Pearson Addison Wesley Data & Analytics Series

Pandas for Everyone

Python Data Analysis

DANIEL Y. CHEN

Pandas for Everyone

The Pearson Addison-Wesley Data and Analytics Series

Visit **informit.com/awdataseries** for a complete list of available publications.

The **Pearson Addison-Wesley Data and Analytics Series** provides readers with practical knowledge for solving problems and answering questions with data. Titles in this series primarily focus on three areas:

1. **Infrastructure:** how to store, move, and manage data
2. **Algorithms:** how to mine intelligence or make predictions based on data
3. **Visualizations:** how to represent data and insights in a meaningful and compelling way

The series aims to tie all three of these areas together to help the reader build end-to-end systems for fighting spam; making recommendations; building personalization; detecting trends, patterns, or problems; and gaining insight from the data exhaust of systems and user interactions.

Make sure to connect with us!
informit.com/socialconnect

Pearson Addison-Wesley

informIT.com
the trusted technology learning source

O'REILLY Safari

Pandas for Everyone

Python Data Analysis

Daniel Y. Chen

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2017956175

ISBN-13: 978-0-13-454693-3
ISBN-10: 0-13-454693-8

1 17

❦

To my family: Mom, Dad, Eric, and Julia

❦

This page intentionally left blank

Contents

This page intentionally left blank

Foreword

With each passing year data becomes more important to the world, as does the ability to compute on this growing abundance of data. When deciding how to interact with data, most people make a decision between R and Python. This does not reflect a language war but rather a luxury of choice where data scientists and engineers can work in the language with which they feel most comfortable. These tools make it possible for everyone to work with data for machine learning and statistical analysis. That is why I am happy to see what I started with *R for Everyone* extended to Python with *Pandas for Everyone*.

I first met Dan Chen when he stumbled into the "Introduction to Data Science" course while working toward a master's in public health at Columbia University's Mailman School of Public Health. He was part of a cohort of MPH students who cross-registered into the graduate school course and quickly developed a knack for data science, embracing statistical learning and reproducibility. By the end of the semester he was devoted to, and evangelizing, the merits of data science.

This coincided with the rise of Pandas, improving Python's use as a tool for data science and enabling engineers already familiar with the language to use it for data science as well. This fortuitous timing meant Dan developed into a true multilingual data scientist, mastering both R and Pandas. This puts him in a great position to reach different audiences, as shown by his frequent and popular talks at both R and Python conferences and meetups. His enthusiasm and knowledge shine through and resonate in everything he does, from educating new users to building Python libraries. Along the way he fully embraces the ethos of the open-source movement.

As the name implies, this book is meant for everyone who wants to use Python for data science, whether they are veteran Python users, experienced programmers, statisticians, or entirely new to the field. For people brand new to Python the book contains a collection of appendixes for getting started with the language and for installing both Python and Pandas, and it covers the whole analysis pipeline, including reading data, visualization, data manipulation, modeling, and machine learning.

Pandas for Everyone is a tour of data science through the lens of Python, and Dan Chen is perfectly suited to guide that tour. His mixture of academic and industry experience lends valuable insights into the analytics process and how Pandas should be used to greatest effect. All this combines to make for an enjoyable and informative read for everyone.

–Jared Lander, series editor

This page intentionally left blank

Preface

In 2013, I didn't even know the term "data science" existed. I was a master's of public health (MPH) student in epidemiology at the time and was already captivated with the statistical methods beyond the *t*-test, ANOVA, and linear regression from my psychology and neuroscience undergraduate background. It was also in the fall of 2013 that I attended my first Software-Carpentry workshop and that I taught my first recitation section as a teaching assistant for my MPH program's Quantitative Methods course (essentially a combination of a first-semester epidemiology and biostatistics course). I've been learning and teaching ever since.

I've come a long way since taking my first Introduction to Data Science course, which was taught by Rachel Schutt, PhD; Kayur Patel, PhD; and Jared Lander. They opened my eyes to what was possible. Things that were inconceivable (to me) were actually common practices, and anything I could think of was possible (although I now know that "possible" doesn't mean "performs well"). The technical details of data science—the coding aspects—were taught by Jared in R. Jared's friends and colleagues know how much of an aficionado he is of the R language.

At the time, I had been meaning to learn R, but the Python/R language war never breached my consciousness. On the one hand, I saw Python as just a programming language; on the other hand, I had no idea Python had an analytics stack (I've come a long way since then). When I learned about the SciPy stack and Pandas, I saw it as a bridge between what I knew how to do in Python from my undergraduate and high school days and what I had learned in my epidemiology studies and through my newly acquired data science knowledge. As I became more proficient in R, I saw the similarities to Python. I also realized that a lot of the data cleaning tasks (and programming in general) involve thinking about how to get what you need—the rest is more or less syntax. It's important to try to imagine what the steps are and not get bogged down by the programming details. I've always been comfortable bouncing around the languages and never gave too much thought to which language was "better." Having said that, this book is geared toward a newcomer to the Python data analytics world.

This book encapsulates all the people I've met, events I've attended, and skills I've learned over the past few years. One of the more important things I've learned (outside of knowing what things are called so Google can take me to the relevant StackOverflow page) is that reading the documentation is essential. As someone who has worked on collaborative lessons and written Python and R libraries, I can assure you that a lot of time and effort go into writing documentation. That's why I constantly refer to the relevant documentation page throughout this book. Some functions have so many parameters used for varying use cases that it's impractical to go through each of them. If that were the focus of this book, it might as well be titled *Loading Data Into Python*. But, as you practice working with data and become more comfortable with the various data structures, you'll eventually be able to make "educated guesses" about what the output of something will be, even though you've never written that particular line of code before. I hope this book gives you a solid foundation to explore on your own and be a self-guided learner.

I met a lot of people and learned a lot from them during the time I was putting this book together. A lot of the things I learned dealt with best practices, writing vectorized statements instead of loops, formally testing code, organizing project folder structures, and so on. I also learned lot about teaching from actually teaching. Teaching really is the best way to learn material. Many of the things I've learned in the past few years have come to me when I was trying to figure them out to teach others. Once you have a basic foundation of knowledge, learning the next bit of information is relatively easy. Repeat the process enough times, and you'll be surprised how much you actually know. That includes knowing the terms to use for Google and interpreting the StackOverflow answers. The very best of us all search for our questions. Whether this is your first language or your fourth, I hope this book gives you a solid foundation to build upon and learn as well as a bridge to other analytics languages.

Breakdown of the Book

This book is organized into five parts plus a set of appendixes.

Part I

Part I aims to be an introduction to Pandas using a realistic data set.

- Chapter 1: Starts by using Pandas to load a data set and begin looking at various rows and columns of the data. Here you will get a general sense of the syntax of Python and Pandas. The chapter ends with a series of motivating examples that illustrate what Pandas can do.
- Chapter 2: Dives deeper into what the Pandas `DataFrame` and `Series` objects are. This chapter also covers boolean subsetting, dropping values, and different ways to import and export data.
- Chapter 3: Covers plotting methods using `matplotlib`, `seaborn`, and Pandas to create plots for exploratory data analysis.

Part II

Part II focuses on what happens after you load data and need to combine data together. It also introduces "tidy data"—a series of data manipulations aimed at "cleaning" data.

- Chapter 4: Focuses on combining data sets, either by concatenating them together or by merging disparate data.
- Chapter 5: Covers what happens when there is missing data, how data are created to fill in missing data, and how to work with missing data, especially what happens when certain calculations are performed on them.
- Chapter 6: Discusses Hadley Wickham's "Tidy Data" paper, which deals with reshaping and cleaning common data problems.

Part III

Part III covers the topics needed to clean and munge data.

- Chapter 7: Deals with data types and how to convert from different types within `DataFrame columns`.
- Chapter 8: Introduces string manipulation, which is frequently needed as part of the data cleaning task because data are often encoded as text.
- Chapter 9: Focuses on applying functions over data, an important skill that encompasses many programming topics. Understanding how `apply` works will pave the way for more parallel and distributed coding when your data manipulations need to scale.
- Chapter 10: Describes groupby operations. These powerful concepts, like `apply`, are often needed to scale data. They are also great ways to efficiently aggregate, transform, or filter your data.
- Chapter 11: Explores Pandas's powerful date and time capabilities.

Part IV

With the data all cleaned and ready, the next step is to fit some models. Models can be used for exploratory purposes, not just for prediction, clustering, and inference. The goal of Part IV is not to teach statistics (there are plenty of books in that realm), but rather to show you how these models are fit and how they interface with Pandas. Part IV can be used as a bridge to fitting models in other languages.

- Chapter 12: Linear models are the simpler models to fit. This chapter covers fitting these models using the `statsmodels` and `sklean` libraries.
- Chapter 13: Generalized linear models, as the name suggests, are linear models specified in a more general sense. They allow us to fit models with different response variables, such as binary data or count data. This chapter also covers survival models.
- Chapter 14: Since we have a core set of models that we can fit, the next step is to perform some model diagnostics to compare multiple models and pick the "best" one.
- Chapter 15: Regularization is a technique used when the models we are fitting are too complex or overfit our data.
- Chapter 16: Clustering is a technique we use when we don't know the actual answer within our data, but we need a method to cluster or group "similar" data points together.

Part V

The book concludes with a few points about the larger Python ecosystem, and additional references.

- Chapter 17: Quickly summarizes the computation stack in Python, and starts down the path to code performance and scaling.
- Chapter 18: Provides some links and references on learning beyond the book.

Appendixes

The appendixes can be thought as a primer to Python programming. While they are not a complete introduction to Python, the various appendixes do supplement some of the topics throughout the book.

- Appendixes A–G: These appendixes cover all the tasks related to running Python code—from installing Python, to using the command line to execute your scripts, and to organizing your code. They also cover creating Python environments and installing libraries.
- Appendixes H–T: The appendixes cover general programming concepts that are relevant to Python and Pandas. They are supplemental references to the main part of the book.

How to Read This Book

Whether you are a newcomer to Python or a fluent Python programmer, this book is meant to be read from the beginning. Educators, or people who plan to use the book for teaching, may also find the order of the chapters to be suitable for a workshop or class.

Newcomers

Absolute newcomers are encouraged to first look through Appendixes A–F, as they explain how to install Python and get it working. After taking these steps, readers will be ready to jump into the main body of the book. The earlier chapters make references to the relevant appendixes as needed. The concept map and objectives found at the beginning of the earlier chapters help organize and prepare the reader for what will be covered in the chapter, as well as point to the relevant appendixes to be read before continuing.

Fluent Python Programmers

Fluent Python programmers may find the first two chapters to be sufficient to get started and grasp the syntax of Pandas; they can then use the rest of the book as a reference. The objectives at the beginning of the earlier chapters point out which topics are covered in the chapter. The chapter on "tidy data" in Part II, and the chapters in Part III, will be particularly helpful in data manipulation.

Instructors

Instructors who want to use the book as a teaching reference may teach each chapter in the order presented. It should take approximately 45 minutes to 1 hour to teach each chapter. I have sought to structure the book so that chapters do not reference future chapters, so as to minimize the cognitive overload for students—but feel free to shuffle the chapters as needed.

Setup

Everyone will have a different setup, so the best way to get the most updated set of instructions on setting up an environment to code through the book would be on the accompanying GitHub repository:

```
https://github.com/chendaniely/pandas_for_everyone
```

Otherwise, see Appendix A for information on how to install Python on your computer.

Getting the Data

The easiest way to get all the data to code along the book is to download the repository using the following URL:

```
https://github.com/chendaniely/pandas_for_everyone/archive/master.zip
```

This will download everything in the repository, as well as provide a folder in which you can put your Python scripts or notebooks. You can also copy the data folder from the repository and put it in a folder of your choosing. The instructions on the GitHub repository will be updated as necessary to facilitate downloading the data for the book.

Setting up Python

Appendixes F and G cover environments and installing packages, respectively. Following are the commands used to build the book and should be sufficient to help you get started.

```
$ conda create -n book python=3.6
$ source activate book
$ conda install pandas xlwt openpyxl feather -format seaborn numpy \
ipython jupyter statsmodels scikit-learnregex \
wget odo numba
$ conda install -c conda-forge pweave
$ pip install lifelines
$ pip install pandas-datareader
```

Feedback, Please!

Thank you for taking the time to go through this book. If you find any problems, issues, or mistakes within the book, please send me feedback! GitHub issues may be the best place to provide this information, but you can also email me at chendaniely@gmail.com. Just be sure to use the [PFE] tag in the beginning of the subject line so I can make sure your emails do not get flooded by various listserv emails. If there are topics that you feel should be covered in the book, please let me know. I will try my best to put up a notebook in the GitHub repository, and to get it incorporated in a later printing or edition of the book.

Words of encouragement are appreciated.

Register your copy of *Pandas for Everyone* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780134546933) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

L
PANDAS
DE
OPEN
10
CEN

Acknowledgments

Introduction to Data Science: The three people who paved the way for this book were my instructors in the "Introduction to Data Science" course at Columbia—Rachel Schutt, Kayur Patel, and Jared Lander. Without them, I wouldn't even know what the term "data science" means. I learned so much about the field through their lectures and labs; everything I know and do today can be traced back to this class. The instructors were only part of the learning process. The people in my study group, where we fumbled through our homework assignments and applied our skills to the final project of summarizing scientific articles, made learning the material and passing the class possible. They were Niels Bantilan, Thomas Vo, Vivian Peng, and Sabrina Cheng (depicted in the figure here). Perhaps unsurprisingly, they also got me through my master's program (more on that later).

One of the midnight doodles by Vivian Peng for our project group. We have Niels, our project leader, at the top; Thomas, me, and Sabrina in the middle row; and Vivian at the bottom.

Software-Carpentry: As part of the "Introduction to Data Science" course, I attended a Software-Carpentry workshop, where I was first introduced to Pandas. My first instructors were Justin Ely and David Warde-Farley. Since then I've been involved in the community, thanks to Greg Wilson, and still remember the first class I helped teach, led by Aron Ahmadia and Randal S. Olson. The many workshops that I've taught since then, and the fellow instructors whom I've met, gave me the opportunity to master the knowledge and skills I know and practice today, and to disseminate them to new learners, which has cumulated into this book.

Software-Carpentry also introduced me to the NumFOCUS, PyData, and the Scientific Python communities, where all my (Python) heroes can be found. There are too many to list here. My connection to the R world is all thanks to Jared Lander.

Columbia University Mailman School of Public Health: My undergraduate study group evolved into a set of lifelong friends during my master's program. The members of this group got me through the first semester of the program in which epidemiology and biostatistics were first taught. The knowledge I learned in this program later transferred into my knowledge of machine learning. Thanks go to Karen Lin, Sally Cheung, Grace Lee, Wai Yee (Krystal) Khine, Ashley Harper, and Jacquie Cheung. A second set of thanks to go to my old study group alumni: Niels Bantilan, Thomas Vo, and Sabrina Cheng.

To my instructors, Katherine Keyes and Martina Pavlicova, thanks for being exemplary teachers in epidemiology, and biostatistics, respectively. Thanks also to Dana March Palmer, for whom I was a TA and who gave me my first teaching experience. Mark Orr served as my thesis advisor while I was at Mailman. The department of epidemiology had a subset of faculty who did computational and simulation modeling, under the leadership of Sandro Galea, the department chair at the time. After graduation, I got my first job as a data analyst with Jacqueline Merrill at the Columbia University School of Nursing.

Getting to Mailman was a life-altering event. I never would have considered entering an MPH program if it weren't for Ting Ting Guo. As an advisor, Charlotte Glasser was a tremendous help to me in planning out my frequent undergraduate major changes and postgraduate plans.

Virginia Tech: The people with whom I work at the Social and Decision Analytics Laboratory (SDAL) have made Virginia Tech one of the most enjoyable places where I've worked. A second thanks to Mark Orr, who got me here. The administrators of the lab, Kim Lyman and Lori Conerly, make our daily lives that much easier. Sallie Keller and Stephanie Shipp, the director and the deputy lab director, respectively, create a collaborative work environment. The rest of the lab members, past and present (in no particular order)—David Higdon, Gizem Korkmaz, Vicki Lancaster, Mark Orr, Bianca Pires, Aaron Schroeder, Ian Crandell, Joshua Goldstein, Kathryn Ziemer, Emily Molfino, and Ana Aizcorbe—also work hard at making my graduate experience fun. It's also been a pleasure to train and work with the summer undergraduate and graduate students in the lab through the Data Science for the Public Good program. I've learned a lot about teaching and implementing good programming practices. Finally, Brian Goode adds to my experience progressing though the program by always being available to talk about various topics.

The people down in Blacksburg, Virginia, where most of the book was written, have kept me grounded during my coursework. My PhD cohort—Alex Song Qi, Amogh Jalihal, Brittany Boribong, Bronson Weston, Jeff Law, and Long Tian—have always found time for me, and for one another, and offered opportunities to disconnect from the PhD grind. I appreciate their willingness to work to maintain our connections, despite being in an interdisciplinary program where we don't share many classes together, let alone labs.

Brian Lewis and Caitlin Rivers helped me initially get settled in Blacksburg and gave me a physical space to work in the Network Dynamics and Simulation Science Laboratory. Here, I met Gloria Kang, Pyrros (Alex) Telionis, and James Schlitt, who have given me creative and emotional outlets the past few years. NDSSL has also provided and/or been involved with putting together some of the data sets used in the book.

Last but not least, Dennie Munson, my program liaison, can never be thanked enough for putting up with all my shenanigans.

Book Publication Process: Debra Williams Cauley, thank you so much for giving me this opportunity to contribute to the Python and data science community. I've grown tremendously as an educator during this process, and this adventure has opened more doors for me than the number of times I've missed deadlines. A second thanks to Jared Lander for recommending me and putting me up for the task.

Even more thanks go to Gloria Kang, Jacquie Cheung, and Jared Lander for their feedback during the writing process. I also want to thank Chris Zahn for all the work in reviewing the book from cover to cover, and Kaz Sakamoto and Madison Arnsbarger for providing feedback and reviews. Through their many conversations with me, M Pacer, Sebastian Raschka, Andreas Müller, and Tom Augspurger helped me make sure I covered my bases, and did things "properly."

Thanks to all the people involved in the post-manuscript process: Julie Nahil (production editor), Jill Hobbs (copy editor), Rachel Paul (project manager and proofreader), Jack Lewis (indexer), and SPi Global (compositor). Y'all have been a pleasure to work with. More importantly, you polished my writing when it needed a little help and made sure the book was formatted consistently.

Family: My immediate and extended family have always been close. It is always a pleasure when we are together for holidays or random cookouts. It's always surprising how the majority of the 50-plus of us manage to regularly get together throughout the year. I am extremely lucky to have the love and support from this wonderful group of people.

To my younger siblings, Eric and Julia: It's hard being an older sibling! The two of you have always pushed me to be a better person and role model, and you bring humor, joy, and youth into my life.

A second thanks to my sister for providing the drawings in the preface and the appendix.

Last but not least, thank you, Mom and Dad, for all your support over the years. I've had a few last-minute career changes, and you have always been there to support my decisions, financially, emotionally, and physically—including helping me relocate between cities. Thanks to the two of you, I've always been able to pursue my ambitions while knowing full well I can count on your help along the way. This book is dedicated to you.

This page intentionally left blank

About the Author

Daniel Chen is a research associate and data engineer at the Social and Decision Analytics Laboratory at the Biocomplexity Institute of Virginia Tech. He is pursuing a PhD in the interdisciplinary program in Genetics, Bioinformatics, and Computational Biology (GBCB). He completed his master's in public health (MPH in epidemiology) at Columbia University Mailman School of Public Health, where he looked at attitude diffusion in social networks. His current research interest is repurposing administrative data to inform policy decision-making. He is a data scientist at Lander Analytics, an instructor and lesson maintainer for Software Carpentry and Data Carpentry, and a course instructor for DataCamp. In a previous life, he studied psychology and neuroscience and worked in a bench laboratory doing microscopy work looking at proteins in the brain associated with learning and memory.

This page intentionally left blank

6

Tidy Data

6.1 Introduction

As mentioned in Chapter 4, Hadley Wickham,[1] one of the more prominent members of the R community, introduced the concept of *tidy data* in a paper in the *Journal of Statistical Software*.[2] Tidy data is a framework to structure data sets so they can be easily analyzed and visualized. It can be thought of as a goal one should aim for when cleaning data. Once you understand what tidy data is, that knowledge will make your data analysis, visualization, and collection much easier.

What is *tidy* data? Hadley Wickham's paper defines it as meeting the following criteria:

- Each row is an observation.
- Each column is a variable.
- Each type of observational unit forms a table.

This chapter goes through the various ways to tidy data as identified in Wickham's paper.

Concept Map

Prior knowledge:

a. function and method calls
b. subsetting data
c. loops
d. list comprehension

This chapter:

- Reshaping data
    a. unpivot/melt/gather
    b. pivot/cast/spread

1. Hadley Wickham: `http://hadley.nz/`
2. Tidy data paper: `http://vita.had.co.nz/papers/tidy-data.pdf`

   c. subsetting
   d. combining
      1. globbing
      2. concatenation

Objectives

This chapter will cover:

1. Unpivoting/melting/gathering columns into rows
2. Pivoting/casting/spreading rows into columns
3. Normalizing data by separating a dataframe into multiple tables
4. Assembling data from multiple parts

6.2 Columns Contain Values, Not Variables

Data can have columns that contain values instead of variables. This is usually a convenient format for data collection and presentation.

6.2.1 Keep One Column Fixed

We'll use data on income and religion in the United States from the Pew Research Center to illustrate how to work with columns that contain values, rather than variables.

```
import pandas as pd
pew = pd.read_csv('../data/pew.csv')
```

When we look at this data set, we can see that not every column is a variable. The values that relate to income are spread across multiple columns. The format shown is a great choice when presenting data in a table, but for data analytics, the table needs to be reshaped so that we have religion, income, and count variables.

```
# show only the first few columns
print(pew.iloc[:, 0:6])
```

```
                   religion  <$10k  $10-20k  $20-30k  $30-40k  \
0                  Agnostic     27       34       60       81
1                   Atheist     12       27       37       52
2                  Buddhist     27       21       30       34
3                  Catholic    418      617      732      670
4        Don't know/refused     15       14       15       11
5          Evangelical Prot    575      869     1064      982
6                     Hindu      1        9        7        9
7   Historically Black Prot    228      244      236      238
8         Jehovah's Witness     20       27       24       24
9                    Jewish     19       19       25       25
10            Mainline Prot    289      495      619      655
11                   Mormon     29       40       48       51
12                   Muslim      6        7        9       10
```

```
13                  Orthodox     13        17        23        32
14           Other Christian      9         7        11        13
15              Other Faiths     20        33        40        46
16     Other World Religions      5         2         3         4
17               Unaffiliated    217       299       374       365

    $40-50k
0        76
1        35
2        33
3       638
4        10
5       881
6        11
7       197
8        21
9        30
10      651
11       56
12        9
13       32
14       13
15       49
16        2
17      341
```

This view of the data is also known as "wide" data. To turn it into the "long" tidy data format, we will have to unpivot/melt/gather (depending on which statistical programming language we use) our dataframe. Pandas has a function called `melt` that will reshape the dataframe into a tidy format. `melt` takes a few parameters:

- `id_vars` is a container (list, tuple, ndarray) that represents the variables that will remain as is.
- `value_vars` identifies the columns you want to melt down (or unpivot). By default, it will melt all the columns not specified in the `id_vars` parameter.
- `var_name` is a string for the new column name when the `value_vars` is melted down. By default, it will be called `variable`.
- `value_name` is a string for the new column name that represents the values for the `var_name`. By default, it will be called `value`.

```
# we do not need to specify a value_vars since we want to pivot
# all the columns except for the 'religion' column
pew_long = pd.melt(pew, id_vars='religion')

print(pew_long.head())
```

```
              religion variable  value
0             Agnostic    <$10k     27
1              Atheist    <$10k     12
2             Buddhist    <$10k     27
```

```
3            Catholic    <$10k    418
4  Don't know/refused    <$10k     15
```

```
print(pew_long.tail())
```

```
                    religion            variable  value
175                 Orthodox  Don't know/refused     73
176          Other Christian  Don't know/refused     18
177             Other Faiths  Don't know/refused     71
178  Other World Religions  Don't know/refused      8
179             Unaffiliated  Don't know/refused    597
```

We can change the defaults so that the melted/unpivoted columns are named.

```
pew_long = pd.melt(pew,
                   id_vars='religion',
                   var_name='income',
                   value_name='count')
print(pew_long.head())
```

```
             religion income  count
0            Agnostic  <$10k     27
1             Atheist  <$10k     12
2            Buddhist  <$10k     27
3            Catholic  <$10k    418
4  Don't know/refused  <$10k     15
```

```
print(pew_long.tail())
```

```
                    religion              income  count
175                 Orthodox  Don't know/refused     73
176          Other Christian  Don't know/refused     18
177             Other Faiths  Don't know/refused     71
178  Other World Religions  Don't know/refused      8
179             Unaffiliated  Don't know/refused    597
```

6.2.2 Keep Multiple Columns Fixed

Not every data set will have one column to hold still while you unpivot the rest of the columns. As an example, consider the Billboard data set.

```
billboard = pd.read_csv('../data/billboard.csv')

# look at the first few rows and columns
print(billboard.iloc[0:5, 0:16])
```

```
   year        artist                    track  time date.entered  \
0  2000         2 Pac  Baby Don't Cry (Keep...  4:22   2000-02-26
1  2000       2Ge+her  The Hardest Part Of ...  3:15   2000-09-02
2  2000  3 Doors Down               Kryptonite  3:53   2000-04-08
3  2000  3 Doors Down                    Loser  4:24   2000-10-21
4  2000      504 Boyz            Wobble Wobble  3:35   2000-04-15
```

```
   wk1   wk2   wk3   wk4   wk5   wk6   wk7   wk8   wk9  wk10  wk11
0   87  82.0  72.0  77.0  87.0  94.0  99.0   NaN   NaN   NaN   NaN
1   91  87.0  92.0   NaN   NaN   NaN   NaN   NaN   NaN   NaN   NaN
2   81  70.0  68.0  67.0  66.0  57.0  54.0  53.0  51.0  51.0  51.0
3   76  76.0  72.0  69.0  67.0  65.0  55.0  59.0  62.0  61.0  61.0
4   57  34.0  25.0  17.0  17.0  31.0  36.0  49.0  53.0  57.0  64.0
```

You can see here that each week has its own column. Again, there is nothing *wrong* with this form of data. It may be easy to enter the data in this form, and it is much quicker to understand what it means when the data is presented in a table. However, there may be a time when you will need to melt the data. For example, if you wanted to create a faceted plot of the weekly ratings, the facet variable would need to be a column in the dataframe.

```
billboard_long = pd.melt(
    billboard,
    id_vars=['year', 'artist', 'track', 'time', 'date.entered'],
    var_name='week',
    value_name='rating')

print(billboard_long.head())
```

```
   year        artist                    track  time date.entered  \
0  2000         2 Pac  Baby Don't Cry (Keep...  4:22   2000-02-26
1  2000       2Ge+her  The Hardest Part Of ...  3:15   2000-09-02
2  2000  3 Doors Down               Kryptonite  3:53   2000-04-08
3  2000  3 Doors Down                    Loser  4:24   2000-10-21
4  2000      504 Boyz            Wobble Wobble  3:35   2000-04-15

  week  rating
0  wk1    87.0
1  wk1    91.0
2  wk1    81.0
3  wk1    76.0
4  wk1    57.0
```

```
print(billboard_long.tail())
```

```
       year            artist                    track  time  \
24087  2000       Yankee Grey     Another Nine Minutes  3:10
24088  2000  Yearwood, Trisha          Real Live Woman  3:55
24089  2000   Ying Yang Twins  Whistle While You Tw...  4:19
24090  2000     Zombie Nation            Kernkraft 400  3:30
24091  2000   matchbox twenty                     Bent  4:12

      date.entered  week  rating
24087   2000-04-29  wk76     NaN
24088   2000-04-01  wk76     NaN
24089   2000-03-18  wk76     NaN
24090   2000-09-02  wk76     NaN
24091   2000-04-29  wk76     NaN
```

6.3 Columns Contain Multiple Variables

Sometimes columns in a data set may represent multiple variables. This format is commonly seen when working with health data, for example. To illustrate this situation, let's look at the Ebola data set.

```
ebola = pd.read_csv('../data/country_timeseries.csv')
print(ebola.columns)
```

```
Index(['Date', 'Day', 'Cases_Guinea', 'Cases_Liberia',
       'Cases_SierraLeone', 'Cases_Nigeria', 'Cases_Senegal',
       'Cases_UnitedStates', 'Cases_Spain', 'Cases_Mali',
       'Deaths_Guinea', 'Deaths_Liberia', 'Deaths_SierraLeone',
       'Deaths_Nigeria', 'Deaths_Senegal', 'Deaths_UnitedStates',
       'Deaths_Spain', 'Deaths_Mali'],
      dtype='object')
```

```
# print select rows
print(ebola.iloc[:5, [0, 1, 2, 3, 10, 11]])
```

```
         Date  Day  Cases_Guinea  Cases_Liberia  Deaths_Guinea  \
0    1/5/2015  289        2776.0            NaN         1786.0
1    1/4/2015  288        2775.0            NaN         1781.0
2    1/3/2015  287        2769.0         8166.0         1767.0
3    1/2/2015  286           NaN         8157.0            NaN
4  12/31/2014  284        2730.0         8115.0         1739.0

   Deaths_Liberia
0             NaN
1             NaN
2          3496.0
3          3496.0
4          3471.0
```

The column names `Cases_Guinea` and `Deaths_Guinea` actually contain two variables. The individual status (cases and deaths, respectively) as well as the country name, Guinea. The data is also arranged in a wide format that needs to be unpivoted.

```
ebola_long = pd.melt(ebola, id_vars=['Date', 'Day'])
print(ebola_long.head())
```

```
         Date  Day      variable   value
0    1/5/2015  289  Cases_Guinea  2776.0
1    1/4/2015  288  Cases_Guinea  2775.0
2    1/3/2015  287  Cases_Guinea  2769.0
3    1/2/2015  286  Cases_Guinea     NaN
4  12/31/2014  284  Cases_Guinea  2730.0
```

```
print(ebola_long.tail())
```

```
           Date  Day     variable  value
1947  3/27/2014    5  Deaths_Mali    NaN
1948  3/26/2014    4  Deaths_Mali    NaN
```

```
1949  3/25/2014    3  Deaths_Mali     NaN
1950  3/24/2014    2  Deaths_Mali     NaN
1951  3/22/2014    0  Deaths_Mali     NaN
```

6.3.1 Split and Add Columns Individually (Simple Method)

Conceptually, the column of interest can be split based on the underscore in the column name, _. The first part will be the new status column, and the second part will be the new country column. This will require some string parsing and splitting in Python (more on this in Chapter 8). In Python, a string is an object, similar to how Pandas has `Series` and `DataFrame` objects. Chapter 2 showed how `Series` can have method such as `mean`, and `DataFrame`s can have methods such as `to_csv`. Strings have methods as well. In this case we will use the `split` method that takes a string and splits the string up based on a given delimiter. By default, `split` will split the string based on a space, but we can pass in the underscore, _, in our example. To get access to the string methods, we need to use the `str` accessor (see Chapter 8 for more on strings). This will give us access to the Python string methods and allow us to work across the entire column.

```
# get the variable column
# access the string methods
# and split the column based on a delimiter
variable_split = ebola_long.variable.str.split('_')
```

```
print(variable_split[:5])
```

```
0    [Cases, Guinea]
1    [Cases, Guinea]
2    [Cases, Guinea]
3    [Cases, Guinea]
4    [Cases, Guinea]
Name: variable, dtype: object
```

```
print(variable_split[-5:])
```

```
1947    [Deaths, Mali]
1948    [Deaths, Mali]
1949    [Deaths, Mali]
1950    [Deaths, Mali]
1951    [Deaths, Mali]
Name: variable, dtype: object
```

After we split on the underscore, the values are returned in a list. We know it's a list because that's how the split method works,[3] but the visual cue is that the results are surrounded by square brackets.

```
# the entire container
print(type(variable_split))
```

```
<class 'pandas.core.series.Series'>
```

3. String `split` documentation:
https://docs.python.org/3.6/library/stdtypes.html#str.split

```
# the first element in the container
print(type(variable_split[0]))
```

```
<class 'list'>
```

Now that the column has been split into the various pieces, the next step is to assign those pieces to a new column. First, however, we need to extract all the 0-index elements for the `status` column and the 1-index elements for the country column. To do so, we need to access the string methods again, and then use the `get` method to get the index we want for each row.

```
status_values = variable_split.str.get(0)
country_values = variable_split.str.get(1)
```

```
print(status_values[:5])
```

```
0    Cases
1    Cases
2    Cases
3    Cases
4    Cases
Name: variable, dtype: object
```

```
print(status_values[-5:])
```

```
1947    Deaths
1948    Deaths
1949    Deaths
1950    Deaths
1951    Deaths
Name: variable, dtype: object
```

```
print(country_values[:5])
```

```
0    Guinea
1    Guinea
2    Guinea
3    Guinea
4    Guinea
Name: variable, dtype: object
```

```
print(country_values[-5:])
```

```
1947    Mali
1948    Mali
1949    Mali
1950    Mali
1951    Mali
Name: variable, dtype: object
```

Now that we have the vectors we want, we can add them to our dataframe.

```
ebola_long['status'] = status_values
ebola_long['country'] = country_values
```

```
print(ebola_long.head())
```

```
         Date  Day      variable   value status country
0    1/5/2015  289  Cases_Guinea  2776.0  Cases  Guinea
1    1/4/2015  288  Cases_Guinea  2775.0  Cases  Guinea
2    1/3/2015  287  Cases_Guinea  2769.0  Cases  Guinea
3    1/2/2015  286  Cases_Guinea     NaN  Cases  Guinea
4  12/31/2014  284  Cases_Guinea  2730.0  Cases  Guinea
```

6.3.2 Split and Combine in a Single Step (Simple Method)

In this subsection, we'll exploit the fact that the vector returned is in the same order as our data. We can concatenate (see Chapter 4) the new vector or our original data.

```
variable_split = ebola_long.variable.str.split('_', expand=True)
variable_split.columns = ['status', 'country']
ebola_parsed = pd.concat([ebola_long, variable_split], axis=1)
```

```
print(ebola_parsed.head())
```

```
         Date  Day      variable   value status country status  \
0    1/5/2015  289  Cases_Guinea  2776.0  Cases  Guinea  Cases
1    1/4/2015  288  Cases_Guinea  2775.0  Cases  Guinea  Cases
2    1/3/2015  287  Cases_Guinea  2769.0  Cases  Guinea  Cases
3    1/2/2015  286  Cases_Guinea     NaN  Cases  Guinea  Cases
4  12/31/2014  284  Cases_Guinea  2730.0  Cases  Guinea  Cases

  country
0  Guinea
1  Guinea
2  Guinea
3  Guinea
4  Guinea
```

```
print(ebola_parsed.tail())
```

```
           Date  Day     variable  value  status country  status  \
1947  3/27/2014    5  Deaths_Mali    NaN  Deaths    Mali  Deaths
1948  3/26/2014    4  Deaths_Mali    NaN  Deaths    Mali  Deaths
1949  3/25/2014    3  Deaths_Mali    NaN  Deaths    Mali  Deaths
1950  3/24/2014    2  Deaths_Mali    NaN  Deaths    Mali  Deaths
1951  3/22/2014    0  Deaths_Mali    NaN  Deaths    Mali  Deaths

     country
1947    Mali
1948    Mali
1949    Mali
1950    Mali
1951    Mali
```

6.3.3 Split and Combine in a Single Step (More Complicated Method)

In this subsection, we'll again exploit the fact that the vector returned is in the same order as our data. We can concatenate (see Chapter 4) the new vector or our original data.

We can accomplish the same result in a single step by taking advantage of the fact that the split results return a list of two elements, where each element is a new column. We can combine the list of split items with the built-in `zip` function. `zip` takes a set of iterators (e.g., lists, tuples) and creates a new container that is made of the input iterators, but each new container created has the same index as the input containers. For example, if we have two lists of values,

```
constants = ['pi', 'e']
values = ['3.14', '2.718']
```

we can zip the values together:

```
# we have to call list on the zip function
# to show the contents of the zip object
# in Python 3, zip returns an iterator
print(list(zip(constants, values)))
```

```
[('pi', '3.14'), ('e', '2.718')]
```

Each element now has the constant matched with its corresponding value. Conceptually, each container is like a side of a zipper. When we `zip` the containers, the indices are matched up and returned.

Another way to visualize what `zip` is doing is taking each container passed into `zip` and stacking the containers on top of each other (think about the row-wise concatenation described in Section 4.3.1), thereby creating a dataframe of sorts. `zip` then returns the values on a column-by-column basis in a tuple.

We can use the same `ebola_long.variable.str.split(' ')` to split the values in the column. However, since the result is already a container (a `Series` object), we need to unpack it so that we have the contents of the container (each status–country list), rather than the container itself (the series).

In Python, the asterisk operator, `*`, is used to unpack containers.[4] When we `zip` the unpacked containers, the effect is the same as when we created the status values and the country values earlier. We can then assign the vectors to the columns simultaneously using multiple assignment (Appendix Q).

```
ebola_long['status'], ebola_long['country'] = \
    zip(*ebola_long.variable.str.split('_'))

print(ebola_long.head())
```

```
        Date  Day      variable   value status country
0   1/5/2015  289  Cases_Guinea  2776.0  Cases  Guinea
```

4. Unpacking argument lists:
`https://docs.python.org/3/tutorial/controlflow.html#unpacking-argument-lists`

```
1    1/4/2015  288  Cases_Guinea  2775.0  Cases  Guinea
2    1/3/2015  287  Cases_Guinea  2769.0  Cases  Guinea
3    1/2/2015  286  Cases_Guinea     NaN  Cases  Guinea
4  12/31/2014  284  Cases_Guinea  2730.0  Cases  Guinea
```

6.4 Variables in Both Rows and Columns

At times data will be formatted so that variables are in both rows and columns—that is, in some combination of the formats described in previous sections of this chapter. Most of the methods needed to tidy up such data have already been presented. What is left to show is what happens if a column of data actually holds two variables instead of one variable. In this case, we will have to pivot or cast the variable into separate columns.

```
weather = pd.read_csv('../data/weather.csv')
print(weather.iloc[:5, :11])
```

```
        id  year  month element  d1    d2    d3  d4    d5  d6  d7
0  MX17004  2010      1    tmax NaN   NaN   NaN NaN   NaN NaN NaN
1  MX17004  2010      1    tmin NaN   NaN   NaN NaN   NaN NaN NaN
2  MX17004  2010      2    tmax NaN  27.3  24.1 NaN   NaN NaN NaN
3  MX17004  2010      2    tmin NaN  14.4  14.4 NaN   NaN NaN NaN
4  MX17004  2010      3    tmax NaN   NaN   NaN NaN  32.1 NaN NaN
```

The weather data include minimum and maximum (`tmin` and `tmax` values in the element column, respectively) temperatures recorded for each day (`d1`, `d2`, ..., `d31`) of the month (month). The `element` column contains variables that need to be casted/pivoted to become new columns, and the day variables need to be melted into row values. Again, there is nothing wrong with the data in the current format. It is simply not in a shape amenable to analysis, although this kind of formatting can be helpful when presenting data in reports. Let's first melt/unpivot the day values.

```
weather_melt = pd.melt(weather,
                       id_vars=['id', 'year', 'month', 'element'],
                       var_name='day',
                       value_name='temp')
print(weather_melt.head())
```

```
        id  year  month element day  temp
0  MX17004  2010      1    tmax  d1   NaN
1  MX17004  2010      1    tmin  d1   NaN
2  MX17004  2010      2    tmax  d1   NaN
3  MX17004  2010      2    tmin  d1   NaN
4  MX17004  2010      3    tmax  d1   NaN
```

```
print(weather_melt.tail())
```

```
          id  year  month element  day  temp
677  MX17004  2010     10    tmin  d31   NaN
678  MX17004  2010     11    tmax  d31   NaN
679  MX17004  2010     11    tmin  d31   NaN
```

```
680  MX17004  2010     12    tmax  d31   NaN
681  MX17004  2010     12    tmin  d31   NaN
```

Next, we need to pivot up the variables stored in the element column. This process is referred to as casting or spreading in other statistical languages. One of the main differences between `pivot_table` and `melt` is that `melt` is a function within Pandas, whereas `pivot_table` is a method we call on a `DataFrame` object.

```
weather_tidy = weather_melt.pivot_table(
    index=['id', 'year', 'month', 'day'],
    columns='element',
    values='temp')
```

Looking at the pivoted table, we notice that each value in the `element` column is now a separate column. We can leave this table in its current state, but we can also flatten the hierarchical columns.

```
weather_tidy_flat = weather_tidy.reset_index()
print(weather_tidy_flat.head())
```

```
element       id  year  month  day  tmax  tmin
0        MX17004  2010      1   d1   NaN   NaN
1        MX17004  2010      1  d10   NaN   NaN
2        MX17004  2010      1  d11   NaN   NaN
3        MX17004  2010      1  d12   NaN   NaN
4        MX17004  2010      1  d13   NaN   NaN
```

Likewise, we can apply these methods without the intermediate dataframe:

```
weather_tidy = weather_melt.\
    pivot_table(
        index=['id', 'year', 'month', 'day'],
        columns='element',
        values='temp').\
    reset_index()

print(weather_tidy.head())
```

```
element       id  year  month  day  tmax  tmin
0        MX17004  2010      1   d1   NaN   NaN
1        MX17004  2010      1  d10   NaN   NaN
2        MX17004  2010      1  d11   NaN   NaN
3        MX17004  2010      1  d12   NaN   NaN
4        MX17004  2010      1  d13   NaN   NaN
```

6.5 Multiple Observational Units in a Table (Normalization)

One of the simplest ways of knowing whether multiple observational units are represented in a table is by looking at each of the rows, and taking note of any cells or values that are being repeated from row to row. This is very common in government education administration data, where student demographics are reported for each student for each year the student is enrolled.

Let's look again at the Billboard data we cleaned in Section 6.2.2.

```
print(billboard_long.head())
```

```
   year        artist                    track  time date.entered  \
0  2000         2 Pac  Baby Don't Cry (Keep...  4:22   2000-02-26
1  2000       2Ge+her  The Hardest Part Of ...  3:15   2000-09-02
2  2000  3 Doors Down               Kryptonite  3:53   2000-04-08
3  2000  3 Doors Down                    Loser  4:24   2000-10-21
4  2000      504 Boyz            Wobble Wobble  3:35   2000-04-15

  week  rating
0  wk1    87.0
1  wk1    91.0
2  wk1    81.0
3  wk1    76.0
4  wk1    57.0
```

Suppose we subset (Section 2.4.1) the data based on a particular track:

```
print(billboard_long[billboard_long.track == 'Loser'].head())
```

```
      year        artist  track  time date.entered week  rating
3     2000  3 Doors Down  Loser  4:24   2000-10-21  wk1    76.0
320   2000  3 Doors Down  Loser  4:24   2000-10-21  wk2    76.0
637   2000  3 Doors Down  Loser  4:24   2000-10-21  wk3    72.0
954   2000  3 Doors Down  Loser  4:24   2000-10-21  wk4    69.0
1271  2000  3 Doors Down  Loser  4:24   2000-10-21  wk5    67.0
```

We can see that this table actually holds two types of data: the track information and the weekly ranking. It would be better to store the track information in a separate table. This way, the information stored in the `year`, `artist`, `track`, and `time` columns would not be repeated in the data set. This consideration is particularly important if the data is manually entered. Repeating the same values over and over during data entry increases the risk of inconsistent data.

What we should do in this case is to place the `year`, `artist`, `track`, `time`, and `date.entered` in a new dataframe, with each unique set of values being assigned a unique ID. We can then use this unique ID in a second dataframe that represents a song, date, week number, and ranking. This entire process can be thought of as reversing the steps in concatenating and merging data described in Chapter 4.

```
billboard_songs = billboard_long[['year', 'artist', 'track', 'time']]
print(billboard_songs.shape)
```

```
(24092, 4)
```

We know there are duplicate entries in this dataframe, so we need to drop the duplicate rows.

```
billboard_songs = billboard_songs.drop_duplicates()
print(billboard_songs.shape)
```

```
(317, 4)
```

We can then assign a unique value to each row of data.

```
billboard_songs['id'] = range(len(billboard_songs))
print(billboard_songs.head(n=10))
```

```
   year           artist                    track  time  id
0  2000            2 Pac  Baby Don't Cry (Keep...  4:22   0
1  2000          2Ge+her  The Hardest Part Of ...  3:15   1
2  2000     3 Doors Down               Kryptonite  3:53   2
3  2000     3 Doors Down                    Loser  4:24   3
4  2000         504 Boyz            Wobble Wobble  3:35   4
5  2000             98^0  Give Me Just One Nig...  3:24   5
6  2000          A*Teens            Dancing Queen  3:44   6
7  2000          Aaliyah            I Don't Wanna  4:15   7
8  2000          Aaliyah                Try Again  4:03   8
9  2000  Adams, Yolanda            Open My Heart  5:30   9
```

Now that we have a separate dataframe about songs, we can use the newly created `id` column to match a song to its weekly ranking.

```
# Merge the song dataframe to the original data set
billboard_ratings = billboard_long.merge(
    billboard_songs, on=['year', 'artist', 'track', 'time'])
print(billboard_ratings.shape)
```

```
(24092, 8)
```

```
print(billboard_ratings.head())
```

```
   year artist                    track  time date.entered week  \
0  2000  2 Pac  Baby Don't Cry (Keep...  4:22   2000-02-26   wk1
1  2000  2 Pac  Baby Don't Cry (Keep...  4:22   2000-02-26   wk2
2  2000  2 Pac  Baby Don't Cry (Keep...  4:22   2000-02-26   wk3
3  2000  2 Pac  Baby Don't Cry (Keep...  4:22   2000-02-26   wk4
4  2000  2 Pac  Baby Don't Cry (Keep...  4:22   2000-02-26   wk5

   rating  id
0    87.0   0
1    82.0   0
2    72.0   0
3    77.0   0
4    87.0   0
```

Finally, we subset the columns to the ones we want in our ratings dataframe.

```
billboard_ratings = \
    billboard_ratings[['id', 'date.entered', 'week', 'rating']]
print(billboard_ratings.head())
```

```
   id date.entered week  rating
0   0   2000-02-26  wk1    87.0
1   0   2000-02-26  wk2    82.0
2   0   2000-02-26  wk3    72.0
3   0   2000-02-26  wk4    77.0
4   0   2000-02-26  wk5    87.0
```

6.6 Observational Units Across Multiple Tables

The last bit of data tidying relates to the situation in which the same type of data is spread across multiple data sets. This issue was also covered in Chapter 4, when we discussed data concatenation and merging. One reason why data might be split across multiple files would be the size of the files. By splitting up data into various parts, each part would be smaller. This may be good when we need to share data on the Internet or via email, since many services limit the size of a file that can be opened or shared. Another reason why a data set might be split into multiple parts would be to account for the data collection process. For example, a separate data set containing stock information could be created for each day.

Since merging and concatenation have already been covered, this section will focus on techniques for quickly loading multiple data sources and assembling them together.

The Unified New York City Taxi and Uber Data is a good choice to illustrate these processes. The entire data set contains data on more than 1.3 billion taxi and Uber trips from New York City, and is organized into more than 140 files. For illustration purposes, we will work with only five of these data files. When the same data is broken into multiple parts, those parts typically have a structured naming pattern associated with them.

First let's download the data. Do not worry too much about the details in the following block of code. The `raw_data_urls.txt` file contain a list of URLs where each URL is the download link to a part of the taxi data. We begin by opening and reading the file, and iterating through each line of the file (i.e., each data URL). We download only the first 5 data sets since the files are fairly large. We use some string manipulation (Chapter 8) to create the path where the data will be saved, and use the `urllib` library to download our data.

```python
import os
import urllib

# code to download the data
# download only the first 5 data sets from the list of files
with open('../data/raw_data_urls.txt', 'r') as data_urls:
    for line, url in enumerate(data_urls):
        if line == 5:
            break
        fn = url.split('/')[-1].strip()
        fp = os.path.join('..', 'data', fn)
        print(url)
        print(fp)
        urllib.request.urlretrieve(url, fp)
```

In this example, all of the raw taxi trips have the pattern `fhv_tripdata_YYYY_XX.csv`, where `YYYY` represents the year (e.g., 2015), and `XX` represents the part number. We can use the a simple pattern matching function from the glob library in Python to get a list of all the filenames that match a particular pattern.

```
import glob
# get a list of the csv files from the nyc-taxi data folder
nyc_taxi_data = glob.glob('../data/fhv_*')
print(nyc_taxi_data)
```

```
['../data/fhv_tripdata_2015-04.csv',
'../data/fhv_tripdata_2015-05.csv',
'../data/fhv_tripdata_2015-03.csv',
'../data/fhv_tripdata_2015-01.csv',
'../data/fhv_tripdata_2015-02.csv']
```

Now that we have a list of filenames we want to load, we can load each file into a dataframe. We can choose to load each file individually, as we have been doing so far.

```
taxi1 = pd.read_csv(nyc_taxi_data[0])
taxi2 = pd.read_csv(nyc_taxi_data[1])
taxi3 = pd.read_csv(nyc_taxi_data[2])
taxi4 = pd.read_csv(nyc_taxi_data[3])
taxi5 = pd.read_csv(nyc_taxi_data[4])
```

We can look at our data and see how they can be nicely stacked (concatenated) on top of each other.

```
print(taxi1.head(n=2))
print(taxi2.head(n=2))
print(taxi3.head(n=2))
print(taxi4.head(n=2))
print(taxi5.head(n=2))
```

```
  Dispatching_base_num          Pickup_date  locationID
0               B00001  2015-04-01 04:30:00         NaN
1               B00001  2015-04-01 06:00:00         NaN
  Dispatching_base_num          Pickup_date  locationID
0               B00001  2015-05-01 04:30:00         NaN
1               B00001  2015-05-01 05:00:00         NaN
  Dispatching_base_num          Pickup_date  locationID
0               B00029  2015-03-01 00:02:00       213.0
1               B00029  2015-03-01 00:03:00        51.0
  Dispatching_base_num          Pickup_date  locationID
0               B00013  2015-01-01 00:30:00         NaN
1               B00013  2015-01-01 01:22:00         NaN
  Dispatching_base_num          Pickup_date  locationID
0               B00013  2015-02-01 00:00:00         NaN
1               B00013  2015-02-01 00:01:00         NaN
```

We can concatenate them just as we did in Chapter 4.

```
# shape of each dataframe
print(taxi1.shape)
print(taxi2.shape)
print(taxi3.shape)
print(taxi4.shape)
print(taxi5.shape)
```

```
(3917789, 3)
(4296067, 3)
(3281427, 3)
(2746033, 3)
(3126401, 3)
```

```
# concatenate the dataframes together
taxi = pd.concat([taxi1, taxi2, taxi3, taxi4, taxi5])

# shape of final concatenated taxi data
print(taxi.shape)
```

```
(17367717, 3)
```

However, manually saving each dataframe will get tedious when the data is split into many parts. As an alternative approach, we can automate the process using loops and list comprehensions.

6.6.1 Load Multiple Files Using a Loop

An easier way to load multiple files is to first create an empty list, use a loop to iterate though each of the CSV files, load the CSV files into a Pandas dataframe, and finally append the dataframe to the list. The final type of data we want is a list of dataframes because the `concat` function takes a list of dataframes to concatenate.

```
# create an empty list to append to
list_taxi_df = []

# loop though each CSV filename
for csv_filename in nyc_taxi_data:
    # you can choose to print the filename for debugging
    # print(csv_filename)

    # load the CSV file into a dataframe
    df = pd.read_csv(csv_filename)

    # append the dataframe to the list that will hold the dataframes
    list_taxi_df.append(df)

# print the length of the dataframe
print(len(list_taxi_df))
```

```
5
```

```
# type of the first element
print(type(list_taxi_df[0]))
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
# look at the head of the first dataframe
print(list_taxi_df[0].head())
```

```
  Dispatching_base_num          Pickup_date  locationID
0               B00001  2015-04-01 04:30:00         NaN
1               B00001  2015-04-01 06:00:00         NaN
2               B00001  2015-04-01 06:00:00         NaN
3               B00001  2015-04-01 06:00:00         NaN
4               B00001  2015-04-01 06:15:00         NaN
```

Now that we have a list of dataframes, we can concatenate them.

```
taxi_loop_concat = pd.concat(list_taxi_df)
print(taxi_loop_concat.shape)
```

```
(17367717, 3)
```

```
# Did we get the same results as the manual load and concatenation?
print(taxi.equals(taxi_loop_concat))
```

```
True
```

6.6.2 Load Multiple Files Using a List Comprehension

Python has an idiom for looping though something and adding it to a list, called a list comprehension. The loop given previously, which is shown here again without the comments, can be written in a list comprehension (Appendix N).

```
# the loop code without comments
list_taxi_df = []
for csv_filename in nyc_taxi_data:
    df = pd.read_csv(csv_filename)
    list_taxi_df.append(df)

# same code in a list comprehension
list_taxi_df_comp = [pd.read_csv(data) for data in nyc_taxi_data]
```

The result from our list comprehension is a list, just as the earlier loop example.

```
print(type(list_taxi_df_comp))
```

```
<class 'list'>
```

Finally, we can concatenate the results just as we did earlier.

```
taxi_loop_concat_comp = pd.concat(list_taxi_df_comp)

# are the concatenated dataframes the same?
print(taxi_loop_concat_comp.equals(taxi_loop_concat))
```

```
True
```

6.7 Conclusion

This chapter explored how we can reshape data into a format that is conducive tp data analysis, visualization, and collection. We applied the concepts in Hadley Wickham's *Tidy Data* paper to show the various functions and methods to reshape our data. This is an important skill because some functions need data to be organized into a certain shape, tidy or not, to work. Knowing how to reshape your data is an important skill for both the data scientist and the analyst.

This page intentionally left blank

E

F

G

H

N

O

P

Q

R

T

U

Y

Z