Chapter 1. Preliminaries

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 languages,” 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) have 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 have appeared, 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, as 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 intuitive and flexible. 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 enable you to reshape, slice and dice, perform aggregations, and select subsets of data. Since data manipulation, preparation, and cleaning are such important skills 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 an integrated set of data structures and tools providing this functionality did not exist. 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 two thousand 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*.

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 for 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.

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.

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

In the accompanying book materials on GitHub, you will find Jupyter notebooks containing all the code examples from each chapter. If you cannot access GitHub where you are, you can try the mirror on Gitee.

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 2007, scikit-learn has become the premier general-purpose machine learning toolkit for Python programmers. As of this writing, more than two thousand 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

In 2022, 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 that 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 that 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 3.10 throughout, but if you're reading in the future, you are welcome to install a newer version of Python.

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 3.9) from *https://conda.io*. 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 3.9 [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc.
Type "help", "copyright", "credits" or "license" for mor
>>>
```

To exit the Python shell, 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. Most readers will want to download the default 64-bit installer file, which is for x86 architecture (but it's possible in the future more users will have aarch64-based Linux machines). The installer is a shell script that must be executed in the terminal. 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 (although 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 have 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 (select “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) $ python
Python 3.9 | (main) [GCC 10.3.0] on linux
Type "help", "copyright", "credits" or "license" for mor
>>>
```

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

Miniconda on macOS

Download the macOS Miniconda installer, which should be named something like *Miniconda3-latest-MacOSX-arm64.sh* for Apple Silicon-based macOS computers released from 2020 onward, or *Miniconda3-latest-MacOSX-x86_64.sh* for Intel-based Macs released before 2020. Open the Terminal application in macOS, and install by executing the installer (most likely in your `Downloads` directory) with `bash`:

```
$ bash $HOME/Downloads/Miniconda3-latest-MacOSX-arm64.sh
```

When the installer runs, by default it automatically configures Miniconda in your default shell environment in your default shell profile. This is probably located at */Users/$USER/.zshrc*. 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 3.9 (main) [Clang 12.0.1 ] on darwin
Type "help", "copyright", "credits" or "license" for mor
>>>
```

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

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" with the `conda create` command using Python 3.10:

```
(base) $ conda create -y -n pydata-book python=3.10
```

After the installation completes, activate the environment with `conda activate`:

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

NOTE

It is necessary to use `conda activate` to activate your environment each time you open a new terminal. You can see information about the active conda environment at any time from the terminal by running `conda info`.

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 matplotl
```

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`.

NOTE

If you want to install all of the packages used in the rest of the book, you can do that now by running:

```
conda install lxml beautifulsoup4 html5lib openpyxl \
               requests sqlalchemy seaborn scipy statsmodels \
               patsy scikit-learn pyarrow pytables numba
```

On Windows, substitute a carat `^` for the line continuation `\` used on Linux and macOS.

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 that are unavailable with `conda install`.

Integrated Development Environments 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 productive to use in the shell.

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

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

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

1.5 Community and Conferences

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

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

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

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

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

1.6 Navigating This Book

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

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

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

Interacting with the outside world

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

Preparation

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

Transformation

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

Modeling and computation

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

Presentation

Creating interactive or static graphical visualizations or textual summaries

Code Examples

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

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

When you see a code example like this, the intent is for you to type 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.

I changed the default console output settings in NumPy and pandas to improve readability and brevity throughout the book. For example, you may see more digits of precision printed in numeric data. To exactly match the output shown in the book, you can execute the following Python code before running the code examples:

```
import numpy as np
import pandas as pd
pd.options.display.max_columns = 20
pd.options.display.max_rows = 20
pd.options.display.max_colwidth = 80
np.set_printoptions(precision=4, suppress=True)
```

Data for Examples

Datasets for the examples in each chapter are hosted in a GitHub repository (or in a mirror on Gitee if you cannot access GitHub). 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 the book website for up-to-date instructions about obtaining the book materials.

If you download a zip file containing the example datasets, you must then fully extract the contents of the zip file to a directory and navigate to that directory from the terminal before proceeding with running the book's code examples:

```
$ pwd
/home/wesm/book-materials

$ ls
appa.ipynb  ch05.ipynb  ch09.ipynb  ch13.ipynb  README.m
ch02.ipynb  ch06.ipynb  ch10.ipynb  COPYING     requirem
ch03.ipynb  ch07.ipynb  ch11.ipynb  datasets
ch04.ipynb  ch08.ipynb  ch12.ipynb  examples
```

I have made every effort to ensure that the GitHub repository 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.