O'REILLY®

Third Edition

Python for Data Analysis

Data Wrangling with pandas, NumPy & Jupyter

Early Release

RAW & UNEDITED

Wes McKinney

Python for Data Analysis

THIRD EDITION

Data Wrangling with Pandas, NumPy, and Jupyter

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Wes McKinney

Python for Data Analysis

by Wes McKinney

Printed in the United States of America.

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

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

Editor: Angela Rufino

Production Editor: Daniel Elfanbaum

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

October 2012: First Edition

October 2017: Second Edition

September 2022: Third Edition

Revision History for the Early Release

- 2021-05-26: First Release
- 2021-09-09: Second Release
- 2021-11-11: Third Release

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

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

978-1-098-10396-5

[LSI]

Chapter 1. Preliminaries

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at arufino@oreilly.com.

1.1 What Is This Book About?

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

Sometime after I originally published this book in 2012, people started using the term "data science" as an umbrella description for everything from simple descriptive statistics to more advanced statistical analysis and machine learning. The Python open source ecosystem for doing data analysis (or data science) has also expanded significantly since then. There are now many other books which focus specifically on these more advanced methodologies. My hope is that this book serves as adequate preparation to enable you to move on to a more domain-specific resource.

NOTE

Some might characterize much of the content of the book as "data manipulation" as opposed to "data analysis". We also use the terms "wrangling" or "munging" to refer to data manipulation.

What Kinds of Data?

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

- Tabular or spreadsheet-like data in which each column may be a different type (string, numeric, date, or otherwise). This includes most kinds of data commonly stored in relational databases or tab- or comma-delimited text files.
- Multidimensional arrays (matrices).

- Multiple tables of data interrelated by key columns (what would be primary or foreign keys for a SQL user).
- Evenly or unevenly spaced time series.

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

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

1.2 Why Python for Data Analysis?

For many people, the Python programming language has strong appeal. Since its first appearance in 1991, Python has become one of the most popular interpreted programming languages, along with Perl, Ruby, and others. Python and Ruby have become especially popular since 2005 or so for building websites using their numerous web frameworks, like Rails (Ruby) and Django (Python). Such languages are often called *scripting* languages, as they can be used to quickly write small programs, or *scripts* to automate other tasks. I don't like the term "scripting language," as it carries a connotation that they cannot be used for building serious software. Among interpreted languages, for various historical and cultural reasons, Python has developed a large and active scientific computing and data analysis community. In the last 20 years, Python has gone from a bleeding-edge or "at your own risk" scientific computing language to one of the most important languages for data science, machine learning, and general software development in academia and industry.

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

Python as Glue

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

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

Solving the "Two-Language" Problem

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

Over the last decade some new approaches to solving the “two-language” problem, such as the Julia programming language. Getting the most out of Python in many cases *will* require programming in a low-level language like C or C++ and creating Python bindings to that code. That said, “just-in-time” (JIT) compiler technology provided by libraries like Numba have provided a way to achieve excellent performance in many computational algorithms without having to leave the Python programming environment.

Why Not Python?

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

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

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

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

1.3 Essential Python Libraries

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

NumPy

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

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

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

pandas

pandas provides high-level data structures and functions designed to make working with structured or tabular data fast, easy, and expressive. Since its emergence in 2010, it has helped enable Python to be a powerful and productive data analysis environment. The primary objects in pandas that will be used in this book are the `DataFrame`, a tabular, column-oriented data structure with both row and column labels, and the `Series`, a one-dimensional labeled array object.

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

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

- Data structures with labeled axes supporting automatic or explicit data alignment—this prevents common errors resulting from misaligned data and working with differently indexed data coming from different sources
- Integrated time series functionality
- The same data structures handle both time series data and non–time series data
- Arithmetic operations and reductions that preserve metadata

- Flexible handling of missing data
- Merge and other relational operations found in popular databases (SQL-based, for example)

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

I spent a large part of 2011 and 2012 expanding pandas's capabilities with some of my former AQR colleagues, Adam Klein and Chang She. In 2013, I stopped being as involved in day to day project development and pandas has since become a fully community-owned and community-maintained project with well over 2000 unique contributors around the world.

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

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

matplotlib

matplotlib is the most popular Python library for producing plots and other two-dimensional data visualizations. It was originally created by John D. Hunter and is now maintained by a large team of developers. It is designed for creating plots suitable for publication. While there are other visualization libraries available to Python programmers, matplotlib is still widely used and integrates reasonably well with the rest of the ecosystem. I think it is a safe choice as a default visualization tool..

IPython and Jupyter

The IPython project began in 2001 as Fernando Pérez's side project to make a better interactive Python interpreter. Over the subsequent 20 years it has become one of the most important tools in the modern Python data stack. While it does not provide any computational or data analytical tools by itself, IPython is designed both interactive computing and software development work. It encourages an *execute-explore* workflow instead of the typical *edit-compile-run* workflow of many other programming languages. It also provides integrated access to your operating system's shell and filesystem; this reduces the need to switch between a terminal window and a Python session in many cases.. Since much of data analysis coding involves exploration, trial and error, and iteration, IPython can help you get the job done faster.

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

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

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

I personally use IPython and Jupyter regularly in my Python work, whether running, debugging, and testing code.

In the accompanying book materials, you will find Jupyter notebooks containing all the code examples from each chapter.

SciPy

SciPy is a collection of packages addressing a number of foundational problems in scientific computing. Here are some of the tools it contains in its various modules:

`scipy.integrate`

Numerical integration routines and differential equation solvers

`scipy.linalg`

Linear algebra routines and matrix decompositions extending beyond those provided in `numpy.linalg`

`scipy.optimize`

Function optimizers (minimizers) and root finding algorithms

`scipy.signal`

Signal processing tools

`scipy.sparse`

Sparse matrices and sparse linear system solvers

`scipy.special`

Wrapper around SPECFUN, a Fortran library implementing many common mathematical functions, such as the `gamma` function

`scipy.stats`

Standard continuous and discrete probability distributions (density functions, samplers, continuous distribution functions), various statistical tests, and more descriptive statistics

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

scikit-learn

Since the project's inception in 2010, scikit-learn has become the premier general-purpose machine learning toolkit for Python programmers. As of this writing, more than 2,000 different individuals have contributed code to the project. It includes submodules for such models as:

- Classification: SVM, nearest neighbors, random forest, logistic regression, etc.
- Regression: Lasso, ridge regression, etc.
- Clustering: *k*-means, spectral clustering, etc.
- Dimensionality reduction: PCA, feature selection, matrix factorization, etc.
- Model selection: Grid search, cross-validation, metrics
- Preprocessing: Feature extraction, normalization

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

statsmodels

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

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

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

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

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

Other Packages

Now in 2021 as I write this, there are many other Python libraries which might be discussed in a book about data science. This includes some newer projects like TensorFlow or PyTorch which have become popular for machine learning or artificial intelligence work. Now that there are other books out there which focus more specifically on those projects, I would recommend using this book to build a foundation in general purpose Python data wrangling. Then, you should be well-prepared to move on to a more advanced resource which may assume a certain level of expertise.

1.4 Installation and Setup

Since everyone uses Python for different applications, there is no single solution for setting up Python and obtaining the necessary add-on packages. Many readers will not have a complete Python development environment suitable for following along with this book, so here I will give detailed instructions to get set up on each operating system. I will be using Miniconda, a minimal installation of the conda package manager, along with conda-forge, a community-maintained software distribution based on conda. This book uses Python 3.8 throughout, but if you're reading in the future you are welcome to install a newer version of Python. The first edition of this book used Python 2.7, but the entire 2.x line of Python interpreters reached end of life in 2020.

If for some reason these instructions become out of date by the time you are reading this, you can check out my website for the book which I will endeavor to keep up to date with the latest installation instructions.

Miniconda on Windows

To get started on Windows, download the Miniconda installer for the latest Python version available (currently 3.8). I recommend following the installation instructions for Windows available on the conda website, which may have changed between the time this book was published and when you are reading this. Most people will want the 64-bit version, but if that doesn't run on your Windows machine, you can install the 32-bit version instead.

When prompted whether to install for just yourself or for all users on your system, choose the option that's most appropriate for you. Installing just for yourself will be sufficient to follow along with the book. It will also ask you whether you want to add Miniconda to the system PATH environment variable. If you select this (I usually do), then this Miniconda installation may override other versions of Python you have installed. If you do not, then you will need to use the Window Start menu shortcut that's installed to be able to use this Miniconda. This Start menu entry may be called "Anaconda3 (64-bit)".

I'll assume that you haven't added Miniconda to your system PATH. To verify that things are configured correctly, open the "Anaconda Prompt (Miniconda3)" entry under "Anaconda3 (64-bit)" in the start menu. Then try launching the Python interpreter by typing **`python`**. You should see a message like this:

```
(base) C:\Users\Wes>python
Python 3.8.5 [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on
win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

To exit the Python shell, type the command **`exit()`** and press Enter.

Miniconda on macOS

Download the macOS Miniconda installer, which should be named something like *Miniconda3-latest-MacOSX-x86_64.pkg*. Double-click the *.pkg* file to run the installer. When the installer runs, by default it automatically configures Miniconda in your default shell environment in your *.bash_profile* file. This is located at */Users/$USER/.bash_profile*. I recommend letting it do this; if you do not want to allow the installer to modify your default shell environment, you will need to consult the Miniconda documentation to be able to proceed.

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

```
$ python
Python 3.8.5 (default, Sep  4 2020, 02:22:02)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

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

GNU/Linux

Linux details will vary a bit depending on your Linux distribution type, but here I give details for such distributions as Debian, Ubuntu, CentOS, and Fedora. Setup is similar to macOS with the exception of how Miniconda is installed. The installer is a shell script that must be executed in the terminal. Depending on whether you have a 32-bit or 64-bit system, you will either need to install the x86 (32-bit) or x86_64 (64-bit) installer. You will then have a file named something similar to *Miniconda3-latest-Linux-x86_64.sh*. To install it, execute this script with bash:

```
$ bash Miniconda3-latest-Linux-x86_64.sh
```

> **NOTE**
>
> Some Linux distributions have all the required Python packages (though outdated versions, in some cases) in their package managers and can be installed using a tool like apt. The setup described here uses Miniconda, as it’s both easily reproducible across distributions and simpler to upgrade packages to their latest versions.

You will be presented with a choice of where to put the Miniconda files. I recommend installing the files in the default location in your home directory; for example, */home/$USER/miniconda* (with your username, naturally).

The installer will ask if you wish to modify your shell scripts to automatically activate Miniconda. I recommend doing this (say "yes") as a matter of convenience.

After completing the installation, start a new terminal process and verify that you are picking up the new Miniconda installation:

```
(base) $ which python
/home/wesm/miniconda/bin/python

(base) $ python
Python 3.8.5 (default, Sep  4 2020, 07:30:14)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

NOTE

At the time of this writing, ARM-based architecture Linux platforms (versus x86-based) are not ubiquitous among desktop users but this seems likely to change over the coming years. If you are on an ARM-based platform, I advise you to consult the Miniconda website or do an internet search to find the most up-to-date installation instructions for your platform.

Installing Necessary Packages

Now that we have set up Miniconda on your system, it's time to install the main packages we will be using in this book. The first step is to configure conda-forge as your default package channel by running the following commands in a shell:

```
(base) $ conda config --add channels conda-forge
(base) $ conda config --set channel_priority strict
```

Now, we will create a new conda "environment" using the `conda create` command using Python 3.8:

```
(base) $ conda create -n pydata-book python=3.8
```

After indicating yes ("Y") to the environment creation, activate the environment with `conda activate`:

```
(base) $ conda activate pydata-book
(pydata-book) $
```

Now, we will install the essential packages used throughout the book (along with their dependencies) with `conda install`:

```
(pydata-book) $ conda install -y pandas jupyter matplotlib
```

We will be using some other packages, too, but these can be installed later once they are needed. There are two ways to install packages, with `conda install` and with `pip install`. `conda install` should always be preferred when using Miniconda, but some packages are not available through conda so if `conda install $package_name` fails, try `pip install $package_name`.

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

```
conda update package_name
```

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

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

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

CAUTION

While you can use both conda and pip to install packages, you should avoid updating packages originally installed with conda using pip (and vice versa), as doing so can lead to environment problems. I recommend sticking to conda if you can and falling back on pip only for packages which are unavailable with `conda install`.

Integrated Development Environments (IDEs) and Text Editors

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

When building software, however, some users may prefer to use a more richly featured IDE rather than an editor like Emacs or Vim which provide a more minimal environment out of the box. Here are some that you can explore:

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

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

1.5 Community and Conferences

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

- pydata: A Google Group list for questions related to Python for data analysis and pandas
- pystatsmodels: For statsmodels or pandas-related questions
- Mailing list for scikit-learn (*scikit-learn@python.org*) and machine learning in Python, generally
- numpy-discussion: For NumPy-related questions
- scipy-user: For general SciPy or scientific Python questions

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

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

- PyCon and EuroPython: The two main general Python conferences in North America and Europe, respectively
- SciPy and EuroSciPy: Scientific-computing-oriented conferences in North America and Europe, respectively
- PyData: A worldwide series of regional conferences targeted at data science and data analysis use cases
- International and regional PyCon conferences (see *https://pycon.org* for a complete listing)

1.6 Navigating This Book

If you have never programmed in Python before, you will want to spend some time in Chapters 2 and 3, where I have placed a condensed tutorial on Python language features and the IPython shell and Jupyter notebooks. These things are prerequisite knowledge for the remainder of the book. If you have Python experience already, you may instead choose to skim or skip these chapters.

Next, I give a short introduction to the key features of NumPy, leaving more advanced NumPy use for Appendix A. Then, I introduce pandas and devote the rest of the book to data analysis topics applying pandas, NumPy, and matplotlib (for visualization). I have structured the material in an incremental fashion, though there is occasionally some minor cross-over between chapters, with a few cases where concepts are used that haven't been introduced yet.

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

Interacting with the outside world

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

Preparation

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

Transformation

Applying mathematical and statistical operations to groups of datasets to derive new datasets (e.g., aggregating a large table by group variables)

Modeling and computation

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

Presentation

> Creating interactive or static graphical visualizations or textual summaries

Code Examples

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

```
In [5]: CODE EXAMPLE
Out[5]: OUTPUT
```

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

Data for Examples

Datasets for the examples in each chapter are hosted in a GitHub repository. You can download this data either by using the Git version control system on the command line or by downloading a zip file of the repository from the website. If you run into problems, navigate to my website for up-to-date instructions about obtaining the book materials.

CAUTION

There are readers in some countries who may be unable to address GitHub; please see my website for help if this applies to you.

I have made every effort to ensure that it contains everything necessary to reproduce the examples, but I may have made some mistakes or omissions. If so, please send me an email: *book@wesmckinney.com*. The best way to report errors in the book is on the errata page on the O'Reilly website.

Import Conventions

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

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

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

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

When I wrote the first edition of this book in 2011 and 2012, there were fewer resources available for learning about doing data analysis in Python. This was partially a chicken-and-egg problem; many libraries that we now take for granted, like pandas, scikit-learn, and statsmodels, were comparatively immature back then. Now in 2021, there is now a growing literature on data science, data analysis, and machine learning, supplementing the prior works on general-purpose scientific computing geared toward computational scientists, physicists, and professionals in other research fields. There are also excellent books about learning the Python programming language itself and becoming an effective software engineer.

As this book is intended as an introductory text in working with data in Python, I feel it is valuable to have a self-contained overview of some of the most important features of Python's built-in data structures and libraries from the perspective of data manipulation. So, I will only present roughly enough information in this chapter and Chapter 3 to enable you to follow along with the rest of the book.

Much of this book focuses on table-based analytics and data preparation tools for working with datasets that are small enough to fit on your personal computer. In order to use these tools you must sometimes do some wrangling to arrange messy data into a more nicely tabular (or *structured*) form. Fortunately, Python is an ideal language for doing this. The greater your facility with the Python language and its built-in data types, the easier it will be for you to prepare new datasets for analysis.

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

> **NOTE**
>
> There are introductory Python concepts that this chapter does not cover, like classes and object-oriented programming, which you may find useful in your foray into data analysis in Python.
>
> To deepen your Python language knowledge, I recommend that you supplement this chapter with the official Python tutorial and potentially one of the many excellent books on general-purpose Python programming. Some recommendations to get you started include:
>
> - *Python Cookbook*, Third Edition, by David Beazley and Brian K. Jones (O'Reilly)
> - *Fluent Python* by Luciano Ramalho (O'Reilly)
> - *Effective Python* by Brett Slatkin (Pearson)

2.1 The Python Interpreter

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

```
$ python
Python 3.8.6 | packaged by conda-forge | (default, Jan 25 2021,
23:22:12)
[Clang 11.0.1 ] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> a = 5
>>> print(a)
5
```

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

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

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

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

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

While some Python programmers execute all of their Python code in this way, those doing data analysis or scientific computing make use of IPython, an enhanced Python interpreter, or Jupyter notebooks, web-based code notebooks originally created within the IPython project. I give an introduction to using IPython and Jupyter in this chapter and have included a deeper look at IPython functionality in Appendix A. When you use the

`%run` command, IPython executes the code in the specified file in the same process, enabling you to explore the results interactively when it’s done:

```
$ ipython
Python 3.8.6 | packaged by conda-forge | (default, Jan 25 2021,
23:22:12)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.20.0 -- An enhanced Interactive Python. Type '?' for
help.

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

In [2]:
```

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

2.2 IPython Basics

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

Running the IPython Shell

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

```
$ ipython
Python 3.8.6 | packaged by conda-forge | (default, Jan 25 2021,
23:22:12)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.20.0 -- An enhanced Interactive Python. Type '?' for
help.

In [1]: a = 5

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

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

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

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

In [7]: data
Out[7]:
{0: -0.20470765948471295,
 1: 0.47894333805754824,
 2: -0.5194387150567381,
 3: -0.55573030434749,
 4: 1.9657805725027142,
 5: 1.3934058329729904,
 6: 0.09290787674371767}
```

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

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

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

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

Running the Jupyter Notebook

One of the major components of the Jupyter project is the *notebook*, a type of interactive document for code, text (including Markdown), data visualizations, and other output. The Jupyter notebook interacts with *kernels*, which are implementations of the Jupyter interactive computing protocol specific to different programming languages. Python's Jupyter kernel uses the IPython system for its underlying behavior.

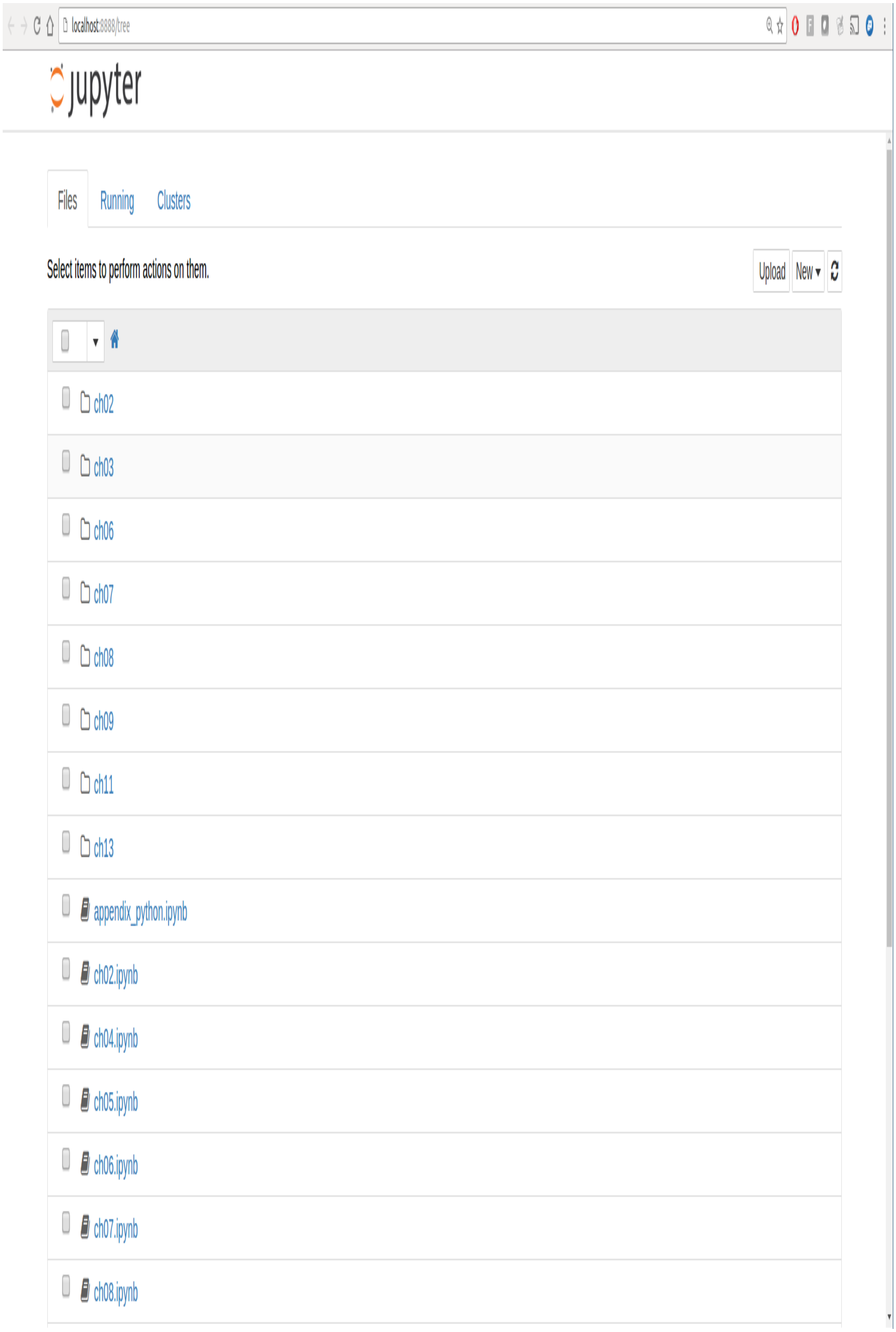
To start up Jupyter, run the command `jupyter notebook` in a terminal:

```
$ jupyter notebook
[I 15:20:52.739 NotebookApp] Serving notebooks from local
directory:
/home/wesm/code/pydata-book
[I 15:20:52.739 NotebookApp] 0 active kernels
[I 15:20:52.739 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/
[I 15:20:52.740 NotebookApp] Use Control-C to stop this server
and shut down
all kernels (twice to skip confirmation).
Created new window in existing browser session.
```

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

NOTE

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



Select items to perform actions on them.

ch02

ch03

ch06

ch07

ch08

ch09

ch11

ch13

appendix_python.ipynb

ch02.ipynb

ch04.ipynb

ch05.ipynb

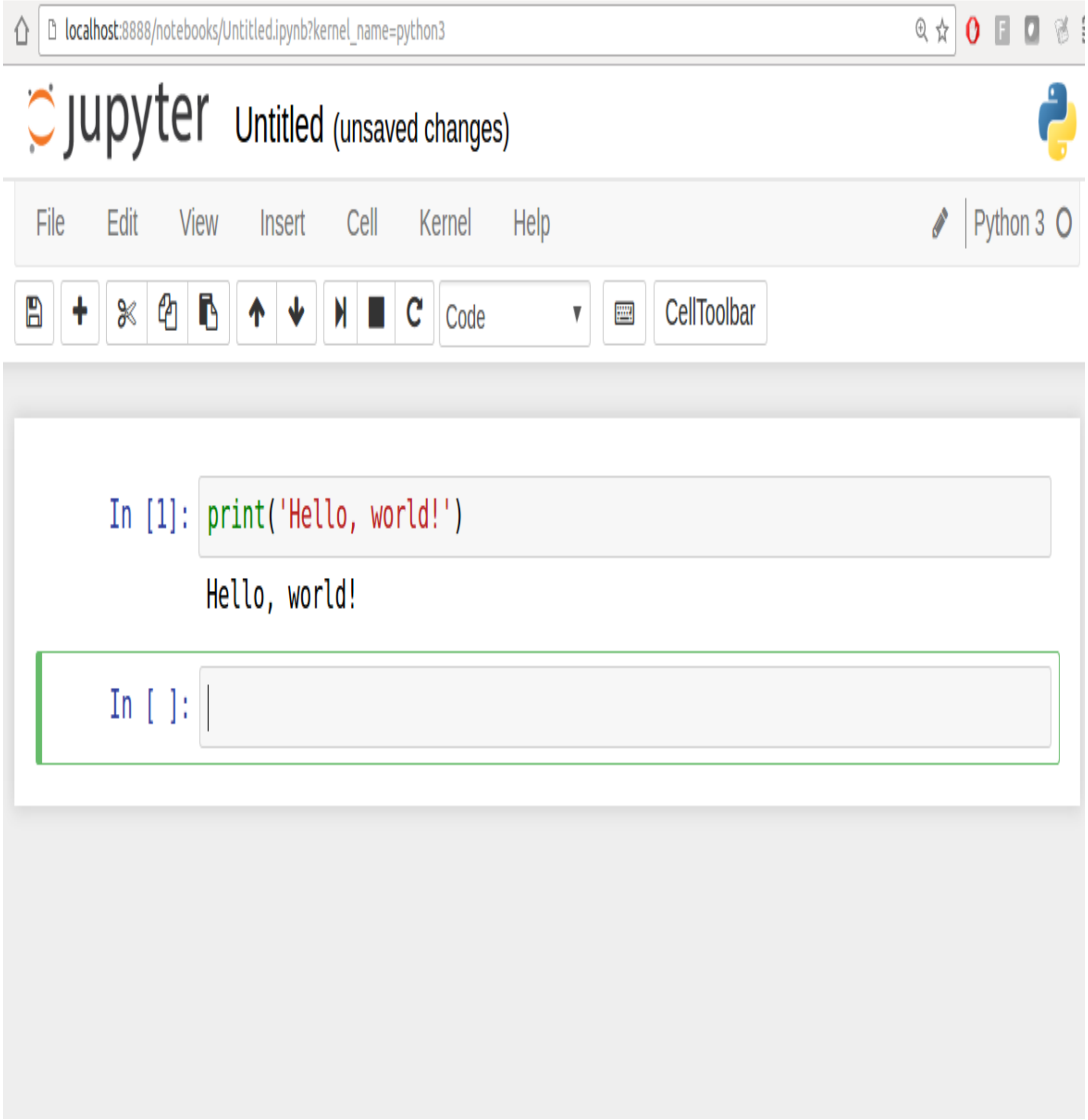
ch06.ipynb

ch07.ipynb

ch08.ipynb

Figure 2-1. Jupyter notebook landing page

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

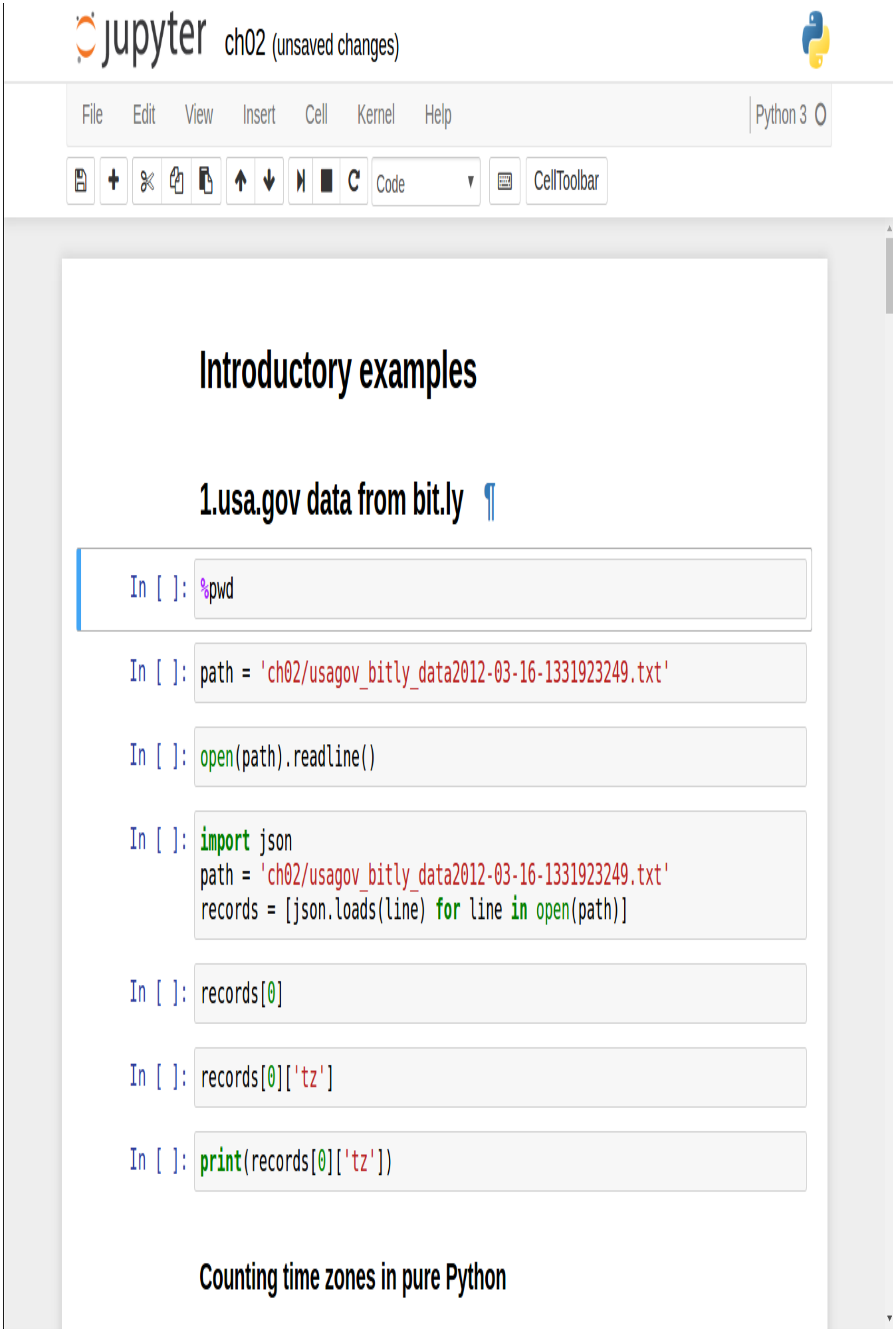

localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3
jupyter
Untitled (unsaved changes)
File
Edit
View
Insert
Cell
Kernel
Help
Python 3
Code
CellToolbar

```
In [1]: print('Hello, world!')

Hello, world!

In [ ]:
```

Figure 2-2. Jupyter new notebook view

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

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



Introductory examples

1.usa.gov data from bit.ly ¶

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

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

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

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

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

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

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

Counting time zones in pure Python

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

Tab Completion

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

```
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`. Also, you can also complete methods and attributes on any object after typing a period:

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

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

The same is true for modules:

```
In [1]: import datetime

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

NOTE

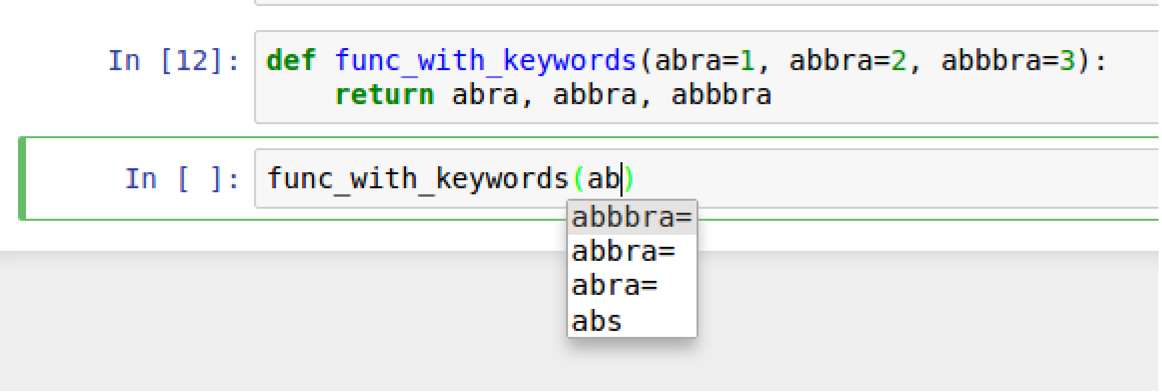
Note that IPython by default hides methods and attributes starting with underscores, such as magic methods and internal "private" methods and attributes, in order to avoid cluttering the display (and confusing novice users!). These, too, can be tab-completed, but you must first type an underscore to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

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

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

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

Combined with the `%run` command (see "The %run Command"), this functionality can save you many keystrokes.

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



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

In [ ]: func_with_keywords(ab)
```

Figure 2-4. Autocomplete function keywords in Jupyter notebook

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

Introspection

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

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

In [2]: b?
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
Built-in mutable sequence.

If no argument is given, the constructor creates a new empty
list.
The argument must be an iterable if specified.

In [3]: print?
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout,
flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current
sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
```

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

This is referred to as *object introspection*. If the object is a function or instance method, the docstring, if defined, will also be shown. Suppose we'd written the following function (which you can reproduce in IPython or Jupyter):

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

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

Then using `?` shows us the docstring:

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

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

```
In [7]: add_numbers??
Signature: add_numbers(a, b)
Source:
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
File:      <ipython-input-9-6a548a216e27>
Type:      function
```

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

```
In [10]: np.*load*?
np.__loader__
np.load
np.loads
np.loadtxt
```

The %run Command

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

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

a = 5
b = 6
c = 7.5

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

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

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

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

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

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

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

> **NOTE**
>
> If you want to give a script access to variables already defined in the interactive IPython namespace, use `%run -i` instead of plain `%run`.

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

```
>>> %load ipython_script_test.py

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

    a = 5
    b = 6
    c = 7.5

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

Interrupting running code

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

WARNING

When a piece of Python code has called into some compiled extension modules, pressing Ctrl-C will not always cause the program execution to stop immediately. In such cases, you will have to either wait until control is returned to the Python interpreter, or in more dire circumstances, forcibly terminate the Python process in your operating system (such as using Task Manager on Windows or the `kill` command on Linux).

Executing Code from the Clipboard

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

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

    y = 8
```

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

```
In [17]: %paste
x = 5
y = 7
if x > 5:
    x += 1

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

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

```
In [18]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:x = 5
:y = 7
:if x > 5:
:    x += 1
:
:    y = 8
:--
```

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

Terminal Keyboard Shortcuts

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

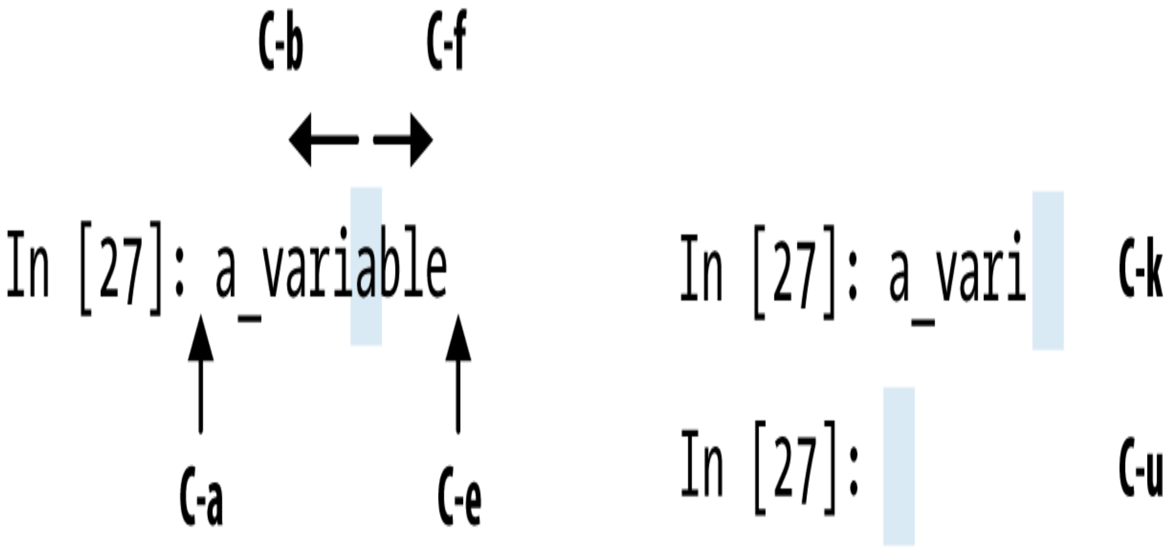


Figure 2-5. Illustration of some keyboard shortcuts in the IPython shell

Table 2-1. Standard IPython keyboard shortcuts

Keyboard shortcut	Description
Ctrl-P or up-arrow	Search backward in command history for commands starting with currently entered text
Ctrl-N or down-arrow	Search forward in command history for commands starting with currently entered text
Ctrl-R	Readline-style reverse history search (partial matching)
Ctrl-Shift-V	Paste text from clipboard
Ctrl-C	Interrupt currently executing code
Ctrl-A	Move cursor to beginning of line
Ctrl-E	Move cursor to end of line
Ctrl-K	Delete text from cursor until end of line
Ctrl-U	Discard all text on current line
Ctrl-F	Move cursor forward one character
Ctrl-B	Move cursor back one character
Ctrl-L	Clear screen

Note that Jupyter notebooks have a largely separate set of keyboard shortcuts for navigation and editing. Since these shortcuts have evolved more rapidly than IPython's, I encourage you to explore the integrated help system in the Jupyter notebook's menus.

About Magic Commands

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

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

In [20]: %timeit np.dot(a, a)
92.5 µs ± 3.43 µs per loop (mean ± std. dev. of 7 runs, 10000
loops each)
```

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

```
In [21]: %debug?
Docstring:
::

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

Activate the interactive debugger.

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

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

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

.. versionchanged:: 7.3
    When running code, user variables are no longer expanded,
    the magic line is always left unmodified.

positional arguments:
  statement             Code to run in debugger. You can omit
this in cell magic mode.

optional arguments:
  --breakpoint <FILE:LINE>, -b <FILE:LINE>
                        Set break point at LINE in FILE.
```

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

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

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

In [23]: foo = %pwd

In [24]: foo
Out[24]: '/home/wesm/code/pydata-book'
```

Since IPython’s documentation is accessible from within the system, I encourage you to explore all of the special commands available by typing `%quickref` or `%magic`. Table 2-2 highlights some of the most critical ones for being productive in interactive computing and Python development in IPython.

Table 2-2. Some frequently used IPython magic commands

Command	Description
%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 preformatted 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 a single statement
%timeit *statement*	Run a statement multiple times to compute an ensemble 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

Matplotlib Integration

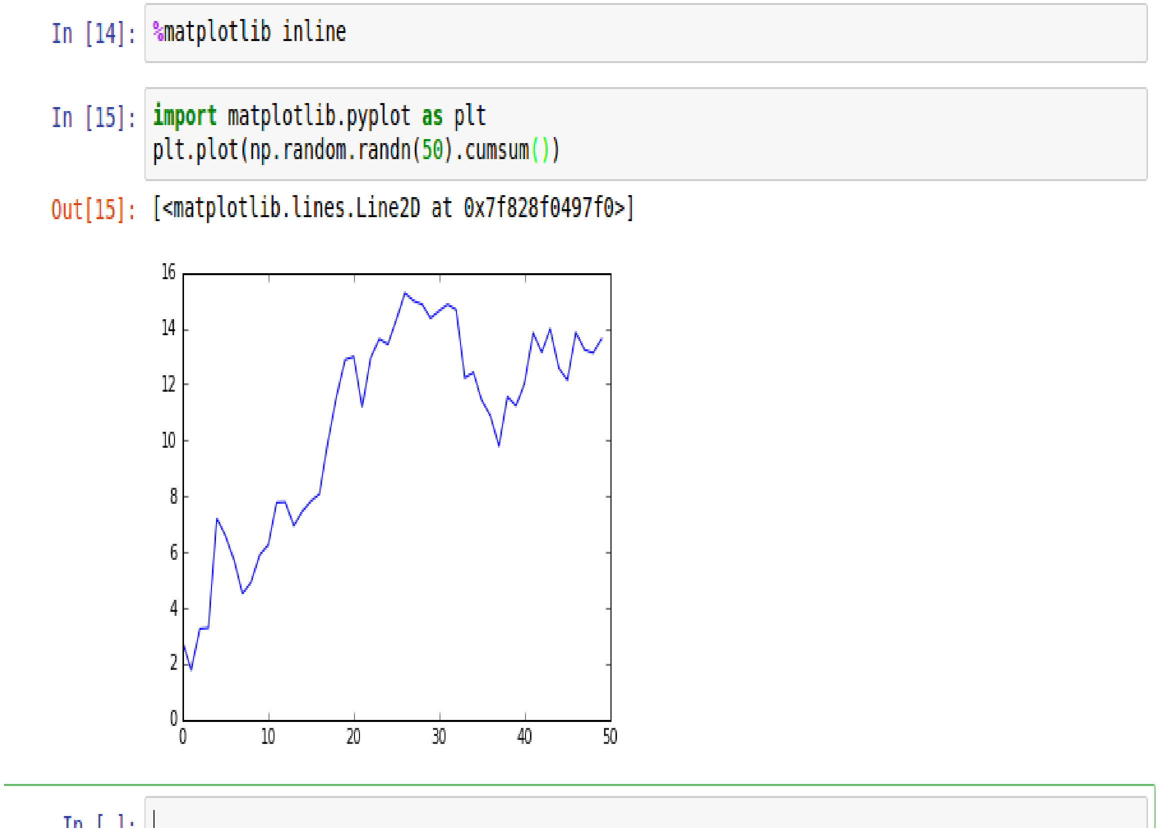
One reason for IPython's popularity in analytical computing is that it integrates well with data visualization and other user interface libraries like matplotlib. Don't worry if you have never used matplotlib before; it will be discussed in more detail later in this book. The `%matplotlib` magic function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

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

```
In [26]: %matplotlib
Using matplotlib backend: Qt5Agg
```

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

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



```
In [14]: %matplotlib inline

In [15]: import matplotlib.pyplot as plt
         plt.plot(np.random.randn(50).cumsum())

Out[15]: [<matplotlib.lines.Line2D at 0x7f828f0497f0>]
```

```
In [ ]: |
```

Figure 2-6. Jupyter inline matplotlib plotting

2.3 Python Language Basics

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

Language Semantics

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

Indentation, not braces

Python uses whitespace (tabs or spaces) to structure code instead of using braces as in many other languages like R, C++, Java, and Perl. Consider a `for` loop from a sorting algorithm:

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

A colon denotes the start of an indented code block after which all of the code must be indented by the same amount until the end of the block.

Love it or hate it, significant whitespace is a fact of life for Python programmers. While it may seem foreign at first, you will hopefully grow accustomed in time.

> **NOTE**
>
> I strongly recommend using *four spaces* as your default indentation and replacing tabs with four spaces. Many text editors have a setting that will replace tab stops with spaces automatically (do this!). Some people use tabs or a different number of spaces, with two spaces not being terribly uncommon. By and large, four spaces is the standard adopted by the vast majority of Python programmers (in particular, it's what is recommended in PEP 8, the official Python style guide), 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 can make code less readable.

Everything is an object

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

Comments

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

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

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

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

Function and object method calls

You call functions using parentheses and passing zero or more arguments, optionally assigning the returned value to a variable:

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

Almost every object in Python has attached functions, known as *methods*, that have access to the object’s internal contents. You can call them using the following syntax:

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

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

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

More on this later.

Variables and argument passing

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

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

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

```
In [9]: b = a

In [10]: b
Out[10]: [1, 2, 3]
```

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

```
In [11]: a.append(4)

In [12]: b
Out[12]: [1, 2, 3, 4]
```

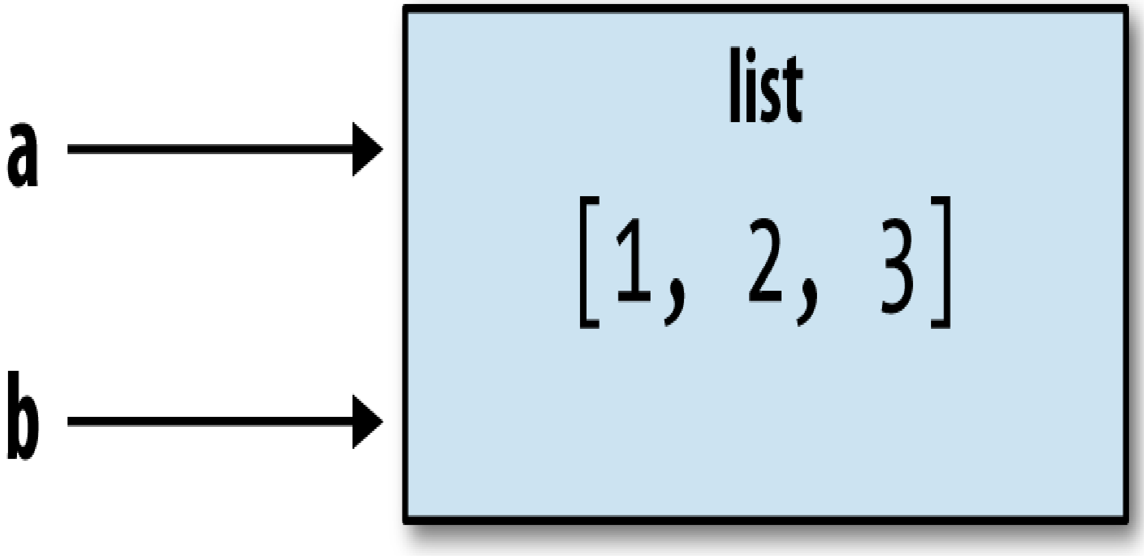


Figure 2-7. Two references for the same object

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

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

When you pass objects as arguments to a function, new local variables are created referencing the original objects without any copying. If you bind a new object to a variable inside a function, that will not overwrite a variable of the same name in the “scope” outside of the function (the “parent scope”). It is therefore possible to alter the internals of a mutable argument. Suppose we had the following function:

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

Then we have:

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

In [28]: append_element(data, 4)

In [29]: data
Out[29]: [1, 2, 3, 4]
```

Dynamic references, strong types

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

```
In [13]: a = 5

In [14]: type(a)
Out[14]: int

In [15]: a = 'foo'

In [16]: type(a)
Out[16]: 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 [17]: '5' + 5
------------------------------------------------------------------
----------
TypeError                                 Traceback (most recent
call last)
<ipython-input-17-4dd8efb5fac1> in <module>
----> 1 '5' + 5
TypeError: can only concatenate str (not "int") to str
```

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 [18]: a = 4.5

In [19]: b = 2

# String formatting, to be visited later
In [20]: print('a is {0}, b is {1}'.format(type(a), type(b)))
a is <class 'float'>, b is <class 'int'>

In [21]: a / b
Out[21]: 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 [22]: a = 5

In [23]: isinstance(a, int)
Out[23]: 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 [24]: a = 5; b = 4.5

In [25]: isinstance(a, (int, float))
Out[25]: True

In [26]: isinstance(b, (int, float))
Out[26]: True
```

Attributes and methods

Objects in Python typically have both attributes (other Python objects stored "inside" the object) and methods (functions associated with an object that can have access to the object's internal data). Both of them are accessed via the syntax *obj.attribute_name*:

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

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

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

```
In [28]: getattr(a, 'split')
Out[28]: <function str.split(sep=None, maxsplit=-1)>
```

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

Duck typing

Often you may not care about the type of an object but rather only whether it has certain methods or behavior. This is sometimes called "duck typing," after the saying "If it walks like a duck and quacks like a duck, then it's a duck." For example, you can verify that an object is iterable if it implements the *iterator protocol*. For many objects, this means it has a `__iter__` "magic method", though an alternative and better way to check is to try using the `iter` function:

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

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

```
In [30]: isiterable('a string')
Out[30]: True

In [31]: isiterable([1, 2, 3])
Out[31]: True

In [32]: isiterable(5)
Out[32]: False
```

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

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

Imports

In Python a *module* is simply a file with the *.py* extension containing Python code. Suppose that we had the following module:

```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

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

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

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

Or equivalently:

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

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

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

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

Binary operators and comparisons

Most of the binary math operations and comparisons use familiar mathematical syntax used in other programming langauges:

```
In [33]: 5 - 7
Out[33]: -2

In [34]: 12 + 21.5
Out[34]: 33.5

In [35]: 5 <= 2
Out[35]: False
```

See Table 2-3 for all of the available binary operators.

To check if two variables refer to the same object, use the `is` keyword. `is not` cann analogously be used to check that two objects are not the same:

```
In [36]: a = [1, 2, 3]
In [37]: b = a

In [38]: c = list(a)

In [39]: a is b
Out[39]: True

In [40]: a is not c
Out[40]: True
```

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

```
In [41]: a == c
Out[41]: True
```

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

```
In [42]: a = None

In [43]: a is None
Out[43]: True
```

Table 2-3. 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 the same Python object
`a is not b`	`True` if `a` and `b` reference different Python objects

Mutable and immutable objects

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

```
In [44]: a_list = ['foo', 2, [4, 5]]

In [45]: a_list[2] = (3, 4)

In [46]: a_list
Out[46]: ['foo', 2, (3, 4)]
```

Others, like strings and tuples, are immutable, which means their internal data cannot be changed:

```
In [47]: a_tuple = (3, 5, (4, 5))

In [48]: a_tuple[1] = 'four'
-----------------------------------------------------------------
----------
TypeError                                 Traceback (most recent
call last)
<ipython-input-48-23fe12da1ba6> in <module>
----> 1 a_tuple[1] = 'four'
TypeError: 'tuple' object does not support item assignment
```

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

Scalar Types

Python along with its standard library has a small set of built-in types for handling numerical data, strings, boolean (`True` or `False`) values, and dates and time. These "single value" types are sometimes called *scalar types* and we refer to them in this book as scalars. See Table 2-4 for a list of the main scalar types. Date and time handling will be discussed separately, as these are provided by the `datetime` module in the standard library.

Table 2-4. Standard Python scalar types

Type	Description
`None`	The Python “null” value (only one instance of the `None` object exists)
`str`	String type; holds Unicode strings using UTF-8 encoding
`bytes`	Raw binary data
`float`	Double-precision (64-bit) floating-point number (note there is no separate `double` type)
`bool`	A `True` or `False` value
`int`	Arbitrary precision signed integer

Numeric types

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

```
In [49]: ival = 17239871

In [50]: ival ** 6
Out[50]: 26254519291092456596965462913230729701102721
```

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

```
In [51]: fval = 7.243

In [52]: fval2 = 6.78e-5
```

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

```
In [53]: 3 / 2
Out[53]: 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 [54]: 3 // 2
Out[54]: 1
```

Strings

Many people use Python for its built-in string handling capabilities. You can write *string literals* using either single quotes `'` or double quotes `"`:

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

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

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

It may surprise you that this string `c` actually contains four lines of text; the line breaks after `"""` and after `lines` are included in the string. We can count the new line characters with the `count` method on `c`:

```
In [56]: c.count('\n')
Out[56]: 3
```

Python strings are immutable; you cannot modify a string:

```
In [57]: a = 'this is a string'

In [58]: a[10] = 'f'
-----------------------------------------------------------------
----------
TypeError                                 Traceback (most recent
call last)
<ipython-input-58-2151a30ed055> in <module>
----> 1 a[10] = 'f'
```

```
TypeError: 'str' object does not support item assignment

In [59]: b = a.replace('string', 'longer string')

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

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

```
In [61]: a
Out[61]: 'this is a string'
```

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

```
In [62]: a = 5.6

In [63]: s = str(a)

In [64]: print(s)
5.6
```

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

```
In [65]: s = 'python'

In [66]: list(s)
Out[66]: ['p', 'y', 't', 'h', 'o', 'n']

In [67]: s[:3]
Out[67]: 'pyt'
```

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

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

```
In [68]: s = '12\\34'

In [69]: print(s)
12\34
```

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

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

In [71]: s
Out[71]: 'this\\has\\no\\special\\characters'
```

The `r` stands for *raw*.

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

```
In [72]: a = 'this is the first half '

In [73]: b = 'and this is the second half'

In [74]: a + b
Out[74]: 'this is the first half and this is the second half'
```

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

```
In [75]: template = '{0:.2f} {1:s} are worth US${2:d}'
```

In this string,

- `{0:.2f}` means to format the first argument as a floating-point number with two decimal places.
- `{1:s}` means to format the second argument as a string.

- `{2:d}` means to format the third argument as an exact integer.

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

```
In [76]: template.format(88.46, 'Argentine Pesos', 1)
Out[76]: '88.46 Argentine Pesos are worth US$1'
```

Python 3.6 introduced a new feature called “f-strings” (short for “Formatted string literals”) which can make creating formatted strings even more convenient. To create an f-string, write the character `f` immediately preceding a string literal. Within the string, enclose Python expressions in curly braces to substitute the value of the expression into the formatted string:

```
In [77]: amount = 10

In [78]: rate = 88.46

In [79]: currency = 'Pesos'

In [80]: result = f'{amount} {currency} is worth US${amount /
rate}'
```

Format specifiers can be added after each expression using the same syntax as with the string templates above:

```
In [81]: f'{amount} {currency} is worth US${amount / rate:.2f}'
Out[81]: '10 Pesos is worth US$0.11'
```

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

I discuss general string processing as it relates to data analysis in more detail in [Link to Come].

Bytes and Unicode

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

```
In [82]: val = "español"

In [83]: val
Out[83]: 'español'
```

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

```
In [84]: val_utf8 = val.encode('utf-8')

In [85]: val_utf8
Out[85]: b'espa\xc3\xb1ol'

In [86]: type(val_utf8)
Out[86]: bytes
```

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

```
In [87]: val_utf8.decode('utf-8')
Out[87]: 'español'
```

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

```
In [88]: val.encode('latin1')
Out[88]: b'espa\xf1ol'

In [89]: val.encode('utf-16')
Out[89]: b'\xff\xfee\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'

In [90]: val.encode('utf-16le')
Out[90]: b'e\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

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

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

```
In [91]: bytes_val = b'this is bytes'

In [92]: bytes_val
Out[92]: b'this is bytes'

In [93]: decoded = bytes_val.decode('utf8')

In [94]: decoded  # this is str (Unicode) now
Out[94]: 'this is bytes'
```

Booleans

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

```
In [95]: True and True
Out[95]: True

In [96]: False or True
Out[96]: True
```

When converted to numbers, False becomes 0 and True becomes 1:

```
In [97]: int(False)
Out[97]: 0

In [98]: int(True)
Out[98]: 1
```

Type casting

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

```
In [99]: s = '3.14159'

In [100]: fval = float(s)

In [101]: type(fval)
Out[101]: float

In [102]: int(fval)
Out[102]: 3

In [103]: bool(fval)
Out[103]: True

In [104]: bool(0)
Out[104]: False
```

None

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

```
In [105]: a = None

In [106]: a is None
Out[106]: True

In [107]: b = 5

In [108]: b is not None
Out[108]: True
```

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

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

    if c is not None:
        result = result * c

    return result
```

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

```
In [109]: type(None)
Out[109]: NoneType

In [110]: None is None
Out[110]: True
```

Dates and times

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

```
In [111]: from datetime import datetime, date, time

In [112]: dt = datetime(2011, 10, 29, 20, 30, 21)

In [113]: dt.day
Out[113]: 29

In [114]: dt.minute
Out[114]: 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 [115]: dt.date()
Out[115]: datetime.date(2011, 10, 29)

In [116]: dt.time()
Out[116]: datetime.time(20, 30, 21)
```

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

```
In [117]: dt.strftime('%m/%d/%Y %H:%M')
Out[117]: '10/29/2011 20:30'
```

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

```
In [118]: datetime.strptime('20091031', '%Y%m%d')
Out[118]: datetime.datetime(2009, 10, 31, 0, 0)
```

See Table 2-5 for a full list of format specifications.

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

```
In [119]: dt_hour = dt.replace(minute=0, second=0)

In [120]: dt_hour
Out[120]: datetime.datetime(2011, 10, 29, 20, 0)
```

Since datetime.datetime is an immutable type, methods like these always produce new objects. So in the above, dt is not modified by replace:

```
In [121]: dt
Out[121]: datetime.datetime(2011, 10, 29, 20, 30, 21)
```

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

```
In [122]: dt2 = datetime(2011, 11, 15, 22, 30)

In [123]: delta = dt2 - dt

In [124]: delta
Out[124]: datetime.timedelta(days=17, seconds=7179)

In [125]: type(delta)
Out[125]: datetime.timedelta
```

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

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

```
In [126]: dt
Out[126]: datetime.datetime(2011, 10, 29, 20, 30, 21)
```

```
In [127]: dt + delta
Out[127]: datetime.datetime(2011, 11, 15, 22, 30)
```

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

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

Control Flow

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

if, elif, and else

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

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

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

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

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

```
In [128]: a = 5; b = 7

In [129]: c = 8; d = 4

In [130]: if a < b or c > d:
   .....:     print('Made it')
Made it
```

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

It is also possible to chain comparisons:

```
In [131]: 4 > 3 > 2 > 1
Out[131]: 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
```

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

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

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

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

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

```
In [132]: for i in range(4):
   .....:     for j in range(4):
   .....:         if j > i:
   .....:             break
   .....:         print((i, j))
   .....:
(0, 0)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
```

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

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

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

while loops

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

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

pass

`pass` is the "no-op" (or "do nothing") statement in Python. It can be used in blocks where no action is to be taken (or as a placeholder for code not yet implemented); it is only required because Python uses whitespace to delimit blocks:

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

range

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

```
In [133]: range(10)
Out[133]: range(0, 10)

In [134]: list(range(10))
Out[134]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

```
In [135]: list(range(0, 20, 2))
Out[135]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [136]: list(range(5, 0, -1))
Out[136]: [5, 4, 3, 2, 1]
```

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

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

While you can use functions like `list` to store all the integers generated by `range` in some other data structure, often the default iterator form will be what you want. This snippet sums all numbers from 0 to 99,999 that are multiples of 3 or 5:

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

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

Ternary expressions

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

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

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

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

This is a more concrete example:

```
In [137]: x = 5

In [138]: 'Non-negative' if x >= 0 else 'Negative'
Out[138]: 'Non-negative'
```

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

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

2.4 Conclusion

This chapter has provided a brief introduction to some basic Python language concepts and the IPython and Jupyter programming environments. In the next chapter, I will discuss many built-in data types, functions, and input-output utilities that will be used continuously throughout the rest of the book.

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

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at arufino@oreilly.com.

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

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

3.1 Data Structures and Sequences

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

Tuple

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

```
In [2]: tup = 4, 5, 6

In [3]: tup
Out[3]: (4, 5, 6)
```

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

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

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

You can convert any sequence or iterator to a tuple by invoking `tuple`:

```
In [6]: tuple([4, 0, 2])
Out[6]: (4, 0, 2)

In [7]: tup = tuple('string')

In [8]: tup
Out[8]: ('s', 't', 'r', 'i', 'n', 'g')
```

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

```
In [9]: tup[0]
Out[9]: 's'
```

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

```
In [10]: tup = tuple(['foo', [1, 2], True])

In [11]: tup[2] = False
-----------------------------------------------------------------
----------
TypeError                                 Traceback (most recent
call last)
<ipython-input-11-b89d0c4ae599> in <module>
----> 1 tup[2] = False
TypeError: 'tuple' object does not support item assignment
```

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

```
In [12]: tup[1].append(3)

In [13]: tup
Out[13]: ('foo', [1, 2, 3], True)
```

You can concatenate tuples using the + operator to produce longer tuples:

```
In [14]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[14]: (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 [15]: ('foo', 'bar') * 4
Out[15]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

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

Unpacking tuples

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

```
In [16]: tup = (4, 5, 6)

In [17]: a, b, c = tup
In [18]: b
Out[18]: 5
```

Even sequences with nested tuples can be unpacked:

```
In [19]: tup = 4, 5, (6, 7)

In [20]: a, b, (c, d) = tup

In [21]: d
Out[21]: 7
```

Using this functionality you can easily swap variable names, a task which in many languages might look like:

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

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

```
In [22]: a, b = 1, 2

In [23]: a
Out[23]: 1

In [24]: b
Out[24]: 2

In [25]: b, a = a, b

In [26]: a
Out[26]: 2

In [27]: b
Out[27]: 1
```

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

```
In [28]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
In [29]: for a, b, c in seq:
   ....:     print('a={0}, b={1}, c={2}'.format(a, b, c))
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

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

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

```
In [30]: values = 1, 2, 3, 4, 5

In [31]: a, b, *rest = values

In [32]: a, b
Out[32]: (1, 2)

In [33]: rest
Out[33]: [3, 4, 5]
```

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

```
In [34]: a, b, *_ = values
```

Tuple methods

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

```
In [35]: a = (1, 2, 2, 2, 3, 4, 2)
In [36]: a.count(2)
Out[36]: 4
```

List

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

```
In [37]: a_list = [2, 3, 7, None]

In [38]: tup = ('foo', 'bar', 'baz')

In [39]: b_list = list(tup)

In [40]: b_list
Out[40]: ['foo', 'bar', 'baz']

In [41]: b_list[1] = 'peekaboo'

In [42]: b_list
Out[42]: ['foo', 'peekaboo', 'baz']
```

Lists and tuples are semantically similar (though tuples cannot be modified) and can be used interchangeably in many functions.

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

```
In [43]: gen = range(10)

In [44]: gen
Out[44]: range(0, 10)

In [45]: list(gen)
Out[45]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Adding and removing elements

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

```
In [46]: b_list.append('dwarf')

In [47]: b_list
Out[47]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

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

```
In [48]: b_list.insert(1, 'red')

In [49]: b_list
Out[49]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

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

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

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

```
In [50]: b_list.pop(2)
Out[50]: 'peekaboo'

In [51]: b_list
Out[51]: ['foo', 'red', 'baz', 'dwarf']
```

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

```
In [52]: b_list.append('foo')

In [53]: b_list
Out[53]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [54]: b_list.remove('foo')
In [55]: b_list
Out[55]: ['red', 'baz', 'dwarf', 'foo']
```

If performance is not a concern, by using `append` and `remove`, you can use a Python list as a set-like data structure (although Python has actual set objects, discussed later).

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

```
In [56]: 'dwarf' in b_list
Out[56]: True
```

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

```
In [57]: 'dwarf' not in b_list
Out[57]: False
```

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

Concatenating and combining lists

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

```
In [58]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[58]: [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 [59]: x = [4, None, 'foo']

In [60]: x.extend([7, 8, (2, 3)])

In [61]: x
Out[61]: [4, None, 'foo', 7, 8, (2, 3)]
```

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

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

is faster than the concatenative alternative:

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

Sorting

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

```
In [62]: a = [7, 2, 5, 1, 3]

In [63]: a.sort()

In [64]: a
Out[64]: [1, 2, 3, 5, 7]
```

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

```
In [65]: b = ['saw', 'small', 'He', 'foxes', 'six']

In [66]: b.sort(key=len)

In [67]: b
Out[67]: ['He', 'saw', 'six', 'small', 'foxes']
```

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

Binary search and maintaining a sorted list

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

```
In [68]: import bisect

In [69]: c = [1, 2, 2, 2, 3, 4, 7]

In [70]: bisect.bisect(c, 2)
Out[70]: 4

In [71]: bisect.bisect(c, 5)
Out[71]: 6

In [72]: bisect.insort(c, 6)

In [73]: c
Out[73]: [1, 2, 2, 2, 3, 4, 6, 7]
```

CAUTION

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

Slicing

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

```
In [74]: seq = [7, 2, 3, 7, 5, 6, 0, 1]
In [75]: seq[1:5]
Out[75]: [2, 3, 7, 5]
```

Slices can also be assigned to with a sequence:

```
In [76]: seq[3:5] = [6, 3]

In [77]: seq
Out[77]: [7, 2, 3, 6, 3, 6, 0, 1]
```

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

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

```
In [78]: seq[:5]
Out[78]: [7, 2, 3, 6, 3]

In [79]: seq[3:]
Out[79]: [6, 3, 6, 0, 1]
```

Negative indices slice the sequence relative to the end:

```
In [80]: seq[-4:]
Out[80]: [3, 6, 0, 1]

In [81]: seq[-6:-2]
Out[81]: [3, 6, 3, 6]
```

Slicing semantics takes a bit of getting used to, especially if you're coming from R or MATLAB. See Figure 3-1 for a helpful illustration of slicing with positive and negative integers. In the figure, the indices are shown at the "bin edges" to help show where the slice selections start and stop using positive or negative indices.

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

```
In [82]: seq[::2]
Out[82]: [7, 3, 3, 0]
```

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

```
In [83]: seq[::-1]
Out[83]: [1, 0, 6, 3, 6, 3, 2, 7]
```

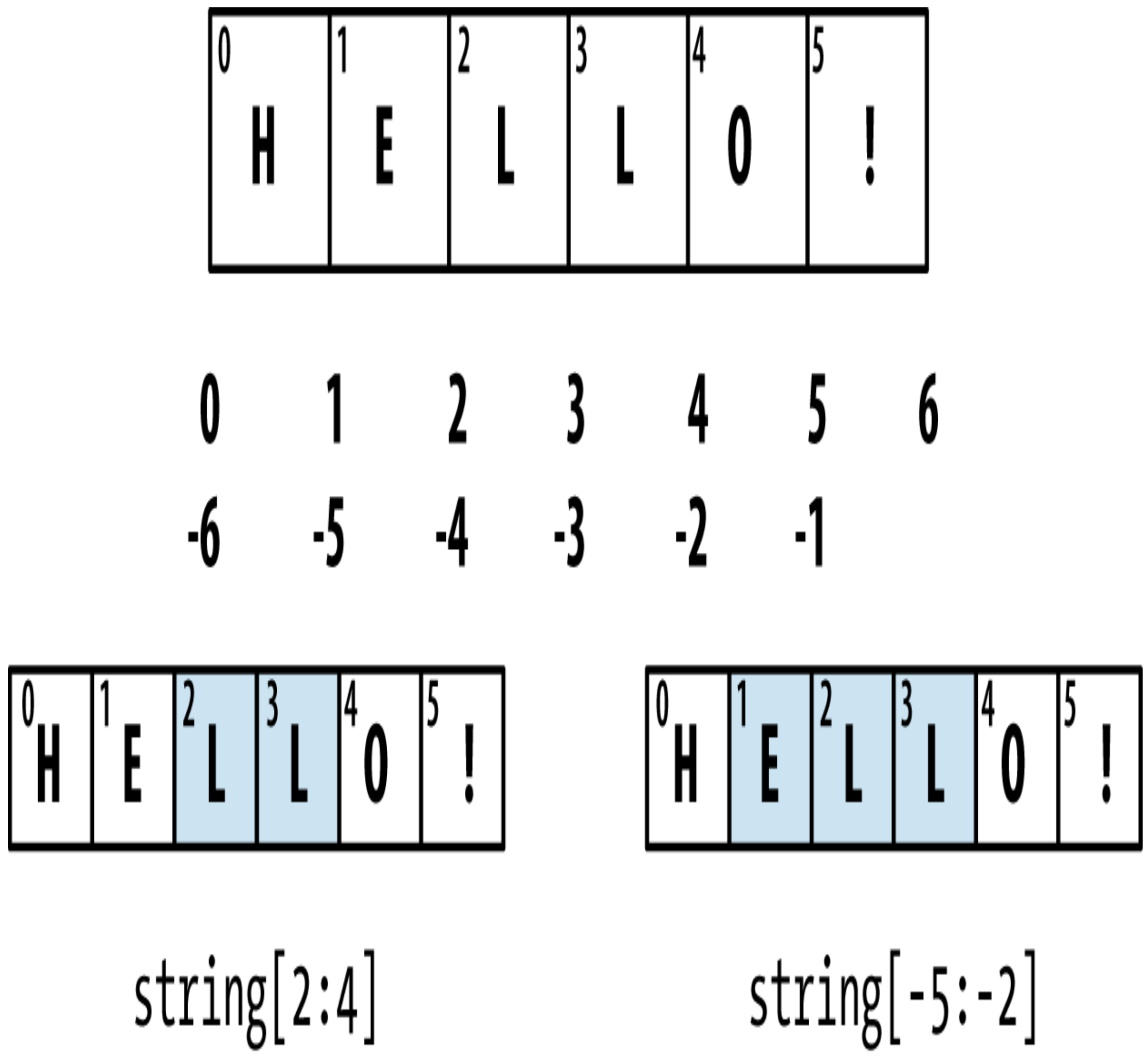


Figure 3-1. Illustration of Python slicing conventions

Built-in Sequence Functions

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

enumerate

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

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

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

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

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

```
In [84]: some_list = ['foo', 'bar', 'baz']

In [85]: mapping = {}

In [86]: for i, v in enumerate(some_list):
   ....:     mapping[v] = i

In [87]: mapping
Out[87]: {'foo': 0, 'bar': 1, 'baz': 2}
```

sorted

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

```
In [88]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[88]: [0, 1, 2, 2, 3, 6, 7]
```

```
In [89]: sorted('horse race')
Out[89]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

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

zip

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

```
In [90]: seq1 = ['foo', 'bar', 'baz']

In [91]: seq2 = ['one', 'two', 'three']

In [92]: zipped = zip(seq1, seq2)

In [93]: list(zipped)
Out[93]: [('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 [94]: seq3 = [False, True]

In [95]: list(zip(seq1, seq2, seq3))
Out[95]: [('foo', 'one', False), ('bar', 'two', True)]
```

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

```
In [96]: for i, (a, b) in enumerate(zip(seq1, seq2)):
   ....:     print('{0}: {1}, {2}'.format(i, a, b))
   ....:
0: foo, one
1: bar, two
2: baz, three
```

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

```
In [97]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
   ....:             ('Curt', 'Schilling')]

In [98]: first_names, last_names = zip(*pitchers)

In [99]: first_names
Out[99]: ('Nolan', 'Roger', 'Curt')

In [100]: last_names
Out[100]: ('Ryan', 'Clemens', 'Schilling')
```

reversed

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

```
In [101]: list(reversed(range(10)))
Out[101]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Keep in mind that `reversed` is a generator (to be discussed in some more detail later), so it does not create the reversed sequence until materialized (e.g., with `list` or a `for` loop).

dict

`dict` may be the most important built-in Python data structure. In other programming languages, dicts are sometimes called *hash maps* or *associative arrays*. A dict is an unordered collection of *key-value* pairs, where *key* and *value* are Python objects. Each key is associated with a value so that a value can be conveniently retrieved, inserted, modified, or deleted given a particular key. One approach for creating one is to use curly braces `{}` and colons to separate keys and values:

```
In [102]: empty_dict = {}

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

In [104]: d1
Out[104]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

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

```
In [105]: d1[7] = 'an integer'

In [106]: d1
Out[106]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}

In [107]: d1['b']
Out[107]: [1, 2, 3, 4]
```

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

```
In [108]: 'b' in d1
Out[108]: True
```

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

```
In [109]: d1[5] = 'some value'

In [110]: d1
Out[110]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value'}

In [111]: d1['dummy'] = 'another value'

In [112]: d1
Out[112]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value',
 'dummy': 'another value'}

In [113]: del d1[5]

In [114]: d1
Out[114]:
```

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

In [115]: ret = d1.pop('dummy')

In [116]: ret
Out[116]: 'another value'

In [117]: d1
Out[117]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

The `keys` and `values` method give you iterators of the dict's keys and values, respectively. The order of the keys depends on the order of their insertion, and these functions output the keys and values in the same respective order:

```
In [118]: list(d1.keys())
Out[118]: ['a', 'b', 7]

In [119]: list(d1.values())
Out[119]: ['some value', [1, 2, 3, 4], 'an integer']
```

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

```
In [120]: d1.update({'b' : 'foo', 'c' : 12})

In [121]: d1
Out[121]: {'a': 'some value', 'b': 'foo', 7: 'an integer', 'c':
12}
```

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

Creating dicts from sequences

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

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

Since a dict is essentially a collection of 2-tuples, the `dict` function accepts a list of 2-tuples:

```
In [122]: mapping = dict(zip(range(5), reversed(range(5))))

In [123]: mapping
Out[123]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Later we'll talk about *dict comprehensions*, another way to construct dicts.

Default values

It's 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, it may be that the values in a dict are another kind of collection, like a list. For example, you could imagine categorizing a list of words by their first letters as a dict of lists:

```
In [124]: words = ['apple', 'bat', 'bar', 'atom', 'book']

In [125]: by_letter = {}

In [126]: for word in words:
   .....:     letter = word[0]
   .....:     if letter not in by_letter:
```

```
   .....:         by_letter[letter] = [word]
   .....:     else:
   .....:         by_letter[letter].append(word)
   .....:

In [127]: by_letter
Out[127]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

The `setdefault` dict method can be used to simplify this workflow. The preceding `for` loop can be rewritten as:

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

The built-in `collections` module has a useful class, `defaultdict`, which makes this even easier. To create one, you pass a type or function for generating the default value for each slot in the dict:

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

Valid dict key types

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

```
In [128]: hash('string')
Out[128]: -9080266779362402528

In [129]: hash((1, 2, (2, 3)))
Out[129]: -9209053662355515447

In [130]: hash((1, 2, [2, 3])) # fails because lists are mutable
-----------------------------------------------------------------
----------
```

```
TypeError                                 Traceback (most recent
call last)
<ipython-input-130-473c35a62c0b> in <module>
----> 1 hash((1, 2, [2, 3])) # fails because lists are mutable
TypeError: unhashable type: 'list'
```

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

```
In [131]: d = {}

In [132]: d[tuple([1, 2, 3])] = 5

In [133]: d
Out[133]: {(1, 2, 3): 5}
```

set

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

```
In [134]: set([2, 2, 2, 1, 3, 3])
Out[134]: {1, 2, 3}

In [135]: {2, 2, 2, 1, 3, 3}
Out[135]: {1, 2, 3}
```

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

```
In [136]: a = {1, 2, 3, 4, 5}

In [137]: b = {3, 4, 5, 6, 7, 8}
```

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

```
In [138]: a.union(b)
Out[138]: {1, 2, 3, 4, 5, 6, 7, 8}

In [139]: a | b
Out[139]: {1, 2, 3, 4, 5, 6, 7, 8}
```

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

```
In [140]: a.intersection(b)
Out[140]: {3, 4, 5}

In [141]: a & b
Out[141]: {3, 4, 5}
```

See Table 3-1 for a list of commonly used set methods.

Table 3-1. Python set operations

Function	Alternative syntax	Description
`a.add(x)`	N/A	Add element `x` to the set `a`
`a.clear()`	N/A	Reset the set `a` to an empty state, discarding all of its elements
`a.remove(x)`	N/A	Remove element `x` from the set `a`
`a.pop()`	N/A	Remove an arbitrary element from the set `a`, raising `KeyError` if the set is empty
`a.union(b)`	`a | b`	All of the unique elements in `a` and `b`
`a.update(b)`	`a |= b`	Set the contents of `a` to be the union of the elements in `a` and `b`
`a.intersection(b)`	`a & b`	All of the elements in *both* `a` and `b`
`a.intersection_update(b)`	`a &= b`	Set the contents of `a` to be the intersection of the elements in `a` and `b`
`a.difference(b)`	`a - b`	The elements in `a` that are not in `b`
`a.difference_update(b)`	`a -= b`	Set `a` to the elements in `a` that are not in `b`
`a.symmetric_difference(b)`	`a ^ b`	All of the elements in either `a` or `b` but *not both*
`a.symmetric_difference_update(b)`	`a ^= b`	Set `a` to contain the elements in either `a` or `b` but *not both*
`a.issubset(b)`	`<=`	`True` if the elements of `a` are all contained in `b`
`a.issuperset(b)`	`>=`	`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

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

```
In [142]: c = a.copy()

In [143]: c |= b

In [144]: c
Out[144]: {1, 2, 3, 4, 5, 6, 7, 8}

In [145]: d = a.copy()

In [146]: d &= b

In [147]: d
Out[147]: {3, 4, 5}
```

Like a dict’s keys, a set’s elements generally must be immutable, and they must be *hashable* (which means that calling `hash` on a value does not raise an exception). In order to store list-like elements (or other mutable sequences) in a set, you can convert them to tuples:

```
In [148]: my_data = [1, 2, 3, 4]

In [149]: my_set = {tuple(my_data)}

In [150]: my_set
Out[150]: {(1, 2, 3, 4)}
```

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

```
In [151]: a_set = {1, 2, 3, 4, 5}

In [152]: {1, 2, 3}.issubset(a_set)
Out[152]: True

In [153]: a_set.issuperset({1, 2, 3})
Out[153]: True
```

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

```
In [154]: {1, 2, 3} == {3, 2, 1}
Out[154]: True
```

List, Set, and Dict Comprehensions

List comprehensions are are a convenient and widely-used Python language feature. They allow you to concisely form a new list by filtering the elements of a collection, transforming the elements passing the filter in one concise expression. They take the basic form:

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

This is equivalent to the following `for` loop:

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

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

```
In [155]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']

In [156]: [x.upper() for x in strings if len(x) > 2]
Out[156]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

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

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

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

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

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

```
In [157]: unique_lengths = {len(x) for x in strings}

In [158]: unique_lengths
Out[158]: {1, 2, 3, 4, 6}
```

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

```
In [159]: set(map(len, strings))
Out[159]: {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 [160]: loc_mapping = {val : index for index, val in
enumerate(strings)}

In [161]: loc_mapping
Out[161]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4,
'python': 5}
```

Nested list comprehensions

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

```
In [162]: all_data = [['John', 'Emily', 'Michael', 'Mary',
'Steven'],
   .....:              ['Maria', 'Juan', 'Javier', 'Natalia',
'Pilar']]
```

Suppose we wanted to get a single list containing all names with two or more `a`'s in them. We could certainly do this with a simple `for` loop:

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

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

```
In [164]: result = [name for names in all_data for name in names
   .....:            if name.count('a') >= 2]

In [165]: result
Out[165]: ['Maria', 'Natalia']
```

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

```
In [166]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [167]: flattened = [x for tup in some_tuples for x in tup]

In [168]: flattened
Out[168]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

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

```
flattened = []

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

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

```
In [170]: [[x for x in tup] for tup in some_tuples]
Out[170]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

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

3.2 Functions

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

Functions are declared with the `def` keyword. A function contains a block of code with an optional use of the with the `return` keyword:

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

When a line with `return` is reached, the value or expression after `return` is sent to the context where the function was called, for example:

```
In [172]: my_function(1, 2)
Out[172]: 4.5

In [173]: result = my_function(1, 2)
In [174]: result
Out[174]: 4.5
```

There is no issue with having multiple `return` statements. If Python reaches the end of a function without encountering a `return` statement, `None` is returned automatically. For example:

```
In [175]: def function_without_return(x):
   .....:     print(x)

In [176]: result = function_without_return('hello!')
hello!

In [177]: print(result)
None
```

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

```
In [178]: my_function(5, 6, z=0.7)
Out[178]: 0.06363636363636363

In [179]: my_function(3.14, 7, 3.5)
Out[179]: 35.49

In [180]: my_function(10, 20)
Out[180]: 45.0
```

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

NOTE

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

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

In some cases this can help with readability.

Namespaces, Scope, and Local Functions

Functions can access variables created inside the function as well as those outside the function in higher (or even *global*) scopes. An alternative and more descriptive name describing a variable scope in Python is a *namespace*. Any variables that are assigned within a function by default are assigned to the local namespace. The local namespace is created when the function is called and immediately populated by the function's arguments. After the function is finished, the local namespace is destroyed (with some exceptions that are outside the purview of this chapter). Consider the following function:

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

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

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

Each call to `func` will modify the list `a`:

```
In [182]: func()

In [183]: a
Out[183]: [0, 1, 2, 3, 4]

In [184]: func()

In [185]: a
Out[185]: [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
```

Assigning variables outside of the function's scope is possible, but those variables must be declared explicitly either using the global the `global` or nonlocal keywords:

```
In [186]: a = None

In [187]: def bind_a_variable():
   .....:     global a
   .....:     a = []
   .....: bind_a_variable()
   .....:

In [188]: print(a)
[]
```

`nonlocal` allows a function to modify variables defined in a higher level scope that is not global. Since its use is somewhat esoteric (I never use it in this book), I refer you to the Python documentation to learn more about it.

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

Returning Multiple Values

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

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

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

In data analysis and other scientific applications, you may find yourself doing this often. What’s happening here is that the function is actually just returning *one* object, a tuple, which is then being unpacked into the result variables. In the preceding example, we could have done this instead:

```
return_value = f()
```

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

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

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

Functions Are Objects

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

```
In [189]: states = ['   Alabama ', 'Georgia!', 'Georgia',
'georgia', 'FlOrIda',
   .....:           'south   carolina##', 'West virginia?']
```

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

```
import re

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

The result looks like this:

```
In [191]: clean_strings(states)
Out[191]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South   Carolina',
 'West Virginia']
```

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

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

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

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

Then we have the following:

```
In [193]: clean_strings(states, clean_ops)
Out[193]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South   Carolina',
 'West Virginia']
```

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

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

```
In [194]: for x in map(remove_punctuation, states):
   .....:     print(x)
Alabama
Georgia
Georgia
georgia
FlOrIda
south   carolina
West virginia
```

Anonymous (Lambda) Functions

Python has support for so-called *anonymous* or *lambda* functions, which are a way of writing functions consisting of a single statement, the result of which is the return value. They are defined with the `lambda` keyword, which has no meaning other than “we are declaring an anonymous function”:

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

equiv_anon = lambda x: x * 2
```

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

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

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

You could also have written `[x * 2 for x in ints]`, but here we were able to succinctly pass a custom operator to the `apply_to_list` function.

As another example, suppose you wanted to sort a collection of strings by the number of distinct letters in each string:

```
In [195]: strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

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

```
In [196]: strings.sort(key=lambda x: len(set(list(x))))

In [197]: strings
Out[197]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

NOTE

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

Currying: Partial Argument Application

Currying is computer science jargon (named after the mathematician Haskell Curry) that means deriving new functions from existing ones by *partial argument application*. For example, suppose we had a trivial function that adds two numbers together:

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

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

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

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

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

Generators

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

```
In [198]: some_dict = {'a': 1, 'b': 2, 'c': 3}

In [199]: for key in some_dict:
   .....:     print(key)
a
b
c
```

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

```
In [200]: dict_iterator = iter(some_dict)

In [201]: dict_iterator
Out[201]: <dict_keyiterator at 0x7fd550fe0090>
```

An iterator is any object that will yield objects to the Python interpreter when used in a context like a `for` loop. Most methods expecting a list or list-like object will also accept any iterable object. This includes built-in methods such as `min`, `max`, and `sum`, and type constructors like `list` and `tuple`:

```
In [202]: list(dict_iterator)
Out[202]: ['a', 'b', 'c']
```

A *generator* is a convenient way, similar to writing a normal function, to construct a new iterable object. Whereas normal functions execute and return a single result at a time, generators return a sequence of multiple results lazily, pausing after each one until the next one is requested. To create a generator, use the `yield` keyword instead of `return` in a function:

```
def squares(n=10):
    print('Generating squares from 1 to {0}'.format(n ** 2))
    for i in range(1, n + 1):
        yield i ** 2
```

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

```
In [204]: gen = squares()

In [205]: gen
Out[205]: <generator object squares at 0x7fd550fdbcf0>
```

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

```
In [206]: for x in gen:
   .....:     print(x, end=' ')
Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

Generator expresssions

Another way to make a generator is by using a *generator expression*. This is a generator analogue to list, dict, and set comprehensions. To create one, enclose what would otherwise be a list comprehension within parentheses instead of brackets:

```
In [207]: gen = (x ** 2 for x in range(100))

In [208]: gen
Out[208]: <generator object <genexpr> at 0x7fd550fdb900>
```

This is equivalent to the following more verbose generator:

```
def _make_gen():
    for x in range(100):
        yield x ** 2
gen = _make_gen()
```

Generator expressions can be used instead of list comprehensions as function arguments in some cases:

```
In [209]: sum(x ** 2 for x in range(100))
Out[209]: 328350

In [210]: dict((i, i **2) for i in range(5))
Out[210]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Depending on the number of elements produced by the comprehension expression, the generator version can sometimes be meaningfully faster.

itertools module

The standard library `itertools` module has a collection of generators for many common data algorithms. For example, `groupby` takes any sequence and a function, grouping consecutive elements in the sequence by return value of the function. Here’s an example:

```
In [211]: import itertools

In [212]: first_letter = lambda x: x[0]

In [213]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert',
'Steven']

In [214]: for letter, names in itertools.groupby(names,
first_letter):
   .....:     print(letter, list(names)) # names is a generator
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

See Table 3-2 for a list of a few other `itertools` functions I’ve frequently found helpful. You may like to check out the official Python documentation for more on this useful built-in utility module.

Table 3-2. Some useful itertools functions

Function	Description
`combinations(iterable, k)`	Generates a sequence of all possible `k`-tuples of elements in the iterable, ignoring order and without replacement (see also the companion function `combinations_with_replacement`)
`permutations(iterable, k)`	Generates a sequence of all possible `k`-tuples of elements in the iterable, respecting order
`groupby(iterable[, keyfunc])`	Generates `(key, sub-iterator)` for each unique key
`product(*iterables, repeat=1)`	Generates the Cartesian product of the input iterables as tuples, similar to a nested `for` loop

Errors and Exception Handling

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

```
In [215]: float('1.2345')
Out[215]: 1.2345

In [216]: float('something')
-----------------------------------------------------------------
----------
ValueError                                Traceback (most recent
call last)
<ipython-input-216-2649e4ade0e6> in <module>
----> 1 float('something')
ValueError: could not convert string to float: 'something'
```

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

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

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

```
In [218]: attempt_float('1.2345')
Out[218]: 1.2345

In [219]: attempt_float('something')
Out[219]: 'something'
```

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

```
In [220]: float((1, 2))
-----------------------------------------------------------------
----------
TypeError                                 Traceback (most recent
call last)
<ipython-input-220-82f777b0e564> in <module>
----> 1 float((1, 2))
TypeError: float() argument must be a string or a number, not
'tuple'
```

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

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

We have then:

```
In [222]: attempt_float((1, 2))
-----------------------------------------------------------------
----------
TypeError                                 Traceback (most recent
call last)
<ipython-input-222-8b0026e9e6b7> in <module>
----> 1 attempt_float((1, 2))
<ipython-input-221-6209ddecd2b5> in attempt_float(x)
      1 def attempt_float(x):
      2     try:
----> 3         return float(x)
      4     except ValueError:
      5         return x
TypeError: float() argument must be a string or a number, not
'tuple'
```

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

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

In some cases, you may not want to suppress an exception, but you want some code to be executed regardless of whether the code in the `try` block succeeds or not. To do this, use `finally`:

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

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

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

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

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

Exceptions in IPython

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

```
In [10]: %run examples/ipython_bug.py
-----------------------------------------------------------------
----------
AssertionError                          Traceback (most recent
call last)
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
     13     throws_an_exception()
     14
---> 15 calling_things()

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

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

AssertionError:
```

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

3.3 Files and the Operating System

Most of this book uses high-level tools like `pandas.read_csv` to read data files from disk into Python data structures. However, it’s important to understand the basics of how to work with files in Python. Fortunately, it’s relatively straightforward, which is one reason why Python is so popular for text and file munging.

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

```
In [224]: path = 'examples/segismundo.txt'

In [225]: f = open(path)
```

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

```
for line in f:
    pass
```

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

```
In [226]: lines = [x.rstrip() for x in open(path)]

In [227]: lines
Out[227]:
['Sueña el rico en su riqueza,',
 'que más cuidados le ofrece;',
 '',
 'sueña el pobre que padece',
 'su miseria y su pobreza;',
 '',
 'sueña el que a medrar empieza,',
 'sueña el que afana y pretende,',
 'sueña el que agravia y ofende,',
 '',
 'y en el mundo, en conclusión,',
 'todos sueñan lo que son,',
 'aunque ninguno lo entiende.',
 '']
```

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

```
In [228]: f.close()
```

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

```
In [229]: with open(path) as f:
   .....:     lines = [x.rstrip() for x in f]
```

This will automatically close the file `f` when exiting the `with` block. Failing to ensure that files are closed will not cause problems in many small programs or scripts, but it can be an issue in programs that need to interact with a large number of files.

If we had typed `f = open(path, 'w')`, a *new file* at *examples/segismundo.txt* would have been created (be careful!), overwriting any one in its place. There is also the `'x'` file mode, which creates a writable file but fails if the file path already exists. See Table 3-3 for a list of all valid file read/write modes.

For readable files, some of the most commonly used methods are `read`, `seek`, and `tell`. `read` returns a certain number of characters from the file. What constitutes a "character" is determined by the file's encoding (e.g., UTF-8) or simply raw bytes if the file is opened in binary mode:

```
In [230]: f = open(path)

In [231]: f.read(10)
Out[231]: 'Sueña el r'

In [232]: f2 = open(path, 'rb')  # Binary mode

In [233]: f2.read(10)
Out[233]: b'Sue\xc3\xb1a el '
```

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

```
In [234]: f.tell()
Out[234]: 11

In [235]: f2.tell()
Out[235]: 10
```

Even though we read 10 characters from the file, the position is 11 because it took that many bytes to decode 10 characters using the default encoding. You can check the default encoding in the `sys` module:

```
In [236]: import sys

In [237]: sys.getdefaultencoding()
Out[237]: 'utf-8'
```

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

```
In [238]: f.seek(3)
Out[238]: 3

In [239]: f.read(1)
Out[239]: 'ñ'
```

Lastly, we remember to close the files:

```
In [240]: f.close()

In [241]: f2.close()
```

Table 3-3. Python file modes

Mode	Description
`r`	Read-only mode
`w`	Write-only mode; creates a new file (erasing the data for any file with the same name)
`x`	Write-only mode; creates a new file, but fails if the file path already exists
`a`	Append to existing file (create the file if it does not already exist)
`r+`	Read and write
`b`	Add to mode for binary files (i.e., `'rb'` or `'wb'`)
`t`	Text mode for files (automatically decoding bytes to Unicode). This is the default if not specified. Add `t` to other modes to use this (i.e., `'rt'` or `'xt'`)

To write text to a file, you can use the file's `write` or `writelines` methods. For example, we could create a version of *examples/segismundo.txt* with no blank lines like so:

```
In [242]: path
Out[242]: 'examples/segismundo.txt'

In [243]: with open('tmp.txt', 'w') as handle:
   .....:     handle.writelines(x for x in open(path) if len(x) >
1)

In [244]: with open('tmp.txt') as f:
   .....:     lines = f.readlines()

In [245]: lines
Out[245]:
['Sueña el rico en su riqueza,\n',
 'que más cuidados le ofrece;\n',
 'sueña el pobre que padece\n',
 'su miseria y su pobreza;\n',
 'sueña el que a medrar empieza,\n',
 'sueña el que afana y pretende,\n',
 'sueña el que agravia y ofende,\n',
 'y en el mundo, en conclusión,\n',
 'todos sueñan lo que son,\n',
 'aunque ninguno lo entiende.\n']
```

See Table 3-4 for many of the most commonly used file methods.

Table 3-4. Important Python file methods or attributes

Method	Description
`read([size])`	Return data from file as a string, with optional `size` argument indicating the number of bytes to read
`readable()`	Returns `True` if the file supports `read` operations
`readlines([size])`	Return list of lines in the file, with optional `size` argument
`write(str)`	Write passed string to file
`writable()`	Returns `True` if the file supports `write` operations
`writelines(strings)`	Write passed sequence of strings to the file
`close()`	Close the handle
`flush()`	Flush the internal I/O buffer to disk
`seek(pos)`	Move to indicated file position (integer)
`seekable()`	Returns `True` if the file object supports seeking and thus random access (some file-like objects do not)
`tell()`	Return current file position as integer
`closed`	`True` if the file is closed
`encoding`	The encoding used to interpret bytes in the file as Unicode (typically UTF-8)

Bytes and Unicode with Files

The default behavior for Python files (whether readable or writable) is *text mode*, which means that you intend to work with Python strings (i.e., Unicode). This contrasts with *binary mode*, which you can obtain by appending `b` onto the file mode. Revisiting the file (which contains non-ASCII characters with UTF-8 encoding) from the previous section, we have:

```
In [248]: with open(path) as f:
   .....:     chars = f.read(10)

In [249]: chars
Out[249]: 'Sueña el r'
```

UTF-8 is a variable-length Unicode encoding, so when I requested some number of characters from the file, Python reads enough bytes (which could be as few as 10 or as many as 40 bytes) from the file to decode that many characters. If I open the file in `'rb'` mode instead, `read` requests exact numbers of bytes:

```
In [250]: with open(path, 'rb') as f:
   .....:     data = f.read(10)

In [251]: data
Out[251]: b'Sue\xc3\xb1a el '
```

Depending on the text encoding, you may be able to decode the bytes to a `str` object yourself, but only if each of the encoded Unicode characters is fully formed:

```
In [252]: data.decode('utf8')
Out[252]: 'Sueña el '

In [253]: data[:4].decode('utf8')
-----------------------------------------------------------------
----------
UnicodeDecodeError                        Traceback (most recent
call last)
<ipython-input-253-0ad9ad6a11bd> in <module>
----> 1 data[:4].decode('utf8')
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xc3 in
position 3: unexpecte
d end of data
```

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

```
In [254]: sink_path = 'sink.txt'
In [255]: with open(path) as source:
   .....:     with open(sink_path, 'xt', encoding='iso-8859-1')
as sink:
   .....:         sink.write(source.read())

In [256]: with open(sink_path, encoding='iso-8859-1') as f:
   .....:     print(f.read(10))
Sueña el r
```

Beware using `seek` when opening files in any mode other than binary. If the file position falls in the middle of the bytes defining a Unicode character, then subsequent reads will result in an error:

```
In [258]: f = open(path)

In [259]: f.read(5)
Out[259]: 'Sueña'

In [260]: f.seek(4)
Out[260]: 4

In [261]: f.read(1)
-----------------------------------------------------------------
----------
UnicodeDecodeError                       Traceback (most recent
call last)
<ipython-input-261-5a354f952aa4> in <module>
----> 1 f.read(1)
/miniconda/envs/book-env/lib/python3.8/codecs.py in decode(self,
input, final)
    320         # decode input (taking the buffer into account)
    321         data = self.buffer + input
--> 322         (result, consumed) = self._buffer_decode(data,
self.errors, final
)
    323         # keep undecoded input until the next call
    324         self.buffer = data[consumed:]
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb1 in
position 0: invalid s
tart byte

In [262]: f.close()
```

If you find yourself regularly doing data analysis on non-ASCII text data, mastering Python’s Unicode functionality will prove valuable. See Python’s online documentation for much more.

3.4 Conclusion

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

Chapter 4. NumPy Basics: Arrays and Vectorized Computation

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at arufino@oreilly.com.

NumPy, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python. Many computational packages providing scientific functionality use NumPy’s array objects as one of the standard interfaces *lingua franca* for data exchange.

Here are some of the things you’ll find in NumPy:

- ndarray, an efficient multidimensional array providing fast array-oriented arithmetic operations and flexible *broadcasting* capabilities.

- Mathematical functions for fast operations on entire arrays of data without having to write loops.

- Tools for reading/writing array data to disk and working with memory-mapped files.
- Linear algebra, random number generation, and Fourier transform capabilities.
- A C API for connecting NumPy with libraries written in C, C++, or FORTRAN.

Because NumPy provides an comprehensive and well-documented C API, it is straightforward to pass data to external libraries written in a low-level language and also for external libraries to return data to Python as NumPy arrays. This feature has made Python a language of choice for wrapping legacy C, C++, or Fortran codebases and giving them a dynamic and easy-to-use interface.

While NumPy by itself does not provide modeling or scientific functionality, having an understanding of NumPy arrays and array-oriented computing will help you use tools with array computing semantics, like pandas, much more effectively. Since NumPy is a large topic, I will cover many advanced NumPy features like broadcasting in more depth later (see Appendix A). Many of these advanced features are not needed to follow the rest of this book, but they may help you as you go deeper into scientific computing in Python.

For most data analysis applications, the main areas of functionality I'll focus on are:

- Fast array-based operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Data alignment and relational data manipulations for merging and joining together heterogeneous datasets

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

While NumPy provides a computational foundation for general numerical data processing, many readers will want to use pandas as the basis for most kinds of statistics or analytics, especially on tabular data. pandas also provides some more domain-specific functionality like time series manipulation, which is not present in NumPy.

> **NOTE**
>
> Array-oriented computing in Python traces its roots back to 1995, when Jim Hugunin created the Numeric library. Over the next 10 years, many scientific programming communities began doing array programming in Python, but the library ecosystem had become fragmented in the early 2000s. In 2005, Travis Oliphant was able to forge the NumPy project from the then Numeric and Numarray projects to bring the community together around a single array computing framework.

One of the reasons NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data. There are a number of reasons for this:

- NumPy internally stores data in a contiguous block of memory, independent of other built-in Python objects. NumPy's library of algorithms written in the C language can operate on this memory without any type checking or other overhead. NumPy arrays also use much less memory than built-in Python sequences.
- NumPy operations perform complex computations on entire arrays without the need for Python `for` loops, which can be slow for large sequences. NumPy is faster than regular Python code because its C-based algorithms avoid overhead present with regular interpreted Python code.

To give you an idea of the performance difference, consider a NumPy array of one million integers, and the equivalent Python list:

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

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

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

Now let’s multiply each sequence by 2:

```
In [10]: %time for _ in range(10): my_arr2 = my_arr * 2
CPU times: user 13.5 ms, sys: 3.84 ms, total: 17.3 ms
Wall time: 16.7 ms

In [11]: %time for _ in range(10): my_list2 = [x * 2 for x in
my_list]
CPU times: user 406 ms, sys: 101 ms, total: 507 ms
Wall time: 507 ms
```

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

4.1 The NumPy ndarray: A Multidimensional Array Object

One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large datasets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

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

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

# Generate some random data
```

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

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

I then write mathematical operations with `data`:

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

In [16]: data + data
Out[16]:
array([[-0.4094,  0.9579, -1.0389],
       [-1.1115,  3.9316,  2.7868]])
```

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

NOTE

In this chapter and throughout the book, I use the standard NumPy convention of always using `import numpy as np`. You are, of course, welcome to put `from numpy import *` in your code to avoid having to write `np.`, but I advise against making a habit of this. The `numpy` namespace is large and contains a number of functions whose names conflict with built-in Python functions (like `min` and `max`).

An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same type. Every array has a `shape`, a tuple indicating the size of each dimension, and a `dtype`, an object describing the *data type* of the array:

```
In [17]: data.shape
Out[17]: (2, 3)
```

```
In [18]: data.dtype
Out[18]: dtype('float64')
```

This chapter will introduce you to the basics of using NumPy arrays, and should be sufficient for following along with the rest of the book. While it’s not necessary to have a deep understanding of NumPy for many data analytical applications, becoming proficient in array-oriented programming and thinking is a key step along the way to becoming a scientific Python guru.

NOTE

Whenever you see “array,” “NumPy array,” or “ndarray” in the book text, in most cases they all refer to the ndarray object.

Creating ndarrays

The easiest way to create an array is to use the `array` function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data. For example, a list is a good candidate for conversion:

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

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

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

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

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

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

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

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

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

```
In [25]: arr2.ndim
Out[25]: 2

In [26]: arr2.shape
Out[26]: (2, 4)
```

Unless explicitly specified (more on this later), `np.array` tries to infer a good data type for the array that it creates. The data type is stored in a special `dtype` metadata object; for example, in the previous two examples we have:

```
In [27]: arr1.dtype
Out[27]: dtype('float64')

In [28]: arr2.dtype
Out[28]: dtype('int64')
```

In addition to `np.array`, there are a number of other functions for creating new arrays. As examples, `zeros` and `ones` create arrays of 0s or 1s, respectively, with a given length or shape. `empty` creates an array without initializing its values to any particular value. To create a higher dimensional array with these methods, pass a tuple for the shape:

```
In [29]: np.zeros(10)
Out[29]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [30]: np.zeros((3, 6))
Out[30]:
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])

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

```
Out[31]:
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],
       [[0., 0.],
        [0., 0.],
        [0., 0.]]])
```

CAUTION

It’s not safe to assume that `np.empty` will return an array of all zeros. This function returns uninitialized memory and thus may contain non-zero “garbage” values.

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

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

See Table 4-1 for a short list of standard array creation functions. Since NumPy is focused on numerical computing, the data type, if not specified, will in many cases be `float64` (floating point).

Table 4-1. Some important NumPy 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; 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 instead of a list
`ones, ones_like`	Produce an array of all 1s 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 0s instead
`empty, empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`full, full_like`	Produce an array of the given shape and dtype with all values set to the indicated “fill value” `full_like` takes another array and produces a filled array of the same shape and dtype
`eye, identity`	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

Data Types for ndarrays

The *data type* or `dtype` is a special object containing the information (or *metadata*, data about data) the ndarray needs to interpret a chunk of memory as a particular type of data:

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

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

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

```
In [36]: arr2.dtype
Out[36]: dtype('int32')
```

dtypes are a source of NumPy’s flexibility for interacting with data coming from other systems. In most cases they provide a mapping directly onto an underlying disk or memory representation, which makes it possible to read and write binary streams of data to disk and also to connect to code written in a low-level language like C or Fortran. The numerical dtypes are named the same way: a type name, like `float` or `int`, followed by a number indicating the number of bits per element. A standard double-precision floating-point value (what’s used under the hood in Python’s `float` object) takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as `float64`. See Table 4-2 for a full listing of NumPy’s supported data types.

NOTE

Don’t worry about memorizing the NumPy dtypes, especially if you’re a new user. It’s often only necessary to care about the general *kind* of data you’re dealing with, whether floating point, complex, integer, boolean, string, or general Python object. When you need more control over how data are stored in memory and on disk, especially large datasets, it is good to know that you have control over the storage type.

Table 4-2. NumPy data types

Type	Type code	Description
int8, uint8	i1, 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
int64, uint64	i8, u8	Signed and unsigned 64-bit integer types
float16	f2	Half-precision floating point
float32	f4 or f	Standard single-precision floating point; compatible with C float
float64	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; a value can be any Python object
string_	S	Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use 'S10'
unicode_	U	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')

You can explicitly convert or *cast* an array from one dtype to another using ndarray's astype method:

```
In [37]: arr = np.array([1, 2, 3, 4, 5])

In [38]: arr.dtype
Out[38]: dtype('int64')

In [39]: float_arr = arr.astype(np.float64)

In [40]: float_arr
Out[40]: array([1., 2., 3., 4., 5.])

In [41]: float_arr.dtype
Out[41]: dtype('float64')
```

In this example, integers were cast to floating point. If I cast some floating-point numbers to be of integer dtype, the decimal part will be truncated:

```
In [42]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [43]: arr
Out[43]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])

In [44]: arr.astype(np.int32)
Out[44]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

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

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

In [46]: numeric_strings.astype(float)
Out[46]: array([ 1.25, -9.6 , 42.  ])
```

CAUTION

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

If casting were to fail for some reason (like a string that cannot be converted to `float64`), a `ValueError` will be raised. Here I was a bit lazy and wrote `float` instead of `np.float64`; NumPy aliases the Python types to its own equivalent data dtypes.

You can also use another array's dtype attribute:

```
In [47]: int_array = np.arange(10)

In [48]: calibers = np.array([.22, .270, .357, .380, .44, .50],
dtype=np.float64)

In [49]: int_array.astype(calibers.dtype)
Out[49]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

There are shorthand type code strings you can also use to refer to a dtype:

```
In [50]: empty_uint32 = np.empty(8, dtype='u4')

In [51]: empty_uint32
Out[51]:
array([         0, 1075314688,          0, 1075707904,
0,
       1075838976,          0, 1072693248], dtype=uint32)
```

NOTE

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

Arithmetic with NumPy Arrays

Arrays are important because they enable you to express batch operations on data without writing any `for` loops. NumPy users call this *vectorization*. Any arithmetic operations between equal-size arrays applies the operation element-wise:

```
In [52]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])
In [53]: arr
Out[53]:
array([[1., 2., 3.],
       [4., 5., 6.]])

In [54]: arr * arr
Out[54]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])

In [55]: arr - arr
Out[55]:
array([[0., 0., 0.],
       [0., 0., 0.]])
```

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

```
In [56]: 1 / arr
Out[56]:
array([[1.    , 0.5   , 0.3333],
       [0.25  , 0.2   , 0.1667]])

In [57]: arr ** 0.5
Out[57]:
array([[1.    , 1.4142, 1.7321],
       [2.    , 2.2361, 2.4495]])
```

Comparisons between arrays of the same size yield boolean arrays:

```
In [58]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

In [59]: arr2
Out[59]:
array([[ 0.,  4.,  1.],
       [ 7.,  2., 12.]])

In [60]: arr2 > arr
Out[60]:
array([[False,  True, False],
       [ True, False,  True]])
```

Evaluating operations between differently sized arrays is called *broadcasting* and will be discussed in more detail in Appendix A. Having a deep understanding of broadcasting is not necessary for most of this book.

Basic Indexing and Slicing

NumPy array indexing is a deep topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
In [61]: arr = np.arange(10)

In [62]: arr
Out[62]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [63]: arr[5]
Out[63]: 5

In [64]: arr[5:8]
Out[64]: array([5, 6, 7])

In [65]: arr[5:8] = 12

In [66]: arr
Out[66]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

As you can see, if you assign a scalar value to a slice, as in `arr[5:8] = 12`, the value is propagated (or *broadcasted* henceforth) to the entire selection. An important first distinction from Python's built-in lists is that array slices are *views* on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array.

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

```
In [67]: arr_slice = arr[5:8]

In [68]: arr_slice
Out[68]: array([12, 12, 12])
```

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

```
In [69]: arr_slice[1] = 12345

In [70]: arr
Out[70]:
array([    0,     1,     2,     3,     4,    12, 12345,    12,
8,
            9])
```

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

```
In [71]: arr_slice[:] = 64

In [72]: arr
Out[72]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

If you are new to NumPy, you might be surprised by this, especially if you have used other array programming languages that copy data more eagerly. As NumPy has been designed to be able to work with very large arrays, you could imagine performance and memory problems if NumPy insisted on always copying data.

CAUTION

If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array—for example, `arr[5:8].copy()`.

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

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

In [74]: arr2d[2]
Out[74]: 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 [75]: arr2d[0][2]
Out[75]: 3

In [76]: arr2d[0, 2]
Out[76]: 3
```

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

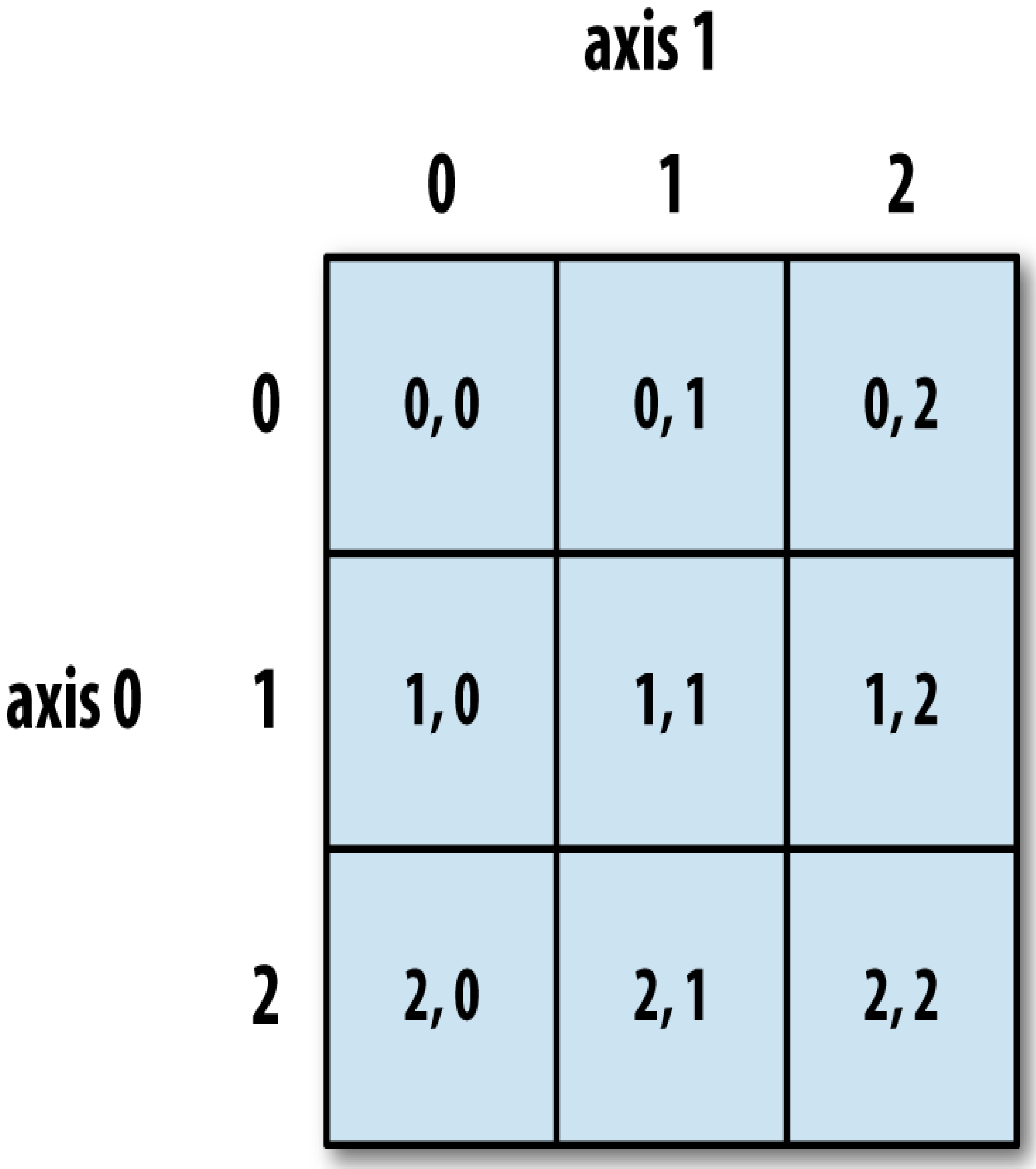


Figure 4-1. Indexing elements in a NumPy array

In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions. So in the 2 × 2 × 3 array `arr3d`:

```
In [77]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9],
[10, 11, 12]]])
```

```
In [78]: arr3d
Out[78]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

arr3d[0] is a 2 × 3 array:

```
In [79]: arr3d[0]
Out[79]:
array([[1, 2, 3],
       [4, 5, 6]])
```

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

```
In [80]: old_values = arr3d[0].copy()

In [81]: arr3d[0] = 42

In [82]: arr3d
Out[82]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [83]: arr3d[0] = old_values

In [84]: arr3d
Out[84]:
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 [85]: arr3d[1, 0]
Out[85]: array([7, 8, 9])
```

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

```
In [86]: x = arr3d[1]

In [87]: x
Out[87]:
array([[ 7,  8,  9],
       [10, 11, 12]])

In [88]: x[0]
Out[88]: array([7, 8, 9])
```

Note that in all of these cases where subsections of the array have been selected, the returned arrays are views.

Indexing with slices

Like one-dimensional objects such as Python lists, ndarrays can be sliced with the familiar syntax:

```
In [89]: arr
Out[89]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])

In [90]: arr[1:6]
Out[90]: array([ 1,  2,  3,  4, 64])
```

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

```
In [91]: arr2d
Out[91]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [92]: arr2d[:2]
Out[92]:
array([[1, 2, 3],
       [4, 5, 6]])
```

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. It can be helpful to read the expression `arr2d[:2]` as “select the first two rows of `arr2d`.”

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

```
In [93]: arr2d[:2, 1:]
Out[93]:
array([[2, 3],
       [5, 6]])
```

When slicing like this, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower dimensional slices.

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

```
In [94]: arr2d[1, :2]
Out[94]: array([4, 5])
```

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

```
In [95]: arr2d[:2, 2]
Out[95]: array([3, 6])
```

See Figure 4-2 for an illustration. Note that a colon by itself means to take the entire axis, so you can slice only higher dimensional axes by doing:

```
In [96]: arr2d[:, :1]
Out[96]:
array([[1],
       [4],
       [7]])
```

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

```
In [97]: arr2d[:2, 1:] = 0

In [98]: arr2d
Out[98]:
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

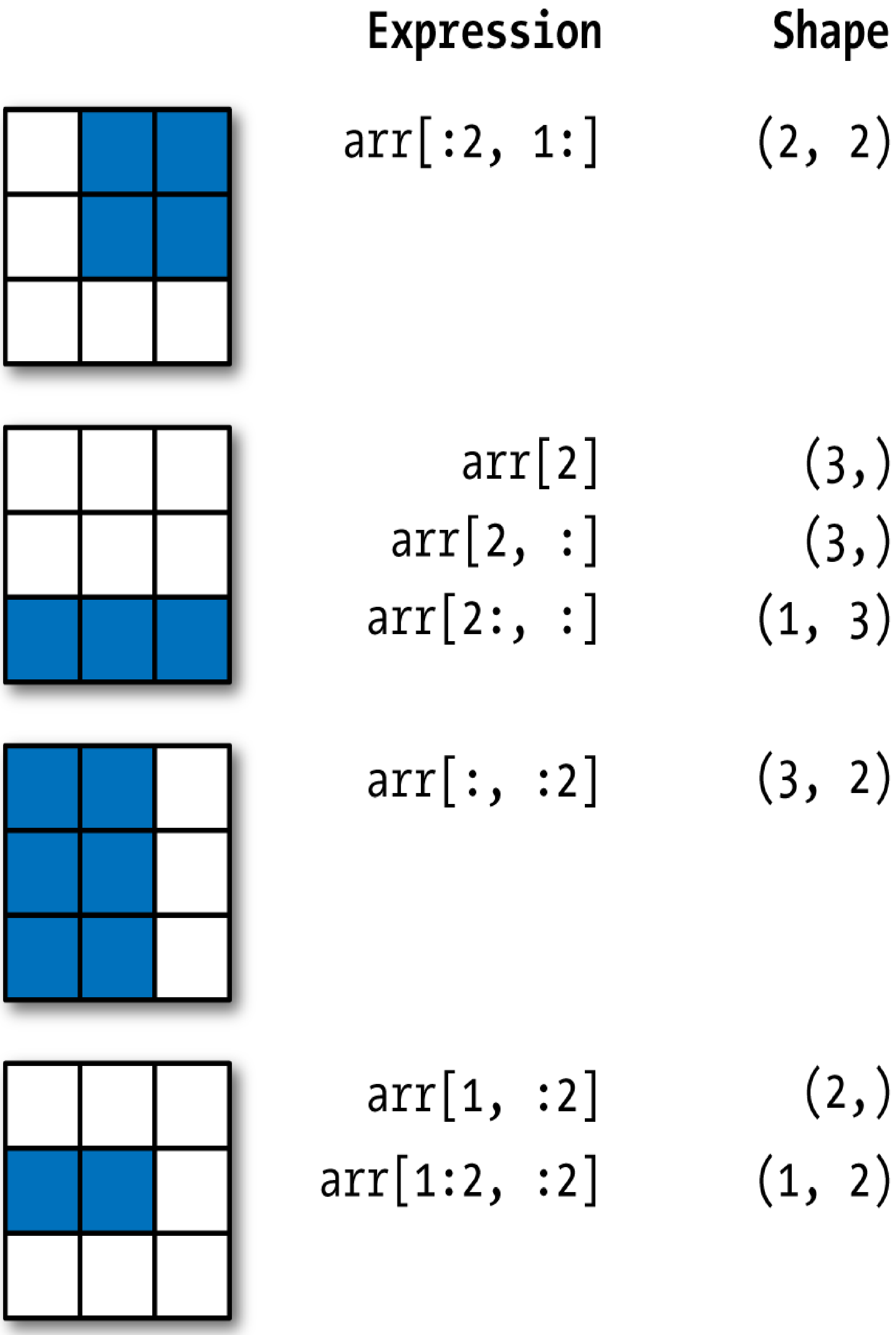

Expression
Shape
arr[:2, 1:]
(2, 2)
arr[2]
(3,)
arr[2, :]
(3,)
arr[2:, :]
(1, 3)
arr[:, :2]
(3, 2)
arr[1, :2]
(2,)
arr[1:2, :2]
(1, 2)

Figure 4-2. Two-dimensional array slicing

Boolean Indexing

Let’s consider an example where we have some data in an array and an array of names with duplicates. I’m going to use here the `randn` function in `numpy.random` to generate some random normally distributed data:

```
In [99]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will',
'Joe', 'Joe'])

In [100]: data = np.random.randn(7, 4)

In [101]: names
Out[101]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe',
'Joe'], dtype='<U4')

In [102]: data
Out[102]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Suppose each name corresponds to a row in the `data` array and we wanted to select all the rows with corresponding name `'Bob'`. Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized. Thus, comparing `names` with the string `'Bob'` yields a boolean array:

```
In [103]: names == 'Bob'
Out[103]: array([ True, False, False,  True, False, False,
False])
```

This boolean array can be passed when indexing the array:

```
In [104]: data[names == 'Bob']
Out[104]:
```

```
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

The boolean array must be of the same length as the array axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers; more on this later).

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

```
In [105]: data[names == 'Bob', 2:]
Out[105]:
array([[ 0.769 ,  1.2464],
       [-0.5397,  0.477 ]])

In [106]: data[names == 'Bob', 3]
Out[106]: array([1.2464, 0.477 ])
```

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

```
In [107]: names != 'Bob'
Out[107]: array([False,  True,  True, False,  True,  True,
True])

In [108]: ~(names == 'Bob')
Out[108]: array([False,  True,  True, False,  True,  True,
True])

In [109]: data[~(names == 'Bob')]
Out[109]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

The ~ operator can be useful when you want to invert a boolean array referenced by a variable:

```
In [110]: cond = names == 'Bob'
In [111]: data[~cond]
Out[111]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like `&` (and) and `|` (or):

```
In [112]: mask = (names == 'Bob') | (names == 'Will')

In [113]: mask
Out[113]: array([ True, False,  True,  True,  True, False,
False])

In [114]: data[mask]
Out[114]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241]])
```

Selecting data from an array by boolean indexing *always* creates a copy of the data, even if the returned array is unchanged.

CAUTION

The Python keywords `and` and `or` do not work with boolean arrays. Use `&` (and) and `|` (or) instead.

Setting values with boolean arrays works by substituting the value or values on the right hand side into the locations where the boolean array's values are `True`. To set all of the negative values in `data` to 0 we need only do:

```
In [115]: data[data < 0] = 0

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

```
array([[0.0929, 0.2817, 0.769 , 1.2464],
       [1.0072, 0.    , 0.275 , 0.2289],
       [1.3529, 0.8864, 0.    , 0.    ],
       [1.669 , 0.    , 0.    , 0.477 ],
       [3.2489, 0.    , 0.    , 0.1241],
       [0.3026, 0.5238, 0.0009, 1.3438],
       [0.    , 0.    , 0.    , 0.    ]])
```

Setting whole rows or columns using a one-dimensional boolean array is also easy:

```
In [117]: data[names != 'Joe'] = 7

In [118]: data
Out[118]:
array([[7.    , 7.    , 7.    , 7.    ],
       [1.0072, 0.    , 0.275 , 0.2289],
       [7.    , 7.    , 7.    , 7.    ],
       [7.    , 7.    , 7.    , 7.    ],
       [7.    , 7.    , 7.    , 7.    ],
       [0.3026, 0.5238, 0.0009, 1.3438],
       [0.    , 0.    , 0.    , 0.    ]])
```

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

Fancy Indexing

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

```
In [119]: arr = np.empty((8, 4))

In [120]: for i in range(8):
   .....:     arr[i] = i

In [121]: arr
Out[121]:
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 [122]: arr[[4, 3, 0, 6]]
Out[122]:
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
```

Hopefully this code did what you expected! Using negative indices selects rows from the end:

```
In [123]: arr[[-3, -5, -7]]
Out[123]:
array([[5., 5., 5., 5.],
       [3., 3., 3., 3.],
       [1., 1., 1., 1.]])
```

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

```
In [124]: arr = np.arange(32).reshape((8, 4))

In [125]: arr
Out[125]:
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 [126]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[126]: array([ 4, 23, 29, 10])
```

We’ll look at the `reshape` method in more detail in Appendix A.

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

The behavior of fancy indexing in this case is a bit different from what some users might have expected (myself included), which is the rectangular region formed by selecting a subset of the matrix’s rows and columns. Here is one way to get that:

```
In [127]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[127]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array.

Transposing Arrays and Swapping Axes

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

```
In [128]: arr = np.arange(15).reshape((3, 5))

In [129]: arr
Out[129]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [130]: arr.T
Out[130]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

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

```
In [131]: arr = np.random.randn(6, 3)

In [132]: arr
Out[132]: 
array([[-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329],
       [-2.3594, -0.1995, -1.542 ],
       [-0.9707, -1.307 ,  0.2863],
       [ 0.378 , -0.7539,  0.3313],
       [ 1.3497,  0.0699,  0.2467]])

In [133]: np.dot(arr.T, arr)
Out[133]: 
array([[ 9.2291,  0.9394,  4.948 ],
       [ 0.9394,  3.7662, -1.3622],
       [ 4.948 , -1.3622,  4.3437]])
```

The `@` infix operator is another way to do matrix multiplication:

```
In [134]: arr.T @ arr
Out[134]: 
array([[ 9.2291,  0.9394,  4.948 ],
       [ 0.9394,  3.7662, -1.3622],
       [ 4.948 , -1.3622,  4.3437]])
```

For higher dimensional arrays, `transpose` will accept a tuple of axis numbers to permute the axes:

```
In [135]: arr = np.arange(16).reshape((2, 2, 4))

In [136]: arr
Out[136]: 
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
```

```
In [137]: arr.transpose((1, 0, 2))
Out[137]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

Here, the axes have been reordered with the second axis first, the first axis second, and the last axis unchanged. While it can be difficult to visualize a multidimensional transposition, it is a “reorientation” of the array which does not result in any data being copied or moved around.

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

```
In [138]: arr
Out[138]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [139]: arr.swapaxes(1, 2)
Out[139]:
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],
       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

`swapaxes` similarly returns a view on the data without making a copy.

4.2 Universal Functions: Fast Element-Wise Array Functions

A universal function, or *ufunc*, is a function that performs element-wise operations on data in ndarrays. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results.

Many ufuncs are simple element-wise transformations, like `sqrt` or `exp`:

```
In [140]: arr = np.arange(10)

In [141]: arr
Out[141]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [142]: np.sqrt(arr)
Out[142]:
array([0.    , 1.    , 1.4142, 1.7321, 2.    , 2.2361, 2.4495,
2.6458,
       2.8284, 3.    ])

In [143]: np.exp(arr)
Out[143]:
array([   1.    ,    2.7183,    7.3891,   20.0855,   54.5982,
148.4132,
        403.4288, 1096.6332, 2980.958 , 8103.0839])
```

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

```
In [144]: x = np.random.randn(8)

In [145]: y = np.random.randn(8)

In [146]: x
Out[146]:
array([-0.0119,  1.0048,  1.3272, -0.9193, -1.5491,  0.0222,
0.7584,
       -0.6605])

In [147]: y
Out[147]:
array([ 0.8626, -0.01  ,  0.05  ,  0.6702,  0.853 , -0.9559,
-0.0235,
       -2.3042])

In [148]: np.maximum(x, y)
```

```
Out[148]:
array([ 0.8626,  1.0048,  1.3272,  0.6702,  0.853 ,  0.0222,
0.7584,
       -0.6605])
```

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

While not common, a ufunc can return multiple arrays. `modf` is one example, a vectorized version of the built-in Python `divmod`; it returns the fractional and integral parts of a floating-point array:

```
In [149]: arr = np.random.randn(7) * 5

In [150]: arr
Out[150]: array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,
3.45  ,  5.0077])

In [151]: remainder, whole_part = np.modf(arr)

In [152]: remainder
Out[152]: array([-0.2623, -0.0915, -0.663 ,  0.3731,  0.6182,
0.45  ,  0.0077])

In [153]: whole_part
Out[153]: array([-3., -6., -6.,  5.,  3.,  3.,  5.])
```

Ufuncs accept an optional `out` argument that allows them to assign their results into an existing array rather than creating a new one:

```
In [154]: arr
Out[154]: array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,
3.45  ,  5.0077])

In [155]: out = np.zeros_like(arr)

In [156]: np.add(arr, 1)
Out[156]: array([-2.2623, -5.0915, -5.663 ,  6.3731,  4.6182,
4.45  ,  6.0077])

In [157]: np.add(arr, 1, out=out)
Out[157]: array([-2.2623, -5.0915, -5.663 ,  6.3731,  4.6182,
4.45  ,  6.0077])
```

```
In [158]: out
Out[158]: array([-2.2623, -5.0915, -5.663 ,  6.3731,  4.6182,
4.45  ,  6.0077])
```

See Tables 4-3 and 4-4 for a listing of some of NumPy’s ufuncs. New ufuncs continue to be added to NumPy, so consulting the online NumPy documentation is the best way to get a comprehensive listing.

Table 4-3. Some unary ufuncs

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating-point, or complex values
`sqrt`	Compute the square root of each element (equivalent to `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 that number)
`floor`	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as a separate array
`isnan`	Return boolean array indicating whether each value is `NaN` (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-`NaN`) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions

Function	Description
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	Inverse trigonometric functions
logical_not	Compute truth value of not x element-wise (equivalent to ~arr).

Table 4-4. Some binary universal functions

Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum; `fmax` ignores `NaN`
`minimum, fmin`	Element-wise minimum; `fmin` ignores `NaN`
`mod`	Element-wise 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 `>, >=, <, <=, ==, !=`)
`logical_and`	Compute element-wise truth value of AND (`&`) logical operation.
`logical_or`	Compute element-wise truth value of OR (`|`) logical operation.
`logical_xor`	Compute element-wise truth value of XOR (`^`) logical operation.

4.3 Array-Oriented Programming with Arrays

Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops. This practice of replacing explicit loops with array expressions is referred to by some people *vectorization*. In general, vectorized array operations will usually be significantly faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations. Later, in Appendix A, I explain *broadcasting*, a powerful method for vectorizing computations.

As a simple example, suppose we wished to evaluate the function `sqrt(x^2 + y^2)` across a regular grid of values. The `np.meshgrid` function takes two 1D arrays and produces two 2D matrices corresponding to all pairs of `(x, y)` in the two arrays:

```
In [159]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced
points

In [160]: xs, ys = np.meshgrid(points, points)

In [161]: ys
Out[161]:
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

Now, evaluating the function is a matter of writing the same expression you would write with two points:

```
In [162]: z = np.sqrt(xs ** 2 + ys ** 2)

In [163]: z
Out[163]:
array([[7.0711, 7.064 , 7.0569, ..., 7.0499, 7.0569, 7.064 ],
       [7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569],
       [7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
       ...,
       [7.0499, 7.0428, 7.0357, ..., 7.0286, 7.0357, 7.0428],
       [7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
       [7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569]])
```

As a preview of [Link to Come], I use matplotlib to create visualizations of this two-dimensional array:

```
In [164]: import matplotlib.pyplot as plt

In [165]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[165]: <matplotlib.colorbar.Colorbar at 0x7fb1cbbfb880>

In [166]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid
of values")
Out[166]: Text(0.5, 1.0, 'Image plot of $\\sqrt{x^2 + y^2}$ for a
grid of values'
)
```

See Figure 4-3. Here I used the matplotlib function `imshow` to create an image plot from a two-dimensional array of function values.

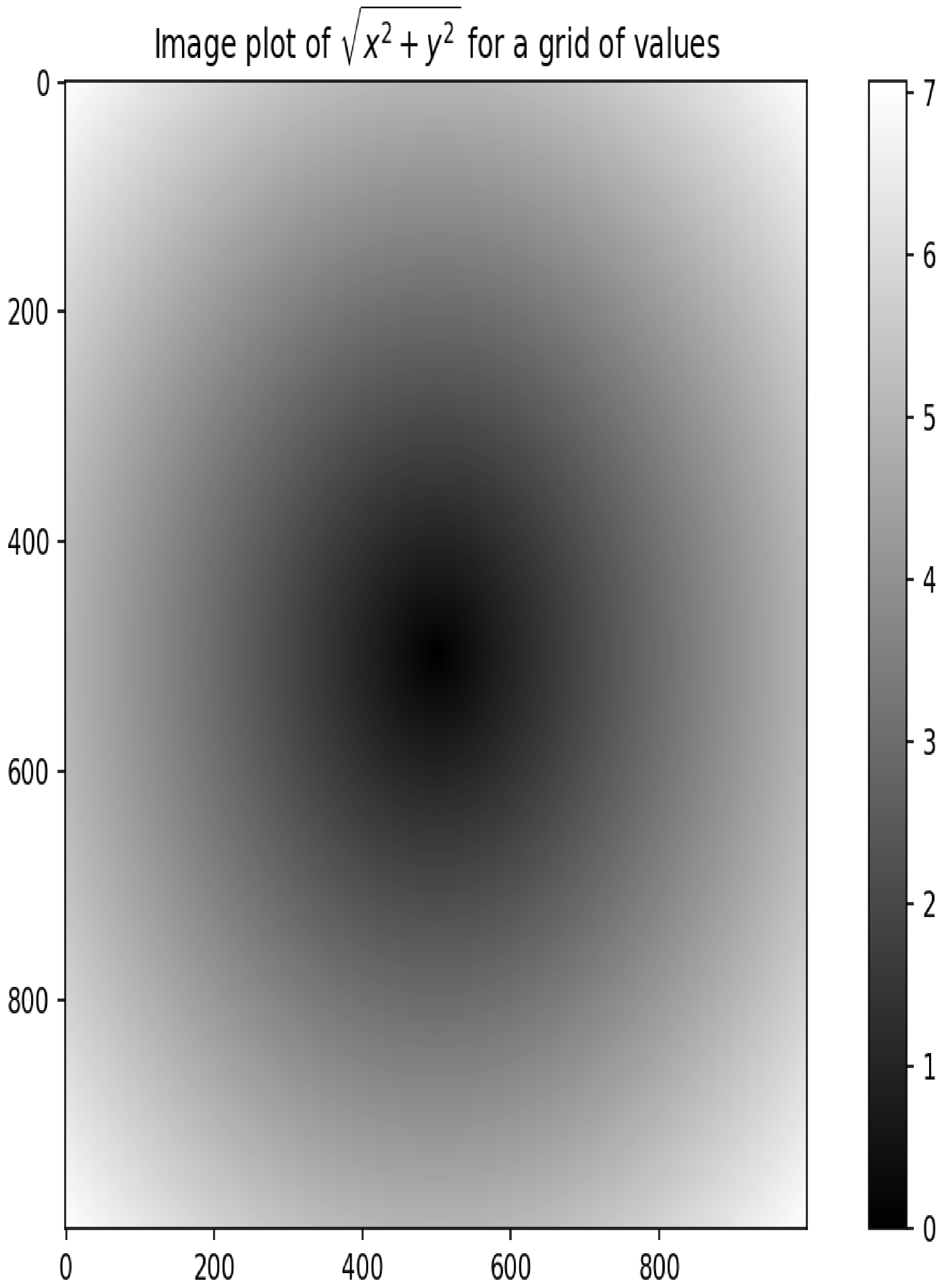


Figure 4-3. Plot of function evaluated on grid

NOTE

The term *vectorization* is used to describe some other computer science concepts, but in this book I use it to describe operations on whole arrays of data at once rather than going value-by-value using a Python for loop.

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 [169]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [170]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [171]: cond = np.array([True, False, True, True, False])
```

Suppose we wanted to take a value from xarr whenever the corresponding value in cond is True, and otherwise take the value from yarr. A list comprehension doing this might look like:

```
In [172]: result = [(x if c else y)
   .....:            for x, y, c in zip(xarr, yarr, cond)]

In [173]: result
Out[173]: [1.1, 2.2, 1.3, 1.4, 2.5]
```

This has multiple problems. First, it will not be very fast for large arrays (because all the work is being done in interpreted Python code). Second, it will not work with multidimensional arrays. With np.where you can do this with a single function call:

```
In [174]: result = np.where(cond, xarr, yarr)

In [175]: result
Out[175]: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

The second and third arguments to `np.where` don't need to be arrays; one or both of them can be scalars. A typical use of `where` in data analysis is to produce a new array of values based on another array. Suppose you had a matrix of randomly generated data and you wanted to replace all positive values with 2 and all negative values with –2. This is very easy to do with `np.where`:

```
In [176]: arr = np.random.randn(4, 4)

In [177]: arr
Out[177]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 0.2229,  0.0513, -1.1577,  0.8167],
       [ 0.4336,  1.0107,  1.8249, -0.9975],
       [ 0.8506, -0.1316,  0.9124,  0.1882]])

In [178]: arr > 0
Out[178]:
array([[False, False, False, False],
       [ True,  True, False,  True],
       [ True,  True,  True, False],
       [ True, False,  True,  True]])

In [179]: np.where(arr > 0, 2, -2)
Out[179]:
array([[-2, -2, -2, -2],
       [ 2,  2, -2,  2],
       [ 2,  2,  2, -2],
       [ 2, -2,  2,  2]])
```

You can combine scalars and arrays when using `np.where`. For example, I can replace all positive values in `arr` with the constant 2 like so:

```
In [180]: np.where(arr > 0, 2, arr) # set only positive values to
2
Out[180]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 2.    ,  2.    , -1.1577,  2.    ],
       [ 2.    ,  2.    ,  2.    , -0.9975],
       [ 2.    , -0.1316,  2.    ,  2.    ]])
```

The arrays passed to `np.where` can be more than just equal-sized arrays or scalars.

Mathematical and Statistical Methods

A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class. You can use aggregations (sometimes called *reductions*) like `sum`, `mean`, and `std` (standard deviation) either by calling the array instance method or using the top-level NumPy function.

Here I generate some normally distributed random data and compute some aggregate statistics:

```
In [181]: arr = np.random.randn(5, 4)

In [182]: arr
Out[182]:
array([[ 2.1695, -0.1149,  2.0037,  0.0296],
       [ 0.7953,  0.1181, -0.7485,  0.585 ],
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363,  1.0954],
       [ 0.9809, -0.5895,  1.5817, -0.5287]])

In [183]: arr.mean()
Out[183]: 0.19607051119998253

In [184]: np.mean(arr)
Out[184]: 0.19607051119998253

In [185]: arr.sum()
Out[185]: 3.9214102239996507
```

Functions like `mean` and `sum` take an optional `axis` argument that computes the statistic over the given axis, resulting in an array with one less dimension:

```
In [186]: arr.mean(axis=1)
Out[186]: array([ 1.022 ,  0.1875, -0.502 , -0.0881,  0.3611])
```

```
In [187]: arr.sum(axis=0)
Out[187]: array([ 3.1693, -2.6345,  2.2381,  1.1486])
```

Here, `arr.mean(1)` (which is the same as `arr.mean(axis=1)`) means “compute mean across the columns” where `arr.sum(0)` means “compute sum down the rows.”

Other methods like `cumsum` and `cumprod` do not aggregate, instead producing an array of the intermediate results:

```
In [188]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [189]: arr.cumsum()
Out[189]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

In multidimensional arrays, accumulation functions like `cumsum` return an array of the same size, but with the partial aggregates computed along the indicated axis according to each lower dimensional slice:

```
In [190]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [191]: arr
Out[191]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [192]: arr.cumsum(axis=0)
Out[192]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])

In [193]: arr.cumprod(axis=1)
Out[193]:
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]])
```

See Table 4-5 for a full listing. We’ll see many examples of these methods in action in later chapters.

Table 4-5. Basic array statistical methods

Method	Description
`sum`	Sum of all the elements in the array or along an axis; zero-length arrays have sum 0
`mean`	Arithmetic mean; invalid (returns `NaN`) on zero-length arrays
`std, var`	Standard deviation and variance, respectively
`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 preceding methods. Thus, `sum` is often used as a means of counting `True` values in a boolean array:

```
In [194]: arr = np.random.randn(100)

In [195]: (arr > 0).sum() # Number of positive values
Out[195]: 42

In [196]: (arr <= 0).sum() # Number of non-positive values
Out[196]: 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 [197]: bools = np.array([False, False, True, False])

In [198]: bools.any()
Out[198]: True
```

```
In [199]: bools.all()
Out[199]: False
```

These methods also work with non-boolean arrays, where non-zero elements are treated as `True`.

Sorting

Like Python's built-in list type, NumPy arrays can be sorted in-place with the `sort` method:

```
In [200]: arr = np.random.randn(6)

In [201]: arr
Out[201]: array([ 0.6095, -0.4938,  1.24  , -0.1357,  1.43  ,
-0.8469])

In [202]: arr.sort()

In [203]: arr
Out[203]: array([-0.8469, -0.4938, -0.1357,  0.6095,  1.24  ,
1.43  ])
```

You can sort each one-dimensional section of values in a multidimensional array in-place along an axis by passing the axis number to `sort`:

```
In [204]: arr = np.random.randn(5, 3)

In [205]: arr
Out[205]:
array([[ 0.6033,  1.2636, -0.2555],
       [-0.4457,  0.4684, -0.9616],
       [-1.8245,  0.6254,  1.0229],
       [ 1.1074,  0.0909, -0.3501],
       [ 0.218 , -0.8948, -1.7415]])

In [206]: arr.sort(1)

In [207]: arr
Out[207]:
array([[-0.2555,  0.6033,  1.2636],
       [-0.9616, -0.4457,  0.4684],
       [-1.8245,  0.6254,  1.0229],
       [-0.3501,  0.0909,  1.1074],
       [-1.7415, -0.8948,  0.218 ]])
```

The top-level method `np.sort` returns a sorted copy of an array instead of modifying the array in-place. For example:

```
In [208]: arr2 = np.array([5, -10, 7, 1, 0, -3])

In [209]: sorted_arr2 = np.sort(arr2)

In [210]: sorted_arr2
Out[210]: array([-10,  -3,   0,   1,   5,   7])
```

For more details on using NumPy's sorting methods, and more advanced techniques like indirect sorts, see Appendix A. Several other kinds of data manipulations related to sorting (e.g., sorting a table of data by one or more columns) can also be found in pandas.

Unique and Other Set Logic

NumPy has some basic set operations for one-dimensional ndarrays. A commonly used one is `np.unique`, which returns the sorted unique values in an array:

```
In [211]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will',
'Joe', 'Joe'])

In [212]: np.unique(names)
Out[212]: array(['Bob', 'Joe', 'Will'], dtype='<U4')

In [213]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [214]: np.unique(ints)
Out[214]: array([1, 2, 3, 4])
```

Contrast `np.unique` with the pure Python alternative:

```
In [215]: sorted(set(names))
Out[215]: ['Bob', 'Joe', 'Will']
```

Another function, `np.in1d`, tests membership of the values in one array in another, returning a boolean array:

```
In [216]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [217]: np.in1d(values, [2, 3, 6])
Out[217]: array([ True, False, False,  True,  True, False,
True])
```

See Table 4-6 for a listing of set functions in NumPy.

Table 4-6. Array set operations

Method	Description
`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

4.4 File Input and Output with Arrays

NumPy is able to save and load data to and from disk in some text or binary formats. In this section I only discuss NumPy's built-in binary format, since most users will prefer pandas and other tools for loading text or tabular data (see Chapter 6 for much more).

`np.save` and `np.load` are the two workhorse functions for efficiently saving and loading array data on disk. Arrays are saved by default in an uncompressed raw binary format with file extension *.npy*:

```
In [218]: arr = np.arange(10)
In [219]: np.save('some_array', arr)
```

If the file path does not already end in *.npy*, the extension will be appended. The array on disk can then be loaded with `np.load`:

```
In [220]: np.load('some_array.npy')
Out[220]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

You save multiple arrays in an uncompressed archive using `np.savez` and passing the arrays as keyword arguments:

```
In [221]: np.savez('array_archive.npz', a=arr, b=arr)
```

When loading an *.npz* file, you get back a dict-like object that loads the individual arrays lazily:

```
In [222]: arch = np.load('array_archive.npz')

In [223]: arch['b']
Out[223]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

If your data compresses well, you may wish to use `numpy.savez_compressed` instead:

```
In [224]: np.savez_compressed('arrays_compressed.npz', a=arr,
b=arr)
```

4.5 Linear Algebra

Linear algebra operations, like matrix multiplication, decompositions, determinants, and other square matrix math, are an important part of many array libraries. Unlike some languages like MATLAB, multiplying two two-dimensional arrays with `*` is an element-wise product instead of a matrix dot product. Thus, there is a function `dot`, both an array method and a function in the `numpy` namespace, for matrix multiplication:

```
In [228]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [229]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [230]: x
Out[230]:
array([[1., 2., 3.],
       [4., 5., 6.]])

In [231]: y
Out[231]:
array([[ 6., 23.],
       [-1.,  7.],
       [ 8.,  9.]])

In [232]: x.dot(y)
Out[232]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

`x.dot(y)` is equivalent to `np.dot(x, y)`:

```
In [233]: np.dot(x, y)
Out[233]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

A matrix product between a two-dimensional array and a suitably sized one-dimensional array results in a one-dimensional array:

```
In [234]: np.dot(x, np.ones(3))
Out[234]: array([ 6., 15.])
```

The `@` symbol (as of Python 3.5) also works as an infix operator that performs matrix multiplication:

```
In [235]: x @ np.ones(3)
Out[235]: array([ 6., 15.])
```

`numpy.linalg` has a standard set of matrix decompositions and things like inverse and determinant. These are implemented under the hood via the same battle-tested optimized linear algebra libraries used in other languages like MATLAB and R, such as BLAS, LAPACK, or possibly (depending on your NumPy build) the proprietary Intel MKL (Math Kernel Library):

```
In [236]: from numpy.linalg import inv, qr

In [237]: X = np.random.randn(5, 5)

In [238]: mat = X.T.dot(X)

In [239]: inv(mat)
Out[239]: 
array([[ 5.4892, -1.8761, -6.2379, -0.8607, -2.075 ],
       [-1.8761,  2.0624,  3.5714,  0.4462,  1.181 ],
       [-6.2379,  3.5714,  9.6618,  1.2522,  2.2175],
       [-0.8607,  0.4462,  1.2522,  0.2479,  0.2776],
       [-2.075 ,  1.181 ,  2.2175,  0.2776,  1.6894]])

In [240]: mat.dot(inv(mat))
Out[240]: 
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 [241]: q, r = qr(mat)

In [242]: r
Out[242]: 
array([[-4.0096,  5.1162, -2.9918, -4.5474, -4.0954],
       [ 0.    , -1.2957,  1.123 , -5.4974,  0.3109],
       [ 0.    ,  0.    , -1.3738, 10.0454,  0.1552],
       [ 0.    ,  0.    ,  0.    , -1.7988,  0.7472],
       [ 0.    ,  0.    ,  0.    ,  0.    ,  0.2717]])
```

The expression `X.T.dot(X)` computes the dot product of `X` with its transpose `X.T`.

See Table 4-7 for a list of some of the most commonly used linear algebra functions.

Table 4-7. Commonly used numpy.linalg functions

Function	Description
`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
`det`	Compute the matrix determinant
`eig`	Compute the eigenvalues and eigenvectors of a square matrix
`inv`	Compute the inverse of a square matrix
`pinv`	Compute the Moore-Penrose pseudo-inverse of a matrix
`qr`	Compute the QR decomposition
`svd`	Compute the singular value decomposition (SVD)
`solve`	Solve the linear system Ax = b for x, where A is a square matrix
`lstsq`	Compute the least-squares solution to `Ax = b`

4.6 Pseudorandom Number Generation

The `numpy.random` module supplements the built-in Python `random` with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions. For example, you can get a 4 × 4 array of samples from the standard normal distribution using `normal`:

```
In [243]: samples = np.random.normal(size=(4, 4))

In [244]: samples
Out[244]:
array([[-0.5816, -1.2604,  0.4646, -1.0702],
       [ 0.8042, -0.1567,  2.0104, -0.8871],
       [-0.9779, -0.2672,  0.4833, -0.4003],
       [ 0.4499,  0.3996, -0.1516, -2.5579]])
```

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 [245]: from random import normalvariate

In [246]: N = 1000000

In [247]: %timeit samples = [normalvariate(0, 1) for _ in
range(N)]
1.07 s +- 14.5 ms per loop (mean +- std. dev. of 7 runs, 1 loop
each)

In [248]: %timeit np.random.normal(size=N)
24.1 ms +- 1.64 ms per loop (mean +- std. dev. of 7 runs, 10
loops each)
```

We say that these are *pseudorandom* numbers because they are generated by an algorithm with deterministic behavior based on the *seed* of the random number generator. You can change NumPy's random number generation seed using `np.random.seed`:

```
In [249]: np.random.seed(1234)
```

The data generation functions in `numpy.random` use a global random seed. To avoid global state, you can use `numpy.random.RandomState` to create a random number generator isolated from others:

```
In [250]: rng = np.random.RandomState(1234)

In [251]: rng.randn(10)
Out[251]:
array([ 0.4714, -1.191 ,  1.4327, -0.3127, -0.7206,  0.8872,
0.8596,
       -0.6365,  0.0157, -2.2427])
```

See Table 4-8 for a partial list of functions available in `numpy.random`. I'll give some examples of leveraging these functions' ability to generate large arrays of samples all at once in the next section.

Table 4-8. Partial list of numpy.random functions

Function	Description
`seed`	Seed the random number generator
`permutation`	Return a random permutation of a sequence, or return a permuted range
`shuffle`	Randomly permute a sequence in-place
`rand`	Draw samples from a uniform distribution
`randint`	Draw random integers from a given low-to-high range
`randn`	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
`binomial`	Draw samples from a binomial distribution
`normal`	Draw samples from a normal (Gaussian) distribution
`beta`	Draw samples from a beta distribution
`chisquare`	Draw samples from a chi-square distribution
`gamma`	Draw samples from a gamma distribution
`uniform`	Draw samples from a uniform [0, 1) distribution

4.7 Example: Random Walks

The simulation of random walks provides an illustrative application of utilizing array operations. Let's first consider a simple random walk starting at 0 with steps of 1 and –1 occurring with equal probability.

Here is a pure Python way to implement a single random walk with 1,000 steps using the built-in `random` module:

```
In [252]: import random
   .....: position = 0
   .....: walk = [position]
   .....: steps = 1000
   .....: for i in range(steps):
   .....:     step = 1 if random.randint(0, 1) else -1
   .....:     position += step
   .....:     walk.append(position)
   .....:
```

See Figure 4-4 for an example plot of the first 100 values on one of these random walks:

```
In [254]: plt.plot(walk[:100])
```

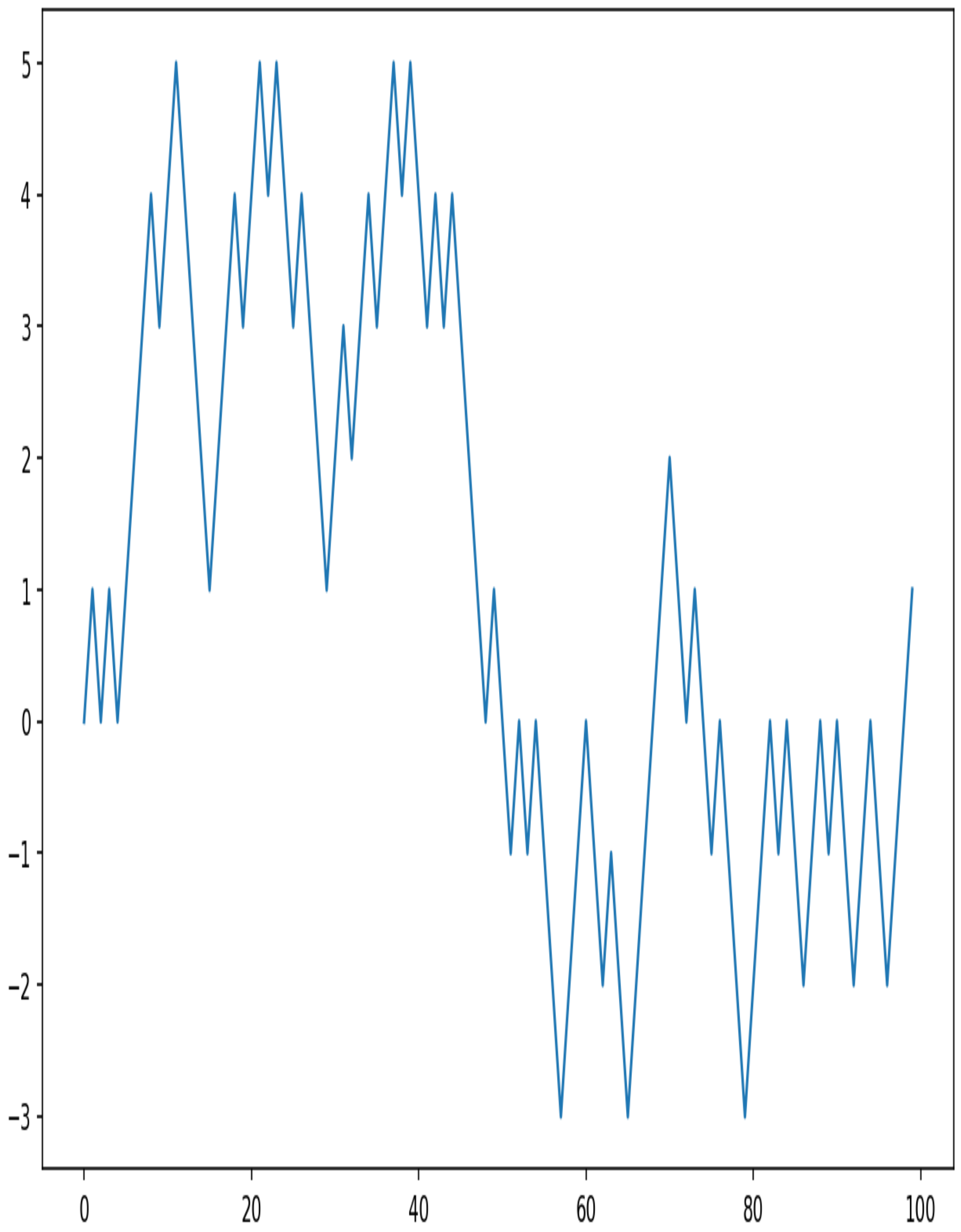


Figure 4-4. A simple random walk

You might make the observation that `walk` is the cumulative sum of the random steps and could be evaluated as an array expression. Thus, I use the np.random module to draw 1,000 coin flips at once, set these to 1 and –1, and compute the cumulative sum:

```
In [256]: nsteps = 1000

In [257]: draws = np.random.randint(0, 2, size=nsteps)

In [258]: steps = np.where(draws > 0, 1, -1)

In [259]: walk = steps.cumsum()
```

From this we can begin to extract statistics like the minimum and maximum value along the walk's trajectory:

```
In [260]: walk.min()
Out[260]: -3

In [261]: walk.max()
Out[261]: 31
```

A more complicated statistic is the *first crossing time*, the step at which the random walk reaches a particular value. Here we might want to know how long it took the random walk to get at least 10 steps away from the origin 0 in either direction. `np.abs(walk) >= 10` gives us a boolean array indicating where the walk has reached or exceeded 10, but we want the index of the *first* 10 or –10. Turns out, we can compute this using `argmax`, which returns the first index of the maximum value in the boolean array (`True` is the maximum value):

```
In [262]: (np.abs(walk) >= 10).argmax()
Out[262]: 37
```

Note that using `argmax` here is not always efficient because it always makes a full scan of the array. In this special case, once a `True` is observed we know it to be the maximum value.

Simulating Many Random Walks at Once

If your goal was to simulate many random walks, say 5,000 of them, you can generate all of the random walks with minor modifications to the preceding code. If passed a 2-tuple, the `numpy.random` functions will generate a two-dimensional array of draws, and we can compute the cumulative sum for each row to compute all 5,000 random walks in one shot:

```
In [263]: nwalks = 5000

In [264]: nsteps = 1000

In [265]: draws = np.random.randint(0, 2, size=(nwalks, nsteps))
# 0 or 1

In [266]: steps = np.where(draws > 0, 1, -1)

In [267]: walks = steps.cumsum(1)

In [268]: walks
Out[268]:
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 [269]: walks.max()
Out[269]: 138

In [270]: walks.min()
Out[270]: -133
```

Out of these walks, let’s compute the minimum crossing time to 30 or –30. This is slightly tricky because not all 5,000 of them reach 30. We can check this using the `any` method:

```
In [271]: hits30 = (np.abs(walks) >= 30).any(1)

In [272]: hits30
Out[272]: array([False,  True, False, ..., False,  True, False])

In [273]: hits30.sum() # Number that hit 30 or -30
Out[273]: 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 [274]: crossing_times = (np.abs(walks[hits30]) >=
30).argmax(1)

In [275]: crossing_times
Out[275]: array([735, 409, 253, ..., 327, 453, 447])
```

Lastly, we compute the average minimum crossing time:

```
In [276]: crossing_times.mean()
Out[276]: 498.8897360703812
```

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 [277]: steps = np.random.normal(loc=0, scale=0.25,
   .....:                          size=(nwalks, nsteps))
```

NOTE

Keep in mind that this vectorized approach requires creating an array with `nwalks * nsteps` elements, which may use a large amount of memory for large simulations. If memory is more constrained, then a different approach will be required.

4.8 Conclusion

While much of the rest of the book will focus on building data wrangling skills with pandas, we will continue to work in a similar array-based style. In Appendix A, we will dig deeper into NumPy features to help you further develop your array computing skills.

Chapter 5. Getting Started with pandas

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at arufino@oreilly.com.

pandas will be a major tool of interest throughout much of the rest of the book. It contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python. pandas is often used in tandem with numerical computing tools like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and data visualization libraries like matplotlib. pandas adopts significant parts of NumPy’s idiomatic style of array-based computing, especially array-based functions and a preference for data processing without `for` loops.

While pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with tabular or heterogeneous data. NumPy, by contrast, is best suited for working with homogeneously-typed numerical array data.

Since becoming an open source project in 2010, pandas has matured into a quite large library that’s applicable in a broad set of real-world use cases.

The developer community has grown to over 2500 distinct contributors, who've been helping build the project as they've used it to solve their day-to-day data problems. pandas's vibrant developer and user community have been a key part of its success.

> **NOTE**
>
> Many people don't know that I haven't been actively involved in day-to-day pandas development since 2013; it has been an entirely community-managed project since then. Be sure to pass on your thanks to the core development and all the contributors for their hard work!

Throughout the rest of the book, I use the following import convention for pandas:

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

Thus, whenever you see `pd.` in code, it's referring to pandas. You may also find it easier to import Series and DataFrame into the local namespace since they are so frequently used:

```
In [2]: from pandas import Series, DataFrame
```

5.1 Introduction to pandas Data Structures

To get started with pandas, you will need to get comfortable with its two workhorse data structures: *Series* and *DataFrame*. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for a wide variety of data applications.

Series

A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its *index*. The simplest Series is formed from only an array of data:

```
In [13]: obj = pd.Series([4, 7, -5, 3])

In [14]: obj
Out[14]:
0    4
1    7
2   -5
3    3
dtype: int64
```

The string representation of a Series displayed interactively shows the index on the left and the values on the right. Since we did not specify an index for the data, a default one consisting of the integers `0` through `N - 1` (where `N` is the length of the data) is created. You can get the array representation and index object of the Series via its `array` and `index` attributes, respectively:

```
In [15]: obj.array
Out[15]:
<PandasArray>
[4, 7, -5, 3]
Length: 4, dtype: int64

In [16]: obj.index
Out[16]: RangeIndex(start=0, stop=4, step=1)
```

Often it will be desirable to create a Series with an index identifying each data point with a label:

```
In [17]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a',
'c'])

In [18]: obj2
Out[18]:
d    4
b    7
a   -5
c    3
dtype: int64
```

```
In [19]: obj2.index
Out[19]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Compared with NumPy arrays, you can use labels in the index when selecting single values or a set of values:

```
In [20]: obj2['a']
Out[20]: -5

In [21]: obj2['d'] = 6

In [22]: obj2[['c', 'a', 'd']]
Out[22]:
c    3
a   -5
d    6
dtype: int64
```

Here `['c', 'a', 'd']` is interpreted as a list of indices, even though it contains strings instead of integers.

Using NumPy functions or NumPy-like operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

```
In [23]: obj2[obj2 > 0]
Out[23]:
d    6
b    7
c    3
dtype: int64

In [24]: obj2 * 2
Out[24]:
d    12
b    14
a   -10
c     6
dtype: int64

In [25]: import numpy as np

In [26]: np.exp(obj2)
```

```
Out[26]:
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

Another way to think about a Series is as a fixed-length, ordered dict, as it is a mapping of index values to data values. It can be used in many contexts where you might use a dict:

```
In [27]: 'b' in obj2
Out[27]: True

In [28]: 'e' in obj2
Out[28]: False
```

Should you have data contained in a Python dict, you can create a Series from it by passing the dict:

```
In [29]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000,
'Utah': 5000}

In [30]: obj3 = pd.Series(sdata)

In [31]: obj3
Out[31]:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

When you are only passing a dict, the index in the resulting Series will respect the order of the keys according to the dict’s `keys` method, which depends on the key insertion order. You can override this by passing an index with the dict keys in the order you want them to appear in the resulting Series:

```
In [32]: states = ['California', 'Ohio', 'Oregon', 'Texas']
In [33]: obj4 = pd.Series(sdata, index=states)

In [34]: obj4
Out[34]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

Here, three values found in `sdata` were placed in the appropriate locations, but since no value for `'California'` was found, it appears as `NaN` (not a number), which is considered in pandas to mark missing or *NA* values. Since `'Utah'` was not included in `states`, it is excluded from the resulting object.

I will use the terms “missing”, “NA”, or “null” interchangeably to refer to missing data. The `isna` and `notna` functions in pandas should be used to detect missing data:

```
In [35]: pd.isna(obj4)
Out[35]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool

In [36]: pd.notna(obj4)
Out[36]:
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Series also has these as instance methods:

```
In [37]: obj4.isna()
Out[37]:
California     True
Ohio          False
Oregon        False
```

```
Texas         False
dtype: bool
```

I discuss working with missing data in more detail in [Link to Come].

A useful Series feature for many applications is that it automatically aligns by index label in arithmetic operations:

```
In [38]: obj3
Out[38]:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64

In [39]: obj4
Out[39]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64

In [40]: obj3 + obj4
Out[40]:
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

Data alignment features will be addressed in more detail later. If you have experience with databases, you can think about this as being similar to a join operation.

Both the Series object itself and its index have a `name` attribute, which integrates with other areas of pandas functionality:

```
In [41]: obj4.name = 'population'

In [42]: obj4.index.name = 'state'
In [43]: obj4
Out[43]:
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

A Series's index can be altered in-place by assignment:

```
In [44]: obj
Out[44]:
0    4
1    7
2   -5
3    3
dtype: int64

In [45]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [46]: obj
Out[46]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

DataFrame

A DataFrame represents a rectangular table of data and contains an ordered, named collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series all sharing the same index.

NOTE

While a DataFrame is physically two-dimensional, you can use it to represent higher dimensional data in a tabular format using hierarchical indexing, a subject we will discuss in [Link to Come] and an ingredient in some of the more advanced data-handling features in pandas.

There are many ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada',
'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed according to the order of the keys in `data` (which depends on their insertion order in the `dict`):

```
In [48]: frame
Out[48]:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

> **NOTE**
>
> If you are using the Jupyter notebook, pandas DataFrame objects will be displayed as a more browser-friendly HTML table.

For large DataFrames, the `head` method selects only the first five rows:

```
In [49]: frame.head()
Out[49]:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
```

If you specify a sequence of columns, the DataFrame's columns will be arranged in that order:

```
In [50]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[50]:
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
5  2003  Nevada  3.2
```

If you pass a column that isn't contained in the dict, it will appear with missing values in the result:

```
In [51]: frame2 = pd.DataFrame(data, columns=['year', 'state',
'pop', 'debt'],
   ....:                       index=['one', 'two', 'three',
'four',
   ....:                              'five', 'six'])

In [52]: frame2
Out[52]:
       year   state  pop debt
one    2000    Ohio  1.5  NaN
two    2001    Ohio  1.7  NaN
three  2002    Ohio  3.6  NaN
four   2001  Nevada  2.4  NaN
five   2002  Nevada  2.9  NaN
six    2003  Nevada  3.2  NaN

In [53]: frame2.columns
Out[53]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

A column in a DataFrame can be retrieved as a Series either by dict-like notation, or by attribute:

```
In [54]: frame2['state']
Out[54]:
one        Ohio
two        Ohio
three      Ohio
```

```
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object

In [55]: frame2.year
Out[55]:
one      2000
two      2001
three    2002
four     2001
five     2002
six      2003
Name: year, dtype: int64
```

NOTE

Attribute-like access (e.g., `frame2.year`) and tab completion of column names in IPython is provided as a convenience.

`frame2[column]` works for any column name, but `frame2.column` only works when the column name is a valid Python variable name and does not conflict with any of the method names in DataFrame.

Note that the returned Series have the same index as the DataFrame, and their `name` attribute has been appropriately set.

Rows can also be retrieved by position or name with the special `loc` attribute (much more on this later):

```
In [56]: frame2.loc['three']
Out[56]:
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

Columns can be modified by assignment. For example, the empty `'debt'` column could be assigned a scalar value or an array of values:

```
In [57]: frame2['debt'] = 16.5

In [58]: frame2
Out[58]:
       year   state  pop  debt
one    2000    Ohio  1.5  16.5
two    2001    Ohio  1.7  16.5
three  2002    Ohio  3.6  16.5
four   2001  Nevada  2.4  16.5
five   2002  Nevada  2.9  16.5
six    2003  Nevada  3.2  16.5

In [59]: frame2['debt'] = np.arange(6.)

In [60]: frame2
Out[60]:
       year   state  pop  debt
one    2000    Ohio  1.5   0.0
two    2001    Ohio  1.7   1.0
three  2002    Ohio  3.6   2.0
four   2001  Nevada  2.4   3.0
five   2002  Nevada  2.9   4.0
six    2003  Nevada  3.2   5.0
```

When you are assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If you assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any index values not present:

```
In [61]: val = pd.Series([-1.2, -1.5, -1.7], index=['two',
'four', 'five'])

In [62]: frame2['debt'] = val

In [63]: frame2
Out[63]:
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4  -1.5
five   2002  Nevada  2.9  -1.7
six    2003  Nevada  3.2   NaN
```

Assigning a column that doesn’t exist will create a new column. The del keyword will delete columns like with a dict.

As an example of del, I first add a new column of boolean values where the state column equals 'Ohio':

```
In [64]: frame2['eastern'] = frame2.state == 'Ohio'

In [65]: frame2
Out[65]:
       year   state  pop  debt  eastern
one    2000    Ohio  1.5   NaN     True
two    2001    Ohio  1.7  -1.2     True
three  2002    Ohio  3.6   NaN     True
four   2001  Nevada  2.4  -1.5    False
five   2002  Nevada  2.9  -1.7    False
six    2003  Nevada  3.2   NaN    False
```

CAUTION

New columns cannot be created with the frame2.eastern syntax.

The del method can then be used to remove this column:

```
In [66]: del frame2['eastern']

In [67]: frame2.columns
Out[67]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

CAUTION

The column returned from indexing a DataFrame is a *view* on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame. The column can be explicitly copied with the Series’s copy method.

Another common form of data is a nested dict of dicts:

```
In [68]: pop = {'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6},
   ....:        'Nevada': {2001: 2.4, 2002: 2.9}}
```

If the nested dict is passed to the DataFrame, pandas will interpret the outer dict keys as the columns and the inner keys as the row indices:

```
In [69]: frame3 = pd.DataFrame(pop)

In [70]: frame3
Out[70]:
      Ohio  Nevada
2000   1.5     NaN
2001   1.7     2.4
2002   3.6     2.9
```

You can transpose the DataFrame (swap rows and columns) with similar syntax to a NumPy array:

```
In [71]: frame3.T
Out[71]:
        2000  2001  2002
Ohio     1.5   1.7   3.6
Nevada   NaN   2.4   2.9
```

WARNING

Note that transposing discards the column data types if the columns do not all have the same data type, so transposing and then transposing back may result in the previous type information being lost. The columns become arrays of pure Python objects in this case.

The keys in the inner dicts are combined and sorted to form the index in the result. This isn’t true if an explicit index is specified:

```
In [72]: pd.DataFrame(pop, index=[2001, 2002, 2003])
Out[72]:
      Ohio  Nevada
2001   1.7     2.4
2002   3.6     2.9
2003   NaN     NaN
```

Dicts of Series are treated in much the same way:

```
In [73]: pdata = {'Ohio': frame3['Ohio'][:-1],
   ....:          'Nevada': frame3['Nevada'][:2]}

In [74]: pd.DataFrame(pdata)
Out[74]:
      Ohio  Nevada
2000   1.5     NaN
2001   1.7     2.4
```

For a list of many of the things you can pass the DataFrame constructor, see Table 5-1.

If a DataFrame's `index` and `columns` have their `name` attributes set, these will also be displayed:

```
In [75]: frame3.index.name = 'year'; frame3.columns.name =
'state'

In [76]: frame3
Out[76]:
state  Ohio  Nevada
year
2000    1.5     NaN
2001    1.7     2.4
2002    3.6     2.9
```

DataFrame's `to_numpy` method returns the data contained in the DataFrame as a two-dimensional ndarray:

```
In [77]: frame3.to_numpy()
Out[77]:
array([[1.5, nan],
       [1.7, 2.4],
       [3.6, 2.9]])
```

If the DataFrame's columns are different dtypes, the dtype of the returned array will be chosen to accommodate all of the columns:

```
In [78]: frame2.to_numpy()
Out[78]:
```

```
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, -1.2],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

Table 5-1. Possible data inputs to DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame; all sequences must be the same length
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column; indexes from each Series are unioned together to form the result's row index if no explicit index is passed
dict of dicts	Each inner dict becomes a column; keys are unioned to form the row index as in the "dict of Series" case
List of dicts or Series	Each item becomes a row in the DataFrame; union of dict keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become missing in the DataFrame result

Index Objects

pandas's Index objects are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or other sequence of labels you use when constructing a Series or DataFrame is internally converted to an Index:

```
In [79]: obj = pd.Series(range(3), index=['a', 'b', 'c'])

In [80]: index = obj.index

In [81]: index
Out[81]: Index(['a', 'b', 'c'], dtype='object')

In [82]: index[1:]
Out[82]: Index(['b', 'c'], dtype='object')
```

Index objects are immutable and thus can’t be modified by the user:

```
index[1] = 'd'  # TypeError
```

Immutability makes it safer to share Index objects among data structures:

```
In [83]: labels = pd.Index(np.arange(3))

In [84]: labels
Out[84]: Int64Index([0, 1, 2], dtype='int64')

In [85]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [86]: obj2
Out[86]:
0    1.5
1   -2.5
2    0.0
dtype: float64

In [87]: obj2.index is labels
Out[87]: True
```

CAUTION

Some users will not often take advantage of the capabilities provided by indexes, but because some operations will yield results containing indexed data, it’s important to understand how they work.

In addition to being array-like, an Index also behaves like a fixed-size set:

```
In [88]: frame3
Out[88]:
state  Ohio  Nevada
year
2000    1.5     NaN
2001    1.7     2.4
2002    3.6     2.9

In [89]: frame3.columns
Out[89]: Index(['Ohio', 'Nevada'], dtype='object', name='state')

In [90]: 'Ohio' in frame3.columns
Out[90]: True

In [91]: 2003 in frame3.index
Out[91]: False
```

Unlike Python sets, a pandas Index can contain duplicate labels:

```
In [92]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])

In [93]: dup_labels
Out[93]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Selections with duplicate labels will select all occurrences of that label.

Each Index has a number of methods and properties for set logic, which answer other common questions about the data it contains. Some useful ones are summarized in Table 5-2.

Table 5-2. Some Index methods and properties

Method	Description
`append`	Concatenate with additional Index objects, producing a new Index
`difference`	Compute set difference as an Index
`intersection`	Compute set intersection
`union`	Compute set union
`isin`	Compute boolean array indicating whether each value is contained in the passed collection
`delete`	Compute new Index with element at index `i` deleted
`drop`	Compute new Index by deleting passed values
`insert`	Compute new Index by inserting element at index `i`
`is_monotonic`	Returns `True` if each element is greater than or equal to the previous element
`is_unique`	Returns `True` if the Index has no duplicate values
`unique`	Compute the array of unique values in the Index

5.2 Essential Functionality

This section will walk you through the fundamental mechanics of interacting with the data contained in a Series or DataFrame. In the chapters to come, we will delve more deeply into data analysis and manipulation topics using pandas. This book is not intended to serve as exhaustive documentation for the pandas library; instead, we’ll focus on familiarizing you with heavily-used features, leaving the less common (i.e., more esoteric) things for you to learn more about by reading the online pandas documentation.

Reindexing

An important method on pandas objects is `reindex`, which means to create a new object with the values rearranged to align with the new index. Consider an example:

```
In [94]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b',
'a', 'c'])

In [95]: obj
Out[95]:
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

Calling `reindex` on this Series rearranges the data according to the new index, introducing missing values if any index values were not already present:

```
In [96]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [97]: obj2
Out[97]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

For ordered data like time series, it may be desirable to do some interpolation or filling of values when reindexing. The `method` option allows us to do this, using a method such as `ffill`, which forward-fills the values:

```
In [98]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0,
2, 4])

In [99]: obj3
Out[99]:
0      blue
```

```
2    purple
4    yellow
dtype: object

In [100]: obj3.reindex(range(6), method='ffill')
Out[100]:
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

With DataFrame, `reindex` can alter either the (row) index, columns, or both. When passed only a sequence, it reindexes the rows in the result:

```
In [101]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
   .....:                      index=['a', 'c', 'd'],
   .....:                      columns=['Ohio', 'Texas',
'California'])

In [102]: frame
Out[102]:
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

In [103]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [104]: frame2
Out[104]:
   Ohio  Texas  California
a   0.0    1.0         2.0
b   NaN    NaN         NaN
c   3.0    4.0         5.0
d   6.0    7.0         8.0
```

The columns can be reindexed with the `columns` keyword:

```
In [105]: states = ['Texas', 'Utah', 'California']

In [106]: frame.reindex(columns=states)
Out[106]:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

See Table 5-3 for more about the arguments to `reindex`.

As we’ll explore later, you can also reindex by using the `loc` operator, and many users prefer to always do it that way. This only works if all of the new index labels already exist in the DataFrame (whereas `reindex` will insert missing data for new labels):

```
In [107]: frame.loc[['a', 'c', 'd'], ['Texas', 'California']]
Out[107]:
   Texas  California
a      1           2
c      4           5
d      7           8
```

Table 5-3. reindex function arguments

Argument	Description
`labels`	New sequence to use as index. Can be Index instance or any other sequence-like Python data structure. An Index will be used exactly as is without any copying.
`axis`	The axis to reindex, whether `'index'` (rows) or `'columns'`. The default is `'index'`. You can alternately do `reindex(index=new_labels)` or `reindex(columns=new_labels)`.
`method`	Interpolation (fill) method; `'ffill'` fills forward, while `'bfill'` fills backward.
`fill_value`	Substitute value to use when introducing missing data by reindexing. Use `fill_value='missing'` (the default behavior) when you want absent labels to have null values in the result.
`limit`	When forward- or backfilling, maximum size gap (in number of elements) to fill.
`tolerance`	When forward- or backfilling, maximum size gap (in absolute numeric distance) to fill for inexact matches.
`level`	Match simple Index on level of MultiIndex; otherwise select subset of.
`copy`	If `True`, always copy underlying data even if new index is equivalent to old index; if `False`, do not copy the data when the indexes are equivalent.

Dropping Entries from an Axis

Dropping one or more entries from an axis is easy if you already have an index array or list without those entries. As that can require a bit of munging and set logic, the `drop` method will return a new object with the indicated value or values deleted from an axis:

```
In [108]: obj = pd.Series(np.arange(5.), index=['a', 'b', 'c',
'd', 'e'])

In [109]: obj
Out[109]:
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64

In [110]: new_obj = obj.drop('c')

In [111]: new_obj
Out[111]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64

In [112]: obj.drop(['d', 'c'])
Out[112]:
a    0.0
b    1.0
e    4.0
dtype: float64
```

With DataFrame, index values can be deleted from either axis. To illustrate this, we first create an example DataFrame:

```
In [113]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                      index=['Ohio', 'Colorado', 'Utah',
'New York'],
   .....:                      columns=['one', 'two', 'three',
'four'])

In [114]: data
Out[114]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Calling `drop` with a sequence of labels will drop values from the row labels (axis 0):

```
In [115]: data.drop(['Colorado', 'Ohio'])
Out[115]:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

You can drop values from the columns by passing `axis=1` or `axis='columns'`:

```
In [116]: data.drop('two', axis=1)
Out[116]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

In [117]: data.drop(['two', 'four'], axis='columns')
Out[117]:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

Many functions, like `drop`, which modify the size or shape of a Series or DataFrame, can manipulate an object *in-place* without returning a new object:

```
In [118]: obj.drop('c', inplace=True)

In [119]: obj
Out[119]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

Be careful with the `inplace`, as it destroys any data that is dropped. If we could go back in time, we probably would choose to not provide in-place options in many functions.

Indexing, Selection, and Filtering

Series indexing (`obj[...]`) works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers. Here are some examples of this:

```
In [120]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c',
'd'])

In [121]: obj
Out[121]:
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64

In [122]: obj['b']
Out[122]: 1.0

In [123]: obj[1]
Out[123]: 1.0

In [124]: obj[2:4]
Out[124]:
c    2.0
d    3.0
dtype: float64

In [125]: obj[['b', 'a', 'd']]
Out[125]:
b    1.0
a    0.0
d    3.0
dtype: float64

In [126]: obj[[1, 3]]
Out[126]:
b    1.0
d    3.0
```

```
dtype: float64

In [127]: obj[obj < 2]
Out[127]:
a    0.0
b    1.0
dtype: float64
```

While you can select data by label this way, the preferred way to select index values is the special `loc` operator:

```
In [128]: obj.loc[['b', 'a', 'd']]
Out[128]:
b    1.0
a    0.0
d    3.0
dtype: float64
```

The reason to prefer `loc` is because of the different treatment of integers when indexing with `[]`. Regular `[]`-based indexing will treat integers as labels if the index contains integers, so the behavior differs depending on the data type of the index. For example:

```
In [129]: obj1 = pd.Series([1, 2, 3], index=[2, 0, 1])

In [130]: obj2 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

In [131]: obj1
Out[131]:
2    1
0    2
1    3
dtype: int64

In [132]: obj2
Out[132]:
a    1
b    2
c    3
dtype: int64

In [133]: obj1[[0, 1, 2]]
Out[133]:
```

```
0    2
1    3
2    1
dtype: int64

In [134]: obj2[[0, 1, 2]]
Out[134]:
a    1
b    2
c    3
dtype: int64
```

With `loc`, the expression `obj.loc[[0, 1, 2]]` will fail when the index does not contain integers:

```
In [135]: obj1.loc[[0, 1, 2]]
Out[135]:
0    2
1    3
2    1
dtype: int64
```

Since `loc` operator indexes exclusively with labels, there is also a `iloc` operator that indexes exclusively with integers to work consistently whether the index contains integers or not:

```
In [136]: obj1.iloc[[0, 1, 2]]
Out[136]:
2    1
0    2
1    3
dtype: int64

In [137]: obj2.iloc[[0, 1, 2]]
Out[137]:
a    1
b    2
c    3
dtype: int64
```

You can also slice with labels, but it works differently from normal Python slicing in that the endpoint is inclusive:

```
In [138]: obj.loc['b':'c']
Out[138]:
b    1.0
c    2.0
dtype: float64
```

Setting using these methods modifies the corresponding section of the Series:

```
In [139]: obj.loc['b':'c'] = 5

In [140]: obj
Out[140]:
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

```
In [141]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=['Ohio', 'Colorado', 'Utah',
'New York'],
   .....:                     columns=['one', 'two', 'three',
'four'])

In [142]: data
Out[142]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15

In [143]: data['two']
Out[143]:
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64
```

```
In [144]: data[['three', 'one']]
Out[144]:
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

Indexing like this has a few special cases. First, slicing or selecting data with a boolean array:

```
In [145]: data[:2]
Out[145]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7

In [146]: data[data['three'] > 5]
Out[146]:
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

The row selection syntax `data[:2]` is provided as a convenience. Passing a single element or a list to the `[]` operator selects columns.

Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

```
In [147]: data < 5
Out[147]:
            one    two  three   four
Ohio       True   True   True   True
Colorado   True  False  False  False
Utah      False  False  False  False
New York  False  False  False  False

In [148]: data[data < 5] = 0

In [149]: data
Out[149]:
          one  two  three  four
Ohio        0    0      0     0
```

```
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Selection on DataFrame with loc and iloc

Like Series, DataFrame has special operators `loc` and `iloc` for label-based and integer-based indexing, respectively. Since DataFrame is two-dimensional, you can select a subset of the rows and columns with NumPy-like notation using either axis labels (`loc`) or integers (`iloc`).

As a first example, let's select a single row and multiple columns by label:

```
In [150]: data.loc['Colorado', ['two', 'three']]
Out[150]:
two      5
three    6
Name: Colorado, dtype: int64
```

We'll then perform some similar selections with integers using `iloc`:

```
In [151]: data.iloc[2, [3, 0, 1]]
Out[151]:
four    11
one      8
two      9
Name: Utah, dtype: int64

In [152]: data.iloc[2]
Out[152]:
one       8
two       9
three    10
four     11
Name: Utah, dtype: int64

In [153]: data.iloc[[1, 2], [3, 0, 1]]
Out[153]:
          four  one  two
Colorado     7    0    5
Utah        11    8    9
```

Both indexing functions work with slices in addition to single labels or lists of labels:

```
In [154]: data.loc[:'Utah', 'two']
Out[154]:
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int64

In [155]: data.iloc[:, :3][data.three > 5]
Out[155]:
          one  two  three
Colorado    0    5      6
Utah        8    9     10
New York   12   13     14
```

So there are many ways to select and rearrange the data contained in a pandas object. For DataFrame, Table 5-4 provides a short summary of many of them. As you'll see later, there are a number of additional options for working with hierarchical indexes.

Table 5-4. Indexing options with DataFrame

Type	Notes
`df[val]`	Select single column or sequence of columns from the DataFrame; special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion)
`df.loc[val]`	Selects single row or subset of rows from the DataFrame by label
`df.loc[:, val]`	Selects single column or subset of columns by label
`df.loc[val1, val2]`	Select both rows and columns by label
`df.iloc[where]`	Selects single row or subset of rows from the DataFrame by integer position
`df.iloc[:, where]`	Selects single column or subset of columns by integer position
`df.iloc[where_i, where_j]`	Select both rows and columns by integer position
`df.at[label_i, label_j]`	Select a single scalar value by row and column label
`df.iat[i, j]`	Select a single scalar value by row and column position (integers)
`reindex` method	Select either rows or columns by labels
`get_value, set_value` methods	Select single value by row and column label

Integer Indexing Pitfalls

Working with pandas objects indexed by integers is something that often trips up new users due to some differences with indexing semantics on built-in Python data structures like lists and tuples. For example, you might not expect the following code to generate an error:

```
ser = pd.Series(np.arange(3.))
ser
```

```
ser[-1]
```

In this case, pandas could "fall back" on integer indexing, but it's difficult to do this in general without introducing subtle bugs. Here we have an index containing 0, 1, 2, but inferring what the user wants (label-based indexing or position-based) is difficult:

```
In [157]: ser
Out[157]:
0    0.0
1    1.0
2    2.0
dtype: float64
```

On the other hand, with a non-integer index, there is no potential for ambiguity:

```
In [158]: ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])

In [159]: ser2[-1]
Out[159]: 2.0
```

If you have an axis index containing integers, data selection will always be label-oriented. As I said above, If you use `loc` (for labels) or `iloc` (for integers) you will get exactly what you want:

```
In [160]: ser[-1]
-----------------------------------------------------------------
----------
ValueError                                Traceback (most recent
call last)
/miniconda/envs/book-env/lib/python3.8/site-
packages/pandas/core/indexes/range.py
 in get_loc(self, key, method, tolerance)
    350                 try:
--> 351                     return self._range.index(new_key)
    352                 except ValueError as err:
ValueError: -1 is not in range
The above exception was the direct cause of the following
exception:
KeyError                                  Traceback (most recent
call last)
```

```
<ipython-input-160-44969a759c20> in <module>
----> 1 ser[-1]
/miniconda/envs/book-env/lib/python3.8/site-
packages/pandas/core/series.py in __g
etitem__(self, key)
    822
    823         elif key_is_scalar:
--> 824             return self._get_value(key)
    825
    826         if is_hashable(key):
/miniconda/envs/book-env/lib/python3.8/site-
packages/pandas/core/series.py in _ge
t_value(self, label, takeable)
    930
    931         # Similar to Index.get_value, but we do not fall
back to position
al
--> 932         loc = self.index.get_loc(label)
    933         return self.index._get_values_for_loc(self, loc,
label)
    934
/miniconda/envs/book-env/lib/python3.8/site-
packages/pandas/core/indexes/range.py
 in get_loc(self, key, method, tolerance)
    351                     return self._range.index(new_key)
    352                 except ValueError as err:
--> 353                     raise KeyError(key) from err
    354             raise KeyError(key)
    355         return super().get_loc(key, method=method,
tolerance=tolerance)
KeyError: -1

In [161]: ser.iloc[-1]
Out[161]: 2.0
```

On the other hand, slicing with integers is always integer-oriented:

```
In [162]: ser[:2]
Out[162]:
0    0.0
1    1.0
dtype: float64
```

Arithmetic and Data Alignment

An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes. When you are adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs. For users with database experience, this is similar to an automatic outer join on the index labels. Let’s look at an example:

```
In [163]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c',
'd', 'e'])

In [164]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
   .....:                index=['a', 'c', 'e', 'f', 'g'])

In [165]: s1
Out[165]:
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64

In [166]: s2
Out[166]:
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

Adding these together yields:

```
In [167]: s1 + s2
Out[167]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

The internal data alignment introduces missing values in the label locations that don’t overlap. Missing values will then propagate in further arithmetic computations.

In the case of DataFrame, alignment is performed on both the rows and the columns:

```
In [168]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)),
columns=list('bcd'),
   .....:                      index=['Ohio', 'Texas', 'Colorado'])

In [169]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)),
columns=list('bde'),
   .....:                      index=['Utah', 'Ohio', 'Texas',
'Oregon'])

In [170]: df1
Out[170]:
            b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0

In [171]: df2
Out[171]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

Adding these together returns a DataFrame whose index and columns are the unions of the ones in each DataFrame:

```
In [172]: df1 + df2
Out[172]:
            b   c     d   e
Colorado  NaN NaN   NaN NaN
Ohio      3.0 NaN   6.0 NaN
Oregon    NaN NaN   NaN NaN
Texas     9.0 NaN  12.0 NaN
Utah      NaN NaN   NaN NaN
```

Since the `'c'` and `'e'` columns are not found in both DataFrame objects, they appear as all missing in the result. The same holds for the rows whose labels are not common to both objects.

If you add DataFrame objects with no column or row labels in common, the result will contain all nulls:

```
In [173]: df1 = pd.DataFrame({'A': [1, 2]})

In [174]: df2 = pd.DataFrame({'B': [3, 4]})

In [175]: df1
Out[175]:
   A
0  1
1  2

In [176]: df2
Out[176]:
   B
0  3
1  4

In [177]: df1 + df2
Out[177]:
    A   B
0 NaN NaN
1 NaN NaN
```

Arithmetic methods with fill values

In arithmetic operations between differently indexed objects, you might want to fill with a special value, like 0, when an axis label is found in one object but not the other. Here is an example where we set a particular value to NA (null) by assigning `np.nan` to it:

```
In [178]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
   .....:                     columns=list('abcd'))

In [179]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
   .....:                     columns=list('abcde'))

In [180]: df2.loc[1, 'b'] = np.nan
In [181]: df1
Out[181]:
     a    b     c     d
0  0.0  1.0   2.0   3.0
1  4.0  5.0   6.0   7.0
2  8.0  9.0  10.0  11.0

In [182]: df2
Out[182]:
      a     b     c     d     e
0   0.0   1.0   2.0   3.0   4.0
1   5.0   NaN   7.0   8.0   9.0
2  10.0  11.0  12.0  13.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

Adding these together results in missing values in the locations that don’t overlap:

```
In [183]: df1 + df2
Out[183]:
      a     b     c     d   e
0   0.0   2.0   4.0   6.0 NaN
1   9.0   NaN  13.0  15.0 NaN
2  18.0  20.0  22.0  24.0 NaN
3   NaN   NaN   NaN   NaN NaN
```

Using the `add` method on `df1`, I pass `df2` and an argument to `fill_value`:

```
In [184]: df1.add(df2, fill_value=0)
Out[184]:
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0   5.0  13.0  15.0   9.0
2  18.0  20.0  22.0  24.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

See Table 5-5 for a listing of Series and DataFrame methods for arithmetic. Each of them has a counterpart, starting with the letter `r`, that has arguments flipped. So these two statements are equivalent:

```
In [185]: 1 / df1
Out[185]:
       a         b         c         d
0    inf  1.000000  0.500000  0.333333
1  0.250  0.200000  0.166667  0.142857
2  0.125  0.111111  0.100000  0.090909

In [186]: df1.rdiv(1)
Out[186]:
       a         b         c         d
0    inf  1.000000  0.500000  0.333333
1  0.250  0.200000  0.166667  0.142857
2  0.125  0.111111  0.100000  0.090909
```

Relatedly, when reindexing a Series or DataFrame, you can also specify a different fill value:

```
In [187]: df1.reindex(columns=df2.columns, fill_value=0)
Out[187]:
     a    b     c     d  e
0  0.0  1.0   2.0   3.0  0
1  4.0  5.0   6.0   7.0  0
2  8.0  9.0  10.0  11.0  0
```

Table 5-5. Flexible arithmetic methods

Method	Description
`add, radd`	Methods for addition (+)
`sub, rsub`	Methods for subtraction (-)
`div, rdiv`	Methods for division (/)
`floordiv, rfloordiv`	Methods for floor division (//)
`mul, rmul`	Methods for multiplication (*)
`pow, rpow`	Methods for exponentiation (**)

Operations between DataFrame and Series

As with NumPy arrays of different dimensions, arithmetic between DataFrame and Series is also defined. First, as a motivating example, consider the difference between a two-dimensional array and one of its rows:

```
In [188]: arr = np.arange(12.).reshape((3, 4))

In [189]: arr
Out[189]:
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])

In [190]: arr[0]
Out[190]: array([0., 1., 2., 3.])

In [191]: arr - arr[0]
Out[191]:
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

When we subtract `arr[0]` from `arr`, the subtraction is performed once for each row. This is referred to as *broadcasting* and is explained in more detail as it relates to general NumPy arrays in Appendix A. Operations between a DataFrame and a Series are similar:

```
In [192]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
   .....:                      columns=list('bde'),
   .....:                      index=['Utah', 'Ohio', 'Texas',
'Oregon'])

In [193]: series = frame.iloc[0]

In [194]: frame
Out[194]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [195]: series
Out[195]:
b    0.0
d    1.0
```

```
e    2.0
Name: Utah, dtype: float64
```

By default, arithmetic between DataFrame and Series matches the index of the Series on the DataFrame’s columns, broadcasting down the rows:

```
In [196]: frame - series
Out[196]:
          b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
```

If an index value is not found in either the DataFrame’s columns or the Series’s index, the objects will be reindexed to form the union:

```
In [197]: series2 = pd.Series(range(3), index=['b', 'e', 'f'])

In [198]: frame + series2
Out[198]:
          b   d     e   f
Utah    0.0 NaN   3.0 NaN
Ohio    3.0 NaN   6.0 NaN
Texas   6.0 NaN   9.0 NaN
Oregon  9.0 NaN  12.0 NaN
```

If you want to instead broadcast over the columns, matching on the rows, you have to use one of the arithmetic methods. For example:

```
In [199]: series3 = frame['d']

In [200]: frame
Out[200]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [201]: series3
Out[201]:
Utah       1.0
```

```
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64

In [202]: frame.sub(series3, axis='index')
Out[202]:
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0
```

The axis number that you pass is the *axis to match on*. In this case we mean to match on the DataFrame's row index (`axis='index'` or `axis=0`) and broadcast across.

Function Application and Mapping

NumPy ufuncs (element-wise array methods) also work with pandas objects:

```
In [203]: frame = pd.DataFrame(np.random.randn(4, 3),
columns=list('bde'),
   .....:                      index=['Utah', 'Ohio', 'Texas',
'Oregon'])

In [204]: frame
Out[204]:
               b         d         e
Utah   -0.204708  0.478943 -0.519439
Ohio   -0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189 -1.296221

In [205]: np.abs(frame)
Out[205]:
               b         d         e
Utah    0.204708  0.478943  0.519439
Ohio    0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189  1.296221
```

Another frequent operation is applying a function on one-dimensional arrays to each column or row. DataFrame’s apply method does exactly this:

```
In [206]: f = lambda x: x.max() - x.min()

In [207]: frame.apply(f)
Out[207]:
b    1.802165
d    1.684034
e    2.689627
dtype: float64
```

Here the function f, which computes the difference between the maximum and minimum of a Series, is invoked once on each column in frame. The result is a Series having the columns of frame as its index.

If you pass axis='columns' to apply, the function will be invoked once per row instead:

```
In [208]: frame.apply(f, axis='columns')
Out[208]:
Utah      0.998382
Ohio      2.521511
Texas     0.676115
Oregon    2.542656
dtype: float64
```

Many of the most common array statistics (like sum and mean) are DataFrame methods, so using apply is not necessary.

The function passed to apply need not return a scalar value; it can also return a Series with multiple values:

```
In [209]: def f(x):
   .....:     return pd.Series([x.min(), x.max()], index=['min',
'max'])

In [210]: frame.apply(f)
Out[210]:
            b         d         e
```

```
min -0.555730  0.281746 -1.296221
max  1.246435  1.965781  1.393406
```

Element-wise Python functions can be used, too. Suppose you wanted to compute a formatted string from each floating-point value in `frame`. You can do this with `applymap`:

```
In [211]: format = lambda x: '%.2f' % x

In [212]: frame.applymap(format)
Out[212]:
            b     d      e
Utah    -0.20  0.48  -0.52
Ohio    -0.56  1.97   1.39
Texas    0.09  0.28   0.77
Oregon   1.25  1.01  -1.30
```

The reason for the name `applymap` is that Series has a `map` method for applying an element-wise function:

```
In [213]: frame['e'].map(format)
Out[213]:
Utah      -0.52
Ohio       1.39
Texas      0.77
Oregon    -1.30
Name: e, dtype: object
```

Sorting and Ranking

Sorting a dataset by some criterion is another important built-in operation. To sort lexicographically by row or column index, use the `sort_index` method, which returns a new, sorted object:

```
In [214]: obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])

In [215]: obj.sort_index()
Out[215]:
a    1
b    2
c    3
```

```
d    0
dtype: int64
```

With a DataFrame, you can sort by index on either axis:

```
In [216]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
   .....:                      index=['three', 'one'],
   .....:                      columns=['d', 'a', 'b', 'c'])

In [217]: frame.sort_index()
Out[217]: 
       d  a  b  c
one    4  5  6  7
three  0  1  2  3

In [218]: frame.sort_index(axis=1)
Out[218]: 
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

The data is sorted in ascending order by default, but can be sorted in descending order, too:

```
In [219]: frame.sort_index(axis=1, ascending=False)
Out[219]: 
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

To sort a Series by its values, use its `sort_values` method:

```
In [220]: obj = pd.Series([4, 7, -3, 2])

In [221]: obj.sort_values()
Out[221]: 
2   -3
3    2
0    4
1    7
dtype: int64
```

Any missing values are sorted to the end of the Series by default:

```
In [222]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

In [223]: obj.sort_values()
Out[223]:
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

When sorting a DataFrame, you can use the data in one or more columns as the sort keys. To do so, pass one or more column names to the `by` option of `sort_values`:

```
In [224]: frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1,
0, 1]})

In [225]: frame
Out[225]:
   b  a
0  4  0
1  7  1
2 -3  0
3  2  1

In [226]: frame.sort_values(by='b')
Out[226]:
   b  a
2 -3  0
3  2  1
0  4  0
1  7  1
```

To sort by multiple columns, pass a list of names:

```
In [227]: frame.sort_values(by=['a', 'b'])
Out[227]:
   b  a
2 -3  0
0  4  0
3  2  1
1  7  1
```

Ranking assigns ranks from one through the number of valid data points in an array. The `rank` methods for Series and DataFrame are the place to look; by default `rank` breaks ties by assigning each group the mean rank:

```
In [228]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

In [229]: obj.rank()
Out[229]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

Ranks can also be assigned according to the order in which they're observed in the data:

```
In [230]: obj.rank(method='first')
Out[230]:
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

Here, instead of using the average rank 6.5 for the entries 0 and 2, they instead have been set to 6 and 7 because label 0 precedes label 2 in the data.

You can rank in descending order, too:

```
# Assign tie values the maximum rank in the group
In [231]: obj.rank(ascending=False, method='max')
Out[231]:
0    2.0
1    7.0
2    2.0
3    4.0
```

```
4    5.0
5    6.0
6    4.0
dtype: float64
```

See Table 5-6 for a list of tie-breaking methods available.

DataFrame can compute ranks over the rows or the columns:

```
In [232]: frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1,
0, 1],
   .....:                       'c': [-2, 5, 8, -2.5]})

In [233]: frame
Out[233]:
     b  a    c
0  4.3  0 -2.0
1  7.0  1  5.0
2 -3.0  0  8.0
3  2.0  1 -2.5

In [234]: frame.rank(axis='columns')
Out[234]:
     b    a    c
0  3.0  2.0  1.0
1  3.0  1.0  2.0
2  1.0  2.0  3.0
3  3.0  2.0  1.0
```

Table 5-6. Tie-breaking methods with rank

Method	Description
`'average'`	Default: assign the average rank to each entry in the equal group
`'min'`	Use the minimum rank for the whole group
`'max'`	Use the maximum rank for the whole group
`'first'`	Assign ranks in the order the values appear in the data
`'dense'`	Like `method='min'`, but ranks always increase by 1 in between groups rather than the number of equal elements in a group

Axis Indexes with Duplicate Labels

Up until now all of the examples we’ve looked at have had unique axis labels (index values). While many pandas functions (like `reindex`) require that the labels be unique, it’s not mandatory. Let’s consider a small Series with duplicate indices:

```
In [235]: obj = pd.Series(range(5), index=['a', 'a', 'b', 'b',
'c'])

In [236]: obj
Out[236]:
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

The index’s `is_unique` property can tell you whether its labels are unique or not:

```
In [237]: obj.index.is_unique
Out[237]: False
```

Data selection is one of the main things that behaves differently with duplicates. Indexing a label with multiple entries returns a Series, while single entries return a scalar value:

```
In [238]: obj['a']
Out[238]:
a    0
a    1
dtype: int64

In [239]: obj['c']
Out[239]: 4
```

This can make your code more complicated, as the output type from indexing can vary based on whether a label is repeated or not.

The same logic extends to indexing rows in a DataFrame:

```
In [240]: df = pd.DataFrame(np.random.randn(5, 3), index=['a',
'a', 'b', 'b', 'c'
])

In [241]: df
Out[241]:
          0         1         2
a  0.274992  0.228913  1.352917
a  0.886429 -2.001637 -0.371843
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
c -0.577087  0.124121  0.302614

In [242]: df.loc['b']
Out[242]:
          0         1         2
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228

In [243]: df.loc['c']
Out[243]:
0   -0.577087
1    0.124121
2    0.302614
Name: c, dtype: float64
```

5.3 Summarizing and Computing Descriptive Statistics

pandas objects are equipped with a set of common mathematical and statistical methods. Most of these fall into the category of *reductions* or *summary statistics*, methods that extract a single value (like the sum or mean) from a Series or a Series of values from the rows or columns of a DataFrame. Compared with the similar methods found on NumPy arrays, they have built-in handling for missing data. Consider a small DataFrame:

```
In [244]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
   .....:                     [np.nan, np.nan], [0.75, -1.3]],
   .....:                    index=['a', 'b', 'c', 'd'],
   .....:                    columns=['one', 'two'])
```

```
In [245]: df
Out[245]:
    one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
```

Calling DataFrame’s `sum` method returns a Series containing column sums:

```
In [246]: df.sum()
Out[246]:
one    9.25
two   -5.80
dtype: float64
```

Passing `axis='columns'` or `axis=1` sums across the columns instead:

```
In [247]: df.sum(axis='columns')
Out[247]:
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

When an entire row or column contains all NA values, the sum is 0. This can be disabled with the `skipna` option:

```
In [248]: df.sum(axis='columns', skipna=False)
Out[248]:
a     NaN
b    2.60
c     NaN
d   -0.55
dtype: float64
```

Some aggregations, like `mean`, require at least one non-NA value to yield a value result, so here we have:

```
In [249]: df.mean(axis='columns')
Out[249]:
a    1.400
b    1.300
c      NaN
d   -0.275
dtype: float64
```

See Table 5-7 for a list of common options for each reduction method.

Table 5-7. Options for reduction methods

Method	Description
`axis`	Axis to reduce over; 0 for DataFrame’s rows and 1 for columns
`skipna`	Exclude missing values; `True` by default
`level`	Reduce grouped by level if the axis is hierarchically indexed (MultiIndex)

Some methods, like `idxmin` and `idxmax`, return indirect statistics like the index value where the minimum or maximum values are attained:

```
In [250]: df.idxmax()
Out[250]:
one    b
two    d
dtype: object
```

Other methods are *accumulations*:

```
In [251]: df.cumsum()
Out[251]:
    one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
```

Another type of method is neither a reduction nor an accumulation. `describe` is one such example, producing multiple summary statistics in one shot:

```
In [252]: df.describe()
Out[252]:
            one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

On non-numeric data, `describe` produces alternative summary statistics:

```
In [253]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)

In [254]: obj.describe()
Out[254]:
count     16
unique     3
top        a
freq       8
dtype: object
```

See Table 5-8 for a full list of summary statistics and related methods.

Table 5-8. Descriptive and summary statistics

Method	Description
`count`	Number of non-NA values
`describe`	Compute set of summary statistics for Series or each DataFrame column
`min, max`	Compute minimum and maximum values
`argmin, argmax`	Compute index locations (integers) at which minimum or maximum value obtained, respectively
`idxmin, idxmax`	Compute index labels at which minimum or maximum value obtained, respectively
`quantile`	Compute sample quantile ranging from 0 to 1
`sum`	Sum of values
`mean`	Mean of values
`median`	Arithmetic median (50% quantile) of values
`mad`	Mean absolute deviation from mean value
`prod`	Product of all values
`var`	Sample variance of values
`std`	Sample standard deviation of values
`skew`	Sample skewness (third moment) of values
`kurt`	Sample kurtosis (fourth moment) of values
`cumsum`	Cumulative sum of values
`cummin, cummax`	Cumulative minimum or maximum of values, respectively
`cumprod`	Cumulative product of values
`diff`	Compute first arithmetic difference (useful for time series)
`pct_change`	Compute percent changes

Correlation and Covariance

Some summary statistics, like correlation and covariance, are computed from pairs of arguments. Let's consider some DataFrames of stock prices and volumes originally obtained from Yahoo! Finance and available in binary Python pickle files you can find in the book's accompanying datasets on GitHub:

```
In [255]: price = pd.read_pickle('examples/yahoo_price.pkl')

In [256]: volume = pd.read_pickle('examples/yahoo_volume.pkl')
```

I now compute percent changes of the prices, a time series operation which will be explored further in [Link to Come]:

```
In [257]: returns = price.pct_change()

In [258]: returns.tail()
Out[258]:
                AAPL      GOOG       IBM      MSFT
Date
2016-10-17 -0.000680  0.001837  0.002072 -0.003483
2016-10-18 -0.000681  0.019616 -0.026168  0.007690
2016-10-19 -0.002979  0.007846  0.003583 -0.002255
2016-10-20 -0.000512 -0.005652  0.001719 -0.004867
2016-10-21 -0.003930  0.003011 -0.012474  0.042096
```

The `corr` method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series. Relatedly, `cov` computes the covariance:

```
In [259]: returns['MSFT'].corr(returns['IBM'])
Out[259]: 0.49976361144151144

In [260]: returns['MSFT'].cov(returns['IBM'])
Out[260]: 8.870655479703546e-05
```

Since `MSFT` is a valid Python attribute, we can also select these columns using more concise syntax:

```
In [261]: returns.MSFT.corr(returns.IBM)
Out[261]: 0.49976361144151144
```

DataFrame's `corr` and `cov` methods, on the other hand, return a full correlation or covariance matrix as a DataFrame, respectively:

```
In [262]: returns.corr()
Out[262]:
          AAPL      GOOG       IBM      MSFT
AAPL  1.000000  0.407919  0.386817  0.389695
GOOG  0.407919  1.000000  0.405099  0.465919
IBM   0.386817  0.405099  1.000000  0.499764
MSFT  0.389695  0.465919  0.499764  1.000000

In [263]: returns.cov()
Out[263]:
          AAPL      GOOG       IBM      MSFT
AAPL  0.000277  0.000107  0.000078  0.000095
GOOG  0.000107  0.000251  0.000078  0.000108
IBM   0.000078  0.000078  0.000146  0.000089
MSFT  0.000095  0.000108  0.000089  0.000215
```

Using DataFrame's `corrwith` method, you can compute pairwise correlations between a DataFrame's columns or rows with another Series or DataFrame. Passing a Series returns a Series with the correlation value computed for each column:

```
In [264]: returns.corrwith(returns.IBM)
Out[264]:
AAPL    0.386817
GOOG    0.405099
IBM     1.000000
MSFT    0.499764
dtype: float64
```

Passing a DataFrame computes the correlations of matching column names. Here I compute correlations of percent changes with volume:

```
In [265]: returns.corrwith(volume)
Out[265]:
AAPL   -0.075565
GOOG   -0.007067
IBM    -0.204849
MSFT   -0.092950
dtype: float64
```

Passing `axis='columns'` does things row-by-row instead. In all cases, the data points are aligned by label before the correlation is computed.

Unique Values, Value Counts, and Membership

Another class of related methods extracts information about the values contained in a one-dimensional Series. To illustrate these, consider this example:

```
In [266]: obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b',
'c', 'c'])
```

The first function is `unique`, which gives you an array of the unique values in a Series:

```
In [267]: uniques = obj.unique()

In [268]: uniques
Out[268]: array(['c', 'a', 'd', 'b'], dtype=object)
```

The unique values are not necessarily returned in sorted order, but could be sorted after the fact if needed (`uniques.sort()`). Relatedly, `value_counts` computes a Series containing value frequencies:

```
In [269]: obj.value_counts()
Out[269]:
a    3
c    3
b    2
d    1
dtype: int64
```

The Series is sorted by value in descending order as a convenience. `value_counts` is also available as a top-level pandas method that can be used with any array or sequence:

```
In [270]: pd.value_counts(obj.values, sort=False)
Out[270]:
b    2
a    3
d    1
c    3
dtype: int64
```

`isin` performs a vectorized set membership check and can be useful in filtering a dataset down to a subset of values in a Series or column in a DataFrame:

```
In [271]: obj
Out[271]:
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object

In [272]: mask = obj.isin(['b', 'c'])

In [273]: mask
Out[273]:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool

In [274]: obj[mask]
Out[274]:
0    c
5    b
6    b
7    c
8    c
dtype: object
```

Related to `isin` is the `Index.get_indexer` method, which gives you an index array from an array of possibly non-distinct values into another array of distinct values:

```
In [275]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])

In [276]: unique_vals = pd.Series(['c', 'b', 'a'])

In [277]: indices = pd.Index(unique_vals).get_indexer(to_match)

In [278]: indices
Out[278]: array([0, 2, 1, 1, 0, 2])
```

See Table 5-9 for a reference on these methods.

Table 5-9. Unique, value counts, and set membership methods

Method	Description
`isin`	Compute boolean array indicating whether each Series value is contained in the passed sequence of values
`get_indexer`	Compute integer indices for each value in an array into another array of distinct values; helpful for data alignment and join-type operations
`unique`	Compute array of unique values in a Series, returned in the order observed
`value_counts`	Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order

In some cases, you may want to compute a histogram on multiple related columns in a DataFrame. Here's an example:

```
In [279]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
   .....:                      'Qu2': [2, 3, 1, 2, 3],
   .....:                      'Qu3': [1, 5, 2, 4, 4]})

In [280]: data
Out[280]:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

Passing `pandas.value_counts` to this DataFrame's `apply` function gives:

```
In [281]: result = data.apply(pd.value_counts).fillna(0)

In [282]: result
Out[282]:
   Qu1  Qu2  Qu3
1  1.0  1.0  1.0
2  0.0  2.0  1.0
3  2.0  2.0  0.0
4  2.0  0.0  2.0
5  0.0  0.0  1.0
```

Here, the row labels in the result are the distinct values occurring in all of the columns. The values are the respective counts of these values in each column.

There is also a `DataFrame.value_counts` method, but it computes counts considering each row of the DataFrame as a tuple to determine the number of occurrence of each distinct row:

```
In [283]: data = pd.DataFrame({'a': [1, 1, 1, 2, 2], 'b': [0, 0,
1, 0, 0]})

In [284]: data
Out[284]:
   a  b
0  1  0
1  1  0
2  1  1
3  2  0
4  2  0

In [285]: data.value_counts()
Out[285]:
a  b
1  0    2
2  0    2
1  1    1
dtype: int64
```

In this case, the result has an index representing the distinct rows as a hierarchical index, a topic we will explore in greater detail in [Link to Come].

5.4 Conclusion

In the next chapter, we’ll discuss tools for reading (or *loading*) and writing datasets with pandas. After that, we’ll dig deeper into data cleaning, wrangling, analysis, and visualization tools using pandas.

Chapter 6. Data Loading, Storage, and File Formats

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at arufino@oreilly.com.

Accessing data is a necessary first step for using most of the tools in this book. I’m going to be focused on data input and output using pandas, though there are numerous tools in other libraries to help with reading and writing data in various formats.

Input and output typically falls into a few main categories: reading text files and other more efficient on-disk formats, loading data from databases, and interacting with network sources like web APIs.

6.1 Reading and Writing Data in Text Format

pandas features a number of functions for reading tabular data as a DataFrame object. Table 6-1 summarizes some of them; `read_csv` is one of the most frequently used in this book.

Table 6-1. Parsing functions in pandas

Function	Description
`read_csv`	Load delimited data from a file, URL, or file-like object; use comma as default delimiter
`read_fwf`	Read data in fixed-width column format (i.e., no delimiters)
`read_clipboard`	Version of `read_csv` that reads data from the clipboard; useful for converting tables from web pages
`read_excel`	Read tabular data from an Excel XLS or XLSX file
`read_hdf`	Read HDF5 files written by pandas
`read_html`	Read all tables found in the given HTML document
`read_json`	Read data from a JSON (JavaScript Object Notation) string representation
`read_feather`	Read the Feather binary file format
`read_orc`	Read the Apache ORC binary file format
`read_parquet`	Read the Apache Parquet binary file format
`read_pickle`	Read an arbitrary object stored in Python pickle format
`read_sas`	Read a SAS dataset stored in one of the SAS system's custom storage formats
`read_spss`	Read a data file created by SPSS.
`read_sql`	Read the results of a SQL query (using SQLAlchemy) as a pandas DataFrame
`read_sql_table`	Read a whole SQL table (using SQLAlchemy) as a pandas DataFrame (equivalent to using a query that selects everything in that table using `read_sql`).
`read_stata`	Read a dataset from Stata file format

I'll give an overview of the mechanics of these functions, which are meant to convert text data into a DataFrame. The optional arguments for these functions may fall into a few categories:

Indexing

Can treat one or more columns as the returned DataFrame, and whether to get column names from the file, arguments you provide, or not at all.

Type inference and data conversion

This includes the user-defined value conversions and custom list of missing value markers.

Date and time parsing

Includes combining capability, including combining date and time information spread over multiple columns into a single column in the result.

Iterating

Support for iterating over chunks of very large files.

Unclean data issues

Skipping rows or a footer, comments, or other minor things like numeric data with thousands separated by commas.

Because of how messy data in the real world can be, some of the data loading functions (especially `read_csv`) have accumulated a long list of optional arguments over time. It's normal to feel overwhelmed by the number of different parameters (`read_csv` has around 50). The online pandas documentation has many examples about how each of them works, so if you're struggling to read a particular file, there might be a similar enough example to help you find the right parameters.

Some of these functions, like `pandas.read_csv`, perform *type inference*, because the column data types are not part of the data format. That means you don't necessarily have to specify which columns are numeric, integer, boolean, or string. Other data formats, like HDF5, ORC, and Parquet, have the data type information embedded in the format.

Handling dates and other custom types can require extra effort. Let's start with a small comma-separated (CSV) text file:

```
In [10]: !cat examples/ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

NOTE

Here I used the Unix `cat` shell command to print the raw contents of the file to the screen. If you're on Windows, you can use `type` instead of `cat` to achieve the same effect.

Since this is comma-delimited, we can use `read_csv` to read it into a DataFrame:

```
In [11]: df = pd.read_csv('examples/ex1.csv')

In [12]: df
Out[12]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

A file will not always have a header row. Consider this file:

```
In [13]: !cat examples/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

To read this file, you have a couple of options. You can allow pandas to assign default column names, or you can specify names yourself:

```
In [14]: pd.read_csv('examples/ex2.csv', header=None)
Out[14]:
   0   1   2   3      4
```

```
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo

In [15]: pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c',
'd', 'message'])
Out[15]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

Suppose you wanted the `message` column to be the index of the returned DataFrame. You can either indicate you want the column at index 4 or named `'message'` using the `index_col` argument:

```
In [16]: names = ['a', 'b', 'c', 'd', 'message']

In [17]: pd.read_csv('examples/ex2.csv', names=names,
index_col='message')
Out[17]:
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

In the event that you want to form a hierarchical index (discussed in more detail later) from multiple columns, pass a list of column numbers or names:

```
In [18]: !cat examples/csv_mindex.csv
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16

In [19]: parsed = pd.read_csv('examples/csv_mindex.csv',
   ....:                       index_col=['key1', 'key2'])
```

```
In [20]: parsed
Out[20]:
           value1  value2
key1 key2
one  a          1       2
     b          3       4
     c          5       6
     d          7       8
two  a          9      10
     b         11      12
     c         13      14
     d         15      16
```

In some cases, a table might not have a fixed delimiter, using whitespace or some other pattern to separate fields. Consider a text file that looks like this:

```
In [21]: list(open('examples/ex3.txt'))
Out[21]:
['            A         B         C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb  0.927272  0.302904 -0.032399\n',
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382  1.100491\n']
```

While you could do some munging by hand, the fields here are separated by a variable amount of whitespace. In these cases, you can pass a regular expression as a delimiter for `read_csv`. This can be expressed by the regular expression `\s+`, so we have then:

```
In [22]: result = pd.read_csv('examples/ex3.txt', sep='\s+')

In [23]: result
Out[23]:
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

Because there was one fewer column name than the number of data rows, `read_csv` infers that the first column should be the DataFrame’s index in this special case.

The parser functions have many additional arguments to help you handle the wide variety of exception file formats that occur (see a partial listing in Table 6-2). For example, you can skip the first, third, and fourth rows of a file with `skiprows`:

```
In [24]: !cat examples/ex4.csv
# hey!
a,b,c,d,message
# just wanted to make things more difficult for you
# who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
In [25]: pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
Out[25]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

Handling missing values is an important and frequently nuanced part of the file parsing process. Missing data is usually either not present (empty string) or marked by some *sentinel* (placeholder) value. By default, pandas uses a set of commonly occurring sentinels, such as `NA` and `NULL`:

```
In [26]: !cat examples/ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
In [27]: result = pd.read_csv('examples/ex5.csv')

In [28]: result
Out[28]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo

In [29]: pd.isna(result)
Out[29]:
   something      a      b      c      d  message
0      False  False  False  False  False     True
1      False  False  False   True  False    False
2      False  False  False  False  False    False
```

The `na_values` option can take either a list or set of strings to consider missing values:

```
In [30]: result = pd.read_csv('examples/ex5.csv', na_values=
['NULL'])

In [31]: result
Out[31]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

Different NA sentinels can be specified for each column in a dict:

```
In [32]: sentinels = {'message': ['foo', 'NA'], 'something':
['two']}

In [33]: pd.read_csv('examples/ex5.csv', na_values=sentinels)
Out[33]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       NaN  5   6   NaN   8   world
2     three  9  10  11.0  12     NaN
```

Table 6-2 lists some frequently used options in `pandas.read_csv`.

Table 6-2. Some read_csv function arguments

Argument	Description
`path`	String indicating filesystem location, URL, or file-like object
`sep` or `delimiter`	Character sequence or regular expression to use to split fields in each row
`header`	Row number to use as column names; defaults to 0 (first row), but should be `None` if there is no header row
`index_col`	Column numbers or names to use as the row index in the result; can be a single name/number or a list of them for a hierarchical index
`names`	List of column names for result.
`skiprows`	Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip.
`na_values`	Sequence of values to replace with NA.
`comment`	Character(s) to split comments off the end of lines.
`parse_dates`	Attempt to parse data to `datetime`; `False` by default. If `True`, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (e.g., if date/time split across two columns).
`keep_date_col`	If joining columns to parse date, keep the joined columns; `False` by default.
`converters`	Dict containing column number of name mapping to functions (e.g., `{'foo': f}` would apply the function `f` to all values in the `'foo'` column).
`dayfirst`	When parsing potentially ambiguous dates, treat as international format (e.g., 7/6/2012 -> June 7, 2012); `False` by default.
`date_parser`	Function to use to parse dates.
`nrows`	Number of rows to read from beginning of file.
`iterator`	Return a `TextFileReader` object for reading file piecemeal. This object can also used with the `with` statement.

Argument	Description
`chunksize`	For iteration, size of file chunks.
`skip_footer`	Number of lines to ignore at end of file.
`verbose`	Print various parser output information, like the number of missing values placed in non-numeric columns.
`encoding`	Text encoding for Unicode (e.g., `'utf-8'` for UTF-8 encoded text).
`squeeze`	If the parsed data only contains one column, return a Series.
`thousands`	Separator for thousands (e.g., `','` or `'.'`).

Reading Text Files in Pieces

When processing very large files or figuring out the right set of arguments to correctly process a large file, you may only want to read in a small piece of a file or iterate through smaller chunks of the file.

Before we look at a large file, we make the pandas display settings more compact:

```
In [34]: pd.options.display.max_rows = 10
```

Now we have:

```
In [35]: result = pd.read_csv('examples/ex6.csv')

In [36]: result
Out[36]:
           one       two     three      four key
0     0.467976 -0.038649 -0.295344 -1.824726   L
1    -0.358893  1.404453  0.704965 -0.200638   B
2    -0.501840  0.659254 -0.421691 -0.057688   G
3     0.204886  1.074134  1.388361 -0.982404   R
4     0.354628 -0.133116  0.283763 -0.837063   Q
...        ...       ...       ...       ...  ..
9995  2.311896 -0.417070 -1.409599 -0.515821   L
9996 -0.479893 -0.650419  0.745152 -0.646038   E
9997  0.523331  0.787112  0.486066  1.093156   K
9998 -0.362559  0.598894 -1.843201  0.887292   G
9999 -0.096376 -1.012999 -0.657431 -0.573315   0
[10000 rows x 5 columns]
```

If you want to only read a small number of rows (avoiding reading the entire file), specify that with `nrows`:

```
In [37]: pd.read_csv('examples/ex6.csv', nrows=5)
Out[37]:
        one       two     three      four key
0  0.467976 -0.038649 -0.295344 -1.824726   L
1 -0.358893  1.404453  0.704965 -0.200638   B
2 -0.501840  0.659254 -0.421691 -0.057688   G
3  0.204886  1.074134  1.388361 -0.982404   R
4  0.354628 -0.133116  0.283763 -0.837063   Q
```

To read a file in pieces, specify a `chunksize` as a number of rows:

```
In [38]: chunker = pd.read_csv('examples/ex6.csv',
chunksize=1000)

In [39]: chunker
Out[39]: <pandas.io.parsers.TextFileReader at 0x7f728a742970>
```

The `TextFileReader` object returned by `read_csv` allows you to iterate over the parts of the file according to the `chunksize`. For example, we can iterate over `ex6.csv`, aggregating the value counts in the `'key'` column like so:

```
chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

tot = pd.Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)
```

We have then:

```
In [41]: tot[:10]
Out[41]:
E    368.0
X    364.0
L    346.0
O    343.0
```

```
Q    340.0
M    338.0
J    337.0
F    335.0
K    334.0
H    330.0
dtype: float64
```

TextFileReader is also equipped with a get_chunk method that enables you to read pieces of an arbitrary size.

Writing Data to Text Format

Data can also be exported to a delimited format. Let’s consider one of the CSV files read before:

```
In [42]: data = pd.read_csv('examples/ex5.csv')

In [43]: data
Out[43]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

Using DataFrame’s to_csv method, we can write the data out to a comma-separated file:

```
In [44]: data.to_csv('examples/out.csv')

In [45]: !cat examples/out.csv
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

Other delimiters can be used, of course (writing to sys.stdout so it prints the text result to the console rather than a file):

```
In [46]: import sys
In [47]: data.to_csv(sys.stdout, sep='|')
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6||8|world
2|three|9|10|11.0|12|foo
```

Missing values appear as empty strings in the output. You might want to denote them by some other sentinel value:

```
In [48]: data.to_csv(sys.stdout, na_rep='NULL')
,something,a,b,c,d,message
0,one,1,2,3.0,4,NULL
1,two,5,6,NULL,8,world
2,three,9,10,11.0,12,foo
```

With no other options specified, both the row and column labels are written. Both of these can be disabled:

```
In [49]: data.to_csv(sys.stdout, index=False, header=False)
one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

You can also write only a subset of the columns, and in an order of your choosing:

```
In [50]: data.to_csv(sys.stdout, index=False, columns=['a', 'b',
'c'])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

Series also has a `to_csv` method:

```
In [51]: index = np.arange(10, 17)

In [52]: series = pd.Series(np.arange(7), index=index)

In [53]: series.to_csv('examples/series.csv')

In [54]: !cat examples/series.csv
```

```
,0
10,0
11,1
12,2
13,3
14,4
15,5
16,6
```

Working with Delimited Formats

It’s possible to load most forms of tabular data from disk using functions like `pandas.read_csv`. In some cases, however, some manual processing may be necessary. It’s not uncommon to receive a file with one or more malformed lines that trip up `read_csv`. To illustrate the basic tools, consider a small CSV file:

```
In [55]: !cat examples/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3"
```

For any file with a single-character delimiter, you can use Python’s built-in `csv` module. To use it, pass any open file or file-like object to `csv.reader`:

```
import csv
f = open('examples/ex7.csv')

reader = csv.reader(f)
```

Iterating through the reader like a file yields lists of values with any quote characters removed:

```
In [57]: for line in reader:
   ....:     print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']
```

From there, it's up to you to do the wrangling necessary to put the data in the form that you need it. Let's take this step by step. First, we read the file into a list of lines:

```
In [58]: with open('examples/ex7.csv') as f:
   ....:     lines = list(csv.reader(f))
```

Then, we split the lines into the header line and the data lines:

```
In [59]: header, values = lines[0], lines[1:]
```

Then we can create a dictionary of data columns using a dictionary comprehension and the expression `zip(*values)` (beware that this will use a lot of memory on large files), which transposes rows to columns:

```
In [60]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [61]: data_dict
Out[61]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

CSV files come in many different flavors. To define a new format with a different delimiter, string quoting convention, or line terminator, we define a simple subclass of `csv.Dialect`:

```
class my_dialect(csv.Dialect):
    lineterminator = '\n'
    delimiter = ';'
    quotechar = '"'
    quoting = csv.QUOTE_MINIMAL

reader = csv.reader(f, dialect=my_dialect)
```

We can also give individual CSV dialect parameters as keywords to `csv.reader` without having to define a subclass:

```
reader = csv.reader(f, delimiter='|')
```

The possible options (attributes of `csv.Dialect`) and what they do can be found in Table 6-3.

Table 6-3. CSV dialect options

Argument	Description
`delimiter`	One-character string to separate fields; defaults to `','`.
`lineterminator`	Line terminator for writing; defaults to `'\r\n'`. Reader ignores this and recognizes cross-platform line terminators.
`quotechar`	Quote character for fields with special characters (like a delimiter); default is `'"'`.
`quoting`	Quoting convention. Options include `csv.QUOTE_ALL` (quote all fields), `csv.QUOTE_MINIMAL` (only fields with special characters like the delimiter), `csv.QUOTE_NONNUMERIC`, and `csv.QUOTE_NONE` (no quoting). See Python's documentation for full details. Defaults to `QUOTE_MINIMAL`.
`skipinitialspace`	Ignore whitespace after each delimiter; default is `False`.
`doublequote`	How to handle quoting character inside a field; if `True`, it is doubled (see online documentation for full detail and behavior).
`escapechar`	String to escape the delimiter if `quoting` is set to `csv.QUOTE_NONE`; disabled by default.

NOTE

For files with more complicated or fixed multicharacter delimiters, you will not be able to use the `csv` module. In those cases, you'll have to do the line splitting and other cleanup using string's `split` method or the regular expression method `re.split`. Thankfully, `pandas.read_csv` is capable of doing almost anything you need if you pass the necessary options, so you only rarely will have to parse files by hand.

To *write* delimited files manually, you can use `csv.writer`. It accepts an open, writable file object and the same dialect and format options as csv.reader:

```
with open('mydata.csv', 'w') as f:
    writer = csv.writer(f, dialect=my_dialect)
    writer.writerow(('one', 'two', 'three'))
    writer.writerow(('1', '2', '3'))
    writer.writerow(('4', '5', '6'))
    writer.writerow(('7', '8', '9'))
```

JSON Data

JSON (short for JavaScript Object Notation) has become one of the standard formats for sending data by HTTP request between web browsers and other applications. It is a much more free-form data format than a tabular text form like CSV. Here is an example:

```
obj = """
{"name": "Wes",
 "cities_lived": ["Akron", "Nashville", "New York", "San
Francisco"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 34, "hobbies": ["guitars",
"soccer"]},
              {"name": "Katie", "age": 42, "hobbies": ["diving",
"art"]}]
}
"""
```

JSON is very nearly valid Python code with the exception of its null value `null` and some other nuances (such as disallowing trailing commas at the end of lists). The basic types are objects (dicts), arrays (lists), strings, numbers, booleans, and nulls. All of the keys in an object must be strings. There are several Python libraries for reading and writing JSON data. I'll use `json` here, as it is built into the Python standard library. To convert a JSON string to Python form, use `json.loads`:

```
In [63]: import json

In [64]: result = json.loads(obj)
In [65]: result
Out[65]:
{'name': 'Wes',
 'cities_lived': ['Akron', 'Nashville', 'New York', 'San
Francisco'],
 'pet': None,
 'siblings': [{'name': 'Scott',
   'age': 34,
   'hobbies': ['guitars', 'soccer']},
  {'name': 'Katie', 'age': 42, 'hobbies': ['diving', 'art']}]}
```

`json.dumps`, on the other hand, converts a Python object back to JSON:

```
In [66]: asjson = json.dumps(result)
```

How you convert a JSON object or list of objects to a DataFrame or some other data structure for analysis will be up to you. Conveniently, you can pass a list of dicts (which were previously JSON objects) to the DataFrame constructor and select a subset of the data fields:

```
In [67]: siblings = pd.DataFrame(result['siblings'], columns=
['name', 'age'])

In [68]: siblings
Out[68]:
    name  age
0  Scott   34
1  Katie   42
```

The `pandas.read_json` can automatically convert JSON datasets in specific arrangements into a Series or DataFrame. For example:

```
In [69]: !cat examples/example.json
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

The default options for `pandas.read_json` assume that each object in the JSON array is a row in the table:

```
In [70]: data = pd.read_json('examples/example.json')

In [71]: data
Out[71]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

For an extended example of reading and manipulating JSON data (including nested records), see the USDA Food Database example in [Link to Come].

If you need to export data from pandas to JSON, one way is to use the `to_json` methods on Series and DataFrame:

```
In [72]: print(data.to_json())
{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":
{"0":3,"1":6,"2":9}}

In [73]: print(data.to_json(orient='records'))
[{"a":1,"b":2,"c":3},{"a":4,"b":5,"c":6},{"a":7,"b":8,"c":9}]
```

XML and HTML: Web Scraping

Python has many libraries for reading and writing data in the ubiquitous HTML and XML formats. Examples include lxml, Beautiful Soup, and html5lib. While lxml is comparatively much faster in general, the other libraries can better handle malformed HTML or XML files.

pandas has a built-in function, `read_html`, which uses libraries like lxml and Beautiful Soup to automatically parse tables out of HTML files as DataFrame objects. To show how this works, I downloaded an HTML file (used in the pandas documentation) from the United States FDIC government agency showing bank failures.[1] First, you must install some additional libraries used by `read_html`:

```
conda install lxml
pip install beautifulsoup4 html5lib
```

If you are not using conda, `pip install lxml` should also work.

The `pandas.read_html` function has a number of options, but by default it searches for and attempts to parse all tabular data contained within `<table>` tags. The result is a list of DataFrame objects:

```
In [74]: tables =
pd.read_html('examples/fdic_failed_bank_list.html')

In [75]: len(tables)
Out[75]: 1

In [76]: failures = tables[0]

In [77]: failures.head()
Out[77]:
                      Bank Name             City  ST   CERT   \
0                   Allied Bank         Mulberry  AR     91
1  The Woodbury Banking Company         Woodbury  GA  11297
2        First CornerStone Bank  King of Prussia  PA  35312
3            Trust Company Bank          Memphis  TN   9956
4    North Milwaukee State Bank        Milwaukee  WI  20364
                 Acquiring Institution        Closing Date
Updated Date
0                         Today's Bank  September 23, 2016
November 17, 2016
1                          United Bank     August 19, 2016
November 17, 2016
2  First-Citizens Bank & Trust Company         May 6, 2016
September 6, 2016
3            The Bank of Fayette County      April 29, 2016
September 6, 2016
4  First-Citizens Bank & Trust Company      March 11, 2016
June 16, 2016
```

Because `failures` has many columns, pandas inserts a line break character `\`.

As you will learn in later chapters, from here we could proceed to do some data cleaning and analysis, like computing the number of bank failures by year:

```
In [78]: close_timestamps = pd.to_datetime(failures['Closing
Date'])

In [79]: close_timestamps.dt.year.value_counts()
Out[79]:
2010    157
2009    140
2011     92
2012     51
2008     25
       ...
2001      4
2004      4
2003      3
2007      3
2000      2
Name: Closing Date, Length: 15, dtype: int64
```

Parsing XML with lxml.objectify

XML is another common structured data format supporting hierarchical, nested data with metadata. The book you are currently reading was actually created from a series of large XML documents.

Earlier, I showed the `pandas.read_html` function, which uses either lxml or Beautiful Soup under the hood to parse data from HTML. XML and HTML are structurally similar, but XML is more general. Here, I will show an example of how to use lxml to parse data from a more general XML format.

For many years, the New York Metropolitan Transportation Authority (MTA) published a number of data series about its bus and train services in XML format. Here we'll look at the performance data, which is contained in a set of XML files. Each train or bus service has a different file (like *Performance_MNR.xml* for the Metro-North Railroad) containing monthly data as a series of XML records that look like this:

```
<INDICATOR>
  <INDICATOR_SEQ>373889</INDICATOR_SEQ>
  <PARENT_SEQ></PARENT_SEQ>
  <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
  <INDICATOR_NAME>Escalator Availability</INDICATOR_NAME>
  <DESCRIPTION>Percent of the time that escalators are
operational
  systemwide. The availability rate is based on physical
observations performed
  the morning of regular business days only. This is a new
indicator the agency
  began reporting in 2009.</DESCRIPTION>
  <PERIOD_YEAR>2011</PERIOD_YEAR>
  <PERIOD_MONTH>12</PERIOD_MONTH>
  <CATEGORY>Service Indicators</CATEGORY>
  <FREQUENCY>M</FREQUENCY>
  <DESIRED_CHANGE>U</DESIRED_CHANGE>
  <INDICATOR_UNIT>%</INDICATOR_UNIT>
  <DECIMAL_PLACES>1</DECIMAL_PLACES>
  <YTD_TARGET>97.00</YTD_TARGET>
  <YTD_ACTUAL></YTD_ACTUAL>
  <MONTHLY_TARGET>97.00</MONTHLY_TARGET>
  <MONTHLY_ACTUAL></MONTHLY_ACTUAL>
</INDICATOR>
```

Using `lxml.objectify`, we parse the file and get a reference to the root node of the XML file with `getroot`:

```python
from lxml import objectify

path = 'datasets/mta_perf/Performance_MNR.xml'
parsed = objectify.parse(open(path))
root = parsed.getroot()
```

`root.INDICATOR` returns a generator yielding each `<INDICATOR>` XML element. For each record, we can populate a dict of tag names (like `YTD_ACTUAL`) to data values (excluding a few tags):

```python
data = []

skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ',
               'DESIRED_CHANGE', 'DECIMAL_PLACES']

for elt in root.INDICATOR:
    el_data = {}
    for child in elt.getchildren():
        if child.tag in skip_fields:
            continue
        el_data[child.tag] = child.pyval
    data.append(el_data)
```

Lastly, convert this list of dicts into a DataFrame:

```
In [82]: perf = pd.DataFrame(data)

In [83]: perf.head()
Out[83]:
            AGENCY_NAME                        INDICATOR_NAME  \
0  Metro-North Railroad  On-Time Performance (West of Hudson)
1  Metro-North Railroad  On-Time Performance (West of Hudson)
2  Metro-North Railroad  On-Time Performance (West of Hudson)
3  Metro-North Railroad  On-Time Performance (West of Hudson)
4  Metro-North Railroad  On-Time Performance (West of Hudson)

DESCRIPTION  \
0  Percent of commuter trains that arrive at their destinations
within 5 m...
1  Percent of commuter trains that arrive at their destinations
within 5 m...
2  Percent of commuter trains that arrive at their destinations
within 5 m...
3  Percent of commuter trains that arrive at their destinations
within 5 m...
4  Percent of commuter trains that arrive at their destinations
within 5 m...
   PERIOD_YEAR  PERIOD_MONTH            CATEGORY FREQUENCY
INDICATOR_UNIT  \
0         2008             1  Service Indicators         M
%
1         2008             2  Service Indicators         M
%
2         2008             3  Service Indicators         M
%
3         2008             4  Service Indicators         M
%
4         2008             5  Service Indicators         M
%
  YTD_TARGET YTD_ACTUAL MONTHLY_TARGET MONTHLY_ACTUAL
0       95.0       96.9           95.0           96.9
1       95.0       96.0           95.0           95.0
2       95.0       96.3           95.0           96.9
3       95.0       96.8           95.0           98.3
4       95.0       96.6           95.0           95.8
```

XML data can get much more complicated than this example. Each tag can have metadata, too. Consider an HTML link tag, which is also valid XML:

```
from io import StringIO
tag = '<a href="http://www.google.com">Google</a>'
root = objectify.parse(StringIO(tag)).getroot()
```

You can now access any of the fields (like `href`) in the tag or the link text:

```
In [85]: root
Out[85]: <Element a at 0x7f72821a9200>

In [86]: root.get('href')
Out[86]: 'http://www.google.com'

In [87]: root.text
Out[87]: 'Google'
```

6.2 Binary Data Formats

One easy way to store data (also known as *serialization*) efficiently in binary format is using Python’s built-in `pickle` serialization. pandas objects all have a `to_pickle` method that writes the data to disk in pickle format:

```
In [88]: frame = pd.read_csv('examples/ex1.csv')

In [89]: frame
Out[89]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo

In [90]: frame.to_pickle('examples/frame_pickle')
```

You can read any “pickled” object stored in a file by using the built-in `pickle` directly, or even more conveniently using `pandas.read_pickle`:

```
In [91]: pd.read_pickle('examples/frame_pickle')
Out[91]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

CAUTION

`pickle` is only recommended as a short-term storage format. The problem is that it is hard to guarantee that the format will be stable over time; an object pickled today may not unpickle with a later version of a library. pandas has tried to maintain backward compatibility when possible, but at some point in the future it may be necessary to “break” the pickle format.

pandas has built-in support for several other open source binary data formats, such as HDF5, ORC, and Parquet. I will give some HDF5 examples in the next section, but I encourage you to explore different file formats to see how fast they are and how well they work for your analysis.

Using HDF5 Format

HDF5 is a respected file format intended for storing large quantities of scientific array data. It is available as a C library, and it has interfaces available in many other languages, including Java, Julia, MATLAB, and Python. The “HDF” in HDF5 stands for *hierarchical data format*. Each HDF5 file can store multiple datasets and supporting metadata. Compared with simpler formats, HDF5 supports on-the-fly compression with a variety of compression modes, enabling data with repeated patterns to be stored more efficiently. HDF5 can be a good choice for working with datasets that don’t fit into memory, as you can efficiently read and write small sections of much larger arrays.

To get started with HDF5 and pandas, you must first install PyTables by installing the `tables` package either with conda or pip:

```
conda install tables
```

While it’s possible to directly access HDF5 files using either the PyTables or h5py libraries, pandas provides a high-level interface that simplifies storing Series and DataFrame object. The `HDFStore` class works like a dict and handles the low-level details:

```
In [94]: frame = pd.DataFrame({'a': np.random.randn(100)})

In [95]: store = pd.HDFStore('mydata.h5')

In [96]: store['obj1'] = frame

In [97]: store['obj1_col'] = frame['a']

In [98]: store
Out[98]:
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
```

Objects contained in the HDF5 file can then be retrieved with the same dict-like API:

```
In [99]: store['obj1']
Out[99]:
           a
0  -0.204708
1   0.478943
2  -0.519439
3  -0.555730
4   1.965781
..       ...
95  0.795253
96  0.118110
97 -0.748532
98  0.584970
99  0.152677
[100 rows x 1 columns]
```

`HDFStore` supports two storage schemas, `'fixed'` and `'table'`. The latter is generally slower, but it supports query operations using a special syntax:

```
In [100]: store.put('obj2', frame, format='table')

In [101]: store.select('obj2', where=['index >= 10 and index <=
15'])
Out[101]:
           a
10  1.007189
11 -1.296221
12  0.274992
13  0.228913
14  1.352917
15  0.886429

In [102]: store.close()
```

The `put` is an explicit version of the `store['obj2'] = frame` method but allows us to set other options like the storage format.

The `pandas.read_hdf` function gives you a shortcut to these tools:

```
In [103]: frame.to_hdf('mydata.h5', 'obj3', format='table')

In [104]: pd.read_hdf('mydata.h5', 'obj3', where=['index < 5'])
Out[104]:
          a
0 -0.204708
1  0.478943
2 -0.519439
3 -0.555730
4  1.965781
```

> **NOTE**
>
> If you are processing data that is stored on remote servers, like Amazon S3 or HDFS, using a different binary format designed for distributed storage like Apache Parquet may be more suitable.

If you work with large quantities of data locally, I would encourage you to explore PyTables and h5py to see how they can suit your needs. Since many data analysis problems are I/O-bound (rather than CPU-bound), using a tool like HDF5 can massively accelerate your applications.

CAUTION

HDF5 is *not* a database. It is best suited for write-once, read-many datasets. While data can be added to a file at any time, if multiple writers do so simultaneously, the file can become corrupted.

Reading Microsoft Excel Files

pandas also supports reading tabular data stored in Excel 2003 (and higher) files using either the `ExcelFile` class or `pandas.read_excel` function. Internally these tools use the add-on packages `xlrd` and `openpyxl` to read XLS and XLSX files, respectively. These must be installed separately from pandas using pip or conda.

To use `ExcelFile`, create an instance by passing a path to an `xls` or `xlsx` file:

```
In [106]: xlsx = pd.ExcelFile('examples/ex1.xlsx')
```

Data stored in a sheet can then be read into DataFrame with `parse`:

```
In [107]: pd.read_excel(xlsx, 'Sheet1')
Out[107]:
   Unnamed: 0  a   b   c   d message
0           0  1   2   3   4   hello
1           1  5   6   7   8   world
2           2  9  10  11  12     foo
```

If you are reading multiple sheets in a file, then it is faster to create the `ExcelFile`, but you can also simply pass the filename to `pandas.read_excel`:

```
In [108]: frame = pd.read_excel('examples/ex1.xlsx', 'Sheet1')

In [109]: frame
Out[109]:
   Unnamed: 0  a   b   c   d message
0           0  1   2   3   4   hello
1           1  5   6   7   8   world
2           2  9  10  11  12     foo
```

To write pandas data to Excel format, you must first create an ExcelWriter, then write data to it using pandas objects' to_excel method:

```
In [110]: writer = pd.ExcelWriter('examples/ex2.xlsx')

In [111]: frame.to_excel(writer, 'Sheet1')

In [112]: writer.save()
```

You can also pass a file path to to_excel and avoid the ExcelWriter:

```
In [113]: frame.to_excel('examples/ex2.xlsx')
```

6.3 Interacting with Web APIs

Many websites have public APIs providing data feeds via JSON or some other format. There are a number of ways to access these APIs from Python; one method that I recommend is the requests package.

To find the last 30 GitHub issues for pandas on GitHub, we can make a GET HTTP request using the add-on requests library:

```
In [115]: import requests

In [116]: url = 'https://api.github.com/repos/pandas-
dev/pandas/issues'

In [117]: resp = requests.get(url)

In [118]: resp
Out[118]: <Response [200]>
```

The Response object's json method will return a dictionary containing JSON parsed into native Python objects:

```
In [119]: data = resp.json()

In [120]: data[0]['title']
```

```
Out[120]: 'ENH: Support timespec argument in
Timestamp.isoformat()'
```

Each element in `data` is a dictionary containing all of the data found on a GitHub issue page (except for the comments). We can pass `data` directly to DataFrame and extract fields of interest:

```
In [121]: issues = pd.DataFrame(data, columns=['number', 'title',
   .....:                                        'labels',
'state'])

In [122]: issues
Out[122]:
    number  \
0    44397
1    44396
2    44395
3    44393
4    44392
..     ...
25   44352
26   44351
27   44350
28   44349
29   44347

title  \
0                      ENH: Support timespec argument in
Timestamp.isoformat()
1                                [ArrayManager] Array version of
putmask logic
2                                            TST: parametrize
arithmetic tests
3   BUG: `date_range` ignores `nanoseconds` component of
`DateOffset` frequ...
4    BUG: `Series.map` passes `DatetimeIndex` as first argument
on some series
..
...
25  changed shape argument for ndarray from int to tuple in
./core/strings/...
26     REGR: Series.duplicated with category dtype and nulls
raises ValueError
27
BUG: groupby
```

```
28                                                     TYP: misc
typing in _libs
29                                        ENH: Use
find_stack_level universally

labels  \
0
[]
1   [{'id': 49094459, 'node_id': 'MDU6TGFiZWw0OTA5NDQ1OQ==',
'url': 'https:...
2
[]
3   [{'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url':
'https://api.g...
4   [{'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url':
'https://api.g...
..
...
25
[]
26  [{'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url':
'https://api.g...
27  [{'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url':
'https://api.g...
28  [{'id': 1280988427, 'node_id': 'MDU6TGFiZWwxMjgwOTg4NDI3',
'url': 'http...
29  [{'id': 76812, 'node_id': 'MDU6TGFiZWw3NjgxMg==', 'url':
'https://api.g...
   state
0   open
1   open
2   open
3   open
4   open
..   ...
25  open
26  open
27  open
28  open
29  open
[30 rows x 4 columns]
```

With a bit of elbow grease, you can create some higher-level interfaces to common web APIs that return DataFrame objects for easy analysis.

6.4 Interacting with Databases

In a business setting, most data may not be stored in text or Excel files. SQL-based relational databases (such as SQL Server, PostgreSQL, and MySQL) are in wide use, and many alternative databases have become quite popular. The choice of database is usually dependent on the performance, data integrity, and scalability needs of an application.

pandas has some functions to simplify loading the results of a SQL query into a DataFrame. As an example, I'll create a SQLite database using Python's built-in `sqlite3` driver:

```
In [123]: import sqlite3

In [124]: query = """
   .....: CREATE TABLE test
   .....: (a VARCHAR(20), b VARCHAR(20),
   .....:  c REAL,        d INTEGER
   .....: );"""

In [125]: con = sqlite3.connect('mydata.sqlite')

In [126]: con.execute(query)
Out[126]: <sqlite3.Cursor at 0x7f7280ab0b20>

In [127]: con.commit()
```

Then, insert a few rows of data:

```
In [128]: data = [('Atlanta', 'Georgia', 1.25, 6),
   .....:         ('Tallahassee', 'Florida', 2.6, 3),
   .....:         ('Sacramento', 'California', 1.7, 5)]

In [129]: stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

In [130]: con.executemany(stmt, data)
Out[130]: <sqlite3.Cursor at 0x7f7280b57f10>

In [131]: con.commit()
```

Most Python SQL drivers (PyODBC, psycopg2, MySQLdb, turbodbc, etc.) return a list of tuples when selecting data from a table:

```
In [132]: cursor = con.execute('select * from test')

In [133]: rows = cursor.fetchall()

In [134]: rows
Out[134]:
[('Atlanta', 'Georgia', 1.25, 6),
 ('Tallahassee', 'Florida', 2.6, 3),
 ('Sacramento', 'California', 1.7, 5)]
```

You can pass the list of tuples to the DataFrame constructor, but you also need the column names, contained in the cursor's `description` attribute:

```
In [135]: cursor.description
Out[135]:
(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))

In [136]: pd.DataFrame(rows, columns=[x[0] for x in
cursor.description])
Out[136]:
             a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

This is quite a bit of munging that you'd rather not repeat each time you query the database. The SQLAlchemy project is a popular Python SQL toolkit that abstracts away many of the common differences between SQL databases. pandas has a `read_sql` function that enables you to read data easily from a general SQLAlchemy connection. Here, we'll connect to the same SQLite database with SQLAlchemy and read data from the table created before:

```
In [137]: import sqlalchemy as sqla

In [138]: db = sqla.create_engine('sqlite:///mydata.sqlite')

In [139]: pd.read_sql('select * from test', db)
Out[139]:
```

```
             a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

6.5 Conclusion

Getting access to data is frequently the first step in the data analysis process. We have looked at a number of useful tools in this chapter that should help you get started. In the upcoming chapters we will dig deeper into data wrangling, data visualization, time series analysis, and other topics.

[1] For the full list, see *https://www.fdic.gov/bank/individual/failed/banklist.html*.

Appendix A. Advanced NumPy

In this appendix, I will go deeper into the NumPy library for array computing. This will include more internal details about the ndarray type and more advanced array manipulations and algorithms.

This appendix contains miscellaneous topics and does not necessarily need to be read linearly.

A.1 ndarray Object Internals

The NumPy ndarray provides a way to interpret a block of homogeneously-typed data (either contiguous or strided) as a multidimensional array object. The data type, or *dtype*, determines how the data is interpreted as being floating point, integer, boolean, or any of the other types we've been looking at.

Part of what makes ndarray flexible is that every array object is a *strided* view on a block of data. You might wonder, for example, how the array view `arr[::2, ::-1]` does not copy any data. The reason is that the ndarray is more than just a chunk of memory and a dtype; it also has "striding" information that enables the array to move through memory with varying step sizes. More precisely, the ndarray internally consists of the following:

- A *pointer to data*—that is, a block of data in RAM or in a memory-mapped file
- The *data type* or dtype, describing fixed-size value cells in the array
- A tuple indicating the array's *shape*
- A tuple of *strides*, integers indicating the number of bytes to "step" in order to advance one element along a dimension

See Figure A-1 for a simple mockup of the ndarray innards.

For example, a 10 × 5 array would have shape `(10, 5)`:

```
In [12]: np.ones((10, 5)).shape
Out[12]: (10, 5)
```

A typical (C order) 3 × 4 × 5 array of `float64` (8-byte) values has strides `(160, 40, 8)` (knowing about the strides can be useful because, in general, the larger the strides on a particular axis, the more costly it is to perform computation along that axis):

```
In [13]: np.ones((3, 4, 5), dtype=np.float64).strides
Out[13]: (160, 40, 8)
```

While it is rare that a typical NumPy user would be interested in the array strides, they are needed to construct "zero-copy" array views. Strides can even be negative, which enables an array to move "backward" through memory (this would be the case, for example, in a slice like `obj[::-1]` or `obj[:, ::-1]`).

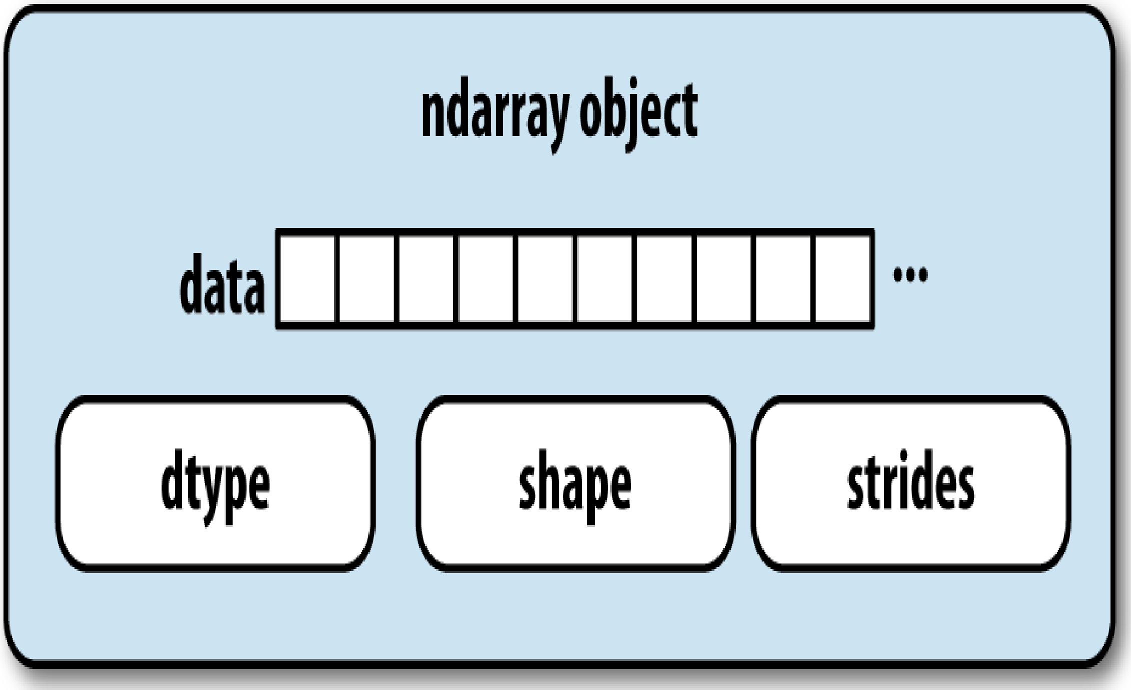


Figure A-1. The NumPy ndarray object

NumPy dtype Hierarchy

You may occasionally have code that needs to check whether an array contains integers, floating-point numbers, strings, or Python objects. Because there are multiple types of floating-point numbers (`float16` through `float128`), checking that the dtype is among a list of types would be very verbose. Fortunately, the dtypes have superclasses such as `np.integer` and `np.floating`, which can be used in conjunction with the `np.issubdtype` function:

```
In [14]: ints = np.ones(10, dtype=np.uint16)

In [15]: floats = np.ones(10, dtype=np.float32)

In [16]: np.issubdtype(ints.dtype, np.integer)
Out[16]: True

In [17]: np.issubdtype(floats.dtype, np.floating)
Out[17]: True
```

You can see all of the parent classes of a specific dtype by calling the type's `mro` method:

```
In [18]: np.float64.mro()
Out[18]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
 numpy.generic,
 float,
 object]
```

Therefore, we also have:

```
In [19]: np.issubdtype(ints.dtype, np.number)
Out[19]: True
```

Most NumPy users will never have to know about this, but it is occasionally useful. See Figure A-2 for a graph of the dtype hierarchy and parent–subclass relationships.[1]

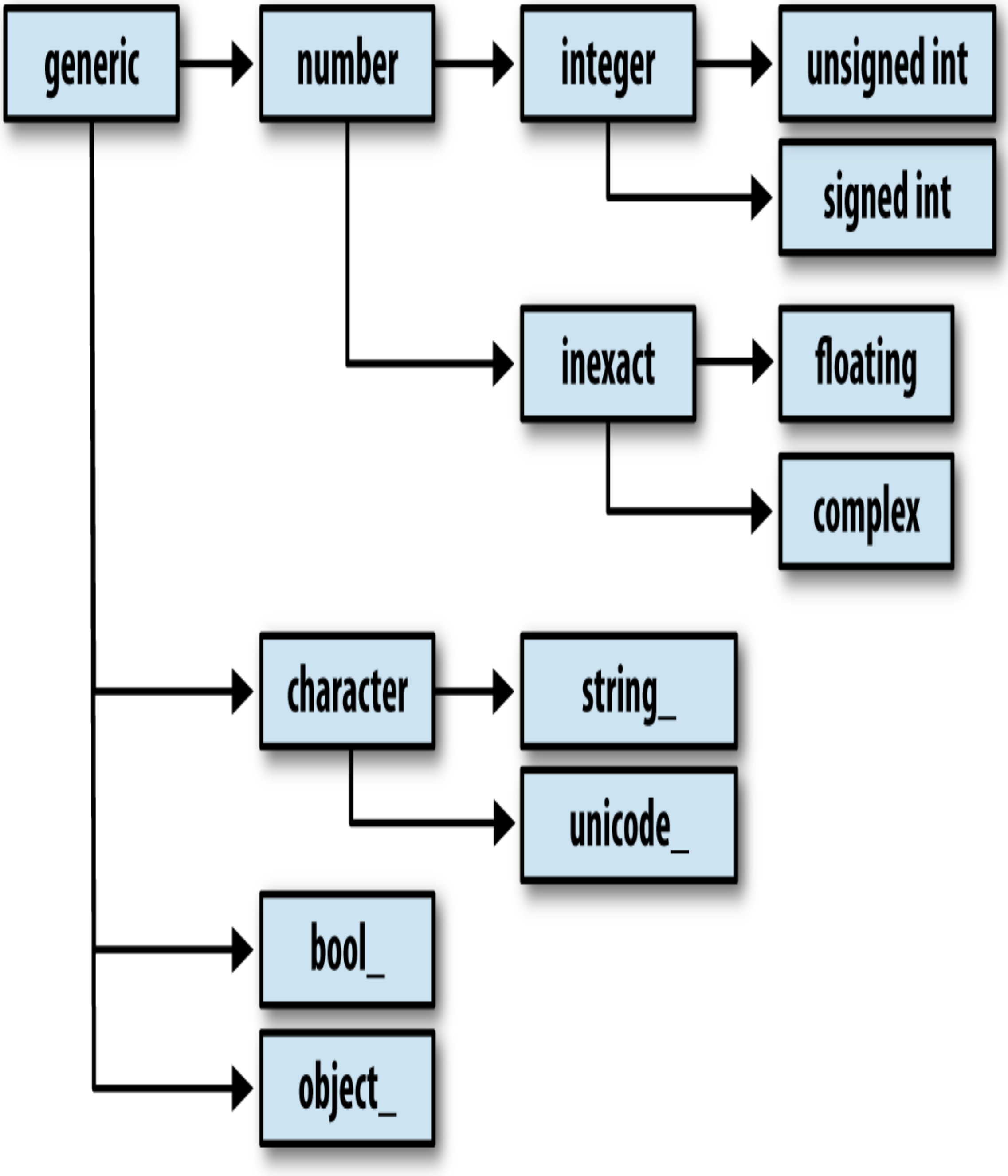


Figure A-2. The NumPy dtype class hierarchy

A.2 Advanced Array Manipulation

There are many ways to work with arrays beyond fancy indexing, slicing, and boolean subsetting. While much of the heavy lifting for data analysis applications is handled by higher-level functions in pandas, you may at some point need to write a data algorithm that is not found in one of the existing libraries.

Reshaping Arrays

In many cases, you can convert an array from one shape to another without copying any data. To do this, pass a tuple indicating the new shape to the `reshape` array instance method. For example, suppose we had a one-dimensional array of values that we wished to rearrange into a matrix (this is illustrated in Figure A-3):

```
In [20]: arr = np.arange(8)

In [21]: arr
Out[21]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [22]: arr.reshape((4, 2))
Out[22]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

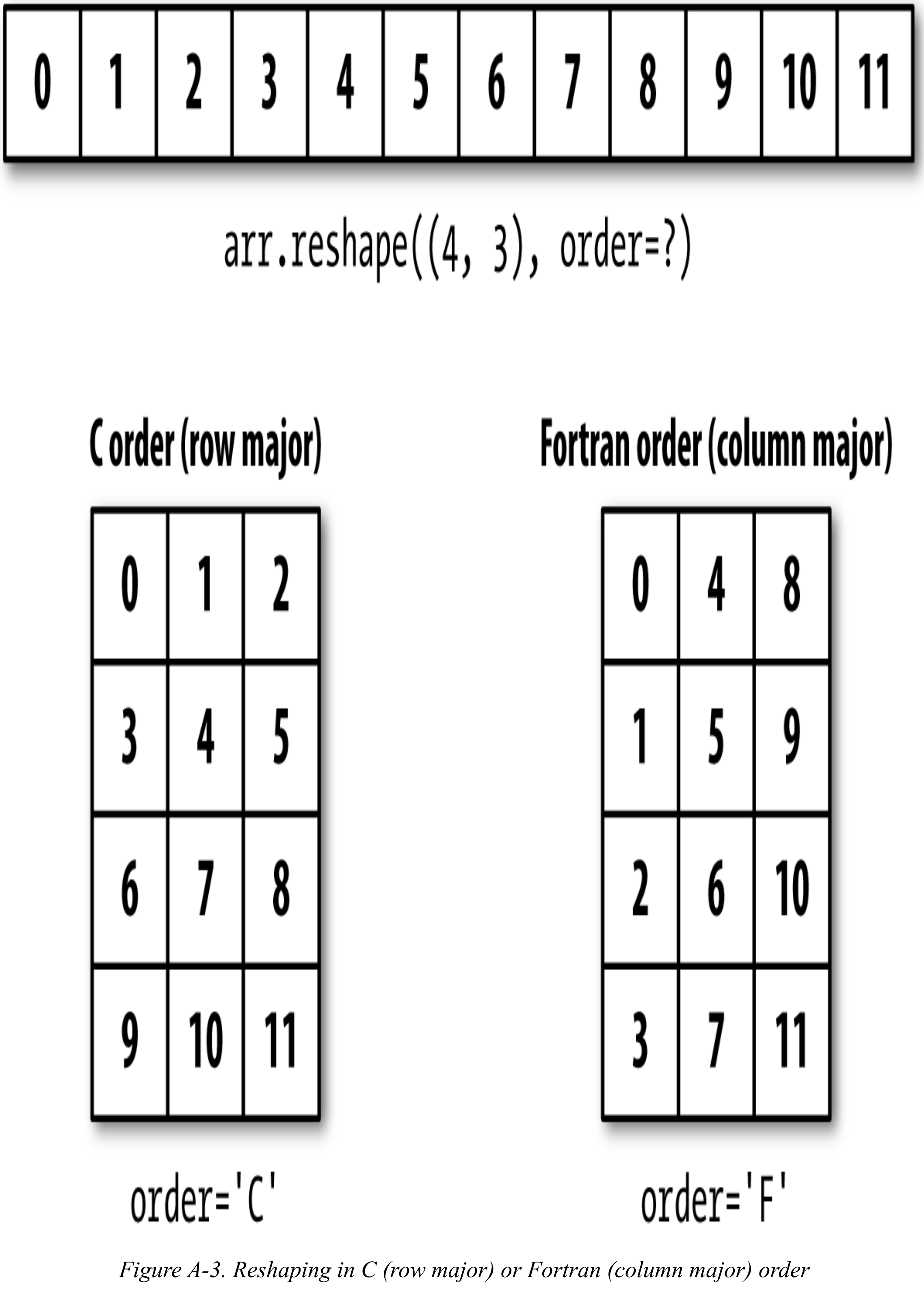


Figure A-3. Reshaping in C (row major) or Fortran (column major) order

A multidimensional array can also be reshaped:

```
In [23]: arr.reshape((4, 2)).reshape((2, 4))
Out[23]:
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 [24]: arr = np.arange(15)

In [25]: arr.reshape((5, -1))
Out[25]:
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 [26]: other_arr = np.ones((3, 5))

In [27]: other_arr.shape
Out[27]: (3, 5)

In [28]: arr.reshape(other_arr.shape)
Out[28]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

The opposite operation of `reshape` from one-dimensional to a higher dimension is typically known as *flattening* or *raveling*:

```
In [29]: arr = np.arange(15).reshape((5, 3))

In [30]: arr
Out[30]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])

In [31]: arr.ravel()
```

```
Out[31]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12,
13, 14])
```

`ravel` does not produce a copy of the underlying values if the values in the result were contiguous in the original array. The `flatten` method behaves like `ravel` except it always returns a copy of the data:

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

The data can be reshaped or raveled in different orders. This is a slightly nuanced topic for new NumPy users and is therefore the next subtopic.

C Versus Fortran Order

NumPy is able to adapt to many different layouts of your data in memory. By default, NumPy arrays are created in *row major* order. Spatially this means that if you have a two-dimensional array of data, the items in each row of the array are stored in adjacent memory locations. The alternative to row major ordering is *column major* order, which means that values within each column of data are stored in adjacent memory locations.

For historical reasons, row and column major order are also know as C and Fortran order, respectively. In the FORTRAN 77 language, matrices are all column major.

Functions like `reshape` and `ravel` accept an `order` argument indicating the order to use the data in the array. This is usually set to `'C'` or `'F'` in most cases (there are also less commonly used options `'A'` and `'K'`; see the NumPy documentation, and refer back to Figure A-3 for an illustration of these options):

```
In [33]: arr = np.arange(12).reshape((3, 4))

In [34]: arr
Out[34]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
In [35]: arr.ravel()
Out[35]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

In [36]: arr.ravel('F')
Out[36]: array([ 0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7, 11])
```

Reshaping arrays with more than two dimensions can be a bit mind-bending (see Figure A-3). The key difference between C and Fortran order is the way in which the dimensions are walked:

C/row major order

Traverse higher dimensions *first* (e.g., axis 1 before advancing on axis 0).

Fortran/column major order

Traverse higher dimensions *last* (e.g., axis 0 before advancing on axis 1).

Concatenating and Splitting Arrays

`numpy.concatenate` takes a sequence (tuple, list, etc.) of arrays and joins them together in order along the input axis:

```
In [37]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])

In [38]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])

In [39]: np.concatenate([arr1, arr2], axis=0)
Out[39]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [40]: np.concatenate([arr1, arr2], axis=1)
Out[40]:
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

There are some convenience functions, like `vstack` and `hstack`, for common kinds of concatenation. The preceding operations could have been expressed as:

```
In [41]: np.vstack((arr1, arr2))
Out[41]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [42]: np.hstack((arr1, arr2))
Out[42]:
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 [43]: arr = np.random.randn(5, 2)

In [44]: arr
Out[44]:
array([[-0.2047,  0.4789],
       [-0.5194, -0.5557],
       [ 1.9658,  1.3934],
       [ 0.0929,  0.2817],
       [ 0.769 ,  1.2464]])

In [45]: first, second, third = np.split(arr, [1, 3])

In [46]: first
Out[46]: array([[-0.2047,  0.4789]])

In [47]: second
Out[47]:
array([[-0.5194, -0.5557],
       [ 1.9658,  1.3934]])

In [48]: third
Out[48]:
array([[0.0929, 0.2817],
       [0.769 , 1.2464]])
```

The value `[1, 3]` passed to `np.split` indicate the indices at which to split the array into pieces.

See Table A-1 for a list of all relevant concatenation and splitting functions, some of which are provided only as a convenience of the very general-purpose `concatenate`.

Table A-1. Array concatenation functions

Function	Description
`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`	Convenience functions for splitting on axis 0 and 1, respectively

Stacking helpers: r_ and c_

There are two special objects in the NumPy namespace, `r_` and `c_`, that make stacking arrays more concise:

```
In [49]: arr = np.arange(6)

In [50]: arr1 = arr.reshape((3, 2))

In [51]: arr2 = np.random.randn(3, 2)

In [52]: np.r_[arr1, arr2]
Out[52]:
array([[ 0.    ,  1.    ],
       [ 2.    ,  3.    ],
       [ 4.    ,  5.    ],
       [ 1.0072, -1.2962],
       [ 0.275 ,  0.2289],
       [ 1.3529,  0.8864]])

In [53]: np.c_[np.r_[arr1, arr2], arr]
Out[53]:
array([[ 0.    ,  1.    ,  0.    ],
       [ 2.    ,  3.    ,  1.    ],
       [ 4.    ,  5.    ,  2.    ],
       [ 1.0072, -1.2962,  3.    ],
       [ 0.275 ,  0.2289,  4.    ],
       [ 1.3529,  0.8864,  5.    ]])
```

These additionally can translate slices to arrays:

```
In [54]: np.c_[1:6, -10:-5]
Out[54]:
array([[  1, -10],
       [  2,  -9],
       [  3,  -8],
       [  4,  -7],
       [  5,  -6]])
```

See the docstring for more on what you can do with `c_` and `r_`.

Repeating Elements: tile and repeat

Two useful tools for repeating or replicating arrays to produce larger arrays are the `repeat` and `tile` functions. `repeat` replicates each element in an array some number of times, producing a larger array:

```
In [55]: arr = np.arange(3)

In [56]: arr
Out[56]: array([0, 1, 2])

In [57]: arr.repeat(3)
Out[57]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

> **NOTE**
>
> The need to replicate or repeat arrays can be less common with NumPy than it is with other array programming frameworks like MATLAB. One reason for this is that *broadcasting* often fills this need better, which is the subject of the next section.

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 [58]: arr.repeat([2, 3, 4])
Out[58]: array([0, 0, 1, 1, 1, 2, 2, 2, 2])
```

Multidimensional arrays can have their elements repeated along a particular axis.

```
In [59]: arr = np.random.randn(2, 2)

In [60]: arr
Out[60]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [61]: arr.repeat(2, axis=0)
Out[61]:
array([[-2.0016, -0.3718],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386]])
```

Note that if no axis is passed, the array will be flattened first, which is likely not what you want. Similarly, you can pass an array of integers when repeating a multidimensional array to repeat a given slice a different number of times:

```
In [62]: arr.repeat([2, 3], axis=0)
Out[62]:
array([[-2.0016, -0.3718],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386]])

In [63]: arr.repeat([2, 3], axis=1)
Out[63]:
array([[-2.0016, -2.0016, -0.3718, -0.3718, -0.3718],
       [ 1.669 ,  1.669 , -0.4386, -0.4386, -0.4386]])
```

`tile`, on the other hand, is a shortcut for stacking copies of an array along an axis. Visually you can think of it as being akin to “laying down tiles”:

```
In [64]: arr
Out[64]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [65]: np.tile(arr, 2)
Out[65]:
```

```
array([[-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386]])
```

The second argument is the number of tiles; with a scalar, the tiling is made row by row, rather than column by column. The second argument to `tile` can be a tuple indicating the layout of the "tiling":

```
In [66]: arr
Out[66]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [67]: np.tile(arr, (2, 1))
Out[67]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [68]: np.tile(arr, (3, 2))
Out[68]:
array([[-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386],
       [-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386],
       [-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386]])
```

Fancy Indexing Equivalents: take and put

As you may recall from Chapter 4, one way to get and set subsets of arrays is by *fancy* indexing using integer arrays:

```
In [69]: arr = np.arange(10) * 100

In [70]: inds = [7, 1, 2, 6]

In [71]: arr[inds]
Out[71]: array([700, 100, 200, 600])
```

There are alternative ndarray methods that are useful in the special case of only making a selection on a single axis:

```
In [72]: arr.take(inds)
Out[72]: array([700, 100, 200, 600])

In [73]: arr.put(inds, 42)

In [74]: arr
Out[74]: array([  0,  42,  42, 300, 400, 500,  42,  42, 800, 900])

In [75]: arr.put(inds, [40, 41, 42, 43])

In [76]: arr
Out[76]: array([  0,  41,  42, 300, 400, 500,  43,  40, 800, 900])
```

To use `take` along other axes, you can pass the `axis` keyword:

```
In [77]: inds = [2, 0, 2, 1]

In [78]: arr = np.random.randn(2, 4)

In [79]: arr
Out[79]:
array([[-0.5397,  0.477 ,  3.2489, -1.0212],
       [-0.5771,  0.1241,  0.3026,  0.5238]])

In [80]: arr.take(inds, axis=1)
Out[80]:
array([[ 3.2489, -0.5397,  3.2489,  0.477 ],
       [ 0.3026, -0.5771,  0.3026,  0.1241]])
```

`put` does not accept an `axis` argument but rather indexes into the flattened (one-dimensional, C order) version of the array. Thus, when you need to set elements using an index array on other axes, it is best to use `[]`-based indexing.

A.3 Broadcasting

Broadcasting governs how operations work between arrays of different shapes. It can be a powerful feature, but one that can cause confusion, even for experienced users. The simplest example of broadcasting occurs when combining a scalar value with an array:

```
In [81]: arr = np.arange(5)
In [82]: arr
Out[82]: array([0, 1, 2, 3, 4])

In [83]: arr * 4
Out[83]: array([ 0,  4,  8, 12, 16])
```

Here we say that the scalar value 4 has been *broadcast* to all of the other elements in the multiplication operation.

For example, we can demean each column of an array by subtracting the column means. In this case, it is necessary only to subtract an array containing the mean of each column:

```
In [84]: arr = np.random.randn(4, 3)

In [85]: arr.mean(0)
Out[85]: array([-0.3928, -0.3824, -0.8768])

In [86]: demeaned = arr - arr.mean(0)

In [87]: demeaned
Out[87]:
array([[ 0.3937,  1.7263,  0.1633],
       [-0.4384, -1.9878, -0.9839],
       [-0.468 ,  0.9426, -0.3891],
       [ 0.5126, -0.6811,  1.2097]])

In [88]: demeaned.mean(0)
Out[88]: array([-0.,  0., -0.])
```

See Figure A-4 for an illustration of this operation. Demeaning the rows as a broadcast operation requires a bit more care. Fortunately, broadcasting potentially lower dimensional values across any dimension of an array (like subtracting the row means from each column of a two-dimensional array) is possible as long as you follow the rules.

This brings us to:

THE BROADCASTING RULE

Two arrays are compatible for broadcasting if for each *trailing dimension* (i.e., starting from the end) the axis lengths match or if either of the lengths is 1. Broadcasting is then performed over the missing or length 1 dimensions.

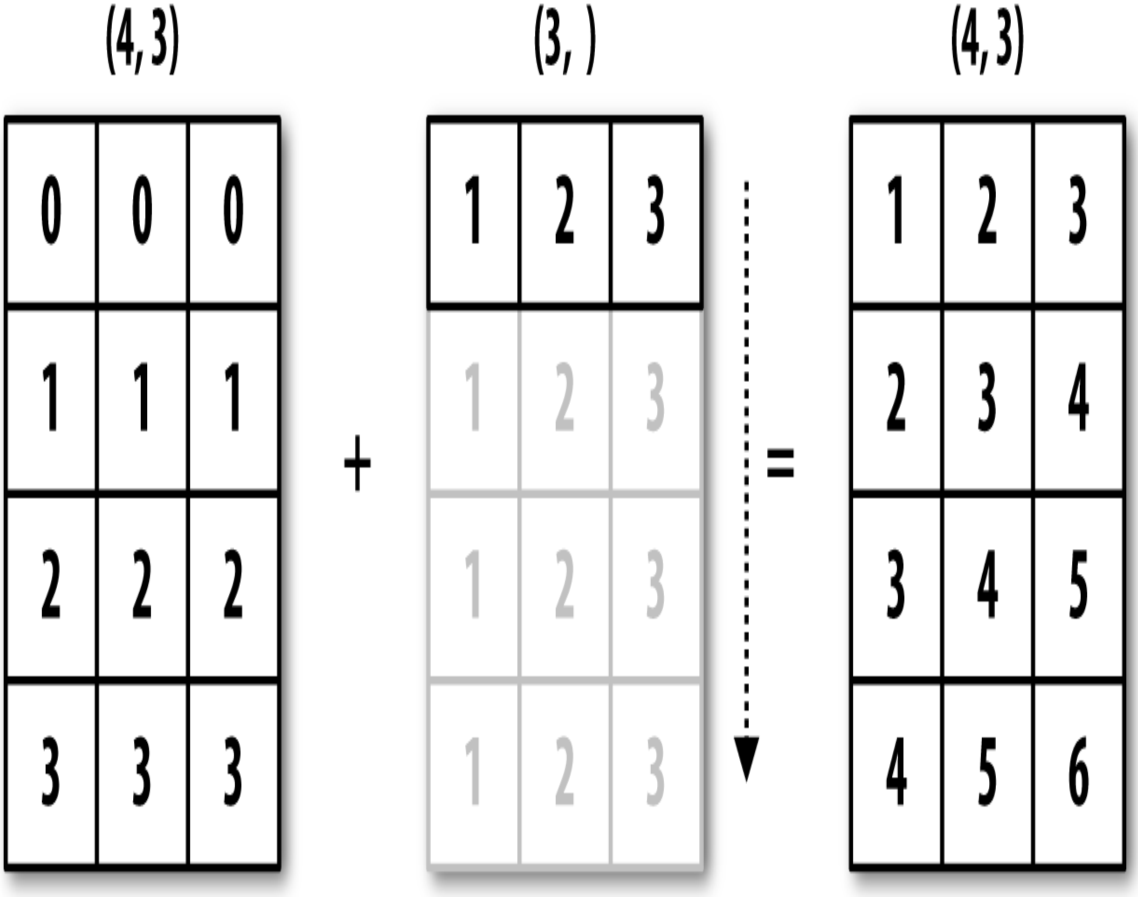


Figure A-4. Broadcasting over axis 0 with a 1D array

Even as an experienced NumPy user, I often find myself having to pause and draw a diagram as I think about the broadcasting rule. Consider the last example and suppose we wished instead to subtract the mean value from each row. Since `arr.mean(0)` has length 3, it is compatible for broadcasting across axis 0 because the trailing dimension in `arr` is 3 and therefore matches. According to the rules, to subtract over axis 1 (i.e., subtract the row mean from each row), the smaller array must have shape `(4, 1)`:

```
In [89]: arr
Out[89]:
array([[ 0.0009,  1.3438, -0.7135],
       [-0.8312, -2.3702, -1.8608],
       [-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329]])

In [90]: row_means = arr.mean(1)

In [91]: row_means.shape
Out[91]: (4,)

In [92]: row_means.reshape((4, 1))
Out[92]:
array([[ 0.2104],
       [-1.6874],
       [-0.5222],
       [-0.2036]])

In [93]: demeaned = arr - row_means.reshape((4, 1))

In [94]: demeaned.mean(1)
Out[94]: array([ 0., -0.,  0.,  0.])
```

See Figure A-5 for an illustration of this operation.

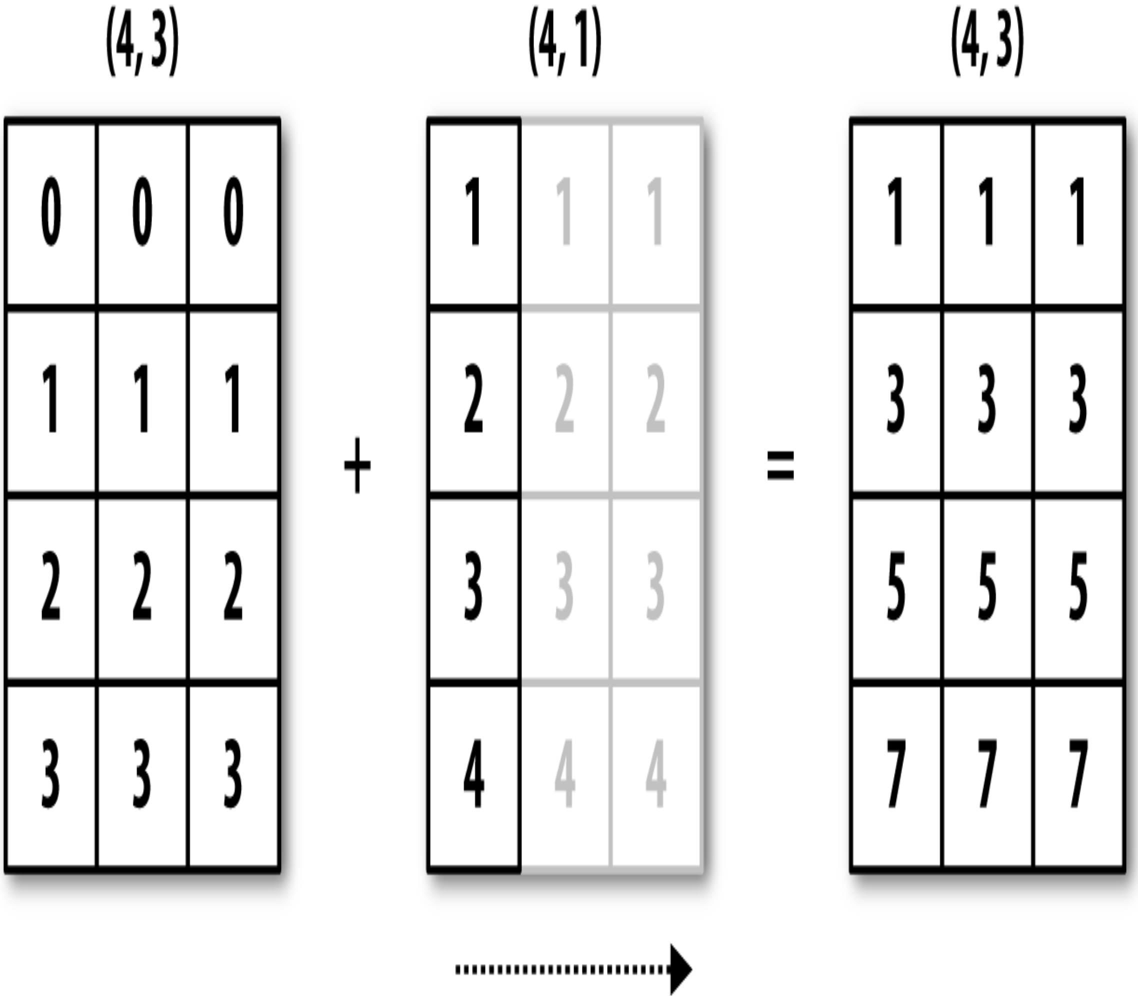


Figure A-5. Broadcasting over axis 1 of a 2D array

See Figure A-6 for another illustration, this time adding a two-dimensional array to a three-dimensional one across axis 0.

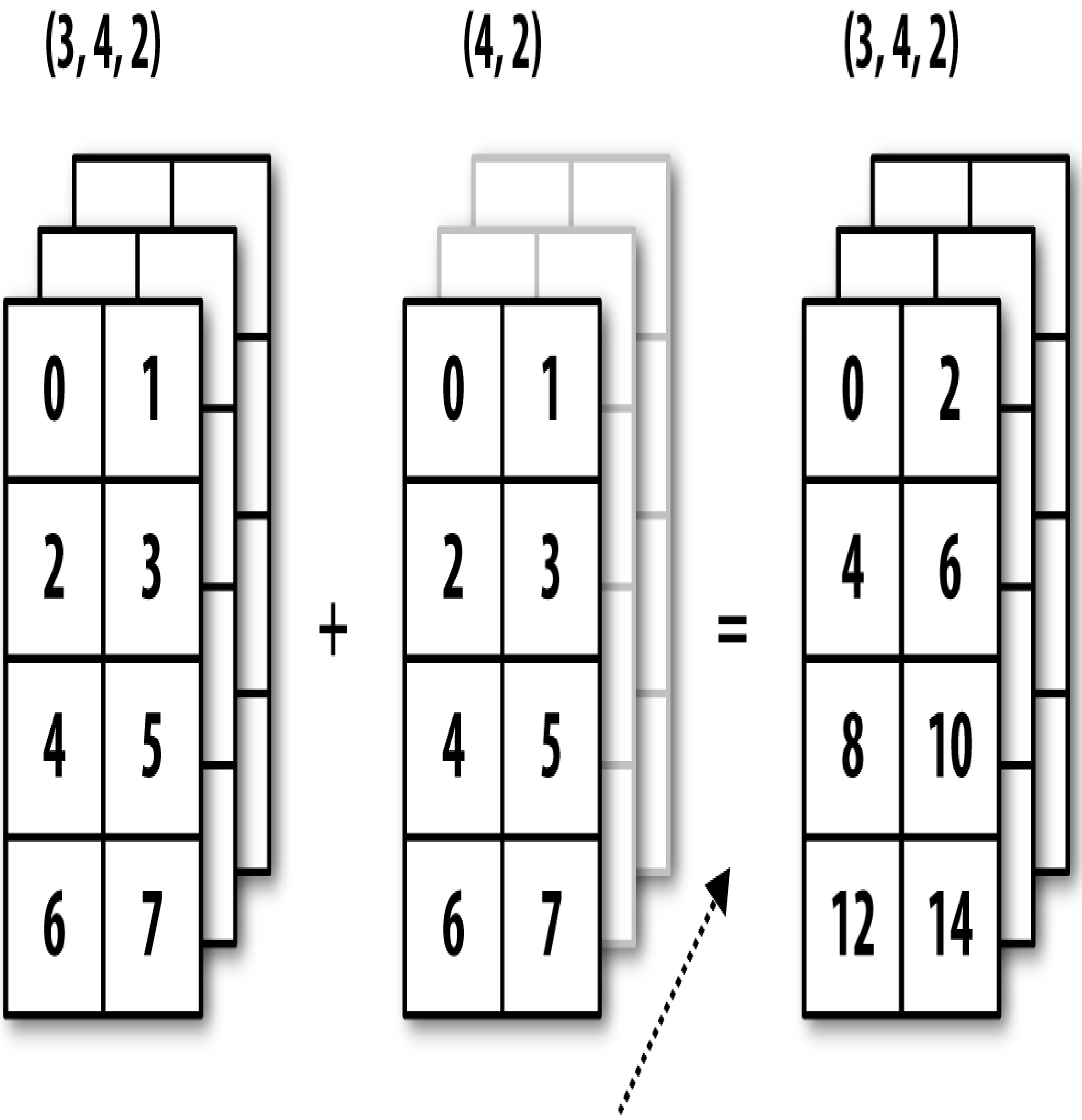


Figure A-6. Broadcasting over axis 0 of a 3D array

Broadcasting Over Other Axes

Broadcasting with higher dimensional arrays can seem even more mind-bending, but it is really a matter of following the rules. If you don't, you'll get an error like this:

```
In [95]: arr - arr.mean(1)
---------------------------------------------------------------------
-------
ValueError                                Traceback (most recent
call last)
```

```
<ipython-input-95-8b8ada26fac0> in <module>
----> 1 arr - arr.mean(1)
ValueError: operands could not be broadcast together with shapes
(4,3) (4,)
```

It’s quite common to want to perform an arithmetic operation with a lower dimensional array across axes other than axis 0. According to the broadcasting rule, the “broadcast dimensions” must be 1 in the smaller array. In the example of row demeaning shown here, this meant reshaping the row means to be shape `(4, 1)` instead of `(4,)`:

```
In [96]: arr - arr.mean(1).reshape((4, 1))
Out[96]:
array([[-0.2095,  1.1334, -0.9239],
       [ 0.8562, -0.6828, -0.1734],
       [-0.3386,  1.0823, -0.7438],
       [ 0.3234, -0.8599,  0.5365]])
```

In the three-dimensional case, broadcasting over any of the three dimensions is only a matter of reshaping the data to be shape-compatible. Figure A-7 nicely visualizes the shapes required to broadcast over each axis of a three-dimensional array.

A common problem, therefore, is needing to add a new axis with length 1 specifically for broadcasting purposes. Using `reshape` is one option, but inserting an axis requires constructing a tuple indicating the new shape. This can often be a tedious exercise. Thus, NumPy arrays offer a special syntax for inserting new axes by indexing. We use the special `np.newaxis` attribute along with “full” slices to insert the new axis:

```
In [97]: arr = np.zeros((4, 4))

In [98]: arr_3d = arr[:, np.newaxis, :]

In [99]: arr_3d.shape
Out[99]: (4, 1, 4)

In [100]: arr_1d = np.random.normal(size=3)

In [101]: arr_1d[:, np.newaxis]
Out[101]:
array([[-2.3594],
       [-0.1995],
       [-1.542 ]])

In [102]: arr_1d[np.newaxis, :]
Out[102]: array([[-2.3594, -0.1995, -1.542 ]])
```

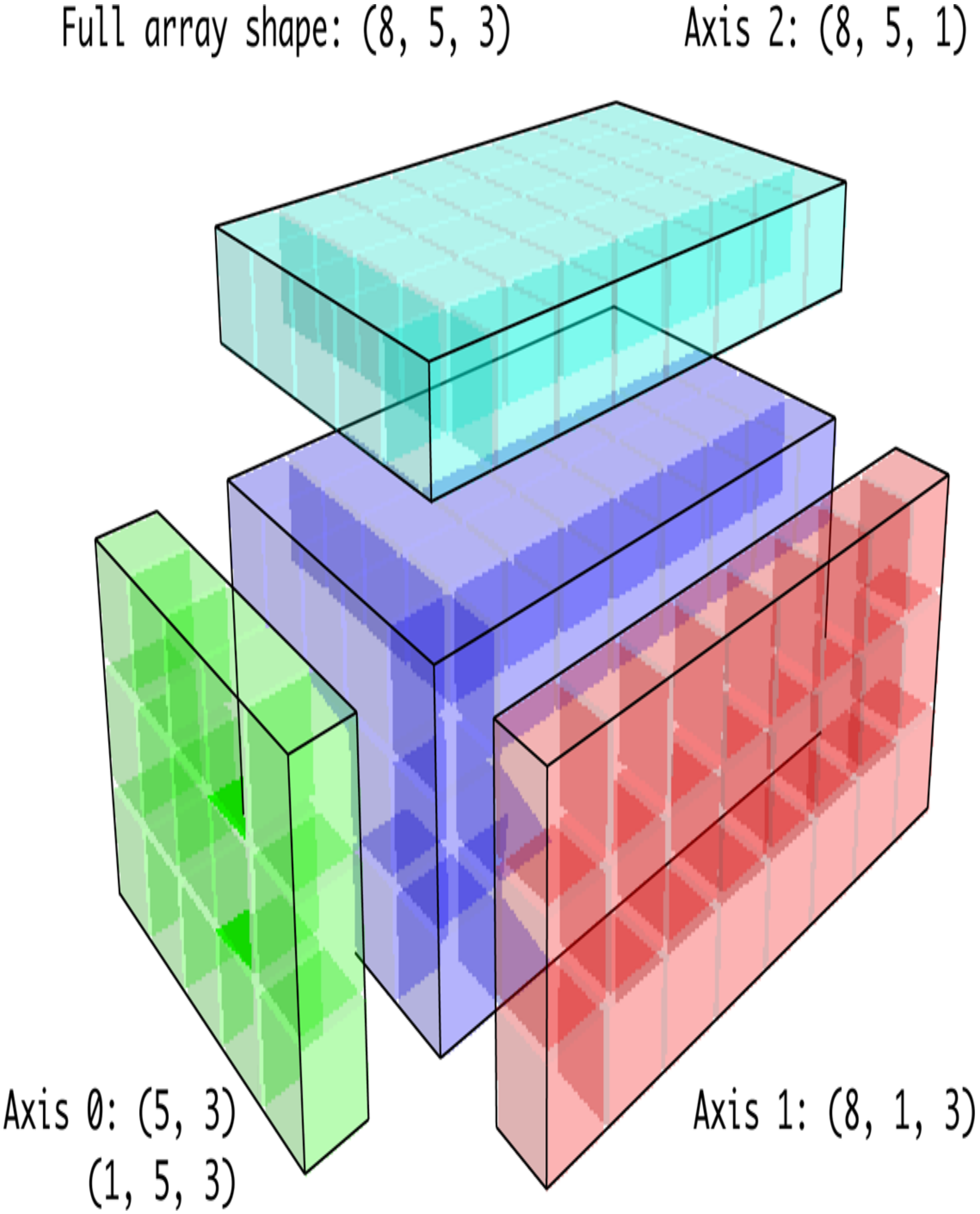


Figure A-7. Compatible 2D array shapes for broadcasting over a 3D array

Thus, if we had a three-dimensional array and wanted to demean axis 2, we would need to write:

```
In [103]: arr = np.random.randn(3, 4, 5)

In [104]: depth_means = arr.mean(2)

In [105]: depth_means
Out[105]:
array([[-0.4735,  0.3971, -0.0228,  0.2001],
       [-0.3521, -0.281 , -0.071 , -0.1586],
       [ 0.6245,  0.6047,  0.4396, -0.2846]])

In [106]: depth_means.shape
Out[106]: (3, 4)

In [107]: demeaned = arr - depth_means[:, :, np.newaxis]

In [108]: demeaned.mean(2)
Out[108]:
array([[ 0.,  0., -0., -0.],
       [ 0.,  0., -0.,  0.],
       [ 0.,  0., -0., -0.]])
```

You might be wondering if there's a way to generalize demeaning over an axis without sacrificing performance. There is, but it requires some indexing gymnastics:

```
def demean_axis(arr, axis=0):
    means = arr.mean(axis)

    # This generalizes things like [:, :, np.newaxis] to N
dimensions
    indexer = [slice(None)] * arr.ndim
    indexer[axis] = np.newaxis
    return arr - means[indexer]
```

Setting Array Values by Broadcasting

The same broadcasting rule governing arithmetic operations also applies to setting values via array indexing. In a simple case, we can do things like:

```
In [109]: arr = np.zeros((4, 3))

In [110]: arr[:] = 5
In [111]: arr
Out[111]:
array([[5., 5., 5.],
       [5., 5., 5.],
       [5., 5., 5.],
       [5., 5., 5.]])
```

However, if we had a one-dimensional array of values we wanted to set into the columns of the array, we can do that as long as the shape is compatible:

```
In [112]: col = np.array([1.28, -0.42, 0.44, 1.6])

In [113]: arr[:] = col[:, np.newaxis]

In [114]: arr
Out[114]:
array([[ 1.28,  1.28,  1.28],
       [-0.42, -0.42, -0.42],
       [ 0.44,  0.44,  0.44],
       [ 1.6 ,  1.6 ,  1.6 ]])

In [115]: arr[:2] = [[-1.37], [0.509]]

In [116]: arr
Out[116]:
array([[-1.37 , -1.37 , -1.37 ],
       [ 0.509,  0.509,  0.509],
       [ 0.44 ,  0.44 ,  0.44 ],
       [ 1.6  ,  1.6  ,  1.6  ]])
```

A.4 Advanced ufunc Usage

While many NumPy users will only make use of the fast element-wise operations provided by the universal functions, there are a number of additional features that occasionally can help you write more concise code without explicit loops.

ufunc Instance Methods

Each of NumPy’s binary ufuncs has special methods for performing certain kinds of special vectorized operations. These are summarized in Table A-2, but I’ll give a few concrete examples to illustrate how they work.

`reduce` takes a single array and aggregates its values, optionally along an axis, by performing a sequence of binary operations. For example, an alternative way to sum elements in an array is to use `np.add.reduce`:

```
In [117]: arr = np.arange(10)

In [118]: np.add.reduce(arr)
Out[118]: 45

In [119]: arr.sum()
Out[119]: 45
```

The starting value (for example, 0 for `add`) depends on the ufunc. If an axis is passed, the reduction is performed along that axis. This allows you to answer certain kinds of questions in a concise way. As a less mundane example, we can use `np.logical_and` to check whether the values in each row of an array are sorted:

```
In [120]: np.random.seed(12346)  # for reproducibility

In [121]: arr = np.random.randn(5, 5)

In [122]: arr[::2].sort(1) # sort a few rows

In [123]: arr[:, :-1] < arr[:, 1:]
Out[123]:
array([[ True,  True,  True,  True],
       [False,  True, False, False],
       [ True,  True,  True,  True],
       [ True, False,  True,  True],
       [ True,  True,  True,  True]])

In [124]: np.logical_and.reduce(arr[:, :-1] < arr[:, 1:], axis=1)
Out[124]: array([ True, False,  True, False,  True])
```

Note that `logical_and.reduce` is equivalent to the `all` method.

The `accumulate` ufunc method is related to `reduce` like `cumsum` is related to `sum`. It produces an array of the same size with the intermediate “accumulated” values:

```
In [125]: arr = np.arange(15).reshape((3, 5))
In [126]: np.add.accumulate(arr, axis=1)
Out[126]:
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 [127]: arr = np.arange(3).repeat([1, 2, 2])

In [128]: arr
Out[128]: array([0, 1, 1, 2, 2])

In [129]: np.multiply.outer(arr, np.arange(5))
Out[129]:
array([[0, 0, 0, 0, 0],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8],
       [0, 2, 4, 6, 8]])
```

The output of `outer` will have a dimension that is the concatenation of the dimensions of the inputs:

```
In [130]: x, y = np.random.randn(3, 4), np.random.randn(5)

In [131]: result = np.subtract.outer(x, y)

In [132]: result.shape
Out[132]: (3, 4, 5)
```

The last method, `reduceat`, performs a “local reduce,” in essence an array “group by” operation in which slices of the array are aggregated together. It accepts a sequence of “bin edges” that indicate how to split and aggregate the values:

```
In [133]: arr = np.arange(10)

In [134]: np.add.reduceat(arr, [0, 5, 8])
Out[134]: array([10, 18, 17])
```

The results are the reductions (here, sums) performed over `arr[0:5]`, `arr[5:8]`, and `arr[8:]`. As with the other methods, you can pass an axis argument:

```
In [135]: arr = np.multiply.outer(np.arange(4), np.arange(5))

In [136]: arr
Out[136]:
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  3,  6,  9, 12]])

In [137]: np.add.reduceat(arr, [0, 2, 4], axis=1)
Out[137]:
array([[ 0,  0,  0],
       [ 1,  5,  4],
       [ 2, 10,  8],
       [ 3, 15, 12]])
```

See Table A-2 for a partial listing of ufunc methods.

Table A-2. ufunc methods

Method	Description
`accumulate(x)`	Aggregate values, preserving all partial aggregates
`at(x, indices, b=None)`	Perform operation in place on `x` at the specified indices. The argument `b` is the second input to ufuncs that require two array inputs.
`reduce(x)`	Aggregate values by successive applications of the operation
`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`; the resulting array has shape `x.shape + y.shape`

Writing New ufuncs in Python

There are a number of ways to create your own NumPy ufuncs. The most general is to use the NumPy C API, but that is beyond the scope of this book. In this section, we will look at pure Python ufuncs.

numpy.frompyfunc accepts a Python function along with a specification for the number of inputs and outputs. For example, a simple function that adds element-wise would be specified as:

```
In [138]: def add_elements(x, y):
   .....:     return x + y

In [139]: add_them = np.frompyfunc(add_elements, 2, 1)

In [140]: add_them(np.arange(8), np.arange(8))
Out[140]: array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)
```

Functions created using frompyfunc always return arrays of Python objects, which can be inconvenient. Fortunately, there is an alternative (but slightly less featureful) function, numpy.vectorize, that allows you to specify the output type:

```
In [141]: add_them = np.vectorize(add_elements, otypes=[np.float64])

In [142]: add_them(np.arange(8), np.arange(8))
Out[142]: 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 [143]: arr = np.random.randn(10000)

In [144]: %timeit add_them(arr, arr)
3.05 ms +- 62.8 us per loop (mean +- std. dev. of 7 runs, 100 loops
each)

In [145]: %timeit np.add(arr, arr)
3.42 us +- 113 ns per loop (mean +- std. dev. of 7 runs, 100000
loops each)
```

Later in this chapter we'll show how to create fast ufuncs in Python using the Numba library.

A.5 Structured and Record Arrays

You may have noticed up until now that ndarray is a *homogeneous* data container; that is, it represents a block of memory in which each element takes up the same number of bytes, as determined by the dtype. On the surface, this would appear to not allow you to represent heterogeneous or tabular 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 [146]: dtype = [('x', np.float64), ('y', np.int32)]

In [147]: sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=dtype)

In [148]: sarr
Out[148]: array([(1.5   ,  6), (3.1416, -2)], dtype=[('x', '<f8'),
('y', '<i4')])
```

There are several ways to specify a structured dtype (see the online NumPy documentation). One typical way is as a list of tuples with `(field_name, field_data_type)`. Now, the elements of the array are tuple-like objects whose elements can be accessed like a dictionary:

```
In [149]: sarr[0]
Out[149]: (1.5, 6)

In [150]: sarr[0]['y']
Out[150]: 6
```

The field names are stored in the `dtype.names` attribute. When you access a field on the structured array, a strided view on the data is returned, thus copying nothing:

```
In [151]: sarr['x']
Out[151]: 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 [152]: dtype = [('x', np.int64, 3), ('y', np.int32)]

In [153]: arr = np.zeros(4, dtype=dtype)

In [154]: arr
Out[154]: 
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 [155]: arr[0]['x']
Out[155]: 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 [156]: arr['x']
Out[156]: 
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

This enables you to express more complicated, nested structures as a single block of memory in an array. You can also nest dtypes to make more complex structures. Here is an example:

```
In [157]: dtype = [('x', [('a', 'f8'), ('b', 'f4')]), ('y',
np.int32)]

In [158]: data = np.array([((1, 2), 5), ((3, 4), 6)], dtype=dtype)

In [159]: data['x']
Out[159]: array([(1., 2.), (3., 4.)], dtype=[('a', '<f8'), ('b',
'<f4')])

In [160]: data['y']
Out[160]: array([5, 6], dtype=int32)

In [161]: data['x']['a']
Out[161]: array([1., 3.])
```

pandas DataFrame does not support this feature in the same way, though it is similar to hierarchical indexing.

Why Use Structured Arrays?

Compared with a pandas DataFrame, NumPy structured arrays are a lower level tool. They provide a means to interpreting a block of memory as a tabular structure with nested columns. Since each element in the array is represented in memory as a fixed number of bytes, structured arrays provide an efficient way of writing data to and from disk (including memory maps), transporting it over the network, and other such uses. The memory layout of each value in a structured array is based on the binary representation of struct data types in the C programming language.

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 sometimes found in legacy systems in industry. As long as the format of the file is known (the size of each record and the order, byte size, and data type of each element), the data can be read into memory with `np.fromfile`. Specialized uses like this are beyond the scope of this book, but it's worth knowing that such things are possible.

A.6 More About Sorting

Like Python's built-in list, the ndarray `sort` instance method is an *in-place* sort, meaning that the array contents are rearranged without producing a new array:

```
In [162]: arr = np.random.randn(6)

In [163]: arr.sort()

In [164]: arr
Out[164]: 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 [165]: arr = np.random.randn(3, 5)

In [166]: arr
Out[166]:
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 [167]: arr[:, 0].sort()  # Sort first column values in-place

In [168]: arr
Out[168]:
array([[-1.0111, -1.4711,  0.8705, -0.0847, -1.1329],
       [-0.3318, -0.3436,  2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424,  0.8548,  1.038 , -0.329 ]])
```

On the other hand, `numpy.sort` creates a new, sorted copy of an array. Otherwise, it accepts the same arguments (such as `kind`) as ndarray's `sort` method:

```
In [169]: arr = np.random.randn(5)

In [170]: arr
Out[170]: array([-1.1181, -0.2415, -2.0051,  0.7379, -1.0614])

In [171]: np.sort(arr)
Out[171]: array([-2.0051, -1.1181, -1.0614, -0.2415,  0.7379])

In [172]: arr
Out[172]: 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 [173]: arr = np.random.randn(3, 5)

In [174]: arr
Out[174]:
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 [175]: arr.sort(axis=1)

In [176]: arr
Out[176]:
```

```
array([[-0.2682, -0.1872,  0.5955,  0.9111,  1.3389],
       [-0.5168, -0.3215, -0.1989,  1.0054,  1.1925],
       [-1.7638, -0.2222, -0.2171,  0.3969,  0.6071]])
```

You may notice that none of the sort methods have an option to sort in descending order. This is a problem in practice because array slicing produces views, thus not producing a copy or requiring any computational work. Many Python users are familiar with the "trick" that for a list `values`, `values[::-1]` returns a list in reverse order. The same is true for ndarrays:

```
In [177]: arr[:, ::-1]
Out[177]:
array([[ 1.3389,  0.9111,  0.5955, -0.1872, -0.2682],
       [ 1.1925,  1.0054, -0.1989, -0.3215, -0.5168],
       [ 0.6071,  0.3969, -0.2171, -0.2222, -1.7638]])
```

Indirect Sorts: argsort and lexsort

In data analysis you may need to reorder datasets by one or more keys. For example, a table of data about some students might need to be sorted by last name, then by first name. This is an example of an *indirect* sort, and if you've read the pandas-related chapters you have already seen many higher-level examples. Given a key or keys (an array of values or multiple arrays of values), you wish to obtain an array of integer *indices* (I refer to them colloquially as *indexers*) that tells you how to reorder the data to be in sorted order. Two methods for this are `argsort` and `numpy.lexsort`. As an example:

```
In [178]: values = np.array([5, 0, 1, 3, 2])

In [179]: indexer = values.argsort()

In [180]: indexer
Out[180]: array([1, 2, 4, 3, 0])

In [181]: values[indexer]
Out[181]: array([0, 1, 2, 3, 5])
```

As a more complicated example, this code reorders a two-dimensional array by its first row:

```
In [182]: arr = np.random.randn(3, 5)

In [183]: arr[0] = values

In [184]: arr
Out[184]:
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 [185]: arr[:, arr[0].argsort()]
Out[185]:
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 [186]: first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill',
'Barbara'])

In [187]: last_name = np.array(['Jones', 'Arnold', 'Arnold',
'Jones', 'Walters'])

In [188]: sorter = np.lexsort((first_name, last_name))

In [189]: sorter
Out[189]: array([1, 2, 3, 0, 4])

In [190]: list(zip(last_name[sorter], first_name[sorter]))
Out[190]:
[('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. Here, `last_name` was used before `first_name`.

Alternative Sort Algorithms

A *stable* sorting algorithm preserves the relative position of equal elements. This can be especially important in indirect sorts where the relative ordering is meaningful:

```
In [191]: values = np.array(['2:first', '2:second', '1:first',
'1:second',
   .....:                     '1:third'])

In [192]: key = np.array([2, 2, 1, 1, 1])

In [193]: indexer = key.argsort(kind='mergesort')

In [194]: indexer
Out[194]: array([2, 3, 4, 0, 1])

In [195]: values.take(indexer)
Out[195]:
array(['1:first', '1:second', '1:third', '2:first', '2:second'],
      dtype='<U8')
```

The only stable sort available is *mergesort*, which has guaranteed `O(n log n)` performance, but its performance is on average worse than the default quicksort method. See Table A-3 for a summary of available methods and their relative performance (and performance guarantees). This is not something that most users will ever have to think about, but it's useful to know that it's there.

Table A-3. Array sorting methods

Kind	Speed	Stable	Work space	Worst case
`'quicksort'`	1	No	0	`O(n^2)`
`'mergesort'`	2	Yes	`n / 2`	`O(n log n)`
`'heapsort'`	3	No	0	`O(n log n)`

Partially Sorting Arrays

One of the goals of sorting can be to determine the largest or smallest elements in an array. NumPy has fast methods, `numpy.partition` and `np.argpartition`, for partitioning an array around the `k`-th smallest element:

```
In [196]: np.random.seed(12345)

In [197]: arr = np.random.randn(20)

In [198]: arr
Out[198]:
array([-0.2047,  0.4789, -0.5194, -0.5557,  1.9658,  1.3934,
0.0929,
        0.2817,  0.769 ,  1.2464,  1.0072, -1.2962,  0.275 ,
0.2289,
        1.3529,  0.8864, -2.0016, -0.3718,  1.669 , -0.4386])

In [199]: np.partition(arr, 3)
Out[199]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386,
-0.2047,
        0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  0.275 ,
0.2289,
        1.3529,  0.8864,  1.3934,  1.9658,  1.669 ,  1.2464])
```

After you call `partition(arr, 3)`, the first three elements in the result are the smallest three values in no particular order. `numpy.argpartition`, similar to `numpy.argsort`, returns the indices that rearrange the data into the equivalent order:

```
In [200]: indices = np.argpartition(arr, 3)

In [201]: indices
Out[201]:
array([16, 11,  3,  2, 17, 19,  0,  7,  8,  1, 10,  6, 12, 13, 14,
15,  5,
        4, 18,  9])

In [202]: arr.take(indices)
Out[202]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -0.4386,
-0.2047,
        0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  0.275 ,
0.2289,
        1.3529,  0.8864,  1.3934,  1.9658,  1.669 ,  1.2464])
```

numpy.searchsorted: Finding Elements in a Sorted Array

`searchsorted` is an array method that performs a binary search on a sorted array, returning the location in the array where the value would need to be inserted to maintain sortedness:

```
In [203]: arr = np.array([0, 1, 7, 12, 15])

In [204]: arr.searchsorted(9)
Out[204]: 3
```

You can also pass an array of values to get an array of indices back:

```
In [205]: arr.searchsorted([0, 8, 11, 16])
Out[205]: 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 [206]: arr = np.array([0, 0, 0, 1, 1, 1, 1])

In [207]: arr.searchsorted([0, 1])
Out[207]: array([0, 3])

In [208]: arr.searchsorted([0, 1], side='right')
Out[208]: array([3, 7])
```

As another application of `searchsorted`, suppose we had an array of values between 0 and 10,000, and a separate array of “bucket edges” that we wanted to use to bin the data:

```
In [209]: data = np.floor(np.random.uniform(0, 10000, size=50))

In [210]: bins = np.array([0, 100, 1000, 5000, 10000])

In [211]: data
Out[211]:
array([9940., 6768., 7908., 1709.,  268., 8003., 9037.,  246.,
4917.,
       5262., 5963.,  519., 8950., 7282., 8183., 5002., 8101.,
959.,
       2189., 2587., 4681., 4593., 7095., 1780., 5314., 1677.,
7688.,
       9281., 6094., 1501., 4896., 3773., 8486., 9110., 3838.,
3154.,
       5683., 1878., 1258., 6875., 7996., 5735., 9732., 6340.,
8884.,
       4954., 3516., 7142., 5039., 2256.])
```

To then get a labeling of which interval each data point belongs to (where 1 would mean the bucket `[0, 100)`), we can simply use `searchsorted`:

```
In [212]: labels = bins.searchsorted(data)

In [213]: labels
Out[213]:
array([4, 4, 4, 3, 2, 4, 4, 2, 3, 4, 4, 2, 4, 4, 4, 4, 4, 2, 3, 3,
3, 3,
       4, 3, 4, 3, 4, 4, 4, 3, 3, 3, 4, 4, 3, 3, 4, 3, 3, 4, 4, 4,
4, 4,
       4, 3, 3, 4, 4, 3])
```

This, combined with pandas's `groupby`, can be used to bin data:

```
In [214]: pd.Series(data).groupby(labels).mean()
Out[214]:
2     498.000000
3    3064.277778
4    7389.035714
dtype: float64
```

A.7 Writing Fast NumPy Functions with Numba

Numba is an open source project that creates fast functions for NumPy-like data using CPUs, GPUs, or other hardware. It uses the LLVM Project to translate Python code into compiled machine code.

To introduce Numba, let's consider a pure Python function that computes the expression `(x - y).mean()` using a `for` loop:

```
import numpy as np

def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

This function is very slow:

```
In [209]: x = np.random.randn(10000000)

In [210]: y = np.random.randn(10000000)

In [211]: %timeit mean_distance(x, y)
1 loop, best of 3: 2 s per loop

In [212]: %timeit (x - y).mean()
100 loops, best of 3: 14.7 ms per loop
```

The NumPy version is over 100 times faster. We can turn this function into a compiled Numba function using the `numba.jit` function:

```
In [213]: import numba as nb

In [214]: numba_mean_distance = nb.jit(mean_distance)
```

We could also have written this as a decorator:

```
@nb.jit
def numba_mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

The resulting function is actually faster than the vectorized NumPy version:

```
In [215]: %timeit numba_mean_distance(x, y)
100 loops, best of 3: 10.3 ms per loop
```

Numba cannot compile all pure Python code, but it supports a significant subset of Python that is most useful for writing numerical algorithms.

Numba is a deep library, supporting different kinds of hardware, modes of compilation, and user extensions. It is also able to compile a substantial subset of the NumPy Python API without explicit `for` loops. Numba is able to recognize constructs that can be compiled to machine code, while substituting calls to the CPython API for functions that it does not know how to compile. Numba's `jit` function has an option, `nopython=True`, which restricts allowed code to Python code that can be compiled to LLVM without any Python C API calls. `jit(nopython=True)` has a shorter alias `numba.njit`.

In the previous example, we could have written:

```
from numba import float64, njit

@njit(float64(float64[:], float64[:]))
def mean_distance(x, y):
    return (x - y).mean()
```

I encourage you to learn more by reading the online documentation for Numba. The next section shows an example of creating custom NumPy ufunc objects.

Creating Custom numpy.ufunc Objects with Numba

The `numba.vectorize` function creates compiled NumPy ufuncs, which behave like built-in ufuncs. Let's consider a Python implementation of `numpy.add`:

```
from numba import vectorize

@vectorize
def nb_add(x, y):
    return x + y
```

Now we have:

```
In [13]: x = np.arange(10)

In [14]: nb_add(x, x)
Out[14]: array([  0.,    2.,    4.,    6.,    8.,   10.,   12.,   14.,
16.,   18.])
```

```
In [15]: nb_add.accumulate(x, 0)
Out[15]: array([  0.,   1.,   3.,   6.,  10.,  15.,  21.,  28.,
36.,  45.])
```

A.8 Advanced Array Input and Output

In Chapter 4, we became acquainted with `np.save` and `np.load` for storing arrays in binary format on disk. There are a number of additional options to consider for more sophisticated use. In particular, memory maps have the additional benefit of enabling you to do certain operations with datasets that do not fit into RAM.

Memory-Mapped Files

A *memory-mapped* file is a method for interacting with binary data on disk as though it is stored in an in-memory array. NumPy implements a `memmap` object that is ndarray-like, enabling small segments of a large file to be read and written without reading the whole array into memory. Additionally, a `memmap` has the same methods as an in-memory array and thus can be substituted into many algorithms where an ndarray would be expected.

To create a new memory map, use the function `np.memmap` and pass a file path, dtype, shape, and file mode:

```
In [216]: mmap = np.memmap('mymmap', dtype='float64', mode='w+',
   .....:                  shape=(10000, 10000))

In [217]: mmap
Out[217]:
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 [218]: section = mmap[:5]
```

If you assign data to these, it will be buffered in memory, which means that the changes will not be immediately reflected in the on-disk file if you were to read the file in a different application. Any modifications can be synchronized to disk by calling `flush`:

```
In [219]: section[:] = np.random.randn(5, 10000)

In [220]: mmap.flush()

In [221]: mmap
Out[221]:
memmap([[ 0.7584, -0.6605,  0.8626, ...,  0.6046, -0.6212,  2.0542],
        [-1.2113, -1.0375,  0.7093, ..., -1.4117, -0.1719, -0.8957],
        [-0.1419, -0.3375,  0.4329, ...,  1.2914, -0.752 , -0.44  ],
        ...,
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ],
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ],
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    
]])

In [222]: del mmap
```

Whenever a memory map falls out of scope and is garbage-collected, any changes will be flushed to disk also. When *opening an existing memory map*, you still have to specify the dtype and shape, as the file is only a block of binary data without any data type information, shape, or strides:

```
In [223]: mmap = np.memmap('mymmap', dtype='float64', shape=(10000,
10000))

In [224]: mmap
Out[224]:
memmap([[ 0.7584, -0.6605,  0.8626, ...,  0.6046, -0.6212,  2.0542],
        [-1.2113, -1.0375,  0.7093, ..., -1.4117, -0.1719, -0.8957],
        [-0.1419, -0.3375,  0.4329, ...,  1.2914, -0.752 , -0.44  ],
        ...,
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ],
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    ],
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.    ,  0.    
]])
```

Memory maps also work with structured or nested dtypes as described in a previous section.

If you ran this example on your computer, you may want to delete the large file that we created above:

```
In [225]: %xdel mmap

In [226]: !rm mymmap
```

HDF5 and Other Array Storage Options

PyTables and h5py are two Python projects providing NumPy-friendly interfaces for storing array data in the efficient and compressible HDF5 format (HDF stands for *hierarchical data format*). You can safely store hundreds of gigabytes or even terabytes of data in HDF5 format. To learn more about using HDF5 with Python, I recommend reading the pandas online documentation.

A.9 Performance Tips

Adapting data processing code to use NumPy generally makes things much faster, as array operations typically replace otherwise comparatively extremely slow pure Python loops. Here are some tips to help get the best performance out of the library:

- Convert Python loops and conditional logic to array operations and boolean array operations
- Use broadcasting whenever possible
- Use arrays views (slicing) to avoid copying data
- Utilize ufuncs and ufunc methods

If you can’t get the performance you require after exhausting the capabilities provided by NumPy alone, consider writing code in C, Fortran, or Cython. I use Cython frequently in my own work as a way to get C-like performance with often much less development time.

The Importance of Contiguous Memory

While the full extent of this topic is a bit outside the scope of this book, in some applications the memory layout of an array can significantly affect the speed of computations. This is based partly on performance differences having to do with the cache hierarchy of the CPU; operations accessing contiguous blocks of memory (e.g., summing the rows of a C order array) will generally be the fastest because the memory subsystem will buffer the appropriate blocks of memory into the low latency L1 or L2 CPU caches. Also, certain code paths inside NumPy's C codebase have been optimized for the contiguous case in which generic strided memory access can be avoided.

To say that an array's memory layout is *contiguous* means that the elements are stored in memory in the order that they appear in the array with respect to Fortran (column major) or C (row major) ordering. By default, NumPy arrays are created as *C-contiguous* or just simply contiguous. A column major array, such as the transpose of a C-contiguous array, is thus said to be Fortran-contiguous. These properties can be explicitly checked via the `flags` attribute on the ndarray:

```
In [227]: arr_c = np.ones((100, 10000), order='C')

In [228]: arr_f = np.ones((100, 10000), order='F')

In [229]: arr_c.flags
Out[229]:
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False

In [230]: arr_f.flags
Out[230]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

```
In [231]: arr_f.flags.f_contiguous
Out[231]: True
```

In this example, summing the rows of these arrays should, in theory, be faster for `arr_c` than `arr_f` since the rows are contiguous in memory. Here I check using `%timeit` in IPython (these results may differ on your machine):

```
In [232]: %timeit arr_c.sum(1)
444 us +- 15.7 us per loop (mean +- std. dev. of 7 runs, 1000 loops
each)

In [233]: %timeit arr_f.sum(1)
622 us +- 30.2 us per loop (mean +- std. dev. of 7 runs, 1000 loops
each)
```

When you're looking to squeeze more performance out of NumPy, this is often a place to invest some effort. If you have an array that does not have the desired memory order, you can use `copy` and pass either `'C'` or `'F'`:

```
In [234]: arr_f.copy('C').flags
Out[234]:
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

When constructing a view on an array, keep in mind that the result is not guaranteed to be contiguous:

```
In [235]: arr_c[:50].flags.contiguous
Out[235]: True

In [236]: arr_c[:, :50].flags
Out[236]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

[1] Some of the dtypes have trailing underscores in their names. These are there to avoid variable name conflicts between the NumPy-specific types and the Python built-in ones.

Appendix B. More on the IPython System

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd appendix of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at arufino@oreilly.com.

In Chapter 2 we looked at the basics of using the IPython shell and Jupyter notebook. In this chapter, we explore some deeper functionality in the IPython system that can either be used from the console or within Jupyter.

B.1 Using the Command History

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

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

These features are more useful in the shell than in the notebook, since the notebook by design keeps a log of the input and output in each code cell.

Searching and Reusing the Command History

The IPython shell lets you search and execute previous code or other commands. This is useful, as you may often find yourself repeating the same commands, such as a `%run` command or some other code snippet. Suppose you had run:

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

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

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

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

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

Pressing Ctrl-R will cycle through the history for each line matching the characters you’ve typed.

Input and Output Variables

Forgetting to assign the result of a function call to a variable can be very annoying. An IPython session stores references to *both* the input commands and output Python objects in special variables. The previous two outputs are stored in the _ (one underscore) and __ (two underscores) variables, respectively:

```
In [18]: 'input1'
Out[18]: 'input1'

In [19]: 'input2'
Out[19]: 'input2'

In [20]: __
Out[20]: 'input1'

In [21]: 'input3'
Out[21]: 'input3'

In [22]: _
Out[22]: 'input3'
```

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

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

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

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

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

Since the input variables are strings they can be executed again with the Python `eval` keyword:

```
In [30]: eval(_i27)
Out[30]: 'bar'
```

Here `_i27` refers to the code input in `In [27]`.

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

WARNING

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

B.2 Interacting with the Operating System

Another feature of IPython is that it allows you to access the filesystem and operating system shell. This means, among other things, that you can perform most standard command-line actions as you would in the Windows or Unix (Linux, macOS) shell without having to exit IPython. This includes shell commands, changing directories, and storing the results of a command in a Python object (list or string). There are also command aliasing and directory bookmarking features.

See Table B-1 for a summary of magic functions and syntax for calling shell commands. I'll briefly visit these features in the next few sections.

Table B-1. IPython system-related commands

Command	Description
`!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`	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
`%matplotlib`	Configure matplotlib integration options

Shell Commands and Aliases

Starting a line in IPython with an exclamation point `!`, or bang, tells IPython to execute everything after the bang in the system shell. This means that you can delete files (using `rm` or `del`, depending on your OS), change directories, or execute any other process.

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

```
In [1]: ip_info = !ifconfig wlan0 | grep "inet "
In [2]: ip_info[0].strip()
Out[2]: 'inet addr:10.0.0.11  Bcast:10.0.0.255
Mask:255.255.255.0'
```

The returned Python object `ip_info` is actually a custom list type containing various versions of the console output.

IPython can also substitute in Python values defined in the current environment when using `!`. To do this, preface the variable name by the dollar sign `$`:

```
In [3]: foo = 'test*'

In [4]: !ls $foo
test4.py  test.py  test.xml
```

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

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

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

You can execute multiple commands just as on the command line by separating them with semicolons:

```
In [558]: %alias test_alias (cd examples; ls; cd ..)

In [559]: test_alias
macrodata.csv  spx.csv  tips.csv
```

You’ll notice that IPython “forgets” any aliases you define interactively as soon as the session is closed. To create permanent aliases, you will need to use the configuration system.

Directory Bookmark System

IPython has a directory bookmarking system to enable you to save aliases for common directories so that you can jump around easily. For example, suppose you wanted to create a bookmark that points to the supplementary materials for this book:

```
In [6]: %bookmark py4da /home/wesm/code/pydata-book
```

Once you’ve done this, when we use the `%cd` magic, we can use any bookmarks we’ve defined:

```
In [7]: cd py4da
(bookmark:py4da) -> /home/wesm/code/pydata-book
/home/wesm/code/pydata-book
```

If a bookmark name conflicts with a directory name in your current working directory, you can use the `-b` flag to override and use the bookmark location. Using the `-l` option with `%bookmark` lists all of your bookmarks:

```
In [8]: %bookmark -l
Current bookmarks:
py4da -> /home/wesm/code/pydata-book-source
```

Bookmarks, unlike aliases, are automatically persisted between IPython sessions.

B.3 Software Development Tools

In addition to being a comfortable environment for interactive computing and data exploration, IPython can also be a useful companion for general

Python software development. In data analysis applications, it's important first to have *correct* code. Fortunately, IPython has closely integrated and enhanced the built-in Python `pdb` debugger. Secondly you want your code to be *fast*. For this IPython has easy-to-use code timing and profiling tools. I will give an overview of these tools in detail here.

Interactive Debugger

IPython's debugger enhances `pdb` with tab completion, syntax highlighting, and context for each line in exception tracebacks. One of the best times to debug code is right after an error has occurred. The `%debug` command, when entered immediately after an exception, invokes the "post-mortem" debugger and drops you into the stack frame where the exception was raised:

```
In [2]: run examples/ipython_bug.py
-----------------------------------------------------------------
----------
AssertionError                          Traceback (most recent
call last)
/home/wesm/code/pydata-book/examples/ipython_bug.py in <module>()
     13     throws_an_exception()
     14
---> 15 calling_things()

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

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

AssertionError:
```

```
In [3]: %debug
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(9)throws_an_exception()
      8     b = 6
----> 9     assert(a + b == 10)
     10

ipdb>
```

Once inside the debugger, you can execute arbitrary Python code and explore all of the objects and data (which have been "kept alive" by the interpreter) inside each stack frame. By default you start in the lowest level, where the error occurred. By pressing u (up) and d (down), you can switch between the levels of the stack trace:

```
ipdb> u
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(13)calling_things()
     12     works_fine()
---> 13     throws_an_exception()
     14
```

Executing the %pdb command makes it so that IPython automatically invokes the debugger after any exception, a mode that many users will find useful.

It's also easy to use the debugger to help develop code, especially when you wish to set breakpoints or step through the execution of a function or script to examine the state at each stage. There are several ways to accomplish this. The first is by using %run with the -d flag, which invokes the debugger before executing any code in the passed script. You must immediately press s (step) to enter the script:

```
In [5]: run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-
book/examples/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()

ipdb> s
```

```
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py(1)<module>
()
1---> 1 def works_fine():
      2     a = 5
      3     b = 6
```

After this point, it's up to you how you want to work your way through the file. For example, in the preceding exception, we could set a breakpoint right before calling the `works_fine` function and run the script until we reach the breakpoint by pressing `c` (continue):

```
ipdb> b 12
ipdb> c
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(12)calling_things()
     11 def calling_things():
2--> 12     works_fine()
     13     throws_an_exception()
```

At this point, you can `step` into `works_fine()` or execute `works_fine()` by pressing `n` (next) to advance to the next line:

```
ipdb> n
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(13)calling_things()
2    12     works_fine()
---> 13     throws_an_exception()
     14
```

Then, we could step into `throws_an_exception` and advance to the line where the error occurs and look at the variables in the scope. Note that debugger commands take precedence over variable names; in such cases, preface the variables with `!` to examine their contents:

```
ipdb> s
--Call--
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(6)throws_an_exception()
      5
----> 6 def throws_an_exception():
      7     a = 5

ipdb> n
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(7)throws_an_exception()
      6 def throws_an_exception():
----> 7     a = 5
      8     b = 6

ipdb> n
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(8)throws_an_exception()
      7     a = 5
----> 8     b = 6
      9     assert(a + b == 10)

ipdb> n
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(9)throws_an_exception()
      8     b = 6
----> 9     assert(a + b == 10)
     10

ipdb> !a
5
ipdb> !b
6
```

In my experience, developing proficiency with the interactive debugger takes time and practice. See Table B-2 for a full catalog of the debugger commands. If you are accustomed to using an IDE, you might find the terminal-driven debugger to be a bit unforgiving at first, but that will improve in time. Some of the Python IDEs have excellent GUI debuggers, so most users can find something that works for them.

Table B-2. (I)Python debugger commands

Command	Action
`h(elp)`	Display command list
`help` *`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 a couple of other useful ways to invoke the debugger. The first is by using a special `set_trace` function (named after `pdb.set_trace`), which is basically a "poor man's breakpoint." Here are two small recipes you might want to put somewhere for your general use (potentially adding them to your IPython profile as I do):

```
from IPython.core.debugger import Pdb

def set_trace():
    Pdb(.set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):
    pdb = Pdb()
    return pdb.runcall(f, *args, **kwargs)
```

The first function, `set_trace` provides a convenient way to put a breakpoint somewhere in your code. You can use a `set_trace` in any part of your code that you want to temporarily stop in order to more closely examine it (e.g., right before an exception occurs):

```
In [7]: run examples/ipython_bug.py
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(16)calling_things()
     15     set_trace()
---> 16     throws_an_exception()
     17
```

Pressing `c` (continue) will cause the code to resume normally with no harm done.

The `debug` function we just looked at enables you to invoke the interactive debugger easily on an arbitrary function call. Suppose we had written a function like the following and we wished to step through its logic:

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

Ordinarily using `f` would look like `f(1, 2, z=3)`. To instead step into `f`, pass `f` as the first argument to `debug` followed by the positional and keyword arguments to be passed to `f`:

```
In [6]: debug(f, 1, 2, z=3)
> <ipython-input>(2)f()
      1 def f(x, y, z):
----> 2     tmp = x + y
      3     return tmp / z

ipdb>
```

These two recipes have saved me a lot of time over the years.

Lastly, the debugger can be used in conjunction with `%run`. By running a script with `%run -d`, you will be dropped directly into the debugger, ready to set any breakpoints and start the script:

```
In [1]: %run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-
book/examples/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()

ipdb>
```

Adding `-b` with a line number starts the debugger with a breakpoint set already:

```
In [2]: %run -d -b2 examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-
book/examples/ipython_bug.py:2
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()

ipdb> c
> /home/wesm/code/pydata-
book/examples/ipython_bug.py(2)works_fine()
      1 def works_fine():
1---> 2     a = 5
      3     b = 6

ipdb>
```

Timing Code: %time and %timeit

For larger-scale or longer-running data analysis applications, you may wish to measure the execution time of various components or of individual statements or function calls. You may want a report of which functions are taking up the most time in a complex process. Fortunately, IPython enables you to get this information conveniently 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 list of 600,000 strings and two identical methods of selecting only the ones that start with `'foo'`:

```
# a very large list of strings
strings = ['foo', 'foobar', 'baz', 'qux',
           'python', 'Guido Van Rossum'] * 100000

method1 = [x for x in strings if x.startswith('foo')]

method2 = [x for x in strings if x[:3] == 'foo']
```

It looks like they should be about the same performance-wise, right? We can check for sure using `%time`:

```
In [561]: %time method1 = [x for x in strings if
x.startswith('foo')]
CPU times: user 0.19 s, sys: 0.00 s, total: 0.19 s
Wall time: 0.19 s

In [562]: %time method2 = [x for x in strings if x[:3] == 'foo']
CPU times: user 0.09 s, sys: 0.00 s, total: 0.09 s
Wall time: 0.09 s
```

The `Wall time` (short for “wall-clock time”) is the main number of interest. So, it looks like the first method takes more than twice as long, but it’s not a very precise measurement. If you try `%time`-ing those statements multiple times yourself, you’ll find that the results are somewhat variable. To get a more precise measurement, use the `%timeit` magic function. Given an arbitrary statement, it has a heuristic to run a statement multiple times to produce a more accurate average runtime (these results may be different on your system):

```
In [563]: %timeit [x for x in strings if x.startswith('foo')]
10 loops, best of 3: 159 ms per loop

In [564]: %timeit [x for x in strings if x[:3] == 'foo']
10 loops, best of 3: 59.3 ms per loop
```

This seemingly innocuous example illustrates that it is worth understanding the performance characteristics of the Python standard library, NumPy, pandas, and other libraries used in this book. In larger-scale data analysis applications, those milliseconds will start to add up!

`%timeit` is especially useful for analyzing statements and functions with very short execution times, even at the level of microseconds (millionths of a second) or nanoseconds (billionths of a second). These may seem like insignificant amounts of time, but of course a 20 microsecond function invoked 1 million times takes 15 seconds longer than a 5 microsecond function. In the preceding example, we could very directly compare the two string operations to understand their performance characteristics:

```
In [565]: x = 'foobar'

In [566]: y = 'foo'

In [567]: %timeit x.startswith(y)
1000000 loops, best of 3: 267 ns per loop

In [568]: %timeit x[:3] == y
10000000 loops, best of 3: 147 ns per loop
```

Basic Profiling: %prun and %run -p

Profiling code is closely related to timing code, except it is concerned with determining *where* time is spent. The main Python profiling tool is the `cProfile` module, which is not specific to IPython at all. `cProfile` executes a program or any arbitrary block of code while keeping track of how much time is spent in each function.

A common way to use `cProfile` is on the command line, running an entire program and outputting the aggregated time per function. Suppose we had a script that does some linear algebra in a loop (computing the maximum absolute eigenvalues of a series of 100 × 100 matrices):

```
import numpy as np
from numpy.linalg import eigvals

def run_experiment(niter=100):
    K = 100
    results = []
    for _ in range(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: {0}'.format(np.max(some_results)))
```

You can run this script through `cProfile` using the following in the command line:

```
python -m cProfile cprof_example.py
```

If you try that, you'll find that the output is sorted by function name. This makes it a bit hard to get an idea of where the most time is spent, so it's useful to specify a *sort order* using the `-s` flag:

```
$ python -m cProfile -s cumulative cprof_example.py
Largest one we saw: 11.923204422
    15116 function calls (14927 primitive calls) in 0.720 seconds
```

```
Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
     1    0.001    0.001    0.721    0.721
cprof_example.py:1(<module>)
   100    0.003    0.000    0.586    0.006 linalg.py:702(eigvals)
   200    0.572    0.003    0.572    0.003
{numpy.linalg.lapack_lite.dgeev}
     1    0.002    0.002    0.075    0.075
__init__.py:106(<module>)
   100    0.059    0.001    0.059    0.001 {method 'randn')
     1    0.000    0.000    0.044    0.044
add_newdocs.py:9(<module>)
     2    0.001    0.001    0.037    0.019
__init__.py:1(<module>)
     2    0.003    0.002    0.030    0.015
__init__.py:2(<module>)
     1    0.000    0.000    0.030    0.030
type_check.py:3(<module>)
     1    0.001    0.001    0.021    0.021
__init__.py:15(<module>)
     1    0.013    0.013    0.013    0.013 numeric.py:1(<module>)
     1    0.000    0.000    0.009    0.009
__init__.py:6(<module>)
     1    0.001    0.001    0.008    0.008
__init__.py:45(<module>)
   262    0.005    0.000    0.007    0.000
function_base.py:3178(add_newdoc)
   100    0.003    0.000    0.005    0.000
linalg.py:162(_assertFinite)
   ...
```

Only the first 15 rows of the output are shown. It's easiest to read by scanning down the `cumtime` column to see how much total time was spent *inside* each function. Note that if a function calls some other function, *the clock does not stop running*. `cProfile` records the start and end time of each function call and uses that to produce the timing.

In addition to the command-line usage, `cProfile` can also be used programmatically to profile arbitrary blocks of code without having to run a new process. IPython has a convenient interface to this capability using the `%prun` command and the `-p` option to `%run`. `%prun` takes the same “command-line options” as `cProfile` but will profile an arbitrary Python statement instead of a whole *.py* file:

```
In [4]: %prun -l 7 -s cumulative run_experiment()
         4203 function calls in 0.643 seconds

Ordered by: cumulative time
List reduced from 32 to 7 due to restriction <7>

ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
     1    0.000    0.000    0.643    0.643 <string>:1(<module>)
     1    0.001    0.001    0.643    0.643
cprof_example.py:4(run_experiment)
   100    0.003    0.000    0.583    0.006 linalg.py:702(eigvals)
   200    0.569    0.003    0.569    0.003
{numpy.linalg.lapack_lite.dgeev}
   100    0.058    0.001    0.058    0.001 {method 'randn'}
   100    0.003    0.000    0.005    0.000
linalg.py:162(_assertFinite)
   200    0.002    0.000    0.002    0.000 {method 'all' of
'numpy.ndarray'}
```

Similarly, calling `%run -p -s cumulative cprof_example.py` has the same effect as the command-line approach, except you never have to leave IPython.

In the Jupyter notebook, you can use the `%%prun` magic (two `%` signs) to profile an entire code block. This pops up a separate window with the profile output. This can be useful in getting possibly quick answers to questions like, “Why did that code block take so long to run?”

There are other tools available that help make profiles easier to understand when you are using IPython or Jupyter. One of these is SnakeViz, which produces an interactive visualization of the profile results using d3.js.

Profiling a Function Line by Line

In some cases the information you obtain from `%prun` (or another `cProfile`-based profile method) may not tell the whole story about a function’s execution time, or it may be so complex that the results, aggregated by function name, are hard to interpret. For this case, there is a small library called `line_profiler` (obtainable via PyPI or one of the package management tools). It contains an IPython extension enabling a new magic function `%lprun` that computes a line-by-line-profiling of one or more functions. You can enable this extension by modifying your IPython configuration (see the IPython documentation or the section on configuration later in this chapter) to include the following line:

```
# A list of dotted module names of IPython extensions to load.
c.InteractiveShellApp.extensions = ['line_profiler']
```

You can also run the command:

```
%load_ext line_profiler
```

`line_profiler` can be used programmatically (see the full documentation), but it is perhaps most powerful when used interactively in IPython. Suppose you had a module `prof_mod` with the following code doing some NumPy array operations (if you want to reproduce this example, put this code into a new file `prof_mod.py`):

```
from numpy.random import randn

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

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

If we wanted to understand the performance of the `add_and_sum` function, `%prun` gives us the following:

```
In [569]: %run prof_mod

In [570]: x = randn(3000, 3000)
In [571]: y = randn(3000, 3000)

In [572]: %prun add_and_sum(x, y)
         4 function calls in 0.049 seconds
   Ordered by: internal time
   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
        1    0.036    0.036    0.046    0.046
prof_mod.py:3(add_and_sum)
        1    0.009    0.009    0.009    0.009 {method 'sum' of
'numpy.ndarray'}
        1    0.003    0.003    0.049    0.049
<string>:1(<module>)
```

This is not especially enlightening. With the `line_profiler` IPython extension activated, a new command `%lprun` is available. The only difference in usage is that we must instruct `%lprun` which function or functions we wish to profile. The general syntax is:

```
%lprun -f func1 -f func2 statement_to_profile
```

In this case, we want to profile `add_and_sum`, so we run:

```
In [573]: %lprun -f add_and_sum add_and_sum(x, y)
Timer unit: 1e-06 s
File: prof_mod.py
Function: add_and_sum at line 3
Total time: 0.045936 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           def
add_and_sum(x, y):
     4         1        36510  36510.0     79.5      added = x +
y
     5         1         9425   9425.0     20.5      summed =
added.sum(axis=1)
     6         1            1      1.0      0.0      return
summed
```

This can be much easier to interpret. In this case we profiled the same function we used in the statement. Looking at the preceding module code, we could call `call_function` and profile that as well as `add_and_sum`, thus getting a full picture of the performance of the code:

```
In [574]: %lprun -f add_and_sum -f call_function call_function()
Timer unit: 1e-06 s
File: prof_mod.py
Function: add_and_sum at line 3
Total time: 0.005526 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     3                                           def
add_and_sum(x, y):
     4         1         4375   4375.0     79.2      added = x +
y
     5         1         1149   1149.0     20.8      summed =
added.sum(axis=1)
     6         1            2      2.0      0.0      return
summed
File: prof_mod.py
Function: call_function at line 8
Total time: 0.121016 s
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     8                                           def
call_function():
     9         1        57169  57169.0     47.2      x =
randn(1000, 1000)
    10         1        58304  58304.0     48.2      y =
randn(1000, 1000)
    11         1         5543   5543.0      4.6      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.

NOTE

The reason that you must explicitly specify the names of the functions you want to profile with `%lprun` is that the overhead of “tracing” the execution time of each line is substantial. Tracing functions that are not of interest has the potential to significantly alter the profile results.

B.4 Tips for Productive Code Development Using IPython

Writing code in a way that makes it easy to develop, debug, and ultimately *use* interactively may be a paradigm shift for many users. There are procedural details like code reloading that may require some adjustment as well as coding style concerns.

Therefore, implementing most of the strategies described in this section is more of an art than a science and will require some experimentation on your part to determine a way to write your Python code that is effective for you. Ultimately you want to structure your code in a way that makes it easy to use iteratively and to be able to explore the results of running a program or function as effortlessly as possible. I have found software designed with IPython in mind to be easier to work with than code intended only to be run as 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 IPython code development comes when you, say, `%run` a script that depends on some other module where you may have made changes. Suppose I had the following code in *test_script.py*:

```
import some_lib

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

If you were to execute `%run test_script.py` then modify *some_lib.py*, the next time you execute `%run test_script.py` you will still get the *old version* of *some_lib.py* because of Python's "load-once" module system. This behavior differs from some other data analysis environments, like MATLAB, which automatically propagate code changes.[1] To cope with this, you have a couple of options. The first way is to use the `reload` function in the `importlib` module in the standard library:

```
import some_lib
import importlib

importlib.reload(some_lib)
```

This guarantees that you will get a fresh copy of *some_lib.py* every time you run *test_script.py*. Obviously, if the dependencies go deeper, it might be a bit tricky to be inserting usages of `reload` all over the place. For this problem, IPython has a special `dreload` function (*not* a magic function) for "deep" (recursive) reloading of modules. If I were to run *some_lib.py* then type `dreload(some_lib)`, it will attempt to reload `some_lib` as well as all of its dependencies. This will not work in all cases, unfortunately, but when it does it beats having to restart IPython.

Code Design Tips

There's no simple recipe for this, but here are some high-level principles I have found effective in my own work.

Keep relevant objects and data alive

It's not unusual to see a program written for the command line with a structure somewhat like the following:

```
from my_functions import g

def f(x, y):
    return g(x + y)
```

```
def main():
    x = 6
    y = 7.5
    result = x + y

if __name__ == '__main__':
    main()
```

Do you see what might go wrong if we were to run this program in IPython? After it’s done, none of the results or objects defined in the `main` function will be accessible in the IPython shell. A better way is to have whatever code is in `main` execute directly in the module’s global namespace (or in the `if __name__ == '__main__':` block, if you want the module to also be importable). That way, when you `%run` the code, you’ll be able to look at all of the variables defined in `main`. This is equivalent to defining top-level variables in cells in the Jupyter notebook.

Flat is better than nested

Deeply nested code makes me think about the many layers of an onion. When testing or debugging a function, how many layers of the onion must you peel back in order to reach the code of interest? The idea that “flat is better than nested” is a part of the Zen of Python, and it applies generally to developing code for interactive use as well. Making functions and classes as decoupled and modular as possible makes them easier to test (if you are writing unit tests), debug, and use interactively.

Overcome a fear of longer files

If you come from a Java (or another such language) background, you may have been told to keep files short. In many languages, this is sound advice; long length is usually a bad “code smell,” indicating refactoring or reorganization may be necessary. However, while developing code using IPython, working with 10 small but interconnected files (under, say, 100 lines each) is likely to cause you more headaches in general than two or three longer files. Fewer files means fewer modules to reload and less jumping between files while editing, too. I have found maintaining larger modules, each with high *internal* cohesion (the code all relates to solving the same kinds of problems, to be much more useful and Pythonic. After iterating toward a solution, of course it sometimes will make sense to refactor larger files into smaller ones.

Obviously, I don’t support taking this argument to the extreme, which would 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.

B.5 Advanced IPython Features

Making full use of the IPython system may lead you to write your code in a slightly different way, or to dig into the configuration.

Making Your Own Classes IPython-Friendly

IPython makes every effort to display a console-friendly string representation of any object that you inspect. For many objects, like dicts, lists, and tuples, the built-in `pprint` module is used to do the nice formatting. In user-defined classes, however, you have to generate the desired string output yourself. Suppose we had the following small class:

```
class Message:
    def __init__(self, msg):
        self.msg = msg
```

If you wrote this, you would be disappointed to discover that the default output for your class isn’t too nice:

```
In [576]: x = Message('I have a secret')

In [577]: x
Out[577]: <__main__.Message instance at 0x60ebbd8>
```

IPython takes the string returned by the `__repr__` magic method (by doing `output = repr(obj)`) and prints that to the console. Thus, we can add a simple `__repr__` method to the preceding class to get a more helpful output:

```
class Message:
    def __init__(self, msg):
        self.msg = msg

    def __repr__(self):
        return 'Message: %s' % self.msg

In [579]: x = Message('I have a secret')

In [580]: x
Out[580]: Message: I have a secret
```

Profiles and Configuration

Most aspects of the appearance (colors, prompt, spacing between lines, etc.) and behavior of the IPython and Jupyter environments are configurable through an extensive configuration system. Here are some things you can do via configuration:

- Change the color scheme
- Change how the input and output prompts look, or remove the blank line after `Out` and before the next `In` prompt
- Execute an arbitrary list of Python statements (e.g., imports that you use all the time or anything else you want to happen each time you launch IPython)
- Enable always-on IPython extensions, like the `%lprun` magic in `line_profiler`
- Enabling Jupyter extensions
- Define your own magics or system aliases

Configurations for the IPython shell are specified in special *ipython_config.py* files, which are usually found in the *.ipython/* directory in your user home directory. Configuration is performed based on a particular *profile*. When you start IPython normally, you load up, by default, the *default profile*, stored in the *profile_default* directory. Thus, on my Linux OS the full path to my default IPython configuration file is:

```
/home/wesm/.ipython/profile_default/ipython_config.py
```

To initialize this file on your system, run in the terminal:

```
ipython profile create default
```

I'll spare you the complete details of what's in this file. Fortunately it has comments describing what each configuration option is for, so I will leave it to the reader to tinker and customize. One additional useful feature is that it's possible to have *multiple profiles*. Suppose you wanted to have an alternative IPython configuration tailored for a particular application or project. Creating a new profile involves typing the following:

```
ipython profile create secret_project
```

Once you've done this, edit the config files in the newly created *profile_secret_project* directory and then launch IPython like so:

```
$ ipython --profile=secret_project
Python 3.8.0 | packaged by conda-forge | (default, Nov 22 2019,
19:11:19)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.22.0 -- An enhanced Interactive Python. Type '?' for
help.

IPython profile: secret_project
```

As always, the online IPython documentation is an excellent resource for more on profiles and configuration.

Configuration for Jupyter works a little differently because you can use its notebooks with languages other than Python. To create an analogous Jupyter config file, run:

```
jupyter notebook --generate-config
```

This writes a default config file to the *.jupyter/jupyter_notebook_config.py* directory in your home directory. After editing this to suit your needs, you may rename it to a different file, like:

```
$ mv ~/.jupyter/jupyter_notebook_config.py
~/.jupyter/my_custom_config.py
```

When launching Jupyter, you can then add the `--config` argument:

```
jupyter notebook --config=~/.jupyter/my_custom_config.py
```

B.6 Conclusion

As you work through the code examples in this book and grow your skills as a Python programmer, I encourage you to keep learning about the IPython and Jupyter ecosystems. Since these projects have been designed to assist user productivity, you may discover tools that enable you to do your work more easily than using the Python language and its computational libraries by themselves.

You can also find a wealth of interesting Jupyter notebooks on the nbviewer website.

[1] Since a module or package may be imported in many different places in a particular program, Python caches a module's code the first time it is imported rather than executing the code in the module every time. Otherwise, modularity and good code organization could potentially cause inefficiency in an application.

About the Author

Wes McKinney is a New York-based software developer and entrepreneur. After finishing his undergraduate degree in mathematics at MIT in 2007, he went on to do quantitative finance work at AQR Capital Management in Greenwich, CT. Frustrated by cumbersome data analysis tools, he learned Python and started building what would later become the pandas project. He's now an active member of the Python data community and is an advocate for the use of Python in data analysis, finance, and statistical computing applications.

Wes was later the cofounder and CEO of DataPad, whose technology assets and team were acquired by Cloudera in 2014. He has since become involved in big data technology, joining the Project Management Committees for the Apache Arrow and Apache Parquet projects in the Apache Software Foundation. In 2016, he joined Two Sigma Investments in New York City, where he continues working to make data analysis faster and easier through open source software.