Agile Tools for Real World Data

Python for Data Analysis

O'REILLY®

Wes McKinney

Python for Data Analysis

Wes McKinney

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Python for Data Analysis
by Wes McKinney

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

Editor: Julie Steele
Production Editor:
Copyeditor:
Proofreader: FIX ME!

Indexer:
Cover Designer:
Interior Designer: FIX ME!
Illustrator: Robert Romano

Revision History for the :
2012-05-07 Early release revision 1
See *http://oreilly.com/catalog/errata.csp?isbn=9781449319793* for release details.

ISBN: 978-1-449-31979-3

[?]

1336408413

Table of Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

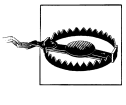
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

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

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

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



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does

require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2011 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

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

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/<catalog page>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

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

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

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

CHAPTER 1
Preliminaries

What is this book about?

This is the book that I wish existed when I started using Python for financial data analysis back in late 2007. But then again, some of the tools you will learn about in this book are ones that did not exist at that time either! I built some of them myself to solve the data challenges I was facing. 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 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) of data
- 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 various keys (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, say, 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 types

TODO DIAGRAM OF A RELATIONAL DATA SET

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, interpreted 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 that term carries with it 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 many others. For example, many of the applications detailed in this book could be just as easily done in R as in Python. So you might be wondering, especially if you're already a proficient R or MATLAB user: what's the point of using Python when the same kinds of tools already exist elsewhere? I can't promise to convince you, but it's my belief that: TODO TODO TODO

- The Python language itself, with its emphasis on readability and code clarity and breadth of general purpose libraries, is a compelling platform for building reusable, extensible software for data analysis and scientific computing
- Python-based software is easy to learn and maintain for both scientists and software engineers alike
- The Python combined with NumPy, SciPy, pandas, and matplotlib, provide the starting point

Get things done, fast

Python is a high level, *high productivity* language. You can accomplish many standard programming tasks using Python in less time and with far fewer lines of code than you may be used to.

Batteries included

*** UNDER CONSTRUCTION ***

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.

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 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 strongly 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.

Embedding Python in Larger Systems

*** UNDER CONSTRUCTION ***

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 areas where Python is 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 written in pure Python. Later in this book we will show how the Cython extension language can be combined with OpenMP, a C framework for parallel computing, to parallelize loops and thus significantly speed up numerical algorithms.

Essential Python Libraries

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, 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 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.

Throughout the book we will use the convention of using the shorthand `np` for referring to NumPy in code, typically achieved by the following import statement:

```
import numpy as np
```

So whenever you see `np` in a code example, that refers to NumPy.

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       3.5   Male    No      Sun  Dinner  3
4   23.68       3.31  Male    No      Sun  Dinner  2
5   24.59       3.61  Female  No      Sun  Dinner  4
6   25.29       4.71  Male    No      Sun  Dinner  4
7   8.77        2     Male    No      Sun  Dinner  2
8   26.88       3.12  Male    No      Sun  Dinner  4
9   15.04       1.96  Male    No      Sun  Dinner  2
10  14.78       3.23  Male    No      Sun  Dinner  2
```

pandas 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.

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 (R) along with some of its add-on toolboxes.

matplotlib

matplotlib is the most widely used Python library for producing plots and other 2D data visualizations. It was originally written by John D. Hunter (JDH) and is now maintained by a large team of developers. It is well-suited for creating production-quality plots. 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 section of the plot and pan around the plot using the toolbar in the plot window: INSERT A PRETTY PLOT HERE

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. I personally do 90% or more of my Python work, including developing and testing code, from within an IPython shell.

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 rich Qt GUI framework-based console with inline plotting
- An infrastructure for interactive parallel and distributed computing

```
TODO INSERT GRAPHIC OF QT CONSOLE
```

We 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.

Installation and Setup

Since everyone uses Python for different applications, there is no single solution for setting up Python on your computer and all the requisite libraries. Even if you use a one-click installer, it's is worth learning how to customize your environment and how to find and install new Python packages.

Getting Python

I imagine since you're reading a book about Python that many of you will have already installed the interpreter and perhaps some of the libraries listed above. But even if you have, there are a number of distributions options to consider:

- Python from python.org: just the interpreter, and nothing else
- EPDFree: a free packaged distribution for all major platforms (Windows, Mac, Linux, Solaris) from Enthought (http://www.enthought.com) with NumPy, SciPy, matplotlib, and IPython. I recommend starting here
- Python(x,y) (`http://pythonxy.googlecode.com`): A free scientific-oriented Python distribution for Windows.
- Enthought Python Distribution: The full distribution from Enthought with access to the subscriber package repository (for routine package updates). A kitchen sink scientific Python distribution with higher performance linear algebra using the Intel MKL. Currently free to academic (with a .edu e-mail address) users but available to commercial organizations and individuals for an annual licenseing fee.
- ActiveState Python: a packaged distribution similar to EPD but with less of a scientific computing focus. Also good for larger organizations which need to manage Python distributions on many desktops and servers
- Linux package repositories. For Linux users, this will depend on your distribution and package management system.

Using EPDFree or Python(x,y) are good options as they are free and come bundled with many of the libraries we will use in this book.

Installing Python Packages

Whether or not you choose to use a packaged distribution, you will need to know how to install and upgrade libraries yourself. The tool I recommend using is `pip`, which works simply by:

```
$ pip install pkg_name
```

Similarly, a package can be uninstalled by

```
$ pip uninstall pkg_name
```

`pip` can be obtained from the Python Package Index (http://pypi.python.org), henceforth known as PyPI. Many scientific packages have compiled extensions (written in C or Fortran) and may not be easily installed using `pip`, especially on Windows. Fortunately, many Python library authors make one-click installers available on PyPI.

Packages you'll need

The main packages that we'll use throughout the book are:

- NumPy
- SciPy
- matplotlib
- IPython
- pandas

For some of the applications and other sections you may wish to also install:

- statsmodels: for statistics and econometrics
- scikit-learn: for machine learning
- CVXOPT: for convex optimization
- dateutil: for date/time manipulation
- PyTables: efficient binary data storage with optional compression using the HDF5 format

Getting help

Outside of performing internet searches, the scientific Python mailing lists are generally helpful and responsive to inquiries. Some ones to take a look at are:

- pydata: a Google Group list for questions related to Python for data 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.

Finding new packages

They say the best line of code is the one you don't have to write yourself. Before you set about building a new library, it's worthwhile to make sure that you aren't reinventing the wheel. PyPI is a good place to start as it has over 17000 Python packages as of this

writing. The numerous mailing lists are a good place to ask questions like, "Is there a library that does...?"

Integrated Development Environments (IDEs)

I personally do not use an IDE. 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 is a handful of ones that you can look into:

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

Navigating this book

Naming conventions

Jargon

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

Munge / Munging
: Describes the overall process of manipulating unstructured and / or messy data into a structured and / or clean form. The word has snuck its way into the jargon of many modern day data scientists. Rhymes with the first syllable of "bungie"

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

Python 2 and Python 3

The timing of this book's publication is a bit unfortunate as the Python community is undergoing a 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. This was also an excellent opportunity to fix the mistakes from the previous 15 years of Python language design and development (and maybe make a few new ones along the way!).

I am writing this book with Python 2.7.2 as its basis. This is primarily since the majority of the *scientific* Python community seems to be still using Python 2. The good news is that almost all of the libraries discussed in this book run perfectly fine in Python 3, so you should have no trouble following along if you happen to be using Python 3.1 or later.

Other Python Implementations

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

A Brief History of pandas

The story of how I personally became involved in building data analysis tools for Python begins in early 2008 during my tenure at AQR Capital Management, a quantitative investment management firm. I wrote the first lines of the library that would become pandas in April 2008. At the time, I had a distinct set of requirements that were not well-addressed by any single existing open source tool:

- Data structures with labeled axes supporting automatic or explicit data alignment. This simultaneously prevents common errors resulting from misaligned data and similarly enables easily working with differently-indexed data coming from different sources.
- Ability to easily combine differently-indexed data series into a tabular structure.
- The same data structures could handle either time series data or non-time series labeled data.
- Arithmetic operations and reductions (like summing across an axis) would pass on the metadata (axis labels).
- Robust handling of missing data.
- Integrated time series functionality.
- Data manipulations similar to relational databases (like SQL).

Each of these requirements could have been independently addressed by various existing tools, but there was no integrated solution combining all of these capabilities into a single consistent set of data structures. I also wanted to do these things in Python, a highly productive language that is also a joy to program in. Thus, I set about building "the data structures of my dreams".

Over the last 4 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 data analysis tool as I do.

CHAPTER 2

Whetting your appetite

This book is focused on providing you with the tools that you will need to work productively with data in Python. 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, slicing, dicing, and transforming data for analysis
- 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 couple of data sets that will be used in the book for illustration purposes. 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. These examples are just intended to give you a flavor of the kinds of things we'll be looking at and thus will only be explained at a high level. In the code examples you'll see input and output prompts like `In [15]:`; these are from the IPython shell.

Example: 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.

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

```
In [15]: path = '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\\/l\\/7AQEFzjSi\\/1.usa.gov\\/wfLQtf", "u":
"http:\\/\\/www.ncbi.nlm.nih.gov\\/pubmed\\/22415991", "t": 1331923247, "hc":
1331822918, "cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }\n'
```

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

If you've never programmed in Python before, 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/l/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'}
```

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.

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/Dropbox/book/svn/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. 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 standrad 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
```

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

Counting time zones with pandas

pandas is a rich data manipulation library for Python which is the primary tool used in this book. 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 [12]: from pandas import DataFrame, Series
```

```
In [13]: frame = DataFrame(records)

In [14]: frame
Out[14]:
<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 [15]: frame['tz'][:10]
Out[15]:
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 [16]: counts = frame['tz'].value_counts()

In [17]: counts[:10]
Out[17]:
America/New_York       1251
                        521
America/Chicago         400
America/Los_Angeles     382
America/Denver          191
Europe/London            74
```

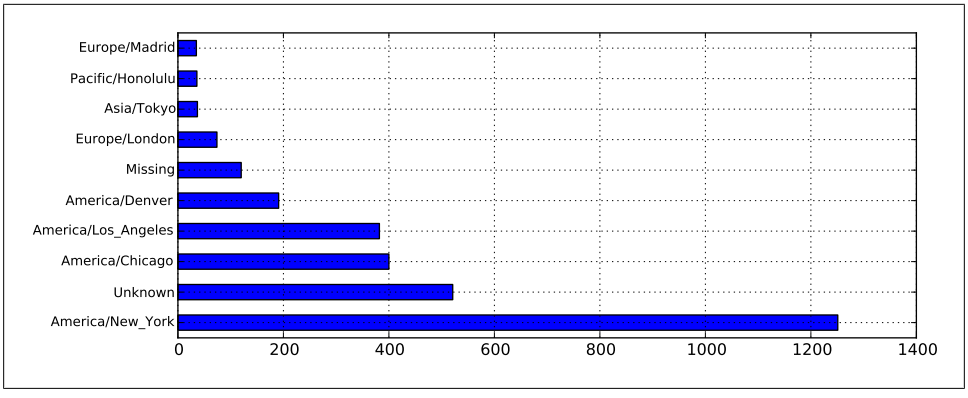


Figure 2-1. Top time zones in the 1.usa.gov sample data

```
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 a comparison and boolean array indexing:

```
In [18]: clean_tz = frame['tz'].fillna('Missing')

In [19]: clean_tz[clean_tz == ''] = 'Unknown'

In [20]: counts = clean_tz.value_counts()

In [21]: counts[:10]
Out[21]:
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 [23]: 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 [24]: frame['a'][1]
Out[24]: u'GoogleMaps/RochesterNY'

In [25]: frame['a'][50]
Out[25]: u'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Firefox/10.0.2'

In [26]: frame['a'][51]
Out[26]: u'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P925/V10e Build/FRG83G) AppleWebKit/533.1 (KH
```

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

In [28]: results[:5]
Out[28]:
0               Mozilla/5.0
1    GoogleMaps/RochesterNY
2               Mozilla/4.0
3               Mozilla/5.0
4               Mozilla/5.0

In [29]: results.value_counts()[:8]
Out[29]:
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
```

Example: MovieLens 1M data set

CHAPTER 3
Python Language Essentials

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

People often ask me about good resources for learning Python for data-centric applications. While there are many excellent Python language books, I am usually hesitant to recommend some of them as they are intended for a general audience rather than tailored for someone who wants to load in some data sets, do some computations, and plot some of the results. There are actually a couple of books on "scientific programming in Python", but they are geared toward numerical computing and engineering applications: solving differential equations, computing integrals, doing Monte Carlo simulations, and various topics that are more mathematically-oriented rather than being about data analysis and statistics. As this is a book is about becoming proficient at working with data in Python, I think it is valuable to spend some time highlighting the most important features of the Python's built-in data structures and libraries from the perspective of processing and manipulating structured and unstructured data. As such, I will only present roughly enough information to enable you to follow along with the rest of the book.

This chapter is not intended to be a self-contained tutorial for new Python programmers but rather a biased, no-frills overview of features which will be used repeatedly throughout this book. For new Python programmers, I recommend that you supplement this chapter with the official Python tutorial (*http://docs.python.org*) and potentially one of the many excellent (and much longer) books on general purpose Python programming. In my opinion, it is *not* necessary to become a proficient Python *software developer* in order to be a proficient Python *data analyst*. Especially after learning about the IPython console in the next chapter, the reader is encouraged to experiment with the code examples and explore the documentation for the various types, functions, and methods. Note that some of the code used in the examples may not necessarily be fully-introduced at this point.

Later in the book I will focus on much higher performance array-based computing tools for working with large data sets in Python. In order to use those tools you must often

first do some munging to corral messy data into a more nicely structured form. Fortunately, Python is one of the easiest-to-use languages for rapidly whipping your data into shape. The greater your facility with Python, the language, the easier it will be for you to prepare new data sets for analysis.

Readers already very familiar with the Python language can safely move on to the next chapter.

The Python interpreter

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

```
$ python
Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul  3 2011, 15:17:51)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)] on linux2
Type "packages", "demo" or "enthought" for more information.
>>> a = 5
>>> print a
5
```

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

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

```
print 'Hello world'
```

This can be run from the terminal simple as:

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

While many Python programmers execute all of their Python code in this way, many *scientific* Python programmers make use of IPython, an enhanced interactive Python interpreter. We will discuss IPython in great detail in the next chapter and use it throughout the book.

```
$ ipython
Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul  3 2011, 15:17:51)
Type "copyright", "credits" or "license" for more information.

IPython 0.12 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
```

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

In [2]:
```

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

Language Semantics

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

Indentation, not braces

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

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

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

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

One major reason that whitespace matters is that it results in most Python code looking cosmetically similar, which means less cognitive dissonance when you read a piece of code that you didn't write yourself (or wrote in a hurry a year ago!). In a language without significant whitespace, you might stumble on some differently formatted code like:

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

```
        }
    }
```

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

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

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

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

Putting multiple statements on one line is generally discouraged in Python as it often makes code less readable. It may come in handy at times, though.

Everything is an object

An important characteristic of the Python language is the consistency of its *object model*. Every number, string, data structure, function, class, module, and so on exists in the Python interpreter in its own "box" which is referred to as a *Python object*. Each object has an associated *type* (for example, *string* or *function*) and internal data. In practice this makes the language very flexible, as even functions can be treated just like any other object.

Comments

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

```
results = []
for line in file_handle:
    # keep the empty lines for now
    # if len(line) == 0:
    #   continue

    results.append(line.replace('foo', 'bar'))
```

Most text editors designed for programming have a feature to easily comment out selected blocks of code by pressing a keyboard shortcut. Usually there is a related keyboard shortcut to *uncomment* code also. Take a moment to figure out how to do this in your text editor of choice.

Function and object method calls

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

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

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

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

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

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

More on this later.

Variables and pass-by-reference

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

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

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

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

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

```
In [3]: a.append(4)

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

Understanding the semantics of references in Python and when, how, and why data is copied is especially critical when working with larger data sets in Python. I will address this topic periodically throughout the book.

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

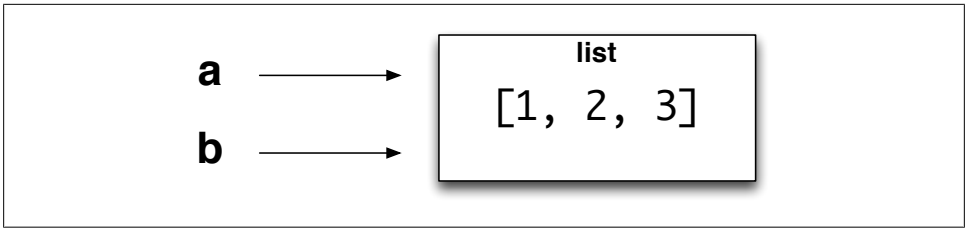


Figure 3-1. Two references for the same object

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

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

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

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

In [3]: append_element(data, 4)

In [4]: data
Out[4]: [1, 2, 3, 4]
```

Dynamic references, strong types

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

```
In [5]: a = 5

In [6]: type(a)
Out[6]: int

In [7]: a = 'foo'

In [8]: type(a)
Out[8]: str
```

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

```
In [9]: '5' + 5
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/home/wesm/Dropbox/book/svn/<ipython-input-9-f9dbf5f0b234> in <module>()
----> 1 '5' + 5
TypeError: cannot concatenate 'str' and 'int' objects
```

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

```
In [10]: a = 4.5

In [11]: b = 2

In [12]: print 'a is %s, b is %s' % (type(a), type(b))
a is <type 'float'>, b is <type 'int'>

In [13]: a / b
Out[13]: 2.25
```

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

```
In [14]: a = 5

In [15]: isinstance(a, int)
Out[15]: True
```

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

```
In [16]: a = 5; b = 4.5

In [17]: isinstance(a, (int, float))
Out[17]: True

In [18]: isinstance(b, (int, float))
Out[18]: True
```

Attributes and methods

Objects in Python typically have both attributes, other Python objects stored "inside" the object, and methods, functions associated with an object which can have access to the object's internal data. They are all accessed via *`obj.attribute_name`*:

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

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

```
a.expandtabs  a.isspace     a.replace     a.splitlines
a.find        a.istitle     a.rfind       a.startswith
```

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

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

While we will not extensively use the functions `getattr` and related functions `hasattr` and `setattr` in this book, they can be used very effectively to write generic, reusable code.

"Duck" typing

Often you may not care about the type of an object but rather only whether it has certain methods or behavior. For example, you can verify that an object is iterable if it implemented the *iterator protocol*, i.e. the `__iter__` "magic method":

```
In [19]: def isiterable(obj):
   ....:     return hasattr(obj, '__iter__')
```

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

```
In [20]: isiterable('a string')
Out[20]: False

In [21]: isiterable([1, 2, 3])
Out[21]: True

In [22]: isiterable(5)
Out[22]: False
```

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

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

Imports

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

```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2
```

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

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

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

Or equivalently:

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

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

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

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

We'll look at modules, packages, and code organization in more detail later.

Binary operators and comparisons

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

```
In [23]: 5 - 7
Out[23]: -2

In [24]: 12 + 21.5
Out[24]: 33.5

In [25]: 5 <= 2
Out[25]: False
```

See Table 3-1 for all of the available binary operators.

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

```
In [26]: a = [1, 2, 3]

In [27]: b = a

In [28]: c = list(a)

In [29]: a is b
Out[29]: True

In [30]: a is not c
Out[30]: True
```

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

```
In [31]: a == c
Out[31]: True
```

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

```
In [32]: a = None

In [33]: a is None
Out[33]: True
```

Table 3-1. Binary operators

Operation	Description
`a + b`	Add a and b
`a - b`	Subtract b from a
`a * b`	Multiply a by b
`a / b`	Divide a by b
`a // b`	Floor-divide a by b, dropping any fractional remainder
`a ** b`	Raise a to the b power
`a & b`	`True` if both a and b are `True`. For integers, take the bitwise AND.
`a | b`	`True` if either a or b is `True`. For integers, take the bitwise OR.
`a ^ b`	For booleans, `True` if a or b is `True`, but not both. For integers, take the bitwise EXCLUSIVE-OR.
`a == b`	`True` if a equals b
`a != b`	`True` if a is not equal to b
`a <= b, a < b`	`True` if a is less than (less than or equal) to b
`a > b, a >= b`	`True` if a is greater than (greater than or equal) to b
`a is b`	`True` if a and b reference same Python object
`a is not b`	`True` if a and b reference different Python objects

Eagerness versus laziness

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

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

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

immediately. Even in the above simple expression, the result of `b * c` is computed as a separate step before adding it to `a`.

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

Mutable and immutable objects

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

```
In [34]: a_list = ['foo', 2, [4, 5]]

In [35]: a_list[2] = (3, 4)

In [36]: a_list
Out[36]: ['foo', 2, (3, 4)]
```

Others, like strings and tuples, are immutable:

```
In [37]: a_tuple = (3, 5, (4, 5))

In [38]: a_tuple[1] = 'four'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/home/wesm/Dropbox/book/svn/<ipython-input-38-b7966a9ae0f1> in <module>()
----> 1 a_tuple[1] = 'four'
TypeError: 'tuple' object does not support item assignment
```

Remember that just because you *can* mutate an object does not mean that you always *should*. Such actions are known in programming as *side effects*. For example, when writing a function, any side effects should be explicitly communicated to the user in the function's documentation or comments. In general, I try to avoid side effects and *favor immutability*, even though there may be mutable objects involved. We'll revisit this topic throughout the book.

The Zen of Python

The ethos of the Python language and its community is often summarized by *The Zen of Python*, a list of guiding principles frequently cited to resolve Python software design disputes:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
```

```
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Some of these statements may not make sense out of context, but it is occasionally helpful to revisit them from time to time.

Scalar Types

Python has a small set of built-in types for handling numerical data, strings, boolean (`True` or `False`) values, and dates and time. See Table 3-2 for a list of the main scalar types. Date and time handling will be discussed separately as these are provided by the `datetime` module in the standard library.

Table 3-2. Standard Python Scalar Types

Type	Description
None	The Python "null" value (only one None exists)
`str`	String type. ASCII-valued only in Python 2.x and Unicode in Python 3
`unicode`	Unicode string type
`float`	Double-precision (64-bit) floating point number. Note there is no separate `double` type.
`bool`	A `True` or `False` value
`int`	Signed integer with maximum value determined by the platform.
`long`	Arbitrary precision signed integer. Large `int` values are seamlessly converted to `long`.

Numeric types

The primary Python types for numbers are `int` and `float`. The size of the integer which can be stored as an `int` is dependent on your platform (whether 32 or 64-bit), but Python will transparently convert a very large integer to `long`, which can store arbitrarily large integers.

```
In [39]: ival = 17239871
```

```
In [40]: ival ** 6
Out[40]: 26254519291092456596965462913230729701102721L
```

Decimal numbers are represented with the Python `float` type. Under the hood it is a double-precision (64-bit) floating point number. They can also be expressed using scientific notation:

```
In [41]: fval = 7.243

In [42]: fval2 = 6.78e-5
```

In Python 3, integer division not resulting in a whole number will always yield a floating point number:

```
In [44]: 3 / 2
Out[44]: 1.5
```

In Python 2.7 and below (which some readers will likely be using), you can enable this behavior by default by putting the following cryptic-looking statement at the top of your module:

```
from __future__ import division
```

Without this in place, you can always explicitly convert the denominator into a floating point number:

```
In [45]: 3 / float(2)
Out[45]: 1.5
```

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

```
In [46]: 3 // 2
Out[46]: 1
```

Complex numbers are written using `j` for the imaginary part:

```
In [47]: cval = 1 + 2j

In [48]: cval * (1 - 2j)
Out[48]: (5+0j)
```

Strings

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

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

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

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

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

```
In [49]: a = 'this is a string'

In [50]: a[10] = 'f'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/home/wesm/Dropbox/book/svn/<ipython-input-50-5ca625d1e504> in <module>()
----> 1 a[10] = 'f'
TypeError: 'str' object does not support item assignment

In [51]: b = a.replace('string', 'longer string')

In [52]: b
Out[52]: 'this is a longer string'
```

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

```
In [53]: a = 5.6

In [54]: s = str(a)

In [55]: s
Out[55]: '5.6'
```

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

```
In [56]: s = 'python'

In [57]: list(s)
Out[57]: ['p', 'y', 't', 'h', 'o', 'n']

In [58]: s[:3]
Out[58]: 'pyt'
```

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

```
In [59]: s = '12\\34'

In [60]: print s
12\34
```

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

```
In [61]: s = r'this\has\no\special\characters'

In [62]: s
Out[62]: 'this\\has\\no\\special\\characters'
```

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

```
In [63]: a = 'this is the first half '

In [64]: b = 'and this is the second half'

In [65]: a + b
Out[65]: 'this is the first half and this is the second half'
```

I will discuss advanced string processing and formatting in more detail later.

Table 3-3. String methods

Method(s)	Description
count	
startswith	
endswith	
upper	
lower	
title	
capitalize	
partition, rpartition	
split, rsplit	
strip, lstrip, rstrip	
format	
join	
replace	

Booleans

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

```
In [66]: True and True
Out[66]: True

In [67]: False or True
Out[67]: True
```

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

```
In [68]: a = [1, 2, 3]
   ....: if a:
   ....:     print 'I found something!'
   ....:
I found something!

In [69]: b = []
```

```
   ....: if not b:
   ....:     print 'Empty!'
   ....:
Empty!
```

Type casting

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

```
In [70]: s = '3.14159'

In [71]: fval = float(s)

In [72]: type(fval)
Out[72]: float

In [73]: int(fval)
Out[73]: 3

In [74]: bool(fval)
Out[74]: True

In [75]: bool(0)
Out[75]: False
```

None

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

```
In [76]: a = None

In [77]: a is None
Out[77]: True

In [78]: b = 5

In [79]: b is not None
Out[79]: True
```

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

```
def add_and_maybe_multiply(a, b, c=None):
result = a + b

if c is not None:
    result = result * c

return result
```

Dates and times

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

```
In [80]: from datetime import datetime, date, time

In [81]: dt = datetime(2011, 10, 29, 20, 30, 21)

In [82]: dt.day
Out[82]: 29

In [83]: dt.minute
Out[83]: 30
```

Given a `datetime` instance, you can extract the equivalent `date` and `time` objects by calling methods on the `datetime` of the same name:

```
In [84]: dt.date()
Out[84]: datetime.date(2011, 10, 29)

In [85]: dt.time()
Out[85]: datetime.time(20, 30, 21)
```

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

```
In [86]: dt.strftime('%m/%d/%Y %H:%M')
Out[86]: '10/29/2011 20:30'
```

```
TODO DATETIME FORMAT TABLE
```

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

```
In [87]: datetime.strptime('20091031', '%Y%m%d')
Out[87]: datetime.datetime(2009, 10, 31, 0, 0)
```

When binning time series data, it will occasionally be useful to replace fields of a series of datetimes, for example replacing the minute and second fields with zero, producing a new object:

```
In [88]: dt.replace(minute=0, second=0)
Out[88]: datetime.datetime(2011, 10, 29, 20, 0)
```

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

```
In [89]: dt2 = datetime(2011, 11, 15, 22, 30)

In [90]: delta = dt2 - dt

In [91]: delta
Out[91]: datetime.timedelta(17, 7179)

In [92]: type(delta)
Out[92]: datetime.timedelta
```

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

```
In [93]: dt
Out[93]: datetime.datetime(2011, 10, 29, 20, 30, 21)

In [94]: dt + delta
Out[94]: datetime.datetime(2011, 11, 15, 22, 30)
```

Control Flow

If, elif, and else

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

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

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

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

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

```
In [95]: a = 5; b = 7

In [96]: c = 8; d = 4

In [97]: if a < b and c > d:
   ....:     print 'Made it'
Made it
```

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

For loops

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

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

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

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

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

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

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

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

While loops

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

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

pass

`pass` is the "no-op" statement in Python. It can be used in blocks where no action is to be taken:

```
if x < 0:
    print 'negative!'
elif x == 0:
    # TODO: put something smart here
    pass
else:
    print 'positive!'
```

It's common to use `pass` as a place-holder in code while working on a new piece of functionality:

```
def f(x, y, z):
    # TODO: implement this function!
    pass
```

range and xrange

The `range` function produces a list of evenly-spaced integers:

```
In [98]: range(10)
Out[98]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

```
In [99]: range(0, 20, 2)
Out[99]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

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

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

For very long ranges, it's recommended to use `xrange`, which takes the same arguments as `range` but returns an iterator that generates integers one by one rather than generating all of them up-front and storing them in a (potentially very large) list. This snippet sums all numbers from 0 to 9999 that are multiples of 3 or 5:

```
sum = 0
for i in xrange(10000):
    # % is the modulo operator
    if x % 3 == 0 or x % 5 == 0:
        sum += i
```

In Python 3, `range` always returns an iterator, and thus it is not necessary to use the `xrange` function

Ternary Expressions

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

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

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

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

This is a more concrete example:

```
In [100]: x = 5

In [101]: 'Non-negative' if x >= 0 else 'Negative'
Out[101]: 'Non-negative'
```

As with `if-else` blocks, only one of the expressions will be evaluated. While it may be tempting to always use ternary expressions to condense your code, realize that you may sacrifice readability if the condition as well and the true and false expressions are very complex.

Tuple

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

```
In [102]: tup = 4, 5, 6

In [103]: tup
Out[103]: (4, 5, 6)
```

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

```
In [104]: nested_tup = (4, 5, 6), (7, 8)

In [105]: nested_tup
Out[105]: ((4, 5, 6), (7, 8))
```

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

```
In [106]: tuple([4, 0, 2])
Out[106]: (4, 0, 2)

In [107]: tup = tuple('string')

In [108]: tup
Out[108]: ('s', 't', 'r', 'i', 'n', 'g')
```

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

```
In [109]: tup[0]
Out[109]: 's'
```

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

```
In [110]: tup = tuple(['foo', [1, 2], True])

In [111]: tup[2] = False
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/home/wesm/Dropbox/book/svn/<ipython-input-111-c7308343b841> in <module>()
----> 1 tup[2] = False
TypeError: 'tuple' object does not support item assignment

# however
In [112]: tup[1].append(3)

In [113]: tup
Out[113]: ('foo', [1, 2, 3], True)
```

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

```
In [114]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[114]: (4, None, 'foo', 6, 0, 'bar')
```

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

```
In [115]: ('foo', 'bar') * 4
Out[115]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

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

Unpacking tuples

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

```
In [116]: tup = (4, 5, 6)

In [117]: a, b, c = tup

In [118]: b
Out[118]: 5
```

Even sequences with nested tuples can be unpacked:

```
In [119]: tup = 4, 5, (6, 7)

In [120]: a, b, (c, d) = tup

In [121]: d
Out[121]: 7
```

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

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

b, a = a, b
```

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

```
seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
for a, b, c in seq:
    pass
```

Another common use is for returning multiple values from a function. More on this later.

Tuple methods

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

```
In [122]: a = (1, 2, 2, 2, 3, 4, 2)

In [123]: a.count(2)
Out[123]: 4
```

List

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

```
In [124]: a_list = [2, 3, 7, None]

In [125]: tup = ('foo', 'bar', 'baz')

In [126]: b_list = list(tup)

In [127]: b_list
Out[127]: ['foo', 'bar', 'baz']

In [128]: b_list[1] = 'peekaboo'

In [129]: b_list
Out[129]: ['foo', 'peekaboo', 'baz']
```

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

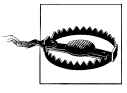
Adding and removing elements

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

```
In [130]: b_list.append('dwarf')

In [131]: b_list
Out[131]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

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

```
In [132]: b_list.insert(1, 'red')

In [133]: b_list
Out[133]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```



`insert` is computationally expensive compared with `append` as subsequent elements have to be shifted (in the internal array of objects) to make room for the new element.

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

```
In [134]: b_list.pop(2)
Out[134]: 'peekaboo'

In [135]: b_list
Out[135]: ['foo', 'red', 'baz', 'dwarf']
```

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

```
In [136]: b_list.append('foo')

In [137]: b_list.remove('foo')

In [138]: b_list
Out[138]: ['red', 'baz', 'dwarf', 'foo']
```

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

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

```
In [139]: 'dwarf' in b_list
Out[139]: True
```

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

Concatenating and combining lists

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

```
In [140]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[140]: [4, None, 'foo', 7, 8, (2, 3)]
```

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

```
In [141]: x = [4, None, 'foo']
```

```
In [142]: x.extend([7, 8, (2, 3)])

In [143]: x
Out[143]: [4, None, 'foo', 7, 8, (2, 3)]
```

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

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

is faster than than the concatenative alternative

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

Sorting

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

```
In [144]: a = [7, 2, 5, 1, 3]

In [145]: a.sort()

In [146]: a
Out[146]: [1, 2, 3, 5, 7]
```

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

```
In [147]: b = ['saw', 'small', 'He', 'foxes', 'six']

In [148]: b.sort(key=len)

In [149]: b
Out[149]: ['He', 'saw', 'six', 'small', 'foxes']
```

Binary search and maintaining a sorted list

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

```
In [150]: import bisect

In [151]: c = [1, 2, 2, 2, 3, 4, 7]

In [152]: bisect.bisect(c, 2)
Out[152]: 4
```

```
In [153]: bisect.bisect(c, 5)
Out[153]: 6

In [154]: bisect.insort(c, 6)

In [155]: c
Out[155]: [1, 2, 2, 2, 3, 4, 6, 7]
```

Slicing

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

```
In [156]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [157]: seq[1:5]
Out[157]: [2, 3, 7, 5]
```

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

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

```
In [158]: seq[:5]
Out[158]: [7, 2, 3, 7, 5]

In [159]: seq[3:]
Out[159]: [7, 5, 6, 0, 1]
```

Negative indices slice the sequence relative to the end:

```
In [160]: seq[-4:]
Out[160]: [5, 6, 0, 1]

In [161]: seq[-6:-2]
Out[161]: [3, 7, 5, 6]
```

Slicing semantics takes a bit of getting used to, especially if you're coming from R or MATLAB. See Figure 3-2 for a helpful illustrating of slicing with positive and negative integers.

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

```
In [162]: seq[::2]
Out[162]: [7, 3, 5, 0]
```

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

```
In [163]: seq[::-1]
Out[163]: [1, 0, 6, 5, 7, 3, 2, 7]
```

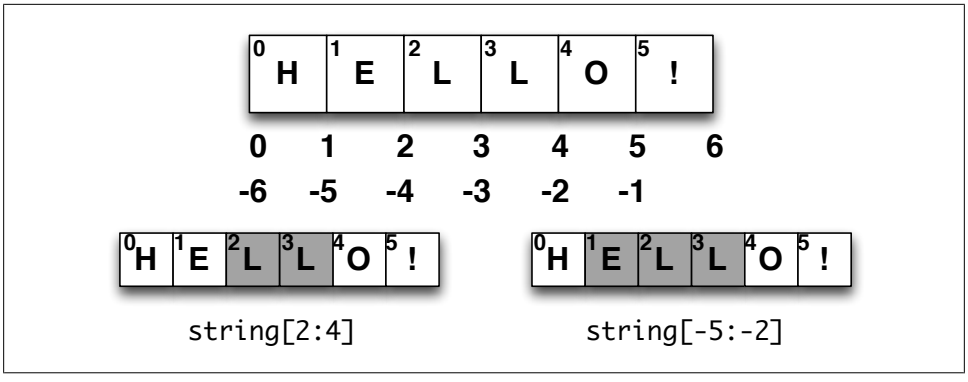


Figure 3-2. Illustration of Python slicing conventions

Built-in Sequence Functions

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

enumerate

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

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

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

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

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

```
In [164]: some_list = ['foo', 'bar', 'baz']

In [165]: mapping = dict((v, i) for i, v in enumerate(some_list))

In [166]: some_list
Out[166]: ['foo', 'bar', 'baz']
```

sorted

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

```
In [167]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[167]: [0, 1, 2, 2, 3, 6, 7]

In [168]: sorted('horse race')
Out[168]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

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

```
In [169]: sorted(set('this is just some string'))
Out[169]: [' ', 'e', 'g', 'h', 'i', 'j', 'm', 'n', 'o', 'r', 's', 't', 'u']
```

zip

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

```
In [170]: seq1 = ['foo', 'bar', 'baz']

In [171]: seq2 = ['one', 'two', 'three']

In [172]: zip(seq1, seq2)
Out[172]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

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

```
In [173]: seq3 = [False, True]

In [174]: zip(seq1, seq2, seq3)
Out[174]: [('foo', 'one', False), ('bar', 'two', True)]
```

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

```
In [175]: for i, (a, b) in enumerate(zip(seq1, seq2)):
   .....:     print('%d: %s, %s' % (i, a, b))
   .....:
0: foo, one
1: bar, two
2: baz, three
```

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

```
In [176]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens')]

In [177]: first_names, last_names = zip(*pitchers)

In [178]: first_names
Out[178]: ('Nolan', 'Roger')

In [179]: last_names
Out[179]: ('Ryan', 'Clemens')
```

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

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

reversed

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

```
In [180]: list(reversed(range(10)))
Out[180]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Dict

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

```
In [181]: empty_dict = {}

In [182]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}

In [183]: d1
Out[183]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

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

```
In [184]: d1[7] = 'an integer'

In [185]: d1
Out[185]: {7: 'an integer', 'a': 'some value', 'b': [1, 2, 3, 4]}

In [186]: d1['b']
Out[186]: [1, 2, 3, 4]
```

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

```
In [187]: 'b' in d1
Out[187]: True
```

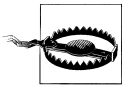
Values can be deleted either using the `del` keyword or the `pop` method (which simultaneously returns the value and deletes the key):

```
In [188]: d1[5] = 'some value'

In [189]: d1['dummy'] = 'another value'

In [190]: del d1[5]

In [191]: ret = d1.pop('dummy')
```

```
In [192]: ret
Out[192]: 'another value'
```

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

```
In [193]: d1.keys()
Out[193]: ['a', 'b', 7]

In [194]: d1.values()
Out[194]: ['some value', [1, 2, 3, 4], 'an integer']
```



If you're using Python 3, `dict.keys()` nad `dict.values()` are iterators instead of lists.

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

```
In [195]: d1.update({'b' : 'foo', 'c' : 12})

In [196]: d1
Out[196]: {7: 'an integer', 'a': 'some value', 'b': 'foo', 'c': 12}
```

Creating dicts from sequences

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

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

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

```
In [197]: seq1 = range(5)

In [198]: seq2 = reversed(range(5))

In [199]: mapping = dict(zip(seq1, seq2))

In [200]: mapping
Out[200]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

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

Default values

It's very common to have logic like:

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

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

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

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

```
In [201]: words = ['apple', 'bat', 'bar', 'atom', 'book']

In [202]: by_letter = {}

In [203]: for word in words:
   .....:     letter = word[0]
   .....:     if letter not in by_letter:
   .....:         by_letter[letter] = [word]
   .....:     else:
   .....:         by_letter[letter].append(word)
   .....:

In [204]: by_letter
Out[204]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

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

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

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

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

The initializer to `defaultdict` only needs to be a function, not necessarily a type. Thus, if you wanted the default value to be 4 you could pass a function returning 4

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

Valid dict key types

While the values of a dict can be any Python object, the keys have to be immutable objects like scalar types (int, float, string) or tuples (all the objects in the tuple need to

be immutable, too). The technical term here is *hashability*. You can check whether an object is hashable (can be used as a key in a dict) with the `hash` function:

```
In [205]: hash('string')
Out[205]: -9167918882415130555

In [206]: hash((1, 2, (2, 3)))
Out[206]: 1097636502276347782

In [207]: hash((1, 2, [2, 3])) # fails because lists are mutable
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/home/wesm/Dropbox/book/svn/<ipython-input-207-800cd14ba8be> in <module>()
----> 1 hash((1, 2, [2, 3])) # fails because lists are mutable
TypeError: unhashable type: 'list'
```

To use a list as a key, an easy fix is to convert it to a tuple:

```
In [208]: d = {}

In [209]: d[tuple([1, 2, 3])] = 5

In [210]: d
Out[210]: {(1, 2, 3): 5}
```

Set

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

```
In [211]: set([2, 2, 2, 1, 3, 3])
Out[211]: set([1, 2, 3])

In [212]: {2, 2, 2, 1, 3, 3}
Out[212]: set([1, 2, 3])
```

Sets support mathematical *set operations* like union, intersection, difference, and symmetric difference. See Table 3-4 for a list of commonly used set methods.

```
In [213]: a = {1, 2, 3, 4, 5}

In [214]: b = {3, 4, 5, 6, 7, 8}

In [215]: a | b  # union (or)
Out[215]: set([1, 2, 3, 4, 5, 6, 7, 8])

In [216]: a & b  # intersection (and)
Out[216]: set([3, 4, 5])

In [217]: a - b  # difference
Out[217]: set([1, 2])

In [218]: a ^ b  # symmetric difference (xor)
Out[218]: set([1, 2, 6, 7, 8])
```

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

```
In [219]: a_set = {1, 2, 3, 4, 5}

In [220]: {1, 2, 3}.issubset(a_set)
Out[220]: True

In [221]: a_set.issuperset({1, 2, 3})
Out[221]: True
```

As you might guess, sets are equal if their contents are equal:

```
In [222]: {1, 2, 3} == {3, 2, 1}
Out[222]: True
```

Table 3-4. Python Set Operations

Function	Alternate Syntax	Description
`a.add(x)`	N/A	Add element x to the set a
`a.remove(x)`	N/A	Remove element x from the set a
`a.union(b)`	`a | b`	All of the unique elements in a and b.
`a.intersection(b)`	`a & b`	All of the elements in *both* a and b.
`a.difference(b)`	`a - b`	The elements in a that are not in b.
`a.symmetric_difference(b)`	`a ^ b`	All of the elements in a or b but *not both*.
`a.issubset(b)`	N/A	`True` if the elements of a are all contained in b.
`a.issuperset(b)`	N/A	`True` if the elements of b are all contained in a.
`a.isdisjoint(b)`	N/A	`True` if a and b have no elements in common.

List, set, and dict comprehensions

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

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

This is equivalent to the following `for` loop:

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

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

```
In [223]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
```

```
In [224]: [x.upper() for x in strings if len(x) > 2]
Out[224]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

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

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

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

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

Like list comprehensions, set and dict comprehensions are just syntactic sugar, but they similarly can make code both easier to write and read. Consider the list of strings above. Suppose we wanted a set containing just the lengths of the strings contained in the collection; this could be easily computed using a set comprehension:

```
In [225]: unique_lengths = {len(x) for x in strings}

In [226]: unique_lengths
Out[226]: set([1, 2, 3, 4, 6])
```

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

```
In [227]: loc_mapping = {val : index for index, val in enumerate(strings)}

In [228]: loc_mapping
Out[228]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Note that this dict could be equivalently be constructed by:

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

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

Nested list comprehensions

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

```
In [229]: all_data = [['Tom', 'Billy', 'Jefferson', 'Andrew', 'Wesley', 'Steven', 'Joe'],
   .....:             ['Susie', 'Casey', 'Jill', 'Ana', 'Eva', 'Jennifer', 'Stephanie']]
```

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

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

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

```
In [230]: result = [name for names in all_data for name in names
   .....:           if name.count('e') >= 2]

In [231]: result
Out[231]: ['Jefferson', 'Wesley', 'Steven', 'Jennifer', 'Stephanie']
```

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

You can have arbitrarily many levels of nesting, though if you have more than 2 or 3 levels of nesting you should probably start to question your data structure design. It's important to distinguish the above syntax from a list comprehension inside a list comprehension, which is also perfectly valid:

```
In [229]: [[x for x in tup] for tup in some_tuples]
```

Functions

Functions are the primary and most important method of code organization and reuse in Python. There may not be such a thing as having too many functions. In fact, I would argue that most programmers doing data analysis don't write enough functions! As you have likely inferred from prior examples, functions are declared using the `def` keyword and returned from using the `return` keyword:

```
def my_function(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

There is no issue with having multiple `return` statements. If the end of a function is reached without encountering a `return` statement, `None` is returned.

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

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

The main restriction on function arguments it that the keyword arguments *must* follow the positional arguments (if any). You can specify keyword arguments in any order; this frees you from having to remember which order the function arguments were specified in and only what their names are.

Namespaces, scope, and local functions

Functions can access variables in two different scopes: *global* and *local*. An alternate and more descriptive name describing a variable scope in Python is a *namespace*. Any variables that are assigned within a function by default are assigned to the local namespace. The local namespace is created when the function is called and immediately populated by the function's arguments. After the function is finished, the local namespace is destroyed (with some exceptions, see section on closures below). Consider the following function:

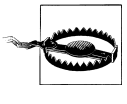
```
def func():
    a = []
    for i in range(5):
        a.append(i)
```

Upon calling `func()`, the empty list `a` is created, 5 elements are appended, then `a` is destroyed when the function exists. Suppose instead we had declared `a`

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

Binding variables in the global namespace is possible, but those variables be declared as global using the `global` keyword:

```
In [232]: a = None

In [233]: def bind_a_variable():
   .....:     global a
   .....:     a = []
   .....: bind_a_variable()
   .....:

In [234]: print a
[]
```



I generally discourage people from using the `global` keyword frequently. Typically global variables are used to store some kind of state in a system. If you find yourself using a lot of them, it's probably a sign that some object-oriented programming (using classes, a bit more on this later) is in order.

Functions can be declared anywhere, and this is no problem with having *local* functions that are dynamically created when a function is called:

```
def outer_function(x, y, z):
    def inner_function(a, b, c):
        pass
    pass
```

In the above code, the the `inner_function` will not exist until `outer_function` is called. As soon as `outer_function` is done executing, the `inner_function` is destroyed.

Nested inner functions can access the local namespace of the enclosing function, but they cannot bind new variables in it. I'll talk a bit more about this in the section on closures.

In a strict sense, all functions are local to some scope, that scope may just be the module level scope.

Returning multiple values

When I first programmed in Python after having programmed in Java and C++ in the past, one of my favorite features was the ability to return multiple values from a function. Here's a simple example:

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

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

In data analysis and other scientific applications, you will likely find yourself doing this very often as many functions may have multiple outputs, whether those are data structures or other auxiliary data computed inside the function. If you think about tuple packing and unpacking from earlier in this chapter, you may realize that what's happening here is that the function is actually just returning *one* object, namely a tuple, which is then being unpacked into the result variables. In the above example, we could have done instead:

```
return_value = f()
```

In this case, `return_value` would be, as you may guess, a 3-tuple with the three returned variables. A potentially attractive alternative to returning multiple values like above might be to return a dict instead:

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

Functions are objects

Since Python functions are objects, many constructs can be easily expressed that are difficult to do in other languages. Suppose we were doing some data cleaning and needed to apply a bunch of transformations to the following list of strings:

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

Anyone who has ever worked with user-submitted survey data can expect messy results like these. Lots of things need to happen to make this list of strings uniform and ready for analysis: whitespace stripping, removing punctuation symbols, and proper capitalization. As a first pass, we might write some code like:

```
import re  # Regular expression module

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

The result looks like this:

```
In [15]: clean_strings(states)
Out[15]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

An useful alternate approach that you may find useful is to make a list of the operations you want to apply to a particular set of strings:

```
def remove_punctuation(value):
    return re.sub('[!#?]', '', value)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result
```

Then we have

```
In [22]: clean_strings(states, clean_ops)
Out[22]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

A more *functional* pattern like this enables you to easily modify how the strings are transformed at a very high level. The `clean_strings` function is also now more reusable!

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

```
In [23]: map(remove_punctuation, states)
Out[23]:
['   Alabama ',
 'Georgia',
 'Georgia',
 'georgia',
 'FlOrIda',
 'south   carolina',
 'West virginia']
```

Anonymous (lambda) functions

Python has support for so-called *anonymous* or *lambda* functions, which are really just simple functions consisting of a single statement, the result of which is the return value. They are defined using the `lambda` keyword, which has no meaning other than "we are declaring an anonymous function."

```
def short_function(x):
    return x * 2

equiv_anon = lambda x: x * 2
```

I'll usually refer to these as lambda functions through the rest of the book. They are especially convenient in data analysis because, as you'll see, there are many cases where data transformation functions will take functions as arguments. It's often less typing (and clearer) to pass a lambda function as opposed to writing a full-out function declaration or even assigning the lambda function to a local variable. For example, consider this silly example:

```
def apply_to_list(some_list, f):
    return [f(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

Obvious in this case, you would ordinarily have written ``[x * 2 for x in ints]``, but here we were able to succintly pass a custom operator to the `apply_to_list` function. As you'll see later, this is very common in practice.

The main reason lambda functions are called anonymous functions is that the function object itself is never given a name attribute.

Closures: functions that return functions

Closures are nothing to fear. They can actually be a very useful and powerful tool in the right circumstance! In a nutshell, a closure is any *dynamically-generated* function returned by another function. Here is a very simple example:

```
def make_closure(a):
    def closure():
        print('I know the secret: %d' % a)
    return closure

closure = make_closure(5)
```

The difference between a closure and a regular Python function is that the closure continues to have access to the namespace (the function) where it was created, even though that function is done executing. So in the above case, the returned closure will always print `I know the secret: 5` whenever you call it. While it's common to create closures whose internal state (in this example, only the value of `a`) is static, you can just as easily have a mutable object like a dict, set, or list that can be modified:

```
def make_
```

However, one technical limitation to keep in mind is that while you can mutate any internal state objects (like adding key-value pairs to a dict), you cannot *bind* variables in the enclosing function scope. One way to work around this is to modify a dict or list rather than binding variables:

```
def make_counter():
    count = [0]
    def counter():
        # increment and return the current count
        count[0] += 1
        return count[0]
    return counter

counter = make_counter()
```

You might be wondering why this is useful. The basic story is that you can use this capability to *manufacture* functions that both

```
TODO TODO
```

"Extended call" syntax with *args, **kwargs

The way that function arguments work under the hood in Python is actually very simple. When you write `func(a, b, c, d=some, e=value)`, the positional and keyword arguments are actually packed up into a tuple and dict, respectively. So the internal function receives a tuple `args` and dict `kwargs` and immediate does:

```
a, b, c = args
d = kwargs.get('d', d_default_value)
e = kwargs.get('e', e_default_value)
```

This all happens nicely behind the scenes. Of course, it also does some error checking and allows you to specify some of the positional arguments as keywords also (even if they aren't keyword in the function declaration!).

```
def say_hello_then_call_f(f, *args, **kwargs):
    print 'args is', args
    print 'kwargs is', kwargs
    print("Hello! Now I'm going to call %s" % f)
    return f(*args, **kwargs)

def g(x, y, z=1):
    return (x + y) / z
```

Then if we call `g` with `say_hello_then_call_f` we get:

```
In [8]:  say_hello_then_call_f(g, 1, 2, z=5.)
args is (1, 2)
kwargs is {'z': 5.0}
Hello! Now I'm going to call <function g at 0x2dd5cf8>
Out[8]: 0.6
```

Currying: partial argument application

Currying is a fun computer science term which means deriving new functions from existing ones by *partial argument application*. For example, suppose we had a trivial function that adds two numbers together:

```
def add_numbers(x, y):
    return x + y
```

Using this function, we could derive a new function of one variable, `add_five`, that adds 5 to its argument:

```
add_five = lambda y: add_numbers(5, y)
```

The second argument to `add_numbers` is said to be *curried*. There's nothing very fancy here as we really only have defined a new function that calls an existing function. The built-in `functools` module can simplify this process using the `partial` function:

```
from functools import partial
add_five = partial(add_numbers, 5)
```

The name "curry" comes from Haskell Curry, a well-known logician, whose first name was also adopted for the Haskell pure functional programming language

Later, when discussing pandas and time series data, we'll use this technique to create specialized functions for transforming data series

```
# compute 60-day moving average of time series x
ma60 = lambda x: pandas.rolling_mean(x, 60)
```

```
# Take the 60-day moving average of of all time series in data
data.apply(ma60)
```

Files and the operating system

In most of this book I'll be using high level tools like `pandas.read_csv` to read data files from disk into Python data structures. However, it's important to understand the basics of how to work with files in Python. Fortunately, it's very simple, which is part of why Python is so popular for text and file munging.

To open a file for reading or writing, use the built-in `open` function with either a relative or absolute file path:

```
In [31]: f = open('prof_mod.py')
```

By default, the file is opened in read-only mode `'r'`. We can then treat the file handle `f` like a list and iterate over the lines like so

```
In [34]: for line in f:
    print line,
   ....:
from numpy.random import randn

def add_and_sum(x, y):
    added = x + y
    summed = added.sum(axis=1)
    return summed

def call_function():
    x = randn(1000, 1000)
    y = randn(1000, 1000)
    return add_and_sum(x, y)
```

The lines come out of the file with the end-of-line (EOL) markers intact, so you'll often see code to get an EOL-free list of lines in a file like

```
In [39]: lines = [x.rstrip() for x in open('prof_mod.py')]

In [40]: lines
Out[40]:
['from numpy.random import randn',
 '',
 'def add_and_sum(x, y):',
 'added = x + y',
 'summed = added.sum(axis=1)',
 'return summed',
 '',
 'def call_function():',
 'x = randn(1000, 1000)',
 'y = randn(1000, 1000)',
 'return add_and_sum(x, y)']
```

If we had typed `f = open('prof_mod.py', 'w')`, a *new file* `prof_mod.py` would have been created, overwriting any one in its place.

Table 3-5. Python file methods

Method	Description
`read([size])`	Return data from file as a string, with optional `size` argument indicating the number of bytes to read
`readlines([size])`	Return list of lines in the file, with optional `size` argument
`readlines([size])`	Return list of lines (as strings) in the file
`close`	Close the handle
`flush`	Flush the internal I/O buffer to disk

Table 3-6. Python file modes

Mode	Description
`r`	Read-only mode
`w`	Write-only mode. Creates a new file (deleting any file with the same name)
`a`	Append to existing file (create it if it does not exist)
`r+`	Read and write
`b`	Add to mode for binary files, that is `'rb'` or `'wb'`
`U`	Use universal newline mode. Pass by itself `'U'` or appended to one of the read modes like `'rU'`

Built-in `csv` module

Generators

Having a consistent way to iterate over sequences, like objects in a list or lines in a file, is an important Python feature. This is accomplished by means of the *iterator protocol*, a generic way to make objects iterable. For example, iterating over a dict yields the dict keys:

```
In [235]: some_dict = {'a': 1, 'b': 2, 'c': 3}

In [236]: for key in some_dict:
   .....:     print key,
a c b
```

When you write `for key in some_dict`, the Python interpreter first attempts to create an iterator out of `some_dict`:

```
In [237]: dict_iterator = iter(some_dict)

In [238]: dict_iterator
Out[238]: <dictionary-keyiterator at 0x2b33e10>
```

The important feature about an iterator is its `next()` method which produces each element. Most methods expecting a list or list-like object will also accept any iterable

object. This includes built-in methods such as `min`, `max`, and `sum`, and type constructors like `list` and `tuple`:

```
In [239]: list(dict_iterator)
Out[239]: ['a', 'c', 'b']
```

A *generator* is a simple way to construct a new iterable object. Whereas normal functions execute and return a single value, generators lazily return a sequence of values, pausing after each one until the next one is requested. To create a generator, use the `yield` keyword instead of `return` in a function:

```
def squares(n=10):
    for i in xrange(1, n + 1):
        print 'Generating squares from 1 to %d' % (n ** 2)
        yield i ** 2
```

When you actually call the generator, no code is immediately executed:

```
In [2]: gen = squares()

In [3]: gen
Out[3]: <generator object squares at 0x34c8280>
```

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

```
In [4]: for x in gen:
   ...:     print x,
   ...:
Generating squares from 0 to 100
1 4 9 16 25 36 49 64 81 100
```

As a less trivial example, suppose we wished to find all unique ways to make change for $1 (100 cents) using an arbitrary set of coins. You can probably think of various ways to implement this and how to store the unique combinations as you come up with them. One way is to write a generator that yields lists of coins (represented as integers):

```
def make_change(amount, coins=[1, 5, 10, 25], hand=None):
    hand = [] if hand is None else hand
    if amount == 0:
        yield hand
    for coin in coins:
        # ensures we don't give too much change, and combinations are unique
        if coin > amount or (len(hand) > 0 and hand[-1] < coin):
            continue

        for result in make_change(amount - coin, coins=coins,
                                  hand=hand + [coin]):
            yield result
```

The details of the algorithm are not that important (can you think of a shorter way?). Then we can write:

```
In [241]: for way in make_change(100, coins=[10, 25, 50]):
   .....:     print way
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
[25, 25, 10, 10, 10, 10, 10]
```

```
[25, 25, 25, 25]
[50, 10, 10, 10, 10, 10]
[50, 25, 25]
[50, 50]

In [242]: len(list(make_change(100)))
Out[242]: 242
```

Why care about generators?

The itertools module

Table 3-7. Some itertools functions

Function	Description
`imap(func, *iterables)`	
`ifilter(func, iterable)`	
`combinations(iterable, k)`	
`permutations(iterable, k)`	
`groupby(iterable[, keyfunc])`	Generates `(key, sub-iterator)` for each unique key

In Python 3, several built-in functions (`zip`, `map`, `filter`) producing lists have been replaced by their generator versions found in `itertools` in Python 2.

CHAPTER 4

IPython: an interactive computing and development environment

> Act without doing; work without effort. Think of the small as large and the few as many. Confront the difficult while it is still easy; accomplish the great task by a series of small acts.
>
> —Laozi

People often ask me, "What is your Python development environment?" My answer is almost always the same, "IPython and a text editor". You may choose to substitute an Integrated Development Environment (IDE) for a text editor in order to take advantage of more advanced graphical tools and code completion capabilities. Even if so, I strongly recommend making IPython an important part of your workflow. Some IDEs even provide IPython integration, so it's possible to get the best of both worlds.

The IPython project (`http://ipython.org`) began in 2001 as Fernando Pérez's side project to make a better interactive Python interpreter. In the subsequent 10 years it has grown into what's widely considered one of the most important tools in the modern scientific Python computing stack. While it does not provide any computational or data analytical tools by itself, IPython is designed from the ground up to maximize your productivity in both interactive computing and software development. It encourages an *execute-explore* workflow instead of the typical *edit-compile-run* workflow of many other programming languages. It also provides very tight integration with the operating system's shell and file system. Since much of data analysis coding involves exploration, trial and error, and iteration, IPython will, in almost all cases, help you get the job done faster.

Of course, the IPython project now encompasses a great deal more than just an enhanced, interactive Python shell. It also includes a rich GUI console with inline plotting, a web-based interactive notebook format, and a lightweight, fast parallel computing engine. And, as with so many other tools designed for and by programmers, it is highly customizable. I'll discuss some of these features later in the chapter.

Since IPython has interactivity at its core, some of the features in this chapter are difficult to fully illustrate without a live console. If this is your first time learning about IPython, I recommend that you follow along with the examples to get a feel for how things work. As with any keyboard-driven console-like environment, developing muscle-memory for the common commands is part of the learning curve.

Many parts of this chapter (for example: profiling and debugging) can be safely omitted on a first reading as they are not necessary for understanding the rest of the book. This chapter is intended to provide a standalone, rich overview of the functionality provided by IPython.

IPython Basics

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

```
$ ipython
Python 2.7.2
Type "copyright", "credits" or "license" for more information.

IPython 0.12 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: a = 5

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

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

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

In [7]: data
Out[7]:
{0: 0.6934056585717416,
 1: 0.10241242040227709,
 2: -1.4294467562596778,
 3: 0.9715222124893931,
 4: -0.9828753467138842,
 5: -1.0547072060154692,
 6: 0.4741695390205941}
```

Many kinds of Python objects are formatted to be more readable, or *pretty-printed*. If you printed a dict like the above in the standard Python interpreter, it would be much less readable:

```
>>> from numpy.random import randn
>>> data = {i : randn() for i in range(7)}
>>> data
{0: -1.5948255432744511, 1: 0.10569006472787983, 2: 1.972367135977295,
3: 0.15455217573074576, 4: -0.24058577449429575, 5: -1.2904897053651216,
6: 0.3308507317325902}
```

IPython also provides facilities to make it easy to execute arbitrary blocks of code (via somewhat glorified copy-and-pasting) and whole Python scripts. These will be discussed shortly.

Tab completion

On the surface, the IPython shell looks like a cosmetically slightly-different interactive Python interpreter. Users of Mathematica may find the enumerated input and output prompts familiar. One of the major improvements over the standard Python shell is *tab completion*, a feature common to most interactive data analysis environments. While entering expressions in the shell, pressing `<Tab>` will search the namespace for any variables (objects, functions, etc.) matching the characters you have typed so far:

```
In [1]: an_apple = 27

In [2]: an_example = 42

In [3]: an<Tab>
an_apple    and         an_example  any
```

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

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

In [4]: b.<Tab>
b.append   b.extend   b.insert   b.remove   b.sort
b.count    b.index    b.pop      b.reverse
```

The same goes for modules:

```
In [1]: import datetime

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

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

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

```
In [3]: book_scripts/<Tab>
book_scripts/cprof_example.py        book_scripts/ipython_script_test.py
book_scripts/ipython_bug.py          book_scripts/prof_mod.py

In [3]: path = 'book_scripts/<Tab>
book_scripts/cprof_example.py        book_scripts/ipython_script_test.py
book_scripts/ipython_bug.py          book_scripts/prof_mod.py
```

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

Introspection

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

```
In [9]: b?
Type:       list
Base Class: <type 'list'>
String Form:[1, 2, 3]
Namespace:  Interactive
Length:     3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

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

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

    Returns
    -------
    the_sum : type of arguments
```

```
    """
    return a + b
```

Then using `?` shows us the docstring:

```
In [11]: add_numbers?
Type:       function
Base Class: <type 'function'>
String Form:<function add_numbers at 0x1d26d70>
Namespace:  Interactive
File:       /home/wesm/Dropbox/book/svn/book_scripts/<ipython-input-10-5473012eeb65>
Definition: add_numbers(a, b)
Docstring:
Add two numbers together
Returns
-------
the_sum : type of arguments
```

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

```
In [12]: add_numbers??
Type:       function
Base Class: <type 'function'>
String Form:<function add_numbers at 0x1d26d70>
Namespace:  Interactive
File:       /home/wesm/Dropbox/book/svn/book_scripts/<ipython-input-10-5473012eeb65>
Definition: add_numbers(a, b)
Source:
def add_numbers(a, b):
    """
    Add two numbers together
    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
```

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

```
In [13]: np.*load*?
np.load
np.loads
np.loadtxt
np.pkgload
```

The `%run` command

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

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

```
a = 5
b = 6
c = 7.5

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

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

```
In [14]: %run ipython_script_test.py
```

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

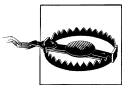
```
In [15]: c
Out[15]: 7.5

In [16]: result
Out[16]: 1.4666666666666666
```

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

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

Interrupting running code

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



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

Executing code from the clipboard

A quick-and-dirty way to execute code in IPython is via pasting from the clipboard. This might seem fairly crude, but in practice it is very useful. For example, while developing a complex or time-consuming application, you may wish to execute a script piece by piece, pausing at each stage to examine the currently loaded data and results.

Or, you might find a code snippet on the internet that you want to run and play around with, but you'd rather not create a new `.py` file for it.

Code snippets can be pasted from the clipboard in many cases by pressing `<Ctrl-Shift-V>`. Note that it is not completely robust as this mode of pasting mimics typing each line into IPython, and line breaks are treated as `<return>`. This means that if you paste code with an indented block and there is a blank line, IPython will think that the indented block is over. Once the next line in the block is then executed, an `IndentationError` will be raised. For example the following code

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

    y = 8
```

will not work if simply pasted:

```
In [1]: x = 5

In [2]: y = 7

In [3]: if x > 5:
   ...:         x += 1
   ...:

In [4]:     y = 8
IndentationError: unexpected indent

If you want to paste code into IPython, try the %paste and %cpaste
magic functions.
```

As the error message suggests, we should instead use the `%paste` and `%cpaste` magic functions. `%paste` takes whatever text is in the clipboard and executes it as a single block in the shell:

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

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

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

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

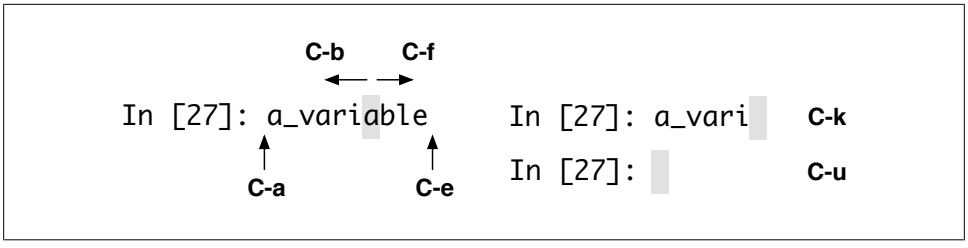


Figure 4-1. Illustration of some of IPython's keyboard shortcuts

```
:    y = 8
:--
```

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

Later, I'll introduce the IPython HTML Notebook which brings a new level of sophistication for developing analyses block-by-block in a browser-based notebook format with executable code cells.

IPython interaction with editors and IDEs

Some text editors, such as Emacs and vim, have 3rd party extensions enabling blocks of code to be sent directly from the editor to a running IPython shell. Refer to the IPython website or do an internet search to find out more.

Some integrated development environments, such as the PyDev plugin for Eclipse and Python Tools for Visual Studio from Microsoft (and possibly others), have integration with the IPython terminal application. If you want to work in an IDE but don't want to give up the IPython console features, this may be a good option for you.

Keyboard shortcuts

IPython has many keyboard shortcuts for navigating the prompt (which will be familiar to users of the Emacs text editor or the UNIX bash shell) and interacting with the shell's command history (see later section). Table 4-1 summarizes some of the most commonly used shortcuts. See Figure 4-1 for an illustration of a few of these, such as cursor movement.

Table 4-1. Standard IPython Keyboard Shortcuts

Command	Description
`Ctrl-P` or `up-arrow`	Search backward in command history for commands starting with currently-entered text
`Ctrl-N` or `down-arrow`	Search forward in command history for commands starting with currently-entered text
`Ctrl-R`	Readline-style reverse history search (partial matching)

Command	Description
Ctrl-Shift-V	Paste text from clipboard
Ctrl-C	Interrupt currently-executing code
Ctrl-A	Move cursor to beginning of line
Ctrl-E	Move cursor to end of line
Ctrl-K	Delete text from cursor until end of line
Ctrl-U	Discard all text on current line
Ctrl-F	Move cursor forward one character
Ctrl-B	Move cursor back one character
Ctrl-L	Clear screen

Exceptions and tracebacks

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

```
In [17]: %run ipython_bug.py
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/home/wesm/code/ipython/IPython/utils/py3compat.pyc in execfile(fname, *where)
    173             else:
    174                 filename = fname
--> 175             __builtin__.execfile(filename, *where)
/home/wesm/Dropbox/book/svn/book_scripts/ipython_bug.py in <module>()
     13     throws_an_exception()
     14
---> 15 calling_things()
/home/wesm/Dropbox/book/svn/book_scripts/ipython_bug.py in calling_things()
     11 def calling_things():
     12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()
/home/wesm/Dropbox/book/svn/book_scripts/ipython_bug.py in throws_an_exception()
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():
AssertionError:
```

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

Magic commands

IPython has many special commands, known as "magic" commands, which are designed to faciliate common tasks and enable you to easily control the behavior of the IPython system. A magic command is any command prefixed by the the percent symbol `%`. For example, you can check the execution time of any Python statement, such as a matrix multiplication, using the `%timeit` magic function (which will be discussed in more detail later):

```
In [18]: a = np.random.randn(100, 100)

In [19]: %timeit np.dot(a, a)
10000 loops, best of 3: 73.4 us per loop
```

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

```
In [1]: %reset?
Resets the namespace by removing all names defined by the user.

Parameters
----------
  -f : force reset without asking for confirmation.

  -s : 'Soft' reset: Only clears your namespace, leaving history intact.
  References to objects may be kept. By default (without this option),
  we do a 'hard' reset, giving you a new session and removing all
  references to objects from the current session.

Examples
--------
In [6]: a = 1

In [7]: a
Out[7]: 1

In [8]: 'a' in _ip.user_ns
Out[8]: True

In [9]: %reset -f

In [1]: 'a' in _ip.user_ns
Out[1]: False
```

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

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

Table 4-2. Frequently-used IPython Magic Commands

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

Qt-based Rich GUI Console

Matplotlib integration and pylab mode

Part of why IPython is so widely used in scientific computing is that it is designed as a companion to libraries like matplotlib and other GUI toolkits. Don't worry if you have never used matplotlib before; it will be discussed in much more detail in the rest of this book. If you create a matplotlib plot window in the regular Python shell, you'll be sad to find that the GUI event loop "takes control" of the Python session until the plot window is closed. That won't work for interactive data analysis and visualization, so IPython has implemented special handling of the GUI component so that it will work seamlessly with the shell.

The typical way to launch IPython with matplotlib integration is by adding the `--pylab` flag (two dashes).

```
$ ipython --pylab
```

This will cause several things to happen. First IPython will launch with the default GUI (graphical user interface) backend integration enabled so that matplotlib plot windows can be created with no issues. Secondly, most of NumPy and matplotlib will be imported into the top level interactive namespace to produce an interactive computing

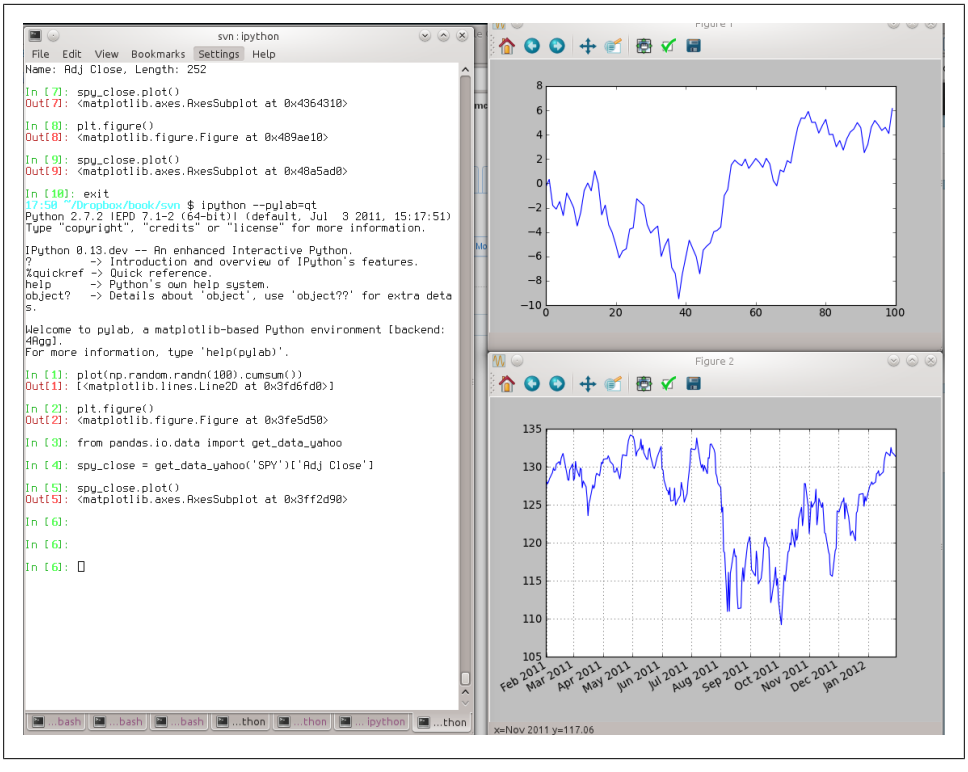


Figure 4-2. Pylab mode: IPython wtih matplotlib windows

environment reminiscent of MATLAB and other domain-specific scientific computing environments. It's possible to do this setup by hand (using the `%gui` magic and executing a few imports) but I won't go into detail about that here.

Using the Command History

IPython maintains a small on-disk database containing the text of each command that you execute. This serves various purposes:

- Searching, completing, and executing previously-executed commands with minimal typing
- Persisting the command history between sessions.
- Logging the input / output history to a file

Searching and reusing the command history

Being able to search and execute previous commands is, for many people, the most useful feature. Since IPython encourages an iterative, interactive code development

workflow, you may often find yourself repeating the same commands, such as a `%run` command or some other code snippet. Suppose you had run:

```
In[7]: %run first/second/third/data_script.py
```

and then explored the results of the script (assuming it ran successfully), only to find that you made an incorrect calculation. After figuring out the problem and modifying `data_script.py`, you can start typing a few letters of the `%run` command then press either the `<Ctrl-P>` key combination or the `<up arrow>` key. This will search the command history for the first prior command matching the letters you typed. Pressing either `<Ctrl-P>` or `<up arrow>` multiple times will continue to search through the history. If you pass over the command you wish to execute, fear not. You can move *forward* through the command history by pressing either `<Ctrl-N>` or `<down arrow>`. After doing this a few times you may start pressing these keys without thinking!

Using `<Ctrl-R>` gives you the same partial incremental searching capability provided by the `readline` used in UNIX-style shells, such as the bash shell. On Windows, `read line` functionality is emulated by IPython. To use this, press `<Ctrl-R>` then type a few characters contained in the input line you want to search for:

```
In [1]: a_command = foo(x, y, z)

(reverse-i-search)`com': a_command = foo(x, y, z)
```

Pressing `<Ctrl-R>` will cycle through the history for each line matching the characters you've typed.

Input and output variables

Forgetting to assign the result of a function call to a variable can be very annoying. Fortunately, IPython stores references to *both* the input (what you type) and output (what is returned) in special variables. The previous two outputs are stored in the `_` (one underscore) and `__` (two underscores) variables, respectively:

```
In [20]: 2 ** 27
Out[20]: 134217728

In [21]: _
Out[21]: 134217728
```

Input variables are stored in variables named like `_iX`, where `X` is the input line number. For each such input variables there is a corresponding output variable `_X`. So after input line 27, say, there will be two new variables `_27` (for the output) and `_i27` for the input.

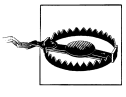
```
In [26]: foo = 'bar'

In [27]: foo
Out[27]: 'bar'

In [28]: _i27
Out[28]: u'foo'
```

```
In [29]: _27
Out[29]: 'bar'
```

Since the input variables are strings, that can be executed again using the Python `exec` keyword:

```
In [30]: exec _i27
```

Several magic functions allow you to work with the input and output history. `%hist` is capable of printing all or part of the input history, with or without line numbers. `%reset` is for clearing the interactive namespace and optionally the input and output caches. The `%xdel` magic function is intended for removing all references to a *particular* object from the IPython machinery. See the documentation for both of these magics for more details.



When working with very large data sets, keep in mind that IPython's input and output history causes any object referenced there to not be garbage collected (freeing up the memory), even if you delete the variables from the interactive namespace using the `del` keyword. In such cases, careful usage of `%xdel` and `%reset` can help you avoid running into memory problems.

Logging the input and output

IPython is capable of logging the entire console session including input and output. Logging is turned on by typing `%logstart`:

```
In [3]: %logstart
Activating auto-logging. Current session state plus future input saved.
Filename       : ipython_log.py
Mode           : rotate
Output logging : False
Raw input log  : False
Timestamping   : False
State          : active
```

IPython logging is that it can be enabled at any time and it will record your entire session up to that point. Thus, if you are working on something and you decide you want to save everything you did, you can simply enable logging. See the docstring of `%log start` for more options (including changing the output file path), as well as the companion functions `%logoff, %logon, %logstate,` and `%logstop`.

Interacting with the Operating System

Another important feature of IPython is that it provides very strong integration with the operating system shell. This means, among other things, that you can perform most standard command line actions as you would in the Windows or UNIX (Linux, OS X) shell without having to exit IPython. This includes executing shell commands, changing

directories, and storing the results of a command in a Python object (list or string). There are also simple shell command aliasing and directory bookmarking features.

See Table 4-3 for a summary of magic functions and syntax for calling shell commands. I'll briefly visit these features in the next few sections.

Table 4-3. IPython System-related Commands

Command	Description
`!cmd`	Execute cmd in the system shell
`output = !cmd args`	Run cmd and store the stdout in output
`%alias alias_name cmd`	Define an alias for a system (shell) command
`%bookmark`	Utilize IPython's directory bookmarking system
`%cd directory`	Change system working directory to passed directory
`%pwd`	Return the current system working directory
`%pushd directory`	Place current directory on stack and change to target directory
`%popd directory`	Change to directory popped off the top of the stack
`%dirs`	Return a list containing the current directory stack
`%dhist`	Print the history of visited directories
`%env`	Return the system environment variables as a dict

Shell commands and aliases

Starting a line in IPython with an exclamation point `!`, or bang, tells IPython to execute everything after the bang in the system shell. This means that you can delete files (using `rm` or `del`, depending on your OS), change directories, or execute any other process. It's even possible to start processes that take control away from IPython, even another Python interpreter:

```
In [2]: !python
Enthought Python Distribution -- www.enthought.com
Version: 7.1-2 (64-bit)

Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul  3 2011, 15:17:51)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)] on linux2
Type "packages", "demo" or "enthought" for more information.
>>>
```

The console output of a shell command can be stored in a variable by assigning the `!`-escaped expression to a variable. For example, on my Linux-based machine connected to the internet via ethernet, I can get my IP address as a Python variable:

```
In [1]: ip_info = !ifconfig eth0 | grep "inet "

In [2]: ip_info[0].strip()
Out[2]: 'inet addr:192.168.1.137  Bcast:192.168.1.255  Mask:255.255.255.0'
```

The returned Python object `ip_info` is actually a custom list type containing various versions of the console output. Check out the docstring (using `?`) for more information.

The `%alias` magic function can define custom shortcuts for shell commands. As a simple example:

```
In [1]: %alias ll ls -l

In [2]: ll /usr
total 332
drwxr-xr-x   2 root root  69632 2012-01-29 20:36 bin/
drwxr-xr-x   2 root root   4096 2010-08-23 12:05 games/
drwxr-xr-x 123 root root  20480 2011-12-26 18:08 include/
drwxr-xr-x 265 root root 126976 2012-01-29 20:36 lib/
drwxr-xr-x  44 root root  69632 2011-12-26 18:08 lib32/
lrwxrwxrwx   1 root root      3 2010-08-23 16:02 lib64 -> lib/
drwxr-xr-x  15 root root   4096 2011-10-13 19:03 local/
drwxr-xr-x   2 root root  12288 2012-01-12 09:32 sbin/
drwxr-xr-x 387 root root  12288 2011-11-04 22:53 share/
drwxrwsr-x  24 root src    4096 2011-07-17 18:38 src/
```

Multiple commands can be executed just as on the command line by separating them with semicolons:

```
In [22]: %cd ..
/home/wesm/Dropbox/book/svn

In [23]: %alias test_alias (cd book_scripts; ls; cd ..)

In [24]: test_alias
array_ex.txt         generated_timeseries.xml  numpy_option_pricing.py
baby_names         ipython_bug.py           prof_mod.py
broadcast_mayavi.py  ipython_script_test.py    twitter
cprof_example.py     movielens               whetting

In [25]: %cd book_scripts
/home/wesm/Dropbox/book/svn/book_scripts
```

You'll notice that IPython "forgets" any aliases you define interactively as soon as the session is closed. To create permanent aliases, you will need to use the configuration system. See later in the chapter.

Directory bookmark system

IPython has a simple directory bookmarking system to enable you to save aliases for common directories so that you can jump around very easily. For example, I'm an avid user of Dropbox, so I can define a bookmark to make it very each to change directories to my Dropbox:

```
In [6]: %bookmark db /home/wesm/Dropbox/
```

Once I've done this, when I use the `%cd` magic, I can use any bookmarks I've defined

```
In [7]: cd db
(bookmark:db) -> /home/wesm/Dropbox/
/home/wesm/Dropbox
```

If a bookmark name conflicts with a directory name in your current working directory, you can use the `-b` flag to override and use the bookmark location. Using the `-l` option with `%bookmark` lists all of your bookmarks:

```
In [8]: %bookmark -l
Current bookmarks:
db -> /home/wesm/Dropbox/
```

Bookmarks, unlike aliases, are automatically persisted between IPython sessions.

If you would like to have directory bookmarks in your regular terminal, too, and you use the bash shell in Linux or OS X, take a look at Huy Nguyen's `bashmarks` project.

Software Development Tools

In addition to being a comfortable environment for interactive computing and data exploration, IPython is well suited as a software development environment. In data analysis applications, it's important first to have *correct* code. Fortunately, IPython has closely integrated and enhanced the built-in Python `pdb` debugger. Secondly you want your code to be *fast*. For this IPython has easy-to-use code timing and profiling tools. I will give an overview of these tools in detail here.

Interactive Debugger

IPython's debugger enhances `pdb` with tab completion, syntax highlighting, and context each line in exception tracebacks. One of the best times to debug code is right after an error has occurred. The `%debug` command, when typed right after an error, invokes the "post-mortem" debugger and drops you into the stack frame where the exception was raised:

```
In [2]: run ipython_bug.py
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/home/wesm/Dropbox/book/svn/book_scripts/ipython_bug.py in <module>()
     13     throws_an_exception()
     14
---> 15 calling_things()

/home/wesm/Dropbox/book/svn/book_scripts/ipython_bug.py in calling_things()
     11 def calling_things():
     12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()
```

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

AssertionError:

In [3]: %debug
> /home/wesm/Dropbox/book/svn/book_scripts/ipython_bug.py(9)throws_an_exception()
      8     b = 6
----> 9     assert(a + b == 10)
     10

ipdb>
```

Once inside the debugger, you can execute arbitrary Python code and explore all of the objects and data (which have been "kept alive" by the interpreter) inside each stack frame. By default you start in the lowest level, where the error occurred. By pressing `u` (up) and `d` (down), you can switch between the levels of the stack trace:

```
ipdb> u
> /home/wesm/book/book_scripts/ipython_bug.py(13)calling_things()
     12     works_fine()
---> 13     throws_an_exception()
     14
```

Executing the `%pdb` command makes it so that IPython automatically invokes the debugger after any exception, a mode that many users will find especially useful.

It's also easy to use the debugger to help develop code, especially when you wish to set breakpoints or step through the execution of a function or script to examine the state at each stage. There are several ways to accomplish this. The first is by using `%run` with the `-d` flag, which invokes the debugger before executing any code in the passed script. You must immediately press `s` (step) to enter the script:

```
In [5]: run -d book_scripts/ipython_bug.py
Breakpoint 1 at /home/wesm/Dropbox/book/svn/book_scripts/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()

ipdb> s
--Call--
> /home/wesm/book/book_scripts/ipython_bug.py(1)<module>()
1---> 1 def works_fine():
      2     a = 5
      3     b = 6
```

After this point, it's up to you how you want to work your way through the file. For example, in the above exception, we could set a breakpoint right before calling the `works_fine` method and run the script until we reach the breakpoint by pressing `c` (continue):

```
ipdb> b 12
ipdb> c
> /home/wesm/Dropbox/book/svn/book_scripts/ipython_bug.py(12)calling_things()
     11 def calling_things():
2--> 12     works_fine()
     13     throws_an_exception()
```

At this point, you can `s`-tep into `works_fine()` or execute `works_fine()` by pressing `n` (next) to advance to the next line:

```
ipdb> n
> /home/wesm/Dropbox/book/svn/book_scripts/ipython_bug.py(13)calling_things()
2    12     works_fine()
---> 13     throws_an_exception()
     14
```

Then, we could step into `throws_an_exception` and advance to the line where the error occurs and look at the variables in the scope. Note that I made the unfortunate decision to name the variables the same as the debugger commands (which take precedence), so I have to use `print` to display them.

```
ipdb> s
--Call--
> /home/wesm/Dropbox/book/svn/book_scripts/ipython_bug.py(6)throws_an_exception()
      5
----> 6 def throws_an_exception():
      7     a = 5

ipdb> n
> /home/wesm/Dropbox/book/svn/book_scripts/ipython_bug.py(7)throws_an_exception()
      6 def throws_an_exception():
----> 7     a = 5
      8     b = 6

ipdb> n
> /home/wesm/Dropbox/book/svn/book_scripts/ipython_bug.py(8)throws_an_exception()
      7     a = 5
----> 8     b = 6
      9     assert(a + b == 10)

ipdb> n
> /home/wesm/Dropbox/book/svn/book_scripts/ipython_bug.py(9)throws_an_exception()
      8     b = 6
----> 9     assert(a + b == 10)
     10

ipdb> print a
5
ipdb> print b
6
```

Becoming proficient in the interactive debugger is largely a matter of practice and experience. See Table 4-3 for a full catalogue of the debugger commands. If you are used to an IDE, you might find the terminal-driven debugger to be a bit bewildering at first, but that will improve in time. Most of the Python IDEs have excellent GUI debuggers,

but it is usually a significant productivity gain to not have to leave IPython to do debugging.

Table 4-4. (I)Python debugger commands

Command	Action
h(elp)	Display command list
help *command*	Show documentation for *command*
c(ontinue)	Resume program execution
q(uit)	Exit debugger without executing any more code
b(reak) *number*	Set breakpoint at *number* in current file
b *path/to/file.py:number*	Set breakpoint at line *number* in specified file
s(tep)	Step *into* function call
n(ext)	Execute current line and advance to next line at current level
u(p) / d(own)	Move up / down in function call stack
a(rgs)	Show arguments for current function
debug *statement*	Invoke statement *statement* in new (recursive) debugger
l(ist) *statement*	Show current position and context at current level of stack
w(here)	Print full stack trace with context at current position

Other ways to make use of the debugger

There are couple of other useful ways to invoke the debugger. The first is by using a special `set_trace` function (named after `pdb.set_trace`), which is basically a "poor man's breakpoint". Here are two small recipes that you might want to put somewhere for your general use (potentially adding them to your IPython profile as I do):

```
def set_trace():
    from IPython.core.debugger import Pdb
    Pdb(color_scheme='Linux').set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):
    from IPython.core.debugger import Pdb
    pdb = Pdb(color_scheme='Linux')
    return pdb.runcall(f, *args, **kwargs)
```

The first function, `set_trace`, is very simple. Put `set_trace()` anywhere in your code that you want to stop and take a look around (for example, right before an exception occurs):

```
In [7]: run ipython_bug.py
> /home/wesm/book/book_scripts/ipython_bug.py(16)calling_things()
     15     set_trace()
---> 16     throws_an_exception()
     17
```

Pressing `c` (continue) will cause the code to resume normally with no harm done.

The `debug` function above enables you to invoke the interactive debugger easily on an arbitrary function call. Suppose we had written a function like

```
def f(x, y, z=1):
    tmp = x + y
    return tmp / z
```

and we wished to step through its logic. Ordinarily using `f` would look like `f(1, 2, z=3)`. To instead step into `f`, pass `f` as the first argument to `debug` followed by the positional and keyword arguments to be passed to `f`:

```
In [6]: debug(f, 1, 2, z=3)
> <ipython-input>(2)f()
      1 def f(x, y, z):
----> 2     tmp = x + y
      3     return tmp / z

ipdb>
```

I find that these two simple recipes save me a lot of time on a day-to-day basis.

Timing code: `%time` and `%timeit`

For larger-scale or longer-running data analysis applications, you may wish to measure the execution time of various components or of individual statements or function calls. You may want to have a report of which functions are taking up the most time in a complex process. Fortunately, IPython enables you to get this information very easily while you are developing and testing your code.

Timing code by hand using the built-in `time` module and its functions `time.clock` and `time.time` is often tedious and repetitive, as you must write the same uninteresting boilerplate code:

```
import time
start = time.time()
for i in range(iterations):
    # some code to run here
elapsed_per = (time.time() - start) / iterations
```

Since this is such a common operation, IPython has two magic functions `%time` and `%timeit` to automate this process for you. `%time` runs a statement once, reporting the total execution time. Suppose we had a large list of strings and we wanted to compare different methods of selecting all strings starting with a particular prefix. Here is a simple list of 700,000 strings and two identical methods of selecting only the ones that start with `'foo'`:

```
# a very large list of strings
strings = ['foo', 'foobar', 'baz', 'qux',
           'python', 'Guido Van Rossum'] * 100000

method1 = [x for x in strings if x.startswith('foo')]

method2 = [x for x in strings if x[:3] == 'foo']
```

It looks like they should be about the same performance-wise, right? We can check for sure using `%time`:

```
In [27]: %time method1 = [x for x in strings if x.startswith('foo')]
CPU times: user 0.20 s, sys: 0.01 s, total: 0.20 s
Wall time: 0.20 s

In [28]: %time method2 = [x for x in strings if x[:3] == 'foo']
CPU times: user 0.10 s, sys: 0.01 s, total: 0.11 s
Wall time: 0.11 s
```

The `Wall time` is the main number of interest. So, it looks like the first method takes more than twice as long, but it's not a very precise measurement. If you try `%time`-ing those statements multiple times yourself, you'll find that the results are somewhat variable. To get a more precise measurement, use the `%timeit` magic function. Given an arbitrary statement, it has a heuristic to run a statement multiple times to produce a fairly accurate average runtime.

```
In [29]: %timeit [x for x in strings if x.startswith('foo')]
10 loops, best of 3: 156 ms per loop

In [30]: %timeit [x for x in strings if x[:3] == 'foo']
10 loops, best of 3: 61.7 ms per loop
```

This seemingly innocuous example illustrates that it is worth understanding the performance characteristics of the Python standard library, NumPy, pandas, and other libraries used in this book. In larger-scale data analysis applications, those milliseconds will start to add up!

`%timeit` is especially useful for analyzing statements and functions with very short execution times, even at the level of microseconds (1e-6 seconds) or nanoseconds (1e-9 seconds). These may seem like insignificant amounts of time, but of course a 20 microsecond function invoked 1 million times takes 15 seconds longer than a 5 microsecond function. In the above example, we could very directly compare the two string operations to understand their performance characteristics:

```
In [31]: x = 'foobar'

In [32]: y = 'foo'

In [33]: %timeit x.startswith(y)
1000000 loops, best of 3: 273 ns per loop

In [34]: %timeit x[:3] == y
10000000 loops, best of 3: 137 ns per loop
```

Basic profiling: `%prun` and `%run -p`

Profiling code is closely related to timing code, except it is concerned with determining *where* time is spent. The main Python profiling tool is the `cProfile` module, which is

not specific to IPython at all. `cProfile` executes program or any arbitrary block of code while keeping track of how much time is spent in each function.

A common way to use `cProfile` is on the command line, running an entire program and outputting the aggregated time per function. Suppose we had a simple script which does some linear algebra in a loop (computing the maximum absolute eigenvalues of a series of `100 x 100` matrices):

```
import numpy as np
from numpy.linalg import eigvals

def run_experiment(niter=100):
    K = 100
    results = []
    for _ in xrange(niter):
        mat = np.random.randn(K, K)
        max_eigenvalue = np.abs(eigvals(mat)).max()
        results.append(max_eigenvalue)
    return results
some_results = run_experiment()
print 'Largest one we saw: %s' % np.max(some_results)
```

Don't worry if you are not familiar with NumPy. You can run this script through `cProfile` by running the following in the command line:

```
python -m cProfile cprof_example.py
```

If you try that, you'll find that the results are outputted sorted by function name. This makes it a bit hard to get an idea of where the most time is spent, so it's very common to specify a *sort order* using the `-s` flag:

```
$ python -m cProfile -s cumulative cprof_example.py
Largest one we saw: 11.923204422
    15116 function calls (14927 primitive calls) in 0.720 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.001    0.001    0.721    0.721 cprof_example.py:1(<module>)
   100    0.003    0.000    0.586    0.006 linalg.py:702(eigvals)
   200    0.572    0.003    0.572    0.003 {numpy.linalg.lapack_lite.dgeev}
     1    0.002    0.002    0.075    0.075 __init__.py:106(<module>)
   100    0.059    0.001    0.059    0.001 {method 'randn')
     1    0.000    0.000    0.044    0.044 add_newdocs.py:9(<module>)
     2    0.001    0.001    0.037    0.019 __init__.py:1(<module>)
     2    0.003    0.002    0.030    0.015 __init__.py:2(<module>)
     1    0.000    0.000    0.030    0.030 type_check.py:3(<module>)
     1    0.001    0.001    0.021    0.021 __init__.py:15(<module>)
     1    0.013    0.013    0.013    0.013 numeric.py:1(<module>)
     1    0.000    0.000    0.009    0.009 __init__.py:6(<module>)
     1    0.001    0.001    0.008    0.008 __init__.py:45(<module>)
   262    0.005    0.000    0.007    0.000 function_base.py:3178(add_newdoc)
   100    0.003    0.000    0.005    0.000 linalg.py:162(_assertFinite)
   ...
```

Only the first 15 rows of the output are shown. It's easiest to read by scanning down the `cumtime` column to see how much total time was spent *inside* each function. Note that if a function calls some other function, *the clock does not stop running*. `cProfile` records the start and end time of each function call and uses that to produce the timing.

In addition to the above command line usage, `cProfile` can also be used programmatically to profile arbitrary blocks of code without having to run a new process. IPython has a convenient interface to this capability using the `%prun` command and the `-p` option to `%run`. `%prun` takes the same "command line options" as `cProfile` but will profile an arbitrary Python statement instead of a while `.py` file:

```
In [4]: %prun -l 7 -s cumulative run_experiment()
         4203 function calls in 0.643 seconds

Ordered by: cumulative time
List reduced from 32 to 7 due to restriction <7>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.643    0.643 <string>:1(<module>)
     1    0.001    0.001    0.643    0.643 cprof_example.py:4(run_experiment)
   100    0.003    0.000    0.583    0.006 linalg.py:702(eigvals)
   200    0.569    0.003    0.569    0.003 {numpy.linalg.lapack_lite.dgeev}
   100    0.058    0.001    0.058    0.001 {method 'randn'}
   100    0.003    0.000    0.005    0.000 linalg.py:162(_assertFinite)
   200    0.002    0.000    0.002    0.000 {method 'all' of 'numpy.ndarray' objects}
```

Similarly, calling `%run -p -s cumulative cprof_example.py` has the same effect as the command line approach above, except you never have to leave IPython.

Profiling a function line-by-line

In some cases the information you obtain from `%prun` (or another `cProfile`-based profile method) may not tell the whole story about a function's execution time, or it may be so complex that the results, aggregated by function name, are hard to interpret. For this case, there is a small library called `line_profiler` (obtainable via PyPI or one of the package management tools). It contains an IPython extension enabling a new magic function `%lprun` that computes a line-by-line-profiling of one or more functions. You can enable this extension by modifying your IPython configuration (see the IPython documentation or the section on configuration later in this chapter) to include the following line:

```
# A list of dotted module names of IPython extensions to load.
c.TerminalIPythonApp.extensions = ['line_profiler']
```

`line_profiler` can be used programmatically (see the full documentation), but it is perhaps most powerful when used interactively in IPython. Suppose you had a module `prof_mod` with the following code doing some NumPy array operations:

```
from numpy.random import randn

def add_and_sum(x, y):
```

```
    added = x + y
    summed = added.sum(axis=1)
    return summed

def call_function():
    x = randn(1000, 1000)
    y = randn(1000, 1000)
    return add_and_sum(x, y)
```

If we wanted to understand the performance of the `add_and_sum` function, `%prun` gives us the following:

```
In [35]: %run prof_mod

In [36]: x = randn(3000, 3000)

In [37]: y = randn(3000, 3000)

In [38]: %prun add_and_sum(x, y)
         4 function calls in 0.061 seconds
   Ordered by: internal time
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.045    0.045    0.056    0.056 prof_mod.py:3(add_and_sum)
        1    0.011    0.011    0.011    0.011 {method 'sum' of 'numpy.ndarray' objects}
        1    0.005    0.005    0.061    0.061 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

This is not especially enlightening. With the `line_profiler` IPython extension activated, a new command `%lprun` is available. The only difference in usage is that we must instruct `%lprun` which function or functions we wish to profile. The general syntax is:

```
%lprun -f func1 -f func2 statement_to_profile
```

In this case, we want to profile `add_and_sum`, so we run:

```
In [39]: %lprun -f add_and_sum add_and_sum(x, y)
Timer unit: 1e-06 s
File: /home/wesm/Dropbox/book/svn/book_scripts/prof_mod.py
Function: add_and_sum at line 3
Total time: 0.054916 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           def add_and_sum(x, y):
     4         1        44030  44030.0     80.2      added = x + y
     5         1        10884  10884.0     19.8      summed = added.sum(axis=1)
     6         1            2      2.0      0.0      return summed
```

You'll probably agree this is much easier to interpret. In this case we profiled the same function we used in the statement. Looking at the module code above, we could call `call_function` and profile that as well as `add_and_sum`, thus getting a full picture of the performance of the code:

```
In [40]: %lprun -f add_and_sum -f call_function call_function()
Timer unit: 1e-06 s
File: /home/wesm/Dropbox/book/svn/book_scripts/prof_mod.py
Function: add_and_sum at line 3
```

```
Total time: 0.006782 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           def add_and_sum(x, y):
     4         1         5347   5347.0     78.8      added = x + y
     5         1         1433   1433.0     21.1      summed = added.sum(axis=1)
     6         1            2      2.0      0.0      return summed
File: /home/wesm/Dropbox/book/svn/book_scripts/prof_mod.py
Function: call_function at line 8
Total time: 0.130883 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     8                                           def call_function():
     9         1        61874  61874.0     47.3      x = randn(1000, 1000)
    10         1        61696  61696.0     47.1      y = randn(1000, 1000)
    11         1         7313   7313.0      5.6      return add_and_sum(x, y)
```

As a general rule of thumb, I tend to prefer `%prun` (`cProfile`) for "macro" profiling and `%lprun` (`line_profiler`) for "micro" profiling. It's worthwhile to have a good understanding of both tools.

Tips for productive code development using IPython

Writing code in a way that makes it easy to develop, debug, and ultimately simply *use* interactively may be a bit of a paradigm shift for many users. There are procedural details like code reloading that may require some adjustment as well as coding style concerns.

As such, most of this section is more of an art than a science and will require some experimentation on your part to determine a way to write your Python code that is effective and productive for you. Ultimately you want to structure your code in a way to make it easy to use iteratively and to be able to explore the results of running a program or function as effortlessly as possible. I have found software designed with IPython in mind to be often easier to work with than code intended only to be run as as standalone command line application. This becomes especially important when something goes wrong and you have to diagnose an error in code that you or someone else might have written months or years beforehand.

Reloading module dependencies

In Python, when you type `import some_lib`, the code in `some_lib` is executed and all the variables, functions, and imports defined within are stored in the newly created `some_lib` module namespace. The next time you type `import some_lib`, you will get a reference to the existing module namespace. The potential difficulty in interactive code development in IPython comes when you, say, `%run` a script that depends on some other module where you may have made changes. Suppose I had the following code in `test_script.py`:

```
import some_lib

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

If you were to execute `%run test_script.py` then modify `some_lib.py`, the next time you execute `%run test_script.py` you will still get the *old version* of `some_lib` because of Python's "load-once" module system. This behavior differs from some other data analysis environments, like MATLAB, which automatically propagate code changes. To cope with this, you have a couple of options. The first way is to use the Python built-in `reload` function, altering `test_script.py` to be

```
import some_lib
reload(some_lib)

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

This guarantees that you will get a fresh copy of `some_lib` every time you run `test_script.py`. Obviously, if the dependencies go deeper, it might be a bit tricky to be inserting usages of `reload` all over the place. For this problem, IPython has a special `dreload` function (*not* a magic function) for "deep" (recursive) reloading of modules. If I were to run `import some_lib` then type `dreload(some_lib)`, it will attempt to reload `some_lib` as well as all of its dependencies. This will not work in all cases, unfortunately, but when it does it beats having to restart IPython.

Code design tips

There's no simple recipe for this, but here are some high level principles I have found effective in my own work.

Keep relevant objects and data alive

It's not unusual to see a program written for the command line with a structure somewhat like the following trivial example

```
from my_functions import g

def f(x, y):
    return g(x + y)

def main():
    x = 6
    y = 7.5
    result = x + y

if __name__ == '__main__':
    main()
```

Do you see what might be wrong with this program if we were to run it in IPython? After it's done, none of the results or objects defined in the `main` function willl be accessible in the IPython shell. A better way is to have whatever code is in `main` execute directly in the module's global namespace (or in the `if __name__ == '__main__':` block, if you want the module to also be importable). That way, when you `%run` the code, you'll be able to look at all of the variables defined in `main`. It's less meaningful in this simple example, but in this book we'll be looking at some complex data analysis problems involving large data sets that you will want to be able to play with in IPython.

Flat is better than nested

Deeply nested code makes me think about the many layers of an onion. When testing or debugging a function, how many layers of the onion must you peel back in order to reach the code of interest? The idea that "flat is better than nested" is a part of the Zen of Python, and it applies generally to developing code for interactive use as well. Making functions and classes as decoupled and modular as possible makes them easier to test (if you are writing unit tests), debug, and use interactively.

Overcome a fear of longer files

If you come from a Java (or other such language) background, you may have been told to keep files short. In many languages, this is sound advice; long length is usually a bad "code smell", indicating refactoring or reorganization may be necessary. However, while developing code using IPython, working with 10 small, but interconnected files (under, say, 100 lines each) is likely to cause you more headache in general than a single large file or two or three longer files. Fewer files means fewer modules to reload and less jumping between files while editing, too. I have found maintaining larger modules, each with high *internal* cohesion, to be much more useful and pythonic.

Obviously, I don't support taking this argument to the extreme, which would to be to put all of your code in a single monstrous file. Finding a sensible and intuitive module and package structure for a large codebase often takes a bit of work, but it is especially important to get right in teams. Each module should be internally cohesive, and it should be as obvious as possible where to find functions and classes responsible for each area of functionality.

Advanced IPython Features

Making your own classes IPython-friendly

IPython makes every effort to display a console-friendly string representation of any object that you inspect. For many objects, like dicts, lists, and tuples, the built-in `pprint` module is used to do the nice formatting. In user-defined classes, however, you

have to generate the desired string output yourself. Suppose we had the following simple class:

```
class Message:
    def __init__(self, msg):
        self.msg = msg
```

If you wrote this, you would be disappointed to that the default output for your class isn't very nice:

```
In [42]: x = Message('I have a secret')

In [43]: x
Out[43]: <__main__.Message instance at 0x1deefc8>
```

IPython takes the string returned by the `__repr__` magic method (by doing `output = repr(obj)`) and prints that to the console. Thus, we can add a simple `__repr__` method to the above class to get a more helpful output:

```
class Message:
    def __init__(self, msg):
        self.msg = msg

    def __repr__(self):
        return 'Message: %s' % self.msg
```

```
In [45]: x = Message('I have a secret')

In [46]: x
Out[46]: Message: I have a secret
```

Profiles and Configuration

Most aspects of the appearance (colors, prompt, spacing between lines, etc.) and behavior of the IPython shell are configurable through an extensive configuration system. Here are some of the things you can do via configuration:

- Change the color scheme
- Change how the input and output prompts look, or remove the blank line after `Out` and before the next `In` prompt
- Change how the input and output prompts look
- Execute an arbitrary list of Python statements. These could be imports that you use all the time or anything else you want to happen each time you launch IPython.
- Enable IPython extensions, like the `%lprun` magic in `line_profiler`.

All of these configuration options are specified in a special `ipython_config.py` file which will be found in the `~/.config/ipython/` directory on UNIX-like systems and `%HOME %/.ipython/` directory on Windows. Where your home directory is depends on your system. Configuration is performed based on a particular *profile*. When you start IPython normally, you load up, by default, the *default profile*, stored in the `pro`

`file_default` directory. Thus, on my Linux OS the full path to my default IPython configuration file is:

```
/home/wesm/.config/ipython/profile_default/ipython_config.py
```

I'll spare you the gory details of what's in this file. Fortunately it has comments describing what each configuration options is for, so I will leave it to the reader to tinker and customize. One additional useful feature is that it's possible to have *multiple profiles*. Suppose you wanted to have an alternate IPython configuration tailored for a particular application or project. Creating a new profile is as simple is typing something like

```
ipython profile create secret_project
```

Once you've done this, edit the config files in the newly-created `pro file_secret_project` directory then launch IPython like so

```
$ ipython --profile=secret_project
Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul  3 2011, 15:17:51)
Type "copyright", "credits" or "license" for more information.

IPython 0.13 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

IPython profile: secret_project

In [1]:
```

As always, the online IPython documentation is an excellent resource for more on profiles and configuration.

Credits

Parts of this chapter were derived from the wonderful documentation put together by the IPython Development Team. I can't thank them enough for all of their work building this amazing set of tools.

CHAPTER 5

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 with no 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. Especially 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 the chapter later in the book.

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, 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 the pandas chapters.

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 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 arrays of data using similar syntax to the equivalent operations between scalar elements:

```
In [8]: data
Out[8]:
array([[-0.6947,  0.5459, -1.0446, -1.9789,  0.4151],
       [-0.7934, -1.4306, -1.635 ,  0.3645,  1.2094],
       [-1.8712,  0.0709, -0.4736, -0.2657, -1.1278],
       [-0.3593,  0.6283,  0.0046,  0.1597,  0.027 ]])

In [9]: data * 10
Out[9]:
array([[ -6.9474,   5.4587, -10.4461, -19.7892,   4.1507],
       [ -7.934 , -14.3061, -16.3503,   3.6447,  12.0936],
       [-18.7123,   0.7088,  -4.7361,  -2.6571, -11.278 ],
       [ -3.5931,   6.2833,   0.0463,   1.5973,   0.2704]])

In [10]: data + data
Out[10]:
array([[-1.3895,  1.0917, -2.0892, -3.9578,  0.8301],
       [-1.5868, -2.8612, -3.2701,  0.7289,  2.4187],
```

```
       [-3.7425,  0.1418, -0.9472, -0.5314, -2.2556],
       [-0.7186,  1.2567,  0.0093,  0.3195,  0.0541]])
```

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` indicating the size of each dimension and a *data type* or `dtype` indicating the kind of data it contains:

```
In [11]: data.shape
Out[11]: (4, 5)

In [12]: data.dtype
Out[12]: dtype('float64')
```

This chapter will introduce you to the basics of using NumPy array 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
```

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 [20]: arr1.dtype
Out[20]: dtype('float64')

In [21]: arr2.dtype
Out[21]: 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 [22]: np.zeros(10)
Out[22]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [23]: np.zeros((3, 7))
Out[23]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]])

In [24]: np.empty((2, 3, 2))
Out[24]:
array([[[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]],
       [[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]]])
```

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

```
In [25]: np.arange(15)
Out[25]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

See Table 5-1 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). More on this in the next section.

Table 5-1. Array creation functions

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Always copies the input data by default.
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray
`ones, ones_like`	Produce an array of all 1's with the given shape and dtype. `ones_like` takes another array and produces a ones array of the same shape and dtype.
`zeros, zeros_like`	Like `ones` and `ones_like` but producing arrays of 0's instead

Function	Description
`empty, empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`eye, identity`	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 [26]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [27]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [28]: arr1.dtype
Out[28]: dtype('float64')

In [29]: arr2.dtype
Out[29]: 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 5-2 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.

You can explicitly convert or *cast* an array from one dtype to another using ndarray's `astype` method:

```
In [30]: arr = np.array([1, 2, 3, 4, 5])

In [31]: arr.dtype
Out[31]: dtype('int64')

In [32]: float_arr = arr.astype(np.float64)

In [33]: float_arr.dtype
Out[33]: dtype('float64')
```

In this example, integers were casted to floating point. If I cast some floating point numbers to integer dtype, the decimal part will be truncated:

```
In [34]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [35]: arr
Out[35]: array([  3.7,  -1.2,  -2.6,   0.5,  12.9,  10.1])

In [36]: arr.astype(np.int32)
Out[36]: 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 [37]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)

In [38]: numeric_strings.astype(float)
Out[38]: 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 [39]: int_array = np.arange(10)

In [40]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

In [41]: int_array.astype(calibers.dtype)
Out[41]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

As you see in Table 5-2, there are shorthand type code strings you can also use to refer to a dtype:

```
In [42]: empty_uint32 = np.empty(8, dtype='u4')

In [43]: empty_uint32
Out[43]:
array([1527582424,      32517,   51371856,          0,   50883056,
                0,   51328608,          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.

Table 5-2. NumPy data types

Type	Type Code	Description
`int8, uint8`	`i1, u1`	Signed and unsigned 8-bit (1 byte) integer types
`int16, uint16`	`i2, u2`	Signed and unsigned 16-bit integer types
`int32, uint32`	`i4, u4`	Signed and unsigned 32-bit integer types

Type	Type Code	Description
int64, uint64	i8, u8	Signed and unsigned 32-bit integer types
float16	f2	Half-precision floating point
float32	f4 or f	Standard single--precision floating point. Compatible with C float
float64, float128	f8 or d	Standard double--precision floating point. Compatible with C double and Python float object
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	O	Python object type
string_	S	Fixed-length string type (1 byte per character)
unicode_	U	Fixed-length unicode type (item size platform specific)

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

In [45]: arr
Out[45]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [46]: arr * arr
Out[46]:
array([[  1.,   4.,   9.],
       [ 16.,  25.,  36.]])

In [47]: arr - arr
Out[47]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Arithmetic operations with scalars are as you would expect, propagating the value to each element:

```
In [48]: 1 / arr
Out[48]:
array([[ 1.    ,  0.5   ,  0.3333],
       [ 0.25  ,  0.2   ,  0.1667]])

In [49]: arr ** 0.5
Out[49]:
```

```
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 the Advanced NumPy chapter. 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 out a subset of a data set or individual elements. One dimensional arrays are simple; on the surface they act similarly to Python lists:

```
In [50]: arr = np.arange(10)

In [51]: arr
Out[51]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [52]: arr[5]
Out[52]: 5

In [53]: arr[5:8]
Out[53]: array([5, 6, 7])

In [54]: arr[5:8] = 12

In [55]: arr
Out[55]: 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 [56]: arr_slice = arr[5:8]

In [57]: arr_slice[1] = 12345

In [58]: arr
Out[58]: array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,     9])

In [59]: arr_slice[:] = 64

In [60]: arr
Out[60]: 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.

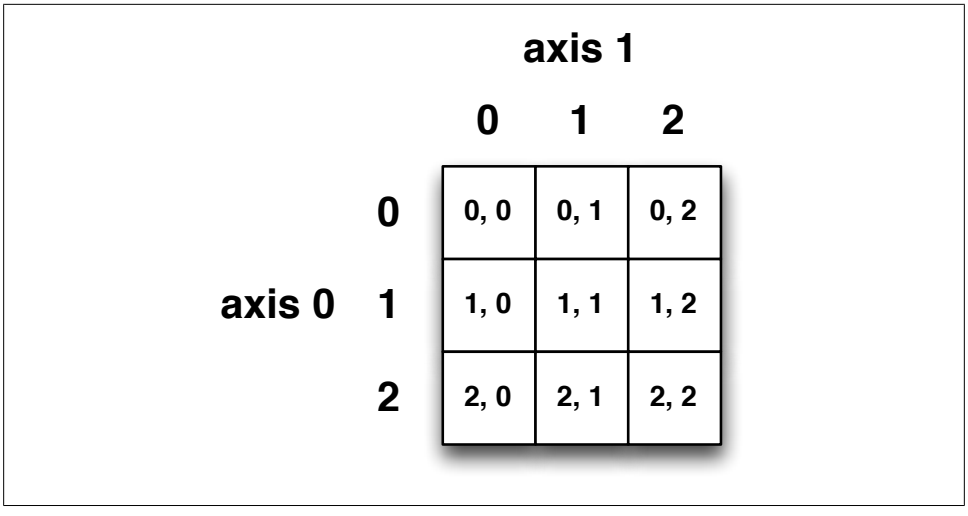
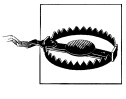


Figure 5-1. Indexing elements in a NumPy array



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 1-dimensional arrays:

```
In [61]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [62]: arr2d[2]
Out[62]: 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 [63]: arr2d[0][2]
Out[63]: 3

In [64]: arr2d[2, 0]
Out[64]: 7
```

See Figure 5-1 for an illustration of indexing on a 2D 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 x 2 x 3 array `arr3d`

```
In [65]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [66]: arr3d
Out[66]:
array([[[ 1,  2,  3],
```

```
         [ 4,  5,  6]],
        [[ 7,  8,  9],
         [10, 11, 12]]])
```

`arr3d[0]` is a 2 x 3 array:

```
In [67]: arr3d[0]
Out[67]:
array([[1, 2, 3],
       [4, 5, 6]])
```

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

```
In [68]: old_values = arr3d[0].copy()

In [69]: arr3d[0] = 42

In [70]: arr3d
Out[70]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [71]: arr3d[0] = old_values

In [72]: arr3d
Out[72]:
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 [73]: arr3d[1, 0]
Out[73]: 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 1-dimensional objects such as Python lists, ndarrays can be sliced using the familiar syntax:

```
In [74]: arr[1:6]
Out[74]: 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 [75]: arr2d
Out[75]:
```

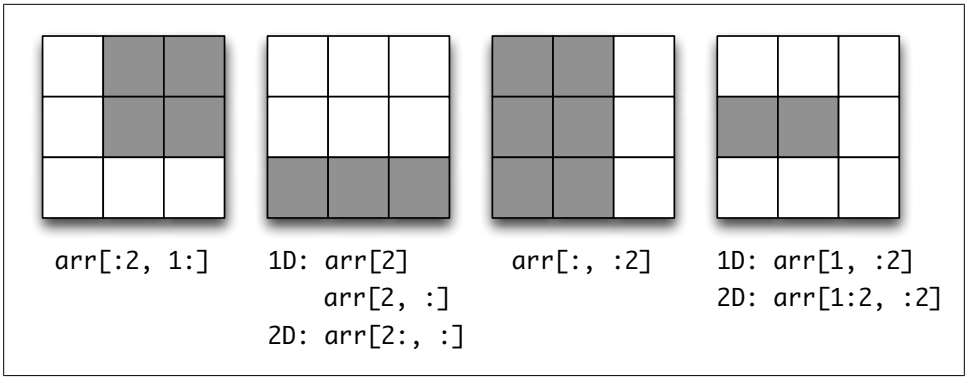


Figure 5-2. 2-dimensional array slicing

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

In [76]: arr2d[:2]
Out[76]:
array([[1, 2, 3],
       [4, 5, 6]])
```

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. You can pass multiple slices just like you can pass multiple indexes:

```
In [77]: arr2d[:2, 1:]
Out[77]:
array([[2, 3],
       [5, 6]])
```

When slicing like this, you always obtain array views of the same dimension. By mixing integer indexes and slices, you get lower dimensional slices:

```
In [78]: arr2d[1, :2]
Out[78]: array([4, 5])

In [79]: arr2d[2, :1]
Out[79]: array([7])
```

See Figure 5-2 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 [80]: arr2d[:, :1]
Out[80]:
array([[1],
       [4],
       [7]])
```

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

```
In [81]: arr2d[:2, 1:] = 0
```

Boolean selection, subsetting, filtering

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

In [83]: data = randn(7, 4)

In [84]: names
Out[84]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
      dtype='|S4')

In [85]: data
Out[85]:
array([[ 0.9317,  0.076 ,  0.8077, -0.9221],
       [-1.0917, -0.884 ,  1.2598, -1.1849],
       [-0.6768, -0.0479,  1.0651,  0.7774],
       [-1.3252,  0.4683, -0.6874, -1.152 ],
       [ 0.9529, -0.3587,  0.3172,  0.8237],
       [-1.4846, -0.6819,  1.3349, -0.7586],
       [ 1.3388, -0.5516,  0.9979,  1.351 ]])
```

Suppose each name corresponds to a row in the `data` array. If we wanted to select all the rows with corresponding name `'Bob'`. Comparing `names` with the string `'Bob'` gives us a boolean array:

```
In [86]: names == 'Bob'
Out[86]: array([ True, False, False,  True, False, False, False], dtype=bool)
```

This boolean array can be pass when indexing the array:

```
In [87]: data[names == 'Bob']
Out[87]:
array([[ 0.9317,  0.076 ,  0.8077, -0.9221],
       [-1.3252,  0.4683, -0.6874, -1.152 ]])
```

The boolean array must be 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 [88]: data[names == 'Bob', 2:]
Out[88]:
array([[ 0.8077, -0.9221],
       [-0.6874, -1.152 ]])

In [89]: data[names == 'Bob', 3]
Out[89]: array([-0.9221, -1.152 ])
```

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

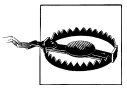
```
In [90]: names != 'Bob'
Out[90]: array([False,  True,  True, False,  True,  True,  True], dtype=bool)
```

```
In [91]: data[-(names == 'Bob')]
Out[91]:
array([[-1.0917, -0.884 ,  1.2598, -1.1849],
       [-0.6768, -0.0479,  1.0651,  0.7774],
       [ 0.9529, -0.3587,  0.3172,  0.8237],
       [-1.4846, -0.6819,  1.3349, -0.7586],
       [ 1.3388, -0.5516,  0.9979,  1.351 ]])
```

Selecting two of the three names To combine multiple boolean conditions, use boolean arithmetic operators like `&` (and) and `|` (or):

```
In [92]: mask = (names == 'Bob') | (names == 'Will')

In [93]: mask
Out[93]: array([ True, False,  True,  True,  True, False, False], dtype=bool)

In [94]: data[mask]
Out[94]:
array([[ 0.9317,  0.076 ,  0.8077, -0.9221],
       [-0.6768, -0.0479,  1.0651,  0.7774],
       [-1.3252,  0.4683, -0.6874, -1.152 ],
       [ 0.9529, -0.3587,  0.3172,  0.8237]])
```

Note that the Python keywords `and` and `or` do not work with boolean arrays.



Selecting data from an array by boolean indexing *always* creates a copy of the data, even if the returned array is unchanged.

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 [95]: data[data < 0] = 0

In [96]: data
Out[96]:
array([[ 0.9317,  0.076 ,  0.8077,  0.    ],
       [ 0.    ,  0.    ,  1.2598,  0.    ],
       [ 0.    ,  0.    ,  1.0651,  0.7774],
       [ 0.    ,  0.4683,  0.    ,  0.    ],
       [ 0.9529,  0.    ,  0.3172,  0.8237],
       [ 0.    ,  0.    ,  1.3349,  0.    ],
       [ 1.3388,  0.    ,  0.9979,  1.351 ]])
```

Setting whole rows or columns using a 1D boolean array is also easy:

```
In [97]: data[names != 'Joe'] = 7

In [98]: data
Out[98]:
array([[ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 0.    ,  0.    ,  1.2598,  0.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
```

```
       [ 0.    ,  0.    ,  1.3349,  0.    ],
       [ 1.3388,  0.    ,  0.9979,  1.351 ]])
```

Fancy indexing

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

```
In [99]: arr = np.empty((8, 4))

In [100]: for i in range(8):
   .....:     arr[i] = i

In [101]: arr
Out[101]:
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 [102]: arr[[4, 3, 0, 6]]
Out[102]:
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 [103]: arr[[-3, -5, -7]]
Out[103]:
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 later
In [104]: arr = np.arange(32).reshape((8, 4))

In [105]: arr
Out[105]:
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 [106]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[106]: array([ 4, 23, 29, 10])
```

Take a moment to understand what just happened. 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 defined by the "cross-product" of the two arrays. Here is one way to get:

```
In [107]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[107]:
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 [108]: arr[np.ix_([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]])
```

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 [109]: arr = np.arange(15).reshape((3, 5))

In [110]: arr
Out[110]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [111]: arr.T
Out[111]:
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 option, like for example computing the inner matrix product X^TX using `np.dot`:

```
In [112]: arr = np.random.randn(6, 3)

In [113]: np.dot(arr.T, arr)
Out[113]:
array([[ 1.2991, -0.0334,  1.4976],
       [-0.0334,  5.5907,  1.5883],
       [ 1.4976,  1.5883,  8.6516]])
```

For higher dimensional arrays, `transpose` will accept a tuple of axis numbers to permute the axes (for extra mind bending):

```
In [114]: arr = np.arange(16).reshape((2, 2, 4))

In [115]: arr
Out[115]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [116]: arr.transpose((1, 0, 2))
Out[116]:
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 [117]: arr
Out[117]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [118]: arr.swapaxes(1, 2)
Out[118]:
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 [119]: arr = np.arange(10)

In [120]: np.sqrt(arr)
Out[120]:
array([ 0.    ,  1.    ,  1.4142,  1.7321,  2.    ,  2.2361,  2.4495,
        2.6458,  2.8284,  3.    ])

In [121]: np.exp(arr)
Out[121]:
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 [122]: x = randn(8)

In [123]: y = randn(8)

In [124]: x
Out[124]:
array([-0.2404,  0.3755,  1.0943, -0.7896, -1.2244,  1.4285,  0.7196,
        0.271 ])

In [125]: y
Out[125]:
array([ 1.7472,  0.3535, -0.5711, -0.8272,  1.4999, -0.3368,  0.5091,
       -1.1059])

In [126]: np.maximum(x, y) # element-wise maximum
Out[126]:
array([ 1.7472,  0.3755,  1.0943, -0.7896,  1.4999,  1.4285,  0.7196,
        0.271 ])
```

While not extremely 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 [127]: arr = randn(7) * 5

In [128]: np.modf(arr)
Out[128]:
(array([ 0.1773, -0.8801,  0.0108,  0.2262,  0.2228,  0.5268,  0.9255]),
 array([ 2., -1.,  0.,  2.,  3.,  4.,  9.]))
```

See Table 5-3 and Table 5-4 for a listing of available ufuncs.

Table 5-3. Unary ufuncs

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

Table 5-4. Binary universal functions

Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Elementwise maximum. `fmax` ignores `NaN`
`minimum, fmin`	Elementwise minimum. `fmin` ignores `NaN`
`mod`	Elementwise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators `>, >=, <, <=, ==, !=`

Function	Description
`logical_and, logi cal_or, logical_xor`	Compute elementwise truth value of logical operation. Equivalent to infix operators & |, ^

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 loops. This practice of replacing explicit loops with array expressions is commonly referred to as *vectorization*. In general, vectorized array operations will often be 1 or 2 (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations.

To really "take it to the next level" with vectorized array operations with NumPy, you should learn more about *broadcasting*, with is detailed in the later chapter on Advanced NumPy

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 array:

```
In [129]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points

In [130]: xs, ys = np.meshgrid(points, points)

In [131]: ys
Out[131]:
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 [133]: z = np.sqrt(xs ** 2 + ys ** 2)

In [134]: z
Out[134]:
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]])
```

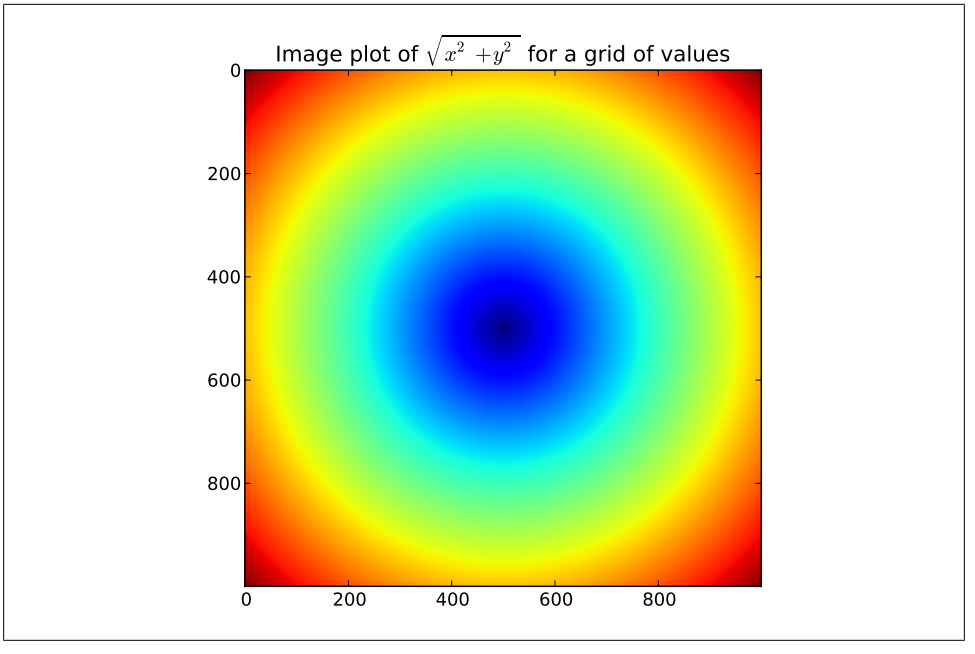


Figure 5-3. Plot of function evaluated on grid

```
In [135]: imshow(z)
Out[135]: <matplotlib.image.AxesImage at 0x41a3d50>

In [136]: title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
```

See Figure 5-3. Here I used the matplotlib function `imshow` to create an image plot from a 2D array of function values.

Expressing conditional logic as array operations

The `numpy.where` function is a vectorized version of the ternary expression `x if condition else y`. Suppose we had a boolean array and two arrays of values:

```
In [137]: xarr = np.array([6, 27, 42, -5, 8])

In [138]: yarr = np.array([-14, 3, 0, 1.5, 12])

In [139]: cond = np.array([True, False, True, True, False])
```

Suppose we wanted to take a value from `xarr` whenever the corresponding value in `cond` is `True` otherwise take the value from `yarr`. A list comprehension doing this might look like:

```
In [140]: result = [x if c else y
   .....:           for x, y, c in zip(xarr, yarr, cond)]
```

```
In [141]: result
Out[141]: [6, 3.0, 42, -5, 12.0]
```

This has multiple problems. First, it will not be very fast for large arrays (because all the work is being done in pure Python). Secondly, it will not work with multidimensional arrays. With `np.where` you can write this very concisely:

```
In [142]: result = np.where(cond, xarr, yarr)

In [143]: result
Out[143]: array([  6.,   3.,  42.,  -5.,  12.])
```

The second and third arguments to `np.where` don't need to be arrays; one or both of them can be scalars. A typical use of `where` in data analysis is to produce a new of values based on another array. Suppose you had a matrix of randomly generated data and you wanted to replace all positive values with 2 and all negative values with -2. This is very easy to do with `np.where`:

```
In [144]: arr = randn(4, 4)

In [145]: arr
Out[145]:
array([[ 0.777 ,  0.3125,  0.0856,  0.024 ],
       [-0.0326, -0.8053,  0.7928, -1.269 ],
       [ 0.2784, -0.3609, -0.3022,  1.9753],
       [ 1.3175,  1.7878,  0.1404,  0.1435]])

In [146]: np.where(arr > 0, 2, -2)
Out[146]:
array([[ 2,  2,  2,  2],
       [-2, -2,  2, -2],
       [ 2, -2, -2,  2],
       [ 2,  2,  2,  2]])

In [147]: np.where(arr > 0, 2, arr) # set only positive values to 2
Out[147]:
array([[ 2.    ,  2.    ,  2.    ,  2.    ],
       [-0.0326, -0.8053,  2.    , -1.269 ],
       [ 2.    , -0.3609, -0.3022,  2.    ],
       [ 2.    ,  2.    ,  2.    ,  2.    ]])
```

The arrays passed to where can be more than just equal sizes array or scalers

With some cleverness you can use where to express more complicated logic; consider this example where I have two boolean arrays, `cond1` and `cond2`, and wish to assign a different for each of the 4 possible pairs of boolean values:

```
result = []
for i in range(n):
    if cond1[i] and cond2[i]:
        result.append(0)
    elif cond1[i]:
        result.append(1)
    elif cond2[i]:
        result.append(2)
```

```
    else:
        result.append(3)
```

While perhaps not immediately obvious, this `for` loop can be converted into a nested `where` expression:

```
np.where(cond1 & cond2, 0,
         np.where(cond1, 1,
                  np.where(cond2, 2, 3)))
```

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 alternately be expressed 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 [148]: arr = np.random.randn(5, 4) # normally-distributed data

In [149]: arr.mean()
Out[149]: -0.048197187161194303

In [150]: np.mean(arr)
Out[150]: -0.048197187161194303

In [151]: arr.sum()
Out[151]: -0.9639437432238861
```

Functions like `mean` and `sum` take an optional `axis` argument which computes the statistic over the passed axis, resulting in an array with one less dimension:

```
In [152]: arr.mean(axis=1)
Out[152]: array([ 0.139 ,  0.5474, -0.8802, -0.3675,  0.3204])

In [153]: arr.sum(0)
Out[153]: array([-2.7259, -1.4553,  0.0839,  3.1333])
```

Other methods like `cumsum` and `cumprod` do not aggregate, instead producing an array of the intermediate results:

```
In [154]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [155]: arr.cumsum(0)
Out[155]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])
```

```
In [156]: arr.cumprod(1)
Out[156]:
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]])
```

See Table 5-5 for a full listing. We'll see many examples of these methods in action in later chapters.

Table 5-5. Basic array statistical methods

Method	Description
`sum`	Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.
`mean`	Arithmetic mean. Zero-length arrays have NaN mean.
`std, var`	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n).
`min, max`	Minimum and maximum.
`argmin, argmax`	Indices of minimum and maximum elements, respectively.
`cumsum`	Cumulative sum of elements starting from 0
`cumprod`	Cumulative product of elements starting from 1

Methods for boolean arrays

Boolean values are coerced to 1 (`True`) and 0 (`False`) in the above methods. Thus, `sum` is often used as a means of counting `True` values in a boolean array:

```
In [157]: arr = randn(100)

In [158]: (arr > 0).sum() # Number of positive values
Out[158]: 58
```

There are two additional methods, `any` and `all`, useful especially for boolean arrays. `any` tests whether one or more values in an array is `True`, while `all` checks if every value is `True`:

```
In [159]: bools = np.array([False, False, True, False])

In [160]: bools.any()
Out[160]: True

In [161]: bools.all()
Out[161]: False
```

These methods also work with non-boolean arrays, where non-zero elements evaluate to `True`.

Sorting

Like Python's built-in list type, NumPy arrays can be sorted in-place using the `sort` method:

```
In [162]: arr = randn(8)

In [163]: arr
Out[163]:
array([ 0.5464, -0.1103, -1.0037, -0.4231, -0.7327,  0.0247, -0.0568,
        0.2414])

In [164]: arr.sort()

In [165]: arr
Out[165]:
array([-1.0037, -0.7327, -0.4231, -0.1103, -0.0568,  0.0247,  0.2414,
        0.5464])
```

Multidimensional arrays can have each 1D section of values sorted in-place along an axis by passing the axis number to `sort`:

```
In [166]: arr = randn(5, 3)

In [167]: arr
Out[167]:
array([[-0.7309,  0.6969,  0.1067],
       [ 1.7424,  0.8691, -1.6146],
       [-1.0013,  0.7109,  1.1533],
       [ 1.1051, -1.665 ,  0.6037],
       [-2.697 , -0.3173, -0.8175]])

In [168]: arr.sort(1)

In [169]: arr
Out[169]:
array([[-0.7309,  0.1067,  0.6969],
       [-1.6146,  0.8691,  1.7424],
       [-1.0013,  0.7109,  1.1533],
       [-1.665 ,  0.6037,  1.1051],
       [-2.697 , -0.8175, -0.3173]])
```

The top level method `np.sort` returns a sorted copy of an array. A quick and dirty way to compute the quantiles of an array is to sort it and select the value at a particular rank:

```
In [170]: large_arr = randn(1000)

In [171]: large_arr.sort()

In [172]: large_arr[int(0.05 * len(large_arr))] # 5% quantile
Out[172]: -1.5667911405755599
```

For more details on using NumPy's sorting methods, and more advanced techniques like indirect sorts, see the Advanced NumPy chapter. Several other kinds of data ma-

nipulations related to sorting (for example, sorting a table of data by one or more columns) are also to be found in pandas.

Unique and other set logic

NumPy has some basic set operations are for 1-dimensional ndarrays. Probably the most commonly used one is `np.unique`, which returns the sorted unique values in an array:

```
In [173]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [174]: np.unique(names)
Out[174]:
array(['Bob', 'Joe', 'Will'],
      dtype='|S4')

In [175]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [176]: np.unique(ints)
Out[176]: array([1, 2, 3, 4])
```

Another function, `np.in1d`, tests membership of the values in one array in another, returning a boolean array:

```
In [177]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [178]: np.in1d(values, [2, 3, 6])
Out[178]: array([ True, False, False,  True,  True, False,  True], dtype=bool)
```

See Table 5-6 for a listing of set functions in NumPy.

Table 5-6. Array set operations

Method	Description
`unique(x)`	Compute the sorted, unique elements in x
`intersect1d(x, y)`	Compute the sorted, common elements in x and y
`union1d(x, y)`	Compute the sorted union of elements
`in1d(x, y)`	Compute a boolean array indicating whether each element of x is contained in y
`setdiff1d(x, y)`	Set difference, elements in x that are not in y
`setxor1d(x, y)`	Set symmetric differences; elements that are in either of the arrays, but not both

File input and output with arrays

NumPy is able to save and load data to and from disk either in text or binary format. In later chapters you will learn about tools in pandas for reading tabular data into memory

Storing arrays on disk in binary format

`np.save` and `np.load` are the two workhorse functions for efficiently saving and loading array data on disk. Arrays are saved by default in a special efficient binary format with file extension `.npy`.

```
In [179]: arr = np.arange(10)

In [180]: np.save('some_array', arr)
```

If the file path does not already end in `.npy`, the extension will be appended. The array on disk can then be loaded using `np.load`:

```
In [181]: np.load('some_array.npy')
Out[181]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

You save multiple arrays in a zip archive using `np.savez` and passing the arrays as keyword arguments:

```
In [182]: np.savez('array_archive.npz', a=arr, b=arr)
```

When loading an `.npz` file, you get back a dict-like object which loads the individual arrays lazily:

```
In [183]: arch = np.load('array_archive.npz')

In [184]: arch['b']
Out[184]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Saving and loading text files

Loading text from files is a fairly standard task. The landscape of file reading and writing functions in Python can be a bit confusing for a newcomer, so I will focus mainly on the `read_csv` and `read_table` functions in pandas. It will at times be useful to load data into vanilla NumPy arrays using `np.loadtxt` or the more specialized `np.genfromtxt`.

These functions have many options allowing you to specify different delimiters, converter functions for certain columns, skipping rows, and other things. Take a simple case of a comma-separated file (CSV) like this:

```
In [187]: !cat array_ex.txt
0.580052,0.186730,1.040717,1.134411
0.194163,-0.636917,-0.938659,0.124094
-0.126410,0.268607,-0.695724,0.047428
-1.484413,0.004176,-0.744203,0.005487
2.302869,0.200131,1.670238,-1.881090
-0.193230,1.047233,0.482803,0.960334
```

This can be loaded into a 2D array like so:

```
In [188]: arr = np.loadtxt('array_ex.txt', delimiter=',')

In [189]: arr
Out[189]:
array([[ 0.5801,  0.1867,  1.0407,  1.1344],
```

```
[ 0.1942, -0.6369, -0.9387,  0.1241],
[-0.1264,  0.2686, -0.6957,  0.0474],
[-1.4844,  0.0042, -0.7442,  0.0055],
[ 2.3029,  0.2001,  1.6702, -1.8811],
[-0.1932,  1.0472,  0.4828,  0.9603]])
```

`np.savetxt` performs the inverse operation: writing an array to a delimited text file. `genfromtxt` is similar to `loadtxt` but is geared for structured arrays and missing data handling; see the Advanced NumPy chapter for more on structured arrays.

For more on file reading and writing, especially tabular or spreadsheet-like data, see the later chapters involving pandas and DataFrame objects.

Linear algebra

Linear algebra, like matrix multiplication, decompositions, determinants, and other square matrix math, is an important part of any array library. Unlike some languages like MATLAB, multiplying two two-dimensional arrays with * is an elementwise product instead of a matrix dot product. As such, there is a function `dot`, both an array method, and a function in the `numpy` namespace, for matrix multiplication:

```
In [190]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [191]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [192]: x
Out[192]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [193]: y
Out[193]:
array([[  6.,  23.],
       [ -1.,   7.],
       [  8.,   9.]])

In [194]: x.dot(y)
Out[194]:
array([[  28.,   64.],
       [  67.,  181.]])

In [195]: np.dot(x, y) # equivalent
Out[195]:
array([[  28.,   64.],
       [  67.,  181.]])
```

A matrix product between a 2D array and a suitably sized 1D array results in a 1D array:

```
In [196]: np.dot(x, np.ones(3))
Out[196]: array([  6.,  15.])
```

`numpy.linalg` has a standard set of matrix decompositions and things like inverse and determinant. These are implemented under the hood using the same industry-standard Fortran libraries used in other languages like MATLAB and R, such as like BLAS, LAPACK, or possibly (depending on your NumPy build) the Intel MKL:

```
In [198]: from numpy.linalg import inv, qr

In [199]: X = randn(5, 5)

In [200]: mat = X.T.dot(X)

In [201]: inv(mat)
Out[201]:
array([[ 3.0361, -0.1808, -0.6878, -2.8285, -1.1911],
       [-0.1808,  0.5035,  0.1215,  0.6702,  0.0956],
       [-0.6878,  0.1215,  0.2904,  0.8081,  0.3049],
       [-2.8285,  0.6702,  0.8081,  3.4152,  1.1557],
       [-1.1911,  0.0956,  0.3049,  1.1557,  0.6051]])

In [202]: mat.dot(inv(mat))
Out[202]:
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 [203]: q, r = qr(mat)

In [204]: r
Out[204]:
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 5-7 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 5-7. Commonly-used NumPy linear algebra functions

Function	Description
`diag`	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
`dot`	Matrix multiplication
`trace`	Compute the sum of the diagonal elements

Function	Description
linalg.det	Compute the matrix determinant
linalg.eig	Compute the eigenvalues and eigenvectors of a square matrix
linalg.inv	Compute the inverse of a square matrix
linalg.pinv	Compute the Moore-Penrose pseudo-inverse inverse of a square matrix
linalg.qr	Compute the QR decomposition
linalg.svd	Compute the singular value decomposition (SVD)
linalg.solve	Solve the linear system Ax = b for x, where A is a square matrix
linalg.lstsq	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 sampling whole arrays of 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 [205]: samples = np.random.normal(size=(4, 4))

In [206]: samples
Out[206]:
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 [16]: N = 1000000

In [17]: timeit samples = [random.normalvariate(0, 1) for _ in xrange(N)]
1 loops, best of 3: 1.33 s per loop

In [18]: timeit np.random.normal(size=N)
10 loops, best of 3: 59.4 ms per loop
```

See table Table 5-8 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 5-8. Partial list of numpy.random functions

Function	Description
seed	Seed the random number generator
permutation	Return a random permutation of a sequence, or return a permuted range

Function	Description
shuffle	Randomly permute a sequence in place
rand	Draw samples from a uniform distribution
randint	Draw random integers from a given low-to-high range
randn	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
binomial	Draw samples a binomial distribution
normal	Draw samples from a normal (Gaussian) distribution
beta	Draw samples from a beta distribution
chisquare	Draw samples from a chi-square distribution
gamma	Draw samples from a gamma distribution
uniform	Draw samples from a uniform [0, 1) distribution

Example: Random Walks

An illustrative application of utilizing array operations is in the simulation of random walks. Let's first consider a simple random walk starting at 0 with steps of 1 and -1 occurring with equal probability. A pure Python way to implement a single random walk with 1000 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 5-4 for an example plot of the first 100 values on one of these random walks.

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 1000 coin flips at once, set these to 1 and -1, and compute the cumulative sum:

```
In [208]: nsteps = 1000

In [209]: draws = np.random.randint(0, 2, size=nsteps)

In [210]: steps = np.where(draws > 0, 1, -1)

In [211]: walk = steps.cumsum()
```

From this we can begin to extract statistics like the minimum and maximum value along the walk's trajectory:

```
In [212]: walk.min()
Out[212]: -3
```

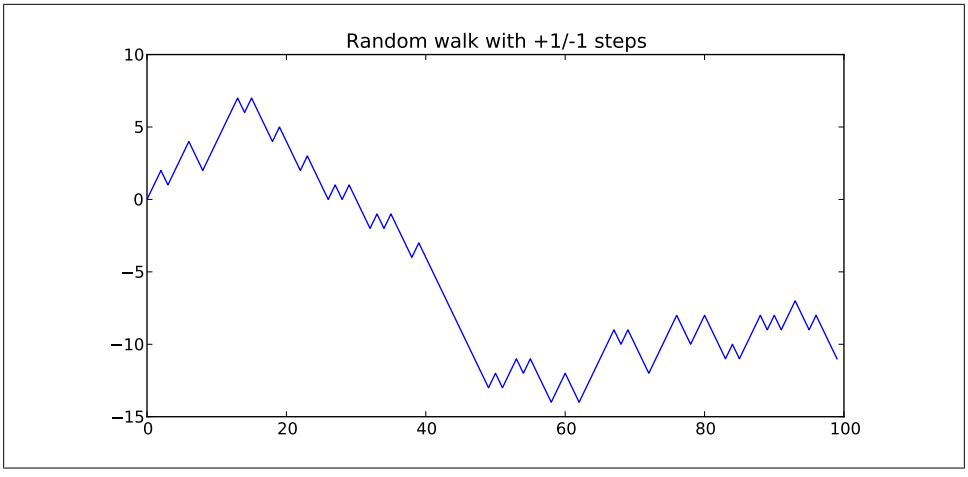


Figure 5-4. A simple random walk

```
In [213]: walk.max()
Out[213]: 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 [214]: (np.abs(walk) >= 10).argmax()
Out[214]: 37
```

Simulating many random walks at once

If your goal was to simulate many random walks, say 5000 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 5000 random walks in one shot:

```
In [215]: nwalks = 5000

In [216]: nsteps = 1000

In [217]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 or 1

In [218]: steps = np.where(draws > 0, 1, -1)

In [219]: walks = steps.cumsum(1)

In [220]: walks
```

```
Out[220]:
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 [221]: walks.max()
Out[221]: 138

In [222]: walks.min()
Out[222]: -133
```

Out of these walks, let's compute the minimum crossing time to 30 or -30. This is slightly tricky because not all 5000 of them reach 30. We can check this using the `any` method:

```
In [223]: hits30 = (np.abs(walks) >= 30).any(1)

In [224]: hits30
Out[224]: array([False,  True, False, ..., False,  True, False], dtype=bool)

In [225]: hits30.sum() # Number that hit 30 or -30
Out[225]: 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 [226]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)

In [227]: crossing_times.mean()
Out[227]: 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 [228]: steps = np.random.normal(loc=0, scale=0.25,
   .....:                          size=(nwalks, nsteps))
```

CHAPTER 6

Overview and First Steps with pandas

CHAPTER 7

Data loading and storage

CHAPTER 8

Data Wrangling and Text Processing

CHAPTER 9

Reshaping, aggregation, and transformation

```
Filling and interpolating missing data
--------------------------------------

Trimming and filtering outliers
-------------------------------

Removing duplicates
-------------------

Computations with missing data
------------------------------

Renaming, mapping, and transformations
--------------------------------------

Examples
--------
```

CHAPTER 10

Data Aggregation and Group Operations

```
Group By Fundamentals
---------------------

Aggregation
-----------

Transformation
--------------

Flexible application
--------------------

Applications
------------


Examples
--------
```

CHAPTER 11

Plotting and Visualization

CHAPTER 12
Time series

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

- *Timestamps*, specific instants in time
- Fixed *periods*, such as the month January 2007 or the full year 2010.
- *Intervals* of time, indicated by a start and end timestamp. Periods can be thought of as special cases of intervals.
- Experiment time; a measure of elapsed time with respect to 0 since the start of an experiment

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 ones indexed by timestamps.

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, 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 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 one of the most oft-occurring:

```
In [12]: from datetime import datetime

In [13]: now = datetime.now()

In [14]: now
Out[14]: datetime.datetime(2012, 5, 5, 16, 17, 28, 652302)

In [15]: now.year, now.month, now.day
Out[15]: (2012, 5, 5)
```

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

```
In [16]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)

In [17]: delta
Out[17]: datetime.timedelta(926, 56700)

In [18]: delta.days
Out[18]: 926

In [19]: delta.seconds
Out[19]: 56700
```

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

```
In [20]: from datetime import timedelta

In [21]: start = datetime(2011, 1, 7)

In [22]: start + timedelta(12)
Out[22]: datetime.datetime(2011, 1, 19, 0, 0)

In [23]: start - 2 * timedelta(12)
Out[23]: datetime.datetime(2010, 12, 14, 0, 0)
```

The data types in the `datetime` module are summarized in Table 12-1. 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 12-1. Types in datetime module

Type	Description
`date`	Store calendar date (year, month, day) using the Gregorian calendar from 1 C.E.
`time`	Store time of day as hours, minutes, seconds, and microseconds

Type	Description
`datetime`	Stores both date and time
`timedelta`	Represents the difference between two datetime values (as days, seconds, and microseconds)

Converting Between string to datetime

`datetime` objects (and later, pandas `Timestamp` objects) can be formatted as strings using `str` or the `strftime` method, passing a format specification:

```
In [24]: stamp = datetime(2011, 1, 3)

In [25]: str(stamp)
Out[25]: '2011-01-03 00:00:00'

In [26]: stamp.strftime('%Y-%m-%d')
Out[26]: '2011-01-03'
```

See Table 12-2 for a complete list of the format codes. These same format codes can be used to convert strings to dates using `datetime.strptime`:

```
In [27]: value = '2011-01-03'

In [28]: datetime.strptime(value, '%Y-%m-%d')
Out[28]: datetime.datetime(2011, 1, 3, 0, 0)

In [29]: datestrs = ['7/6/2011', '8/6/2011']

In [30]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]
Out[30]: [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 3rd party `dateutil` package:

```
In [31]: from dateutil.parser import parse

In [32]: parse('2011-01-03')
Out[32]: datetime.datetime(2011, 1, 3, 0, 0)
```

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

```
In [33]: parse('Jan 31, 1997 10:45 PM')
Out[33]: 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 [34]: parse('6/12/2011', dayfirst=True)
Out[34]: datetime.datetime(2011, 12, 6, 0, 0)
```

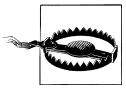
pandas is generally oriented toward working with arrays of dates, whether used as an axis index or a column in a DataFrame. The `to_datetime` method wraps the `dateutil` smart parsing capability to parse lists and arrays of strings.

```
In [35]: from pandas import to_datetime

In [36]: datestrs
Out[36]: ['7/6/2011', '8/6/2011']

In [37]: to_datetime(datestrs)
Out[37]: array([2011-07-06 00:00:00, 2011-08-06 00:00:00], dtype=datetime64[us])

In [38]: datestrs
Out[38]: ['7/6/2011', '8/6/2011']
```

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

```
# TODO: fixme per NaT / NumPy >= 1.7
In [39]: to_datetime(datestrs + [None])
Out[39]:
array([2011-07-06 00:00:00, 2011-08-06 00:00:00,
       32103-01-09 19:59:05.224192], dtype=datetime64[us])
```

`NaT` is NumPy's NA value for `datetime64`.



`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 12-2. Datetime format specification

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

Type	Description
%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. See Table 12-3 for a listing.

Table 12-3. 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 often represented external to pandas as Python strings or `datetime` objects:

```
In [40]: from datetime import datetime

In [41]: from pandas import *

In [42]: 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 [43]: ts = Series(np.random.randn(6), index=dates)

In [44]: ts
Out[44]:
2011-01-02    0.037581
2011-01-05    0.110537
2011-01-07   -0.762574
2011-01-08   -0.964972
2011-01-10    0.724459
2011-01-12   -0.596480
```

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

```
In [45]: type(ts)
Out[45]: pandas.core.series.TimeSeries
```

```
In [46]: ts.index
Out[46]:
DatetimeIndex([2011-01-02 00:00:00, 2011-01-05 00:00:00, 2011-01-07 00:00:00,
       2011-01-08 00:00:00, 2011-01-10 00:00:00, 2011-01-12 00:00:00], dtype=datetime64[us])
```

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

```
In [47]: ts + ts[::2]
Out[47]:
2011-01-02    0.075163
2011-01-05         NaN
2011-01-07   -1.525148
2011-01-08         NaN
2011-01-10    1.448918
2011-01-12         NaN
```

pandas stores timestamps using NumPy's `datetime64` dtype at the microsecond resolution:

```
In [48]: ts.index.dtype
Out[48]: dtype('datetime64[us]')
```

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

```
In [49]: stamp = ts.index[0]

In [50]: stamp
Out[50]: Timestamp(2011, 1, 2, 0, 0)
```

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 [51]: stamp = ts.index[2]

In [52]: ts[stamp]
Out[52]: -0.76257392255211698
```

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

```
In [53]: ts['1/10/2011']
Out[53]: 0.72445912650451649

In [54]: ts['20110110']
Out[54]: 0.72445912650451649
```

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

```
In [55]: longer_ts = Series(np.random.randn(1000),
   ....:                    index=date_range('1/1/2000', periods=1000))

In [56]: longer_ts
Out[56]:
2000-01-01   -1.864031
2000-01-02   -0.876941
2000-01-03    2.098564
2000-01-04    0.647432
...
2002-09-23    0.884062
2002-09-24    1.827827
2002-09-25    1.079432
2002-09-26    0.573749
Freq: D, Length: 1000

In [57]: longer_ts['2001']
Out[57]:
2001-01-01   -0.174135
2001-01-02   -0.603083
2001-01-03   -1.027874
2001-01-04    1.108478
...
2001-12-28    0.750293
2001-12-29    0.260371
2001-12-30    0.380072
2001-12-31    1.060716
Freq: D, Length: 365

In [58]: longer_ts['2001-05']
Out[58]:
2001-05-01    0.723757
2001-05-02   -0.171533
2001-05-03   -0.034109
2001-05-04    0.434207
...
2001-05-28    0.561820
2001-05-29   -0.601611
2001-05-30    0.171724
2001-05-31    0.337958
Freq: D, Length: 31
```

Slicing with dates works just like with a regular Series:

```
In [59]: ts[datetime(2011, 1, 7):]
Out[59]:
2011-01-07   -0.762574
2011-01-08   -0.964972
2011-01-10    0.724459
2011-01-12   -0.596480
```

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

```
In [60]: ts
Out[60]:
```

```
2011-01-02    0.037581
2011-01-05    0.110537
2011-01-07   -0.762574
2011-01-08   -0.964972
2011-01-10    0.724459
2011-01-12   -0.596480

In [61]: ts['1/6/2011':'1/11/2011']
Out[61]:
2011-01-07   -0.762574
2011-01-08   -0.964972
2011-01-10    0.724459
```

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 [62]: ts.truncate(after='1/9/2011')
Out[62]:
2011-01-02    0.037581
2011-01-05    0.110537
2011-01-07   -0.762574
2011-01-08   -0.964972
```

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

```
In [63]: long_df = DataFrame(np.random.randn(100, 4),
   ....:                     index=date_range('1/1/2000', periods=100, freq='W-WED'),
   ....:                     columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [64]: long_df.ix['5-2001']
Out[64]:
            Colorado     Texas  New York      Ohio
2001-05-02  0.818398 -0.341768  0.097921  0.877285
2001-05-09  0.210619 -0.222197  0.140245  0.431561
2001-05-16  1.392955  0.563072 -2.299041  0.708001
2001-05-23  0.510172  0.087207  1.297072  0.569280
2001-05-30 -2.356698  0.067923 -0.060318 -0.605804
```

Time series with duplicate dates

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

```
In [65]: dates = DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000', '1/2/2000',
   ....:                        '1/3/2000'])

In [66]: dup_ts = Series(np.arange(5), index=dates)

In [67]: dup_ts
Out[67]:
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 [68]: dup_ts.index.is_unique
Out[68]: False
```

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

```
In [69]: dup_ts['1/3/2000']  # not duplicated
Out[69]: 4

In [70]: dup_ts['1/2/2000']  # duplicated
Out[70]:
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 [71]: dup_ts.groupby(level=0).count()
Out[71]:
key_0
2000-01-01    1
2000-01-02    3
2000-01-03    1

In [72]: dup_ts.groupby(level=0).mean()
Out[72]:
key_0
2000-01-01    0
2000-01-02    2
2000-01-03    4
```

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 writing:

```
In [73]: ts
Out[73]:
2011-01-02    0.037581
2011-01-05    0.110537
2011-01-07   -0.762574
2011-01-08   -0.964972
2011-01-10    0.724459
```

```
2011-01-12   -0.596480

In [74]: ts.resample('D')
Out[74]:
2011-01-02    0.037581
2011-01-03         NaN
2011-01-04         NaN
2011-01-05    0.110537
2011-01-06         NaN
2011-01-07   -0.762574
2011-01-08   -0.964972
2011-01-09         NaN
2011-01-10    0.724459
2011-01-11         NaN
2011-01-12   -0.596480
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 above without explanation, you may have guessted that `pandas.date_range` is responsible for generating a `DatetimeIndex` with indicated length according to a particular frequency:

```
In [75]: index = date_range('4/1/2012', '6/1/2012')

In [76]: index
Out[76]:
<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 [77]: date_range(start='4/1/2012', periods=20)
Out[77]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-04-01 00:00:00, ..., 2012-04-20 00:00:00]
Length: 20, Freq: D, Timezone: None

In [78]: date_range(end='6/1/2012', periods=20)
Out[78]:
<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 and only dates falling on or inside the date interval will be included:

```
In [79]: date_range('1/1/2000', '12/1/2000', freq='BM')
Out[79]:
<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 [80]: date_range('5/2/2012 12:56:31', periods=5)
Out[80]:
<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 [81]: date_range('5/2/2012 12:56:31', periods=5, normalize=True)
Out[81]:
<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 [82]: from pandas.tseries.api import Hour, Minute

In [83]: hour = Hour()

In [84]: hour
Out[84]: <1 Hour>
```

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

```
In [85]: four_hours = Hour(4)

In [86]: four_hours
Out[86]: <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 [87]: date_range('1/1/2000', '1/3/2000 23:59', freq='4h')
Out[87]:
<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 [88]: Hour(2) + Minute(30)
Out[88]: <150 Minutes>
```

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

```
In [89]: date_range('1/1/2000', periods=10, freq='1h30min')
Out[89]:
<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 12-4 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 12-4. Base Time Series Frequencies

Alias	Offset Type	Description
D	Day	Calendar daily
B	BusinessDay	Business daily
H	Hour	Hourly
T or min	Minute	Minutely
L or ms	Milli	Millisecond (1/1000th of 1 second)
U	Micro	Microsecond (1/100000th 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.

Alias	Offset Type	Description
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 [90]: rng = date_range('1/1/2012', '9/1/2012', freq='WOM-3FRI')

In [91]: list(rng)
Out[91]:
[Timestamp(2012, 1, 20, 0, 0),
 Timestamp(2012, 2, 17, 0, 0),
 Timestamp(2012, 3, 16, 0, 0),
 Timestamp(2012, 4, 20, 0, 0),
 Timestamp(2012, 5, 18, 0, 0),
 Timestamp(2012, 6, 15, 0, 0),
 Timestamp(2012, 7, 20, 0, 0),
 Timestamp(2012, 8, 17, 0, 0)]
```

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

In [93]: ts
Out[93]:
```

```
2000-01-31   -1.048008
2000-02-29    0.378662
2000-03-31    0.720188
2000-04-30    1.250153
Freq: M

In [94]: ts.shift(2)  # shift forward
Out[94]:
2000-01-31         NaN
2000-02-29         NaN
2000-03-31   -1.048008
2000-04-30    0.378662
Freq: M

In [95]: ts.shift(-2) # shift backward
Out[95]:
2000-01-31    0.720188
2000-02-29    1.250153
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
```

Since 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 [96]: ts.shift(2, freq='M')
Out[96]:
2000-03-31   -1.048008
2000-04-30    0.378662
2000-05-31    0.720188
2000-06-30    1.250153
Freq: M
```

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

```
In [97]: ts.shift(3, freq='D')
Out[97]:
2000-02-03   -1.048008
2000-03-03    0.378662
2000-04-03    0.720188
2000-05-03    1.250153

In [98]: ts.shift(1, freq='3D')  # equivalently
Out[98]:
2000-02-03   -1.048008
2000-03-03    0.378662
2000-04-03    0.720188
2000-05-03    1.250153
```

```
In [99]: ts.shift(1, freq='90T')
Out[99]:
2000-01-31 01:30:00   -1.048008
2000-02-29 01:30:00    0.378662
2000-03-31 01:30:00    0.720188
2000-04-30 01:30:00    1.250153
```

Shifting dates with offsets

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

```
In [100]: now = datetime(2011, 11, 17)

In [101]: now + 3 * Day()
Out[101]: 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 [102]: from pandas.tseries.api import MonthEnd

In [103]: now + MonthEnd()
Out[103]: datetime.datetime(2011, 11, 30, 0, 0)

In [104]: now + MonthEnd(2)
Out[104]: datetime.datetime(2011, 12, 31, 0, 0)
```

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

```
In [105]: offset = MonthEnd()

In [106]: offset.rollforward(now)
Out[106]: datetime.datetime(2011, 11, 30, 0, 0)

In [107]: offset.rollback(now)
Out[107]: datetime.datetime(2011, 10, 31, 0, 0)
```

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

```
In [108]: ts = Series(np.random.randn(20),
   .....:             index=date_range('1/15/2000', periods=20, freq='4d'))

In [109]: ts.groupby(offset.rollforward).mean()
Out[109]:
key_0
2000-01-31   -0.144888
2000-02-29   -0.102858
2000-03-31    0.075511
```

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

```
In [110]: ts.resample('M', how='mean')
Out[110]:
2000-01-31   -0.144888
2000-02-29   -0.102858
```

```
2000-03-31    0.075511
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`, 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 timezone names. Time zone names can be found interactively and in the docs:

```
In [111]: import pytz

In [112]: pytz.common_timezones[-5:]
Out[112]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']
```

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

```
In [113]: tz = pytz.timezone('US/Eastern')

In [114]: tz
Out[114]: <DstTzInfo 'US/Eastern' EST-1 day, 19:00:00 STD>
```

Methods in pandas will accept either time zone names or these objects.

Localization and Conversion

By default, time series in pandas are *time zone naive*.

```
In [115]: rng = date_range('3/9/2012 9:30', periods=6, freq='D')

In [116]: ts = Series(np.random.randn(len(rng)), index=rng)

In [117]: ts
Out[117]:
2012-03-09 09:30:00   -1.395758
2012-03-10 09:30:00    1.110732
2012-03-11 09:30:00    0.225506
```

```
2012-03-12 09:30:00   -1.731937
2012-03-13 09:30:00   -1.109232
2012-03-14 09:30:00   -1.836787
Freq: D

In [118]: ts.index
Out[118]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-09 09:30:00, ..., 2012-03-14 09:30:00]
Length: 6, Freq: D, Timezone: None

In [119]: print(ts.index.tz)
None
```

Date ranges can be generated with a time zone:

```
In [120]: date_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC')
Out[120]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-09 09:30:00+00:00, ..., 2012-03-18 09:30:00+00:00]
Length: 10, Freq: D, Timezone: UTC
```

Conversion from naive to *localized* or from one time zone to another is handled by the `tz_convert` method:

```
In [121]: ts_utc = ts.tz_convert('UTC')

In [122]: ts_utc
Out[122]:
2012-03-09 09:30:00+00:00   -1.395758
2012-03-10 09:30:00+00:00    1.110732
2012-03-11 09:30:00+00:00    0.225506
2012-03-12 09:30:00+00:00   -1.731937
2012-03-13 09:30:00+00:00   -1.109232
2012-03-14 09:30:00+00:00   -1.836787
Freq: D

In [123]: ts_utc.index
Out[123]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-09 09:30:00+00:00, ..., 2012-03-14 09:30:00+00:00]
Length: 6, Freq: D, Timezone: UTC
```

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 [124]: ts_eastern = ts.tz_convert('US/Eastern')

In [125]: ts_eastern.tz_convert('UTC')
Out[125]:
2012-03-09 14:30:00+00:00   -1.395758
2012-03-10 14:30:00+00:00    1.110732
2012-03-11 13:30:00+00:00    0.225506
2012-03-12 13:30:00+00:00   -1.731937
2012-03-13 13:30:00+00:00   -1.109232
2012-03-14 13:30:00+00:00   -1.836787
Freq: D
```

```
In [126]: ts_eastern.tz_convert('Europe/Berlin')
Out[126]:
2012-03-09 15:30:00+01:00   -1.395758
2012-03-10 15:30:00+01:00    1.110732
2012-03-11 14:30:00+01:00    0.225506
2012-03-12 14:30:00+01:00   -1.731937
2012-03-13 14:30:00+01:00   -1.109232
2012-03-14 14:30:00+01:00   -1.836787
Freq: D
```

Operations with time series in different time zones

If two time series are combined with different time zones, 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 [127]: rng = date_range('3/7/2012 9:30', periods=10, freq='B')

In [128]: ts = Series(np.random.randn(len(rng)), index=rng)

In [129]: ts
Out[129]:
2012-03-07 09:30:00    0.223385
2012-03-08 09:30:00   -0.093047
2012-03-09 09:30:00    1.092307
2012-03-12 09:30:00   -0.185525
2012-03-13 09:30:00    0.231302
2012-03-14 09:30:00    0.384892
2012-03-15 09:30:00   -0.515611
2012-03-16 09:30:00   -0.541779
2012-03-19 09:30:00    0.335141
2012-03-20 09:30:00    0.160317
Freq: B
```

Periods and Period Arithmetic

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

```
In [130]: p = Period(2007, freq='A-DEC')
```

Conveniently, adding and subtracting integers from periods has the effect of shifting by their frequency:

```
In [131]: p + 5
Out[131]: Period('2012', 'A-DEC')

In [132]: p - 2
Out[132]: Period('2005', 'A-DEC')
```

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

```
In [133]: Period('2014', freq='A-DEC') - p
Out[133]: 7
```

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

```
In [134]: rng = period_range('1/1/2000', '6/30/2000', freq='M')

In [135]: rng
Out[135]:
<class 'pandas.tseries.period.PeriodIndex'>
freq: M
[Jan-2000, ..., Jun-2000]
length: 6
```

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

```
In [136]: Series(np.random.randn(6), index=rng)
Out[136]:
Jan-2000   -0.125782
Feb-2000   -0.814412
Mar-2000   -0.809709
Apr-2000    0.211328
May-2000    0.025454
Jun-2000    1.150870
Freq: M
```

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

```
In [137]: values = ['2001Q3', '2002Q2', '2003Q1']

In [138]: index = PeriodIndex(values, freq='Q-DEC')

In [139]: index
Out[139]:
<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 of end of the year. This is fairly straightforward:

```
In [140]: p = Period('2007', freq='A-DEC')

In [141]: p.asfreq('M', how='start')
Out[141]: Period('Jan-2007', 'M')
```

```
In [142]: p.asfreq('M', how='end')
Out[142]: Period('Dec-2007', '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 12-1 for an illustration of this. For a *fiscal year* ending on a month other than December, the monthly subperiods belonging are different:

```
In [143]: p = Period('2007', freq='A-JUN')

In [144]: p.asfreq('M', 's')  # shorthand for 'start'
Out[144]: Period('Jul-2006', 'M')

In [145]: p.asfreq('M', 'e')
Out[145]: Period('Jun-2007', '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 [146]: p = Period('Aug-2007', 'M')

In [147]: p.asfreq('A-JUN')
Out[147]: Period('2008', 'A-JUN')
```

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

```
In [148]: rng = period_range('2006', '2009', freq='A-DEC')

In [149]: ts = Series(np.random.randn(len(rng)), index=rng)

In [150]: ts
Out[150]:
2006   -0.891737
2007    0.145818
2008   -0.086782
2009    1.206198
Freq: A-DEC

In [151]: ts.asfreq('M', how='s')
Out[151]:
Jan-2006   -0.891737
Jan-2007    0.145818
Jan-2008   -0.086782
Jan-2009    1.206198
Freq: M

In [152]: ts.asfreq('B', how='e')
Out[152]:
29-Dec-2006   -0.891737
31-Dec-2007    0.145818
31-Dec-2008   -0.086782
31-Dec-2009    1.206198
Freq: B
```

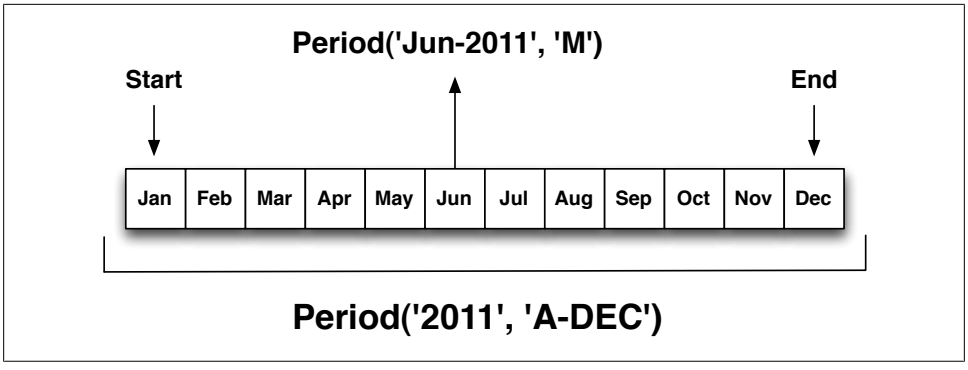


Figure 12-1. Period frequency conversion illustration

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 [153]: p = Period('2012Q4', freq='Q-JAN')

In [154]: p
Out[154]: Period('2013Q3', '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 12-2 for an illustration:

```
In [155]: p.asfreq('D', 'start')
Out[155]: Period('01-Aug-2012', 'D')

In [156]: p.asfreq('D', 'end')
Out[156]: Period('31-Oct-2012', '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 [157]: p4pm = (p.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60

In [158]: p4pm
Out[158]: Period('30-Oct-2012 16:00', 'T')

In [159]: p4pm.to_timestamp()
Out[159]: Timestamp(2012, 10, 30, 16, 0)
```

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

```
In [160]: rng = period_range('2011Q3', '2012Q4', freq='Q-JAN')
```

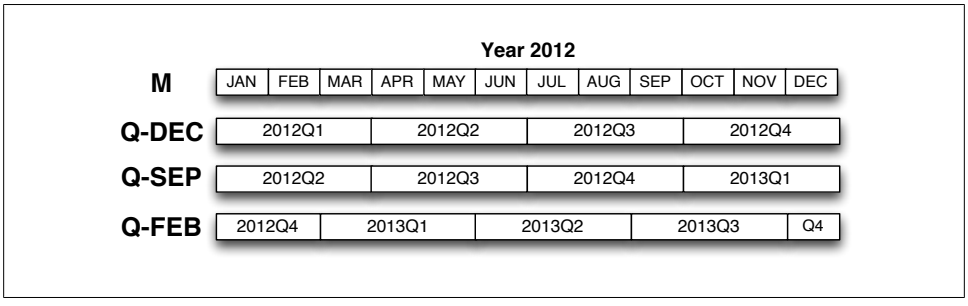


Figure 12-2. Different quarterly frequency conventions

```
In [161]: ts = Series(np.arange(len(rng)), index=rng)

In [162]: ts
Out[162]:
2012Q2    0
2012Q3    1
2012Q4    2
2013Q1    3
2013Q2    4
2013Q3    5
Freq: Q-JAN

In [163]: new_rng = (rng.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60

In [164]: ts.index = new_rng.to_timestamp()

In [165]: ts
Out[165]:
2011-07-28 16:00:00    0
2011-10-28 16:00:00    1
2012-01-30 16:00:00    2
2012-04-27 16:00:00    3
2012-07-30 16:00:00    4
2012-10-30 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 [166]: rng = date_range('1/1/2000', periods=3, freq='M')

In [167]: ts = Series(randn(3), index=rng)

In [168]: ts
Out[168]:
2000-01-31   -0.928956
2000-02-29    1.190408
2000-03-31    1.780740
Freq: M
```

```
In [169]: pts = ts.to_period()

In [170]: pts
Out[170]:
Jan-2000   -0.928956
Feb-2000    1.190408
Mar-2000    1.780740
Freq: M
```

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

```
In [171]: rng = date_range('1/29/2000', periods=6, freq='D')

In [172]: ts = Series(randn(6), index=rng)

In [173]: ts.to_period('M')
Out[173]:
Jan-2000   -0.672532
Jan-2000    0.161767
Jan-2000   -1.196776
Feb-2000    0.015445
Feb-2000    0.046616
Feb-2000   -1.334922
Freq: M
```

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` to `W-FRI` is neither upsampling nor downstampling.

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

```
In [174]: rng = date_range('1/1/2000', periods=100, freq='D')

In [175]: ts = Series(randn(len(rng)), index=rng)

In [176]: ts.resample('M', how='mean')
Out[176]:
2000-01-31   -0.196157
2000-02-29   -0.343955
2000-03-31    0.150365
2000-04-30   -0.301952
Freq: M

In [177]: ts.resample('M', how='mean', kind='period')
```

```
Out[177]:
Jan-2000   -0.196157
Feb-2000   -0.343955
Mar-2000    0.150365
Apr-2000   -0.301952
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 12-5. Resample method arguments

Argument	Description
`freq`	String or DateOffset indicating desired resampled frequency, e.g. 'M', '5min', or `Second(15)`
`how='mean'`	Function name or array function producing aggregated value, for example `'mean'`, `'ohlc'`, `np.max`. Defaults to `'mean'`
`axis=0`	Axis to resample on, default axis=0
`fill_method=None`	How to interpolate when upsampling, as in `'ffill'` or `'bfill'`. By default does no interpolation.
`closed='right'`	In downsampling, which end of each interval is closed (inclusive), `'right'` or `'left'`. Defaults to `'right'`
`label='right'`	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).
`loffset=None`	Time adjustment to the bin labels, such as `'-1s'` / `Second(-1)` to shift the aggregate labels one second earlier
`limit=None`	When forward or backward filling, the maximum number of periods to fill
`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 serise task. The data you're aggregating don't need to be fixed frequency; the desired frequency defines `bin edges` that are used to slice of 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 1-minute data:

```
In [178]: rng = date_range('1/1/2000', periods=12, freq='T')

In [179]: ts = Series(np.arange(12), index=rng)

In [180]: ts
Out[180]:
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 5-minute chunks or *bars* by taking the sum of each group:

```
In [181]: ts.resample('5min', how='sum')
Out[181]:
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 5-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. Passing `closed='left'` changes the interval to be closed on the left:

```
In [182]: ts.resample('5min', how='sum', closed='left')
Out[182]:
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 [183]: ts.resample('5min', how='sum', closed='left', label='left')
Out[183]:
2000-01-01 00:00:00    10
2000-01-01 00:05:00    35
2000-01-01 00:10:00    21
Freq: 5T
```

Lastly, you might want to shift the result index by some amount, say subtracting 1 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 [184]: ts.resample('5min', how='sum', loffset='-1s')
Out[184]:
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
```

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 [185]: ts.resample('5min', how='ohlc')
Out[185]:
                     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
```

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

In [187]: frame[:5]
Out[187]:
            Colorado     Texas  New York      Ohio
2000-01-05  1.091199  0.386905  0.854945 -2.114168
2000-01-12 -1.569938  0.621393 -2.173323 -0.251033
```

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

```
In [188]: df_daily = frame.resample('D')

In [189]: df_daily
Out[189]:
            Colorado     Texas  New York      Ohio
2000-01-05  1.091199  0.386905  0.854945 -2.114168
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 -1.569938  0.621393 -2.173323 -0.251033
```

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 [190]: frame.resample('D', fill_method='ffill')
Out[190]:
            Colorado     Texas  New York      Ohio
2000-01-05  1.091199  0.386905  0.854945 -2.114168
2000-01-06  1.091199  0.386905  0.854945 -2.114168
2000-01-07  1.091199  0.386905  0.854945 -2.114168
2000-01-08  1.091199  0.386905  0.854945 -2.114168
2000-01-09  1.091199  0.386905  0.854945 -2.114168
2000-01-10  1.091199  0.386905  0.854945 -2.114168
2000-01-11  1.091199  0.386905  0.854945 -2.114168
2000-01-12 -1.569938  0.621393 -2.173323 -0.251033
```

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

```
In [191]: frame.resample('D', fill_method='ffill', limit=2)
Out[191]:
            Colorado     Texas  New York      Ohio
2000-01-05  1.091199  0.386905  0.854945 -2.114168
2000-01-06  1.091199  0.386905  0.854945 -2.114168
2000-01-07  1.091199  0.386905  0.854945 -2.114168
2000-01-08  1.091199  0.386905  0.854945 -2.114168
2000-01-09  1.091199  0.386905  0.854945 -2.114168
2000-01-10  1.091199  0.386905  0.854945 -2.114168
2000-01-11  1.091199  0.386905  0.854945 -2.114168
2000-01-12 -1.569938  0.621393 -2.173323 -0.251033
```

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

```
In [192]: frame.resample('W-THU', fill_method='ffill')
Out[192]:
            Colorado     Texas  New York      Ohio
2000-01-06  1.091199  0.386905  0.854945 -2.114168
2000-01-13 -1.569938  0.621393 -2.173323 -0.251033
```

Resampling with periods

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

```
In [193]: frame = DataFrame(np.random.randn(24, 4),
   .....:                   index=period_range('1-2000', '12-2001', freq='M'),
   .....:                   columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [194]: frame[:5]
Out[194]:
          Colorado     Texas  New York      Ohio
```

```
Jan-2000  0.484047 -0.286571 -0.237090 -0.349256
Feb-2000 -0.633293 -1.201630  0.414999 -1.545243
Mar-2000  1.130478  0.362223  1.424480 -0.113922
Apr-2000 -0.742755 -0.712483  1.406768  0.913603
May-2000 -1.258661  0.341150  0.135018  0.144044

In [195]: annual_frame = frame.resample('A-DEC', how='mean')

In [196]: annual_frame
Out[196]:
      Colorado     Texas  New York      Ohio
2000  0.018258 -0.042459  0.446573 -0.317642
2001  0.102721 -0.114773  0.041384 -0.383991
```

Upsampling is slightly more nuanced as you must make a decision about which end of the timespan to start resampling, very much like the `asfreq` method. The `convention` argument defaults to `'end'` but can also be `'start'`:

```
In [197]: annual_frame.resample('Q-DEC', fill_method='ffill')
Out[197]:
        Colorado     Texas  New York      Ohio
2000Q4  0.018258 -0.042459  0.446573 -0.317642
2001Q1  0.018258 -0.042459  0.446573 -0.317642
2001Q2  0.018258 -0.042459  0.446573 -0.317642
2001Q3  0.018258 -0.042459  0.446573 -0.317642
2001Q4  0.102721 -0.114773  0.041384 -0.383991

In [198]: annual_frame.resample('Q-DEC', fill_method='ffill', convention='start')
Out[198]:
        Colorado     Texas  New York      Ohio
2000Q1  0.018258 -0.042459  0.446573 -0.317642
2000Q2  0.018258 -0.042459  0.446573 -0.317642
2000Q3  0.018258 -0.042459  0.446573 -0.317642
2000Q4  0.018258 -0.042459  0.446573 -0.317642
2001Q1  0.102721 -0.114773  0.041384 -0.383991
```

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.

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-MAR`:

```
In [199]: annual_frame.resample('Q-MAR', fill_method='ffill')
Out[199]:
        Colorado     Texas  New York      Ohio
2001Q3  0.018258 -0.042459  0.446573 -0.317642
2001Q4  0.018258 -0.042459  0.446573 -0.317642
2002Q1  0.018258 -0.042459  0.446573 -0.317642
2002Q2  0.018258 -0.042459  0.446573 -0.317642
2002Q3  0.102721 -0.114773  0.041384 -0.383991
```

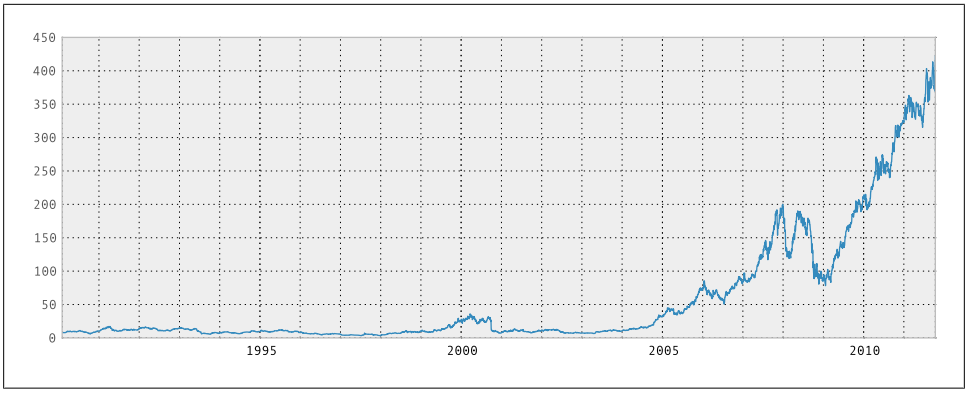


Figure 12-3. AAPL Daily Price

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 [205]: close_px
Out[205]:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 7926 entries, 1990-02-01 00:00:00 to 2011-10-14 00:00:00
Freq: D
Data columns:
AAPL    7926  non-null values
MSFT    7926  non-null values
XOM     7926  non-null values
dtypes: float64(3)
```

Calling `plot` on one of the columns a simple plot, seen in Figure 12-3.

```
In [207]: 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 12-4.

```
In [209]: close_px.ix['2009'].plot()

In [211]: 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 12-6.

```
In [213]: appl_q = close_px['AAPL'].resample('Q-DEC', fill_method='ffill')

In [214]: appl_q.ix['2009':].plot()
```

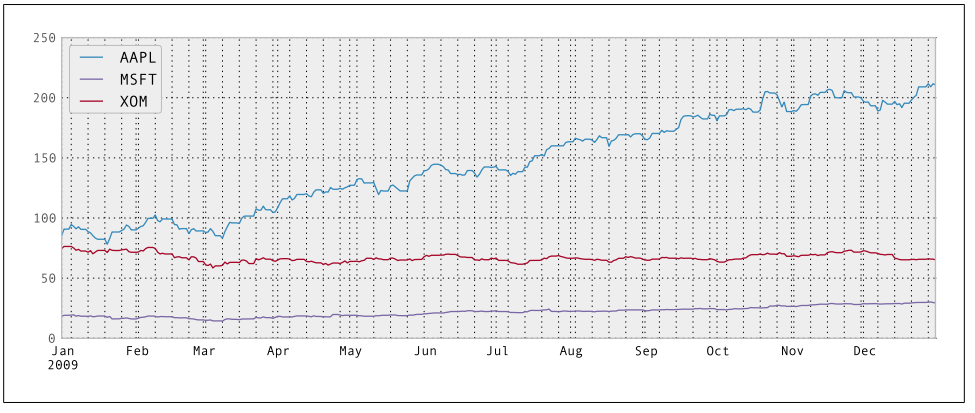


Figure 12-4. Stock Prices in 2009

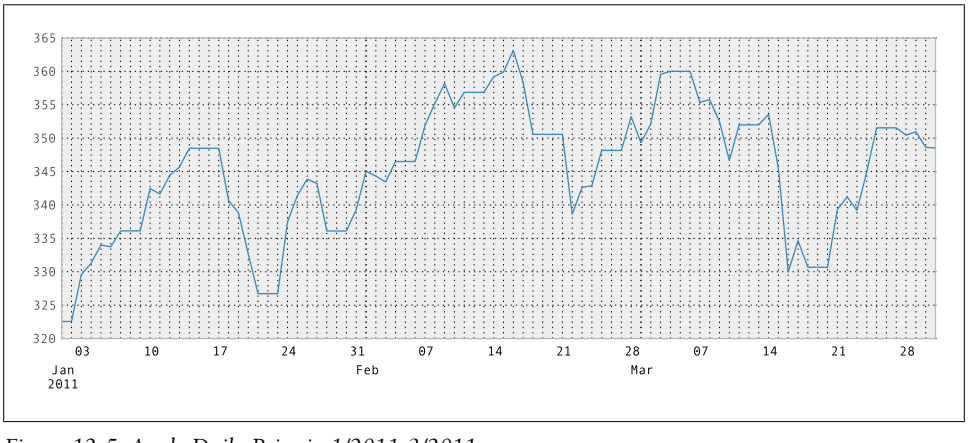


Figure 12-5. Apple Daily Price in 1/2011-3/2011

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

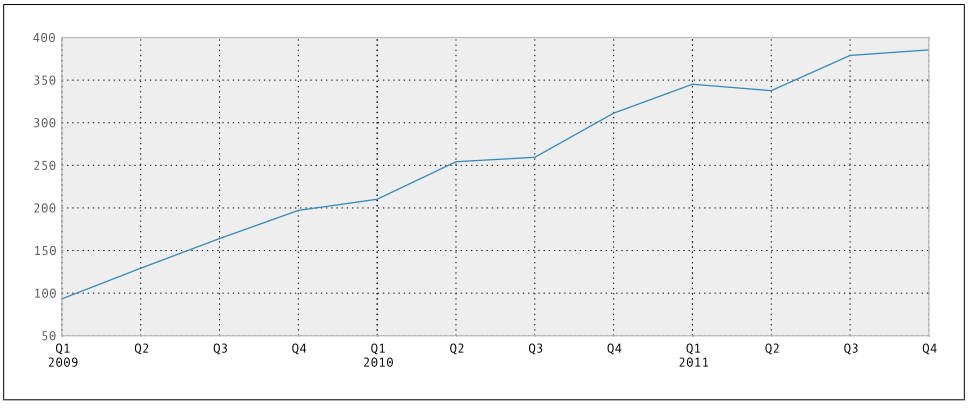


Figure 12-6. Apple Quarterly Price 2009-2011

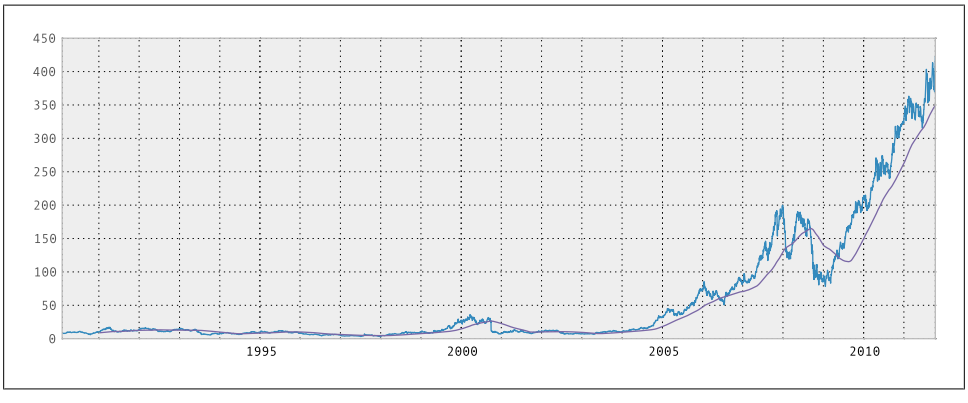


Figure 12-7. Apple Price with 250-day MA

`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 [218]: close_px.AAPL.plot()
Out[218]: <matplotlib.axes.AxesSubplot at 0x41f2c90>

In [219]: rolling_mean(close_px.AAPL, 250).plot()
```

See Figure 12-7 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 12-8):

```
In [221]: appl_std250 = rolling_std(close_px.AAPL, 250, min_periods=10)

In [222]: appl_std250[5:12]
Out[222]:
1990-02-08         NaN
```

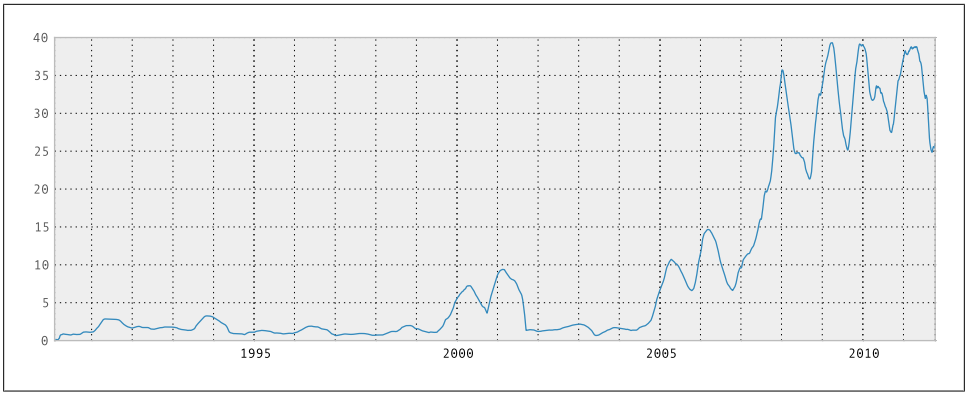


Figure 12-8. Apple Price with 250-day Stdev

```
1990-02-09          NaN
1990-02-12          NaN
1990-02-13          NaN
1990-02-14     0.148189
1990-02-15     0.141003
1990-02-16     0.135454
Freq: B

In [223]: appl_std250.plot()
```

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 [224]: 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 12-9):

```
In [226]: rolling_mean(close_px, 60).plot(logy=True)
```

See Table 12-6 for a listing of related functions in pandas, detailed throughout the rest of this section.

Table 12-6. Moving window functions in pandas

Function	Description
`rolling_count`	
`rolling_sum`	
`rolling_mean`	
`rolling_median`	
`rolling_var, rolling_std`	
`rolling_skew, rolling_kurt`	

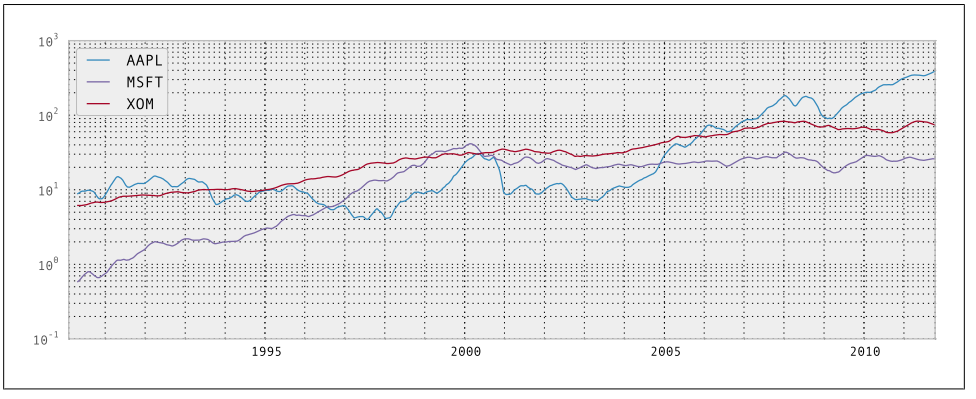


Figure 12-9. Stocks Prices 60-day MA (log Y-axis)

Function	Description
`rolling_min, rolling_max`	
`rolling_quantile`	
`rolling_corr, rolling_cov`	
`rolling_apply`	
`ewma`	
`ewmvar, ewmstd`	
`ewmcorr, ewmcov`	

`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

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 12-10):

```
In [229]: spx_rets = spx_px / spx_px.shift(1) - 1

In [230]: returns = close_px / close_px.shift(1) - 1

In [231]: corr = rolling_corr(returns.AAPL, spx_rets, 125)
```

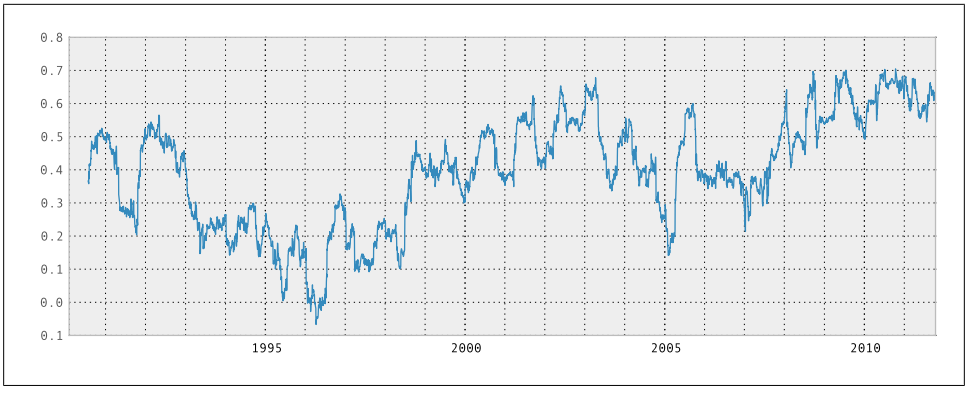


Figure 12-10. 6-month AAPL return correlation to S&P 500

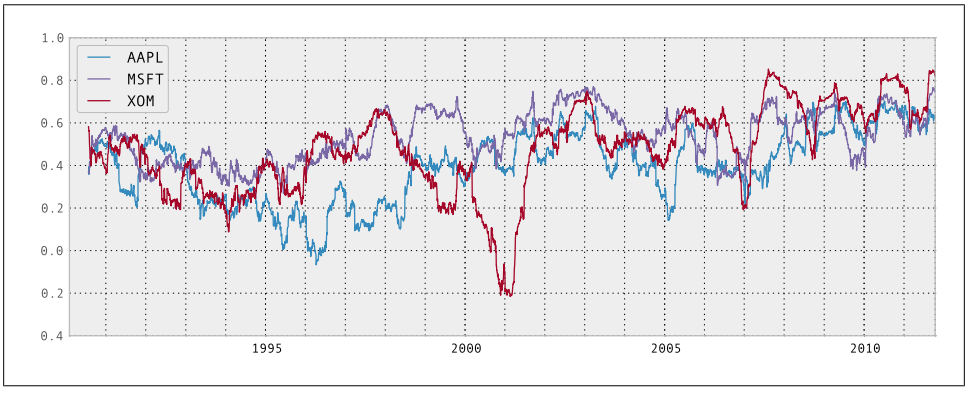


Figure 12-11. 6-month return correlations to S&P 500

```
In [232]: corr.plot()
```

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 12-11 for the plot of the result:

```
In [234]: corr = rolling_corr(returns, spx_rets, 125)

In [235]: corr.plot()
```

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

Figure 12-12. Percentile rank of 2% AAPL return over 1 year window

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 [237]: from scipy.stats import percentileofscore

In [238]: score_at_2percent = lambda x: percentileofscore(x, 0.02)

In [239]: result = rolling_apply(returns.AAPL, 250, score_at_2percent)

In [240]: result.plot()
```

Fixed time length windows

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 [241]: rng = date_range('1/1/2000', periods=10000000, freq='10ms')

In [242]: ts = Series(randn(len(rng)), index=rng)

In [243]: ts
Out[243]:
2000-01-01 00:00:00          -1.439631
2000-01-01 00:00:00.010000    0.520146
2000-01-01 00:00:00.020000   -2.213762
2000-01-01 00:00:00.030000   -0.087513
...
2000-01-02 03:46:39.960000   -1.571317
2000-01-02 03:46:39.970000   -0.833588
2000-01-02 03:46:39.980000   -0.427000
2000-01-02 03:46:39.990000    0.175047
Freq: 10L, Length: 10000000

In [244]: ts.resample('15min', how='ohlc')
Out[244]:
<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 [245]: %timeit ts.resample('15min', how='ohlc')
10 loops, best of 3: 63.3 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 [246]: rng = date_range('1/1/2000', periods=10000000, freq='1s')

In [247]: ts = Series(randn(len(rng)), index=rng)

In [248]: %timeit ts.resample('15s', how='ohlc')
10 loops, best of 3: 95.9 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.

CHAPTER 13

Application: Financial Data

CHAPTER 14

Case Study: US Baby Names 1880-2010

To get the most of this and all of the other case studies, I encourage you to obtain the data set and recreate the analysis presented while reading through the chapter.

The United States Social Security Administration (SSA) has made available data on the frequency of baby names from 1880 through the present. Hadley Wickham, a statistics professor at Rice University and popular R package author, has often made use of this data set in illustrating data manipulation in R.

```
In [4]: names.head(10)
Out[4]:
        name sex  births  year
0       Mary   F    7065  1880
1       Anna   F    2604  1880
2       Emma   F    2003  1880
3  Elizabeth   F    1939  1880
4     Minnie   F    1746  1880
5   Margaret   F    1578  1880
6        Ida   F    1472  1880
7      Alice   F    1414  1880
8     Bertha   F    1320  1880
9      Sarah   F    1288  1880
```

There are many things you might want to do with the data set:

- Visualize the proportion of babies given a particular name (your own, or another name) over time.
- Determine the relative rank of a name.
- Determine the most popular names in each year or the names with largest increases or decreases.
- Analyze trends in names: vowels, consonants, length, overall diversity, changes in spelling, first and last letters

- Analyze external sources of trends: biblical names, celebrities, demographic changes

Using the tools we've looked at so far, most of these kinds of analyses are very straightforward, so I will walk you through many of them. I encourage you to download and explore the data yourself. If you find an interesting pattern in the data, I would love to hear about it.

Loading and Preparing the Data

As of this writing, the US Social Security Administration makes available data files, one per year, containing the total number of births for each sex / name combination. The raw archive of these files can be obtained here:

```
http://www.ssa.gov/oact/babynames/limits.html
```

In the event that this page has been moved by the time you're reading this, it can most likely be located again by internet search. After downloading the "National data" file `names.zip` and unzipping it, you will have a directory containing a series of files like `yob1880.txt`:

```
In [9]: !head -n 10 names/yob1880.txt
Mary,F,7065
Anna,F,2604
Emma,F,2003
Elizabeth,F,1939
Minnie,F,1746
Margaret,F,1578
Ida,F,1472
Alice,F,1414
Bertha,F,1320
Sarah,F,1288
```

As this is in nicely comma-separated form, it can be loaded into a DataFrame with `pandas.read_csv`:

```
In [10]: from pandas import *

In [11]: names1880 = read_csv('names/yob1880.txt', names=['name', 'sex', 'births'])

In [12]: names1880
Out[12]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2000 entries, 0 to 1999
Data columns:
name      2000  non-null values
sex       2000  non-null values
births    2000  non-null values
dtypes: int64(1), object(2)
```

These files only contain names with at least 5 occurrences in each year, so for simplicity's sake we can use the sum of the births column by sex as the total number of births in that year:

```
In [13]: names1880.groupby('sex').births.sum()
Out[13]:
sex
F       90993
M      110493
Name: births
```

Since the data set is split into files by year, one of the first things to do is to assemble all of the data into a single DataFrame and further to add a `year` field. This is easy to do using `pandas.concat`:

```
# 2010 is the last available year right now
years = range(1880, 2011)

pieces = []
columns = ['name', 'sex', 'births']

for year in years:
    path = 'names/yob%d.txt' % year
    frame = read_csv(path, names=columns)

    frame['year'] = year
    pieces.append(frame)

# Concatenate everything into a single DataFrame
names = concat(pieces, ignore_index=True)
```

There are a couple things to note here. First, remember that `concat` glues the DataFrame objects together row-wise by default. Secondly, you have to pass `ignore_index=True` because we're not interested in preserving the original row numbers returned from `read_csv`. So we have now a very large DataFrame containing all of the names data:

Now the `names` DataFrame looks like:

```
In [15]: names
Out[15]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1690784 entries, 0 to 1690783
Data columns:
name      1690784  non-null values
sex       1690784  non-null values
births    1690784  non-null values
year      1690784  non-null values
dtypes: int64(2), object(2)
```

With this data in hand, we can already start aggregating the data at the year and sex level using `groupby` or `pivot_table`, see Figure 14-1:

```
In [16]: total_births = names.pivot_table('births', rows='year',
   ....:                                  cols='sex', aggfunc=sum)
```

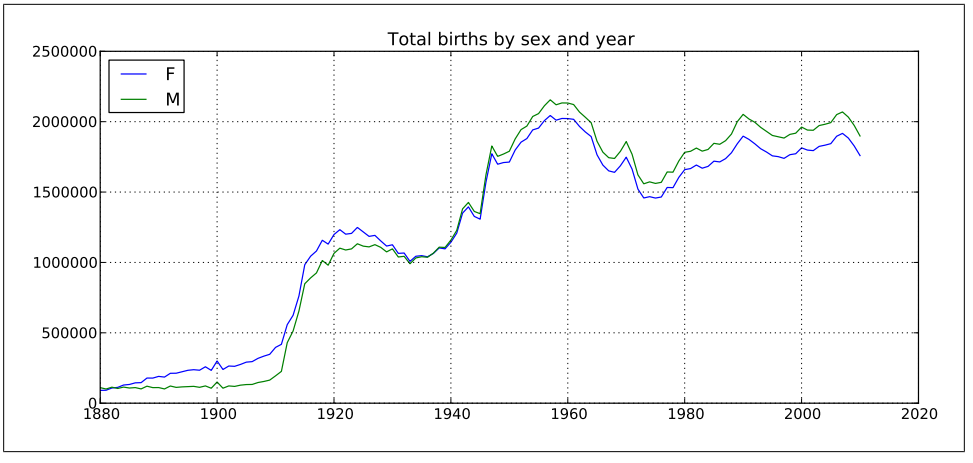


Figure 14-1. Total births by sex and year

```
In [17]: total_births.tail()
Out[17]:
sex         F        M
year
2006  1896468  2050234
2007  1916888  2069242
2008  1883645  2032310
2009  1827643  1973359
2010  1759010  1898382

In [18]: total_births.plot(title='Total births by sex and year')
```

Next, let's insert a column `prop` with the fraction of babies given each name relative to the total number of births. A `prop` value of `0.02` would indicate that 2 out of every 100 babies was given a particular name. Thus, we group the data by year and sex, then add the new column to each group:

```
def add_prop(group):
    # Integer division truncates
    births = group.births.astype(float)

    group['prop'] = births / births.sum()
    return group
names = names.groupby(['year', 'sex']).apply(add_prop)
```

Remember that because `births` is integer type, we have to cast either the numerator or denominator to floating point to compute a fraction (unless you are using Python 3!).

The resulting complete data set now has the following columns:

```
In [20]: names
Out[20]:
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1690784 entries, 0 to 1690783
Data columns:
name      1690784  non-null values
sex       1690784  non-null values
births    1690784  non-null values
year      1690784  non-null values
prop      1690784  non-null values
dtypes: float64(1), int64(2), object(2)
```

Whenever doing a group operation like this, it's sometimes valuable to do a sanity check, like verifying that the `prop` column sums to 1 within all the groups. Since this is floating point data, use `np.allclose` to check that the group sums are sufficiently close to (but perhaps not exactly equal to) 1:

```
In [21]: np.allclose(names.groupby(['year', 'sex']).prop.sum(), 1)
Out[21]: True
```

Now that this is done, I'm going to extract a subset of the data to facilitate further analysis: the top 1000 names for each sex/year combination. This is yet another group operation:

```
def get_top1000(group):
    return group.sort_index(by='births', ascending=False)[:1000]
grouped = names.groupby(['year', 'sex'])
top1000 = grouped.apply(get_top1000)
```

If you prefer a bit more do-it-yourself approach, you could also do:

```
pieces = []
for year, group in names.groupby(['year', 'sex']):
    pieces.append(group.sort_index(by='births', ascending=False)[:1000])
top1000 = concat(pieces, ignore_index=True)
```

The resulting data set is now quite a bit smaller:

```
In [24]: top1000
Out[24]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 261877 entries, 0 to 261876
Data columns:
name      261877  non-null values
sex       261877  non-null values
births    261877  non-null values
year      261877  non-null values
prop      261877  non-null values
dtypes: float64(1), int64(2), object(2)
```

We'll use this Top 1000 data set in most of the following investigations into the data.

Analyzing naming trends

With the full data set and Top 1000 data set in hand, we can start analyzing various naming trends of interest. Splitting the Top 1000 names into the boy and girl portions is easy to do first:

```
In [25]: boys = top1000[top1000.sex == 'M']

In [26]: girls = top1000[top1000.sex == 'F']
```

Simple time series, like the number of Johns or Marys for each year can be plotted but require a bit of munging to be a bit more useful. Let's form a pivot table of the total number of births by year and name:

```
In [27]: total_births = top1000.pivot_table('births', rows='year', cols='name',
   ....:                                     aggfunc=sum)
```

Now, this can be plotted for a handful of names using DataFrame's `plot` method:

```
In [28]: total_births
Out[28]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 131 entries, 1880 to 2010
Columns: 6865 entries, Aaden to Zuri
dtypes: float64(6865)

In [29]: subset = total_births[['John', 'Harry', 'Mary', 'Marilyn']]

In [30]: subset.plot(subplots=True, figsize=(12, 10), grid=False,
   ....:             title="Number of births per year")
```

See Figure 14-2 for the result. On looking at this, you might conclude that these names have grown out of favor with the American population. But the story is actually more complicated than that, as will be explored in the next section.

Measuring the increase in naming diversity

One explanation for the decrease in plots above is that fewer parents are choosing common names for their children. This hypothesis can be explored and confirmed in the data. One measure is the proportion of births represented by the top 1000 most popular names, which I aggregate and plot by year and sex:

```
In [31]: table = top1000.pivot_table('prop', rows='year',
   ....:                              cols='sex', aggfunc=sum)

In [32]: table.plot(title='Sum of table1000.prop by year and sex',
   ....:            yticks=np.linspace(0, 1.2, 13), xticks=range(1880, 2020, 10))
```

See Figure 14-3 for this plot. So you can see that, indeed, there appears to be increasing name diversity (decreasing total proportion in the top 1000). Another interesting metric is the number of distinct names, taken in order of popularity from highest to lowest,

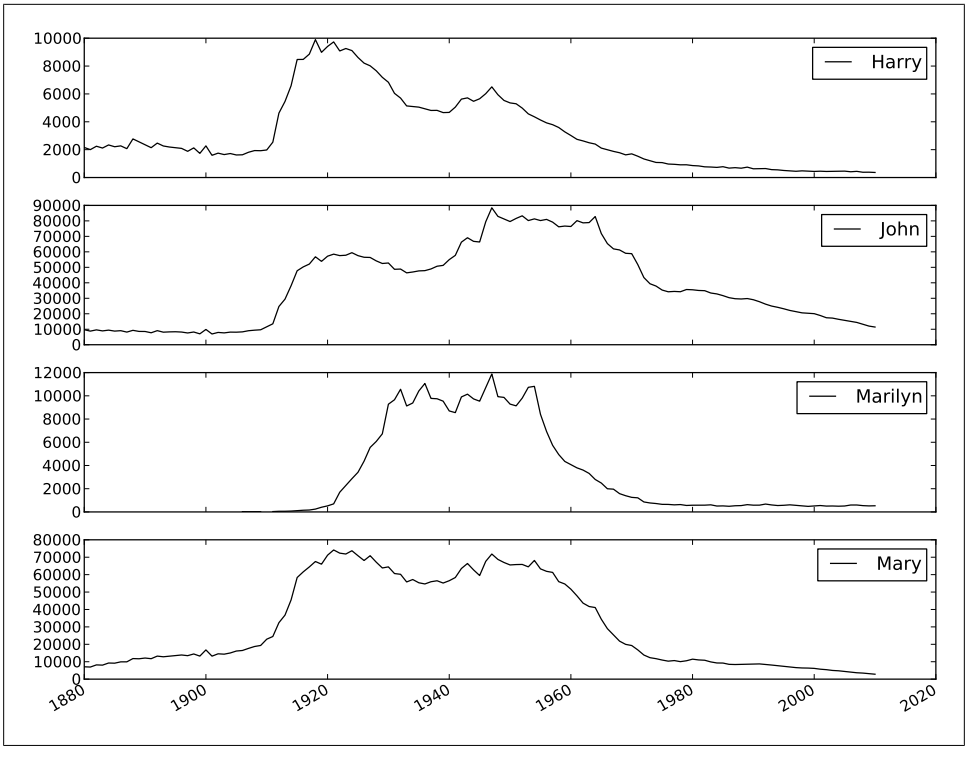


Figure 14-2. A few boy and girl names over time

in the top 50% of births. This number is a bit more tricky to compute. Let's consider just the boy names from 2010:

```
In [33]: df = boys[boys.year == 2010]

In [34]: df
Out[34]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000 entries, 260877 to 261876
Data columns:
name      1000  non-null values
sex       1000  non-null values
births    1000  non-null values
year      1000  non-null values
prop      1000  non-null values
dtypes: float64(1), int64(2), object(2)
```

After sorting `prop` in descending order, we want to know how many of the most popular names it takes to reach 50%. You could write a `for` loop to do this, but a vectorized NumPy way is a bit more clever. Taking the cumulative sum, `cumsum`, of `prop` then calling the method `searchsorted` returns the position in the cumulative sum at which `0.5` would need to be inserted to keep it in sorted order:

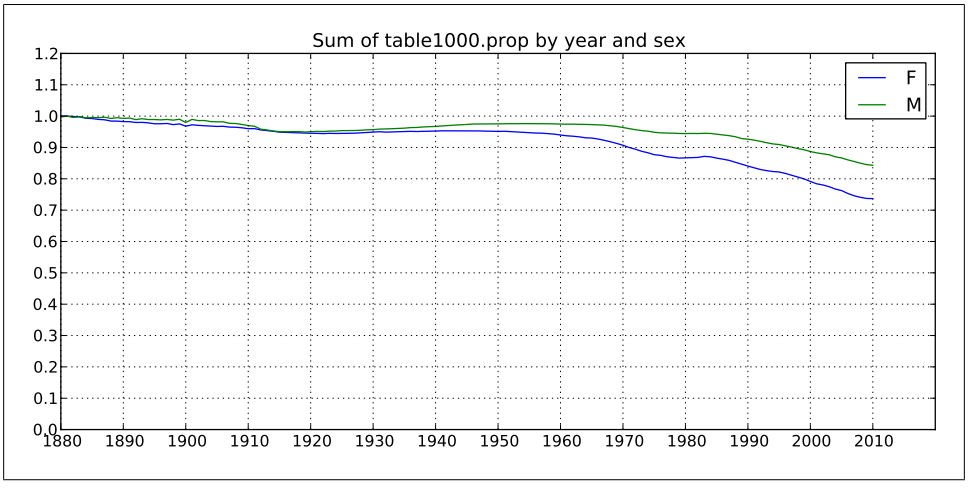


Figure 14-3. Proportion of births represented in top 1000 names by sex

```
In [35]: prop_cumsum = df.sort_index(by='prop', ascending=False).prop.cumsum()

In [36]: prop_cumsum[:10]
Out[36]:
260877    0.011523
260878    0.020934
260879    0.029959
260880    0.038930
260881    0.047817
260882    0.056579
260883    0.065155
260884    0.073414
260885    0.081528
260886    0.089621

In [37]: prop_cumsum.searchsorted(0.5)
Out[37]: 116
```

Since arrays are zero-indexed, adding 1 to this result gives you a result of 117. By contrast, in 1900 this number was much smaller:

```
In [38]: df = boys[boys.year == 1900]

In [39]: in1900 = df.sort_index(by='prop', ascending=False).prop.cumsum()

In [40]: in1900.searchsorted(0.5) + 1
Out[40]: 25
```

It should now be fairly straightforward to apply this operation to each year/sex combination; `groupby` those fields and `apply` a function returning the count for each group:

```
def get_quantile_count(group, q=0.5):
    group = group.sort_index(by='prop', ascending=False)
    return group.prop.cumsum().searchsorted(q) + 1
```

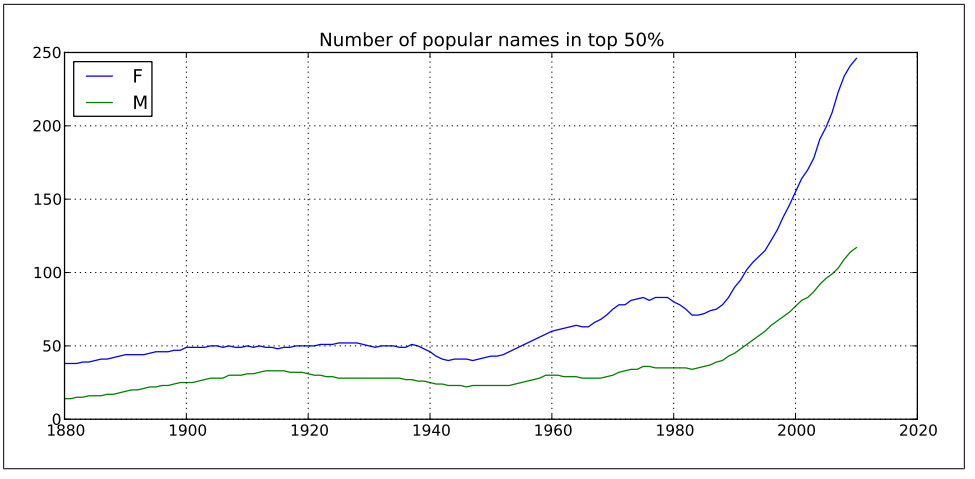


Figure 14-4. Plot of diversity metric by year

```
diversity = top1000.groupby(['year', 'sex']).apply(get_quantile_count)
diversity = diversity.unstack('sex')
```

This resulting DataFrame `diversity` now has two time series, one for each sex, indexed by year. This can be inspected in IPython and plotted as before (see Figure 14-4):

```
In [42]: diversity.head()
Out[42]:
sex    F   M
year
1880  38  14
1881  38  14
1882  38  15
1883  39  15
1884  39  16

In [43]: diversity.plot(title="Number of popular names in top 50%")
```

As you can see, girl names have always been more diverse than boy names, and they have only become more so over time. Further analysis of what exactly is driving the diversity, like the increase of alternate spellings, is left to the reader.

The "Last letter" Revolution

In 2007, a baby name researcher Laura Wattenberg pointed out on her blog that the distribution of boy names by final letter has changed significantly over the last 100 years. To see this, we first aggregate all of the births in the full data set by year, sex, and final letter:

```
# extract last letter from name column
In [44]: get_last_letter = lambda x: x[-1]
```

```
In [45]: last_letters = names.name.map(get_last_letter)

In [46]: last_letters.name = 'last_letter'

In [47]: total_births = names.pivot_table('births', rows='year',
   ....:                                  cols=['sex', last_letters], aggfunc=sum)

In [49]: total_births.head()
Out[49]:
sex               F                       M
year           1910    1960    2010   1910    1960    2010
last_letter
a            108376  691247  670605    977    5204   28438
b               NaN     694     450    411    3912   38859
c                 5      49     946    482   15476   23125
d              6750    3729    2607  22111  262112   44398
e            133569  435013  313833  28655  178823  129012

In [50]: total_births.sum()
Out[50]:
sex  year
F    1910     396416
     1960    2022062
     2010    1759010
M    1910     194198
     1960    2132588
     2010    1898382

In [51]: letter_prop = total_births / total_births.sum().astype(float)

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
letter_prop['M'].plot(kind='bar', rot=0, ax=axes[0], title='Male')
letter_prop['F'].plot(kind='bar', rot=0, ax=axes[1], title='Female')
```

Other curious and fun name trends

The fall of Lesley/Leslie as a boy name

```
In [57]: all_names = top1000.name.unique()

In [58]: mask = np.array(['lesl' in x.lower() for x in all_names])

In [59]: lesley_like = all_names[mask]

In [60]: lesley_like
Out[60]: array([Leslie, Lesley, Leslee, Lesli, Lesly], dtype=object)

In [61]: name_totals = names.groupby('name').births.sum()

In [62]: name_totals[lesley_like].order()
Out[62]:
Leslee      4863
Lesli       5473
```

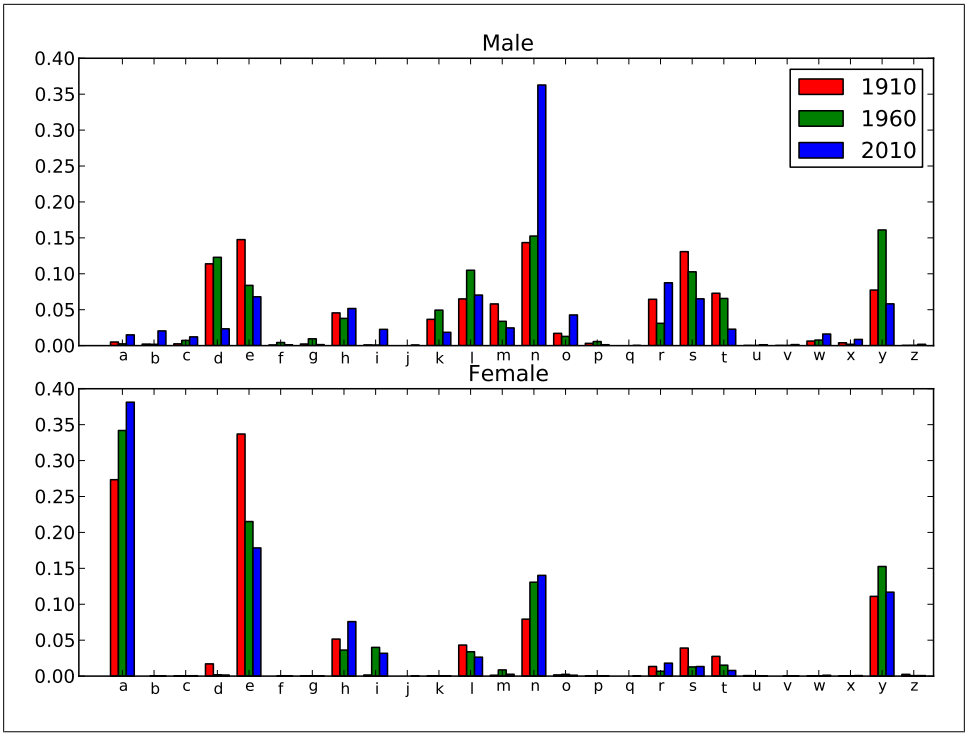


Figure 14-5. Proportion of boy and girl names ending in each letter

```
Lesly      12407
Lesley     37945
Leslie    371686
Name: births
```

Wesley, Westley, and the Princess Bride effect?

Being named Wesley myself (my middle name, actually), it was hard to grow up without developing an extra fondness for *The Princess Bride (1987)*, whose protagonist is named Westley (yes, that *t* is deliberate). I often have to remind people that our names sound the same but are spelled differently.

```
In [63]: weses = names[names.name.isin(['Westley'])]

In [64]: by_year = weses.groupby(['year', 'name']).births.sum()

In [65]: by_year = by_year.unstack('name')

In [66]: by_year.plot(title='Incidence of Westley')
Out[66]: <matplotlib.axes.AxesSubplot at 0x10704910>

In [67]: plt.ylabel('Number of births')
Out[67]: <matplotlib.text.Text at 0x10713910>
```

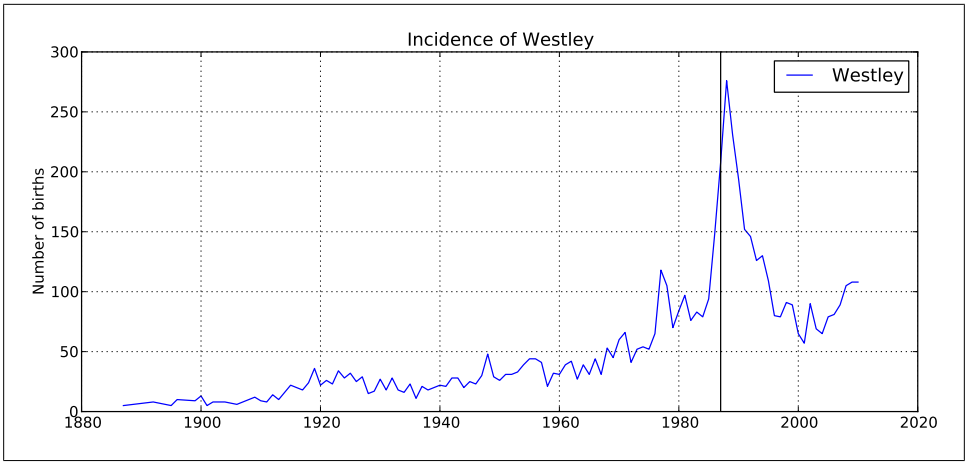


Figure 14-6. Westley over time

```
In [68]: plt.axvline(1987, color='k')
```

Conclusions and takeaways

CHAPTER 15
Advanced NumPy

ndarray object internals

The NumPy ndarray provides a means to interpret a block of homogeneous data (either contiguous or strided, more on this later) as a multidimensional array object. As you've seen, the data type, or dtype, determines how the data is interpreted as being floating point, integer, boolean, or any of the other types we've been looking at.

Part of what makes ndarray powerful is that every array object is a *strided* view on a block of data. You might wonder, for example, how the array view `arr[::2, ::-1]` does not copy any data. Simply put, the ndarray is more than just a chunk of memory and a dtype; it also has striding information which enables the array to move through memory with varying step sizes. More precisely, the ndarray internally consists of the following:

- A *pointer to data*, that is a chunk of CPU memory
- The *data type* or dtype
- A tuple indicating the array's *shape*. For example, a 10 by 5 array would have shape `(10, 5)`.

  ```
  In [8]: np.ones((10, 5)).shape
  Out[8]: (10, 5)
  ```

- A tuple of *strides*, integers indicating the number of bytes to "step" in order to advance one element along a dimension. For example, a typical (C order, more on this later) 3 x 4 x 5 array of `float64` (8-byte) values has strides `(160, 40, 8)`

  ```
  In [9]: np.ones((3, 4, 5), dtype=np.float64).strides
  Out[9]: (160, 40, 8)
  ```

  While it is rare that a typical NumPy user would be interested in the array strides, they are the critical ingredient in constructing copyless array views. Strides can

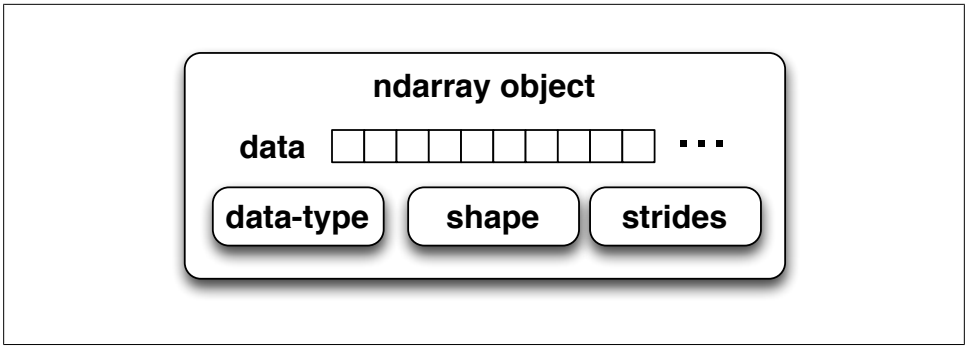


Figure 15-1. The NumPy ndarray object

even be negative which enables an array to move *backward* through memory, which would be the case in a slice like `obj[::-1]` or `obj[:, ::-1]`.

See Figure 15-1 for a simple mockup the ndarray innards.

NumPy dtype hierarchy

You may occasionally have code which needs to check whether an array contains integers, floating point numbers, strings, or Python objects. Because there are many types of floating point numbers (`float16` through `float128`), checking that the dtype is among a list of types would be very verbose. Fortunately, the dtypes have superclasses such as `np.integer` and `np.floating` which can be used in conjunction with the `np.issubdtype` function:

```
In [10]: ints = np.ones(10, dtype=np.uint16)

In [11]: floats = np.ones(10, dtype=np.float32)

In [12]: np.issubdtype(ints.dtype, np.integer)
Out[12]: True

In [13]: np.issubdtype(floats.dtype, np.floating)
Out[13]: True
```

You can see all of the parent classes of a specific dtype by callings the type's `mro` method:

```
In [14]: np.float64.mro()
Out[14]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
 numpy.generic,
 float,
 object]
```

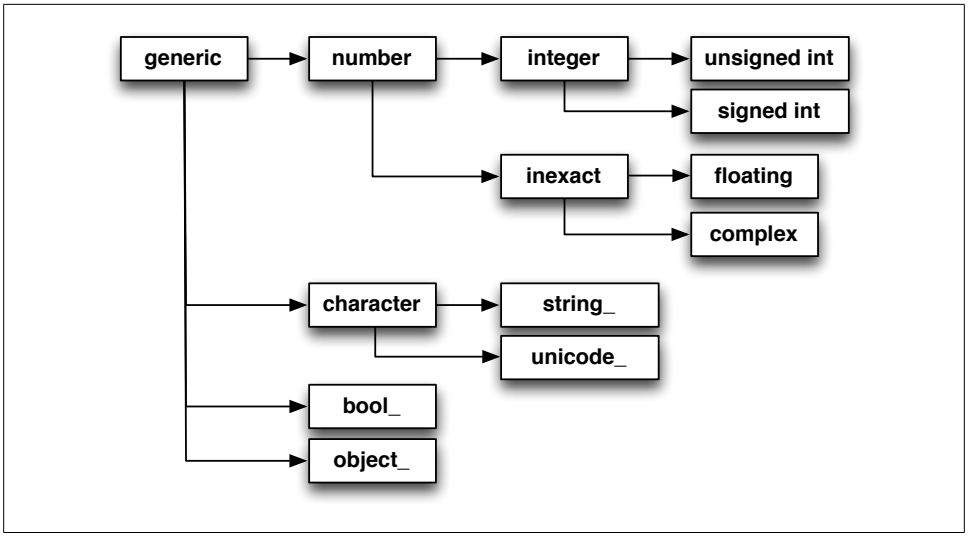


Figure 15-2. The NumPy dtype class hierarchy

Most NumPy users will never have to know about this, but it occasionally comes in handy. See Figure 15-2 for a graph of the dtype hierarchy and parent-subclass relationships.

Advanced array manipulation

There are many ways to work with arrays beyond fancy indexing, slicing, and boolean subsetting. While much of the heavy lifting for data analysis applications is handled by higher level functions in pandas, you may at some point need to write a data algorithm that is not found in one of the existing libraries.

Reshaping arrays

Given what we know about NumPy arrays, it should come as little surprise that you can convert an array from one shape to another without copying any data. To do this, pass a tuple indicating the new shape to the `reshape` array instance method. For example, suppose we had a one-dimensional array of values that we wished to rearrange into a matrix:

```
In [15]: arr = np.arange(8)

In [16]: arr
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [17]: arr.reshape((4, 2))
Out[17]:
array([[0, 1],
```

```
       [2, 3],
       [4, 5],
       [6, 7]])
```

A multidimensional array can also be reshaped:

```
In [18]: arr.reshape((4, 2)).reshape((2, 4))
Out[18]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

One of the passed shape dimensions can be -1, in which case the value used for that dimension will be inferred from the data:

```
In [19]: arr = np.arange(15)

In [20]: arr.reshape((5, -1))
Out[20]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

Since an array's `shape` attribute is a tuple, it can be passed to `reshape`, too:

```
In [21]: other_arr = np.ones((3, 5))

In [22]: other_arr.shape
Out[22]: (3, 5)

In [23]: arr.reshape(other_arr.shape)
Out[23]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

The opposite operation of `reshape` from 1D to higher dimension typically known as *flattening* or *raveling*:

```
In [24]: arr = np.arange(15).reshape((5, 3))

In [25]: arr
Out[25]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])

In [26]: arr.ravel()
Out[26]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

`ravel` does not produce a copy the underlying data if it does not have to (more on this below). The `flatten` method behaves like `ravel` except it always returns a copy of the data:

```
In [27]: arr.flatten()
Out[27]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

The data can be reshaped or raveled in different orders. This is a slightly nuanced topic for new NumPy users and is therefore the next subtopic.

C vs. Fortran order

Contrary to some other scientific computing environments like R and MATLAB, NumPy gives you much more control and flexibility over the layout of your data in memory. By default, NumPy arrays are created in *row major* order. Spatially this means that if you have a 2D array of data, the items in each row of the array are stored in adjacent memory locations. The alternative to row major ordering is *column major* order, which means that (you guessed it) values within each column of data are stored in adjacent memory locations.

For historical reasons, row and column major order are also know as C and Fortran order respectively. In FORTRAN 77, the language of our forebears, matrices were all column major.

Functions like `reshape` and `ravel`, accept an `order` argument indicating the order to use the data in the array. This can be `'C'` or `'F'` in most cases (there are also less commonly-used options `'A'` and `'K'`; see the NumPy documentation). These are illustrated in Figure 15-3.

```
In [28]: arr = np.arange(12).reshape((3, 4))

In [29]: arr
Out[29]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [30]: arr.ravel()
Out[30]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

In [31]: arr.ravel('F')
Out[31]: array([ 0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7, 11])
```

Reshaping arrays with more than 2 dimensions can be a bit mind-bending. The key difference between C and Fortran order is the order in which the dimensions are walked:

- *C / row major order:* traverse higher dimensions *first* (e.g. axis 1 before advancing on axis 0).
- *Fortran / column major order:* traverse higher dimensions *last* (e.g. axis 0 before advancing on axis 1).

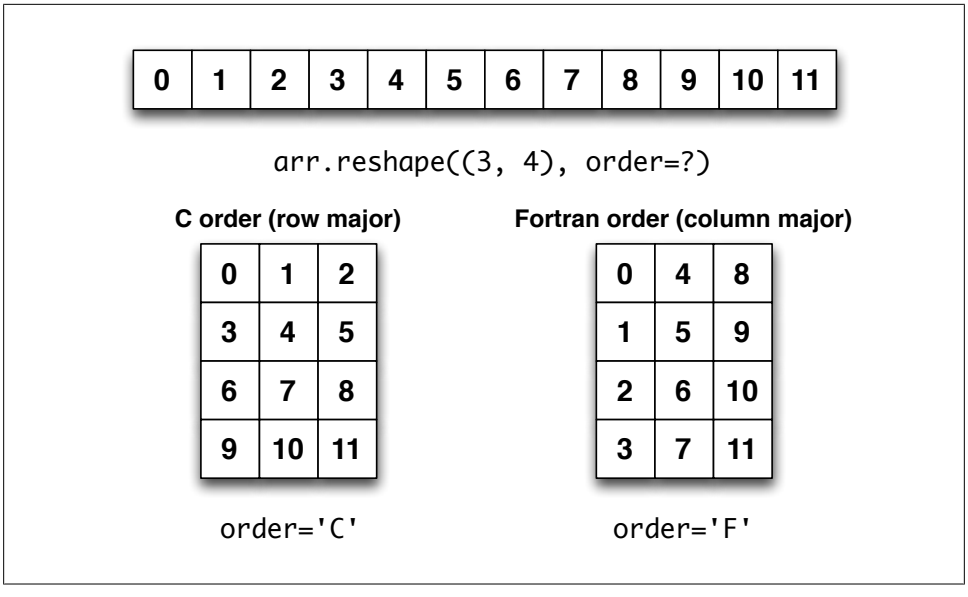


Figure 15-3. Reshaping in C (row major) or Fortran (column major) order

Concatenating and splitting arrays

`numpy.concatenate` takes a sequence (tuple, list, etc.) of arrays and joins them together in order along the input axis.

```
In [32]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])

In [33]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])

In [34]: np.concatenate([arr1, arr2], axis=0)
Out[34]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [35]: np.concatenate([arr1, arr2], axis=1)
Out[35]:
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

There are some convenience functions, like `vstack` and `hstack`, for common kinds of concatenation. The above operations could have been expressed as:

```
In [36]: np.vstack((arr1, arr2))
Out[36]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
In [37]: np.hstack((arr1, arr2))
Out[37]:
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

`split`, on the other hand, slices apart an array into multiple arrays along an axis:

```
In [38]: from numpy.random import randn

In [39]: arr = randn(5, 2)

In [40]: arr
Out[40]:
array([[ 0.8599, -0.5968],
       [ 2.3415, -1.5565],
       [ 1.2778, -0.242 ],
       [ 0.9903,  0.1635],
       [-0.4822,  1.5995]])

In [41]: first, second, third = np.split(arr, [1, 3])

In [42]: first
Out[42]: array([[ 0.8599, -0.5968]])

In [43]: second
Out[43]:
array([[ 2.3415, -1.5565],
       [ 1.2778, -0.242 ]])

In [44]: third
Out[44]:
array([[ 0.9903,  0.1635],
       [-0.4822,  1.5995]])
```

See Table 15-1 for a list of all relevant concatenation and splititng functions, some of which are provided only as a convenience of the very general purpose `concatenate`.

Table 15-1. Array concatenation functions

Function	Description
`concatenate`	Most general function, concatenates collection of arrays along one axis
`vstack, row_stack`	Stack arrays row-wise (along axis 0)
`hstack`	Stack arrays column-wise (along axis 1)
`column_stack`	Like hstack, but converts 1D arrays to 2D column vectors first
`dstack`	Stack arrays "depth"-wise (along axis 2)
`split`	Split array at passed locations along a particular axis
`hsplit / vsplit / dsplit`	Convenience functions for splitting on axis 0, 1, and 2, respectively.

Stacking helpers: r_ and c_

There are two special objects in the NumPy namespace, `r_` and `c_`, that make stacking arrays more concise:

```
In [45]: arr = np.arange(6)

In [46]: np.c_[arr, arr]
Out[46]:
array([[0, 0],
       [1, 1],
       [2, 2],
       [3, 3],
       [4, 4],
       [5, 5]])

In [47]: arr1 = arr.reshape((3, 2))

In [48]: arr2 = randn(3, 2)

In [49]: np.r_[arr1, arr2]
Out[49]:
array([[ 0.    ,  1.    ],
       [ 2.    ,  3.    ],
       [ 4.    ,  5.    ],
       [-0.3563, -0.6718],
       [ 0.2908,  0.1887],
       [ 1.8302, -0.219 ]])
```

These additionally can translate slices to arrays:

```
In [50]: np.c_[1:6, -10:-5]
Out[50]:
array([[  1, -10],
       [  2,  -9],
       [  3,  -8],
       [  4,  -7],
       [  5,  -6]])
```

See the docstring for more on what you can do with `c_` and `r_`.

Repeating elements: tile and repeat

The need to replicate or repeat arrays is less common with NumPy than it is with other popular array programming languages like MATLAB. The main reason for this is *broadcasting*, which is the subject of the next section.

The two main tools for repeating or replicating arrays to produce larger arrays are the `repeat` and `tile` functions. `repeat` replicates each element in an array some number of times, producing a larger array:

```
In [51]: arr = np.arange(3)

In [52]: arr.repeat(3)
Out[52]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

By default, if you pass an integer, each element will be repeated that number of times. If you pass an array of integers, each element can be repeated a different number of times:

```
In [53]: arr.repeat([2, 3, 4])
Out[53]: array([0, 0, 1, 1, 1, 2, 2, 2, 2])
```

Multidimensional arrays can have their elements repeated along a particular axis.

```
In [54]: arr = randn(2, 2)

In [55]: arr.repeat(2, axis=0)
Out[55]:
array([[ 0.3991, -0.0092],
       [ 0.3991, -0.0092],
       [ 0.7164,  0.2036],
       [ 0.7164,  0.2036]])
```

Note that if no axis is passed, the array will be flattened first, so likely not what you want. Similarly you can pass an array of integers when repeating a multidimensional array to repeat which slice a different number of times:

```
In [56]: arr.repeat([2, 3], axis=0)
Out[56]:
array([[ 0.3991, -0.0092],
       [ 0.3991, -0.0092],
       [ 0.7164,  0.2036],
       [ 0.7164,  0.2036],
       [ 0.7164,  0.2036]])

In [57]: arr.repeat([2, 3], axis=1)
Out[57]:
array([[ 0.3991,  0.3991, -0.0092, -0.0092, -0.0092],
       [ 0.7164,  0.7164,  0.2036,  0.2036,  0.2036]])
```

`tile`, on the other hand, is a shortcut for stacking copies of an array along an axis. You can visually think about it as like "laying town tiles":

```
In [58]: arr
Out[58]:
array([[ 0.3991, -0.0092],
       [ 0.7164,  0.2036]])

In [59]: np.tile(arr, 2)
Out[59]:
array([[ 0.3991, -0.0092,  0.3991, -0.0092],
       [ 0.7164,  0.2036,  0.7164,  0.2036]])
```

The second argument to `tile` can be a tuple indicating the layout of the "tiling":

```
In [60]: arr
Out[60]:
```

```
array([[ 0.3991, -0.0092],
       [ 0.7164,  0.2036]])

In [61]: np.tile(arr, (3, 2))
Out[61]:
array([[ 0.3991, -0.0092,  0.3991, -0.0092],
       [ 0.7164,  0.2036,  0.7164,  0.2036],
       [ 0.3991, -0.0092,  0.3991, -0.0092],
       [ 0.7164,  0.2036,  0.7164,  0.2036],
       [ 0.3991, -0.0092,  0.3991, -0.0092],
       [ 0.7164,  0.2036,  0.7164,  0.2036]])
```

Fancy indexing equivalents: take and put

As you may recall from the earlier NumPy chapter, one way to get and set subsets of arrays is by *fancy* indexing using integer arrays:

```
In [62]: arr = np.arange(10)

In [63]: inds = [7, 1, 2, 6]

In [64]: arr[inds]
Out[64]: array([7, 1, 2, 6])
```

There are alternate ndarray methods that are useful in the special case of only making a selection on a single axis:

```
In [65]: arr.take(inds)
Out[65]: array([7, 1, 2, 6])

In [66]: arr.put(inds, 42)

In [67]: arr
Out[67]: array([ 0, 42, 42,  3,  4,  5, 42, 42,  8,  9])

In [68]: arr.put(inds, [40, 41, 42, 43])

In [69]: arr
Out[69]: array([ 0, 41, 42,  3,  4,  5, 43, 40,  8,  9])
```

To use `take` along other axes, you can pass the `axis` keyword:

```
In [70]: inds = [3, 0, 3, 1]

In [71]: arr = randn(2, 5)

In [72]: arr
Out[72]:
array([[ 0.8048,  0.0666, -0.663 , -0.1484,  1.0114],
       [ 0.5527, -0.1589, -0.3763,  1.5702,  1.0111]])

In [73]: arr.take(inds, axis=1)
Out[73]:
array([[-0.1484,  0.8048, -0.1484,  0.0666],
       [ 1.5702,  0.5527,  1.5702, -0.1589]])
```

`put` does not accept an `axis` argument but rather indexes into the flattened (1-dimensional, C order) version of the array. Thus, when you need to set elements using an index array on other axes, you will want to use fancy indexing.

As of this writing, the `take` and `put` functions in general have better performance than their fancy indexing equivalents by a significant margin. I regard this as "bug" and something to be fixed in NumPy, but it's something worth keeping in mind if you're selecting subsets of large arrays using integer arrays:

```
In [74]: arr = randn(1000, 50)

# Random sample of 500 rows
In [75]: inds = np.random.permutation(1000)[:500]

In [76]: %timeit arr[inds]
1000 loops, best of 3: 313 us per loop

In [77]: %timeit arr.take(inds, axis=0)
10000 loops, best of 3: 27.4 us per loop
```

Broadcasting

Broadcasting describes how arithmetic works between arrays of different shapes. It is a very powerful feature, but one that can be easily misunderstood, even by experienced users. The simplest example of broadcasting occurs when combining a scalar value with an array:

```
In [78]: arr = np.arange(5)

In [79]: arr
Out[79]: array([0, 1, 2, 3, 4])

In [80]: arr * 4
Out[80]: array([ 0,  4,  8, 12, 16])
```

Here we say that the scalar value 4 has been *broadcast* to all of the other elements in the multiplication operation.

For example, we can "center" each column of an array by subtracting the column means. In this case, it is very simple:

```
In [81]: arr = np.arange(12.).reshape((4, 3))

In [82]: arr.mean(0)
Out[82]: array([ 4.5,  5.5,  6.5])

In [83]: demeaned = arr - arr.mean(0)

In [84]: demeaned
Out[84]:
array([[-4.5, -4.5, -4.5],
       [-1.5, -1.5, -1.5],
```

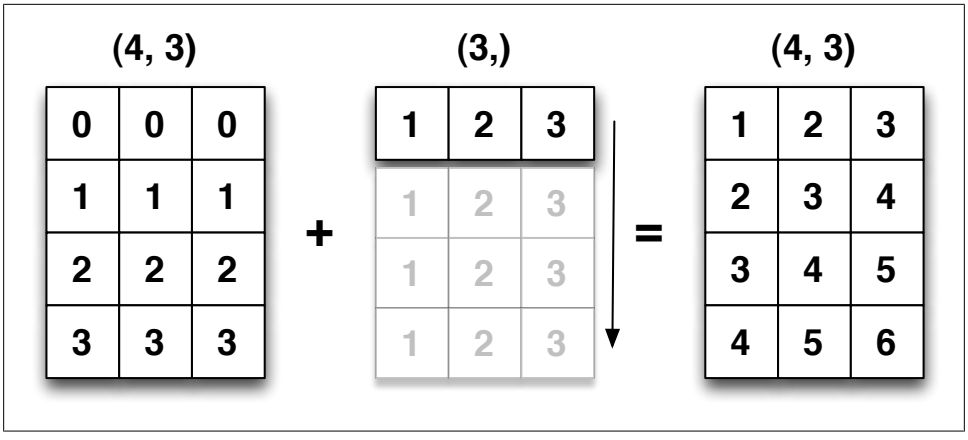


Figure 15-4. Broadcasting over axis 0 with a 1-dimensional array

```
        [ 1.5,  1.5,  1.5],
        [ 4.5,  4.5,  4.5]])

In [85]: demeaned.mean(0)
Out[85]: array([ 0.,  0.,  0.])
```

See Figure 15-4 for an illustration of this operation. Centering the rows as a broadcast operation requires a bit more care. Fortunately, broadcasting potentially lower-dimensional values across any dimension of an array (like subtracting the row means from each column of a 2-dimensional array) is possible as long as you follow the rules. This brings us to:

The Broadcasting Rule

Two arrays are compatible for broadcasting if for each *trailing dimension*, the axis lengths match or if either of the lengths is 1. Broadcasting is then performed over the missing and / or length-1 dimensions.

Even as an experienced NumPy user, I often must stop to draw pictures and think about the broadcasting rule. Consider the last example and suppose we wished instead to subtract the mean value from each row. Since `arr.mean(0)` has length 3, it is compatible for broadcasting across axis 0 because the trailing dimension in `arr` is 3 and therefore matches. According to the rules, to subtract over axis 1 (that is, subtract the row mean from each row), the smaller array must have shape `(4, 1)`:

```
In [86]: arr
Out[86]:
array([[  0.,   1.,   2.],
       [  3.,   4.,   5.],
       [  6.,   7.,   8.],
       [  9.,  10.,  11.]])
```

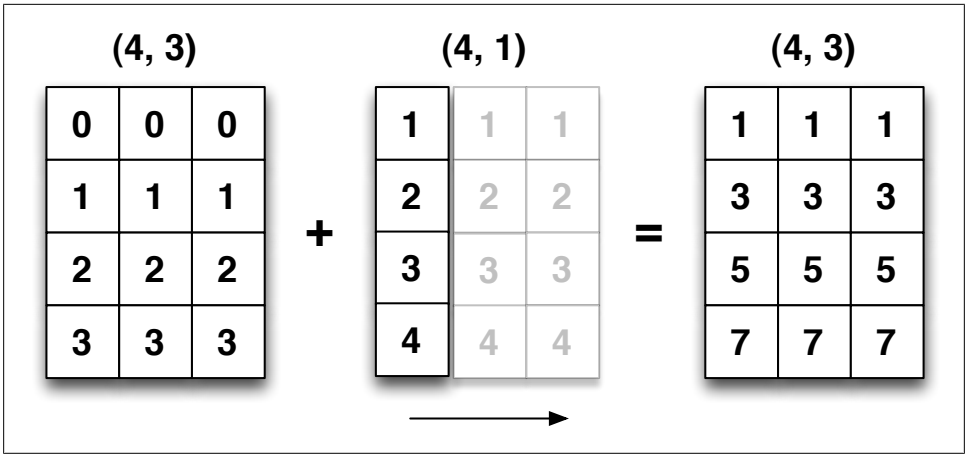


Figure 15-5. Broadcasting over axis 1 of a 2-dimensional array

```
In [87]: arr.mean(1).reshape((-1, 1))
Out[87]:
array([[  1.],
       [  4.],
       [  7.],
       [ 10.]])

In [88]: demeaned = arr - arr.mean(1).reshape((-1, 1))

In [89]: demeaned.mean(1)
Out[89]: array([ 0.,  0.,  0.,  0.])
```

Has your head exploded yet? See Figure 15-5 for an illustration of this operation.

See Figure 15-6 for another illustration, this time subtracting a 2-dimensional array from a 3-dimensional one across axis 0.

Broadcasting over other axes

Broadcasting with higher dimensional arrays can seem even more mind-bending, but it is really a matter of following the rules. If you don't, you'll get an error like this:

```
In [90]: arr - arr.mean(1)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/home/wesm/Dropbox/book/svn/book_scripts/<ipython-input-90-7b87b85a20b2> in <module>()
----> 1 arr - arr.mean(1)
ValueError: operands could not be broadcast together with shapes (4,3) (4)
```

It's quite common to want to perform an arithmetic operation with a lower-dimensional array across axes other than axis 0. According to the broadcasting rule, the "broadcast dimensions" must be 1 in the smaller array. In the example of row demeaning above this meant reshaping the row means to be shape `(4, 1)` instead of `(4,)`:

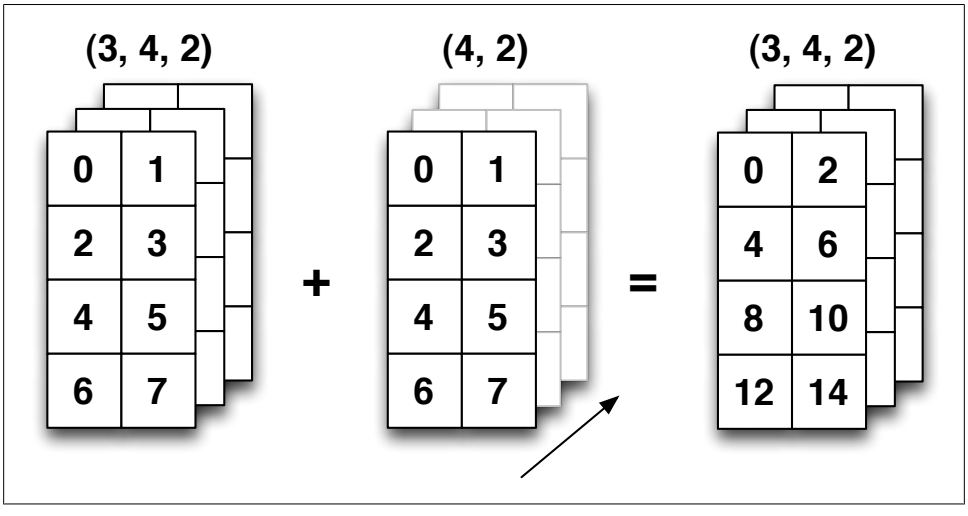


Figure 15-6. Broadcasting over axis 0 of a 3-dimensional array

```
In [91]: arr - arr.mean(1).reshape((4, 1))
Out[91]:
array([[-1.,  0.,  1.],
       [-1.,  0.,  1.],
       [-1.,  0.,  1.],
       [-1.,  0.,  1.]])
```

In the 3-dimensional case, broadcasting over any of the 3 dimensions is only a matter of reshaping the data to be shape-compatible. See Figure 15-7 for a nice visualization of the shapes required to broadcast over each axis of a 3D array.

A very common problem, therefore, is needing to add a new axis with length 1 specifically for broadcasting purposes, especially in generic algorithms. Using `reshape` is one option, but inserting an axis requires constructing a tuple indicating the new shape. This can often be a tedious exercise. Thus, NumPy arrays offer a special syntax for inserting new axes by indexing. We use the special `np.newaxis` attribute along with empty slices to insert the new axis:

```
In [92]: arr = np.zeros((4, 4))

In [93]: arr_3d = arr[:, np.newaxis, :]

In [94]: arr_3d.shape
Out[94]: (4, 1, 4)

In [95]: arr_1d = np.random.normal(size=3)

In [96]: arr_1d[:, np.newaxis]
Out[96]:
array([[-0.2901],
       [ 1.874 ],
       [ 1.1563]])
```

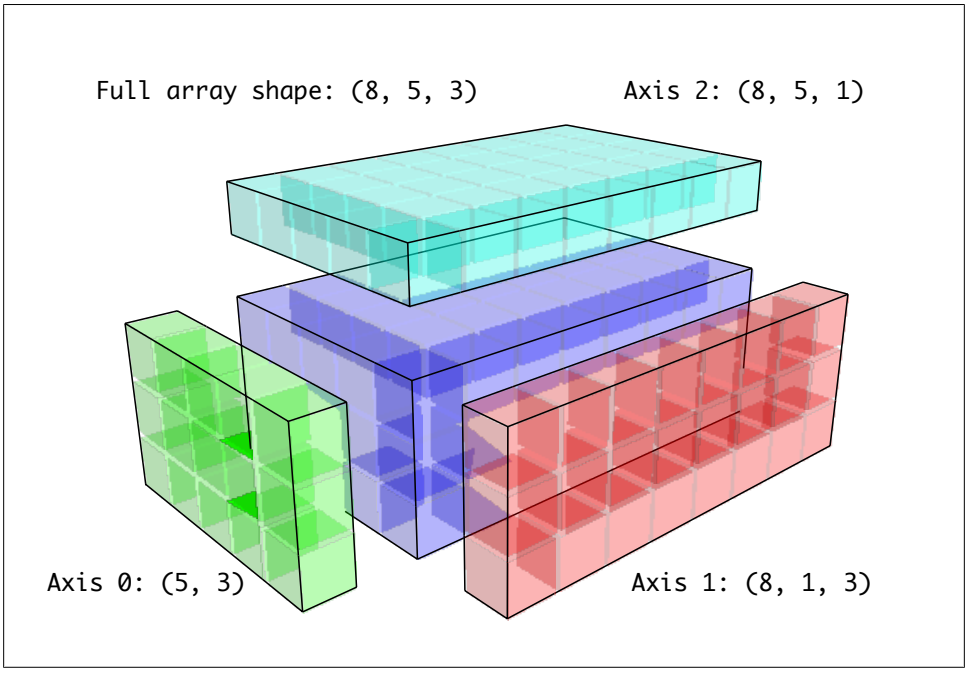


Figure 15-7. Compatible 2D array shapes for broadcasting over a 3D array

```
In [97]: arr_1d[np.newaxis, :]
Out[97]: array([[-0.2901,  1.874 ,  1.1563]])
```

Thus, if we had a 3D array and wanted to demean axis 2, say, we would only need to write:

```
In [98]: arr = randn(3, 4, 5)

In [99]: depth_means = arr.mean(2)

In [100]: depth_means
Out[100]:
array([[ 0.6753, -0.4739, -0.0142, -0.3867],
       [ 0.2635,  0.0103, -1.1938, -0.7631],
       [ 0.486 , -0.6446,  0.239 ,  0.34  ]])

In [101]: demeaned = arr - depth_means[:, :, np.newaxis]

In [102]: demeaned.mean(2)
Out[102]:
array([[ 0.,  0.,  0., -0.],
       [ 0.,  0., -0., -0.],
       [-0., -0.,  0.,  0.]])
```

If you're completely confused by this, don't worry. With practice you will get the hang of it!

Setting array values by broadcasting

The same broadcasting rule governing arithmetic operations also applies to setting values via array indexing. In the simplest case, we can do things like:

```
In [103]: arr = np.zeros((4, 3))

In [104]: arr[:] = 5

In [105]: arr
Out[105]:
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

However, if we had a 1D array of values we wanted to set into the columns of the array, we can do that as long as the shape is compatible:

```
In [106]: col = randn(4)

In [107]: arr[:] = col[:, np.newaxis]

In [108]: arr
Out[108]:
array([[-1.0099, -1.0099, -1.0099],
       [-0.4959, -0.4959, -0.4959],
       [ 0.1422,  0.1422,  0.1422],
       [-1.5318, -1.5318, -1.5318]])

In [109]: arr[:2] = randn(2, 1)

In [110]: arr
Out[110]:
array([[ 0.5008,  0.5008,  0.5008],
       [ 1.3272,  1.3272,  1.3272],
       [ 0.1422,  0.1422,  0.1422],
       [-1.5318, -1.5318, -1.5318]])
```

Advanced ufunc usage

While many NumPy users will only make use of the fast elementwise operations provided by the universal functions, there are a number of additional features that occasionally can help you write more concise code without loops.

Ufunc instance methods

Each of NumPy's binary ufuncs has special methods for performing certain kinds of special vectorized operations. These are summarized in Table 15-2, but I'll give a few concrete examples to illustrate how they work.

`reduce` takes a single array and aggregates its values, optionally along an axis, by performing a sequence of binary operations. For example, an alternate way to sum elements in an array is to use `np.add.reduce`:

```
In [111]: arr = np.arange(10)

In [112]: np.add.reduce(arr)
Out[112]: 45

In [113]: arr.sum()
Out[113]: 45
```

The starting value (0 for `add`) depends on the ufunc. If an axis is passed, the reduction is performed along that axis. This allows you to do answer certain kinds of questions in a concise way. As a less trivial example, we can use `np.logical_and` to check whether the values in each row of an array are sorted:

```
In [115]: arr = randn(5, 5)

In [116]: arr[::2].sort(1) # sort a few rows

In [117]: arr[:, :-1] < arr[:, 1:]
Out[117]:
array([[ True,  True,  True,  True],
       [False,  True, False, False],
       [ True,  True,  True,  True],
       [ True, False,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)

In [118]: np.logical_and.reduce(arr[:, :-1] < arr[:, 1:], axis=1)
Out[118]: array([ True, False,  True, False,  True], dtype=bool)
```

Of course, `logical_and.reduce` is equivalent to the `all` method.

`accumulate` is related to `reduce` like `cumsum` is related to `sum`. It produces an array of the same size with the intermediate "accumulated" values:

```
In [119]: arr = np.arange(15).reshape((3, 5))

In [120]: np.add.accumulate(arr, axis=1)
Out[120]:
array([[ 0,  1,  3,  6, 10],
       [ 5, 11, 18, 26, 35],
       [10, 21, 33, 46, 60]])
```

`outer` performs a pairwise cross-product between two arrays:

```
In [121]: arr = np.arange(3).repeat([1, 2, 2])
```

```
In [122]: arr
Out[122]: array([0, 1, 1, 2, 2])

In [123]: np.multiply.outer(arr, np.arange(5))
Out[123]:
array([[0, 0, 0, 0, 0],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8],
       [0, 2, 4, 6, 8]])
```

The output of `outer` will have dimension that is the sum of the dimensions of the inputs:

```
In [124]: result = np.subtract.outer(randn(3, 4), randn(5))

In [125]: result.shape
Out[125]: (3, 4, 5)
```

The last method, `reduceat`, performs a "local reduce", in essence an array groupby operation in which slices of the array are aggregated together. While it's less flexible than the GroupBy capabilities in pandas, it can be very fast and powerful in the right circumstances. It accepts a sequence of "bin edges" which indicate how to split and aggregate the values:

```
In [126]: arr = np.arange(10)

In [127]: np.add.reduceat(arr, [0, 5, 8])
Out[127]: array([10, 18, 17])
```

The results are the reductions (here, sums) performed over `arr[0:5]`, `arr[5:8]`, and `arr[8:]`. Like the other methods, you can pass an axis argument:

```
In [128]: arr = np.multiply.outer(np.arange(5), np.arange(5))

In [129]: arr
Out[129]:
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  3,  6,  9, 12],
       [ 0,  4,  8, 12, 16]])

In [130]: np.add.reduceat(arr, [0, 2, 4], axis=1)
Out[130]:
array([[ 0,  0,  0],
       [ 1,  5,  4],
       [ 2, 10,  8],
       [ 3, 15, 12],
       [ 4, 20, 16]])
```

Table 15-2. Ufunc methods

Method	Description
`reduce(x)`	Aggregate values by successive applications of the operation
`accumulate(x)`	Aggregate values, preserving all partial aggregates

Method	Description
`reduceat(x, bins)`	"Local" reduce or "group by". Reduce contiguous slices of data to produce aggregated array.
`outer(x, y)`	Apply operation to all pairs of elements in `x` and `y`. Result array has shape `x.shape + y.shape`

Custom ufuncs

There are a couple facilities for creating your own functions with ufunc-like semantics. `numpy.frompyfunc` accepts a Python function along with a specification for the number of inputs and outputs. For example, a simple function that adds elementwise would be specified as:

```
In [131]: def add_elements(x, y):
   .....:     return x + y

In [132]: add_them = np.frompyfunc(add_elements, 2, 1)

In [133]: add_them(np.arange(8), np.arange(8))
Out[133]: array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)
```

Functions created using `frompyfunc` always return arrays of Python objects which isn't very convenient. Fortunately, there is an alternate, but slightly less featureful function `numpy.vectorize` that is a bit more intelligent about type inference:

```
In [134]: add_them = np.vectorize(add_elements, otypes=[np.float64])

In [135]: add_them(np.arange(8), np.arange(8))
Out[135]: array([  0.,   2.,   4.,   6.,   8.,  10.,  12.,  14.])
```

These functions provide a way to create ufunc-like functions, but they are very slow because they require a Python function call to compute each element, which is a lot slower than NumPy's C-based ufunc loops:

```
In [136]: arr = randn(10000)

In [137]: %timeit add_them(arr, arr)
100 loops, best of 3: 1.97 ms per loop

In [138]: %timeit np.add(arr, arr)
100000 loops, best of 3: 11.5 us per loop
```

There are a number of projects under way in the scientific Python community to make it easier to define new ufuncs whose performance is closer to that of the built-in ones.

Structured and record arrays

You may have noticed up until now that ndarray is a *homogeneous* data container; that is, it represents a block of memory in which each element takes up the same number of bytes, determined by the dtype. On the surface, this would appear to not allow you

to represent heterogeneous or tabular-like data. A *structured* array is an ndarray in which each element can be thought of as representing a *struct* in C (hence the "structured" name) or a row in a SQL table with multiple named fields:

```
In [139]: dtype = [('x', np.float64), ('y', np.int32)]

In [140]: sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=dtype)

In [141]: sarr
Out[141]:
array([(1.5, 6), (3.141592653589793, -2)],
      dtype=[('x', '<f8'), ('y', '<i4')])
```

One way (of many, see the full online NumPy documentation on this) to specify a structured dtype is as a list of tuples with `(field_name, field_data_type)`. Now, the elements of the array are tuple-like object whose elements can be accessed like a dictionary:

```
In [142]: sarr[0]
Out[142]: (1.5, 6)

In [143]: sarr[0]['y']
Out[143]: 6
```

The field names are stored in the `dtype.names` attribute. On accessing a field on the structured array, a strided view on the data is returned thus copying nothing:

```
In [144]: sarr['x']
Out[144]: array([ 1.5   ,  3.1416])
```

Nested dtypes and multidimensional fields

When specifying a structured dtype, you can additionally pass a shape (as an int or tuple):

```
In [145]: dtype = [('x', np.int64, 3), ('y', np.int32)]

In [146]: arr = np.zeros(4, dtype=dtype)

In [147]: arr
Out[147]:
array([([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0)],
      dtype=[('x', '<i8', (3,)), ('y', '<i4')])
```

In this case, the `x` field now refers to an array of length 3 for each record:

```
In [148]: arr[0]['x']
Out[148]: array([0, 0, 0])
```

Conveniently, accessing `arr['x']` then returns a two-dimensional array instead of a one-dimensional array as in prior examples:

```
In [149]: arr['x']
Out[149]:
array([[0, 0, 0],
```

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

This enables you to express more complicated, nested structures as a single block of memory in an array. Though, since dtypes can be arbitrarily complex, why not nested dtypes? Here is a simple example:

```
In [150]: dtype = [('x', [('a', 'f8'), ('b', 'f4')]), ('y', np.int32)]

In [151]: data = np.array([((1, 2), 5), ((3, 4), 6)], dtype=dtype)

In [152]: data['x']
Out[152]:
array([(1.0, 2.0), (3.0, 4.0)],
      dtype=[('a', '<f8'), ('b', '<f4')])

In [153]: data['y']
Out[153]: array([5, 6], dtype=int32)

In [154]: data['x']['a']
Out[154]: array([ 1.,  3.])
```

As you can see, variable-shape fields and nested records is a very rich feature that be the right tool in certain circumstances. DataFrame, by contrast, does not support this feature directly, though it is similar to hierarchical indexing.

Why use structured arrays?

Compared with, say, pandas's DataFrame, NumPy structured arrays are a comparatively low-level tool. They provide a means to interpreting a block of memory as a tabular structure with arbitrarily complex nested columns. Since each element in the array is represented in memory as a fixed number of bytes, structured arrays provide a very fast and efficient way of writing data to and from disk (including memory maps, more on this later), transporting it over the network, and other such use.

As another common use for structured arrays, writing data files as fixed length record byte streams is a common way to serialize data in C and C++ code, which is commonly found in legacy systems in the industry. As long as the format of the file is known (the size of each record and the order, byte size, and data type of each element), the data can be read into memory using `np.fromfile`. Specialized uses like this are beyond the scope of this book, but it's worth knowing that such things are possible.

Structured array manipulations: numpy.lib.recfunctions

While there is not as much functionality available for structured arrays as for DataFrames, the NumPy module `numpy.lib.recfunctions` has some helpful tools for adding and dropping fields or doing basic join-like operations. The thing to remember with these tools is that it is typically necessary to create a new array to make any modifica-

tions to the dtype (like adding or dropping a column). These functions are left to the interested reader to explore as I do not use them anywhere in this book.

More about sorting

Like Python's built-in list, the ndarray `sort` instance method is an *in-place* sort, meaning that the array contents are rearranged without producing a new array:

```
In [155]: arr = randn(6)

In [156]: arr.sort()

In [157]: arr
Out[157]: array([-1.082 ,  0.3759,  0.8014,  1.1397,  1.2888,  1.8413])
```

When sorting arrays in-place, remember that if the array is a view on a different ndarray, the original array will be modified:

```
In [158]: arr = randn(3, 5)

In [159]: arr
Out[159]:
array([[-0.3318, -1.4711,  0.8705, -0.0847, -1.1329],
       [-1.0111, -0.3436,  2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424,  0.8548,  1.038 , -0.329 ]])

In [160]: arr[:, 0].sort()  # Sort first column values in-place

In [161]: arr
Out[161]:
array([[-1.0111, -1.4711,  0.8705, -0.0847, -1.1329],
       [-0.3318, -0.3436,  2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424,  0.8548,  1.038 , -0.329 ]])
```

On the other hand, `numpy.sort` creates a new, sorted copy of an array. Otherwise it accepts the same arguments (such as `kind`, more on this below) as `ndarray.sort`:

```
In [162]: arr = randn(5)

In [163]: arr
Out[163]: array([-1.1181, -0.2415, -2.0051,  0.7379, -1.0614])

In [164]: np.sort(arr)
Out[164]: array([-2.0051, -1.1181, -1.0614, -0.2415,  0.7379])

In [165]: arr
Out[165]: array([-1.1181, -0.2415, -2.0051,  0.7379, -1.0614])
```

All of these sort methods take an axis argument for sorting the sections of data along the passed axis independently:

```
In [166]: arr = randn(3, 5)

In [167]: arr
```

```
Out[167]:
array([[ 0.5955, -0.2682,  1.3389, -0.1872,  0.9111],
       [-0.3215,  1.0054, -0.5168,  1.1925, -0.1989],
       [ 0.3969, -1.7638,  0.6071, -0.2222, -0.2171]])

In [168]: arr.sort(axis=1)

In [169]: arr
Out[169]:
array([[-0.2682, -0.1872,  0.5955,  0.9111,  1.3389],
       [-0.5168, -0.3215, -0.1989,  1.0054,  1.1925],
       [-1.7638, -0.2222, -0.2171,  0.3969,  0.6071]])
```

You may notice that none of the sort methods have an option to sort in descending order. This is not actually a big deal because array slicing produces views, thus not producing a copy or requiring any computational work. Many Python users are familiar with the "trick" that for a list `values`, `values[::-1]` returns a list in reverse order. The same is true for ndarrays:

```
In [170]: arr[:, ::-1]
Out[170]:
array([[ 1.3389,  0.9111,  0.5955, -0.1872, -0.2682],
       [ 1.1925,  1.0054, -0.1989, -0.3215, -0.5168],
       [ 0.6071,  0.3969, -0.2171, -0.2222, -1.7638]])
```

Indirect sorts: argsort and lexsort

In data analysis it's very common to need to reorder data sets by one or more keys. For example, a table of data about some students might need to be sorted by last name then by first name. This is an example of an *indirect* sort, and if you've read the pandas-related chapters you have already seen many higher-level examples. Given a key or keys (an array or values or multiple arrays of values), you wish to obtain an array of integer *indices* (I refer to them colloquially as *indexers*) that tells you how to reorder the data to be in sorted order. The two main methods for this are `argsort` and `numpy.lexsort`. As a trivial example:

```
In [171]: values = np.array([5, 0, 1, 3, 2])

In [172]: indexer = values.argsort()

In [173]: indexer
Out[173]: array([1, 2, 4, 3, 0])

In [174]: values[indexer]
Out[174]: array([0, 1, 2, 3, 5])
```

As a less trivial example, this code reorders a 2D array by its first row:

```
In [175]: arr = randn(3, 5)

In [176]: arr[0] = values

In [177]: arr
```

```
Out[177]:
array([[ 5.    ,  0.    ,  1.    ,  3.    ,  2.    ],
       [-0.3636, -0.1378,  2.1777, -0.4728,  0.8356],
       [-0.2089,  0.2316,  0.728 , -1.3918,  1.9956]])

In [178]: arr[:, arr[0].argsort()]
Out[178]:
array([[ 0.    ,  1.    ,  2.    ,  3.    ,  5.    ],
       [-0.1378,  2.1777,  0.8356, -0.4728, -0.3636],
       [ 0.2316,  0.728 ,  1.9956, -1.3918, -0.2089]])
```

`lexsort` is similar to argsort, but it performs an indirect *lexicographical* sort on multiple key arrays. Suppose we wanted to sort some data identified by first and last names:

```
In [179]: first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill', 'Barbara'])

In [180]: last_name = np.array(['Jones', 'Arnold', 'Arnold', 'Jones', 'Walters'])

In [181]: sorter = np.lexsort((first_name, last_name))

In [182]: zip(last_name[sorter], first_name[sorter])
Out[182]:
[('Arnold', 'Jane'),
 ('Arnold', 'Steve'),
 ('Jones', 'Bill'),
 ('Jones', 'Bob'),
 ('Walters', 'Barbara')]
```

`lexsort` can be a bit confusing the first time you use it because the order in which the keys are used to order the data starts with the *last* array passed. As you can see, `last_name` was used before `first_name`.

pandas methods like Series's and DataFrame's `sort_index` methods and the Series `order` method are implemented with variants of these functions (which also must take into account missing values)

Alternate sort algorithms

A *stable* sorting algorithm preserves the relative position of equal elements. This can be especially important in indirect sorts where the relative ordering is meaningful:

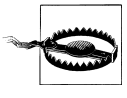
```
In [183]: values = np.array(['2:first', '2:second', '1:first', '1:second', '1:third'])

In [184]: key = np.array([2, 2, 1, 1, 1])

In [185]: indexer = key.argsort(kind='mergesort')

In [186]: indexer
Out[186]: array([2, 3, 4, 0, 1])

In [187]: values.take(indexer)
Out[187]:
```

```
array(['1:first', '1:second', '1:third', '2:first', '2:second'],
      dtype='|S8')
```

The only stable sort available is *mergesort* which has guaranteed O(n log n) performance (for complexity buffs), but its performance is on average worse than the default quicksort method. See Table 15-3 for a summary of available methods and their relative performance (and performance guarantees). This is not something that most users will ever have to think about but useful to know that it's there.

Table 15-3. Array sorting methods

Kind	Speed	Stable	Work space	Worst-case
`'quicksort'`	1	No	0	$O(n^2)$
`'mergesort'`	2	Yes	n / 2	O(n log n)
`'heapsort'`	3	No	0	O(n log n)



At the time of this writing, for whatever reason, sort algorithms other than quicksort are not available on arrays of Python objects (`dtype=object`). This means occasionally that sort algorithms requiring stability will require workarounds when dealing with Python objects.

numpy.searchsorted: Finding elements in a sorted array

`searchsorted` is an array method that performs a binary search on a sorted array, returning the location in the array where the value would need to be inserted to maintain sortedness:

```
In [188]: arr = np.array([0, 1, 7, 12, 15])

In [189]: arr.searchsorted(9)
Out[189]: 3
```

As you might expect, you can also pass an array of values to get an array of indices back:

```
In [190]: arr.searchsorted([0, 8, 11, 16])
Out[190]: array([0, 3, 3, 5])
```

You might have noticed that `searchsorted` returned `0` for the `0` element. This is because the default behavior is to return the index at the left side of a group of equal values:

```
In [191]: arr = np.array([0, 0, 0, 1, 1, 1, 1])

In [192]: arr.searchsorted([0, 1])
Out[192]: array([0, 3])

In [193]: arr.searchsorted([0, 1], side='right')
Out[193]: array([3, 7])
```

As another application of `searchsorted`, suppose we had an array of values between 0 and 10000) and a separate array of "bucket edges" that we wanted to use to bin the data:

```
In [194]: data = np.floor(np.random.uniform(0, 10000, size=50))

In [195]: bins = np.array([0, 2500, 5000, 7500, 10000])

In [196]: data
Out[196]:
array([ 8304.,  4181.,  9352.,  4907.,  3250.,  8546.,  2673.,  6152.,
        2774.,  5130.,  9553.,  4997.,  1794.,  9688.,   426.,  1612.,
         651.,  8653.,  1695.,  4764.,  1052.,  4836.,  8020.,  3479.,
        1513.,  5872.,  8992.,  7656.,  4764.,  5383.,  2319.,  4280.,
        4150.,  8601.,  3946.,  9904.,  7286.,  9969.,  6032.,  4574.,
        8480.,  4298.,  2708.,  7358.,  6439.,  7916.,  3899.,  9182.,
         871.,  7973.])
```

To then get a labeling of which interval each data point belongs to (where 1 would mean the bucket `[0, 100)`), we can simply use `searchsorted`:

```
In [197]: labels = bins.searchsorted(data)

In [198]: labels
Out[198]:
array([4, 2, 4, 2, 2, 4, 2, 3, 2, 3, 4, 2, 1, 4, 1, 1, 1, 4, 1, 2, 1, 2, 4,
       2, 1, 3, 4, 4, 2, 3, 1, 2, 2, 4, 2, 4, 3, 4, 3, 2, 4, 2, 2, 3, 3, 4,
       2, 4, 1, 4])
```

This, combined with pandas's `groupby`, can be used to easily bin data:

```
In [199]: Series(data).groupby(labels).mean()
Out[199]:
key_0
1        1325.888889
2        4028.235294
3        6206.500000
4        8799.312500
Name: result
```

Note that NumPy actually has a function `digitize` that computes this bin labeling:

```
In [200]: np.digitize(data, bins)
Out[200]:
array([4, 2, 4, 2, 2, 4, 2, 3, 2, 3, 4, 2, 1, 4, 1, 1, 1, 4, 1, 2, 1, 2, 4,
       2, 1, 3, 4, 4, 2, 3, 1, 2, 2, 4, 2, 4, 3, 4, 3, 2, 4, 2, 2, 3, 3, 4,
       2, 4, 1, 4])
```

Advanced array input and output

In the earlier chapter on NumPy, I introduced you to `np.save` and `np.load` for storing arrays in binary format on disk. There are a number of additional options to consider for more sophisticated use. In particular, memory maps have the additional benefit of enabling you in some cases to work with much larger-than-RAM data sets.

Memory-mapped files

A *memory-mapped* file is a binary file on disk that represents a potentially very large array. NumPy implements a `memmap` object that is ndarray-like, enabling small segments of a large file to be read and written without reading the whole array into memory. Additionally, a memmap has the same methods as an in-memory array and thus can be substituted into many algorithms where an ndarray would be expected.

To create a new memmap, use the function `np.memmap` and pass a dtype, shape, and file mode:

```
In [201]: mmap = np.memmap('mymmap', dtype='float64', mode='w+', shape=(10000, 10000))

In [202]: mmap
Out[202]:
memmap([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

Slicing a memmap returns views on the data on disk:

```
In [203]: section = mmap[:5]
```

If you assign data to these, it will be buffered in memory (like a Python file object), but can be written to disk by calling `flush`:

```
In [204]: section[:] = np.random.randn(5, 10000)

In [205]: mmap.flush()

In [206]: mmap
Out[206]:
memmap([[-0.1614, -0.1768,  0.422 , ..., -0.2195, -0.1256, -0.4012],
       [ 0.4898, -2.2219, -0.7684, ..., -2.3517, -1.0782,  1.3208],
       [-0.6875,  1.6901, -0.7444, ..., -1.4218, -0.0509,  1.2224],
       ...,
       [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ],
       [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ],
       [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ]])

In [207]: del mmap
```

Whenever a memory map falls out of scope and is garbage-collected, any changes will be flushed to disk also. When *opening an existing memory map*, you still have to specify the dtype and shape as the file is just a block of binary data with no metadata on disk:

```
In [208]: mmap = np.memmap('mymmap', dtype='float64', shape=(10000, 10000))

In [209]: mmap
Out[209]:
memmap([[-0.1614, -0.1768,  0.422 , ..., -0.2195, -0.1256, -0.4012],
```

```
[ 0.4898, -2.2219, -0.7684, ..., -2.3517, -1.0782,  1.3208],
[-0.6875,  1.6901, -0.7444, ..., -1.4218, -0.0509,  1.2224],
...,
[ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ],
[ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ],
[ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ]])
```

Since a memory map is just an on-disk ndarray, there are no issues using a structured dtype as described above.

HDF5 and other array storage options

PyTables and h5py are two Python projects providing NumPy-friendly interfaces for storing array data in the efficient and compressible HDF5 format (HDF stands for *hierarchical data format*). You can safely store hundreds of gigabytes or even terabytes of data in HDF5 format. The use of these libraries is unfortunately outside

PyTables provides a rich facility for working with structured arrays with advanced querying features and the ability to add column indexes to accelerate queries. This is very similar to the table indexing capabilities provided by relational databases.

Performance tips

Getting good performance out of code utilizing NumPy is often straightforward as array operations typically replace otherwise comparatively extremely slow pure Python loops. Here is a brief list of some of the things to keep in mind:

- Convert Python loops and conditional logic to array operations and boolean array operations
- Use broadcasting whenever possible
- Avoid copying data using array views (slicing)
- Utilize ufuncs and ufunc methods

If you can't get the performance you require after exhausing the capabilities provided by NumPy alone, writing code in C, Fortran, or especially Cython (see a bit more on this below) may be in order. I personally use Cython (*http://cython.org*) heavily in my own work as an easy way to get C-like performance with minimal development.

The importance of contiguous memory

While the full extent of this topic is a bit outside the scope of this book, in some applications the memory layout of an array can significantly affect the speed of computations. This is based partly on performance differences having to do with the cache hierarchy of the CPU; operations accessing contiguous blocks of memory (for example, summing the rows of a C order array) will generally be the fastest because the memory subsystem will buffer the appropriate blocks of memory into the ultrafast L1 or L2 CPU

cache. Also, certain code paths inside NumPy's C codebase have been optimized for the contiguous case in which generic strided memory access can be avoided.

To say that an array's memory layout is *contiguous* means that the elements are stored in memory in the order that they appear in the array with respect to Fortran (column major) or C (row major) ordering. By default, NumPy arrays are created as *C-contiguous* or just simply contiguous. A column major array, such as the transpose of a C-contiguous array, is thus said to be Fortran-contiguous. These properties can be explicitly checked via the `flags` ndarray attribute:

```
In [212]: arr_c = np.ones((1000, 1000), order='C')

In [213]: arr_f = np.ones((1000, 1000), order='F')

In [214]: arr_c.flags
Out[214]:
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False

In [215]: arr_f.flags
Out[215]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False

In [216]: arr_f.flags.f_contiguous
Out[216]: True
```

In this some example, summing the rows of these arrays should, in theory, be faster for `arr_c` than `arr_f` since the rows are contiguous in memory. Here I check for sure using `%timeit` in IPython:

```
In [217]: %timeit arr_c.sum(1)
1000 loops, best of 3: 1.01 ms per loop

In [218]: %timeit arr_f.sum(1)
100 loops, best of 3: 7.93 ms per loop
```

When looking to squeeze more performance out of NumPy, this is often a place to invest some effort. If you have an array that does not have the desired memory order, you can use `copy` and pass either `'C'` or `'F'`:

```
In [219]: arr_c.copy('F').flags
Out[219]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
```

```
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

When constructing a view on an array, keep in mind that the result is not guaranteed to be contiguous:

```
In [220]: arr_c[:50].flags.contiguous
Out[220]: True

In [221]: arr_c[:, :50].flags
Out[221]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

Other speed options: Cython, f2py, C

In recent years, the Cython project ((*http://cython.org*) has become the tool of choice for many scientific Python programmers for implementing fast code that may need to interact with C or C++ libraries, but without having to write pure C code. You can think of Cython as Python with static types and the ability to interleave functions implemented in C into Python-like code. For example, a simple Cython function to sum the elements of a 1-dimensional array might look like:

```
from numpy cimport ndarray, float64_t

def sum_elements(ndarray[float64_t] arr):
    cdef Py_ssize_t i, n = len(arr)
    cdef float64_t result = 0

    for i in range(n):
        result += arr[i]

    return result
```

Cython takes this code, translates it to C, then compiles the generated C code to create a Python extension. Cython is an attractive option for performance computing because the code is only slightly more time consuming to write than pure Python code and it integrates closely with NumPy. A common workflow is to get an algorithm working in Python, then translate it to Cython by adding type declarations and a handful of other tweaks. For more, see the project documentation.

Some other options for writing high performance code with NumPy include f2py, a wrapper generator for Fortran 77 and 90 code, and writing pure C extensions.

CHAPTER 16

SciPy, statsmodels, and scikit-learn

CHAPTER 17
Parallel and Distributed Computing

```
Multiprocessing module
----------------------

Parallel computing with OpenMP in Cython
----------------------------------------

Distributed computing with IPython
----------------------------------

Writing MapReduce jobs in Python
--------------------------------


Examples
--------
```