Pandas for Everyone
Python Data Analysis

Daniel Y. Y. Chen
Pandas for Everyone
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.

Visit [informit.com/awdataseriess](http://informit.com/awdataseriess) for a complete list of available publications.

Make sure to connect with us! [informit.com/socialconnect](http://informit.com/socialconnect)
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
To my family: Mom, Dad, Eric, and Julia
This page intentionally left blank
Contents

Foreword xix
Preface xxii
Acknowledgments xxvii
About the Author xxxi

I Introduction 1

1 Pandas DataFrame Basics 3
  1.1 Introduction 3
  1.2 Loading Your First Data Set 4
  1.3 Looking at Columns, Rows, and Cells 7
    1.3.1 Subsetting Columns 7
    1.3.2 Subsetting Rows 8
    1.3.3 Mixing It Up 12
  1.4 Grouped and Aggregated Calculations 18
    1.4.1 Grouped Means 19
    1.4.2 Grouped Frequency Counts 23
  1.5 Basic Plot 23
  1.6 Conclusion 24

2 Pandas Data Structures 25
  2.1 Introduction 25
  2.2 Creating Your Own Data 26
    2.2.1 Creating a Series 26
    2.2.2 Creating a DataFrame 27
  2.3 The Series 28
    2.3.1 The Series Is ndarray-like 30
    2.3.2 Boolean Subsetting: Series 30
    2.3.3 Operations Are Automatically Aligned and Vectorized (Broadcasting) 33
### Contents

2.4 The DataFrame 36
  2.4.1 Boolean Subsetting: DataFrames 36
  2.4.2 Operations Are Automatically Aligned and Vectorized (Broadcasting) 37

2.5 Making Changes to Series and DataFrames 38
  2.5.1 Add Additional Columns 38
  2.5.2 Directly Change a Column 39
  2.5.3 Dropping Values 43

2.6 Exporting and Importing Data 43
  2.6.1 pickle 43
  2.6.2 CSV 45
  2.6.3 Excel 46
  2.6.4 Feather Format to Interface With R 47
  2.6.5 Other Data Output Types 47

2.7 Conclusion 47

3 Introduction to Plotting 49
  3.1 Introduction 49

3.2 Matplotlib 51

3.3 Statistical Graphics Using matplotlib 56
  3.3.1 Univariate 57
  3.3.2 Bivariate 58
  3.3.3 Multivariate Data 59

3.4 Seaborn 61
  3.4.1 Univariate 62
  3.4.2 Bivariate Data 65
  3.4.3 Multivariate Data 73

3.5 Pandas Objects 83
  3.5.1 Histograms 84
  3.5.2 Density Plot 85
  3.5.3 Scatterplot 85
  3.5.4 Hexbin Plot 86
  3.5.5 Boxplot 86

3.6 Seaborn Themes and Styles 86

3.7 Conclusion 90
II Data Manipulation 91

4 Data Assembly 93
  4.1 Introduction 93
  4.2 Tidy Data 93
    4.2.1 Combining Data Sets 94
  4.3 Concatenation 94
    4.3.1 Adding Rows 94
    4.3.2 Adding Columns 98
    4.3.3 Concatenation With Different Indices 99
  4.4 Merging Multiple Data Sets 102
    4.4.1 One-to-One Merge 104
    4.4.2 Many-to-One Merge 105
    4.4.3 Many-to-Many Merge 105
  4.5 Conclusion 107

5 Missing Data 109
  5.1 Introduction 109
  5.2 What Is a NaN Value? 109
  5.3 Where Do Missing Values Come From? 111
    5.3.1 Load Data 111
    5.3.2 Merged Data 112
    5.3.3 User Input Values 114
    5.3.4 Re-indexing 114
  5.4 Working With Missing Data 116
    5.4.1 Find and Count missing Data 116
    5.4.2 Cleaning Missing Data 118
    5.4.3 Calculations With Missing Data 120
  5.5 Conclusion 121

6 Tidy Data 123
  6.1 Introduction 123
  6.2 Columns Contain Values, Not Variables 124
6.2.1 Keep One Column Fixed 124
6.2.2 Keep Multiple Columns Fixed 126
6.3 Columns Contain Multiple Variables 128
6.3.1 Split and Add Columns Individually (Simple Method) 129
6.3.2 Split and Combine in a Single Step (Simple Method) 131
6.3.3 Split and Combine in a Single Step (More Complicated Method) 132
6.4 Variables in Both Rows and Columns 133
6.5 Multiple Observational Units in a Table (Normalization) 134
6.6 Observational Units Across Multiple Tables 137
6.6.1 Load Multiple Files Using a Loop 139
6.6.2 Load Multiple Files Using a List Comprehension 140
6.7 Conclusion 141

III Data Munging 143

7 Data Types 145
7.1 Introduction 145
7.2 Data Types 145
7.3 Converting Types 146
7.3.1 Converting to String Objects 146
7.3.2 Converting to Numeric Values 147
7.4 Categorical Data 152
7.4.1 Convert to Category 152
7.4.2 Manipulating Categorical Data 153
7.5 Conclusion 153
11.6 Date Calculations and Timedeltas 220
11.7 Datetime Methods 221
11.8 Getting Stock Data 224
11.9 Subsetting Data Based on Dates 225
  11.9.1 The DatetimeIndex Object 225
  11.9.2 The TimedeltaIndex Object 226
11.10 Date Ranges 227
  11.10.1 Frequencies 228
  11.10.2 Offsets 229
11.11 Shifting Values 230
11.12 Resampling 237
11.13 Time Zones 238
11.14 Conclusion 240

IV Data Modeling 241

12 Linear Models 243
  12.1 Introduction 243
  12.2 Simple Linear Regression 243
    12.2.1 Using statsmodels 243
    12.2.2 Using sklearn 245
  12.3 Multiple Regression 247
    12.3.1 Using statsmodels 247
    12.3.2 Using statsmodels With Categorical Variables 248
    12.3.3 Using sklearn 249
    12.3.4 Using sklearn With Categorical Variables 250
  12.4 Keeping Index Labels From sklearn 251
  12.5 Conclusion 252

13 Generalized Linear Models 253
  13.1 Introduction 253
  13.2 Logistic Regression 253
    13.2.1 Using Statsmodels 255
    13.2.2 Using Sklearn 256
13.3 Poisson Regression 257
  13.3.1 Using Statsmodels 258
  13.3.2 Negative Binomial Regression for Overdispersion 259
13.4 More Generalized Linear Models 260
13.5 Survival Analysis 260
  13.5.1 Testing the Cox Model Assumptions 263
13.6 Conclusion 264

14 Model Diagnostics 265
  14.1 Introduction 265
  14.2 Residuals 265
    14.2.1 Q-Q Plots 268
  14.3 Comparing Multiple Models 270
    14.3.1 Working With Linear Models 270
    14.3.2 Working With GLM Models 273
  14.4 k-Fold Cross-Validation 275
14.5 Conclusion 278

15 Regularization 279
  15.1 Introduction 279
  15.2 Why Regularize? 279
  15.3 LASSO Regression 281
  15.4 Ridge Regression 283
  15.5 Elastic Net 285
  15.6 Cross-Validation 287
15.7 Conclusion 289

16 Clustering 291
  16.1 Introduction 291
  16.2 k-Means 291
    16.2.1 Dimension Reduction With PCA 294
  16.3 Hierarchical Clustering 297
    16.3.1 Complete Clustering 298
    16.3.2 Single Clustering 298
    16.3.3 Average Clustering 299
16.3.4 Centroid Clustering 299
16.3.5 Manually Setting the Threshold 299

16.4 Conclusion 301

V Conclusion 303

17 Life Outside of Pandas 305
17.1 The (Scientific) Computing Stack 305
17.2 Performance 306
17.2.1 Timing Your Code 306
17.2.2 Profiling Your Code 307
17.3 Going Bigger and Faster 307

18 Toward a Self-Directed Learner 309
18.1 It's Dangerous to Go Alone! 309
18.2 Local Meetups 309
18.3 Conferences 309
18.4 The Internet 310
18.5 Podcasts 310
18.6 Conclusion 311

VI Appendixes 313

A Installation 315
A.1 Installing Anaconda 315
A.1.1 Windows 315
A.1.2 Mac 316
A.1.3 Linux 316
A.2 Uninstall Anaconda 316

B Command Line 317
B.1 Installation 317
B.1.1 Windows 317
B.1.2 Mac 317
B.1.3 Linux 318
B.2 Basics 318
Contents

Q  Multiple Assignment  351

R  numpy ndarray  353

S  Classes  355

T  Odo: The Shapeshifter  357

Index  359
This page intentionally left blank
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
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 a 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 `sklearn` 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-learn regex 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.
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 Ahmadi 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 Ziemen, 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
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
6.1 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:

- function and method calls
- subsetting data
- loops
- list comprehension

This chapter:

- Reshaping data
  - unpivot/melt/gather
  - pivot/cast/spread

---

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.

```python
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.

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

<table>
<thead>
<tr>
<th>religion</th>
<th>&lt;$10k</th>
<th>$10-20k</th>
<th>$20-30k</th>
<th>$30-40k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agnostic</td>
<td>27</td>
<td>34</td>
<td>60</td>
<td>81</td>
</tr>
<tr>
<td>Atheist</td>
<td>12</td>
<td>27</td>
<td>37</td>
<td>52</td>
</tr>
<tr>
<td>Buddhist</td>
<td>27</td>
<td>21</td>
<td>30</td>
<td>34</td>
</tr>
<tr>
<td>Catholic</td>
<td>418</td>
<td>617</td>
<td>732</td>
<td>670</td>
</tr>
<tr>
<td>Don't know/refused</td>
<td>15</td>
<td>14</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>Evangelical Prot</td>
<td>575</td>
<td>869</td>
<td>1064</td>
<td>982</td>
</tr>
<tr>
<td>Hindu</td>
<td>1</td>
<td>9</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Historically Black Prot</td>
<td>228</td>
<td>244</td>
<td>236</td>
<td>238</td>
</tr>
<tr>
<td>Jehovah's Witness</td>
<td>20</td>
<td>27</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Jewish</td>
<td>19</td>
<td>19</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Mainline Prot</td>
<td>289</td>
<td>495</td>
<td>619</td>
<td>655</td>
</tr>
<tr>
<td>Mormon</td>
<td>29</td>
<td>40</td>
<td>48</td>
<td>51</td>
</tr>
<tr>
<td>Muslim</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>
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())
```

```
<table>
<thead>
<tr>
<th>religion</th>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agnostic</td>
<td>&lt;$10k</td>
<td>27</td>
</tr>
<tr>
<td>Atheist</td>
<td>&lt;$10k</td>
<td>12</td>
</tr>
<tr>
<td>Buddhist</td>
<td>&lt;$10k</td>
<td>27</td>
</tr>
<tr>
<td>Buddhist</td>
<td>&lt;$40-50k</td>
<td>27</td>
</tr>
<tr>
<td>Buddhist</td>
<td>&lt;$70-100k</td>
<td>27</td>
</tr>
</tbody>
</table>
```

6.2 Columns Contain Values, Not Variables
Chapter 6  Tidy Data

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

print(pew_long.tail())

<table>
<thead>
<tr>
<th>religion</th>
<th>income</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orthodox</td>
<td>Don't know/refused</td>
<td>73</td>
</tr>
<tr>
<td>Other Christian</td>
<td>Don't know/refused</td>
<td>18</td>
</tr>
<tr>
<td>Other Faiths</td>
<td>Don't know/refused</td>
<td>71</td>
</tr>
<tr>
<td>Other World Religions</td>
<td>Don't know/refused</td>
<td>8</td>
</tr>
<tr>
<td>Unaffiliated</td>
<td>Don't know/refused</td>
<td>597</td>
</tr>
</tbody>
</table>

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())

<table>
<thead>
<tr>
<th>religion</th>
<th>income</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agnostic</td>
<td>&lt;$10k</td>
<td>27</td>
</tr>
<tr>
<td>Atheist</td>
<td>&lt;$10k</td>
<td>12</td>
</tr>
<tr>
<td>Buddhist</td>
<td>&lt;$10k</td>
<td>27</td>
</tr>
<tr>
<td>Catholic</td>
<td>&lt;$10k</td>
<td>418</td>
</tr>
<tr>
<td>Don't know/refused</td>
<td>&lt;$10k</td>
<td>15</td>
</tr>
</tbody>
</table>

print(pew_long.tail())

<table>
<thead>
<tr>
<th>religion</th>
<th>income</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orthodox</td>
<td>Don't know/refused</td>
<td>73</td>
</tr>
<tr>
<td>Other Christian</td>
<td>Don't know/refused</td>
<td>18</td>
</tr>
<tr>
<td>Other Faiths</td>
<td>Don't know/refused</td>
<td>71</td>
</tr>
<tr>
<td>Other World Religions</td>
<td>Don't know/refused</td>
<td>8</td>
</tr>
<tr>
<td>Unaffiliated</td>
<td>Don't know/refused</td>
<td>597</td>
</tr>
</tbody>
</table>

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])

<table>
<thead>
<tr>
<th>year</th>
<th>artist</th>
<th>track</th>
<th>time</th>
<th>date.entered</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>2 Pac</td>
<td>Baby</td>
<td>4:22</td>
<td>2000-02-26</td>
</tr>
<tr>
<td>2000</td>
<td>2 Ge+her</td>
<td>The Hardest Part Of... 3:15</td>
<td>2000-09-02</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>3 Doors Down</td>
<td>Kryptonite</td>
<td>3:53</td>
<td>2000-04-08</td>
</tr>
<tr>
<td>2000</td>
<td>3 Doors Down</td>
<td>Loser</td>
<td>4:24</td>
<td>2000-10-21</td>
</tr>
</tbody>
</table>
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.

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

You can see 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.

```python
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
```

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

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

```python
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
```

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

```
   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.

```python
import pandas as pd

# Read the data set
ebola = pd.read_csv('../data/country_timeseries.csv')
print(ebola.columns)

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

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.

```python
# Unpivot the data
ebola_long = pd.melt(ebola, id_vars=['Date', 'Day'])
print(ebola_long.head())

# Print the last few rows
print(ebola_long.tail())
```
6.3 Columns Contain Multiple Variables

<table>
<thead>
<tr>
<th>Year</th>
<th>Date</th>
<th>Cases</th>
<th>Deaths_Mali</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1949</td>
<td>3/25/2014</td>
<td>3</td>
<td>Deaths_Mali</td>
<td>NaN</td>
</tr>
<tr>
<td>1950</td>
<td>3/24/2014</td>
<td>2</td>
<td>Deaths_Mali</td>
<td>NaN</td>
</tr>
<tr>
<td>1951</td>
<td>3/22/2014</td>
<td>0</td>
<td>Deaths_Mali</td>
<td>NaN</td>
</tr>
</tbody>
</table>

### 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 methods 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.

```python
# 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, but the visual cue is that the results are surrounded by square brackets.

```python
# 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](https://docs.python.org/3.6/library/stdtypes.html#str.split)
Chapter 6  Tidy Data

# 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
6.3 Columns Contain Multiple Variables

print(ebola_long.head())

<table>
<thead>
<tr>
<th>Date</th>
<th>Day</th>
<th>variable</th>
<th>value</th>
<th>status</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1/5</td>
<td>Cases_Guinea</td>
<td>2776.0</td>
<td>Cases</td>
<td>Guinea</td>
</tr>
<tr>
<td>1</td>
<td>1/4</td>
<td>Cases_Guinea</td>
<td>2775.0</td>
<td>Cases</td>
<td>Guinea</td>
</tr>
<tr>
<td>2</td>
<td>1/3</td>
<td>Cases_Guinea</td>
<td>2769.0</td>
<td>Cases</td>
<td>Guinea</td>
</tr>
<tr>
<td>3</td>
<td>1/2</td>
<td>Cases_Guinea</td>
<td>2769.0</td>
<td>Cases</td>
<td>Guinea</td>
</tr>
<tr>
<td>4</td>
<td>12/31</td>
<td>Cases_Guinea</td>
<td>2730.0</td>
<td>Cases</td>
<td>Guinea</td>
</tr>
</tbody>
</table>

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

print(ebola_parsed.head())

<table>
<thead>
<tr>
<th>Date</th>
<th>Day</th>
<th>variable</th>
<th>value</th>
<th>status</th>
<th>country</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1/5</td>
<td>Cases_Guinea</td>
<td>2776.0</td>
<td>Cases</td>
<td>Guinea</td>
<td>Cases</td>
</tr>
<tr>
<td>1</td>
<td>1/4</td>
<td>Cases_Guinea</td>
<td>2775.0</td>
<td>Cases</td>
<td>Guinea</td>
<td>Cases</td>
</tr>
<tr>
<td>2</td>
<td>1/3</td>
<td>Cases_Guinea</td>
<td>2769.0</td>
<td>Cases</td>
<td>Guinea</td>
<td>Cases</td>
</tr>
<tr>
<td>3</td>
<td>1/2</td>
<td>Cases_Guinea</td>
<td>2769.0</td>
<td>Cases</td>
<td>Guinea</td>
<td>Cases</td>
</tr>
<tr>
<td>4</td>
<td>12/31</td>
<td>Cases_Guinea</td>
<td>2730.0</td>
<td>Cases</td>
<td>Guinea</td>
<td>Cases</td>
</tr>
</tbody>
</table>

print(ebola_parsed.tail())

<table>
<thead>
<tr>
<th>Date</th>
<th>Day</th>
<th>variable</th>
<th>value</th>
<th>status</th>
<th>country</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1947</td>
<td>3/27</td>
<td>Deaths_Mali</td>
<td>NaN</td>
<td>Deaths</td>
<td>Mali</td>
<td>Deaths</td>
</tr>
<tr>
<td>1948</td>
<td>3/26</td>
<td>Deaths_Mali</td>
<td>NaN</td>
<td>Deaths</td>
<td>Mali</td>
<td>Deaths</td>
</tr>
<tr>
<td>1949</td>
<td>3/25</td>
<td>Deaths_Mali</td>
<td>NaN</td>
<td>Deaths</td>
<td>Mali</td>
<td>Deaths</td>
</tr>
<tr>
<td>1950</td>
<td>3/24</td>
<td>Deaths_Mali</td>
<td>NaN</td>
<td>Deaths</td>
<td>Mali</td>
<td>Deaths</td>
</tr>
<tr>
<td>1951</td>
<td>3/22</td>
<td>Deaths_Mali</td>
<td>NaN</td>
<td>Deaths</td>
<td>Mali</td>
<td>Deaths</td>
</tr>
</tbody>
</table>

print(ebola_parsed.tail())

<table>
<thead>
<tr>
<th>Date</th>
<th>Day</th>
<th>variable</th>
<th>value</th>
<th>status</th>
<th>country</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1947</td>
<td>Mali</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1948</td>
<td>Mali</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1949</td>
<td>Mali</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1950</td>
<td>Mali</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1951</td>
<td>Mali</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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,

```python
costants = ['pi', 'e']
values = ['3.14', '2.718']
```

we can zip the values together:

```python
# 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. 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).

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

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

The weather data include minimum and maximum \( (t_{\text{min}} \text{ and } t_{\text{max}}) \) values in the element column, respectively) temperatures recorded for each day \( (d_1, d_2, \ldots, d_{31}) \) 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.

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

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

```plaintext
6.4 Variables in Both Rows and Columns

id year month element day temp
0 MX17004 2010 1 tmax NaN NaN NaN NaN NaN NaN
1 MX17004 2010 1 tmin 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 NaN
```

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.

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

The weather data include minimum and maximum \( (t_{\text{min}} \text{ and } t_{\text{max}}) \) values in the element column, respectively) temperatures recorded for each day \( (d_1, d_2, \ldots, d_{31}) \) 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.

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

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

```plaintext
6.4 Variables in Both Rows and Columns

id year month element day temp
0 MX17004 2010 1 tmax NaN NaN NaN NaN NaN NaN
1 MX17004 2010 1 tmin 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 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 \texttt{pivot\_table} and \texttt{melt} is that \texttt{melt} is a function within Pandas, whereas \texttt{pivot\_table} is a method we call on a DataFrame object.

\begin{verbatim}
weather\_tidy = weather\_melt.pivot\_table(
    index=['id', 'year', 'month', 'day'],
    columns='element',
    values='temp')
\end{verbatim}

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

\begin{verbatim}
weather\_tidy\_flat = weather\_tidy.reset\_index()
print(weather\_tidy\_flat.head())
\end{verbatim}

\begin{verbatim}
<table>
<thead>
<tr>
<th>element</th>
<th>id</th>
<th>year</th>
<th>month</th>
<th>day</th>
<th>tmax</th>
<th>tmin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MX17004</td>
<td>2010</td>
<td>1</td>
<td>d1</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>MX17004</td>
<td>2010</td>
<td>1</td>
<td>d10</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>MX17004</td>
<td>2010</td>
<td>1</td>
<td>d11</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>MX17004</td>
<td>2010</td>
<td>1</td>
<td>d12</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>4</td>
<td>MX17004</td>
<td>2010</td>
<td>1</td>
<td>d13</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
\end{verbatim}

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

\begin{verbatim}
weather\_tidy = weather\_melt.
    pivot\_table(
        index=['id', 'year', 'month', 'day'],
        columns='element',
        values='temp').
    reset\_index()
print(weather\_tidy.head())
\end{verbatim}

\begin{verbatim}
<table>
<thead>
<tr>
<th>element</th>
<th>id</th>
<th>year</th>
<th>month</th>
<th>day</th>
<th>tmax</th>
<th>tmin</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MX17004</td>
<td>2010</td>
<td>1</td>
<td>d1</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>MX17004</td>
<td>2010</td>
<td>1</td>
<td>d10</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>MX17004</td>
<td>2010</td>
<td>1</td>
<td>d11</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>3</td>
<td>MX17004</td>
<td>2010</td>
<td>1</td>
<td>d12</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>4</td>
<td>MX17004</td>
<td>2010</td>
<td>1</td>
<td>d13</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
\end{verbatim}

\section{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.

```python
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

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:

```python
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.

```python
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.
Chapter 6  Tidy Data

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

(317, 4)

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

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

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

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.

```python
# 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)
print(billboard_ratings.head())
```

```python
<table>
<thead>
<tr>
<th>year</th>
<th>artist</th>
<th>track time</th>
<th>date.entered</th>
<th>week</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>2 Pac   Baby Don't Cry (Keep... 4:22</td>
<td>2000-02-26</td>
<td>wk1</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>2 Pac   Baby Don't Cry (Keep... 4:22</td>
<td>2000-02-26</td>
<td>wk2</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>2 Pac   Baby Don't Cry (Keep... 4:22</td>
<td>2000-02-26</td>
<td>wk3</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>2 Pac   Baby Don't Cry (Keep... 4:22</td>
<td>2000-02-26</td>
<td>wk4</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>2 Pac   Baby Don't Cry (Keep... 4:22</td>
<td>2000-02-26</td>
<td>wk5</td>
<td></td>
</tr>
</tbody>
</table>

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

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

```python
<table>
<thead>
<tr>
<th>rating</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>87.0</td>
<td>0</td>
</tr>
<tr>
<td>82.0</td>
<td>0</td>
</tr>
<tr>
<td>72.0</td>
<td>0</td>
</tr>
<tr>
<td>77.0</td>
<td>0</td>
</tr>
<tr>
<td>87.0</td>
<td>0</td>
</tr>
</tbody>
</table>
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 contains 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)
```

### Table

<table>
<thead>
<tr>
<th>id</th>
<th>date.entered</th>
<th>week</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2000-02-26</td>
<td>wk1</td>
<td>87.0</td>
</tr>
<tr>
<td>1</td>
<td>2000-02-26</td>
<td>wk2</td>
<td>82.0</td>
</tr>
<tr>
<td>2</td>
<td>2000-02-26</td>
<td>wk3</td>
<td>72.0</td>
</tr>
<tr>
<td>3</td>
<td>2000-02-26</td>
<td>wk4</td>
<td>77.0</td>
</tr>
<tr>
<td>4</td>
<td>2000-02-26</td>
<td>wk5</td>
<td>87.0</td>
</tr>
</tbody>
</table>
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 simple pattern matching function from the glob library in Python to get a list of all filenames that match a particular pattern.

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

```python
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.

```python
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.

```python
# 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 through 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.

```python
# 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))
```
# 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())

<table>
<thead>
<tr>
<th>Dispatching_base_num</th>
<th>Pickup_date</th>
<th>locationID</th>
</tr>
</thead>
<tbody>
<tr>
<td>B00001</td>
<td>2015-04-01 04:30:00</td>
<td>NaN</td>
</tr>
<tr>
<td>B00001</td>
<td>2015-04-01 06:00:00</td>
<td>NaN</td>
</tr>
<tr>
<td>B00001</td>
<td>2015-04-01 06:00:00</td>
<td>NaN</td>
</tr>
<tr>
<td>B00001</td>
<td>2015-04-01 06:15:00</td>
<td>NaN</td>
</tr>
</tbody>
</table>

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

```python
# 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)
```

```python
# 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.

```python
print(type(list_taxi_df_comp))
<class 'list'>
```

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

```python
# 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)
```

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

```python
# Are the concatenated dataframes the same?
print(taxi_loop_concat_comp.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).

```python
# 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)
```

```python
# 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.

```python
print(type(list_taxi_df_comp))
<class 'list'>
```

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

```python
# 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)
```

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

```python
# 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 to 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
Asterisk (*), unpacking containers, 132

**astype** method
- converting column to categorical type, 152–153
- converting to numeric values, 147–149
- converting values to strings, 146

Attributes
- class, 355
- **Series**, 29

Average cluster algorithm, in hierarchical clustering, 299–300

Axes, plotting, 55–56

---

**B**

Bar plots, 70, 72

Bash shell, 317–318

BIC (Bayesian information criteria), 272, 274–275

Binary
- feather format for saving, 47
- logistic regression for binary response variable, 253
- serialize and save data in binary format, 43–45

Bivariate statistics
- in **matplotlib**, 58–59
- in **seaborn**, 65–73

Booleans (bool)
- subsetting **DataFrame**, 36–37
- subsetting **Series**, 30–33

Boxplots
- for bivariate statistics, 58–59, 70
- creating, 85–86, 88

Broadcasting, Pandas support for, 37–38

---

**C**

**C printf** style formatting, 163

Calculations
- **datetime**, 220–221
- involving multiple variables, 203–204
- with missing data (values), 120–121

of multiple functions simultaneously, 195

** timing execution of, 307**

**CAS** (computer algebra systems), 305

**category**
- converting column to, 152–153
- manipulating categorical data, 153
- overview of, 152
- representing categorical variables, 146
- sklearn library used with categorical variables, 250–251
- **statsmodels** library used with categorical variables, 248–249

Centroid cluster algorithm, in hierarchical clustering, 299–300

Characters
- formatting character strings, 162
- getting first character of string, 156
- getting last character of string, 157–158
- slicing multiple letters of string, 156
- strings as series of, 155

Classes, 355–356

Clustering
- average cluster algorithm, 299–300
- centroid cluster algorithm, 299–300
- complete cluster algorithm, 298
- dimension reduction using PCA, 294–297
- hierarchical clustering, 297–298
- **k**-means, 291–294
- manually setting threshold for, 299, 301
- overview of, 291
- single cluster algorithm, 298–299
- summary/conclusion, 301

Code
- profiling, 307
- reuse, 345
- timing execution of, 306–307

Colon (:), use in slicing syntax, 13, 339–340

Colors, multivariate statistics in **seaborn**, 74–77

Columns
- adding, 38–39
- **apply** column-wise operations, 178–180
concatenation generally, 98–99
concatenation with different indices, 101–102
converting to category, 152–153
directly changing, 39–42
dropping values, 43
rows and columns both containing variables, 133–134
slicing, 15–17
subsetting by index position break, 8
subsetting by name, 7–8
subsetting by range, 14–15
subsetting generally, 17–18
subsetting using slicing syntax, 13–14
Columns, with multiple variables
overview of, 128–129
split and add individually, 129–131
split and combine in single step, 131–133
Columns, with values not variables
keeping multiple columns fixed, 126–127
keeping one column fixed, 124–126
overview of, 124
Comma-separated values. See CSV
(command-separated values)
Command line
basic commands, 318
Linux, 318
Mac, 317–318
overview of, 317
Windows, 317
compile, pattern compilation, 169
Complete cluster algorithm, in hierarchical clustering, 298
Comprehensions
function comprehension, 343
list comprehension, 140
overview of, 341–342
Computer algebra systems (CAS), 305
Concatenation (concat)
adding columns, 98–99
adding rows, 94–97
with different indices, 99–102
ignore_index parameter after, 98
loading multiple files, 140
observational units across multiple tables, 137–139
overview of, 94
split and combine in single step, 131–133
concurrent.features, 307
conda
creating environments, 327
managing packages, 329–330
Conferences, resources for self-directed learners, 309–310
Confidence interval, in linear regression example, 245
Containers
join method and, 160
looping over contents, 341–342
types of, 155
unpacking, 132
Conversion, of data types
to category, 152–153
to datetime, 214–216
to numeric, 147–148
odo library and, 357
to string, 146–147
Count (bar) plot, for univariate statistics, 65
Counting
groupby count, 209–211
missing data (values), 116–117
poisson regression and, 257
Covariates
adding to linear models, 270
multiple linear regression with three covariates, 266–268
Cox proportional hazards model
survival analysis, 261–263
testing assumptions, 263–264
CoxPHFitter class, lifelines library, 261, 263–264
cProfile, profiling code, 307
create (environments), 327
Cross-validation
model diagnostics, 275–278
regularization techniques, 287–289
cross_val_scores, 277
CSV (comma-separated values)
for data storage, 45–46
importing CSV files, 46
loading CSV file into DataFrame, 357
loading multiple files using loop, 139–140
Cumulative sum (cumsum), 210–211
cython, performance-related library, 306

D
Dask library, 307
Data assembly
adding columns, 98–99
adding rows, 94–97
combining data sets, 94
concatenation, 94
concatenation with different indices, 99–102
ignore_index parameter after concatenation, 98
many-to-many merges, 105–107
many-to-one merges, 105
merging multiple data sets, 102–104
one-to-one merges, 104
overview of, 93
summary/conclusion, 107
tidy data, 93–94
Data models
diagnostics. See Model diagnostics
generalized linear. See GLM (generalized linear models)
linear. See Linear models
Data sets
cleaning data, 354
combining, 94
equality tests for missing data, 110
exporting/importing data. See Exporting/importing data
going bigger and faster, 307
Indemics (Interactive Epidemic Simulation), 208
lists for data storage, 333
loading, 4–6
many-to-many merges, 105–107
many-to-one merges, 105
merging, 102–104
one-to-one merges, 104
tidy data, 93–94
Data structures
adding columns, 38–39
creating, 26–28
CSV (comma-separated values), 45–46
DataFrame alignment and vectorization, 37–38
DataFrame boolean subsetting, 36–37
DataFrame generally, 36
directly changing columns, 39–42
dropping values, 43
Excel and, 46–47
exporting/importing data, 43
feather format, 47
making changes to, 38
overview of, 25
pickle data, 43–45
Series alignment and vectorization, 33–36
Series boolean subsetting, 30–33
Series generally, 28–29
Series methods, 31
Series similarity with ndarray, 30
summary/conclusion, 47–48
Data types (dtype)
category dtype, 152
converting generally, 357
converting to category, 152–153
converting to datetime, 214–216
converting to numeric, 147–152
converting to string, 146–147
getting list of types stored in column, 152–153
manipulating categorical data, 153
to_numeric downcast, 151–152
to_numeric function, 148–151
overview of, 145
Series attributes, 29
specifying from numpy library, 146–147
summary/conclusion, 153
viewing list of, 145–146
Databases, odo library support, 357
DataCamp site, resources for self-directed learners, 310
DataFrame
adding columns, 38–39
aggregation, 195–196
alignment and vectorization, 37–38
apply function(s), 174–176
basic plots, 23–24
boolean subsetting, 36–37
as class, 355–356
concatenation, 97
creating, 27–28
defined, 3
directly changing columns, 39–42
exporting, 47–48
grouped and aggregated calculations, 18–19
grouped frequency counts, 23
grouped means, 19–22
histogram, 84
loading first data set, 4–6
methods, 37
ndarray save method, 43
odo library support, 357
overview of, 3–4, 36
slicing columns, 15–17
subsetting columns by index position
break, 8
subsetting columns by name, 7–8
subsetting columns by range, 14–15
subsetting columns using slicing syntax, 13–14
subsetting rows and columns, 17–18
subsetting rows by index label, 8–11
subsetting rows by ix attribute, 12
subsetting rows by row number, 11–12
summary/conclusion, 24
type function for checking, 5
writing CSV files (to_csv method), 45–46
date_range function, 227–228
datetime
adding columns to data structures, 38–39
calculations, 220–221
converting to, 214–216
directly changing columns, 41–42
extracting date components (year, month, day), 217–220
frequencies, 228–229
getting stock-related data, 224–225
loading date related data, 217
methods, 221–224
object, 213–214
offsets, 229–230
overview of, 213
ranges, 227–228
resampling, 237–238
shifting values, 230–237
subsetting data based on dates, 225–227
summary/conclusion, 240
time zones, 238–239
DatetimeIndex, 225–226, 228
Day, extracting date components from
datetime object, 217–220
Daylight savings, 238
def keyword, use with functions, 345–346
Density plots
2D density plot, 68–70
plot.kde function, 85
for univariate statistics, 63–64
Diagnostics. See Model diagnostics
Dictionaries (dict)
creating DataFrame, 27–28
overview of, 337–338
passing method to, 195–196
Directories, working, 325–326
distplot, creating histograms, 62–63
dmatrices function, patsy library, 276–279
Docstrings (docstring), function documentation, 172, 345
downcast parameter, to_numeric function, 151–152
dropna parameter
  counting missing values, 116–117
  dropping missing values, 119–120
Dropping (drop)
  data structure values, 43
  missing data (values), 119–120
dtype. See Data types (dtype)

E

EAFP (easier to ask for forgiveness than for permissions), 203
Elastic net, regularization technique, 285–287
Environments
  creating, 327–328
  deleting, 328
Equality tests, for missing data, 110
errors parameter, numeric, 149
Excel
  DataFrame and, 47
  Series and, 46
Exporting/importing data
  CSV (comma-separated values), 45–46
  Excel, 46–47
  feather format, 47
  overview of, 43
  pickle data, 43–45

F

f-strings (formatted literal strings), 163–164
Facets, plotting, 78–83
Feather format, interface with R language, 47
Files
  loading multiple using list comprehension, 140
  loading multiple using loop, 139–140
  odo library support, 357
  working directories and, 325
fillna method, 118–119
Filter (filter), groupby operations, 201–202
Find
  missing data (values), 116–117
  patterns, 168
findall, patterns, 168
float/float64, 146–148
Folders
  project organization, 319
  working directories and, 325
for 1oop. See Loops (for loop)
format method, 162
Formats/formatting
  date formats, 216
  odo library for conversion of data formats, 357
  serialize and save data in binary format, 43–45
  strings (string), 161–164
Formatted literal strings (f-strings), 163–164
formula API, in statsmodels library, 243–244
freq parameter, 228
Frequency
  datetime, 228–229
  grouped frequency counts, 23
  offsets, 229–230
  resampling converting between, 237–238
Functions
  across rows or columns of data, 172
  aggregation, 192–193
  apply over DataFrame, 174–176
  apply over Series, 173–174
  arbitrary parameters, 347–348
  calculating multiple simultaneously, 195
  comprehensions and, 343
  creating/using, 171–172
  custom, 193–195
  default parameters, 347
aggregation functions, 192–195
applying functions in and aggregate
methods, 195–197
built-in aggregation methods, 191–192
calculations generally, 18–19
calculations involving multiple variables, 203–204
calculations of means, 19–22
compared with SQL, 189
filtering, 201–202
flattening results, 206–207
frequency counts, 23
iterating through groups, 204–206
methods and functions, 192
missing value example, 199–201
multiple groups, 206
one-variable grouped aggregation, 190–191
overview of, 189
saving groupby object without running aggregate, transform, or filter
methods, 202–203
selecting groups, 204
summary/conclusion, 211
transform, 197
working with multiIndex, 207–211
z-score example of transforming data, 197–198

Groups
iterating through, 204–206
selecting, 204
working with multiple, 206

Guido, Sarah, 243

**kwargs, 348
lambda, 185–187
options for applying in and aggregate
methods, 195–197
overview of, 345–347
regular expressions (regex), 165
vectorized, 182–184
z-score example of transforming data, 197–198

G

Gapminder data set, 4
Generalized linear models. See GLM
generaLized linear models
Generators
converting to list, 14–15
overview of, 349–350
get
creating dictionaries, 337–338
selecting groups, 204

**glm function, in statsmodels library, 258
GLM (generalized linear models). See also
Linear models
logistic regression, 253–255
model diagnostics, 273–275
more GLM options, 260
negative binomial regression, 259
overview of, 253
poisson regression, 257
sklearn library for logistic regression, 256–257
statsmodels library for logistic regression, 255–256
statsmodels library for poisson regression, 258–259
summary/conclusion, 263–264
survival analysis using Cox model, 260–263
testing Cox model assumptions, 263–264

Groupby (groupy)
aggregation, 190
Hierarchical clustering (continued)
overview of, 297–298
single cluster algorithm, 298–299

Histograms
creating using `plot.hist` functions, 84
of model residuals, 269
for univariate statistics in `matplotlib`, 57–58
for univariate statistics in `seaborn`, 62–63

Id, unique identifiers, 146
IDEs (integrated development environments), Python, 322–323
`ignore_index` parameter, after concatenation, 98
`iloc`
indexing rows or columns, 8
`Series` attributes, 29
subsetting rows and columns, 17–18
subsetting rows by number, 11–12
subsetting rows or columns, 12–14

Importing (import). See also
Exporting/importing data
`itertools` library, 350
libraries, 331–332
loading first data set, 4–5
`matplotlib` library, 51
`pandas`, 353

Indemics (Interactive Epidemic Simulation) data set, 208

Indices
beginning and ending indices in ranges, 339
concatenate columns with different indices, 101–102
concatenate rows with different indices, 99–101
date ranges, 227–228
issues with absolute, 18
out of bounds notification, 176
re-indexing as source of missing values, 114–116
subsetting columns by index position
break, 8
subsetting date based on, 225–227
subsetting rows by index label, 8–11
working with `multiIndex`, 207–211
`inplace` parameter, functions and methods, 42

Installation
of Anaconda, 315–316
from command line, 317–318

Integers (`int/int64`)
converting to string, 146–148
vectors with integers (scalars), 33–34

Interactive Epidemic Simulation (Indemics) data set, 208

Internet resources, for self-directed learners, 310

Interpolation, in filling missing data, 119

IPython (`ipython`)
`ipython` command, 322–323
magic commands, 306

Iteration. See Loops (`for` loop)
`itertools` library, 350
`ix`
indexing rows or columns, 8
`Series` attributes, 29
subsetting rows, 12

Jointplot, creating `seaborn` scatterplot, 66–69, 71

Jupyter command, 322–323

K

k-fold cross validation, 275–278
k-means
clustering, 291–294
using PCA, 295–297
Index 367

KaplanMeierFitter, lifelines library, 261–263
KDE plot, of bivariate statistics, 70–71
keep_default_na parameter, specifying NaN values, 111
Key-value pairs, 337–338
Key-value stores, 348
Keys, creating DataFrame, 27
Keywords
lambda keyword, 187
passing keyword argument, 173
**kwargs, 347–348

L
L1 regularization, 281–282, 285–287
L2 regularization, 283–284, 285–287
lambda functions, applying, 185–187
Lander, Jared, 243
Leap years/leap seconds, 238
Learning resources, for self-directed learners, 309–311
Libraries. See also by individual types
importing, 331–332
performance libraries, 306
lifelines library
CoxPHFitter class, 261, 263–264
KaplanMeierFitter class, 261–263
Linear models. See also GLM (generalized linear models)
cross-validation, 287–289
elastic net, 285–287
LASSO regression regularization, 281–282
model diagnostics, 270–273
multiple regression, 247
overview of, 243
$R^2$ (coefficient of determination)
regression score function, 277
reasons for regularization, 279–280
residuals, 266–268
restoring labels in sklearn models, 251–252
ridge regression, 283–284
simple linear regression, 243
sklearn library for multiple regression, 249–251
sklearn library for simple linear regression, 245–247
statsmodels library for multiple regression, 247–249
statsmodels library for simple linear regression, 243–245
summary/conclusion, 252
Linux
command line, 318
installing Anaconda, 316
running python and ipython commands, 322
viewing working directory, 325
Lists (list)
comprehensions and, 343
converting generator to, 14–15, 349
creating Series, 26–28
of data types, 145–146
loading multiple files using list comprehension, 140
looping, 341–342
multiple assignment, 351–352
overview of, 333
lmplot
creating scatterplots, 66
with hue parameter, 76
Loading data
datetime data, 217
as source of missing data, 111–112
toc
indexing rows or columns, 8–10
Series attributes, 29
subsetting rows and columns, 17–18
subsetting rows or columns, 12–14
Logistic regression
overview of, 253–255
sklearn library for, 256–257
statsmodels library for, 255–256
working with GLM models, 274
logit function, performing logistic regression, 255–256

Loops (for loop)
- comprehensions and, 343
- loading multiple files using, 139–140
- overview of, 341–342
- through groups, 204–206
- through lists, 341–342

Merges (merge)
- many-to-many, 105–107
- many-to-one, 105
- of multiple data sets, 102–104
- one-to-one, 104
- as source of missing data, 112–113

Methods
- built-in aggregation methods, 191–192
- class, 356
- datetime, 221–224
- Series, 31
- string, 158–161

Mirjalili, Vahid, 243

Missing data (NaN values)
- calculations with, 120–121
- cleaning, 118
- concatenation and, 96, 100
- date range for filling in, 232–233
- dropping, 119–120
- fill forward or fill backward, 118–119
- finding and counting, 116–117, 180
- interpolation in filling, 119
- loading data as source of, 111–112
- merged data as source of, 112–113
- overview of, 109
- re-indexing causing, 114–116
- recoding or replacing (fillna method), 118
- sources of, 111
- specifying with na_values parameter, 111
- summary/conclusion, 121
- transform example, 199–201
- user input creating, 114
- what is a NaN value, 109–111
- working with, 116

Model diagnostics
- comparing multiple models, 270
- k-fold cross validation, 275–278
- overview of, 265
- q-q plots, 268–270
- residuals, 265–268
- summary/conclusion, 278
working with GLM models, 273–275
working with linear models, 270–273

Models
- generalized linear. See GLM (generalized linear models)
- linear. See Linear models

Month, extracting date components from `datetime` object, 217–220
Müller, Andreas, 243

Multiple assignment, 351–352

Multiple regression
- overview of, 247
- residuals, 266–268
- `sklearn` library for, 249–251
- `statsmodels` library for, 247–249

Multivariate statistics
In `matplotlib`, 59–61
In `seaborn`, 73–83

Numbers (numeric)
- converting variables to numeric values, 147–148
- formatting number strings, 162
- negative numbers, 156–157
- `to_numeric downcast`, 151–152
- `to_numeric` function, 148–151

`numpy` library
- broadcasting support, 37–38
- exporting/importing data, 43–45
- functions, 178
- `mean`, 192
- `ndarray`, 353–354
- restoring labels in `sklearn` models, 251–252
- `Series` similarity with `numpy.ndarray`, 30
- `sklearn` library taking `numpy` arrays, 246
- specifying `dtype` from, 146–147
- `vectorize`, 184, 306
- `nunique` method, grouped frequency counts, 23

Object-oriented languages, 355

Objects
- classes, 355–356
- converting to `datetime`, 214–216
- `datetime`, 213–214
- lists as, 333
- plots and plotting using Pandas objects, 83–86

Observational units
- across multiple tables, 137–139
- in a table, 134–137

Odds ratios, performing logistic regression, 256

`odo` library, 47, 357

Offsets, frequency, 229–230
One-to-one merges, 104

OSX. See Mac

Overdispersion of data, negative binomial regression for, 259
### P

<table>
<thead>
<tr>
<th>Packages</th>
<th>benefits of isolated environments, 327–328</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>installing, 329–330</td>
</tr>
<tr>
<td></td>
<td>updating, 330</td>
</tr>
<tr>
<td>pairgrid</td>
<td>bivariate statistics, 73</td>
</tr>
<tr>
<td>Pairwise relationships (pairplot)</td>
<td>bivariate statistics, 73–74 with hue parameter, 77</td>
</tr>
<tr>
<td>Parameters</td>
<td>arbitrary function parameters, 347–348</td>
</tr>
<tr>
<td></td>
<td>default function parameters, 347</td>
</tr>
<tr>
<td></td>
<td>functions taking, 346</td>
</tr>
<tr>
<td>patsy library</td>
<td>276–279</td>
</tr>
<tr>
<td>Patterns. See also Regular expressions (regex)</td>
<td>compiling, 169</td>
</tr>
<tr>
<td></td>
<td>matching, 164–168</td>
</tr>
<tr>
<td></td>
<td>substituting, 168–169</td>
</tr>
<tr>
<td>PCA (principal component analysis), 294–297</td>
<td></td>
</tr>
<tr>
<td>pd</td>
<td>alias for pandas, 5</td>
</tr>
<tr>
<td></td>
<td>reading pickle data, 44–45</td>
</tr>
<tr>
<td>Performance</td>
<td>avoiding premature optimization, 306</td>
</tr>
<tr>
<td></td>
<td>profiling code, 307</td>
</tr>
<tr>
<td></td>
<td>timing execution of statements or expressions, 306–307</td>
</tr>
<tr>
<td>pickle data, 43–45</td>
<td></td>
</tr>
<tr>
<td>Pivot/unpivot</td>
<td>columns containing multiple variables, 128–129</td>
</tr>
<tr>
<td></td>
<td>converting wide data into tidy data, 125–126</td>
</tr>
<tr>
<td></td>
<td>keeping multiple columns fixed, 126–127</td>
</tr>
<tr>
<td></td>
<td>rows and columns both containing variables, 133–134</td>
</tr>
<tr>
<td>Placeholders, formatting character strings, 162</td>
<td></td>
</tr>
<tr>
<td>Plots/plotting (plot)</td>
<td>basic plots, 23–24</td>
</tr>
<tr>
<td></td>
<td>bivariate statistics in matplotlib, 58–59</td>
</tr>
<tr>
<td></td>
<td>creating boxplots (plot.box), 85–86, 88</td>
</tr>
<tr>
<td></td>
<td>creating density plots (plot.kde), 85</td>
</tr>
<tr>
<td></td>
<td>creating scatterplots (plot.scatter), 85–86</td>
</tr>
<tr>
<td></td>
<td>linear regression residuals, 266–268</td>
</tr>
<tr>
<td></td>
<td>matplotlib library, 51–56</td>
</tr>
<tr>
<td></td>
<td>multivariate statistics in matplotlib, 59–61</td>
</tr>
<tr>
<td></td>
<td>multivariate statistics in seaborn, 73–83</td>
</tr>
<tr>
<td></td>
<td>overview of, 49–50</td>
</tr>
<tr>
<td></td>
<td>Pandas objects and, 83–85</td>
</tr>
<tr>
<td></td>
<td>q-q plots, 268–270</td>
</tr>
<tr>
<td></td>
<td>seaborn library, 61</td>
</tr>
<tr>
<td></td>
<td>statistical graphics, 56–57</td>
</tr>
<tr>
<td></td>
<td>summary/conclusion, 90</td>
</tr>
<tr>
<td></td>
<td>themes and styles in seaborn, 86–90</td>
</tr>
<tr>
<td></td>
<td>univariate statistics in matplotlib, 57–58</td>
</tr>
<tr>
<td></td>
<td>univariate statistics in seaborn, 62–65</td>
</tr>
<tr>
<td>PLOT_TYPE functions, 83</td>
<td></td>
</tr>
<tr>
<td>plt.hexbin function, 86–87</td>
<td></td>
</tr>
<tr>
<td>Podcast resources, for self-directed learners, 310–311</td>
<td></td>
</tr>
<tr>
<td>Point representation, Anscombe’s data set, 52</td>
<td></td>
</tr>
<tr>
<td>poisson function, in statsmodels library, 258</td>
<td></td>
</tr>
<tr>
<td>Poisson regression</td>
<td>negative binomial regression as alternative to, 259</td>
</tr>
<tr>
<td></td>
<td>overview of, 257</td>
</tr>
<tr>
<td></td>
<td>statsmodels library for, 258–259</td>
</tr>
<tr>
<td>Position, subsetting columns by index position break, 8</td>
<td></td>
</tr>
<tr>
<td>Principal component analysis (PCA), 294–297</td>
<td></td>
</tr>
<tr>
<td>Project templates, 319, 325</td>
<td></td>
</tr>
<tr>
<td>Pycon, conference resource for self-directed learners, 310</td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td>Anaconda distribution, 327</td>
</tr>
<tr>
<td></td>
<td>command line and text editor, 321–322</td>
</tr>
<tr>
<td></td>
<td>comparing Pandas types with, 6</td>
</tr>
<tr>
<td></td>
<td>conferences, 310</td>
</tr>
<tr>
<td></td>
<td>enhanced features in Pandas, 3</td>
</tr>
</tbody>
</table>
IDEs (integrated development environments), 322–323
ipython command, 322–323
jupyter command, 322–323
as object-oriented languages, 355
running from command line, 317–318
scientific computing stack, 305
ways to use, 321
working with objects, 5
as zero-indexed languages, 339

Q

q-q plots, model diagnostics, 268–270

R

R language, interface with (to feather method), 47
random.shuffle method, directly changing columns, 41–42
Ranges (range)
  beginning and ending indices, 339
date ranges, 227–228
filling in missing values, 232–233
overview of, 349–350
passing range of values, 333
subsetting columns, 14–15
Raschka, Sebastian, 243
re module, 164, 170
Regex. See Regular expressions (regex)
regplot, creating scatterplot, 65–66
Regression
  LASSO regression regularization, 281–282
  logistic regression, 253–255
  more GLM options, 260
  multiple regression, 247
  negative binomial regression, 259
  poisson regression, 257
  reasons for regularization, 279–281
  restoring labels in sklearn models, 251–252
  ridge regression regularization, 283–284
  simple linear regression, 243
  sklearn library for logistic regression, 256–257
  sklearn library for multiple regression, 249–251
  sklearn library for simple linear regression, 245–247
  statsmodels library for logistic regression, 255–256
  statsmodels library for multiple regression, 247–249
  statsmodels library for poisson regression, 258–259
  statsmodels library for simple linear regression, 243–245
Regular expressions (regex)
  overview of, 164
  pattern compilation, 169
  pattern matching, 164–168
  pattern substitution, 168–169
  regex library, 170
  syntax, special characters, and functions, 165
Regularization
  cross-validation, 287–289
  elastic net, 285–287
  LASSO regression, 281–282
  overview of, 279
  reasons for, 279–281
  ridge regression, 283–284
  summary/conclusion, 289
reindex method, re-indexing as source of missing values, 114–116
Resampling, datetime, 237–238
Residual sum of squares (RSS), 272
Residuals, model diagnostics, 265–268
Ridge regression
  elastic net and, 285–287
  regularization techniques, 283–284
Rows
  apply row-wise operations, 180–182
  concatenation generally, 94–97
  concatenation with different indices, 99–101
Rows (continued)
  multiple observational units in a table, 134–137
  removing row numbers from output, 46
  rows and columns both containing variables, 133–134
  subsetting rows and columns, 17–18
  subsetting rows by index label, 8–11
  subsetting rows by ix attribute, 12
  subsetting rows by row number, 11–12
RSS (residual sum of squares), 272
Rug plots, for univariate statistics, 63–65

S
Scalars, 33–34
Scaling up, going bigger and faster, 307
Scatterplots
  for bivariate statistics, 58, 65–67
  matplotlib example, 54
  for multivariate statistics, 60–61
  plot.scatter function, 85–86
Scientific computing stack, 305
scipy library
  hierarchical clustering, 297
  performance libraries, 306
  scientific computing stack, 305
Scripts
  project templates for running, 325
  running Python from command line, 317–318
seaborn
  Anscombe’s quartet for data visualization, 50
  bivariate statistics, 65–73
  multivariate statistics, 73–83
  overview of, 61
  themes and styles, 86–90
  tips data set, 199
  titanic data set, 176
  univariate statistics, 62–65
Searches. See Find
Self-directed learners, resources for, 309–311
Semicolon (;), types of delimiters, 45
Serialization, serialize and save data in binary format, 43–45
Series
  adding columns, 38–39
  aggregation functions, 196–197
  alignment and vectorization, 33–36
  apply function(s) over, 173–174
  attributes, 29
  boolean subsetting, 30–33
  categorical attributes or methods, 153
  as class, 355–356
  creating, 26
  defined, 3
  directly changing columns, 39–42
  exporting/importing data, 43–45
  exporting to Excel (to_excel method), 46
  histogram, 84
  methods, 31
  overview of, 28–29
  similarity with ndarray, 30
  writing CSV files (to_csv method), 45–46
shape
  DataFrame attributes, 5
  Series attributes, 29
Shape, in plotting, 77–78
Shell scripts, running Python from command line, 317–318
Simple linear regression
  overview of, 243
  sklearn library, 245–247
  statsmodels library, 243–245
Single cluster algorithm, in hierarchical clustering, 298–299
size attribute, Series, 29
Size, in plotting, 77–78
sklearn library
  importing PCA function, 294
  K-fold cross validation, 276–278
  KMeans function, 293
  for logistic regression, 256–257
  for multiple regression, 249–251
restoring labels in sklearn models, 251–252
for simple linear regression, 245–247
splitting data into training and testing sets, 279–280

Slicing
colon (:) use in slicing syntax, 13, 339–340
columns, 15–17
string from beginning or to end, 157–158
strings, 156–157
strings incrementally, 158
subsetting columns, 13–14
subsetting multiple rows and columns, 17–18
values, 339–340

snakevis, profiling code, 307
sns.distplot, creating histograms, 62–63
Sns.set_style function, 86–90

Special characters, regular expressions, 165
Split-apply-combine, 189
split method
split and add columns individually, 129–131
split and combine in single step, 131–133
splitlines method, strings, 160–161
Spyder IDE, 322
SQL
comparing Pandas to, 104
groupy compared with SQL GROUP BY, 189
odo library support, 357

Square brackets ([[]])
getting first character of string, 156
list syntax, 333

Statistical graphics
bivariate statistics in matplotlib, 58–59
bivariate statistics in seaborn, 65–73
matplotlib library, 51–56
multivariate statistics in matplotlib, 59–61
multivariate statistics in seaborn, 73–83
overview of, 56–57

seaborn library, 61
univariate statistics in matplotlib, 57–58
univariate statistics in seaborn, 62–65

Statistics
basic plots, 23–24
grouped and aggregated calculations, 18–19
grouped frequency counts, 23
grouped means, 19–22

statsmodels library
for logistic regression, 255–256
for multiple regression, 247–249
for poisson regression, 258–259
for simple linear regression, 243–245

Stocks/stock prices, 224–225

Storage
of information in dictionaries, 337–338
lists for data storage, 333

str accessor, 129

Strings (string)
accessing methods, 129
converting values to, 146–147
formatting, 161–164
getting last character in, 157–158
methods, 158–161
overview of, 155
pattern compilation, 169
pattern matching, 164–168
pattern substitution, 168–169
regular expressions (regex) and, 164, 170
subsetting and slicing, 155–157
summary/conclusion, 170

strptime, for date formats, 216
str.replace, pattern substitution, 168–169

Styles, seaborn, 86–90

Subsets/subsetting
columns by index position break, 8
columns by name, 7–8
columns by range, 14–15
columns generally, 17–18
columns using slicing syntax, 13–14
data by dates, 225–227

Dataframe boolean subsetting, 36–37
Subsets/subsetting (continued)
- lists, 333
- multiple rows, 12
- rows by index label, 8–11
- rows by ix attribute, 12
- rows by row number, 11–12
- rows generally, 17–18
- strings, 155–157
- tuples, 335

**sum**
- cumulative (cumsum), 210–211
- custom functions, 194

**Summarization. See Aggregation (or aggregate)**

**Survival analysis, using Cox model, 260–263**

**SWC Windows Installer, 317**

**SymPy, 305**

**T**
- attribute, Series, 29
- Tab separated values (TSV), 45, 217
- Tables
  - observational units across multiple, 137–139
  - observational units in, 134–137
- tail, returning last row, 10
- Templates, project, 319, 325
- Terminal application, Mac, 317–318
- Text. See also Characters; Strings (string)
  - function documentation (docstring), 172
  - overview of, 155
- Themes, seaborn, 86–90
- Tidy data
  - columns containing multiple variables, 128–129
  - columns containing values not variables, 124
  - data assembly, 93–94
  - keeping multiple columns fixed, 126–127
  - keeping one column fixed, 124–126
  - loading multiple files using list comprehension, 140
- loading multiple files using loop, 139–140
- observational units across multiple tables, 137–139
- observational units in a table, 134–137
- overview of, 123–124
- rows and columns both containing variables, 133–134
- split and add columns individually, 129–131
- split and combine in single step, 131–133
- summary/conclusion, 141

**Time. See datetime**

**Time zones, 238–239**

**timedelta object**
- date calculations, 220–221
- subsetting date based data, 226–227
- TimedeltaIndex, 226–227
- timeit function, timing execution of statements or expressions, 306–307
- tips data set, seaborn library, 199, 243
- to_csv method, 45–46
- to_datetime function, 214–216
- to_excel method, 46
- to_feather method, 47
- to_numeric function, 148–152

**Transform (transform)**
- applying to data, 269–270
- missing value example of transforming data, 199–201
- overview of, 197
- z-score example of transforming data, 197–198

**TSV (tab separated values), 45, 217**

**Tuples (tuple), 335**

**type function, working with Python objects, 5**

**U**

**Unique identifiers, 146**

**Univariate statistics**
- in matplotlib, 57–58
- in seaborn, 62–65
Updates, package, 330
urllib library, 134–137
User input, as source of missing data, 114

V

columns containing values not variables. See Columns, with values not variables
converting to strings, 146–147
creating DataFrame values, 27
directly changing columns, 39–42
dropping, 43
functions taking, 346
missing. See Missing data (NaN values)
multiple assignment of list of, 351–352
passing/reassigning, 333
Series attributes, 29
shifting datetime values, 230–237
slicing, 339–340
VanderPlas, Jake, 305
Variables
adding covariates to linear models, 270
bi-variable statistics. See Bivariate statistics
calculations involving multiple, 203–204
columns containing multiple. See Columns, with multiple variables
columns containing values not variables. See Columns, with values not variables
converting to numeric values, 147–148
multiple assignment, 351–352
multiple linear regression with three covariates, 266–268
multiple variable statistics. See Multivariate statistics
one-variable grouped aggregation, 190–191
rows and columns both containing, 133–134
in simple linear regression, 243
single variable statistics. See Univariate statistics

sklearn library used with categorical variables, 250–251
statsmodels library used with categorical variables, 248–249
Vectors (vectorize)
applying vectorized function, 182–184
with common index labels (automatic alignment), 35–36
DataFrame alignment and vectorization, 37–38
Series alignment and vectorization, 33
Series referred to as vectors, 30
using numba library, 185
using numpy library, 184
vectors of different length, 34–35
vectors of same length, 33
vectors with integers (scalars), 33–34
Violin plots
bivariate statistics, 73
creating scatterplots, 71
with hue parameter, 76
Visualization
Anscombe’s quartet for data visualization, 49–50
using plots for, 23–24

W

Wickham, Hadley, 93–94, 123
“Wide” data, converting into tidy data, 125–126
Windows
Anaconda command prompt, 322
cd command for viewing working directory, 325
command line, 317
installing Anaconda, 315

X

xarray library, 305
xrange, 349–350
<table>
<thead>
<tr>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year, extracting date components from <em>datetime</em> object, 217–220</td>
<td><em>z</em>-score, transforming data, 197–198</td>
</tr>
<tr>
<td></td>
<td>Zero-indexed languages, 339</td>
</tr>
</tbody>
</table>