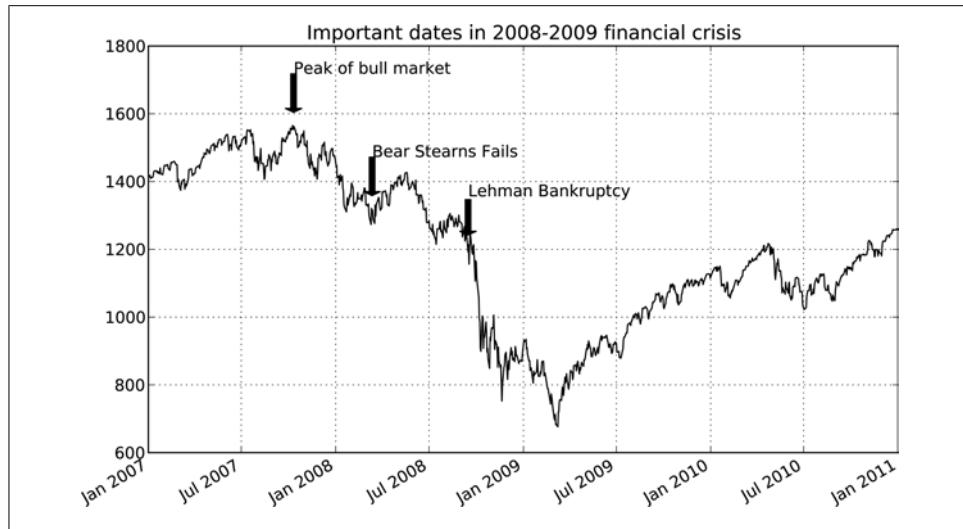


Figure 8-11. Important dates in 2008-2009 financial crisis

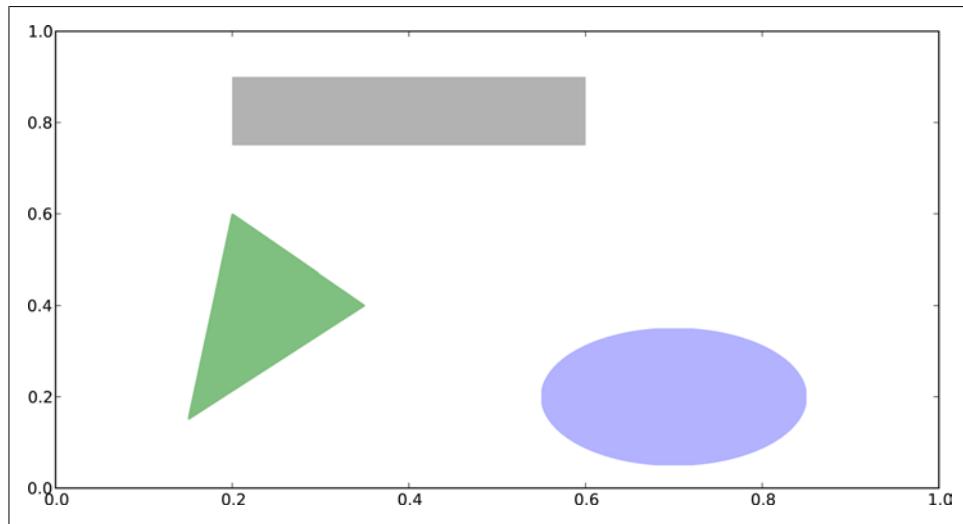


Figure 8-12. Figure composed from 3 different patches

If you look at the implementation of many familiar plot types, you will see that they are assembled from patches.

## Saving Plots to File

The active figure can be saved to file using `plt.savefig`. This method is equivalent to the figure object's `savefig` instance method. For example, to save an SVG version of a figure, you need only type:

```
plt.savefig('figpath.svg')
```

The file type is inferred from the file extension. So if you used `.pdf` instead you would get a PDF. There are a couple of important options that I use frequently for publishing graphics: `dpi`, which controls the dots-per-inch resolution, and `bbox_inches`, which can trim the whitespace around the actual figure. To get the same plot as a PNG above with minimal whitespace around the plot and at 400 DPI, you would do:

```
plt.savefig('figpath.png', dpi=400, bbox_inches='tight')
```

`savefig` doesn't have to write to disk; it can also write to any file-like object, such as a `StringIO`:

```
from io import StringIO
buffer = StringIO()
plt.savefig(buffer)
plot_data = buffer.getvalue()
```

For example, this is useful for serving dynamically-generated images over the web.

*Table 8-2. Figure.savefig options*

Argument	Description
<code>fname</code>	String containing a filepath or a Python file-like object. The figure format is inferred from the file extension, e.g. <code>.pdf</code> for PDF or <code>.png</code> for PNG.
<code>dpi</code>	The figure resolution in dots per inch; defaults to 100 out of the box but can be configured
<code>facecolor</code> , <code>edge color</code>	The color of the figure background outside of the subplots. ' <code>w</code> ' (white), by default
<code>format</code>	The explicit file format to use ( <code>'png'</code> , <code>'pdf'</code> , <code>'svg'</code> , <code>'ps'</code> , <code>'eps'</code> , ...)
<code>bbox_inches</code>	The portion of the figure to save. If ' <code>tight</code> ' is passed, will attempt to trim the empty space around the figure

## matplotlib Configuration

matplotlib comes configured with color schemes and defaults that are geared primarily toward preparing figures for publication. Fortunately, nearly all of the default behavior can be customized via an extensive set of global parameters governing figure size, subplot spacing, colors, font sizes, grid styles, and so on. There are two main ways to interact with the matplotlib configuration system. The first is programmatically from Python using the `rc` method. For example, to set the global default figure size to be 10 x 10, you could enter:

```
plt.rc('figure', figsize=(10, 10))
```

The first argument to `rc` is the component you wish to customize, such as '`figure`', '`axes`', '`xtick`', '`ytick`', '`grid`', '`legend`' or many others. After that can follow a sequence of keyword arguments indicating the new parameters. An easy way to write down the options in your program is as a dict:

```
font_options = {'family' : 'monospace',
                'weight' : 'bold',
                'size'   : 'small'}
plt.rc('font', **font_options)
```

For more extensive customization and to see a list of all the options, matplotlib comes with a configuration file `matplotlibrc` in the `matplotlib/mpl-data` directory. If you customize this file and place it in your home directory titled `.matplotlibrc`, it will be loaded each time you use matplotlib.

## Plotting Functions in pandas

As you've seen, matplotlib is actually a fairly low-level tool. You assemble a plot from its base components: the data display (the type of plot: line, bar, box, scatter, contour, etc.), legend, title, tick labels, and other annotations. Part of the reason for this is that in many cases the data needed to make a complete plot is spread across many objects. In pandas we have row labels, column labels, and possibly grouping information. This means that many kinds of fully-formed plots that would ordinarily require a lot of matplotlib code can be expressed in one or two concise statements. Therefore, pandas has an increasing number of high-level plotting methods for creating standard visualizations that take advantage of how data is organized in DataFrame objects.



As of this writing, the plotting functionality in pandas is undergoing quite a bit of work. As part of the 2012 Google Summer of Code program, a student is working full time to add features and to make the interface more consistent and usable. Thus, it's possible that this code may fall out-of-date faster than the other things in this book. The online pandas documentation will be the best resource in that event.

## Line Plots

Series and DataFrame each have a `plot` method for making many different plot types. By default, they make line plots (see [Figure 8-13](#)):

```
In [55]: s = Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))
```

```
In [56]: s.plot()
```

The Series object's index is passed to matplotlib for plotting on the X axis, though this can be disabled by passing `use_index=False`. The X axis ticks and limits can be adjusted using the `xticks` and `xlim` options, and Y axis respectively using `yticks` and `ylim`. See

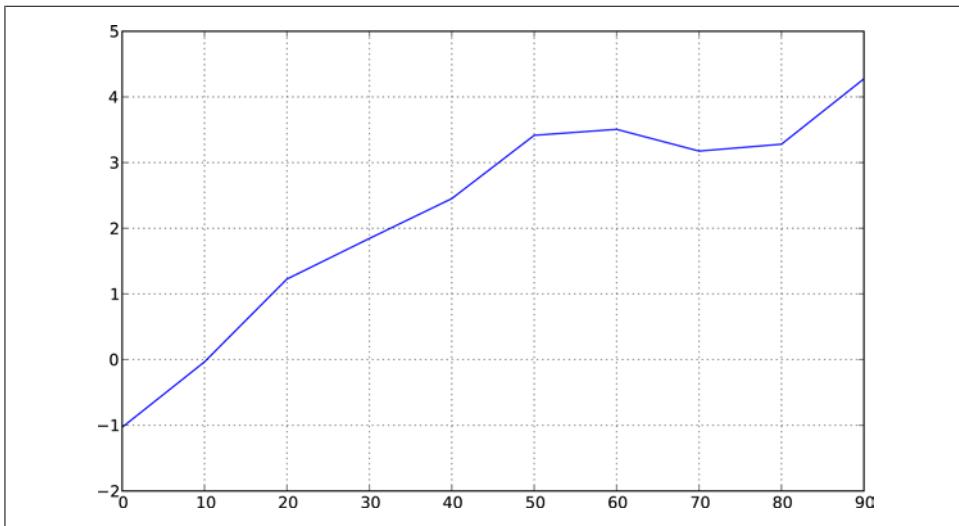


Figure 8-13. Simple Series plot example

[Table 8-3](#) for a full listing of `plot` options. I'll comment on a few more of them throughout this section and leave the rest to you to explore.

Most of pandas's plotting methods accept an optional `ax` parameter, which can be a matplotlib subplot object. This gives you more flexible placement of subplots in a grid layout. There will be more on this in the later section on the matplotlib API.

DataFrame's `plot` method plots each of its columns as a different line on the same subplot, creating a legend automatically (see [Figure 8-14](#)):

```
In [57]: df = DataFrame(np.random.randn(10, 4).cumsum(0),
....:                   columns=['A', 'B', 'C', 'D'],
....:                   index=np.arange(0, 100, 10))

In [58]: df.plot()
```



Additional keyword arguments to `plot` are passed through to the respective matplotlib plotting function, so you can further customize these plots by learning more about the matplotlib API.

[Table 8-3. Series.plot method arguments](#)

Argument	Description
<code>label</code>	Label for plot legend
<code>ax</code>	matplotlib subplot object to plot on. If nothing passed, uses active matplotlib subplot
<code>style</code>	Style string, like ' <code>ko--</code> ', to be passed to matplotlib.
<code>alpha</code>	The plot fill opacity (from 0 to 1)

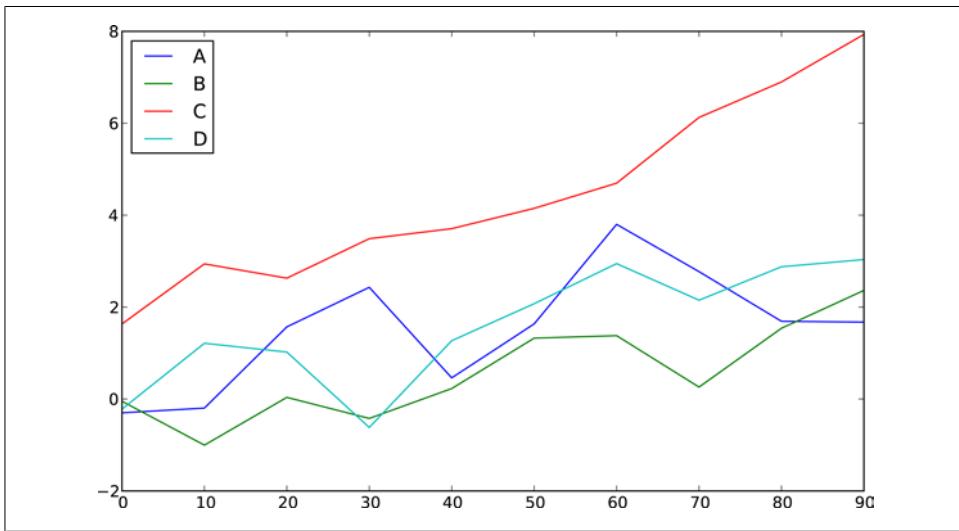


Figure 8-14. Simple DataFrame plot example

Argument	Description
kind	Can be 'line', 'bar', 'barh', 'kde'
logy	Use logarithmic scaling on the Y axis
use_index	Use the object index for tick labels
rot	Rotation of tick labels (0 through 360)
xticks	Values to use for X axis ticks
yticks	Values to use for Y axis ticks
xlim	X axis limits (e.g. [0, 10])
ylim	Y axis limits
grid	Display axis grid (on by default)

DataFrame has a number of options allowing some flexibility with how the columns are handled; for example, whether to plot them all on the same subplot or to create separate subplots. See [Table 8-4](#) for more on these.

Table 8-4. DataFrame-specific plot arguments

Argument	Description
subplots	Plot each DataFrame column in a separate subplot
sharex	If subplots=True, share the same X axis, linking ticks and limits
sharey	If subplots=True, share the same Y axis
figsize	Size of figure to create as tuple

Argument	Description
title	Plot title as string
legend	Add a subplot legend (True by default)
sort_columns	Plot columns in alphabetical order; by default uses existing column order



For time series plotting, see [Chapter 10](#).

## Bar Plots

Making bar plots instead of line plots is as simple as passing `kind='bar'` (for vertical bars) or `kind='barh'` (for horizontal bars). In this case, the Series or DataFrame index will be used as the X (`bar`) or Y (`barh`) ticks (see [Figure 8-15](#)):

```
In [59]: fig, axes = plt.subplots(2, 1)
```

```
In [60]: data = Series(np.random.rand(16), index=list('abcdefghijklmnp'))
```

```
In [61]: data.plot(kind='bar', ax=axes[0], color='k', alpha=0.7)
Out[61]: <matplotlib.axes.AxesSubplot at 0x4ee7750>
```

```
In [62]: data.plot(kind='barh', ax=axes[1], color='k', alpha=0.7)
```



For more on the `plt.subplots` function and matplotlib axes and figures, see the later section in this chapter.

With a DataFrame, bar plots group the values in each row together in a group in bars, side by side, for each value. See [Figure 8-16](#):

```
In [63]: df = DataFrame(np.random.rand(6, 4),
....:                   index=['one', 'two', 'three', 'four', 'five', 'six'],
....:                   columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))
```

```
In [64]: df
```

```
Out[64]:
```

Genus	A	B	C	D
one	0.301686	0.156333	0.371943	0.270731
two	0.750589	0.525587	0.689429	0.358974
three	0.381504	0.667707	0.473772	0.632528
four	0.942408	0.180186	0.708284	0.641783
five	0.840278	0.909589	0.010041	0.653207
six	0.062854	0.589813	0.811318	0.060217

```
In [65]: df.plot(kind='bar')
```

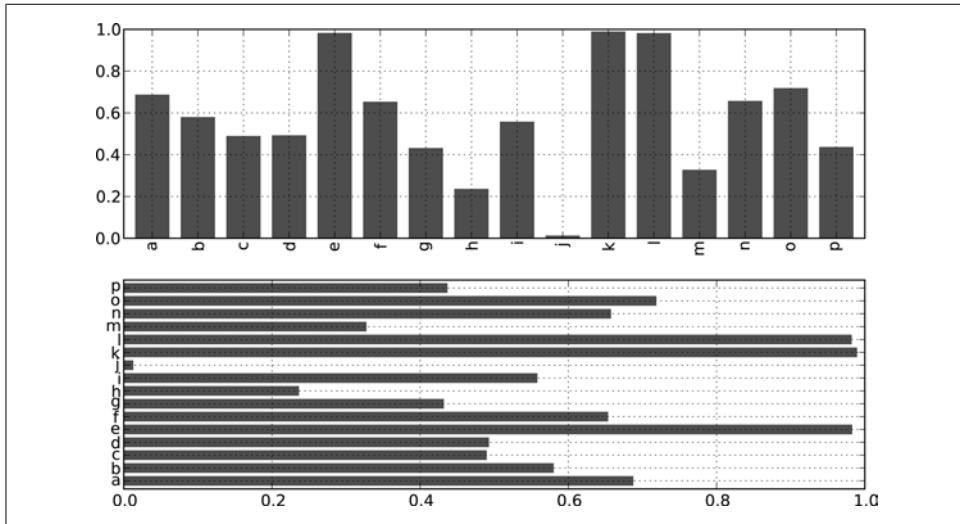


Figure 8-15. Horizontal and vertical bar plot example

Note that the name “Genus” on the DataFrame’s columns is used to title the legend. Stacked bar plots are created from a DataFrame by passing `stacked=True`, resulting in the value in each row being stacked together (see [Figure 8-17](#)):

```
In [67]: df.plot(kind='barh', stacked=True, alpha=0.5)
```



A useful recipe for bar plots (as seen in an earlier chapter) is to visualize a Series’s value frequency using `value_counts`:  
`s.value_counts().plot(kind='bar')`

Returning to the tipping data set used earlier in the book, suppose we wanted to make a stacked bar plot showing the percentage of data points for each party size on each day. I load the data using `read_csv` and make a cross-tabulation by day and party size:

```
In [68]: tips = pd.read_csv('ch08/tips.csv')
```

```
In [69]: party_counts = pd.crosstab(tips.day, tips.size)
```

```
In [70]: party_counts
```

```
Out[70]:
```

size	1	2	3	4	5	6
day						
Fri	1	16	1	1	0	0
Sat	2	53	18	13	1	0
Sun	0	39	15	18	3	1
Thur	1	48	4	5	1	3

```
# Not many 1- and 6-person parties  
In [71]: party_counts = party_counts.ix[:, 2:5]
```

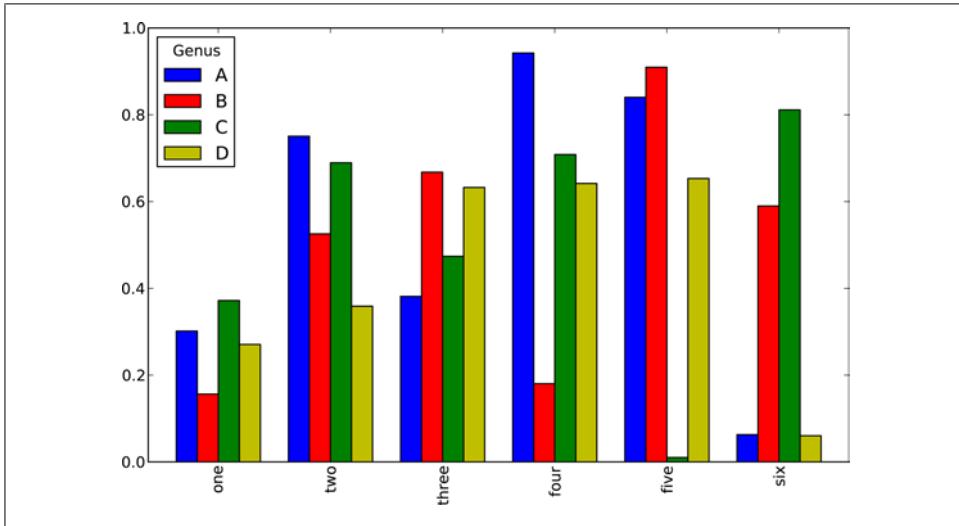


Figure 8-16. DataFrame bar plot example

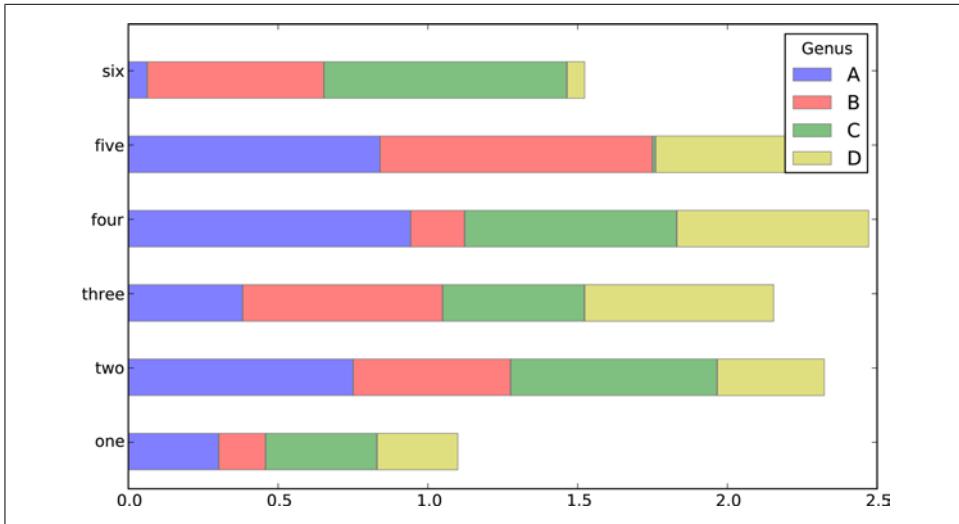


Figure 8-17. DataFrame stacked bar plot example

Then, normalize so that each row sums to 1 (I have to cast to float to avoid integer division issues on Python 2.7) and make the plot (see Figure 8-18):

```
# Normalize to sum to 1  
In [72]: party_pcts = party_counts.div(party_counts.sum(1).astype(float), axis=0)
```

```
In [73]: party_pcts  
Out[73]:  
size      2        3        4        5  
day  
Fri    0.888889  0.055556  0.055556  0.000000  
Sat    0.623529  0.211765  0.152941  0.011765  
Sun    0.520000  0.200000  0.240000  0.040000  
Thur   0.827586  0.068966  0.086207  0.017241
```

```
In [74]: party_pcts.plot(kind='bar', stacked=True)
```

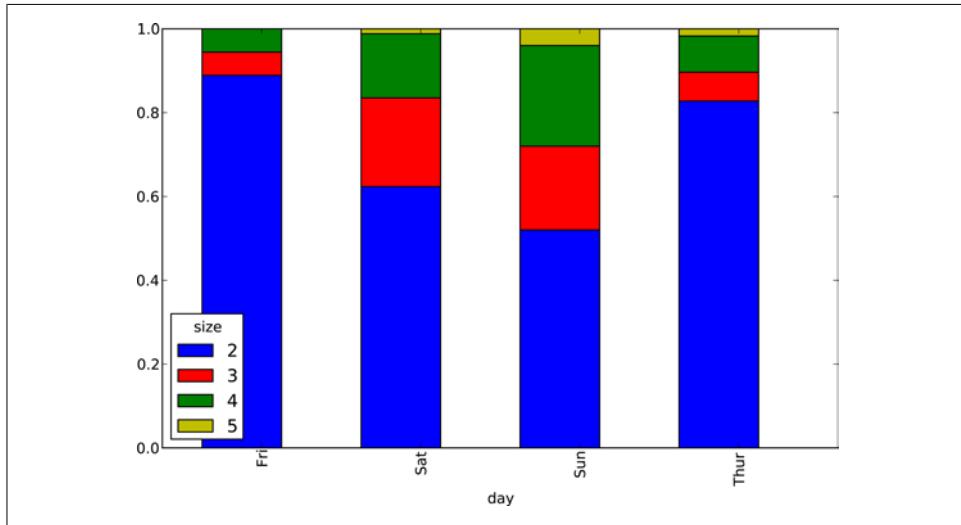


Figure 8-18. Fraction of parties by size on each day

So you can see that party sizes appear to increase on the weekend in this data set.

## Histograms and Density Plots

A histogram, with which you may be well-acquainted, is a kind of bar plot that gives a discretized display of value frequency. The data points are split into discrete, evenly spaced bins, and the number of data points in each bin is plotted. Using the tipping data from before, we can make a histogram of tip percentages of the total bill using the `hist` method on the Series (see Figure 8-19):

```
In [76]: tips['tip_pct'] = tips['tip'] / tips['total_bill']
```

```
In [77]: tips['tip_pct'].hist(bins=50)
```

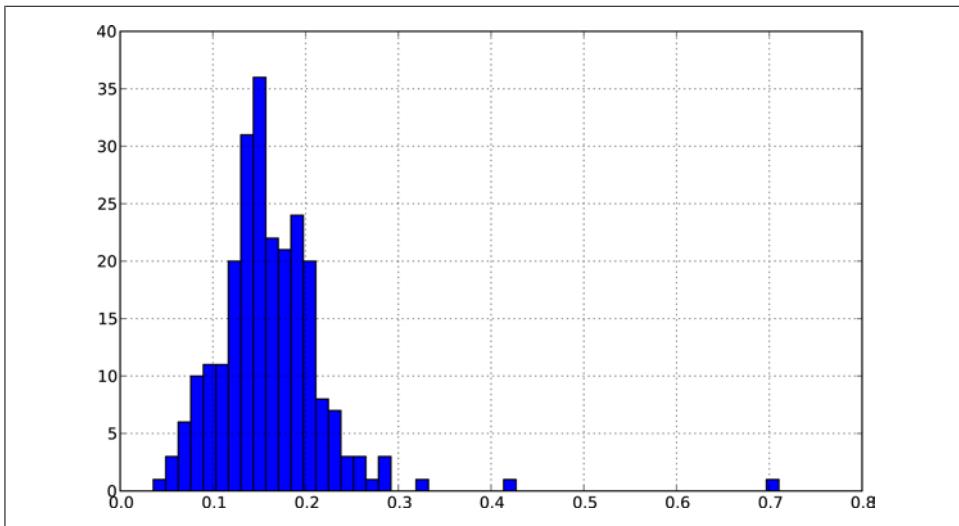


Figure 8-19. Histogram of tip percentages

A related plot type is a *density plot*, which is formed by computing an estimate of a continuous probability distribution that might have generated the observed data. A usual procedure is to approximate this distribution as a mixture of kernels, that is, simpler distributions like the normal (Gaussian) distribution. Thus, density plots are also known as KDE (kernel density estimate) plots. Using `plot` with `kind='kde'` makes a density plot using the standard mixture-of-normals KDE (see Figure 8-20):

```
In [79]: tips['tip_pct'].plot(kind='kde')
```

These two plot types are often plotted together; the histogram in normalized form (to give a binned density) with a kernel density estimate plotted on top. As an example, consider a bimodal distribution consisting of draws from two different standard normal distributions (see Figure 8-21):

```
In [81]: comp1 = np.random.normal(0, 1, size=200) # N(0, 1)
```

```
In [82]: comp2 = np.random.normal(10, 2, size=200) # N(10, 4)
```

```
In [83]: values = Series(np.concatenate([comp1, comp2]))
```

```
In [84]: values.hist(bins=100, alpha=0.3, color='k', normed=True)
Out[84]: <matplotlib.axes.AxesSubplot at 0x5cd2350>
```

```
In [85]: values.plot(kind='kde', style='k--')
```

## Scatter Plots

Scatter plots are a useful way of examining the relationship between two one-dimensional data series. matplotlib has a `scatter` plotting method that is the workhorse of

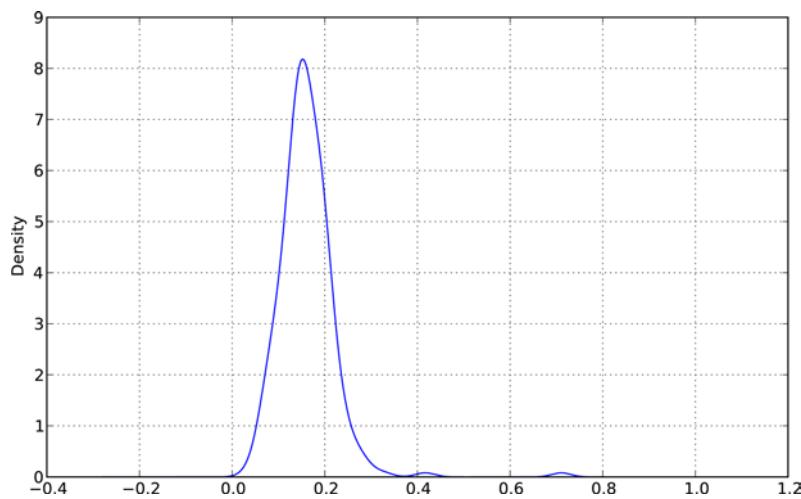


Figure 8-20. Density plot of tip percentages

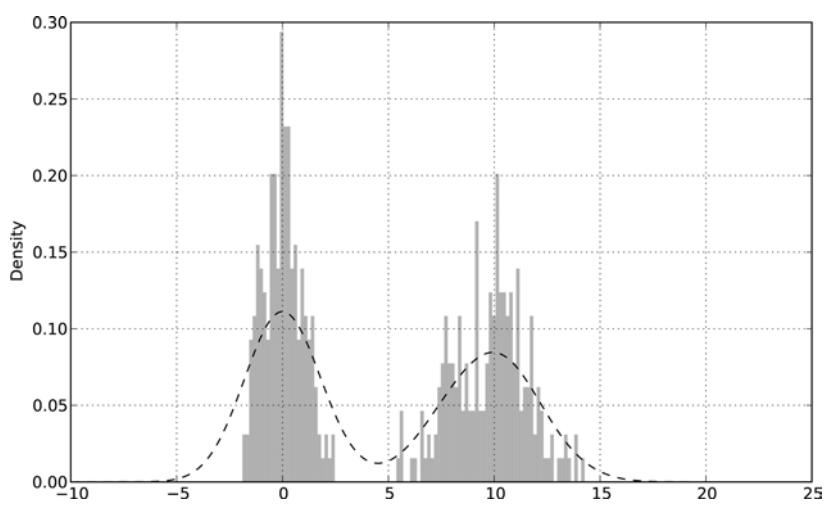


Figure 8-21. Normalized histogram of normal mixture with density estimate

making these kinds of plots. To give an example, I load the `macrodata` dataset from the `statsmodels` project, select a few variables, then compute log differences:

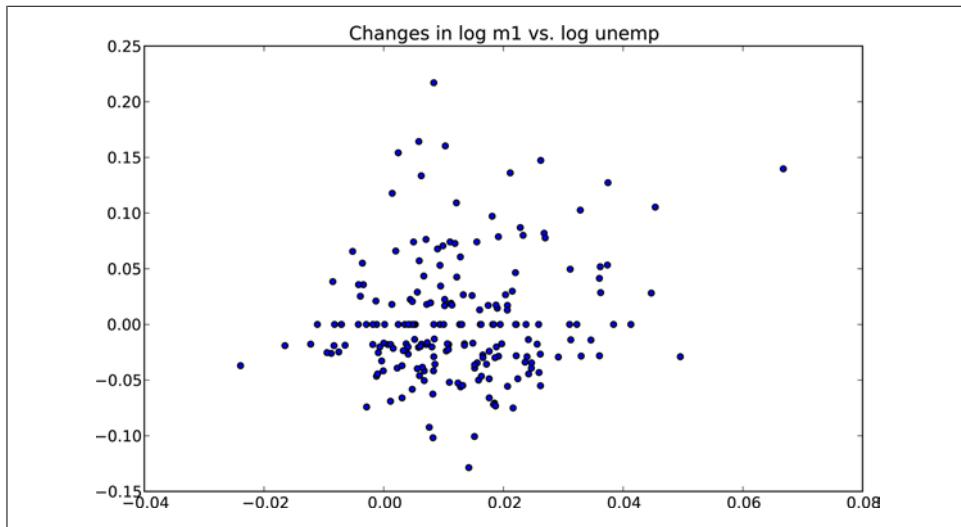
```
In [86]: macro = pd.read_csv('ch08/macrodta.csv')
In [87]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]
In [88]: trans_data = np.log(data).diff().dropna()
```

```
In [89]: trans_data[-5:]
Out[89]:
   cpi      m1  tbilrate    unemp
198 -0.007904  0.045361 -0.396881  0.105361
199 -0.021979  0.066753 -2.277267  0.139762
200  0.002340  0.010286  0.606136  0.160343
201  0.008419  0.037461 -0.200671  0.127339
202  0.008894  0.012202 -0.405465  0.042560
```

It's easy to plot a simple scatter plot using `plt.scatter` (see [Figure 8-22](#)):

```
In [91]: plt.scatter(trans_data['m1'], trans_data['unemp'])
Out[91]: <matplotlib.collections.PathCollection at 0x43c31d0>
```

```
In [92]: plt.title('Changes in log %s vs. log %s' % ('m1', 'unemp'))
```



*Figure 8-22. A simple scatter plot*

In exploratory data analysis it's helpful to be able to look at all the scatter plots among a group of variables; this is known as a *pairs* plot or *scatter plot matrix*. Making such a plot from scratch is a bit of work, so pandas has a `scatter_matrix` function for creating one from a DataFrame. It also supports placing histograms or density plots of each variable along the diagonal. See [Figure 8-23](#) for the resulting plot:

```
In [93]: scatter_matrix(trans_data, diagonal='kde', color='k', alpha=0.3)
```

## Plotting Maps: Visualizing Haiti Earthquake Crisis Data

Ushahidi is a non-profit software company that enables crowdsourcing of information related to natural disasters and geopolitical events via text message. Many of these data sets are then published on their [website](#) for analysis and visualization. I downloaded

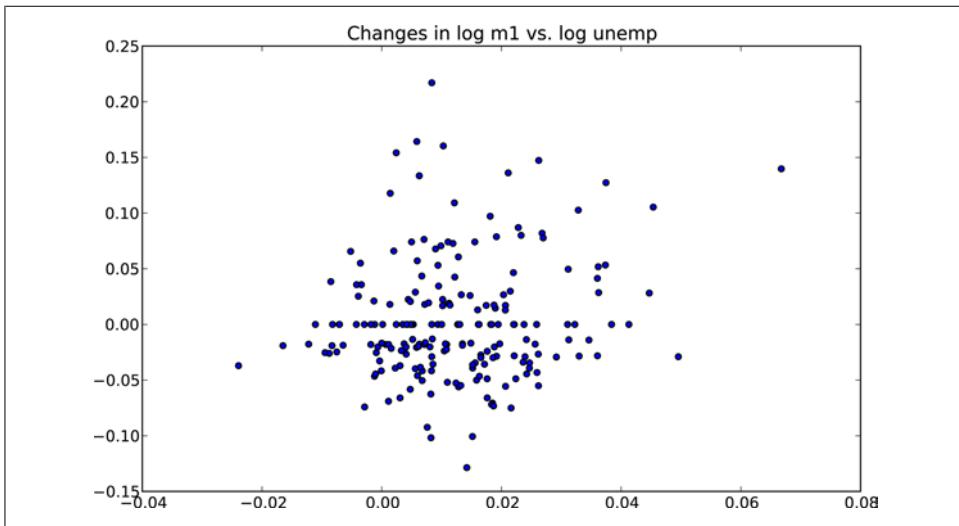


Figure 8-23. Scatter plot matrix of statsmodels macro data

the data collected during the 2010 Haiti earthquake crisis and aftermath, and I'll show you how I prepared the data for analysis and visualization using pandas and other tools we have looked at thus far. After downloading the CSV file from the above link, we can load it into a DataFrame using `read_csv`:

```
In [94]: data = pd.read_csv('ch08/Haiti.csv')
```

```
In [95]: data
Out[95]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3593 entries, 0 to 3592
Data columns:
Serial      3593 non-null values
INCIDENT TITLE 3593 non-null values
INCIDENT DATE 3593 non-null values
LOCATION     3593 non-null values
DESCRIPTION   3593 non-null values
CATEGORY     3587 non-null values
LATITUDE     3593 non-null values
LONGITUDE    3593 non-null values
APPROVED     3593 non-null values
VERIFIED     3593 non-null values
dtypes: float64(2), int64(1), object(7)
```

It's easy now to tinker with this data set to see what kinds of things we might want to do with it. Each row represents a report sent from someone's mobile phone indicating an emergency or some other problem. Each has an associated timestamp and a location as latitude and longitude:

```
In [96]: data[['INCIDENT DATE', 'LATITUDE', 'LONGITUDE']][:10]
Out[96]:
  INCIDENT DATE  LATITUDE  LONGITUDE
0       2010-01-12  18.545000 -72.335000
1       2010-01-12  18.545000 -72.335000
2       2010-01-12  18.545000 -72.335000
3       2010-01-12  18.545000 -72.335000
4       2010-01-12  18.545000 -72.335000
5       2010-01-12  18.545000 -72.335000
6       2010-01-12  18.545000 -72.335000
7       2010-01-12  18.545000 -72.335000
8       2010-01-12  18.545000 -72.335000
9       2010-01-12  18.545000 -72.335000
```

```

0 05/07/2010 17:26 18.233333 -72.533333
1 28/06/2010 23:06 50.226029 5.729886
2 24/06/2010 16:21 22.278381 114.174287
3 20/06/2010 21:59 44.407062 8.933989
4 18/05/2010 16:26 18.571084 -72.334671
5 26/04/2010 13:14 18.593707 -72.310079
6 26/04/2010 14:19 18.482800 -73.638800
7 26/04/2010 14:27 18.415000 -73.195000
8 15/03/2010 10:58 18.517443 -72.236841
9 15/03/2010 11:00 18.547790 -72.410010

```

The `CATEGORY` field contains a comma-separated list of codes indicating the type of message:

```

In [97]: data['CATEGORY'][::6]
Out[97]:
0           1. Urgences | Emergency, 3. Public Health,
1      1. Urgences | Emergency, 2. Urgences logistiques
2      2. Urgences logistiques | Vital Lines, 8. Autre |
3                           1. Urgences | Emergency,
4                           1. Urgences | Emergency,
5          5e. Communication lines down,
Name: CATEGORY

```

If you notice above in the data summary, some of the categories are missing, so we might want to drop these data points. Additionally, calling `describe` shows that there are some aberrant locations:

```

In [98]: data.describe()
Out[98]:
   Serial      LATITUDE    LONGITUDE
count 3593.000000  3593.000000  3593.000000
mean  2080.277484   18.611495 -72.322680
std   1171.100360    0.738572   3.650776
min    4.000000   18.041313 -74.452757
25%  1074.000000   18.524070 -72.417500
50%  2163.000000   18.539269 -72.335000
75%  3088.000000   18.561820 -72.293570
max   4052.000000   50.226029  114.174287

```

Cleaning the bad locations and removing the missing categories is now fairly simple:

```

In [99]: data = data[(data.LATITUDE > 18) & (data.LATITUDE < 20) &
.....:             (data.LONGITUDE > -75) & (data.LONGITUDE < -70)
.....:             & data.CATEGORY.notnull()]

```

Now we might want to do some analysis or visualization of this data by category, but each category field may have multiple categories. Additionally, each category is given as a code plus an English and possibly also a French code name. Thus, a little bit of wrangling is required to get the data into a more agreeable form. First, I wrote these two functions to get a list of all the categories and to split each category into a code and an English name:

```

def to_cat_list(catstr):
    stripped = (x.strip() for x in catstr.split(','))

```

```

        return [x for x in stripped if x]

def get_all_categories(cat_series):
    cat_sets = (set(to_cat_list(x)) for x in cat_series)
    return sorted(set.union(*cat_sets))

def get_english(cat):
    code, names = cat.split('.')
    if '|' in names:
        names = names.split(' | ')[1]
    return code, names.strip()

```

You can test out that the `get_english` function does what you expect:

```
In [101]: get_english('2. Urgences logistiques | Vital Lines')
Out[101]: ('2', 'Vital Lines')
```

Now, I make a `dict` mapping code to name because we'll use the codes for analysis. We'll use this later when adorning plots (note the use of a generator expression in lieu of a list comprehension):

```

In [102]: all_cats = get_all_categories(data.CATEGORY)

# Generator expression
In [103]: english_mapping = dict(get_english(x) for x in all_cats)

In [104]: english_mapping['2a']
Out[104]: 'Food Shortage'

In [105]: english_mapping['6c']
Out[105]: 'Earthquake and aftershocks'

```

There are many ways to go about augmenting the data set to be able to easily select records by category. One way is to add indicator (or dummy) columns, one for each category. To do that, first extract the unique category codes and construct a DataFrame of zeros having those as its columns and the same index as `data`:

```

def get_code(seq):
    return [x.split('.')[0] for x in seq if x]

all_codes = get_code(all_cats)
code_index = pd.Index(np.unique(all_codes))
dummy_frame = DataFrame(np.zeros((len(data), len(code_index))),
                        index=data.index, columns=code_index)

```

If all goes well, `dummy_frame` should look something like this:

```

In [107]: dummy_frame.ix[:, :6]
Out[107]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3569 entries, 0 to 3592
Data columns:
 1      3569 non-null values
 1a     3569 non-null values
 1b     3569 non-null values
 1c     3569 non-null values

```

```
1d    3569 non-null values
2    3569 non-null values
dtypes: float64(6)
```

As you recall, the trick is then to set the appropriate entries of each row to 1, lastly joining this with data:

```
for row, cat in zip(data.index, data.CATEGORY):
    codes = get_code(to_cat_list(cat))
    dummy_frame.ix[row, codes] = 1

data = data.join(dummy_frame.add_prefix('category_'))
```

data finally now has new columns like:

```
In [109]: data.ix[:, 10:15]
Out[109]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3569 entries, 0 to 3592
Data columns:
category_1    3569 non-null values
category_1a   3569 non-null values
category_1b   3569 non-null values
category_1c   3569 non-null values
category_1d   3569 non-null values
dtypes: float64(5)
```

Let's make some plots! As this is spatial data, we'd like to plot the data by category on a map of Haiti. The `basemap` toolkit (<http://matplotlib.github.com/basemap>), an add-on to matplotlib, enables plotting 2D data on maps in Python. `basemap` provides many different globe projections and a means for transforming projecting latitude and longitude coordinates on the globe onto a two-dimensional matplotlib plot. After some trial and error and using the above data as a guideline, I wrote this function which draws a simple black and white map of Haiti:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

def basic_haiti_map(ax=None, lllat=17.25, urlat=20.25,
                    lllon=-75, urlon=-71):
    # create polar stereographic Basemap instance.
    m = Basemap(ax=ax, projection='stere',
                lon_0=(urlon + lllon) / 2,
                lat_0=(urlat + lllat) / 2,
                llcrnrlat=lllat, urcrnrlat=urlat,
                llcrnrlon=lllon, urcrnrlon=urlon,
                resolution='f')
    # draw coastlines, state and country boundaries, edge of map.
    m.drawcoastlines()
    m.drawstates()
    m.drawcountries()
    return m
```

The idea, now, is that the returned `Basemap` object, knows how to transform coordinates onto the canvas. I wrote the following code to plot the data observations for a number

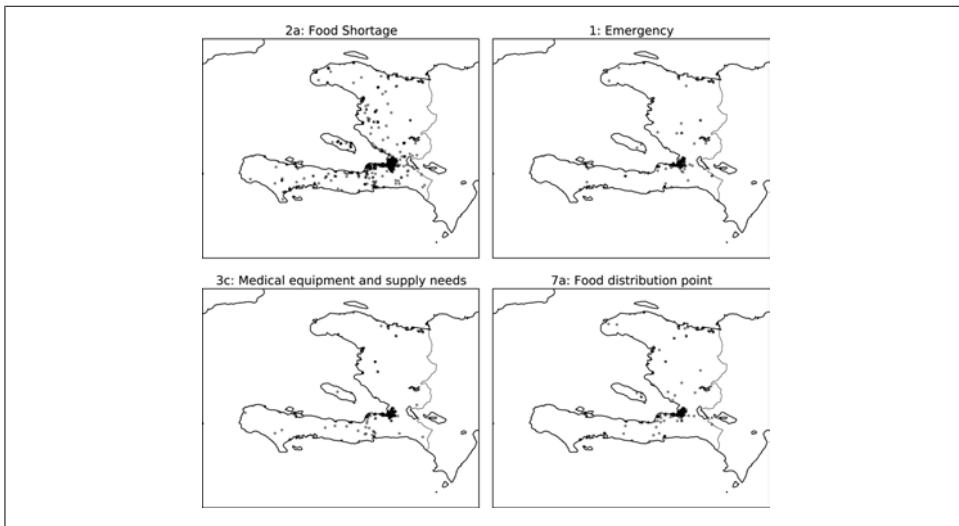


Figure 8-24. Haiti crisis data for 4 categories

of report categories. For each category, I filter down the data set to the coordinates labeled by that category, plot a Basemap on the appropriate subplot, transform the coordinates, then plot the points using the Basemap's plot method:

```
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 10))
fig.subplots_adjust(hspace=0.05, wspace=0.05)

to_plot = ['2a', '1', '3c', '7a']

lllat=17.25; urlat=20.25; llon=-75; urlon=-71

for code, ax in zip(to_plot, axes.flat):
    m = basic_haiti_map(ax, lllat=lllat, urlat=urlat,
                         llon=llon, urlon=urlon)

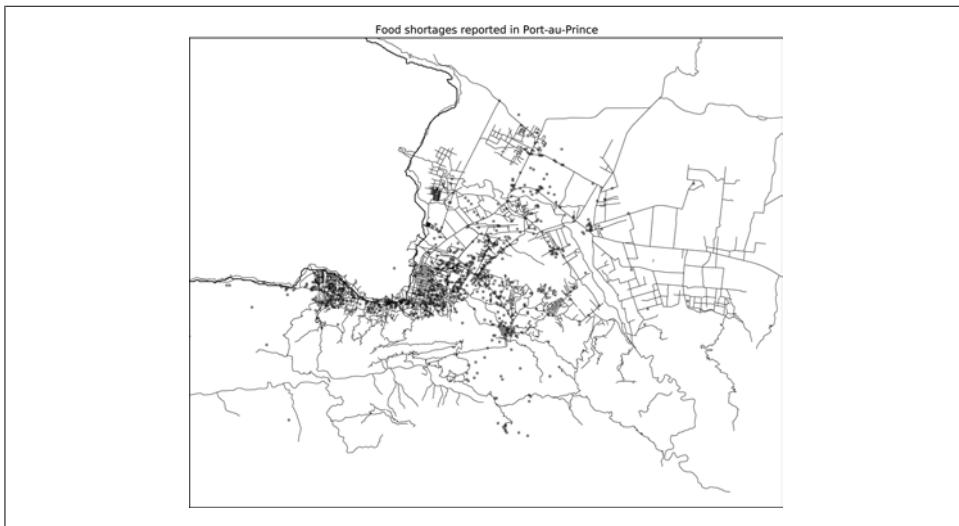
    cat_data = data[data['category_%s' % code] == 1]

    # compute map proj coordinates.
    x, y = m(cat_data.LONGITUDE, cat_data.LATITUDE)

    m.plot(x, y, 'k.', alpha=0.5)
    ax.set_title('%s: %s' % (code, english_mapping[code]))
```

The resulting figure can be seen in Figure 8-24.

It seems from the plot that most of the data is concentrated around the most populous city, Port-au-Prince. `basemap` allows you to overlap additional map data which comes from what are called *shapefiles*. I first downloaded a shapefile with roads in Port-au-Prince (see [http://cegrp.cga.harvard.edu/haiti/?q=resources\\_data](http://cegrp.cga.harvard.edu/haiti/?q=resources_data)). The `Basemap` object conveniently has a `readshapefile` method so that, after extracting the road data archive, I added just the following lines to my code:



*Figure 8-25. Food shortage reports in Port-au-Prince during the Haiti earthquake crisis*

```
shapefile_path = 'ch08/PortAuPrince_Roads/PortAuPrince_Roads'
m.readshapefile(shapefile_path, 'roads')
```

After a little more trial and error with the latitude and longitude boundaries, I was able to make [Figure 8-25](#) for the “Food shortage” category.

## Python Visualization Tool Ecosystem

As is common with open source, there are a plethora of options for creating graphics in Python (too many to list). In addition to open source, there are numerous commercial libraries with Python bindings.

In this chapter and throughout the book, I have been primarily concerned with matplotlib as it is the most widely used plotting tool in Python. While it’s an important part of the scientific Python ecosystem, matplotlib has plenty of shortcomings when it comes to the creation and display of statistical graphics. MATLAB users will likely find matplotlib familiar, while R users (especially users of the excellent `ggplot2` and `trellis` packages) may be somewhat disappointed (at least as of this writing). It is possible to make beautiful plots for display on the web in matplotlib, but doing so often requires significant effort as the library is designed for the printed page. Aesthetics aside, it is sufficient for most needs. In pandas, I, along with the other developers, have sought to build a convenient user interface that makes it easier to make most kinds of plots commonplace in data analysis.

There are a number of other visualization tools in wide use. I list a few of them here and encourage you to explore the ecosystem.

## Chaco

Chaco (<http://code.enthought.com/chaco/>), developed by Enthought, is a plotting toolkit suitable both for static plotting and interactive visualizations. It is especially well-suited for expressing complex visualizations with data interrelationships. Compared with matplotlib, Chaco has much better support for interacting with plot elements and rendering is very fast, making it a good choice for building interactive GUI applications.

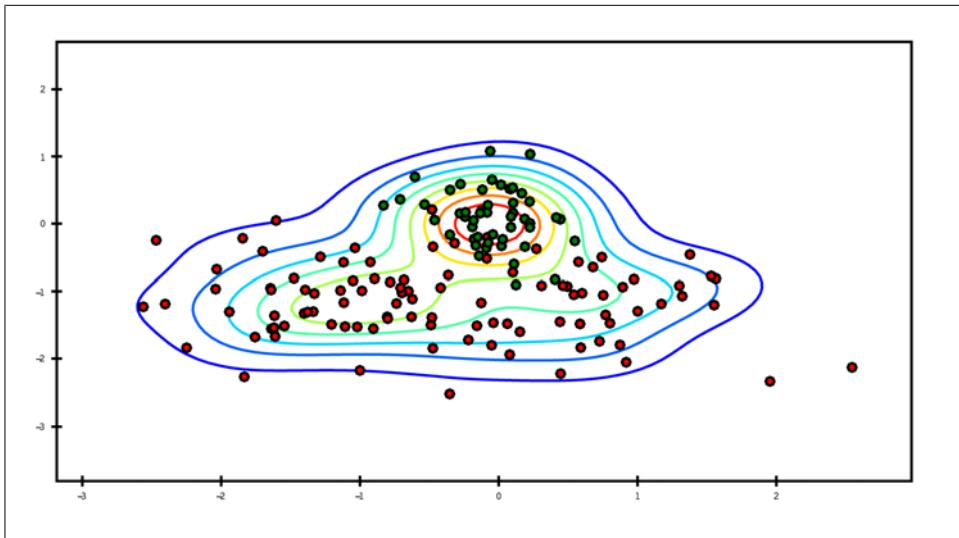


Figure 8-26. A Chaco example plot

## mayavi

The mayavi project, developed by Prabhu Ramachandran, Gaël Varoquaux, and others, is a 3D graphics toolkit built on the open source C++ graphics library VTK. mayavi, like matplotlib, integrates with IPython so that it is easy to use interactively. The plots can be panned, rotated, and zoomed using the mouse and keyboard. I used mayavi to make one of the illustrations of broadcasting in [Chapter 12](#). While I don't show any mayavi-using code here, there is plenty of documentation and examples available online. In many cases, I believe it is a good alternative to a technology like WebGL, though the graphics are harder to share in interactive form.

## Other Packages

Of course, there are numerous other visualization libraries and applications available in Python: PyQwt, Veusz, gnuplot-py, biggles, and others. I have seen PyQwt put to good use in GUI applications built using the Qt application framework using PyQt. While many of these libraries continue to be under active development (some of them

are part of much larger applications), I have noted in the last few years a general trend toward web-based technologies and away from desktop graphics. I'll say a few more words about this in the next section.

## The Future of Visualization Tools?

Visualizations built on web technologies (that is, JavaScript-based) appear to be the inevitable future. Doubtlessly you have used many different kinds of static or interactive visualizations built in Flash or JavaScript over the years. New toolkits (such as d3.js and its numerous off-shoot projects) for building such displays are appearing all the time. In contrast, development in non web-based visualization has slowed significantly in recent years. This holds true of Python as well as other data analysis and statistical computing environments like R.

The development challenge, then, will be in building tighter integration between data analysis and preparation tools, such as pandas, and the web browser. I am hopeful that this will become a fruitful point of collaboration between Python and non-Python users as well.



# Data Aggregation and Group Operations

Categorizing a data set and applying a function to each group, whether an aggregation or transformation, is often a critical component of a data analysis workflow. After loading, merging, and preparing a data set, a familiar task is to compute group statistics or possibly *pivot tables* for reporting or visualization purposes. pandas provides a flexible and high-performance `groupby` facility, enabling you to slice and dice, and summarize data sets in a natural way.

One reason for the popularity of relational databases and SQL (which stands for “structured query language”) is the ease with which data can be joined, filtered, transformed, and aggregated. However, query languages like SQL are rather limited in the kinds of group operations that can be performed. As you will see, with the expressiveness and power of Python and pandas, we can perform much more complex grouped operations by utilizing any function that accepts a pandas object or NumPy array. In this chapter, you will learn how to:

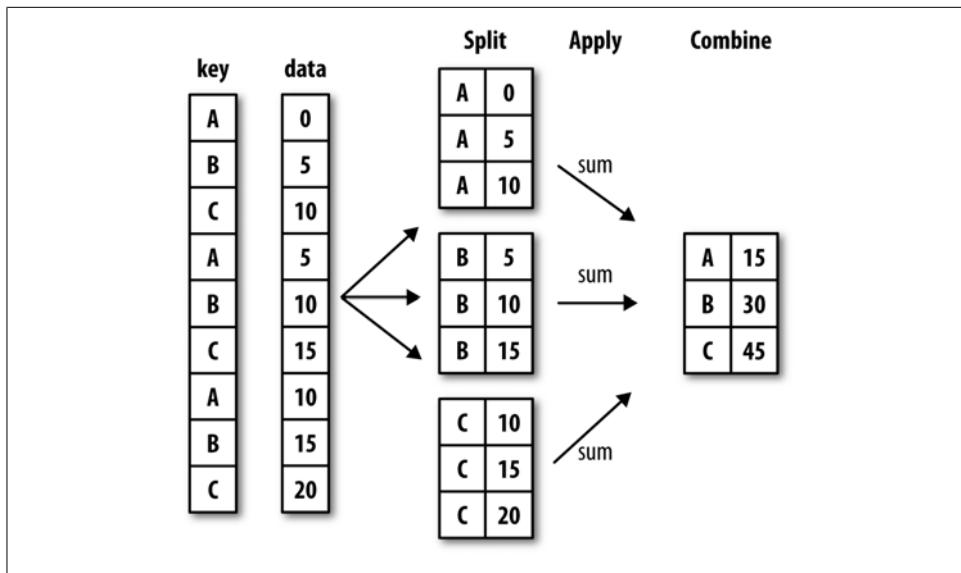
- Split a pandas object into pieces using one or more keys (in the form of functions, arrays, or DataFrame column names)
- Computing group summary statistics, like count, mean, or standard deviation, or a user-defined function
- Apply a varying set of functions to each column of a DataFrame
- Apply within-group transformations or other manipulations, like normalization, linear regression, rank, or subset selection
- Compute pivot tables and cross-tabulations
- Perform quantile analysis and other data-derived group analyses



Aggregation of time series data, a special use case of `groupby`, is referred to as *resampling* in this book and will receive separate treatment in [Chapter 10](#).

## GroupBy Mechanics

Hadley Wickham, an author of many popular packages for the R programming language, coined the term *split-apply-combine* for talking about group operations, and I think that's a good description of the process. In the first stage of the process, data contained in a pandas object, whether a Series, DataFrame, or otherwise, is *split* into groups based on one or more *keys* that you provide. The splitting is performed on a particular axis of an object. For example, a DataFrame can be grouped on its rows (`axis=0`) or its columns (`axis=1`). Once this is done, a function is *applied* to each group, producing a new value. Finally, the results of all those function applications are *combined* into a result object. The form of the resulting object will usually depend on what's being done to the data. See [Figure 9-1](#) for a mockup of a simple group aggregation.



*Figure 9-1. Illustration of a group aggregation*

Each grouping key can take many forms, and the keys do not have to be all of the same type:

- A list or array of values that is the same length as the axis being grouped
- A value indicating a column name in a DataFrame

- A dict or Series giving a correspondence between the values on the axis being grouped and the group names
- A function to be invoked on the axis index or the individual labels in the index

Note that the latter three methods are all just shortcuts for producing an array of values to be used to split up the object. Don't worry if this all seems very abstract. Throughout this chapter, I will give many examples of all of these methods. To get started, here is a very simple small tabular dataset as a DataFrame:

```
In [13]: df = DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
....:                   'key2' : ['one', 'two', 'one', 'two', 'one'],
....:                   'data1' : np.random.randn(5),
....:                   'data2' : np.random.randn(5)})
```

```
In [14]: df
Out[14]:
```

	data1	data2	key1	key2
0	-0.204708	1.393406	a	one
1	0.478943	0.092908	a	two
2	-0.549439	0.281746	b	one
3	-0.555730	0.769023	b	two
4	1.965781	1.246435	a	one

Suppose you wanted to compute the mean of the `data1` column using the groups labels from `key1`. There are a number of ways to do this. One is to access `data1` and call `groupby` with the column (a Series) at `key1`:

```
In [15]: grouped = df['data1'].groupby(df['key1'])

In [16]: grouped
Out[16]: <pandas.core.groupby.SeriesGroupBy at 0x2d78b10>
```

This `grouped` variable is now a `GroupBy` object. It has not actually computed anything yet except for some intermediate data about the group key `df['key1']`. The idea is that this object has all of the information needed to then apply some operation to each of the groups. For example, to compute group means we can call the `GroupBy`'s `mean` method:

```
In [17]: grouped.mean()
Out[17]:
key1
a      0.746672
b     -0.537585
```

Later, I'll explain more about what's going on when you call `.mean()`. The important thing here is that the data (a Series) has been aggregated according to the group key, producing a new Series that is now indexed by the unique values in the `key1` column. The result index has the name '`key1`' because the DataFrame column `df['key1']` did.

If instead we had passed multiple arrays as a list, we get something different:

```
In [18]: means = df['data1'].groupby([df['key1'], df['key2']]).mean()
```

```
In [19]: means
Out[19]:
key1  key2
a    one      0.880536
     two      0.478943
b    one     -0.519439
     two     -0.555730
```

In this case, we grouped the data using two keys, and the resulting Series now has a hierarchical index consisting of the unique pairs of keys observed:

```
In [20]: means.unstack()
Out[20]:
key2      one      two
key1
a      0.880536  0.478943
b     -0.519439 -0.555730
```

In these examples, the group keys are all Series, though they could be any arrays of the right length:

```
In [21]: states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])
In [22]: years = np.array([2005, 2005, 2006, 2005, 2006])
In [23]: df['data1'].groupby([states, years]).mean()
Out[23]:
California  2005      0.478943
              2006     -0.519439
Ohio        2005     -0.380219
              2006     1.965781
```

Frequently the grouping information to be found in the same DataFrame as the data you want to work on. In that case, you can pass column names (whether those are strings, numbers, or other Python objects) as the group keys:

```
In [24]: df.groupby('key1').mean()
Out[24]:
          data1      data2
key1
a      0.746672  0.910916
b     -0.537585  0.525384

In [25]: df.groupby(['key1', 'key2']).mean()
Out[25]:
          data1      data2
key1 key2
a   one    0.880536  1.319920
    two    0.478943  0.092908
b   one   -0.519439  0.281746
    two   -0.555730  0.769023
```

You may have noticed in the first case `df.groupby('key1').mean()` that there is no `key2` column in the result. Because `df['key2']` is not numeric data, it is said to be a *nuisance column*, which is therefore excluded from the result. By default, all of the

numeric columns are aggregated, though it is possible to filter down to a subset as you'll see soon.

Regardless of the objective in using `groupby`, a generally useful GroupBy method is `size` which return a Series containing group sizes:

```
In [26]: df.groupby(['key1', 'key2']).size()
Out[26]:
key1  key2
a      one      2
        two      1
b      one      1
        two      1
```



As of this writing, any missing values in a group key will be excluded from the result. It's possible (and, in fact, quite likely), that by the time you are reading this there will be an option to include the `NA` group in the result.

## Iterating Over Groups

The GroupBy object supports iteration, generating a sequence of 2-tuples containing the group name along with the chunk of data. Consider the following small example data set:

```
In [27]: for name, group in df.groupby('key1'):
....:     print name
....:     print group
....:
a
    data1      data2 key1 key2
0 -0.204708  1.393406   a  one
1  0.478943  0.092908   a  two
4  1.965781  1.246435   a  one
b
    data1      data2 key1 key2
2 -0.519439  0.281746   b  one
3 -0.555730  0.769023   b  two
```

In the case of multiple keys, the first element in the tuple will be a tuple of key values:

```
In [28]: for (k1, k2), group in df.groupby(['key1', 'key2']):
....:     print k1, k2
....:     print group
....:
a one
    data1      data2 key1 key2
0 -0.204708  1.393406   a  one
4  1.965781  1.246435   a  one
a two
    data1      data2 key1 key2
1  0.478943  0.092908   a  two
b one
    data1      data2 key1 key2
```

```
2 -0.519439  0.281746    b  one
b two
    data1      data2 key1 key2
3 -0.55573  0.769023    b  two
```

Of course, you can choose to do whatever you want with the pieces of data. A recipe you may find useful is computing a dict of the data pieces as a one-liner:

```
In [29]: pieces = dict(list(df.groupby('key1')))
```

```
In [30]: pieces['b']
Out[30]:
    data1      data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.55573  0.769023    b  two
```

By default `groupby` groups on `axis=0`, but you can group on any of the other axes. For example, we could group the columns of our example `df` here by `dtype` like so:

```
In [31]: df.dtypes
Out[31]:
data1    float64
data2    float64
key1     object
key2     object
```

```
In [32]: grouped = df.groupby(df.dtypes, axis=1)
```

```
In [33]: dict(list(grouped))
Out[33]:
{dtype('float64'):      data1      data2
0 -0.204708  1.393406
1  0.478943  0.092908
2 -0.519439  0.281746
3 -0.555730  0.769023
4  1.965781  1.246435,
 dtype('object'):   key1 key2
0    a  one
1    a  two
2    b  one
3    b  two
4    a  one}
```

## Selecting a Column or Subset of Columns

Indexing a `GroupBy` object created from a `DataFrame` with a column name or array of column names has the effect of *selecting those columns* for aggregation. This means that:

```
df.groupby('key1')[['data1']]
df.groupby('key1')[[['data2']]]
```

are syntactic sugar for:

```
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])
```

Especially for large data sets, it may be desirable to aggregate only a few columns. For example, in the above data set, to compute means for just the `data2` column and get the result as a DataFrame, we could write:

```
In [34]: df.groupby(['key1', 'key2'])[['data2']].mean()
Out[34]:
          data2
key1 key2
a    one    1.319920
     two    0.092908
b    one    0.281746
     two    0.769023
```

The object returned by this indexing operation is a grouped DataFrame if a list or array is passed and a grouped Series is just a single column name that is passed as a scalar:

```
In [35]: s_grouped = df.groupby(['key1', 'key2'])['data2']
```

```
In [36]: s_grouped
Out[36]: <pandas.core.groupby.SeriesGroupBy at 0x2e215d0>
```

```
In [37]: s_grouped.mean()
Out[37]:
          data2
key1 key2
a    one    1.319920
     two    0.092908
b    one    0.281746
     two    0.769023
```

Name: data2

## Grouping with Dicts and Series

Grouping information may exist in a form other than an array. Let's consider another example DataFrame:

```
In [38]: people = DataFrame(np.random.randn(5, 5),
....:                      columns=['a', 'b', 'c', 'd', 'e'],
....:                      index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])
```

```
In [39]: people.ix[2:3, ['b', 'c']] = np.nan # Add a few NA values
```

```
In [40]: people
Out[40]:
          a         b         c         d         e
Joe    1.007189 -1.296221  0.274992  0.228913  1.352917
Steve   0.886429 -2.001637 -0.371843  1.669025 -0.438570
Wes    -0.539741      NaN      NaN -1.021228 -0.577087
Jim    0.124121  0.302614  0.523772  0.000940  1.343810
Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Now, suppose I have a group correspondence for the columns and want to sum together the columns by group:

```
In [41]: mapping = {'a': 'red', 'b': 'red', 'c': 'blue',
....:                 'd': 'blue', 'e': 'red', 'f': 'orange'}
```

Now, you could easily construct an array from this dict to pass to `groupby`, but instead we can just pass the dict:

```
In [42]: by_column = people.groupby(mapping, axis=1)
```

```
In [43]: by_column.sum()
```

```
Out[43]:
```

	blue	red
Joe	0.503905	1.063885
Steve	1.297183	-1.553778
Wes	-1.021228	-1.116829
Jim	0.524712	1.770545
Travis	-4.230992	-2.405455

The same functionality holds for Series, which can be viewed as a fixed size mapping. When I used Series as group keys in the above examples, pandas does, in fact, inspect each Series to ensure that its index is aligned with the axis it's grouping:

```
In [44]: map_series = Series(mapping)
```

```
In [45]: map_series
```

```
Out[45]:
```

	red
a	red
b	red
c	blue
d	blue
e	red
f	orange

```
In [46]: people.groupby(map_series, axis=1).count()
```

```
Out[46]:
```

	blue	red
Joe	2	3
Steve	2	3
Wes	1	2
Jim	2	3
Travis	2	3

## Grouping with Functions

Using Python functions in what can be fairly creative ways is a more abstract way of defining a group mapping compared with a dict or Series. Any function passed as a group key will be called once per index value, with the return values being used as the group names. More concretely, consider the example DataFrame from the previous section, which has people's first names as index values. Suppose you wanted to group by the length of the names; you could compute an array of string lengths, but instead you can just pass the `len` function:

```
In [47]: people.groupby(len).sum()
```

```
Out[47]:
```

	a	b	c	d	e
3	0.591569	-0.993608	0.798764	-0.791374	2.119639

```
5  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Mixing functions with arrays, dicts, or Series is not a problem as everything gets converted to arrays internally:

```
In [48]: key_list = ['one', 'one', 'one', 'two', 'two']
```

```
In [49]: people.groupby([len, key_list]).min()
```

```
Out[49]:
```

	a	b	c	d	e
3 one	-0.539741	-1.296221	0.274992	-1.021228	-0.577087
two	0.124121	0.302614	0.523772	0.000940	1.343810
5 one	0.886429	-2.001637	-0.371843	1.669025	-0.438570
6 two	-0.713544	-0.831154	-2.370232	-1.860761	-0.860757

## Grouping by Index Levels

A final convenience for hierarchically-indexed data sets is the ability to aggregate using one of the levels of an axis index. To do this, pass the level number or name using the `level` keyword:

```
In [50]: columns = pd.MultiIndex.from_arrays([[['US', 'US', 'US', 'JP', 'JP'],
                                             [1, 3, 5, 1, 3]], names=['cty', 'tenor'])
```

```
In [51]: hier_df = DataFrame(np.random.randn(4, 5), columns=columns)
```

```
In [52]: hier_df
```

```
Out[52]:
```

	cty	US		JP		
		1	3	5	1	3
0	0.560145	-1.265934	0.119827	-1.063512	0.332883	
1	-2.359419	-0.199543	-1.541996	-0.970736	-1.307030	
2	0.286350	0.377984	-0.753887	0.331286	1.349742	
3	0.069877	0.246674	-0.011862	1.004812	1.327195	

```
In [53]: hier_df.groupby(level='cty', axis=1).count()
```

```
Out[53]:
```

cty	JP	US
0	2	3
1	2	3
2	2	3
3	2	3

## Data Aggregation

By aggregation, I am generally referring to any data transformation that produces scalar values from arrays. In the examples above I have used several of them, such as `mean`, `count`, `min` and `sum`. You may wonder what is going on when you invoke `mean()` on a `GroupBy` object. Many common aggregations, such as those found in [Table 9-1](#), have optimized implementations that compute the statistics on the dataset *in place*. However, you are not limited to only this set of methods. You can use aggregations of your

own devising and additionally call any method that is also defined on the grouped object. For example, as you recall `quantile` computes sample quantiles of a Series or a DataFrame's columns<sup>1</sup>:

```
In [54]: df
Out[54]:
   data1      data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one

In [55]: grouped = df.groupby('key1')

In [56]: grouped['data1'].quantile(0.9)
Out[56]:
key1
a      1.668413
b     -0.523068
```

While `quantile` is not explicitly implemented for GroupBy, it is a Series method and thus available for use. Internally, GroupBy efficiently slices up the Series, calls `piece.quantile(0.9)` for each piece, then assembles those results together into the result object.

To use your own aggregation functions, pass any function that aggregates an array to the `aggregate` or `agg` method:

```
In [57]: def peak_to_peak(arr):
....:     return arr.max() - arr.min()

In [58]: grouped.agg(peak_to_peak)
Out[58]:
   data1      data2
key1
a      2.170488  1.300498
b      0.036292  0.487276
```

You'll notice that some methods like `describe` also work, even though they are not aggregations, strictly speaking:

```
In [59]: grouped.describe()
Out[59]:
              data1      data2
key1
a    count    3.000000  3.000000
     mean     0.746672  0.910916
     std      1.109736  0.712217
     min     -0.204708  0.092908
     25%      0.137118  0.669671
     50%      0.478943  1.246435
```

1. Note that `quantile` performs linear interpolation if there is no value at exactly the passed percentile.

```

    75%   1.222362  1.319920
    max   1.965781  1.393406
b  count  2.000000  2.000000
        mean -0.537585  0.525384
        std   0.025662  0.344556
        min   -0.555730  0.281746
        25%   -0.546657  0.403565
        50%   -0.537585  0.525384
        75%   -0.528512  0.647203
        max   -0.519439  0.769023

```

I will explain in more detail what has happened here in the next major section on group-wise operations and transformations.



You may notice that custom aggregation functions are much slower than the optimized functions found in [Table 9-1](#). This is because there is significant overhead (function calls, data rearrangement) in constructing the intermediate group data chunks.

*Table 9-1. Optimized groupby methods*

Function name	Description
count	Number of non-NA values in the group
sum	Sum of non-NA values
mean	Mean of non-NA values
median	Arithmetic median of non-NA values
std, var	Unbiased ( $n - 1$ denominator) standard deviation and variance
min, max	Minimum and maximum of non-NA values
prod	Product of non-NA values
first, last	First and last non-NA values

To illustrate some more advanced aggregation features, I'll use a less trivial dataset, a dataset on restaurant tipping. I obtained it from the R `reshape2` package; it was originally found in Bryant & Smith's 1995 text on business statistics (and found in the book's GitHub repository). After loading it with `read_csv`, I add a tipping percentage column `tip_pct`.

```

In [60]: tips = pd.read_csv('ch08/tips.csv')

# Add tip percentage of total bill
In [61]: tips['tip_pct'] = tips['tip'] / tips['total_bill']

In [62]: tips[:6]
Out[62]:
   total_bill  tip     sex smoker  day    time  size  tip_pct
0       16.99  1.01  Female     No  Sun  Dinner     2  0.059447
1       10.34  1.66   Male     No  Sun  Dinner     3  0.160542

```

```

2      21.01  3.50   Male    No Sun Dinner     3  0.166587
3      23.68  3.31   Male    No Sun Dinner     2  0.139780
4      24.59  3.61 Female  No Sun Dinner     4  0.146808
5      25.29  4.71   Male    No Sun Dinner     4  0.186240

```

## Column-wise and Multiple Function Application

As you've seen above, aggregating a Series or all of the columns of a DataFrame is a matter of using `aggregate` with the desired function or calling a method like `mean` or `std`. However, you may want to aggregate using a different function depending on the column or multiple functions at once. Fortunately, this is straightforward to do, which I'll illustrate through a number of examples. First, I'll group the `tips` by `sex` and `smoker`:

```
In [63]: grouped = tips.groupby(['sex', 'smoker'])
```

Note that for descriptive statistics like those in [Table 9-1](#), you can pass the name of the function as a string:

```
In [64]: grouped_pct = grouped['tip_pct']
```

```
In [65]: grouped_pct.agg('mean')
Out[65]:
sex      smoker
Female  No        0.156921
                  Yes       0.182150
Male    No        0.160669
                  Yes       0.152771
Name: tip_pct
```

If you pass a list of functions or function names instead, you get back a DataFrame with column names taken from the functions:

```
In [66]: grouped_pct.agg(['mean', 'std', 'peak_to_peak'])
Out[66]:
               mean      std  peak_to_peak
sex      smoker
Female  No        0.156921  0.036421    0.195876
                  Yes       0.182150  0.071595    0.360233
Male    No        0.160669  0.041849    0.220186
                  Yes       0.152771  0.090588    0.674707
```

You don't need to accept the names that GroupBy gives to the columns; notably `lambda` functions have the name '`<lambda>`' which make them hard to identify (you can see for yourself by looking at a function's `_name_` attribute). As such, if you pass a list of (`name`, `function`) tuples, the first element of each tuple will be used as the DataFrame column names (you can think of a list of 2-tuples as an ordered mapping):

```
In [67]: grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])
Out[67]:
          foo      bar
sex      smoker
Female  No        0.156921  0.036421
                  Yes       0.182150  0.071595
```

```
Male  No      0.160669  0.041849
      Yes     0.152771  0.090588
```

With a DataFrame, you have more options as you can specify a list of functions to apply to all of the columns or different functions per column. To start, suppose we wanted to compute the same three statistics for the `tip_pct` and `total_bill` columns:

```
In [68]: functions = ['count', 'mean', 'max']
```

```
In [69]: result = grouped[['tip_pct', 'total_bill']].agg(functions)
```

```
In [70]: result
```

```
Out[70]:
```

		tip_pct			total_bill		
		count	mean	max	count	mean	max
sex	smoker						
Female	No	54	0.156921	0.252672	54	18.105185	35.83
	Yes	33	0.182150	0.416667	33	17.977879	44.30
Male	No	97	0.160669	0.291990	97	19.791237	48.33
	Yes	60	0.152771	0.710345	60	22.284500	50.81

As you can see, the resulting DataFrame has hierarchical columns, the same as you would get aggregating each column separately and using `concat` to glue the results together using the column names as the `keys` argument:

```
In [71]: result['tip_pct']
```

```
Out[71]:
```

		count	mean	max
sex	smoker			
Female	No	54	0.156921	0.252672
	Yes	33	0.182150	0.416667
Male	No	97	0.160669	0.291990
	Yes	60	0.152771	0.710345

As above, a list of tuples with custom names can be passed:

```
In [72]: ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]
```

```
In [73]: grouped[['tip_pct', 'total_bill']].agg(ftuples)
```

```
Out[73]:
```

		tip_pct		total_bill	
		Durchschnitt	Abweichung	Durchschnitt	Abweichung
sex	smoker				
Female	No	0.156921	0.001327	18.105185	53.092422
	Yes	0.182150	0.005126	17.977879	84.454517
Male	No	0.160669	0.001751	19.791237	76.152961
	Yes	0.152771	0.008206	22.284500	98.244673

Now, suppose you wanted to apply potentially different functions to one or more of the columns. The trick is to pass a dict to `agg` that contains a mapping of column names to any of the function specifications listed so far:

```
In [74]: grouped.agg({'tip' : np.max, 'size' : 'sum'})
```

```
Out[74]:
```

size	tip
sex	smoker

```

Female No      140   5.2
        Yes     74   6.5
Male   No      263   9.0
        Yes     150  10.0

```

```
In [75]: grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],
....:                  'size' : 'sum'})
Out[75]:
```

sex	smoker	tip_pct				size sum
		min	max	mean	std	
Female	No	0.056797	0.252672	0.156921	0.036421	140
	Yes	0.056433	0.416667	0.182150	0.071595	74
Male	No	0.071804	0.291990	0.160669	0.041849	263
	Yes	0.035638	0.710345	0.152771	0.090588	150

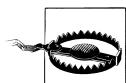
A DataFrame will have hierarchical columns only if multiple functions are applied to at least one column.

## Returning Aggregated Data in “unindexed” Form

In all of the examples up until now, the aggregated data comes back with an index, potentially hierarchical, composed from the unique group key combinations observed. Since this isn’t always desirable, you can disable this behavior in most cases by passing `as_index=False` to `groupby`:

```
In [76]: tips.groupby(['sex', 'smoker'], as_index=False).mean()
Out[76]:
   sex smoker  total_bill      tip      size  tip_pct
0  Female    No    18.105185  2.773519  2.592593  0.156921
1  Female   Yes    17.977879  2.931515  2.242424  0.182150
2   Male    No    19.791237  3.113402  2.711340  0.160669
3   Male   Yes    22.284500  3.051167  2.500000  0.152771
```

Of course, it’s always possible to obtain the result in this format by calling `reset_index` on the result.



Using `groupby` in this way is generally less flexible; results with hierarchical columns, for example, are not currently implemented as the form of the result would have to be somewhat arbitrary.

## Group-wise Operations and Transformations

Aggregation is only one kind of group operation. It is a special case in the more general class of data transformations; that is, it accepts functions that reduce a one-dimensional array to a scalar value. In this section, I will introduce you to the `transform` and `apply` methods, which will enable you to do many other kinds of group operations.

Suppose, instead, we wanted to add a column to a DataFrame containing group means for each index. One way to do this is to aggregate, then merge:

```
In [77]: df
Out[77]:
   data1      data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one

In [78]: k1_means = df.groupby('key1').mean().add_prefix('mean_')

In [79]: k1_means
Out[79]:
   mean_data1  mean_data2
key1
a      0.746672  0.910916
b     -0.537585  0.525384

In [80]: pd.merge(df, k1_means, left_on='key1', right_index=True)
Out[80]:
   data1      data2 key1 key2  mean_data1  mean_data2
0 -0.204708  1.393406    a  one      0.746672  0.910916
1  0.478943  0.092908    a  two      0.746672  0.910916
4  1.965781  1.246435    a  one      0.746672  0.910916
2 -0.519439  0.281746    b  one     -0.537585  0.525384
3 -0.555730  0.769023    b  two     -0.537585  0.525384
```

This works, but is somewhat inflexible. You can think of the operation as transforming the two data columns using the `np.mean` function. Let's look back at the `people` DataFrame from earlier in the chapter and use the `transform` method on GroupBy:

```
In [81]: key = ['one', 'two', 'one', 'two', 'one']

In [82]: people.groupby(key).mean()
Out[82]:
   a      b      c      d      e
one -0.082032 -1.063687 -1.047620 -0.884358 -0.028309
two  0.505275 -0.849512  0.075965  0.834983  0.452620

In [83]: people.groupby(key).transform(np.mean)
Out[83]:
   a      b      c      d      e
Joe   -0.082032 -1.063687 -1.047620 -0.884358 -0.028309
Steve  0.505275 -0.849512  0.075965  0.834983  0.452620
Wes   -0.082032 -1.063687 -1.047620 -0.884358 -0.028309
Jim   0.505275 -0.849512  0.075965  0.834983  0.452620
Travis -0.082032 -1.063687 -1.047620 -0.884358 -0.028309
```

As you may guess, `transform` applies a function to each group, then places the results in the appropriate locations. If each group produces a scalar value, it will be propagated (broadcasted). Suppose instead you wanted to subtract the mean value from each group. To do this, create a demeaning function and pass it to `transform`:

```
In [84]: def demean(arr):
....:     return arr - arr.mean()
```

```
In [85]: demeaned = people.groupby(key).transform(demean)
```

```
In [86]: demeaned
```

```
Out[86]:
```

	a	b	c	d	e
Joe	1.089221	-0.232534	1.322612	1.113271	1.381226
Steve	0.381154	-1.152125	-0.447807	0.834043	-0.891190
Wes	-0.457709		NaN	NaN	-0.136869
Jim	-0.381154	1.152125	0.447807	-0.834043	0.891190
Travis	-0.631512	0.232534	-1.322612	-0.976402	-0.832448

You can check that `demeaned` now has zero group means:

```
In [87]: demeaned.groupby(key).mean()
```

```
Out[87]:
```

	a	b	c	d	e
one	0	-0	0	0	0
two	-0	0	0	0	0

As you'll see in the next section, group demeaning can be achieved using `apply` also.

## Apply: General split-apply-combine

Like `aggregate`, `transform` is a more specialized function having rigid requirements: the passed function must either produce a scalar value to be broadcasted (like `np.mean`) or a transformed array of the same size. The most general purpose GroupBy method is `apply`, which is the subject of the rest of this section. As in [Figure 9-1](#), `apply` splits the object being manipulated into pieces, invokes the passed function on each piece, then attempts to concatenate the pieces together.

Returning to the tipping data set above, suppose you wanted to select the top five `tip_pct` values by group. First, it's straightforward to write a function that selects the rows with the largest values in a particular column:

```
In [88]: def top(df, n=5, column='tip_pct'):  
....:     return df.sort_index(by=column)[-n:]
```

```
In [89]: top(tips, n=6)
```

```
Out[89]:
```

	total_bill	tip	sex	smoker	day	time	size	tip_pct
109	14.31	4.00	Female	Yes	Sat	Dinner	2	0.279525
183	23.17	6.50	Male	Yes	Sun	Dinner	4	0.280535
232	11.61	3.39	Male	No	Sat	Dinner	2	0.291990
67	3.07	1.00	Female	Yes	Sat	Dinner	1	0.325733
178	9.60	4.00	Female	Yes	Sun	Dinner	2	0.416667
172	7.25	5.15	Male	Yes	Sun	Dinner	2	0.710345

Now, if we group by `smoker`, say, and call `apply` with this function, we get the following:

```
In [90]: tips.groupby('smoker').apply(top)
```

```
Out[90]:
```

smoker	total_bill	tip	sex	smoker	day	time	size	tip_pct
--------	------------	-----	-----	--------	-----	------	------	---------

No	88	24.71	5.85	Male	No	Thur	Lunch	2	0.236746
	185	20.69	5.00	Male	No	Sun	Dinner	5	0.241663
	51	10.29	2.60	Female	No	Sun	Dinner	2	0.252672
	149	7.51	2.00	Male	No	Thur	Lunch	2	0.266312
	232	11.61	3.39	Male	No	Sat	Dinner	2	0.291990
Yes	109	14.31	4.00	Female	Yes	Sat	Dinner	2	0.279525
	183	23.17	6.50	Male	Yes	Sun	Dinner	4	0.280535
	67	3.07	1.00	Female	Yes	Sat	Dinner	1	0.325733
	178	9.60	4.00	Female	Yes	Sun	Dinner	2	0.416667
	172	7.25	5.15	Male	Yes	Sun	Dinner	2	0.710345

What has happened here? The `top` function is called on each piece of the DataFrame, then the results are glued together using `pandas.concat`, labeling the pieces with the group names. The result therefore has a hierarchical index whose inner level contains index values from the original DataFrame.

If you pass a function to `apply` that takes other arguments or keywords, you can pass these after the function:

```
In [91]: tips.groupby(['smoker', 'day']).apply(top, n=1, column='total_bill')
Out[91]:
          total_bill    tip   sex smoker  day   time  size  tip_pct
smoker day
No     Fri    94    22.75  3.25 Female    No   Fri Dinner    2  0.142857
      Sat   212    48.33  9.00 Male     No   Sat Dinner    4  0.186220
      Sun   156    48.17  5.00 Male     No   Sun Dinner    6  0.103799
      Thur  142    41.19  5.00 Male     No Thur Lunch    5  0.121389
Yes    Fri    95    40.17  4.73 Male    Yes   Fri Dinner    4  0.117750
      Sat   170    50.81 10.00 Male    Yes   Sat Dinner    3  0.196812
      Sun   182    45.35  3.50 Male    Yes   Sun Dinner    3  0.077178
      Thur  197    43.11  5.00 Female Yes Thur Lunch    4  0.115982
```



Beyond these basic usage mechanics, getting the most out of `apply` is largely a matter of creativity. What occurs inside the function passed is up to you; it only needs to return a pandas object or a scalar value. The rest of this chapter will mainly consist of examples showing you how to solve various problems using `groupby`.

You may recall above I called `describe` on a GroupBy object:

```
In [92]: result = tips.groupby('smoker')['tip_pct'].describe()
```

```
In [93]: result
Out[93]:
smoker
No    count    151.000000
      mean     0.159328
      std     0.039910
      min     0.056797
      25%     0.136906
      50%     0.155625
      75%     0.185014
      max     0.291990
```

```
Yes    count    93.000000
      mean     0.163196
      std      0.085119
      min      0.035638
      25%     0.106771
      50%     0.153846
      75%     0.195059
      max      0.710345
```

```
In [94]: result.unstack('smoker')
Out[94]:
smoker      No      Yes
count  151.000000  93.000000
mean   0.159328  0.163196
std    0.039910  0.085119
min    0.056797  0.035638
25%   0.136906  0.106771
50%   0.155625  0.153846
75%   0.185014  0.195059
max   0.291990  0.710345
```

Inside GroupBy, when you invoke a method like `describe`, it is actually just a shortcut for:

```
f = lambda x: x.describe()
grouped.apply(f)
```

## Suppressing the group keys

In the examples above, you see that the resulting object has a hierarchical index formed from the group keys along with the indexes of each piece of the original object. This can be disabled by passing `group_keys=False` to `groupby`:

```
In [95]: tips.groupby('smoker', group_keys=False).apply(top)
Out[95]:
   total_bill  tip    sex smoker  day   time  size  tip_pct
88       24.71  5.85  Male    No Thur  Lunch     2  0.236746
185      20.69  5.00  Male    No Sun   Dinner    5  0.241663
51        10.29  2.60 Female   No Sun   Dinner    2  0.252672
149        7.51  2.00  Male    No Thur  Lunch     2  0.266312
232      11.61  3.39  Male    No Sat   Dinner    2  0.291990
109      14.31  4.00 Female   Yes Sat   Dinner    2  0.279525
183      23.17  6.50  Male    Yes Sun   Dinner    4  0.280535
67        3.07  1.00 Female   Yes Sat   Dinner    1  0.325733
178      9.60  4.00 Female   Yes Sun   Dinner    2  0.416667
172      7.25  5.15  Male    Yes Sun   Dinner    2  0.710345
```

## Quantile and Bucket Analysis

As you may recall from [Chapter 7](#), pandas has some tools, in particular `cut` and `qcut`, for slicing data up into buckets with bins of your choosing or by sample quantiles. Combining these functions with `groupby`, it becomes very simple to perform bucket or

quantile analysis on a data set. Consider a simple random data set and an equal-length bucket categorization using `cut`:

```
In [96]: frame = DataFrame({'data1': np.random.randn(1000),
....:                      'data2': np.random.randn(1000)})

In [97]: factor = pd.cut(frame.data1, 4)

In [98]: factor[:10]
Out[98]:
Categorical:
array([(-1.23, 0.489], (-2.956, -1.23], (-1.23, 0.489], (0.489, 2.208],
      (-1.23, 0.489], (0.489, 2.208], (-1.23, 0.489], (-1.23, 0.489],
      (0.489, 2.208], (0.489, 2.208]], dtype=object)
Levels (4): Index([(-2.956, -1.23], (-1.23, 0.489], (0.489, 2.208],
                  (2.208, 3.928]], dtype=object)
```

The Factor object returned by `cut` can be passed directly to `groupby`. So we could compute a set of statistics for the `data2` column like so:

```
In [99]: def get_stats(group):
....:     return {'min': group.min(), 'max': group.max(),
....:             'count': group.count(), 'mean': group.mean()}

In [100]: grouped = frame.data2.groupby(factor)

In [101]: grouped.apply(get_stats).unstack()
Out[101]:
          count      max      mean      min
data1
(-1.23, 0.489]    598  3.260383 -0.002051 -2.989741
(-2.956, -1.23]    95  1.670835 -0.039521 -3.399312
(0.489, 2.208]    297  2.954439  0.081822 -3.745356
(2.208, 3.928]     10  1.765640  0.024750 -1.929776
```

These were equal-length buckets; to compute equal-size buckets based on sample quantiles, use `qcut`. I'll pass `labels=False` to just get quantile numbers.

```
# Return quantile numbers
In [102]: grouping = pd.qcut(frame.data1, 10, labels=False)

In [103]: grouped = frame.data2.groupby(grouping)

In [104]: grouped.apply(get_stats).unstack()
Out[104]:
          count      max      mean      min
0     100  1.670835 -0.049902 -3.399312
1     100  2.628441  0.030989 -1.950098
2     100  2.527939 -0.067179 -2.925113
3     100  3.260383  0.065713 -2.315555
4     100  2.074345 -0.111653 -2.047939
5     100  2.184810  0.052130 -2.989741
6     100  2.458842 -0.021489 -2.223506
7     100  2.954439 -0.026459 -3.056990
8     100  2.735527  0.103406 -3.745356
9     100  2.377020  0.220122 -2.064111
```

## Example: Filling Missing Values with Group-specific Values

When cleaning up missing data, in some cases you will filter out data observations using `dropna`, but in others you may want to impute (fill in) the NA values using a fixed value or some value derived from the data. `fillna` is the right tool to use; for example here I fill in NA values with the mean:

```
In [105]: s = Series(np.random.randn(6))
```

```
In [106]: s[::2] = np.nan
```

```
In [107]: s
```

```
Out[107]:
```

```
0      NaN  
1    -0.125921  
2      NaN  
3   -0.884475  
4      NaN  
5    0.227290
```

```
In [108]: s.fillna(s.mean())
```

```
Out[108]:
```

```
0   -0.261035  
1   -0.125921  
2   -0.261035  
3   -0.884475  
4   -0.261035  
5    0.227290
```

Suppose you need the fill value to vary by group. As you may guess, you need only group the data and use `apply` with a function that calls `fillna` on each data chunk. Here is some sample data on some US states divided into eastern and western states:

```
In [109]: states = ['Ohio', 'New York', 'Vermont', 'Florida',  
.....:           'Oregon', 'Nevada', 'California', 'Idaho']
```

```
In [110]: group_key = ['East'] * 4 + ['West'] * 4
```

```
In [111]: data = Series(np.random.randn(8), index=states)
```

```
In [112]: data[['Vermont', 'Nevada', 'Idaho']] = np.nan
```

```
In [113]: data
```

```
Out[113]:
```

```
Ohio        0.922264  
New York   -2.153545  
Vermont     NaN  
Florida    -0.375842  
Oregon     0.329939  
Nevada      NaN  
California  1.105913  
Idaho       NaN
```

```
In [114]: data.groupby(group_key).mean()
```

```
Out[114]:
```

```
East    -0.535707
West     0.717926
```

We can fill the NA values using the group means like so:

```
In [115]: fill_mean = lambda g: g.fillna(g.mean())
```

```
In [116]: data.groupby(group_key).apply(fill_mean)
```

```
Out[116]:
```

```
Ohio        0.922264
New York   -2.153545
Vermont     -0.535707
Florida     -0.375842
Oregon      0.329939
Nevada      0.717926
California  1.105913
Idaho       0.717926
```

In another case, you might have pre-defined fill values in your code that vary by group. Since the groups have a `name` attribute set internally, we can use that:

```
In [117]: fill_values = {'East': 0.5, 'West': -1}
```

```
In [118]: fill_func = lambda g: g.fillna(fill_values[g.name])
```

```
In [119]: data.groupby(group_key).apply(fill_func)
```

```
Out[119]:
```

```
Ohio        0.922264
New York   -2.153545
Vermont     0.500000
Florida     -0.375842
Oregon      0.329939
Nevada      -1.000000
California  1.105913
Idaho       -1.000000
```

## Example: Random Sampling and Permutation

Suppose you wanted to draw a random sample (with or without replacement) from a large dataset for Monte Carlo simulation purposes or some other application. There are a number of ways to perform the “draws”; some are much more efficient than others. One way is to select the first  $K$  elements of `np.random.permutation(N)`, where  $N$  is the size of your complete dataset and  $K$  the desired sample size. As a more fun example, here’s a way to construct a deck of English-style playing cards:

```
# Hearts, Spades, Clubs, Diamonds
suits = ['H', 'S', 'C', 'D']
card_val = (range(1, 11) + [10] * 3) * 4
base_names = ['A'] + range(2, 11) + ['J', 'K', 'Q']
cards = []
for suit in ['H', 'S', 'C', 'D']:
    cards.extend(str(num) + suit for num in base_names)

deck = Series(card_val, index=cards)
```

So now we have a Series of length 52 whose index contains card names and values are the ones used in blackjack and other games (to keep things simple, I just let the ace be 1):

```
In [121]: deck[:13]
Out[121]:
AH      1
2H      2
3H      3
4H      4
5H      5
6H      6
7H      7
8H      8
9H      9
10H     10
JH      10
KH      10
QH      10
```

Now, based on what I said above, drawing a hand of 5 cards from the desk could be written as:

```
In [122]: def draw(deck, n=5):
.....:     return deck.take(np.random.permutation(len(deck))[:n])

In [123]: draw(deck)
Out[123]:
AD      1
8C      8
5H      5
KC      10
2C      2
```

Suppose you wanted two random cards from each suit. Because the suit is the last character of each card name, we can group based on this and use `apply`:

```
In [124]: get_suit = lambda card: card[-1] # last letter is suit

In [125]: deck.groupby(get_suit).apply(draw, n=2)
Out[125]:
C  2C      2
    3C      3
D  KD      10
    8D      8
H  KH      10
    3H      3
S  2S      2
    4S      4

# alternatively
In [126]: deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
Out[126]:
KC      10
JC      10
AD      1
```

```
5D      5
5H      5
6H      6
7S      7
KS     10
```

## Example: Group Weighted Average and Correlation

Under the split-apply-combine paradigm of `groupby`, operations between columns in a DataFrame or two Series, such a group weighted average, become a routine affair. As an example, take this dataset containing group keys, values, and some weights:

```
In [127]: df = DataFrame({'category': ['a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'],
.....:                  'data': np.random.randn(8),
.....:                  'weights': np.random.rand(8)})

In [128]: df
Out[128]:
   category    data   weights
0         a  1.561587  0.957515
1         a  1.219984  0.347267
2         a -0.482239  0.581362
3         a  0.315667  0.217091
4         b -0.047852  0.894406
5         b -0.454145  0.918564
6         b -0.556774  0.277825
7         b  0.253321  0.955905
```

The group weighted average by `category` would then be:

```
In [129]: grouped = df.groupby('category')

In [130]: get_wavg = lambda g: np.average(g['data'], weights=g['weights'])

In [131]: grouped.apply(get_wavg)
Out[131]:
category
a          0.811643
b         -0.122262
```

As a less trivial example, consider a data set from Yahoo! Finance containing end of day prices for a few stocks and the S&P 500 index (the SPX ticker):

```
In [132]: close_px = pd.read_csv('ch09/stock_px.csv', parse_dates=True, index_col=0)

In [133]: close_px
Out[133]:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 00:00:00 to 2011-10-14 00:00:00
Data columns:
AAPL    2214 non-null values
MSFT    2214 non-null values
XOM    2214 non-null values
SPX    2214 non-null values
dtypes: float64(4)
```

```
In [134]: close_px[-4:]
Out[134]:
          AAPL    MSFT     XOM      SPX
2011-10-11  400.29  27.00  76.27  1195.54
2011-10-12  402.19  26.96  77.16  1207.25
2011-10-13  408.43  27.18  76.37  1203.66
2011-10-14  422.00  27.27  78.11  1224.58
```

One task of interest might be to compute a DataFrame consisting of the yearly correlations of daily returns (computed from percent changes) with SPX. Here is one way to do it:

```
In [135]: rets = close_px.pct_change().dropna()

In [136]: spx_corr = lambda x: x.corrwith(x['SPX'])

In [137]: by_year = rets.groupby(lambda x: x.year)

In [138]: by_year.apply(spx_corr)
Out[138]:
          AAPL      MSFT      XOM      SPX
2003  0.541124  0.745174  0.661265      1
2004  0.374283  0.588531  0.557742      1
2005  0.467540  0.562374  0.631010      1
2006  0.428267  0.406126  0.518514      1
2007  0.508118  0.658770  0.786264      1
2008  0.681434  0.804626  0.828303      1
2009  0.707103  0.654902  0.797921      1
2010  0.710105  0.730118  0.839057      1
2011  0.691931  0.800996  0.859975      1
```

There is, of course, nothing to stop you from computing inter-column correlations:

```
# Annual correlation of Apple with Microsoft
In [139]: by_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))
Out[139]:
2003    0.480868
2004    0.259024
2005    0.300093
2006    0.161735
2007    0.417738
2008    0.611901
2009    0.432738
2010    0.571946
2011    0.581987
```

## Example: Group-wise Linear Regression

In the same vein as the previous example, you can use `groupby` to perform more complex group-wise statistical analysis, as long as the function returns a pandas object or scalar value. For example, I can define the following `regress` function (using the `statsmodels` econometrics library) which executes an ordinary least squares (OLS) regression on each chunk of data:

```

import statsmodels.api as sm
def regress(data, yvar, xvars):
    Y = data[yvar]
    X = data[xvars]
    X['intercept'] = 1.
    result = sm.OLS(Y, X).fit()
    return result.params

```

Now, to run a yearly linear regression of AAPL on SPX returns, I execute:

```

In [141]: by_year.apply(regress, 'AAPL', ['SPX'])
Out[141]:
      SPX  intercept
2003  1.195406  0.000710
2004  1.363463  0.004201
2005  1.766415  0.003246
2006  1.645496  0.000080
2007  1.198761  0.003438
2008  0.968016  -0.001110
2009  0.879103  0.002954
2010  1.052608  0.001261
2011  0.806605  0.001514

```

## Pivot Tables and Cross-Tabulation

A *pivot table* is a data summarization tool frequently found in spreadsheet programs and other data analysis software. It aggregates a table of data by one or more keys, arranging the data in a rectangle with some of the group keys along the rows and some along the columns. Pivot tables in Python with pandas are made possible using the `groupby` facility described in this chapter combined with reshape operations utilizing hierarchical indexing. DataFrame has a `pivot_table` method, and additionally there is a top-level `pandas.pivot_table` function. In addition to providing a convenience interface to `groupby`, `pivot_table` also can add partial totals, also known as *margins*.

Returning to the tipping data set, suppose I wanted to compute a table of group means (the default `pivot_table` aggregation type) arranged by `sex` and `smoker` on the rows:

```

In [142]: tips.pivot_table(rows=['sex', 'smoker'])
Out[142]:
           size      tip  tip_pct  total_bill
sex   smoker
Female No        2.592593  2.773519  0.156921  18.105185
      Yes       2.242424  2.931515  0.182150  17.977879
Male   No        2.711340  3.113402  0.160669  19.791237
      Yes       2.500000  3.051167  0.152771  22.284500

```

This could have been easily produced using `groupby`. Now, suppose we want to aggregate only `tip_pct` and `size`, and additionally group by day. I'll put `smoker` in the table columns and day in the rows:

```

In [143]: tips.pivot_table(['tip_pct', 'size'], rows=['sex', 'day'],
.....:                   cols='smoker')
Out[143]:

```

		tip_pct		size	
smoker		No	Yes	No	Yes
sex	day				
Female	Fri	0.165296	0.209129	2.500000	2.000000
	Sat	0.147993	0.163817	2.307692	2.200000
	Sun	0.165710	0.237075	3.071429	2.500000
	Thur	0.155971	0.163073	2.480000	2.428571
Male	Fri	0.138005	0.144730	2.000000	2.125000
	Sat	0.162132	0.139067	2.656250	2.629630
	Sun	0.158291	0.173964	2.883721	2.600000
	Thur	0.165706	0.164417	2.500000	2.300000

This table could be augmented to include partial totals by passing `margins=True`. This has the effect of adding All row and column labels, with corresponding values being the group statistics for all the data within a single tier. In this below example, the All values are means without taking into account smoker vs. non-smoker (the All columns) or any of the two levels of grouping on the rows (the All row):

```
In [144]: tips.pivot_table(['tip_pct', 'size'], rows=['sex', 'day'],
.....:                 cols='smoker', margins=True)
Out[144]:
```

		size		tip_pct			
smoker		No	Yes	All	No	Yes	All
sex	day						
Female	Fri	2.500000	2.000000	2.111111	0.165296	0.209129	0.199388
	Sat	2.307692	2.200000	2.250000	0.147993	0.163817	0.156470
	Sun	3.071429	2.500000	2.944444	0.165710	0.237075	0.181569
	Thur	2.480000	2.428571	2.468750	0.155971	0.163073	0.157525
Male	Fri	2.000000	2.125000	2.100000	0.138005	0.144730	0.143385
	Sat	2.656250	2.629630	2.644068	0.162132	0.139067	0.151577
	Sun	2.883721	2.600000	2.810345	0.158291	0.173964	0.162344
	Thur	2.500000	2.300000	2.433333	0.165706	0.164417	0.165276
All		2.668874	2.408602	2.569672	0.159328	0.163196	0.160803

To use a different aggregation function, pass it to `aggfunc`. For example, 'count' or `len` will give you a cross-tabulation (count or frequency) of group sizes:

```
In [145]: tips.pivot_table('tip_pct', rows=['sex', 'smoker'], cols='day',
.....:                 aggfunc=len, margins=True)
Out[145]:
```

	Fri	Sat	Sun	Thur	All	
day						
sex	smoker					
Female	No	2	13	14	25	54
	Yes	7	15	4	7	33
Male	No	2	32	43	20	97
	Yes	8	27	15	10	60
All		19	87	76	62	244

If some combinations are empty (or otherwise NA), you may wish to pass a `fill_value`:

```
In [146]: tips.pivot_table('size', rows=['time', 'sex', 'smoker'],
.....:                 cols='day', aggfunc='sum', fill_value=0)
Out[146]:
```

day	Fri	Sat	Sun	Thur		
time	sex	smoker				
Dinner	Female	No	2	30	43	2

		Yes	8	33	10	0
	Male	No	4	85	124	0
		Yes	12	71	39	0
Lunch	Female	No	3	0	0	60
		Yes	6	0	0	17
	Male	No	0	0	0	50
		Yes	5	0	0	23

See [Table 9-2](#) for a summary of `pivot_table` methods.

*Table 9-2. pivot\_table options*

Function name	Description
values	Column name or names to aggregate. By default aggregates all numeric columns
rows	Column names or other group keys to group on the rows of the resulting pivot table
cols	Column names or other group keys to group on the columns of the resulting pivot table
aggfunc	Aggregation function or list of functions; 'mean' by default. Can be any function valid in a groupby context
fill_value	Replace missing values in result table
margins	Add row/column subtotals and grand total, False by default

## Cross-Tabulations: Crosstab

A cross-tabulation (or *crosstab* for short) is a special case of a pivot table that computes group frequencies. Here is a canonical example taken from the Wikipedia page on cross-tabulation:

```
In [150]: data
Out[150]:
      Sample  Gender  Handedness
0        1  Female  Right-handed
1        2   Male   Left-handed
2        3  Female  Right-handed
3        4   Male  Right-handed
4        5   Male  Left-handed
5        6   Male  Right-handed
6        7  Female  Right-handed
7        8  Female  Left-handed
8        9   Male  Right-handed
9       10  Female  Right-handed
```

As part of some survey analysis, we might want to summarize this data by gender and handedness. You could use `pivot_table` to do this, but the `pandas.crosstab` function is very convenient:

```
In [151]: pd.crosstab(data.Gender, data.Handedness, margins=True)
Out[151]:
Handedness  Left-handed  Right-handed  All
Gender
Female            1            4            5
Male              2            3            5
All               3            7            10
```

The first two arguments to `crosstab` can each either be an array or Series or a list of arrays. As in the tips data:

```
In [152]: pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)
Out[152]:
smoker      No  Yes  All
time   day
Dinner  Fri    3    9   12
        Sat   45   42   87
        Sun   57   19   76
        Thur   1    0    1
Lunch   Fri    1    6    7
        Thur  44   17   61
All       151   93  244
```

## Example: 2012 Federal Election Commission Database

The US Federal Election Commission publishes data on contributions to political campaigns. This includes contributor names, occupation and employer, address, and contribution amount. An interesting dataset is from the 2012 US presidential election (<http://www.fec.gov/disclosurep/PDownload.do>). As of this writing (June 2012), the full dataset for all states is a 150 megabyte CSV file `P00000001-ALL.csv`, which can be loaded with `pandas.read_csv`:

```
In [13]: fec = pd.read_csv('ch09/P00000001-ALL.csv')

In [14]: fec
Out[14]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1001731 entries, 0 to 1001730
Data columns:
cmte_id           1001731 non-null values
cand_id           1001731 non-null values
cand_nm           1001731 non-null values
contbr_nm          1001731 non-null values
contbr_city         1001716 non-null values
contbr_st           1001727 non-null values
contbr_zip          1001620 non-null values
contbr_employer     994314 non-null values
contbr_occupation   994433 non-null values
contb_receipt_amt   1001731 non-null values
contb_receipt_dt    1001731 non-null values
receipt_desc        14166 non-null values
memo_cd             92482 non-null values
memo_text            97770 non-null values
form_tp              1001731 non-null values
file_num             1001731 non-null values
dtypes: float64(1), int64(1), object(14)
```

A sample record in the DataFrame looks like this:

```
In [15]: fec.ix[123456]
Out[15]:
cmte_id          C00431445
```

```

cand_id          P80003338
cand_nm         Obama, Barack
contbr_nm        ELLMAN, IRA
contbr_city      TEMPE
contbr_st         AZ
contbr_zip       852816719
contbr_employer   ARIZONA STATE UNIVERSITY
contbr_occupation PROFESSOR
contb_receipt_amt    50
contb_receipt_dt   01-DEC-11
receipt_desc      NaN
memo_cd           NaN
memo_text          NaN
form_tp            SA17A
file_num           772372
Name: 123456

```

You can probably think of many ways to start slicing and dicing this data to extract informative statistics about donors and patterns in the campaign contributions. I'll spend the next several pages showing you a number of different analyses that apply techniques you have learned about so far.

You can see that there are no political party affiliations in the data, so this would be useful to add. You can get a list of all the unique political candidates using `unique` (note that NumPy suppresses the quotes around the strings in the output):

```

In [16]: unique_cands = fec.cand_nm.unique()

In [17]: unique_cands
Out[17]:
array(['Bachmann, Michelle', 'Romney, Mitt', 'Obama, Barack',
       'Roemer, Charles E. "Buddy" III', 'Pawlenty, Timothy',
       'Johnson, Gary Earl', 'Paul, Ron', 'Santorum, Rick', 'Cain, Herman',
       'Gingrich, Newt', 'McCotter, Thaddeus G', 'Huntsman, Jon', 'Perry, Rick'],
      dtype=object)

In [18]: unique_cands[2]
Out[18]: 'Obama, Barack'

```

An easy way to indicate party affiliation is using a dict:<sup>2</sup>

```

parties = {'Bachmann, Michelle': 'Republican',
           'Cain, Herman': 'Republican',
           'Gingrich, Newt': 'Republican',
           'Huntsman, Jon': 'Republican',
           'Johnson, Gary Earl': 'Republican',
           'McCotter, Thaddeus G': 'Republican',
           'Obama, Barack': 'Democrat',
           'Paul, Ron': 'Republican',
           'Pawlenty, Timothy': 'Republican',
           'Perry, Rick': 'Republican',
           "Roemer, Charles E. 'Buddy' III": 'Republican',

```

2. This makes the simplifying assumption that Gary Johnson is a Republican even though he later became the Libertarian party candidate.

```
'Romney, Mitt': 'Republican',
'Santorum, Rick': 'Republican'}
```

Now, using this mapping and the `map` method on Series objects, you can compute an array of political parties from the candidate names:

```
In [20]: fec.cand_nm[123456:123461]
Out[20]:
123456    Obama, Barack
123457    Obama, Barack
123458    Obama, Barack
123459    Obama, Barack
123460    Obama, Barack
Name: cand_nm

In [21]: fec.cand_nm[123456:123461].map(parties)
Out[21]:
123456    Democrat
123457    Democrat
123458    Democrat
123459    Democrat
123460    Democrat
Name: cand_nm

# Add it as a column
In [22]: fec['party'] = fec.cand_nm.map(parties)

In [23]: fec['party'].value_counts()
Out[23]:
Democrat      593746
Republican    407985
```

A couple of data preparation points. First, this data includes both contributions and refunds (negative contribution amount):

```
In [24]: (fec.contb_receipt_amt > 0).value_counts()
Out[24]:
True     991475
False    10256
```

To simplify the analysis, I'll restrict the data set to positive contributions:

```
In [25]: fec = fec[fec.contb_receipt_amt > 0]
```

Since Barack Obama and Mitt Romney are the main two candidates, I'll also prepare a subset that just has contributions to their campaigns:

```
In [26]: fec_mrbo = fec[fec.cand_nm.isin(['Obama, Barack', 'Romney, Mitt'])]
```

## Donation Statistics by Occupation and Employer

Donations by occupation is another oft-studied statistic. For example, lawyers (attorneys) tend to donate more money to Democrats, while business executives tend to donate more to Republicans. You have no reason to believe me; you can see for yourself in the data. First, the total number of donations by occupation is easy:

```
In [27]: fec.contbr_occupation.value_counts()[:10]
Out[27]:
RETIRED                      233990
INFORMATION REQUESTED          35107
ATTORNEY                       34286
HOMEMAKER                      29931
PHYSICIAN                      23432
INFORMATION REQUESTED PER BEST EFFORTS  21138
ENGINEER                        14334
TEACHER                         13990
CONSULTANT                      13273
PROFESSOR                       12555
```

You will notice by looking at the occupations that many refer to the same basic job type, or there are several variants of the same thing. Here is a code snippet illustrates a technique for cleaning up a few of them by mapping from one occupation to another; note the “trick” of using `dict.get` to allow occupations with no mapping to “pass through”:

```
occ_mapping = {
    'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
    'INFORMATION REQUESTED' : 'NOT PROVIDED',
    'INFORMATION REQUESTED (BEST EFFORTS)' : 'NOT PROVIDED',
    'C.E.O.' : 'CEO'
}

# If no mapping provided, return x
f = lambda x: occ_mapping.get(x, x)
fec.contbr_occupation = fec.contbr_occupation.map(f)
```

I'll also do the same thing for employers:

```
emp_mapping = {
    'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
    'INFORMATION REQUESTED' : 'NOT PROVIDED',
    'SELF' : 'SELF-EMPLOYED',
    'SELF EMPLOYED' : 'SELF-EMPLOYED',
}

# If no mapping provided, return x
f = lambda x: emp_mapping.get(x, x)
fec.contbr_employer = fec.contbr_employer.map(f)
```

Now, you can use `pivot_table` to aggregate the data by party and occupation, then filter down to the subset that donated at least \$2 million overall:

```
In [34]: by_occupation = fec.pivot_table('contb_receipt_amt',
.....:                                         rows='contbr_occupation',
.....:                                         cols='party', aggfunc='sum')

In [35]: over_2mm = by_occupation[by_occupation.sum(1) > 2000000]

In [36]: over_2mm
Out[36]:
party           Democrat      Republican
contbr_occupation
```

ATTORNEY	11141982.97	7477194.430000
CEO	2074974.79	4211040.520000
CONSULTANT	2459912.71	2544725.450000
ENGINEER	951525.55	1818373.700000
EXECUTIVE	1355161.05	4138850.090000
HOMEMAKER	4248875.80	13634275.780000
INVESTOR	884133.00	2431768.920000
LAWYER	3160478.87	391224.320000
MANAGER	762883.22	1444532.370000
NOT PROVIDED	4866973.96	20565473.010000
OWNER	1001567.36	2408286.920000
PHYSICIAN	3735124.94	3594320.240000
PRESIDENT	1878509.95	4720923.760000
PROFESSOR	2165071.08	296702.730000
REAL ESTATE	528902.09	1625902.250000
RETIRED	25305116.38	23561244.489999
SELF-EMPLOYED	672393.40	1640252.540000

It can be easier to look at this data graphically as a bar plot ('barh' means horizontal bar plot, see [Figure 9-2](#)):

In [38]: `over_2mm.plot(kind='barh')`

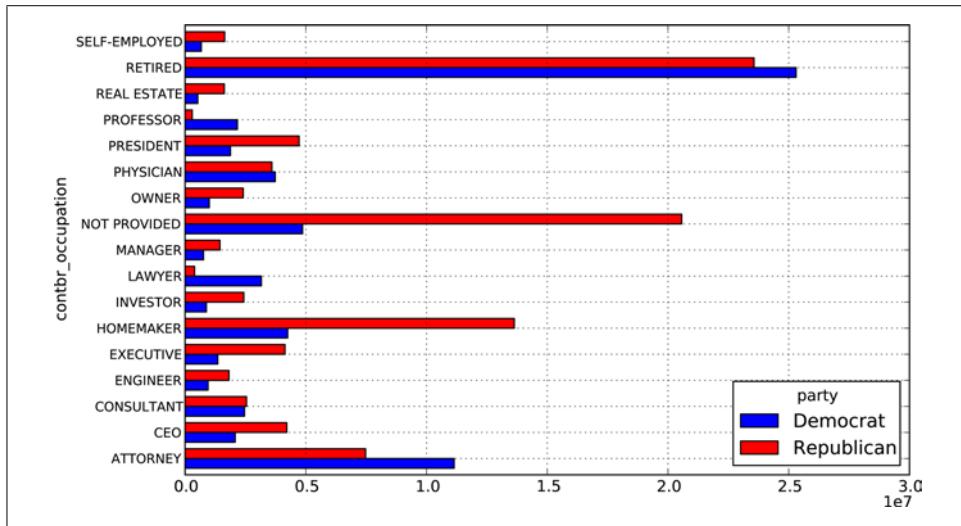


Figure 9-2. Total donations by party for top occupations

You might be interested in the top donor occupations or top companies donating to Obama and Romney. To do this, you can group by candidate name and use a variant of the `top` method from earlier in the chapter:

```
def get_top_amounts(group, key, n=5):
    totals = group.groupby(key)['contb_receipt_amt'].sum()

    # Order totals by key in descending order
    return totals.order(ascending=False)[-n:]
```

Then aggregated by occupation and employer:

```
In [40]: grouped = fec_mrbo.groupby('cand_nm')

In [41]: grouped.apply(get_top_amounts, 'contbr_occupation', n=7)
Out[41]:
cand_nm      contbr_occupation
Obama, Barack    RETIRED          25305116.38
                  ATTORNEY        11141982.97
                  NOT PROVIDED     4866973.96
                  HOMEMAKER        4248875.80
                  PHYSICIAN         3735124.94
                  LAWYER            3160478.87
                  CONSULTANT        2459912.71
Romney, Mitt     RETIRED          11508473.59
                  NOT PROVIDED     11396894.84
                  HOMEMAKER        8147446.22
                  ATTORNEY          5364718.82
                  PRESIDENT         2491244.89
                  EXECUTIVE         2300947.03
                  C.E.O.            1968386.11
Name: contb_receipt_amt

In [42]: grouped.apply(get_top_amounts, 'contbr_employer', n=10)
Out[42]:
cand_nm      contbr_employer
Obama, Barack    RETIRED          22694358.85
                  SELF-EMPLOYED   18626807.16
                  NOT EMPLOYED    8586308.70
                  NOT PROVIDED    5053480.37
                  HOMEMAKER        2605408.54
                  STUDENT          318831.45
                  VOLUNTEER         257104.00
                  MICROSOFT         215585.36
                  SIDLEY AUSTIN LLP 168254.00
                  REFUSED           149516.07
Romney, Mitt     NOT PROVIDED    12059527.24
                  RETIRED           11506225.71
                  HOMEMAKER         8147196.22
                  SELF-EMPLOYED    7414115.22
                  STUDENT           496490.94
                  CREDIT SUISSE       281150.00
                  MORGAN STANLEY    267266.00
                  GOLDMAN SACH & CO. 238250.00
                  BARCLAYS CAPITAL   162750.00
                  H.I.G. CAPITAL     139500.00
Name: contb_receipt_amt
```

## Bucketing Donation Amounts

A useful way to analyze this data is to use the `cut` function to discretize the contributor amounts into buckets by contribution size:

```
In [43]: bins = np.array([0, 1, 10, 100, 1000, 10000, 100000, 1000000, 10000000])
```

```
In [44]: labels = pd.cut(fec_mrbo.contb_receipt_amt, bins)

In [45]: labels
Out[45]:
Categorical:contb_receipt_amt
array([(10, 100], (100, 1000], (100, 1000], ..., (1, 10], (10, 100],
      (100, 1000]], dtype=object)
Levels (8): array([(0, 1], (1, 10], (10, 100], (100, 1000], (1000, 10000],
                  (10000, 100000], (100000, 1000000], (1000000, 10000000]], dtype=object)
```

We can then group the data for Obama and Romney by name and bin label to get a histogram by donation size:

```
In [46]: grouped = fec_mrbo.groupby(['cand_nm', labels])

In [47]: grouped.size().unstack(0)
Out[47]:
cand_nm          Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]              493          77
(1, 10]             40070        3681
(10, 100]            372280       31853
(100, 1000]           153991       43357
(1000, 10000]          22284       26186
(10000, 100000]         2           1
(100000, 1000000]        3           NaN
(1000000, 10000000]       4           NaN
```

This data shows that Obama has received a significantly larger number of small donations than Romney. You can also sum the contribution amounts and normalize within buckets to visualize percentage of total donations of each size by candidate:

```
In [48]: bucket_sums = grouped.contb_receipt_amt.sum().unstack(0)

In [49]: bucket_sums
Out[49]:
cand_nm          Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]              318.24        77.00
(1, 10]             337267.62      29819.66
(10, 100]            20288981.41     1987783.76
(100, 1000]           54798531.46     22363381.69
(1000, 10000]          51753705.67    63942145.42
(10000, 100000]         59100.00       12700.00
(100000, 1000000]        1490683.08       NaN
(1000000, 10000000]       7148839.76       NaN
```

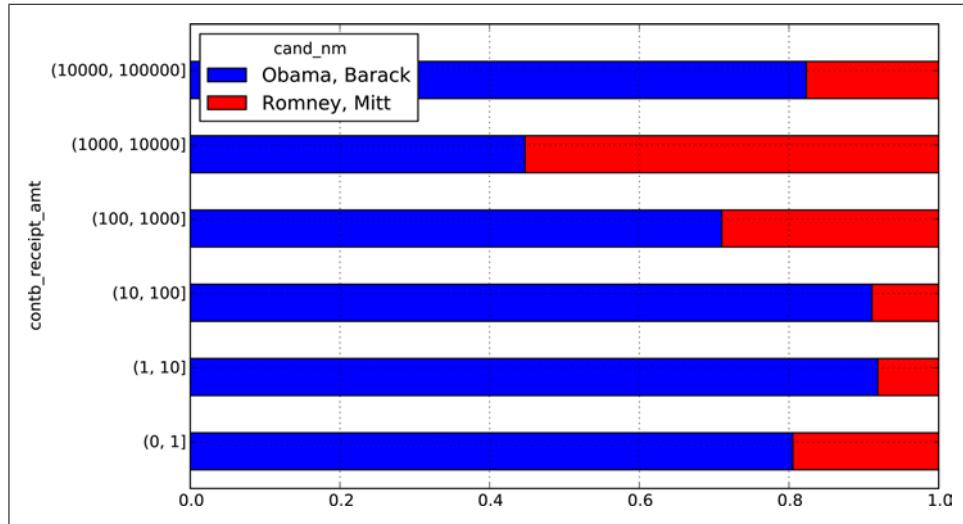
```
In [50]: normed_sums = bucket_sums.div(bucket_sums.sum(axis=1), axis=0)
```

```
In [51]: normed_sums
Out[51]:
cand_nm          Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]              0.805182      0.194818
(1, 10]             0.918767      0.081233
(10, 100]            0.910769      0.089231
```

(100, 1000]	0.710176	0.289824
(1000, 10000]	0.447326	0.552674
(10000, 100000]	0.823120	0.176880
(100000, 1000000]	1.000000	NaN
(1000000, 10000000]	1.000000	NaN

```
In [52]: normed_sums[:-2].plot(kind='barh', stacked=True)
```

I excluded the two largest bins as these are not donations by individuals. See [Figure 9-3](#) for the resulting figure.



*Figure 9-3. Percentage of total donations received by candidates for each donation size*

There are of course many refinements and improvements of this analysis. For example, you could aggregate donations by donor name and zip code to adjust for donors who gave many small amounts versus one or more large donations. I encourage you to download it and explore it yourself.

## Donation Statistics by State

Aggregating the data by candidate and state is a routine affair:

```
In [53]: grouped = fec_mrbo.groupby(['cand_nm', 'contbr_st'])
```

```
In [54]: totals = grouped.contb_receipt_amt.sum().unstack(0).fillna(0)
```

```
In [55]: totals = totals[totals.sum(1) > 100000]
```

```
In [56]: totals[:10]
```

```
Out[56]:
```

cand_nm	Obama, Barack	Romney, Mitt
contbr_st		

AK	281840.15	86204.24
AL	543123.48	527303.51
AR	359247.28	105556.00
AZ	1506476.98	1888436.23
CA	23824984.24	11237636.60
CO	2132429.49	1506714.12
CT	2068291.26	3499475.45
DC	4373538.80	1025137.50
DE	336669.14	82712.00
FL	7318178.58	8338458.81

If you divide each row by the total contribution amount, you get the relative percentage of total donations by state for each candidate:

```
In [57]: percent = totals.div(totals.sum(1), axis=0)
```

```
In [58]: percent[:10]
```

```
Out[58]:
```

cand_nm	Obama, Barack	Romney, Mitt
contbr_st		
AK	0.765778	0.234222
AL	0.507390	0.492610
AR	0.772902	0.227098
AZ	0.443745	0.556255
CA	0.679498	0.320502
CO	0.585970	0.414030
CT	0.371476	0.628524
DC	0.810113	0.189887
DE	0.802776	0.197224
FL	0.467417	0.532583

I thought it would be interesting to look at this data plotted on a map, using ideas from [Chapter 8](#). After locating a shape file for the state boundaries (<http://nationalatlas.gov/atlasftp.html?openChapters=chpbound>) and learning a bit more about matplotlib and its basemap toolkit (I was aided by a blog posting from Thomas Lecocq)<sup>3</sup>, I ended up with the following code for plotting these relative percentages:

```
from mpl_toolkits.basemap import Basemap, cm
import numpy as np
from matplotlib import rcParams
from matplotlib.collections import LineCollection
import matplotlib.pyplot as plt

from shapelib import ShapeFile
import dbflib

obama = percent['Obama, Barack']

fig = plt.figure(figsize=(12, 12))
ax = fig.add_axes([0.1,0.1,0.8,0.8])

lllat = 21; urlat = 53; lllon = -118; urlon = -62
```

3. <http://www.geophysique.be/2011/01/27/matplotlib-basemap-tutorial-07-shapefiles-unleached/>

```

m = Basemap(ax=ax, projection='stere',
            lon_0=(urlon + llon) / 2, lat_0=(urlat + lllat) / 2,
            llcrnrlat=lllat, urcrnrlat=urlat, llcrnrlon=lllon,
            urcrnrlon=urlon, resolution='l')
m.drawcoastlines()
m.drawcountries()

shp = ShapeFile('../states/statesp020')
dbf = dbflib.open('../states/statesp020')

for npoly in range(shp.info()[0]):
    # Draw colored polygons on the map
    shpsegs = []
    shp_object = shp.read_object(npoly)
    verts = shp_object.vertices()
    rings = len(verts)
    for ring in range(rings):
        lons, lats = zip(*verts[ring])
        x, y = m(lons, lats)
        shpsegs.append(zip(x,y))
    if ring == 0:
        shapedict = dbf.read_record(npoly)
        name = shapedict['STATE']
    lines = LineCollection(shpsegs, antialiaseds=(1,))

    # state_to_code dict, e.g. 'ALASKA' -> 'AK', omitted
    try:
        per = obama[state_to_code[name.upper()]]
    except KeyError:
        continue

    lines.set_facecolors('k')
    lines.set_alpha(0.75 * per) # Shrink the percentage a bit
    lines.set_edgecolors('k')
    lines.set_linewidth(0.3)
    ax.add_collection(lines)

plt.show()

```

See [Figure 9-4](#) for the result.

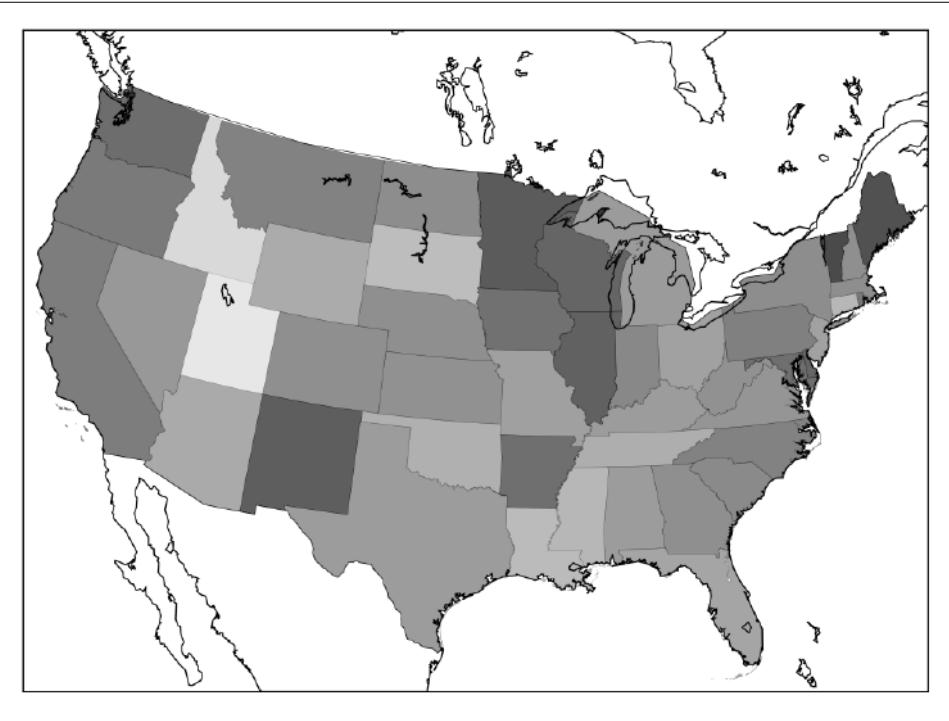


Figure 9-4. US map aggregated donation statistics overlay (darker means more Democratic)

# NumPy Basics: Arrays and Vectorized Computation

NumPy, short for Numerical Python, is the fundamental package required for high performance scientific computing and data analysis. It is the foundation on which nearly all of the higher-level tools in this book are built. Here are some of the things it provides:

- `ndarray`, a fast and space-efficient multidimensional array providing vectorized arithmetic operations and sophisticated *broadcasting* capabilities
- Standard mathematical functions for fast operations on entire arrays of data without having to write loops
- Tools for reading / writing array data to disk and working with memory-mapped files
- Linear algebra, random number generation, and Fourier transform capabilities
- Tools for integrating code written in C, C++, and Fortran

The last bullet point is also one of the most important ones from an ecosystem point of view. Because NumPy provides an easy-to-use C API, it is very easy to pass data to external libraries written in a low-level language and also for external libraries to return data to Python as NumPy arrays. This feature has made Python a language of choice for wrapping legacy C/C++/Fortran codebases and giving them a dynamic and easy-to-use interface.

While NumPy by itself does not provide very much high-level data analytical functionality, having an understanding of NumPy arrays and array-oriented computing will help you use tools like pandas much more effectively. If you’re new to Python and just looking to get your hands dirty working with data using pandas, feel free to give this chapter a skim. For more on advanced NumPy features like broadcasting, see [Chapter 12](#).

For most data analysis applications, the main areas of functionality I'll focus on are:

- Fast vectorized array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Data alignment and relational data manipulations for merging and joining together heterogeneous data sets
- Expressing conditional logic as array expressions instead of loops with `if-elif-else` branches
- Group-wise data manipulations (aggregation, transformation, function application). Much more on this in [Chapter 5](#)

While NumPy provides the computational foundation for these operations, you will likely want to use pandas as your basis for most kinds of data analysis (especially for structured or tabular data) as it provides a rich, high-level interface making most common data tasks very concise and simple. pandas also provides some more domain-specific functionality like time series manipulation, which is not present in NumPy.



In this chapter and throughout the book, I use the standard NumPy convention of always using `import numpy as np`. You are, of course, welcome to put `from numpy import *` in your code to avoid having to write `np`, but I would caution you against making a habit of this.

## The NumPy ndarray: A Multidimensional Array Object

One of the key features of NumPy is its N-dimensional array object, or `ndarray`, which is a fast, flexible container for large data sets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements:

```
In [8]: data  
Out[8]:  
array([[ 0.9526, -0.246 , -0.8856],  
       [ 0.5639,  0.2379,  0.9104]])
```

```
In [9]: data * 10  
Out[9]:  
array([[ 9.5256, -2.4601, -8.8565],  
       [ 5.6385,  2.3794,  9.104 ]])
```

```
In [10]: data + data  
Out[10]:  
array([[ 1.9051, -0.492 , -1.7713],  
       [ 1.1277,  0.4759,  1.8208]])
```

An `ndarray` is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same type. Every array has a `shape`, a tuple indicating the size of each dimension, and a `dtype`, an object describing the *data type* of the array:

```
In [11]: data.shape  
Out[11]: (2, 3)
```

```
In [12]: data.dtype  
Out[12]: dtype('float64')
```

This chapter will introduce you to the basics of using NumPy arrays, and should be sufficient for following along with the rest of the book. While it's not necessary to have a deep understanding of NumPy for many data analytical applications, becoming proficient in array-oriented programming and thinking is a key step along the way to becoming a scientific Python guru.



Whenever you see “array”, “NumPy array”, or “ndarray” in the text, with few exceptions they all refer to the same thing: the ndarray object.

## Creating ndarrays

The easiest way to create an array is to use the `array` function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data. For example, a list is a good candidate for conversion:

```
In [13]: data1 = [6, 7.5, 8, 0, 1]  
  
In [14]: arr1 = np.array(data1)  
  
In [15]: arr1  
Out[15]: array([ 6.,  7.5,  8.,  0.,  1.])
```

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

```
In [16]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]  
  
In [17]: arr2 = np.array(data2)  
  
In [18]: arr2  
Out[18]:  
array([[1, 2, 3, 4],  
       [5, 6, 7, 8]])  
  
In [19]: arr2.ndim  
Out[19]: 2  
  
In [20]: arr2.shape  
Out[20]: (2, 4)
```

Unless explicitly specified (more on this later), `np.array` tries to infer a good data type for the array that it creates. The data type is stored in a special `dtype` object; for example, in the above two examples we have:

```
In [21]: arr1.dtype  
Out[21]: dtype('float64')
```

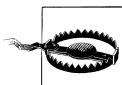
```
In [22]: arr2.dtype  
Out[22]: dtype('int64')
```

In addition to `np.array`, there are a number of other functions for creating new arrays. As examples, `zeros` and `ones` create arrays of 0's or 1's, respectively, with a given length or shape. `empty` creates an array without initializing its values to any particular value. To create a higher dimensional array with these methods, pass a tuple for the shape:

```
In [23]: np.zeros(10)  
Out[23]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

```
In [24]: np.zeros((3, 6))  
Out[24]:  
array([[ 0.,  0.,  0.,  0.,  0.,  0.],  
       [ 0.,  0.,  0.,  0.,  0.,  0.],  
       [ 0.,  0.,  0.,  0.,  0.,  0.]])
```

```
In [25]: np.empty((2, 3, 2))  
Out[25]:  
array([[[ 4.94065646e-324,   4.94065646e-324],  
       [ 3.87491056e-297,   2.46845796e-130],  
       [ 4.94065646e-324,   4.94065646e-324]],  
  
      [[ 1.90723115e+083,   5.73293533e-053],  
       [-2.33568637e+124,  -6.70608105e-012],  
       [ 4.42786966e+160,   1.27100354e+025]]])
```



It's not safe to assume that `np.empty` will return an array of all zeros. In many cases, as previously shown, it will return uninitialized garbage values.

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

```
In [26]: np.arange(15)  
Out[26]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

See [Table 4-1](#) for a short list of standard array creation functions. Since NumPy is focused on numerical computing, the data type, if not specified, will in many cases be `float64` (floating point).

*Table 4-1. Array creation functions*

Function	Description
<code>array</code>	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default.
<code>asarray</code>	Convert input to ndarray, but do not copy if the input is already an ndarray
<code>arange</code>	Like the built-in <code>range</code> but returns an ndarray instead of a list.
<code>ones, ones_like</code>	Produce an array of all 1's with the given shape and dtype. <code>ones_like</code> takes another array and produces a ones array of the same shape and dtype.
<code>zeros, zeros_like</code>	Like <code>ones</code> and <code>ones_like</code> but producing arrays of 0's instead

Function	Description
<code>empty, empty_like</code>	Create new arrays by allocating new memory, but do not populate with any values like ones and zeros
<code>eye, identity</code>	Create a square N x N identity matrix (1's on the diagonal and 0's elsewhere)

## Data Types for ndarrays

The *data type* or `dtype` is a special object containing the information the ndarray needs to interpret a chunk of memory as a particular type of data:

```
In [27]: arr1 = np.array([1, 2, 3], dtype=np.float64)
```

```
In [28]: arr2 = np.array([1, 2, 3], dtype=np.int32)
```

```
In [29]: arr1.dtype
```

```
Out[29]: dtype('float64')
```

```
In [30]: arr2.dtype
```

```
Out[30]: dtype('int32')
```

Dtypes are part of what make NumPy so powerful and flexible. In most cases they map directly onto an underlying machine representation, which makes it easy to read and write binary streams of data to disk and also to connect to code written in a low-level language like C or Fortran. The numerical dtypes are named the same way: a type name, like `float` or `int`, followed by a number indicating the number of bits per element. A standard double-precision floating point value (what's used under the hood in Python's `float` object) takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as `float64`. See Table 4-2 for a full listing of NumPy's supported data types.



Don't worry about memorizing the NumPy dtypes, especially if you're a new user. It's often only necessary to care about the general *kind* of data you're dealing with, whether floating point, complex, integer, boolean, string, or general Python object. When you need more control over how data are stored in memory and on disk, especially large data sets, it is good to know that you have control over the storage type.

Table 4-2. NumPy data types

Type	Type Code	Description
<code>int8, uint8</code>	<code>i1, u1</code>	Signed and unsigned 8-bit (1 byte) integer types
<code>int16, uint16</code>	<code>i2, u2</code>	Signed and unsigned 16-bit integer types
<code>int32, uint32</code>	<code>i4, u4</code>	Signed and unsigned 32-bit integer types
<code>int64, uint64</code>	<code>i8, u8</code>	Signed and unsigned 32-bit integer types
<code>float16</code>	<code>f2</code>	Half-precision floating point
<code>float32</code>	<code>f4 or f</code>	Standard single-precision floating point. Compatible with C float
<code>float64, float128</code>	<code>f8 or d</code>	Standard double-precision floating point. Compatible with C double and Python <code>float</code> object

Type	Type Code	Description
float128	f16 or g	Extended-precision floating point
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	?	Boolean type storing True and False values
object	0	Python object type
string_	S	Fixed-length string type (1 byte per character). For example, to create a string dtype with length 10, use 'S10'.
unicode_	U	Fixed-length unicode type (number of bytes platform specific). Same specification semantics as string_ (e.g. 'U10').

You can explicitly convert or *cast* an array from one dtype to another using ndarray's `astype` method:

```
In [31]: arr = np.array([1, 2, 3, 4, 5])
In [32]: arr.dtype
Out[32]: dtype('int64')
In [33]: float_arr = arr.astype(np.float64)
In [34]: float_arr.dtype
Out[34]: dtype('float64')
```

In this example, integers were cast to floating point. If I cast some floating point numbers to be of integer dtype, the decimal part will be truncated:

```
In [35]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
In [36]: arr
Out[36]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])
In [37]: arr.astype(np.int32)
Out[37]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

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

```
In [38]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
In [39]: numeric_strings.astype(float)
Out[39]: array([ 1.25, -9.6,  42. ])
```

If casting were to fail for some reason (like a string that cannot be converted to `float64`), a `TypeError` will be raised. See that I was a bit lazy and wrote `float` instead of `np.float64`; NumPy is smart enough to alias the Python types to the equivalent dtypes.

You can also use another array's dtype attribute:

```
In [40]: int_array = np.arange(10)
```

```
In [41]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
```

```
In [42]: int_array.astype(calibers.dtype)
Out[42]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

There are shorthand type code strings you can also use to refer to a dtype:

```
In [43]: empty_uint32 = np.empty(8, dtype='u4')
```

```
In [44]: empty_uint32
```

```
Out[44]:
array([      0,      0, 65904672,      0, 64856792,      0,
       39438163,      0], dtype=uint32)
```



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



It's worth keeping in mind that floating point numbers, such as those in `float64` and `float32` arrays, are only capable of approximating fractional quantities. In complex computations, you may accrue some *floating point error*, making comparisons only valid up to a certain number of decimal places.

## Operations between Arrays and Scalars

Arrays are important because they enable you to express batch operations on data without writing any `for` loops. This is usually called *vectorization*. Any arithmetic operations between equal-size arrays applies the operation elementwise:

```
In [45]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])
```

```
In [46]: arr
```

```
Out[46]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

```
In [47]: arr * arr
```

```
Out[47]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

```
In [48]: arr - arr
```

```
Out[48]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Arithmetic operations with scalars are as you would expect, propagating the value to each element:

```
In [49]: 1 / arr
```

```
Out[49]:
array([[ 1.    ,  0.5   ,  0.3333],
       [ 0.25  ,  0.2   ,  0.1667]])
```

```
In [50]: arr ** 0.5
```

```
Out[50]:
array([[ 1.    ,  1.4142,  1.7321],
       [ 2.    ,  2.2361,  2.4495]])
```

Operations between differently sized arrays is called *broadcasting* and will be discussed in more detail in [Chapter 12](#). Having a deep understanding of broadcasting is not necessary for most of this book.

## Basic Indexing and Slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
In [51]: arr = np.arange(10)

In [52]: arr
Out[52]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [53]: arr[5]
Out[53]: 5

In [54]: arr[5:8]
Out[54]: array([5, 6, 7])

In [55]: arr[5:8] = 12

In [56]: arr
Out[56]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

As you can see, if you assign a scalar value to a slice, as in `arr[5:8] = 12`, the value is propagated (or *broadcasted* henceforth) to the entire selection. An important first distinction from lists is that array slices are *views* on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array:

```
In [57]: arr_slice = arr[5:8]

In [58]: arr_slice[1] = 12345

In [59]: arr
Out[59]: array([ 0, 1, 2, 3, 4, 12, 12345, 12, 8, 9])

In [60]: arr_slice[:] = 64

In [61]: arr
Out[61]: array([ 0, 1, 2, 3, 4, 64, 64, 64, 8, 9])
```

If you are new to NumPy, you might be surprised by this, especially if they have used other array programming languages which copy data more zealously. As NumPy has been designed with large data use cases in mind, you could imagine performance and memory problems if NumPy insisted on copying data left and right.



If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array; for example `arr[5:8].copy()`.

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
In [62]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
In [63]: arr2d[2]  
Out[63]: array([7, 8, 9])
```

Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements. So these are equivalent:

```
In [64]: arr2d[0][2]  
Out[64]: 3
```

```
In [65]: arr2d[0, 2]  
Out[65]: 3
```

See [Figure 4-1](#) for an illustration of indexing on a 2D array.

		axis 1			
		0	1	2	
axis 0		0	0, 0	0, 1	0, 2
		1	1, 0	1, 1	1, 2
		2	2, 0	2, 1	2, 2

*Figure 4-1. Indexing elements in a NumPy array*

In multidimensional arrays, if you omit later indices, the returned object will be a lower-dimensional ndarray consisting of all the data along the higher dimensions. So in the  $2 \times 2 \times 3$  array `arr3d`

```
In [66]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

```
In [67]: arr3d  
Out[67]:  
array([[[ 1,  2,  3],
```

```
[ 4,  5,  6]],  
[[ 7,  8,  9],  
[10, 11, 12]]])
```

`arr3d[0]` is a  $2 \times 3$  array:

```
In [68]: arr3d[0]  
Out[68]:  
array([[1, 2, 3],  
       [4, 5, 6]])
```

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

```
In [69]: old_values = arr3d[0].copy()
```

```
In [70]: arr3d[0] = 42
```

```
In [71]: arr3d  
Out[71]:  
array([[42, 42, 42],  
       [42, 42, 42]],  
      [[ 7,  8,  9],  
       [10, 11, 12]]])
```

```
In [72]: arr3d[0] = old_values
```

```
In [73]: arr3d  
Out[73]:  
array([[ 1,  2,  3],  
       [ 4,  5,  6]],  
      [[ 7,  8,  9],  
       [10, 11, 12]]])
```

Similarly, `arr3d[1, 0]` gives you all of the values whose indices start with `(1, 0)`, forming a 1-dimensional array:

```
In [74]: arr3d[1, 0]  
Out[74]: array([7, 8, 9])
```

Note that in all of these cases where subsections of the array have been selected, the returned arrays are views.

## Indexing with slices

Like one-dimensional objects such as Python lists, ndarrays can be sliced using the familiar syntax:

```
In [75]: arr[1:6]  
Out[75]: array([ 1,  2,  3,  4, 64])
```

Higher dimensional objects give you more options as you can slice one or more axes and also mix integers. Consider the 2D array above, `arr2d`. Slicing this array is a bit different:

```
In [76]: arr2d  
Out[76]:
```

```
In [77]: arr2d[:2]  
Out[77]:
```

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

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. You can pass multiple slices just like you can pass multiple indexes:

```
In [78]: arr2d[:2, 1:]
Out[78]:
array([[2, 3],
[5, 6]])
```

When slicing like this, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower dimensional slices:

```
In [79]: arr2d[1, :2]      In [80]: arr2d[2, :1]
Out[79]: array([4, 5])    Out[80]: array([7])
```

See [Figure 4-2](#) for an illustration. Note that a colon by itself means to take the entire axis, so you can slice only higher dimensional axes by doing:

```
In [81]: arr2d[:, :1]
Out[81]:
array([[1],
[4],
[7]])
```

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

```
In [82]: arr2d[:2, 1:] = 0
```

## Boolean Indexing

Let's consider an example where we have some data in an array and an array of names with duplicates. I'm going to use here the `randn` function in `numpy.random` to generate some random normally distributed data:

```
In [83]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Will', 'Joe', 'Joe'])
In [84]: data = randn(7, 4)
In [85]: names
Out[85]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Will', 'Joe'],
      dtype='|S4')
In [86]: data
Out[86]:
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [-0.268 ,  0.5465,  0.0939, -2.0445],
       [-0.047 , -2.026 ,  0.7719,  0.3103],
       [ 2.1452,  0.8799, -0.0523,  0.0672],
       [-1.0023, -0.1698,  1.1503,  1.7289],
```

```
[ 0.1913,  0.4544,  0.4519,  0.5535],  
 [ 0.5994,  0.8174, -0.9297, -1.2564]])
```

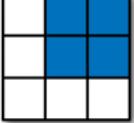
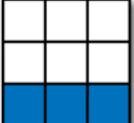
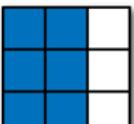
Expression	Shape
	<code>arr[::2, 1:]</code> (2, 2)
	<code>arr[2]</code> <code>arr[2, ::]</code> <code>arr[2:, ::]</code> (3,) (3,) (1, 3)
	<code>arr[:, ::2]</code> (3, 2)
	<code>arr[1, ::2]</code> <code>arr[1:2, ::2]</code> (2,) (1, 2)

Figure 4-2. Two-dimensional array slicing

Suppose each name corresponds to a row in the `data` array. If we wanted to select all the rows with corresponding name '`Bob`'. Like arithmetic operations, comparisons (such as `==`) with arrays are also vectorized. Thus, comparing `names` with the string '`Bob`' yields a boolean array:

```
In [87]: names == 'Bob'  
Out[87]: array([ True, False, False, True, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```
In [88]: data[names == 'Bob']  
Out[88]:  
array([[ -0.048 ,  0.5433, -0.2349,  1.2792],  
       [ 2.1452,  0.8799, -0.0523,  0.0672]])
```

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

```
In [89]: data[names == 'Bob', 2:]  
Out[89]:  
array([[ -0.2349,  1.2792],
```

```
[ -0.0523,  0.0672]])
```

```
In [90]: data[names == 'Bob', 3]
Out[90]: array([ 1.2792,  0.0672])
```

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

```
In [91]: names != 'Bob'
Out[91]: array([False, True, True, False, True, True, True], dtype=bool)
```

```
In [92]: data[-(names == 'Bob')]
Out[92]:
array([[-0.268 ,  0.5465,  0.0939, -2.0445],
       [-0.047 , -2.026 ,  0.7719,  0.3103],
       [-1.0023, -0.1698,  1.1503,  1.7289],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174, -0.9297, -1.2564]])
```

Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like & (and) and | (or):

```
In [93]: mask = (names == 'Bob') | (names == 'Will')
```

```
In [94]: mask
Out[94]: array([True, False, True, True, True, False], dtype=bool)
```

```
In [95]: data[mask]
Out[95]:
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [-0.047 , -2.026 ,  0.7719,  0.3103],
       [ 2.1452,  0.8799, -0.0523,  0.0672],
       [-1.0023, -0.1698,  1.1503,  1.7289]])
```

Selecting data from an array by boolean indexing *always* creates a copy of the data, even if the returned array is unchanged.



The Python keywords and and or do not work with boolean arrays.

Setting values with boolean arrays works in a common-sense way. To set all of the negative values in `data` to 0 we need only do:

```
In [96]: data[data < 0] = 0
```

```
In [97]: data
Out[97]:
array([[ 0.      ,  0.5433,  0.      ,  1.2792],
       [ 0.      ,  0.5465,  0.0939,  0.      ],
       [ 0.      ,  0.      ,  0.7719,  0.3103],
       [ 2.1452,  0.8799,  0.      ,  0.0672],
       [ 0.      ,  0.      ,  1.1503,  1.7289],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174,  0.      ,  0.      ]])
```

Setting whole rows or columns using a 1D boolean array is also easy:

```
In [98]: data[names != 'Joe'] = 7
```

```
In [99]: data
Out[99]:
array([[ 7.,  7.,  7.,  7.],
       [ 0.,  0.5465,  0.0939,  0.],
       [ 7.,  7.,  7.,  7.],
       [ 7.,  7.,  7.,  7.],
       [ 7.,  7.,  7.,  7.],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174,  0.,  0.]])
```

## Fancy Indexing

*Fancy indexing* is a term adopted by NumPy to describe indexing using integer arrays. Suppose we had a  $8 \times 4$  array:

```
In [100]: arr = np.empty((8, 4))
```

```
In [101]: for i in range(8):
.....:     arr[i] = i
```

```
In [102]: arr
Out[102]:
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.],
       [ 5.,  5.,  5.,  5.],
       [ 6.,  6.,  6.,  6.],
       [ 7.,  7.,  7.,  7.]])
```

To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

```
In [103]: arr[[4, 3, 0, 6]]
Out[103]:
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])
```

Hopefully this code did what you expected! Using negative indices select rows from the end:

```
In [104]: arr[[-3, -5, -7]]
Out[104]:
array([[ 5.,  5.,  5.,  5.],
       [ 3.,  3.,  3.,  3.],
       [ 1.,  1.,  1.,  1.]])
```

Passing multiple index arrays does something slightly different; it selects a 1D array of elements corresponding to each tuple of indices:

```
# more on reshape in Chapter 12
In [105]: arr = np.arange(32).reshape((8, 4))
```

```
In [106]: arr
Out[106]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

```
In [107]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[107]: array([ 4, 23, 29, 10])
```

Take a moment to understand what just happened: the elements (1, 0), (5, 3), (7, 1), and (2, 2) were selected. The behavior of fancy indexing in this case is a bit different from what some users might have expected (myself included), which is the rectangular region formed by selecting a subset of the matrix's rows and columns. Here is one way to get that:

```
In [108]: arr[[1, 5, 7, 2]][[:, [0, 3, 1, 2]]
Out[108]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Another way is to use the `np.ix_` function, which converts two 1D integer arrays to an indexer that selects the square region:

```
In [109]: arr[np.ix_([1, 5, 7, 2], [0, 3, 1, 2])]
Out[109]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array.

## Transposing Arrays and Swapping Axes

Transposing is a special form of reshaping which similarly returns a view on the underlying data without copying anything. Arrays have the `transpose` method and also the special `T` attribute:

```
In [110]: arr = np.arange(15).reshape((3, 5))
```

```
In [111]: arr
```

```
In [112]: arr.T
```

```

Out[111]: array([[ 0,  1,  2,  3,  4],
   [ 5,  6,  7,  8,  9],
   [10, 11, 12, 13, 14]])
Out[112]: array([[ 0,  5, 10],
   [ 1,  6, 11],
   [ 2,  7, 12],
   [ 3,  8, 13],
   [ 4,  9, 14]])

```

When doing matrix computations, you will do this very often, like for example computing the inner matrix product  $X^T X$  using `np.dot`:

```
In [113]: arr = np.random.randn(6, 3)
```

```

In [114]: np.dot(arr.T, arr)
Out[114]:
array([[ 2.584 ,  1.8753,  0.8888],
       [ 1.8753,  6.6636,  0.3884],
       [ 0.8888,  0.3884,  3.9781]])

```

For higher dimensional arrays, `transpose` will accept a tuple of axis numbers to permute the axes (for extra mind bending):

```
In [115]: arr = np.arange(16).reshape((2, 2, 4))
```

```

In [116]: arr
Out[116]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

```

```

In [117]: arr.transpose((1, 0, 2))
Out[117]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])

```

Simple transposing with `.T` is just a special case of swapping axes. `ndarray` has the method `swapaxes` which takes a pair of axis numbers:

```

In [118]: arr
Out[118]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
In [119]: arr.swapaxes(1, 2)
Out[119]:
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],
       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])

```

`swapaxes` similarly returns a view on the data without making a copy.

# Universal Functions: Fast Element-wise Array Functions

A universal function, or *ufunc*, is a function that performs elementwise operations on data in ndarrays. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results.

Many ufuncs are simple elementwise transformations, like `sqrt` or `exp`:

```
In [120]: arr = np.arange(10)

In [121]: np.sqrt(arr)
Out[121]:
array([ 0.        ,  1.        ,  1.4142   ,  1.7321   ,  2.        ,
       2.2361   ,  2.4495   ,  2.6458   ,  2.8284   ,  3.        ])

In [122]: np.exp(arr)
Out[122]:
array([ 1.        ,  2.7183   ,  7.3891   ,  20.0855  ,  54.5982  ,
       148.4132  ,  403.4288  ,  1096.6332  ,  2980.958  ,  8103.0839])
```

These are referred to as *unary* ufuncs. Others, such as `add` or `maximum`, take 2 arrays (thus, *binary* ufuncs) and return a single array as the result:

```
In [123]: x = randn(8)

In [124]: y = randn(8)

In [125]: x
Out[125]:
array([ 0.0749,  0.0974,  0.2002, -0.2551,  0.4655,  0.9222,  0.446 ,
       -0.9337])

In [126]: y
Out[126]:
array([ 0.267 , -1.1131, -0.3361,  0.6117, -1.2323,  0.4788,  0.4315,
       -0.7147])

In [127]: np.maximum(x, y) # element-wise maximum
Out[127]:
array([ 0.267 ,  0.0974,  0.2002,  0.6117,  0.4655,  0.9222,  0.446 ,
       -0.7147])
```

While not common, a ufunc can return multiple arrays. `modf` is one example, a vectorized version of the built-in Python `divmod`: it returns the fractional and integral parts of a floating point array:

```
In [128]: arr = randn(7) * 5

In [129]: np.modf(arr)
Out[129]:
(array([-0.6808,  0.0636, -0.386 ,  0.1393, -0.8806,  0.9363, -0.883 ]),
 array([-2.,  4., -3.,  5., -3.,  3., -6.]))
```

See [Table 4-3](#) and [Table 4-4](#) for a listing of available ufuncs.

*Table 4-3. Unary ufuncs*

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

*Table 4-4. Binary universal functions*

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

Function	Description
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators <code>&gt;</code> , <code>&gt;=</code> , <code>&lt;</code> , <code>&lt;=</code> , <code>==</code> , <code>!=</code>
logical_and, logical_or, logical_xor	Compute element-wise truth value of logical operation. Equivalent to infix operators & <code> </code> , <code>^</code>

## Data Processing Using Arrays

Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops. This practice of replacing explicit loops with array expressions is commonly referred to as *vectorization*. In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations. Later, in [Chapter 12](#), I will explain *broadcasting*, a powerful method for vectorizing computations.

As a simple example, suppose we wished to evaluate the function  $\sqrt{x^2 + y^2}$  across a regular grid of values. The `np.meshgrid` function takes two 1D arrays and produces two 2D matrices corresponding to all pairs of  $(x, y)$  in the two arrays:

```
In [130]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points
```

```
In [131]: xs, ys = np.meshgrid(points, points)
```

```
In [132]: ys
```

```
Out[132]:
```

```
array([[-5. , -5. , -5. , ..., -5. , -5. , -5. ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

Now, evaluating the function is a simple matter of writing the same expression you would write with two points:

```
In [134]: import matplotlib.pyplot as plt
```

```
In [135]: z = np.sqrt(xs ** 2 + ys ** 2)
```

```
In [136]: z
```

```
Out[136]:
```

```
array([[ 7.0711,  7.064 ,  7.0569, ...,  7.0499,  7.0569,  7.064 ],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
       ...,
       [ 7.0499,  7.0428,  7.0357, ...,  7.0286,  7.0357,  7.0428],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569]])
```

While perhaps not immediately obvious, this `for` loop can be converted into a nested `where` expression:

```
np.where(cond1 & cond2, 0,
         np.where(cond1, 1,
                  np.where(cond2, 2, 3)))
```

In this particular example, we can also take advantage of the fact that boolean values are treated as 0 or 1 in calculations, so this could alternatively be expressed (though a bit more cryptically) as an arithmetic operation:

```
result = 1 * cond1 + 2 * cond2 + 3 * -(cond1 | cond2)
```

## Mathematical and Statistical Methods

A set of mathematical functions which compute statistics about an entire array or about the data along an axis are accessible as array methods. Aggregations (often called *reductions*) like `sum`, `mean`, and standard deviation `std` can either be used by calling the array instance method or using the top level NumPy function:

```
In [151]: arr = np.random.randn(5, 4) # normally-distributed data
```

```
In [152]: arr.mean()
Out[152]: 0.062814911084854597
```

```
In [153]: np.mean(arr)
Out[153]: 0.062814911084854597
```

```
In [154]: arr.sum()
Out[154]: 1.2562982216970919
```

Functions like `mean` and `sum` take an optional `axis` argument which computes the statistic over the given axis, resulting in an array with one fewer dimension:

```
In [155]: arr.mean(axis=1)
Out[155]: array([-1.2833,  0.2844,  0.6574,  0.6743, -0.0187])
```

```
In [156]: arr.sum(0)
Out[156]: array([-3.1003, -1.6189,  1.4044,  4.5712])
```

Other methods like `cumsum` and `cumprod` do not aggregate, instead producing an array of the intermediate results:

```
In [157]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
```

In [158]: arr.cumsum(0) Out[158]: array([[ 0,  1,  2], [ 3,  5,  7], [ 9, 12, 15]])	In [159]: arr.cumprod(1) Out[159]: array([[ 0,  0,  0], [ 3, 12, 60], [ 6, 42, 336]])
---	---

See [Table 4-5](#) for a full listing. We'll see many examples of these methods in action in later chapters.

```

Out[205]:
array([[ 1.,  0.,  0.,  0., -0.],
       [ 0.,  1., -0.,  0.,  0.],
       [ 0., -0.,  1.,  0.,  0.],
       [ 0., -0., -0.,  1., -0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

```

In [206]: q, r = qr(mat)
```

```

In [207]: r
Out[207]:
array([[-6.9271,  7.389 ,  6.1227, -7.1163, -4.9215],
       [ 0.    , -3.9735, -0.8671,  2.9747, -5.7402],
       [ 0.    ,  0.    , -10.2681,  1.8909,  1.6079],
       [ 0.    ,  0.    ,  0.    , -1.2996,  3.3577],
       [ 0.    ,  0.    ,  0.    ,  0.    ,  0.5571]])
```

See [Table 4-7](#) for a list of some of the most commonly-used linear algebra functions.



The scientific Python community is hopeful that there may be a matrix multiplication infix operator implemented someday, providing syntactically nicer alternative to using `np.dot`. But for now this is the way.

*Table 4-7. Commonly-used numpy.linalg functions*

Function	Description
<code>diag</code>	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
<code>dot</code>	Matrix multiplication
<code>trace</code>	Compute the sum of the diagonal elements
<code>det</code>	Compute the matrix determinant
<code>eig</code>	Compute the eigenvalues and eigenvectors of a square matrix
<code>inv</code>	Compute the inverse of a square matrix
<code>pinv</code>	Compute the Moore-Penrose pseudo-inverse of a square matrix
<code>qr</code>	Compute the QR decomposition
<code>svd</code>	Compute the singular value decomposition (SVD)
<code>solve</code>	Solve the linear system $Ax = b$ for $x$ , where $A$ is a square matrix
<code>lstsq</code>	Compute the least-squares solution to $y = Xb$

## Random Number Generation

The `numpy.random` module supplements the built-in Python `random` with functions for efficiently generating whole arrays of sample values from many kinds of probability

distributions. For example, you can get a 4 by 4 array of samples from the standard normal distribution using `normal`:

```
In [208]: samples = np.random.normal(size=(4, 4))
```

```
In [209]: samples
Out[209]:
array([[ 0.1241,  0.3026,  0.5238,  0.0009],
       [ 1.3438, -0.7135, -0.8312, -2.3702],
       [-1.8608, -0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329, -2.3594]])
```

Python's built-in `random` module, by contrast, only samples one value at a time. As you can see from this benchmark, `numpy.random` is well over an order of magnitude faster for generating very large samples:

```
In [210]: from random import normalvariate
```

```
In [211]: N = 1000000
```

```
In [212]: %timeit samples = [normalvariate(0, 1) for _ in xrange(N)]
1 loops, best of 3: 1.33 s per loop
```

```
In [213]: %timeit np.random.normal(size=N)
10 loops, best of 3: 57.7 ms per loop
```

See table [Table 4-8](#) for a partial list of functions available in `numpy.random`. I'll give some examples of leveraging these functions' ability to generate large arrays of samples all at once in the next section.

*Table 4-8. Partial list of `numpy.random` functions*

Function	Description
<code>seed</code>	Seed the random number generator
<code>permutation</code>	Return a random permutation of a sequence, or return a permuted range
<code>shuffle</code>	Randomly permute a sequence in place
<code>rand</code>	Draw samples from a uniform distribution
<code>randint</code>	Draw random integers from a given low-to-high range
<code>randn</code>	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
<code>binomial</code>	Draw samples from a binomial distribution
<code>normal</code>	Draw samples from a normal (Gaussian) distribution
<code>beta</code>	Draw samples from a beta distribution
<code>chisquare</code>	Draw samples from a chi-square distribution
<code>gamma</code>	Draw samples from a gamma distribution
<code>uniform</code>	Draw samples from a uniform [0, 1] distribution

## Example: Random Walks

An illustrative application of utilizing array operations is in the simulation of random walks. Let's first consider a simple random walk starting at 0 with steps of 1 and -1 occurring with equal probability. A pure Python way to implement a single random walk with 1,000 steps using the built-in `random` module:

```
import random
position = 0
walk = [position]
steps = 1000
for i in xrange(steps):
    step = 1 if random.randint(0, 1) else -1
    position += step
    walk.append(position)
```

See [Figure 4-4](#) for an example plot of the first 100 values on one of these random walks.

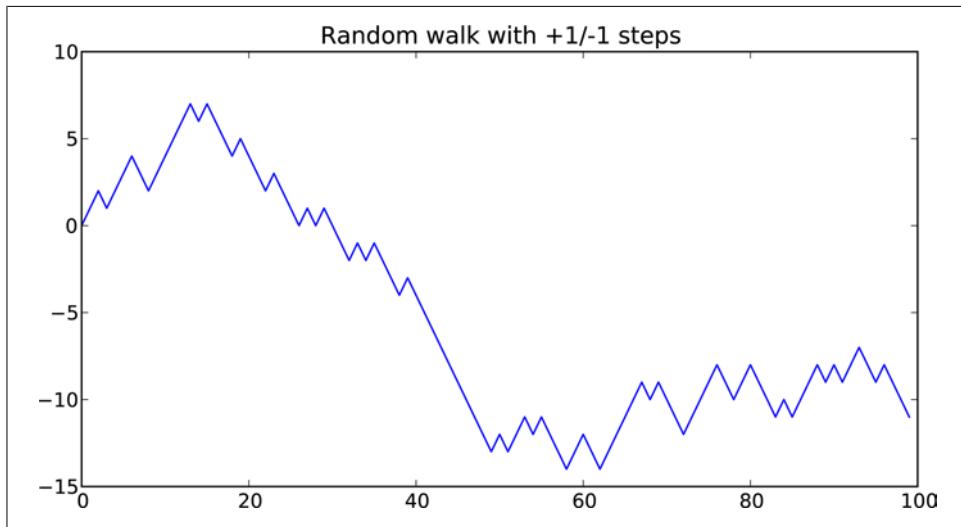


Figure 4-4. A simple random walk

You might make the observation that `walk` is simply the cumulative sum of the random steps and could be evaluated as an array expression. Thus, I use the `np.random` module to draw 1,000 coin flips at once, set these to 1 and -1, and compute the cumulative sum:

```
In [215]: nsteps = 1000
In [216]: draws = np.random.randint(0, 2, size=nsteps)
In [217]: steps = np.where(draws > 0, 1, -1)
In [218]: walk = steps.cumsum()
```

From this we can begin to extract statistics like the minimum and maximum value along the walk's trajectory:

```
In [219]: walk.min()          In [220]: walk.max()  
Out[219]: -3                 Out[220]: 31
```

A more complicated statistic is the *first crossing time*, the step at which the random walk reaches a particular value. Here we might want to know how long it took the random walk to get at least 10 steps away from the origin 0 in either direction. `np.abs(walk) >= 10` gives us a boolean array indicating where the walk has reached or exceeded 10, but we want the index of the *first* 10 or -10. Turns out this can be computed using `argmax`, which returns the first index of the maximum value in the boolean array (`True` is the maximum value):

```
In [221]: (np.abs(walk) >= 10).argmax()  
Out[221]: 37
```

Note that using `argmax` here is not always efficient because it always makes a full scan of the array. In this special case once a `True` is observed we know it to be the maximum value.

## Simulating Many Random Walks at Once

If your goal was to simulate many random walks, say 5,000 of them, you can generate all of the random walks with minor modifications to the above code. The `numpy.random` functions if passed a 2-tuple will generate a 2D array of draws, and we can compute the cumulative sum across the rows to compute all 5,000 random walks in one shot:

```
In [222]: nwalks = 5000  
  
In [223]: nsteps = 1000  
  
In [224]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 or 1  
  
In [225]: steps = np.where(draws > 0, 1, -1)  
  
In [226]: walks = steps.cumsum(1)  
  
In [227]: walks  
Out[227]:  
array([[ 1,  0,  1, ...,  8,  7,  8],  
       [ 1,  0, -1, ..., 34, 33, 32],  
       [ 1,  0, -1, ...,  4,  5,  4],  
       ...,  
       [ 1,  2,  1, ..., 24, 25, 26],  
       [ 1,  2,  3, ..., 14, 13, 14],  
       [-1, -2, -3, ..., -24, -23, -22]])
```

Now, we can compute the maximum and minimum values obtained over all of the walks:

```
In [228]: walks.max()          In [229]: walks.min()  
Out[228]: 138                 Out[229]: -133
```

Out of these walks, let's compute the minimum crossing time to 30 or -30. This is slightly tricky because not all 5,000 of them reach 30. We can check this using the `any` method:

```
In [230]: hits30 = (np.abs(walks) >= 30).any(1)
```

```
In [231]: hits30
```

```
Out[231]: array([False, True, False, ..., False, True, False], dtype=bool)
```

```
In [232]: hits30.sum() # Number that hit 30 or -30
```

```
Out[232]: 3410
```

We can use this boolean array to select out the rows of `walks` that actually cross the absolute 30 level and call `argmax` across axis 1 to get the crossing times:

```
In [233]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)
```

```
In [234]: crossing_times.mean()
```

```
Out[234]: 498.88973607038122
```

Feel free to experiment with other distributions for the steps other than equal sized coin flips. You need only use a different random number generation function, like `normal` to generate normally distributed steps with some mean and standard deviation:

```
In [235]: steps = np.random.normal(loc=0, scale=0.25,  
.....: size=(nwalks, nsteps))
```

# Getting Started with pandas

pandas will be the primary library of interest throughout much of the rest of the book. It contains high-level data structures and manipulation tools designed to make data analysis fast and easy in Python. pandas is built on top of NumPy and makes it easy to use in NumPy-centric applications.

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

- Data structures with labeled axes supporting automatic or explicit data alignment. This prevents common errors resulting from misaligned data and working with differently-indexed data coming from different sources.
- Integrated time series functionality.
- The same data structures handle both time series data and non-time series data.
- Arithmetic operations and reductions (like summing across an axis) would pass on the metadata (axis labels).
- Flexible handling of missing data.
- Merge and other relational operations found in popular database databases (SQL-based, for example).

I wanted to be able to do all of these things in one place, preferably in a language well-suited to general purpose software development. Python was a good candidate language for this, but at that time there was not an integrated set of data structures and tools providing this functionality.

Over the last four years, pandas has matured into a quite large library capable of solving a much broader set of data handling problems than I ever anticipated, but it has expanded in its scope without compromising the simplicity and ease-of-use that I desired from the very beginning. I hope that after reading this book, you will find it to be just as much of an indispensable tool as I do.

Throughout the rest of the book, I use the following import conventions for pandas:

```
In [1]: from pandas import Series, DataFrame
```

```
In [2]: import pandas as pd
```

Thus, whenever you see `pd.` in code, it's referring to pandas. Series and DataFrame are used so much that I find it easier to import them into the local namespace.

## Introduction to pandas Data Structures

To get started with pandas, you will need to get comfortable with its two workhorse data structures: *Series* and *DataFrame*. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

### Series

A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its *index*. The simplest Series is formed from only an array of data:

```
In [4]: obj = Series([4, 7, -5, 3])
```

```
In [5]: obj
```

```
Out[5]:
```

```
0    4  
1    7  
2   -5  
3    3
```

The string representation of a Series displayed interactively shows the index on the left and the values on the right. Since we did not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created. You can get the array representation and index object of the Series via its `values` and `index` attributes, respectively:

```
In [6]: obj.values
```

```
Out[6]: array([ 4,  7, -5,  3])
```

```
In [7]: obj.index
```

```
Out[7]: Int64Index([0, 1, 2, 3])
```

Often it will be desirable to create a Series with an index identifying each data point:

```
In [8]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
```

```
In [9]: obj2
```

```
Out[9]:
```

```
d    4  
b    7  
a   -5  
c    3
```

```
In [10]: obj2.index  
Out[10]: Index([d, b, a, c], dtype=object)
```

Compared with a regular NumPy array, you can use values in the index when selecting single values or a set of values:

```
In [11]: obj2['a']  
Out[11]: -5
```

```
In [12]: obj2['d'] = 6
```

```
In [13]: obj2[['c', 'a', 'd']]  
Out[13]:  
c    3  
a   -5  
d    6
```

NumPy array operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

```
In [14]: obj2  
Out[14]:  
d    6  
b    7  
a   -5  
c    3
```

```
In [15]: obj2[obj2 > 0]  
Out[15]:  
d    6  
b    7  
c    3
```

```
In [16]: obj2 * 2  
Out[16]:  
d    12  
b    14  
a   -10  
c     6
```

```
In [17]: np.exp(obj2)  
Out[17]:  
d    403.428793  
b  1096.633158  
a    0.006738  
c   20.085537
```

Another way to think about a Series is as a fixed-length, ordered dict, as it is a mapping of index values to data values. It can be substituted into many functions that expect a dict:

```
In [18]: 'b' in obj2  
Out[18]: True
```

```
In [19]: 'e' in obj2  
Out[19]: False
```

Should you have data contained in a Python dict, you can create a Series from it by passing the dict:

```
In [20]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

```
In [21]: obj3 = Series(sdata)
```

```
In [22]: obj3  
Out[22]:  
Ohio      35000  
Oregon    16000
```

```
Texas      71000  
Utah       5000
```

When only passing a dict, the index in the resulting Series will have the dict's keys in sorted order.

```
In [23]: states = ['California', 'Ohio', 'Oregon', 'Texas']
```

```
In [24]: obj4 = Series(sdata, index=states)
```

```
In [25]: obj4  
Out[25]:  
California      NaN  
Ohio            35000  
Oregon          16000  
Texas           71000
```

In this case, 3 values found in `sdata` were placed in the appropriate locations, but since no value for 'California' was found, it appears as `NaN` (not a number) which is considered in pandas to mark missing or `NA` values. I will use the terms "missing" or "NA" to refer to missing data. The `isnull` and `notnull` functions in pandas should be used to detect missing data:

```
In [26]: pd.isnull(obj4)      In [27]: pd.notnull(obj4)  
Out[26]:  
California    True           California   False  
Ohio          False          Ohio         True  
Oregon        False          Oregon       True  
Texas         False          Texas        True
```

Series also has these as instance methods:

```
In [28]: obj4.isnull()  
Out[28]:  
California    True  
Ohio          False  
Oregon        False  
Texas         False
```

I discuss working with missing data in more detail later in this chapter.

A critical Series feature for many applications is that it automatically aligns differently-indexed data in arithmetic operations:

```
In [29]: obj3      In [30]: obj4  
Out[29]:  
Ohio      35000    California      NaN  
Oregon    16000    Ohio            35000  
Texas     71000    Oregon          16000  
Utah      5000     Texas           71000
```

```
In [31]: obj3 + obj4  
Out[31]:  
California      NaN  
Ohio            70000  
Oregon          32000
```

```
Texas      142000  
Utah       NaN
```

Data alignment features are addressed as a separate topic.

Both the Series object itself and its index have a `name` attribute, which integrates with other key areas of pandas functionality:

```
In [32]: obj4.name = 'population'  
  
In [33]: obj4.index.name = 'state'  
  
In [34]: obj4  
Out[34]:  
state  
California      NaN  
Ohio            35000  
Oregon          16000  
Texas           71000  
Name: population
```

A Series's index can be altered in place by assignment:

```
In [35]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']  
  
In [36]: obj  
Out[36]:  
Bob     4  
Steve   7  
Jeff    -5  
Ryan    3
```

## DataFrame

A DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series (one for all sharing the same index). Compared with other such DataFrame-like structures you may have used before (like R's `data.frame`), row-oriented and column-oriented operations in DataFrame are treated roughly symmetrically. Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays. The exact details of DataFrame's internals are far outside the scope of this book.



While DataFrame stores the data internally in a two-dimensional format, you can easily represent much higher-dimensional data in a tabular format using hierarchical indexing, a subject of a later section and a key ingredient in many of the more advanced data-handling features in pandas.

There are numerous ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data)
```

The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order:

```
In [38]: frame
Out[38]:
   pop  state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
```

If you specify a sequence of columns, the DataFrame's columns will be exactly what you pass:

```
In [39]: DataFrame(data, columns=['year', 'state', 'pop'])
Out[39]:
   year  state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
```

As with Series, if you pass a column that isn't contained in `data`, it will appear with NA values in the result:

```
In [40]: frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                           ....:                 index=['one', 'two', 'three', 'four', 'five'])
In [41]: frame2
Out[41]:
   year  state  pop  debt
one  2000    Ohio  1.5    NaN
two  2001    Ohio  1.7    NaN
three  2002    Ohio  3.6    NaN
four  2001  Nevada  2.4    NaN
five  2002  Nevada  2.9    NaN
In [42]: frame2.columns
Out[42]: Index(['year', 'state', 'pop', 'debt'], dtype=object)
```

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

In [43]: frame2['state'] Out[43]: one      Ohio	In [44]: frame2.year Out[44]: one      2000
---	---

```
two      Ohio          two    2001
three    Ohio          three   2002
four     Nevada        four    2001
five     Nevada        five    2002
Name: state           Name: year
```

Note that the returned Series have the same index as the DataFrame, and their `name` attribute has been appropriately set.

Rows can also be retrieved by position or name by a couple of methods, such as the `ix` indexing field (much more on this later):

```
In [45]: frame2.ix['three']
Out[45]:
year    2002
state   Ohio
pop     3.6
debt    NaN
Name: three
```

Columns can be modified by assignment. For example, the empty '`debt`' column could be assigned a scalar value or an array of values:

```
In [46]: frame2['debt'] = 16.5

In [47]: frame2
Out[47]:
       year  state  pop  debt
one    2000   Ohio  1.5  16.5
two    2001   Ohio  1.7  16.5
three  2002   Ohio  3.6  16.5
four   2001  Nevada 2.4  16.5
five   2002  Nevada 2.9  16.5

In [48]: frame2['debt'] = np.arange(5.)

In [49]: frame2
Out[49]:
       year  state  pop  debt
one    2000   Ohio  1.5    0
two    2001   Ohio  1.7    1
three  2002   Ohio  3.6    2
four   2001  Nevada 2.4    3
five   2002  Nevada 2.9    4
```

When assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If you assign a Series, it will be instead conformed exactly to the DataFrame's index, inserting missing values in any holes:

```
In [50]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [51]: frame2['debt'] = val

In [52]: frame2
Out[52]:
       year  state  pop  debt
two    -1.2   Ohio  1.5  -1.2
four   -1.5  Nevada 2.4  -1.5
five   -1.7  Nevada 2.9  -1.7
```

```
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4  -1.5
five   2002  Nevada  2.9  -1.7
```

Assigning a column that doesn't exist will create a new column. The `del` keyword will delete columns as with a dict:

```
In [53]: frame2['eastern'] = frame2.state == 'Ohio'
```

```
In [54]: frame2
```

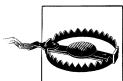
```
Out[54]:
```

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
five	2002	Nevada	2.9	-1.7	False

```
In [55]: del frame2['eastern']
```

```
In [56]: frame2.columns
```

```
Out[56]: Index(['year', 'state', 'pop', 'debt'], dtype=object)
```



The column returned when indexing a DataFrame is a *view* on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame. The column can be explicitly copied using the Series's `copy` method.

Another common form of data is a nested dict of dicts format:

```
In [57]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
....:           'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

If passed to DataFrame, it will interpret the outer dict keys as the columns and the inner keys as the row indices:

```
In [58]: frame3 = DataFrame(pop)
```

```
In [59]: frame3
```

```
Out[59]:
```

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

Of course you can always transpose the result:

```
In [60]: frame3.T
Out[60]:
```

	2000	2001	2002
Nevada	NaN	2.4	2.9
Ohio	1.5	1.7	3.6

The keys in the inner dicts are unioned and sorted to form the index in the result. This isn't true if an explicit index is specified:

```
In [61]: DataFrame(pop, index=[2001, 2002, 2003])
Out[61]:
      Nevada  Ohio
2001      2.4   1.7
2002      2.9   3.6
2003      NaN   NaN
```

Dicts of Series are treated much in the same way:

```
In [62]: pdata = {'Ohio': frame3['Ohio'][:-1],
....:             'Nevada': frame3['Nevada'][:2]}
In [63]: DataFrame(pdata)
Out[63]:
      Nevada  Ohio
2000      NaN   1.5
2001      2.4   1.7
```

For a complete list of things you can pass the DataFrame constructor, see [Table 5-1](#).

If a DataFrame's `index` and `columns` have their `name` attributes set, these will also be displayed:

```
In [64]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [65]: frame3
Out[65]:
      state  Nevada  Ohio
      year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6
```

Like Series, the `values` attribute returns the data contained in the DataFrame as a 2D ndarray:

```
In [66]: frame3.values
Out[66]:
array([[ nan,  1.5],
       [ 2.4,  1.7],
       [ 2.9,  3.6]])
```

If the DataFrame's columns are different dtypes, the dtype of the values array will be chosen to accomodate all of the columns:

```
In [67]: frame2.values
Out[67]:
array([[2000, Ohio, 1.5, nan],
       [2001, Ohio, 1.7, -1.2],
       [2002, Ohio, 3.6, nan],
       [2001, Nevada, 2.4, -1.5],
       [2002, Nevada, 2.9, -1.7]], dtype=object)
```

Table 5-1. Possible data inputs to DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the “dict of arrays” case
dict of Series	Each value becomes a column. Indexes from each Series are unioned together to form the result’s row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are unioned to form the row index as in the “dict of Series” case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame’s column labels
List of lists or tuples	Treated as the “2D ndarray” case
Another DataFrame	The DataFrame’s indexes are used unless different ones are passed
NumPy MaskedArray	Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result

## Index Objects

pandas’s Index objects are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or other sequence of labels used when constructing a Series or DataFrame is internally converted to an Index:

```
In [68]: obj = Series(range(3), index=['a', 'b', 'c'])

In [69]: index = obj.index

In [70]: index
Out[70]: Index([a, b, c], dtype=object)

In [71]: index[1:]
Out[71]: Index([b, c], dtype=object)
```

Index objects are immutable and thus can’t be modified by the user:

```
In [72]: index[1] = 'd'
-----
Exception                                Traceback (most recent call last)
<ipython-input-72-676fdeb26a68> in <module>()
----> 1 index[1] = 'd'
/Users/wesm/code/pandas/pandas/core/index.pyc in __setitem__(self, key, value)
    302     def __setitem__(self, key, value):
    303         """Disable the setting of values."""
--> 304         raise Exception(str(self.__class__) + ' object is immutable')
    305
    306     def __getitem__(self, key):
Exception: <class 'pandas.core.index.Index'> object is immutable
```

Immutability is important so that Index objects can be safely shared among data structures:

```
In [73]: index = pd.Index(np.arange(3))

In [74]: obj2 = Series([1.5, -2.5, 0], index=index)

In [75]: obj2.index is index
Out[75]: True
```

[Table 5-2](#) has a list of built-in Index classes in the library. With some development effort, Index can even be subclassed to implement specialized axis indexing functionality.



Many users will not need to know much about Index objects, but they're nonetheless an important part of pandas's data model.

*Table 5-2. Main Index objects in pandas*

Class	Description
Index	The most general Index object, representing axis labels in a NumPy array of Python objects.
Int64Index	Specialized Index for integer values.
MultiIndex	"Hierarchical" index object representing multiple levels of indexing on a single axis. Can be thought of as similar to an array of tuples.
DatetimeIndex	Stores nanosecond timestamps (represented using NumPy's datetime64 dtype).
PeriodIndex	Specialized Index for Period data (timespans).

In addition to being array-like, an Index also functions as a fixed-size set:

```
In [76]: frame3
Out[76]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6

In [77]: 'Ohio' in frame3.columns
Out[77]: True

In [78]: 2003 in frame3.index
Out[78]: False
```

Each Index has a number of methods and properties for set logic and answering other common questions about the data it contains. These are summarized in [Table 5-3](#).

Table 5-3. Index methods and properties

Method	Description
append	Concatenate with additional Index objects, producing a new Index
diff	Compute set difference as an Index
intersection	Compute set intersection
union	Compute set union
isin	Compute boolean array indicating whether each value is contained in the passed collection
delete	Compute new Index with element at index <i>i</i> deleted
drop	Compute new index by deleting passed values
insert	Compute new Index by inserting element at index <i>i</i>
is_monotonic	Returns True if each element is greater than or equal to the previous element
is_unique	Returns True if the Index has no duplicate values
unique	Compute the array of unique values in the Index

## Essential Functionality

In this section, I'll walk you through the fundamental mechanics of interacting with the data contained in a Series or DataFrame. Upcoming chapters will delve more deeply into data analysis and manipulation topics using pandas. This book is not intended to serve as exhaustive documentation for the pandas library; I instead focus on the most important features, leaving the less common (that is, more esoteric) things for you to explore on your own.

### Reindexing

A critical method on pandas objects is `reindex`, which means to create a new object with the data *conformed* to a new index. Consider a simple example from above:

```
In [79]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])  
  
In [80]: obj  
Out[80]:  
d    4.5  
b    7.2  
a   -5.3  
c    3.6
```

Calling `reindex` on this Series rearranges the data according to the new index, introducing missing values if any index values were not already present:

```
In [81]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])  
  
In [82]: obj2  
Out[82]:  
a   -5.3
```

```

b    7.2
c    3.6
d    4.5
e    NaN

In [83]: obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)
Out[83]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    0.0

```

For ordered data like time series, it may be desirable to do some interpolation or filling of values when reindexing. The `method` option allows us to do this, using a method such as `ffill` which forward fills the values:

```

In [84]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

In [85]: obj3.reindex(range(6), method='ffill')
Out[85]:
0    blue
1    blue
2  purple
3  purple
4  yellow
5  yellow

```

**Table 5-4** lists available `method` options. At this time, interpolation more sophisticated than forward- and backfilling would need to be applied after the fact.

*Table 5-4. reindex method (interpolation) options*

Argument	Description
<code>ffill</code> or <code>pad</code>	Fill (or carry) values forward
<code>bfill</code> or <code>backfill</code>	Fill (or carry) values backward

With DataFrame, `reindex` can alter either the (row) index, columns, or both. When passed just a sequence, the rows are reindexed in the result:

```

In [86]: frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],
....:                           columns=['Ohio', 'Texas', 'California'])

In [87]: frame
Out[87]:
   Ohio  Texas  California
a      0      1      2
c      3      4      5
d      6      7      8

In [88]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [89]: frame2
Out[89]:

```

```
      Ohio  Texas  California
a      0      1          2
b     NaN    NaN        NaN
c      3      4          5
d      6      7          8
```

The columns can be reindexed using the `columns` keyword:

```
In [90]: states = ['Texas', 'Utah', 'California']
```

```
In [91]: frame.reindex(columns=states)
```

```
Out[91]:
```

```
      Texas  Utah  California
a      1  NaN      2
c      4  NaN      5
d      7  NaN      8
```

Both can be reindexed in one shot, though interpolation will only apply row-wise (axis 0):

```
In [92]: frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill',
....:                  columns=states)
```

```
Out[92]:
```

```
      Texas  Utah  California
a      1  NaN      2
b      1  NaN      2
c      4  NaN      5
d      7  NaN      8
```

As you'll see soon, reindexing can be done more succinctly by label-indexing with `ix`:

```
In [93]: frame.ix[['a', 'b', 'c', 'd'], states]
```

```
Out[93]:
```

```
      Texas  Utah  California
a      1  NaN      2
b     NaN    NaN        NaN
c      4  NaN      5
d      7  NaN      8
```

*Table 5-5. reindex function arguments*

Argument	Description
<code>index</code>	New sequence to use as index. Can be <code>Index</code> instance or any other sequence-like Python data structure. An <code>Index</code> will be used exactly as is without any copying.
<code>method</code>	Interpolation (fill) method, see <a href="#">Table 5-4</a> for options.
<code>fill_value</code>	Substitute value to use when introducing missing data by reindexing
<code>limit</code>	When forward- or backfilling, maximum size gap to fill
<code>level</code>	Match simple <code>Index</code> on level of <code>MultIndex</code> , otherwise select subset of
<code>copy</code>	Do not copy underlying data if new index is equivalent to old index. <code>True</code> by default (i.e. always copy data).

## Dropping entries from an axis

Dropping one or more entries from an axis is easy if you have an index array or list without those entries. As that can require a bit of munging and set logic, the `drop` method will return a new object with the indicated value or values deleted from an axis:

```
In [94]: obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [95]: new_obj = obj.drop('c')
```

```
In [96]: new_obj
```

```
Out[96]:
```

```
a    0  
b    1  
d    3  
e    4
```

```
In [97]: obj.drop(['d', 'c'])
```

```
Out[97]:
```

```
a    0  
b    1  
e    4
```

With DataFrame, index values can be deleted from either axis:

```
In [98]: data = DataFrame(np.arange(16).reshape((4, 4)),  
....:                  index=['Ohio', 'Colorado', 'Utah', 'New York'],  
....:                  columns=['one', 'two', 'three', 'four'])
```

```
In [99]: data.drop(['Colorado', 'Ohio'])
```

```
Out[99]:
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

```
In [100]: data.drop('two', axis=1)
```

```
Out[100]:
```

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

```
In [101]: data.drop(['two', 'four'], axis=1)
```

```
Out[101]:
```

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

## Indexing, selection, and filtering

Series indexing (`obj[...]`) works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers. Here are some examples this:

```
In [102]: obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
```

```
In [103]: obj['b']
```

```
Out[103]: 1.0
```

```
In [104]: obj[1]
```

```
Out[104]: 1.0
```

```
In [105]: obj[2:4]
```

```
Out[105]:
```

```
In [106]: obj[['b', 'a', 'd']]
```

```
Out[106]:
```

```

c    2          b    1
d    3          a    0
                  d    3

In [107]: obj[[1, 3]]      In [108]: obj[obj < 2]
Out[107]:                   Out[108]:
b    1          a    0
d    3          b    1

```

Slicing with labels behaves differently than normal Python slicing in that the endpoint is inclusive:

```

In [109]: obj['b':'c']
Out[109]:
b    1
c    2

```

*Setting* using these methods works just as you would expect:

```

In [110]: obj['b':'c'] = 5

In [111]: obj
Out[111]:
a    0
b    5
c    5
d    3

```

As you've seen above, indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

```

In [112]: data = DataFrame(np.arange(16).reshape((4, 4)),
.....:                      index=['Ohio', 'Colorado', 'Utah', 'New York'],
.....:                      columns=['one', 'two', 'three', 'four'])

In [113]: data
Out[113]:
   one  two  three  four
Ohio     0    1     2     3
Colorado  4    5     6     7
Utah     8    9    10    11
New York 12   13    14    15

In [114]: data['two']           In [115]: data[['three', 'one']]
Out[114]:                      Out[115]:
Ohio      1                      three  one
Colorado  5                      Ohio      2    0
Utah     9                      Colorado  6    4
New York 13                     Utah     10   8
Name: two                         New York 14   12

```

Indexing like this has a few special cases. First selecting rows by slicing or a boolean array:

```

In [116]: data[:2]           In [117]: data[data['three'] > 5]
Out[116]:                      Out[117]:
   one  two  three  four          one  two  three  four

```

Ohio	0	1	2	3	Colorado	4	5	6	7
Colorado	4	5	6	7	Utah	8	9	10	11
					New York	12	13	14	15

This might seem inconsistent to some readers, but this syntax arose out of practicality and nothing more. Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

```
In [118]: data < 5
Out[118]:
      one   two  three  four
Ohio    True  True  True  True
Colorado  True False False False
Utah    False False False False
New York False False False False
```

```
In [119]: data[data < 5] = 0
```

```
In [120]: data
Out[120]:
      one   two  three  four
Ohio     0     0     0     0
Colorado  0     5     6     7
Utah     8     9    10    11
New York 12    13    14    15
```

This is intended to make DataFrame syntactically more like an ndarray in this case.

For DataFrame label-indexing on the rows, I introduce the special indexing field `ix`. It enables you to select a subset of the rows and columns from a DataFrame with NumPy-like notation plus axis labels. As I mentioned earlier, this is also a less verbose way to do reindexing:

```
In [121]: data.ix['Colorado', ['two', 'three']]
Out[121]:
two    5
three  6
Name: Colorado
```

```
In [122]: data.ix[['Colorado', 'Utah'], [3, 0, 1]]
Out[122]:
      four  one  two
Colorado    7    0    5
Utah       11   8    9
```

```
In [123]: data.ix[2]
Out[123]:
one    8
two    9
three  10
four   11
Name: Utah
```

```
In [124]: data.ix[:'Utah', 'two']
Out[124]:
Ohio      0
Colorado  5
Utah     9
Name: two
```

```
In [125]: data.ix[data.three > 5, :3]
Out[125]:
```

```

      one  two  three
Colorado    0    5     6
Utah        8    9    10
New York   12   13    14

```

So there are many ways to select and rearrange the data contained in a pandas object. For DataFrame, there is a short summary of many of them in [Table 5-6](#). You have a number of additional options when working with hierarchical indexes as you'll later see.



When designing pandas, I felt that having to type `frame[:, col]` to select a column was too verbose (and error-prone), since column selection is one of the most common operations. Thus I made the design trade-off to push all of the rich label-indexing into `ix`.

*Table 5-6. Indexing options with DataFrame*

Type	Notes
<code>obj[val]</code>	Select single column or sequence of columns from the DataFrame. Special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion).
<code>obj.ix[val]</code>	Selects single row or subset of rows from the DataFrame.
<code>obj.ix[:, val]</code>	Selects single column or subset of columns.
<code>obj.ix[val1, val2]</code>	Select both rows and columns.
<code>reindex</code> method	Conform one or more axes to new indexes.
<code>xs</code> method	Select single row or column as a Series by label.
<code>icol, irow</code> methods	Select single column or row, respectively, as a Series by integer location.
<code>get_value, set_value</code> methods	Select single value by row and column label.

## Arithmetic and data alignment

One of the most important pandas features is the behavior of arithmetic between objects with different indexes. When adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs. Let's look at a simple example:

```

In [126]: s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

In [127]: s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])

In [128]: s1      In [129]: s2
Out[128]:          Out[129]:
a    7.3           a   -2.1
c   -2.5           c    3.6
d    3.4           e   -1.5

```

```
e    1.5          f    4.0
g    3.1
```

Adding these together yields:

```
In [130]: s1 + s2
Out[130]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
```

The internal data alignment introduces NA values in the indices that don't overlap. Missing values propagate in arithmetic computations.

In the case of DataFrame, alignment is performed on both the rows and the columns:

```
In [131]: df1 = DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
.....:                  index=['Ohio', 'Texas', 'Colorado'])

In [132]: df2 = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
.....:                  index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [133]: df1
Out[133]:
      b   c   d
Ohio  0   1   2
Texas 3   4   5
Colorado 6   7   8

In [134]: df2
Out[134]:
      b   d   e
Utah  0   1   2
Ohio  3   4   5
Texas 6   7   8
Oregon 9  10  11
```

Adding these together returns a DataFrame whose index and columns are the unions of the ones in each DataFrame:

```
In [135]: df1 + df2
Out[135]:
      b   c   d   e
Colorado  NaN  NaN  NaN  NaN
Ohio      3   NaN  6   NaN
Oregon    NaN  NaN  NaN  NaN
Texas     9   NaN  12  NaN
Utah      NaN  NaN  NaN  NaN
```

### Arithmetic methods with fill values

In arithmetic operations between differently-indexed objects, you might want to fill with a special value, like 0, when an axis label is found in one object but not the other:

```
In [136]: df1 = DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))
In [137]: df2 = DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))

In [138]: df1
Out[138]:
      a   b   c   d
In [139]: df2
Out[139]:
      a   b   c   d   e
```

```

0 0 1 2 3      0 0 1 2 3 4
1 4 5 6 7      1 5 6 7 8 9
2 8 9 10 11    2 10 11 12 13 14
                           3 15 16 17 18 19

```

Adding these together results in NA values in the locations that don't overlap:

```

In [140]: df1 + df2
Out[140]:
   a   b   c   d   e
0  0   2   4   6  NaN
1  9  11  13  15  NaN
2 18  20  22  24  NaN
3 NaN  NaN  NaN  NaN  NaN

```

Using the add method on df1, I pass df2 and an argument to `fill_value`:

```

In [141]: df1.add(df2, fill_value=0)
Out[141]:
   a   b   c   d   e
0  0   2   4   6   4
1  9  11  13  15   9
2 18  20  22  24  14
3 15  16  17  18  19

```

Relatedly, when reindexing a Series or DataFrame, you can also specify a different fill value:

```

In [142]: df1.reindex(columns=df2.columns, fill_value=0)
Out[142]:
   a   b   c   d   e
0  0   1   2   3   0
1  4   5   6   7   0
2  8   9  10  11   0

```

*Table 5-7. Flexible arithmetic methods*

Method	Description
<code>add</code>	Method for addition (+)
<code>sub</code>	Method for subtraction (-)
<code>div</code>	Method for division (/)
<code>mul</code>	Method for multiplication (*)

## Operations between DataFrame and Series

As with NumPy arrays, arithmetic between DataFrame and Series is well-defined. First, as a motivating example, consider the difference between a 2D array and one of its rows:

```
In [143]: arr = np.arange(12.).reshape((3, 4))
```

```

In [144]: arr
Out[144]:
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])

```

```

[ 8.,  9., 10., 11.]])

In [145]: arr[0]
Out[145]: array([ 0.,  1.,  2.,  3.])

In [146]: arr - arr[0]
Out[146]:
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])

```

This is referred to as *broadcasting* and is explained in more detail in [Chapter 12](#). Operations between a DataFrame and a Series are similar:

```

In [147]: frame = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
                           index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [148]: series = frame.ix[0]

In [149]: frame           In [150]: series
Out[149]:                  Out[150]:
          b   d   e           b   0
Utah    0   1   2           d   1
Ohio    3   4   5           e   2
Texas   6   7   8           Name: Utah
Oregon  9   10  11

```

By default, arithmetic between DataFrame and Series matches the index of the Series on the DataFrame's columns, broadcasting down the rows:

```

In [151]: frame - series
Out[151]:
          b   d   e
Utah    0   0   0
Ohio    3   3   3
Texas   6   6   6
Oregon  9   9   9

```

If an index value is not found in either the DataFrame's columns or the Series's index, the objects will be reindexed to form the union:

```

In [152]: series2 = Series(range(3), index=['b', 'e', 'f'])

In [153]: frame + series2
Out[153]:
          b   d   e   f
Utah    0  NaN  3  NaN
Ohio    3  NaN  6  NaN
Texas   6  NaN  9  NaN
Oregon  9  NaN 12  NaN

```

If you want to instead broadcast over the columns, matching on the rows, you have to use one of the arithmetic methods. For example:

```

In [154]: series3 = frame['d']

In [155]: frame      In [156]: series3

```

```
Out[155]:          Out[156]:  
      b  d  e    Utah      1  
Utah  0  1  2    Ohio      4  
Ohio   3  4  5   Texas     7  
Texas  6  7  8   Oregon    10  
Oregon 9 10 11  Name: d
```

```
In [157]: frame.sub(series3, axis=0)  
Out[157]:  
      b  d  e  
Utah -1  0  1  
Ohio -1  0  1  
Texas -1  0  1  
Oregon -1  0  1
```

The axis number that you pass is the *axis to match on*. In this case we mean to match on the DataFrame's row index and broadcast across.

## Function application and mapping

NumPy ufuncs (element-wise array methods) work fine with pandas objects:

```
In [158]: frame = DataFrame(np.random.randn(4, 3), columns=list('bde'),  
.....:           index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

```
In [159]: frame  
Out[159]:  
      b         d         e  
Utah -0.204708  0.478943 -0.519439  
Ohio -0.555730  1.965781  1.393406  
Texas 0.092908  0.281746  0.769023  
Oregon 1.246435  1.007189 -1.296221
```

```
In [160]: np.abs(frame)  
Out[160]:  
      b         d         e  
Utah  0.204708  0.478943  0.519439  
Ohio  0.555730  1.965781  1.393406  
Texas 0.092908  0.281746  0.769023  
Oregon 1.246435  1.007189  1.296221
```

Another frequent operation is applying a function on 1D arrays to each column or row. DataFrame's `apply` method does exactly this:

```
In [161]: f = lambda x: x.max() - x.min()
```

```
In [162]: frame.apply(f)  
Out[162]:  
      b    1.802165  
      d    1.684034  
      e    2.689627
```

```
In [163]: frame.apply(f, axis=1)  
Out[163]:  
      Utah    0.998382  
      Ohio    2.521511  
      Texas   0.676115  
      Oregon  2.542656
```

Many of the most common array statistics (like `sum` and `mean`) are DataFrame methods, so using `apply` is not necessary.

The function passed to `apply` need not return a scalar value, it can also return a Series with multiple values:

```
In [164]: def f(x):  
.....:     return Series([x.min(), x.max()], index=['min', 'max'])
```

```
In [165]: frame.apply(f)
```

```
Out[165]:  
      b          d          e  
min -0.555730  0.281746 -1.296221  
max  1.246435  1.965781  1.393406
```

Element-wise Python functions can be used, too. Suppose you wanted to compute a formatted string from each floating point value in `frame`. You can do this with `applymap`:

```
In [166]: format = lambda x: '%.2f' % x
```

```
In [167]: frame.applymap(format)  
Out[167]:
```

	b	d	e
Utah	-0.20	0.48	-0.52
Ohio	-0.56	1.97	1.39
Texas	0.09	0.28	0.77
Oregon	1.25	1.01	-1.30

The reason for the name `applymap` is that Series has a `map` method for applying an element-wise function:

```
In [168]: frame['e'].map(format)  
Out[168]:  
Utah      -0.52  
Ohio       1.39  
Texas      0.77  
Oregon     -1.30  
Name: e
```

## Sorting and ranking

Sorting a data set by some criterion is another important built-in operation. To sort lexicographically by row or column index, use the `sort_index` method, which returns a new, sorted object:

```
In [169]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])
```

```
In [170]: obj.sort_index()
```

```
Out[170]:  
a    1  
b    2  
c    3  
d    0
```

With a DataFrame, you can sort by index on either axis:

```
In [171]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],  
.....:           columns=['d', 'a', 'b', 'c'])
```

```
In [172]: frame.sort_index()  
Out[172]:  
      d  a  b  c  
one   4  5  6  7  
three  0  1  2  3
```

```
In [173]: frame.sort_index(axis=1)  
Out[173]:  
      a  b  c  d  
three  1  2  3  0  
one    5  6  7  4
```

The data is sorted in ascending order by default, but can be sorted in descending order, too:

```
In [174]: frame.sort_index(axis=1, ascending=False)
Out[174]:
      d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

To sort a Series by its values, use its `order` method:

```
In [175]: obj = Series([4, 7, -3, 2])
In [176]: obj.order()
Out[176]:
2    -3
3     2
0     4
1     7
```

Any missing values are sorted to the end of the Series by default:

```
In [177]: obj = Series([4, np.nan, 7, np.nan, -3, 2])
In [178]: obj.order()
Out[178]:
4    -3
5     2
0     4
2     7
1    NaN
3    NaN
```

On DataFrame, you may want to sort by the values in one or more columns. To do so, pass one or more column names to the `by` option:

```
In [179]: frame = DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
In [180]: frame           In [181]: frame.sort_index(by='b')
Out[180]:               Out[181]:
      a   b                 a   b
0   0   4                 2   0  -3
1   1   7                 3   1   2
2   0  -3                 0   0   4
3   1   2                 1   1   7
```

To sort by multiple columns, pass a list of names:

```
In [182]: frame.sort_index(by=['a', 'b'])
Out[182]:
      a   b
2   0  -3
0   0   4
3   1   2
1   1   7
```

*Ranking* is closely related to sorting, assigning ranks from one through the number of valid data points in an array. It is similar to the indirect sort indices produced by `numpy.argsort`, except that ties are broken according to a rule. The `rank` methods for Series and DataFrame are the place to look; by default `rank` breaks ties by assigning each group the mean rank:

```
In [183]: obj = Series([7, -5, 7, 4, 2, 0, 4])
```

```
In [184]: obj.rank()
```

```
Out[184]:
```

0	6.5
1	1.0
2	6.5
3	4.5
4	3.0
5	2.0
6	4.5

Ranks can also be assigned according to the order they're observed in the data:

```
In [185]: obj.rank(method='first')
```

```
Out[185]:
```

0	6
1	1
2	7
3	4
4	3
5	2
6	5

Naturally, you can rank in descending order, too:

```
In [186]: obj.rank(ascending=False, method='max')
```

```
Out[186]:
```

0	2
1	7
2	2
3	4
4	5
5	6
6	4

See [Table 5-8](#) for a list of tie-breaking methods available. DataFrame can compute ranks over the rows or the columns:

```
In [187]: frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],  
.....: 'c': [-2, 5, 8, -2.5]})
```

```
In [188]: frame
```

```
Out[188]:
```

a	b	c
0	0	4.3
1	1	7.0
2	0	-3.0
3	1	2.0

```
In [189]: frame.rank(axis=1)
```

```
Out[189]:
```

a	b	c
0	2	3
1	1	3
2	2	1
3	2	3

Table 5-8. Tie-breaking methods with rank

Method	Description
'average'	Default: assign the average rank to each entry in the equal group.
'min'	Use the minimum rank for the whole group.
'max'	Use the maximum rank for the whole group.
'first'	Assign ranks in the order the values appear in the data.

## Axis indexes with duplicate values

Up until now all of the examples I've showed you have had unique axis labels (index values). While many pandas functions (like `reindex`) require that the labels be unique, it's not mandatory. Let's consider a small Series with duplicate indices:

```
In [190]: obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
```

```
In [191]: obj
```

```
Out[191]:
```

```
a    0  
a    1  
b    2  
b    3  
c    4
```

The index's `is_unique` property can tell you whether its values are unique or not:

```
In [192]: obj.index.is_unique
```

```
Out[192]: False
```

Data selection is one of the main things that behaves differently with duplicates. Indexing a value with multiple entries returns a Series while single entries return a scalar value:

```
In [193]: obj['a']      In [194]: obj['c']  
Out[193]:                  Out[194]: 4  
a    0  
a    1
```

The same logic extends to indexing rows in a DataFrame:

```
In [195]: df = DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])
```

```
In [196]: df
```

```
Out[196]:
```

```
          0         1         2  
a  0.274992  0.228913  1.352917  
a  0.886429 -2.001637 -0.371843  
b  1.669025 -0.438570 -0.539741  
b  0.476985  3.248944 -1.021228
```

```
In [197]: df.ix['b']
```

```
Out[197]:
```

```
0         1         2
```

```
b  1.669025 -0.438570 -0.539741  
b  0.476985  3.248944 -1.021228
```

## Summarizing and Computing Descriptive Statistics

pandas objects are equipped with a set of common mathematical and statistical methods. Most of these fall into the category of *reductions* or *summary statistics*, methods that extract a single value (like the sum or mean) from a Series or a Series of values from the rows or columns of a DataFrame. Compared with the equivalent methods of vanilla NumPy arrays, they are all built from the ground up to exclude missing data. Consider a small DataFrame:

```
In [198]: df = DataFrame([[1.4, np.nan], [7.1, -4.5],  
.....:                 [np.nan, np.nan], [0.75, -1.3]],  
.....:                 index=['a', 'b', 'c', 'd'],  
.....:                 columns=['one', 'two'])  
  
In [199]: df  
Out[199]:  
      one  two  
a  1.40  NaN  
b  7.10 -4.5  
c  NaN   NaN  
d  0.75 -1.3
```

Calling DataFrame's `sum` method returns a Series containing column sums:

```
In [200]: df.sum()  
Out[200]:  
one    9.25  
two   -5.80
```

Passing `axis=1` sums over the rows instead:

```
In [201]: df.sum(axis=1)  
Out[201]:  
a    1.40  
b    2.60  
c    NaN  
d   -0.55
```

NA values are excluded unless the entire slice (row or column in this case) is NA. This can be disabled using the `skipna` option:

```
In [202]: df.mean(axis=1, skipna=False)  
Out[202]:  
a    NaN  
b    1.300  
c    NaN  
d   -0.275
```

See [Table 5-9](#) for a list of common options for each reduction method options.

Table 5-9. Options for reduction methods

Method	Description
axis	Axis to reduce over. 0 for DataFrame's rows and 1 for columns.
skipna	Exclude missing values, True by default.
level	Reduce grouped by level if the axis is hierarchically-indexed (MultiIndex).

Some methods, like `idxmin` and `idxmax`, return indirect statistics like the index value where the minimum or maximum values are attained:

```
In [203]: df.idxmax()  
Out[203]:  
one    b  
two    d
```

Other methods are *accumulations*:

```
In [204]: df.cumsum()  
Out[204]:  
      one   two  
a  1.40  NaN  
b  8.50 -4.5  
c  NaN   NaN  
d  9.25 -5.8
```

Another type of method is neither a reduction nor an accumulation. `describe` is one such example, producing multiple summary statistics in one shot:

```
In [205]: df.describe()  
Out[205]:  
      one      two  
count  3.000000  2.000000  
mean   3.083333 -2.900000  
std    3.493685  2.262742  
min    0.750000 -4.500000  
25%   1.075000 -3.700000  
50%   1.400000 -2.900000  
75%   4.250000 -2.100000  
max    7.100000 -1.300000
```

On non-numeric data, `describe` produces alternate summary statistics:

```
In [206]: obj = Series(['a', 'a', 'b', 'c'] * 4)  
  
In [207]: obj.describe()  
Out[207]:  
count    16  
unique     3  
top      a  
freq      8
```

See [Table 5-10](#) for a full list of summary statistics and related methods.

Table 5-10. Descriptive and summary statistics

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index values at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (3rd moment) of values
kurt	Sample kurtosis (4th moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute 1st arithmetic difference (useful for time series)
pct_change	Compute percent changes

## Correlation and Covariance

Some summary statistics, like correlation and covariance, are computed from pairs of arguments. Let's consider some DataFrames of stock prices and volumes obtained from Yahoo! Finance:

```
import pandas.io.data as web

all_data = {}
for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:
    all_data[ticker] = web.get_data_yahoo(ticker, '1/1/2000', '1/1/2010')

price = DataFrame({tic: data['Adj Close']
                  for tic, data in all_data.iteritems()})
volume = DataFrame({tic: data['Volume']
                    for tic, data in all_data.iteritems()})
```

I now compute percent changes of the prices:

```
In [209]: returns = price.pct_change()
In [210]: returns.tail()
```

```
Out[210]:
```

	AAPL	GOOG	IBM	MSFT
Date				
2009-12-24	0.034339	0.011117	0.004420	0.002747
2009-12-28	0.012294	0.007098	0.013282	0.005479
2009-12-29	-0.011861	-0.005571	-0.003474	0.006812
2009-12-30	0.012147	0.005376	0.005468	-0.013532
2009-12-31	-0.004300	-0.004416	-0.012609	-0.015432

The `corr` method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series. Relatedly, `cov` computes the covariance:

```
In [211]: returns.MSFT.corr(returns.IBM)
Out[211]: 0.49609291822168838
```

```
In [212]: returns.MSFT.cov(returns.IBM)
Out[212]: 0.00021600332437329015
```

DataFrame's `corr` and `cov` methods, on the other hand, return a full correlation or covariance matrix as a DataFrame, respectively:

```
In [213]: returns.corr()
Out[213]:
```

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	0.470660	0.410648	0.424550
GOOG	0.470660	1.000000	0.390692	0.443334
IBM	0.410648	0.390692	1.000000	0.496093
MSFT	0.424550	0.443334	0.496093	1.000000

```
In [214]: returns.cov()
Out[214]:
```

	AAPL	GOOG	IBM	MSFT
AAPL	0.001028	0.000303	0.000252	0.000309
GOOG	0.000303	0.000580	0.000142	0.000205
IBM	0.000252	0.000142	0.000367	0.000216
MSFT	0.000309	0.000205	0.000216	0.000516

Using DataFrame's `corrwith` method, you can compute pairwise correlations between a DataFrame's columns or rows with another Series or DataFrame. Passing a Series returns a Series with the correlation value computed for each column:

```
In [215]: returns.corrwith(returns.IBM)
Out[215]:
```

AAPL	0.410648
GOOG	0.390692
IBM	1.000000
MSFT	0.496093

Passing a DataFrame computes the correlations of matching column names. Here I compute correlations of percent changes with volume:

```
In [216]: returns.corrwith(volume)
Out[216]:
```

AAPL	-0.057461
GOOG	0.062644

```
IBM    -0.007900
MSFT   -0.014175
```

Passing `axis=1` does things row-wise instead. In all cases, the data points are aligned by label before computing the correlation.

## Unique Values, Value Counts, and Membership

Another class of related methods extracts information about the values contained in a one-dimensional Series. To illustrate these, consider this example:

```
In [217]: obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

The first function is `unique`, which gives you an array of the unique values in a Series:

```
In [218]: uniques = obj.unique()
```

```
In [219]: uniques
Out[219]: array(['c', 'a', 'd', 'b'], dtype=object)
```

The unique values are not necessarily returned in sorted order, but could be sorted after the fact if needed (`uniques.sort()`). Relatedly, `value_counts` computes a Series containing value frequencies:

```
In [220]: obj.value_counts()
Out[220]:
```

```
c    3
a    3
b    2
d    1
```

The Series is sorted by value in descending order as a convenience. `value_counts` is also available as a top-level pandas method that can be used with any array or sequence:

```
In [221]: pd.value_counts(obj.values, sort=False)
```

```
Out[221]:
a    3
b    2
c    3
d    1
```

Lastly, `isin` is responsible for vectorized set membership and can be very useful in filtering a data set down to a subset of values in a Series or column in a DataFrame:

```
In [222]: mask = obj.isin(['b', 'c'])
```

In [223]: mask	In [224]: obj[mask]
Out[223]:	Out[224]:
0    True	0    c
1    False	5    b
2    False	6    b
3    False	7    c
4    False	8    c
5    True	
6    True	

```
7    True
8    True
```

See [Table 5-11](#) for a reference on these methods.

*Table 5-11. Unique, value counts, and binning methods*

Method	Description
isin	Compute boolean array indicating whether each Series value is contained in the passed sequence of values.
unique	Compute array of unique values in a Series, returned in the order observed.
value_counts	Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order.

In some cases, you may want to compute a histogram on multiple related columns in a DataFrame. Here's an example:

```
In [225]: data = DataFrame({'Qu1': [1, 3, 4, 3, 4],
.....:                 'Qu2': [2, 3, 1, 2, 3],
.....:                 'Qu3': [1, 5, 2, 4, 4]})

In [226]: data
Out[226]:
   Qu1  Qu2  Qu3
0     1     2     1
1     3     3     5
2     4     1     2
3     3     2     4
4     4     3     4
```

Passing `pandas.value_counts` to this DataFrame's `apply` function gives:

```
In [227]: result = data.apply(pd.value_counts).fillna(0)

In [228]: result
Out[228]:
   Qu1  Qu2  Qu3
1     1     1     1
2     0     2     1
3     2     2     0
4     2     0     2
5     0     0     1
```

## Handling Missing Data

Missing data is common in most data analysis applications. One of the goals in designing pandas was to make working with missing data as painless as possible. For example, all of the descriptive statistics on pandas objects exclude missing data as you've seen earlier in the chapter.

pandas uses the floating point value `NaN` (Not a Number) to represent missing data in both floating as well as in non-floating point arrays. It is just used as a *sentinel* that can be easily detected:

```
In [229]: string_data = Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [230]: string_data           In [231]: string_data.isnull()
Out[230]:                         Out[231]:
0    aardvark                   0    False
1    artichoke                  1    False
2    NaN                        2    True
3    avocado                     3    False
```

The built-in Python `None` value is also treated as NA in object arrays:

```
In [232]: string_data[0] = None

In [233]: string_data.isnull()
Out[233]:
0    True
1    False
2    True
3    False
```

I do not claim that pandas's NA representation is optimal, but it is simple and reasonably consistent. It's the best solution, with good all-around performance characteristics and a simple API, that I could concoct in the absence of a true NA data type or bit pattern in NumPy's data types. Ongoing development work in NumPy may change this in the future.

Table 5-12. NA handling methods

Argument	Description
<code>dropna</code>	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
<code>fillna</code>	Fill in missing data with some value or using an interpolation method such as ' <code>ffill</code> ' or ' <code>bfill</code> '.
<code>isnull</code>	Return like-type object containing boolean values indicating which values are missing / NA.
<code>notnull</code>	Negation of <code>isnull</code> .

## Filtering Out Missing Data

You have a number of options for filtering out missing data. While doing it by hand is always an option, `dropna` can be very helpful. On a Series, it returns the Series with only the non-null data and index values:

```
In [234]: from numpy import nan as NA

In [235]: data = Series([1, NA, 3.5, NA, 7])

In [236]: data.dropna()
Out[236]:
```

```
0    1.0  
2    3.5  
4    7.0
```

Naturally, you could have computed this yourself by boolean indexing:

```
In [237]: data[data.notnull()]  
Out[237]:  
0    1.0  
2    3.5  
4    7.0
```

With DataFrame objects, these are a bit more complex. You may want to drop rows or columns which are all NA or just those containing any NAs. `dropna` by default drops any row containing a missing value:

```
In [238]: data = DataFrame([[1., 6.5, 3.], [1., NA, NA],  
.....:           [NA, NA, NA], [NA, 6.5, 3.]])
```

```
In [239]: cleaned = data.dropna()
```

```
In [240]: data      In [241]: cleaned  
Out[240]:          Out[241]:  
0   1   2             0   1   2  
0   1   6.5   3       0   1   6.5   3  
1   1   NaN  NaN  
2  NaN  NaN  NaN  
3  NaN   6.5   3
```

Passing `how='all'` will only drop rows that are all NA:

```
In [242]: data.dropna(how='all')  
Out[242]:  
0   1   2  
0   1   6.5   3  
1   1   NaN  NaN  
3  NaN   6.5   3
```

Dropping columns in the same way is only a matter of passing `axis=1`:

```
In [243]: data[4] = NA
```

```
In [244]: data      In [245]: data.dropna(axis=1, how='all')  
Out[244]:          Out[245]:  
0   1   2   4             0   1   2  
0   1   6.5   3  NaN     0   1   6.5   3  
1   1   NaN  NaN  NaN     1   1   NaN  NaN  
2  NaN  NaN  NaN  NaN     2  NaN  NaN  NaN  
3  NaN   6.5   3  NaN     3  NaN   6.5   3
```

A related way to filter out DataFrame rows tends to concern time series data. Suppose you want to keep only rows containing a certain number of observations. You can indicate this with the `thresh` argument:

```
In [246]: df = DataFrame(np.random.randn(7, 3))
```

```
In [247]: df.ix[:4, 1] = NA; df.ix[:2, 2] = NA
```

```
In [248]: df  
Out[248]:
```

	0	1	2
0	-0.577087	NaN	NaN
1	0.523772	NaN	NaN
2	-0.713544	NaN	NaN
3	-1.860761	NaN	0.560145
4	-1.265934	NaN	-1.063512
5	0.332883	-2.359419	-0.199543
6	-1.541996	-0.970736	-1.307030

```
In [249]: df.dropna(thresh=3)  
Out[249]:
```

	0	1	2
5	0.332883	-2.359419	-0.199543
6	-1.541996	-0.970736	-1.307030

## Filling in Missing Data

Rather than filtering out missing data (and potentially discarding other data along with it), you may want to fill in the “holes” in any number of ways. For most purposes, the `fillna` method is the workhorse function to use. Calling `fillna` with a constant replaces missing values with that value:

```
In [250]: df.fillna(0)
```

```
Out[250]:
```

	0	1	2
0	-0.577087	0.000000	0.000000
1	0.523772	0.000000	0.000000
2	-0.713544	0.000000	0.000000
3	-1.860761	0.000000	0.560145
4	-1.265934	0.000000	-1.063512
5	0.332883	-2.359419	-0.199543
6	-1.541996	-0.970736	-1.307030

Calling `fillna` with a dict you can use a different fill value for each column:

```
In [251]: df.fillna({1: 0.5, 3: -1})
```

```
Out[251]:
```

	0	1	2
0	-0.577087	0.500000	NaN
1	0.523772	0.500000	NaN
2	-0.713544	0.500000	NaN
3	-1.860761	0.500000	0.560145
4	-1.265934	0.500000	-1.063512
5	0.332883	-2.359419	-0.199543
6	-1.541996	-0.970736	-1.307030

`fillna` returns a new object, but you can modify the existing object in place:

```
# always returns a reference to the filled object
```

```
In [252]: _ = df.fillna(0, inplace=True)
```

```
In [253]: df
```

```
Out[253]:
```

	0	1	2
0	-0.577087	0.000000	0.000000
1	0.523772	0.000000	0.000000
2	-0.713544	0.000000	0.000000
3	-1.860761	0.000000	0.560145

```
4 -1.265934  0.000000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

The same interpolation methods available for reindexing can be used with `fillna`:

```
In [254]: df = DataFrame(np.random.randn(6, 3))
```

```
In [255]: df.ix[2:, 1] = NA; df.ix[4:, 2] = NA
```

```
In [256]: df
```

```
Out[256]:
```

```
          0         1         2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674      NaN  1.004812
3  1.327195      NaN -1.549106
4  0.022185      NaN      NaN
5  0.862580      NaN      NaN
```

```
In [257]: df.fillna(method='ffill')
```

```
Out[257]:
```

```
          0         1         2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674  1.349742  1.004812
3  1.327195  1.349742 -1.549106
4  0.022185  1.349742 -1.549106
5  0.862580  1.349742 -1.549106
```

```
In [258]: df.fillna(method='ffill', limit=2)
```

```
Out[258]:
```

```
          0         1         2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674  1.349742  1.004812
3  1.327195  1.349742 -1.549106
4  0.022185      NaN -1.549106
5  0.862580      NaN -1.549106
```

With `fillna` you can do lots of other things with a little creativity. For example, you might pass the mean or median value of a Series:

```
In [259]: data = Series([1., NA, 3.5, NA, 7])
```

```
In [260]: data.fillna(data.mean())
```

```
Out[260]:
```

```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
```

See [Table 5-13](#) for a reference on `fillna`.

*Table 5-13. fillna function arguments*

Argument	Description
value	Scalar value or dict-like object to use to fill missing values
method	Interpolation, by default 'ffill' if function called with no other arguments
axis	Axis to fill on, default axis=0
inplace	Modify the calling object without producing a copy
limit	For forward and backward filling, maximum number of consecutive periods to fill

## Hierarchical Indexing

*Hierarchical indexing* is an important feature of pandas enabling you to have multiple (two or more) index *levels* on an axis. Somewhat abstractly, it provides a way for you to work with higher dimensional data in a lower dimensional form. Let's start with a simple example; create a Series with a list of lists or arrays as the index:

```
In [261]: data = Series(np.random.randn(10),
.....:                 index=[['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'd', 'd'],
.....:                     [1, 2, 3, 1, 2, 3, 1, 2, 2, 3]])
```

```
In [262]: data
Out[262]:
a    0.670216
     2    0.852965
     3   -0.955869
b    1   -0.023493
     2   -2.304234
     3   -0.652469
c    1   -1.218302
     2   -1.332610
d    2    1.074623
     3    0.723642
```

What you're seeing is a prettified view of a Series with a `MultiIndex` as its index. The “gaps” in the index display mean “use the label directly above”:

```
In [263]: data.index
Out[263]:
MultiIndex
[('a', 1) ('a', 2) ('a', 3) ('b', 1) ('b', 2) ('b', 3) ('c', 1)
 ('c', 2) ('d', 2) ('d', 3)]
```

With a hierarchically-indexed object, so-called *partial* indexing is possible, enabling you to concisely select subsets of the data:

```
In [264]: data['b']
Out[264]:
1   -0.023493
2   -2.304234
3   -0.652469
```

```
In [265]: data['b':'c']
Out[265]:
b    1   -0.023493
     2   -2.304234
     3   -0.652469
c    1   -1.218302
     2   -1.332610
```

```
In [266]: data.ix[['b', 'd']]
Out[266]:
b    1   -0.023493
     2   -2.304234
     3   -0.652469
d    2    1.074623
     3    0.723642
```

Selection is even possible in some cases from an “inner” level:

```
In [267]: data[:, 2]
Out[267]:
a    0.852965
```

```
b    -2.304234  
c    -1.332610  
d     1.074623
```

Hierarchical indexing plays a critical role in reshaping data and group-based operations like forming a pivot table. For example, this data could be rearranged into a DataFrame using its `unstack` method:

```
In [268]: data.unstack()  
Out[268]:  
      1         2         3  
a  0.670216  0.852965 -0.955869  
b -0.023493 -2.304234 -0.652469  
c -1.218302 -1.332610      NaN  
d      NaN  1.074623  0.723642
```

The inverse operation of `unstack` is `stack`:

```
In [269]: data.unstack().stack()  
Out[269]:  
a  1    0.670216  
   2    0.852965  
   3   -0.955869  
b  1   -0.023493  
   2   -2.304234  
   3   -0.652469  
c  1   -1.218302  
   2   -1.332610  
d  2    1.074623  
   3   0.723642
```

`stack` and `unstack` will be explored in more detail in [Chapter 7](#).

With a DataFrame, either axis can have a hierarchical index:

```
In [270]: frame = DataFrame(np.arange(12).reshape((4, 3)),  
.....:                 index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],  
.....:                 columns=[['Ohio', 'Ohio', 'Colorado'],  
.....:                           ['Green', 'Red', 'Green']])  
  
In [271]: frame  
Out[271]:  
          Ohio        Colorado  
          Green   Red   Green  
a 1      0     1     2  
   2      3     4     5  
b 1      6     7     8  
   2      9    10    11
```

The hierarchical levels can have names (as strings or any Python objects). If so, these will show up in the console output (don't confuse the index names with the axis labels!):

```
In [272]: frame.index.names = ['key1', 'key2']  
  
In [273]: frame.columns.names = ['state', 'color']  
  
In [274]: frame
```

```
Out[274]:
state      Ohio      Colorado
color      Green    Red      Green
key1 key2
a    1        0        1        2
     2        3        4        5
b    1        6        7        8
     2        9       10       11
```

With partial column indexing you can similarly select groups of columns:

```
In [275]: frame['Ohio']
Out[275]:
color      Green    Red
key1 key2
a    1        0        1
     2        3        4
b    1        6        7
     2        9       10
```

A `MultiIndex` can be created by itself and then reused; the columns in the above Data-Frame with level names could be created like this:

```
MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']],
                      names=['state', 'color'])
```

## Reordering and Sorting Levels

At times you will need to rearrange the order of the levels on an axis or sort the data by the values in one specific level. The `swaplevel` takes two level numbers or names and returns a new object with the levels interchanged (but the data is otherwise unaltered):

```
In [276]: frame.swaplevel('key1', 'key2')
Out[276]:
state      Ohio      Colorado
color      Green    Red      Green
key2 key1
1    a        0        1        2
2    a        3        4        5
1    b        6        7        8
2    b        9       10       11
```

`sortlevel`, on the other hand, sorts the data (stably) using only the values in a single level. When swapping levels, it's not uncommon to also use `sortlevel` so that the result is lexicographically sorted:

```
In [277]: frame.sortlevel(1)
Out[277]:
state      Ohio      Colorado
color      Green    Red      Green
key1 key2
a    1        0        1        2
b    1        6        7        8
a    2        3        4        5
b    2        9       10       11
```

```
In [278]: frame.swaplevel(0, 1).sortlevel(0)
Out[278]:
state      Ohio      Colorado
color      Green    Red      Green
key2 key1
1    a        0        1        2
     b        6        7        8
2    a        3        4        5
     b        9       10       11
```



Data selection performance is much better on hierarchically indexed objects if the index is lexicographically sorted starting with the outermost level, that is, the result of calling `sortlevel(0)` or `sort_index()`.

## Summary Statistics by Level

Many descriptive and summary statistics on DataFrame and Series have a `level` option in which you can specify the level you want to sum by on a particular axis. Consider the above DataFrame; we can sum by level on either the rows or columns like so:

```
In [279]: frame.sum(level='key2')
Out[279]:
state    Ohio      Colorado
color   Green    Red      Green
key2
1          6     8       10
2         12    14       16

In [280]: frame.sum(level='color', axis=1)
Out[280]:
color   Green  Red
key1 key2
a      1     2     1
      2     8     4
b      1    14     7
      2    20    10
```

Under the hood, this utilizes pandas's `groupby` machinery which will be discussed in more detail later in the book.

## Using a DataFrame's Columns

It's not unusual to want to use one or more columns from a DataFrame as the row index; alternatively, you may wish to move the row index into the DataFrame's columns. Here's an example DataFrame:

```
In [281]: frame = DataFrame({'a': range(7), 'b': range(7, 0, -1),
.....:                      'c': ['one', 'one', 'one', 'two', 'two', 'two'],
.....:                      'd': [0, 1, 2, 0, 1, 2, 3]})

In [282]: frame
Out[282]:
   a   b     c   d
0  0   7  one   0
1  1   6  one   1
2  2   5  one   2
3  3   4  two   0
4  4   3  two   1
5  5   2  two   2
6  6   1  two   3
```

DataFrame's `set_index` function will create a new DataFrame using one or more of its columns as the index:

```
In [283]: frame2 = frame.set_index(['c', 'd'])
```

```
In [284]: frame2
```

```
Out[284]:
```

	a	b
c	d	
one	0 0 7	
	1 1 6	
	2 2 5	
two	0 3 4	
	1 4 3	
	2 5 2	
	3 6 1	

By default the columns are removed from the DataFrame, though you can leave them in:

```
In [285]: frame.set_index(['c', 'd'], drop=False)
```

```
Out[285]:
```

	a	b	c	d
c	d			
one	0 0 7	one 0		
	1 1 6	one 1		
	2 2 5	one 2		
two	0 3 4	two 0		
	1 4 3	two 1		
	2 5 2	two 2		
	3 6 1	two 3		

`reset_index`, on the other hand, does the opposite of `set_index`; the hierarchical index levels are moved into the columns:

```
In [286]: frame2.reset_index()
```

```
Out[286]:
```

	c	d	a	b
0	one	0	0	7
1	one	1	1	6
2	one	2	2	5
3	two	0	3	4
4	two	1	4	3
5	two	2	5	2
6	two	3	6	1

## Other pandas Topics

Here are some additional topics that may be of use to you in your data travels.

### Integer Indexing

Working with pandas objects indexed by integers is something that often trips up new users due to some differences with indexing semantics on built-in Python data

structures like lists and tuples. For example, you would not expect the following code to generate an error:

```
ser = Series(np.arange(3.))
ser[-1]
```

In this case, pandas could “fall back” on integer indexing, but there’s not a safe and general way (that I know of) to do this without introducing subtle bugs. Here we have an index containing 0, 1, 2, but inferring what the user wants (label-based indexing or position-based) is difficult::

```
In [288]: ser
Out[288]:
0    0
1    1
2    2
```

On the other hand, with a non-integer index, there is no potential for ambiguity:

```
In [289]: ser2 = Series(np.arange(3.), index=['a', 'b', 'c'])

In [290]: ser2[-1]
Out[290]: 2.0
```

To keep things consistent, if you have an axis index containing indexers, data selection with integers will always be label-oriented. This includes slicing with `ix`, too:

```
In [291]: ser.ix[:1]
Out[291]:
0    0
1    1
```

In cases where you need reliable position-based indexing regardless of the index type, you can use the `iget_value` method from Series and `irow` and `icol` methods from DataFrame:

```
In [292]: ser3 = Series(range(3), index=[-5, 1, 3])

In [293]: ser3.iget_value(2)
Out[293]: 2

In [294]: frame = DataFrame(np.arange(6).reshape(3, 2), index=[2, 0, 1])

In [295]: frame.irow(0)
Out[295]:
0    0
1    1
Name: 2
```

## Panel Data

While not a major topic of this book, pandas has a Panel data structure, which you can think of as a three-dimensional analogue of DataFrame. Much of the development focus of pandas has been in tabular data manipulations as these are easier to reason about,

```
2012-05-31  577.73  12.33  580.86  29.19  
2012-06-01  560.99  12.07  570.98  28.45
```

An alternate way to represent panel data, especially for fitting statistical models, is in “stacked” DataFrame form:

```
In [302]: stacked = pdata.ix[:, '5/30/2012':, :].to_frame()
```

```
In [303]: stacked
```

```
Out[303]:
```

		Open	High	Low	Close	Volume	Adj Close
major	minor						
2012-05-30	AAPL	569.20	579.99	566.56	579.17	18908200	579.17
	DELL	12.59	12.70	12.46	12.56	19787800	12.56
	GOOG	588.16	591.90	583.53	588.23	1906700	588.23
	MSFT	29.35	29.48	29.12	29.34	41585500	29.34
2012-05-31	AAPL	580.74	581.50	571.46	577.73	17559800	577.73
	DELL	12.53	12.54	12.33	12.33	19955500	12.33
	GOOG	588.72	590.00	579.00	580.86	2968300	580.86
	MSFT	29.30	29.42	28.94	29.19	39134000	29.19
2012-06-01	AAPL	569.16	572.65	560.52	560.99	18606700	560.99
	DELL	12.15	12.30	12.05	12.07	19396700	12.07
	GOOG	571.79	572.65	568.35	570.98	3057900	570.98
	MSFT	28.76	28.96	28.44	28.45	56634300	28.45

DataFrame has a related `to_panel` method, the inverse of `to_frame`:

```
In [304]: stacked.to_panel()
```

```
Out[304]:
```

```
<class 'pandas.core.panel.Panel'>
```

```
Dimensions: 6 (items) x 3 (major) x 4 (minor)
```

```
Items: Open to Adj Close
```

```
Major axis: 2012-05-30 00:00:00 to 2012-06-01 00:00:00
```

```
Minor axis: AAPL to MSFT
```

# Data Loading, Storage, and File Formats

The tools in this book are of little use if you can't easily import and export data in Python. I'm going to be focused on input and output with pandas objects, though there are of course numerous tools in other libraries to aid in this process. NumPy, for example, features low-level but extremely fast binary data loading and storage, including support for memory-mapped array. See [Chapter 12](#) for more on those.

Input and output typically falls into a few main categories: reading text files and other more efficient on-disk formats, loading data from databases, and interacting with network sources like web APIs.

## Reading and Writing Data in Text Format

Python has become a beloved language for text and file munging due to its simple syntax for interacting with files, intuitive data structures, and convenient features like tuple packing and unpacking.

pandas features a number of functions for reading tabular data as a DataFrame object. [Table 6-1](#) has a summary of all of them, though `read_csv` and `read_table` are likely the ones you'll use the most.

*Table 6-1. Parsing functions in pandas*

Function	Description
<code>read_csv</code>	Load delimited data from a file, URL, or file-like object. Use comma as default delimiter
<code>read_table</code>	Load delimited data from a file, URL, or file-like object. Use tab ('\t') as default delimiter
<code>read_fwf</code>	Read data in fixed-width column format (that is, no delimiters)
<code>read_clipboard</code>	Version of <code>read_table</code> that reads data from the clipboard. Useful for converting tables from web pages