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

Copyright © 2021 Wesley McKinney. All rights reserved.

Printed in the United States of America.

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

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

Editor: 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

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

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

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

978-1-098-10396-5

[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 [Link to Come], 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 [Link to Come]. 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

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 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.

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 [Link to Come] 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:

```
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 [Link to Come]. 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.

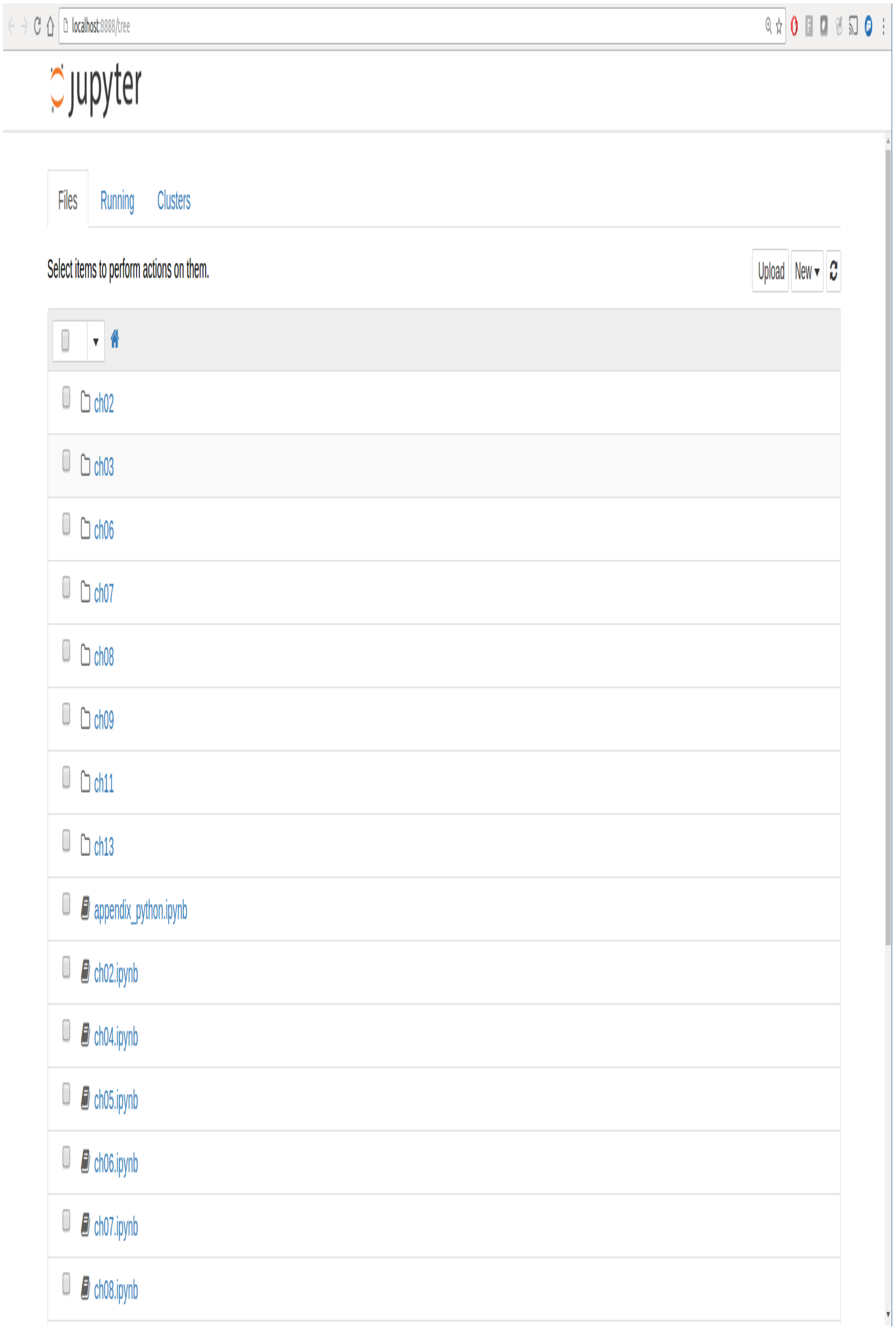
To start up Jupyter, run the command `jupyter notebook` in a terminal:

```
$ 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.

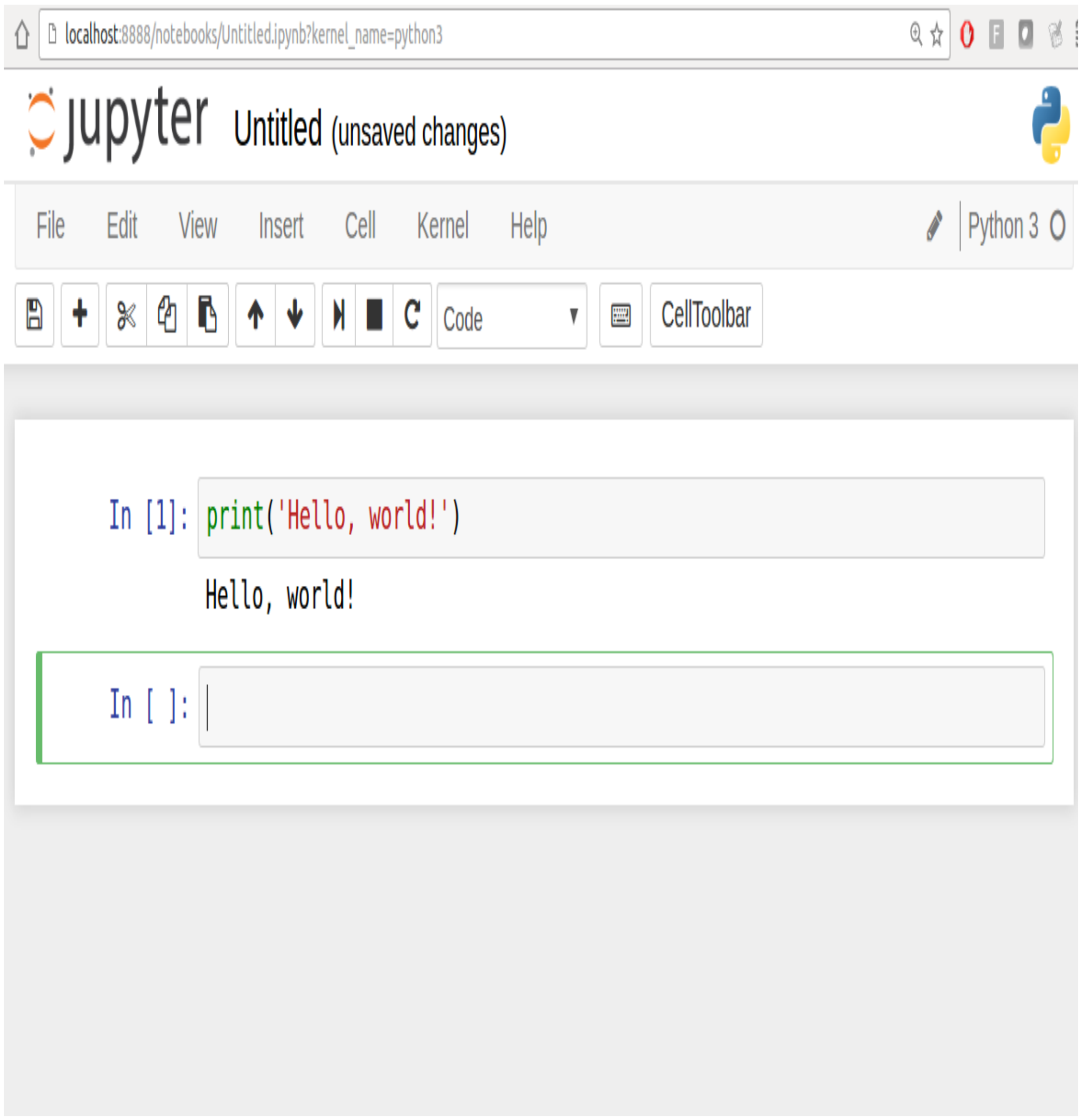


Select items to perform actions on them.

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.

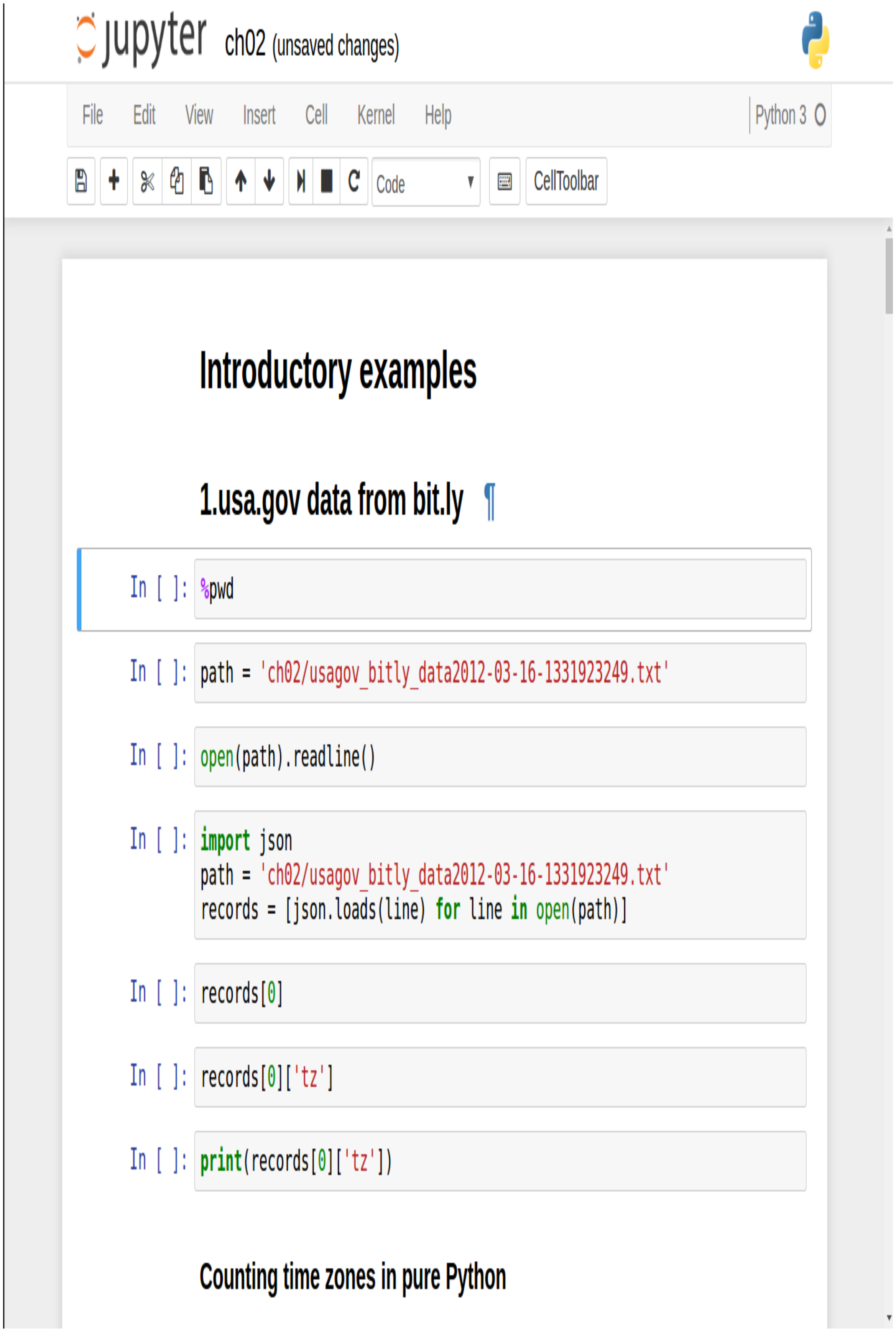

localhost:8888/notebooks/Untitled.ipynb?kernel_name=python3
jupyter
Untitled (unsaved changes)
File
Edit
View
Insert
Cell
Kernel
Help
Python 3
Code
CellToolbar

```
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.


jupyter ch02 (unsaved changes)
File Edit View Insert Cell Kernel Help
Python 3
Code
CellToolbar

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.

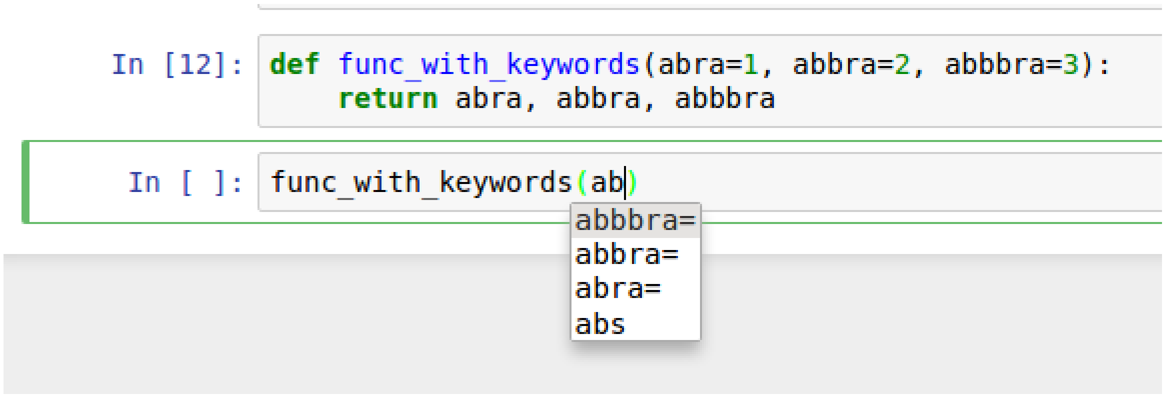
Tab completion works in many contexts outside of searching the interactive namespace and completing object or module attributes. When typing anything that looks like a file path (even in a Python string), pressing the Tab key will complete anything on your computer's filesystem matching what you've typed:

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

Figure 2-4. Autocomplete function keywords in Jupyter notebook

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

Introspection

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

```
In [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.

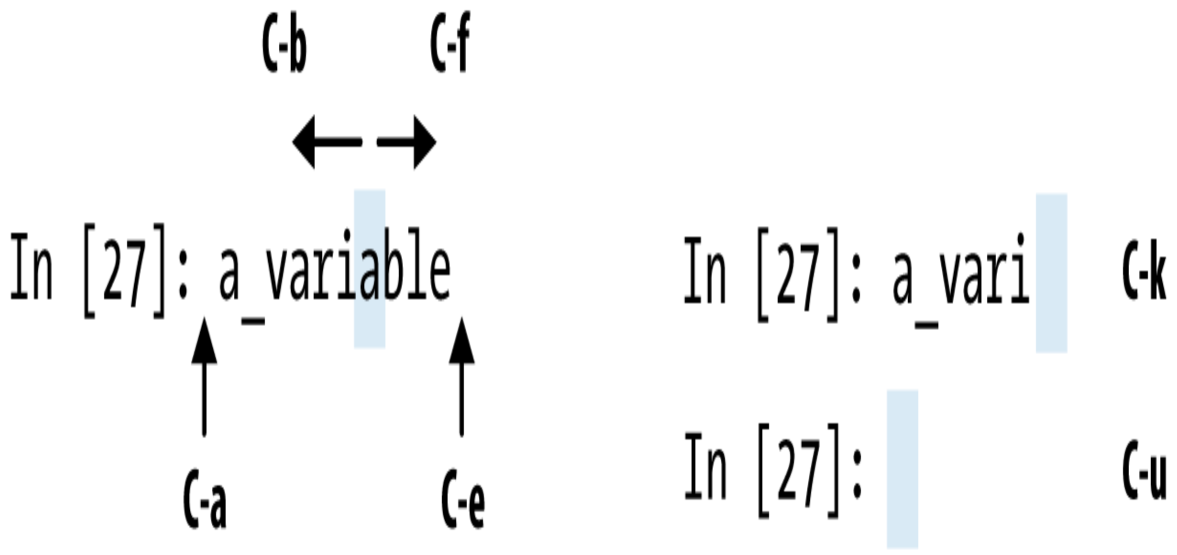


Figure 2-5. Illustration of some keyboard shortcuts in the IPython shell

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

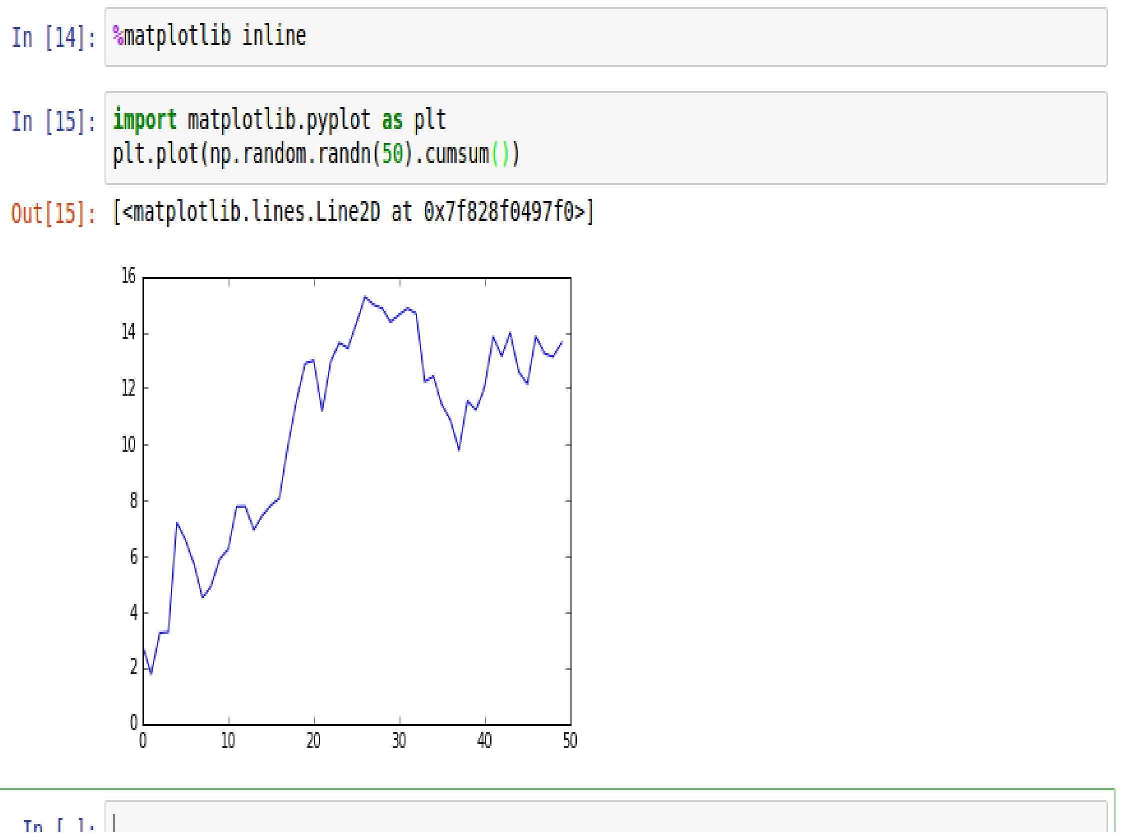
One reason for IPython's popularity in analytical computing is that it integrates well with data visualization and other user interface libraries like matplotlib. Don't worry if you have never used matplotlib before; it will be discussed in more detail later in this book. The `%matplotlib` magic function 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>]
```

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]
```

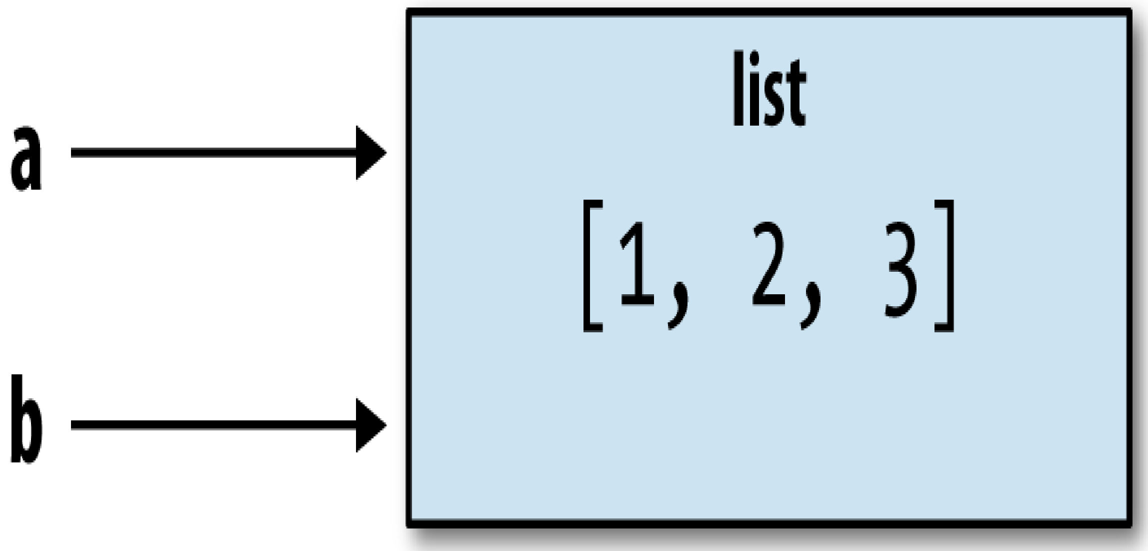


Figure 2-7. Two references for the same object

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:

```python
# 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:

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

Or equivalently:

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

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

```python
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.

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.

d. 2.4 Conclusion