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.

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 10 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 support for 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.

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 seamless 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 high-performance, array-computing ideas of NumPy with the flexible data manipulation capabilities of spreadsheets and relational databases (such as SQL). It provides sophisticated indexing functionality to make it easy to reshape, slice and dice, perform aggregations, and select subsets of data. 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.

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 the most widely used and as such has generally good inte-

gration 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. In the subsequent 16 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 from the ground up to maximize your productivity in both interactive computing and software development. It encourages an *execute-explore* workflow instead of the typical *edit-compile-run* workflow of many other programming languages. It also provides easy access to your operating system's shell and filesystem. 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.

For me personally, IPython is usually involved with the majority of my Python work, including 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 different standard problem domains in scientific computing. Here is a sampling of the packages included:

`scipy.integrate`
: Numerical integration routines and differential equation solvers

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

`scipy.optimize`
: Function optimizers (minimizers) and root finding algorithms

`scipy.signal`
: Signal processing tools

`scipy.sparse`
: Sparse matrices and sparse linear system solvers

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

`scipy.stats`
: Standard continuous and discrete probability distributions (density functions, samplers, continuous distribution functions), various statistical tests, and more descriptive statistics

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. In just seven years, it has had over 1,500 contributors from around the world. 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.

1.4 Installation and Setup

Since everyone uses Python for different applications, there is no single solution for setting up Python and required add-on packages. Many readers will not have a complete 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 recommend using the free Anaconda distribution. At the time of this writing, Anaconda is offered in both Python 2.7 and 3.6 forms, though this might change at some point in the future. This book uses Python 3.6, and I encourage you to use Python 3.6 or higher.

Windows

To get started on Windows, download the Anaconda installer. I recommend following the installation instructions for Windows available on the Anaconda download page, which may have changed between the time this book was published and when you are reading this.

Now, let's verify that things are configured correctly. To open the Command Prompt application (also known as *cmd.exe*), right-click the Start menu and select Command Prompt. Try starting the Python interpreter by typing **`python`**. You should see a message that matches the version of Anaconda you installed:

```
C:\Users\wesm>python
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul  5 2016, 11:41:13)
[MSC v.1900 64 bit (AMD64)] on win32
>>>
```

To exit the shell, press Ctrl-D (on Linux or macOS), Ctrl-Z (on Windows), or type the command **`exit()`** and press Enter.

Apple (OS X, macOS)

Download the OS X Anaconda installer, which should be named something like *Anaconda3-4.1.0-MacOSX-x86_64.pkg*. Double-click the *.pkg* file to run the installer. When the installer runs, it automatically appends the Anaconda executable path to your *.bash_profile* file. This is located at */Users/$USER/.bash_profile*.

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

```
$ ipython
```

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 flavor, but here I give details for such distributions as Debian, Ubuntu, CentOS, and Fedora. Setup is similar to OS X with the exception of how Anaconda 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 *Anaconda3-4.1.0-Linux-x86_64.sh*. To install it, execute this script with bash:

```
$ bash Anaconda3-4.1.0-Linux-x86_64.sh
```

Some Linux distributions have versions of all the required Python packages in their package managers and can be installed using a tool like apt. The setup described here uses Anaconda, as it's both easily reproducible across distributions and simpler to upgrade packages to their latest versions.

After accepting the license, you will be presented with a choice of where to put the Anaconda files. I recommend installing the files in the default location in your home directory—for example, */home/$USER/anaconda* (with your username, naturally).

The Anaconda installer may ask if you wish to prepend its *bin/* directory to your `$PATH` variable. If you have any problems after installation, you can do this yourself by modifying your *.bashrc* (or *.zshrc*, if you are using the zsh shell) with something akin to:

```
export PATH=/home/$USER/anaconda/bin:$PATH
```

After doing this you can either start a new terminal process or execute your *.bashrc* again with `source ~/.bashrc`.

Installing or Updating Python Packages

At some point while reading, you may wish to install additional Python packages that are not included in the Anaconda distribution. In general, these can be installed with the following command:

```
conda install package_name
```

If this does not work, you may also be able to install the package using the pip package management tool:

```
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.

While you can use both conda and pip to install packages, you should not attempt to update conda packages with pip, as doing so can lead to environment problems. When using Anaconda or Miniconda, it's best to first try updating with conda.

Python 2 and Python 3

The first version of the Python 3.x line of interpreters was released at the end of 2008. It included a number of changes that made some previously written Python 2.x code incompatible. Because 17 years had passed since the very first release of Python in 1991, creating a "breaking" release of Python 3 was viewed to be for the greater good given the lessons learned during that time.

In 2012, much of the scientific and data analysis community was still using Python 2.x because many packages had not been made fully Python 3 compatible. Thus, the first edition of this book used Python 2.7. Now, users are free to choose between Python 2.x and 3.x and in general have full library support with either flavor.

However, Python 2.x will reach its development end of life in 2020 (including critical security patches), and so it is no longer a good idea to start new projects in Python 2.7. Therefore, this book uses Python 3.6, a widely deployed, well-supported stable release. We have begun to call Python 2.x "Legacy Python" and Python 3.x simply "Python." I encourage you to do the same.

This book uses Python 3.6 as its basis. Your version of Python may be newer than 3.6, but the code examples should be forward compatible. Some code examples may work differently or not at all in Python 2.7.

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 a comparatively primitive text editor like Emacs or Vim. 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 Atom 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 *http://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 the most incremental way possible, though there is occasionally some minor cross-over between chapters, with a few isolated cases where concepts are used that haven't necessarily 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.

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.

Jargon

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

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

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

Syntactic sugar
: Programming syntax that does not add new features, but makes something more convenient or easier to type.