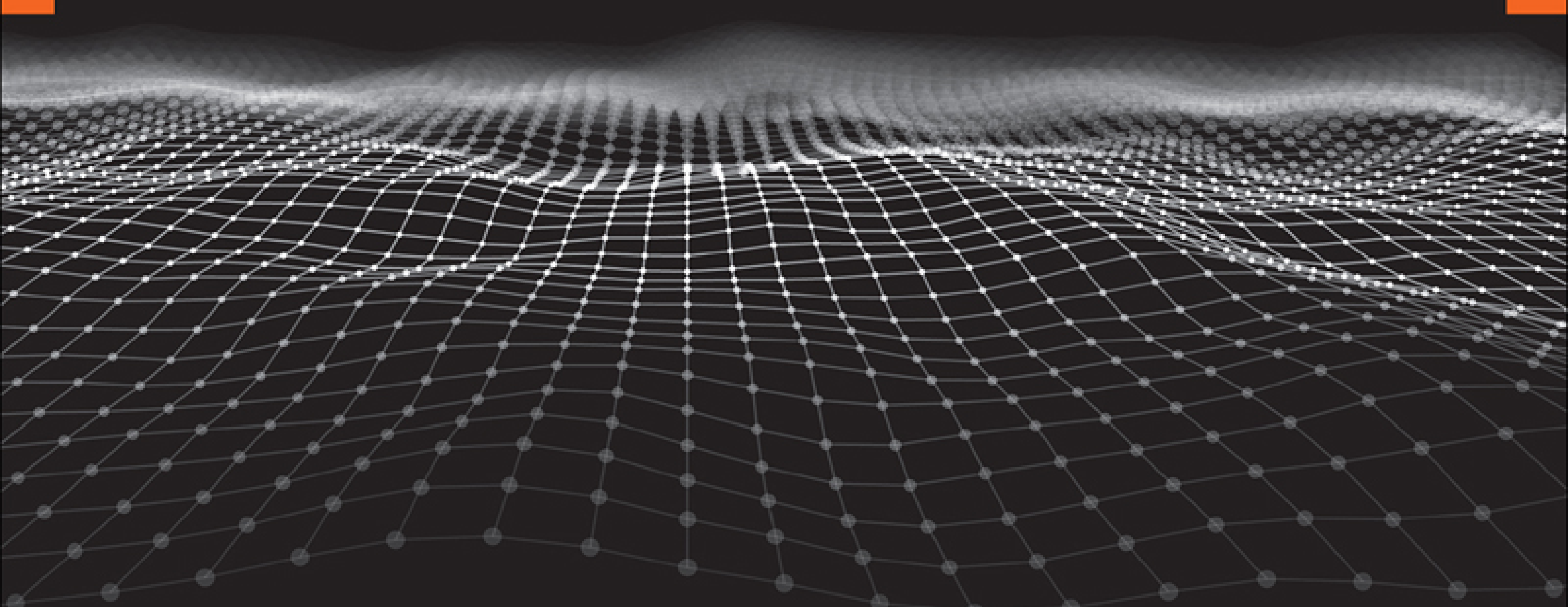
Addison-Wesley Data & Analytics Series

Pandas for Everyone

Python Data Analysis

Second Edition



Daniel Y. Chen

About This eBook

ePUB is an open, industry-standard format for eBooks. However, support of ePUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

Pandas for Everyone

The Pearson Addison-Wesley Data & Analytics Series

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

The **Pearson Addison-Wesley Data & 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.

Pandas for Everyone

Python Data Analysis

Second Edition

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

Cover image: SkillUp / Shutterstock
Figure 3.7: The Matplotlib development team
Figure B1 (Appendix): GitHub, Inc.

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

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

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

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2022948110

ISBN-13: 978-0-13-789115-3
ISBN-10: 0-13-789115-6

ScoutAutomatedPrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

To all the teachers, advisors, and mentors I've had over the years.
And to my family: Mom, Dad, Eric, and Julia

Contents

Foreword to Second Edition

As the data science domain and educational landscape continues to evolve, there is an increasing need to train individuals to critically consider data both holistically and logically. Each year, given the advancement in computational power, magnitude of data, and data-informed decisions to make, more and more individuals are dipping their toes in the water of data science—and most are not aware of how messy their data sets are. Working with messy data is challenging, confusing, and not necessarily exciting, especially for newcomers. To continue to use data for informed decision-making, it is important to introduce concepts in data logic, planning, and purpose early in the stages of training best practices. The how, why, and lessons learned of teaching data science represent huge areas of exploration given the exponential increase in learners. There are numerous resources, MOOCs, Twitter threads, packages, cheat-sheets, and more out there for individuals to learn data science, either on their own or in a class. However, what is effective and what pathways are best for certain learner personas? Moreover, how does someone new to the field choose which educational resources mesh with their needs and background familiarity?

While spending many years as an educator for RStudio and The Carpentries, Dr. Daniel Chen recognized this need, and it has become his passion to introduce learners to core concepts to work with their data in more effective, reproducible, and reliable methods in an environment matching their comfort level with the field. I met Dan by semi-random chance and after a few conversations, we were well on our way with a dissertation topic stemming from these interests. With a shared passion in educating others in foundational data science methods and looking into those “hows” and “whys” of the ways in which we were teaching, we sought to understand our learners first and then create materials. It was a pleasure to work with Dan on his dissertation—and to see those insights incorporated here in *Pandas for Everyone, Second Edition*.

In the second edition, Dan takes learners step-by-step through practical scratch code examples for using Pandas. Using Pandas helps demystify Python data analysis, create organized manageable data sets, and, most importantly, have tidy data sets! It takes a special educator to get individuals (myself included!) excited about cleaning data, but that is what Dan does for his learners in *Pandas for Everyone*. Visualizing and modeling data are taught in easy-to-interpret style once learners become comfortable with manipulating and transforming their data sets, all of which is covered in sequential order. It is this mindset and presentation of materials that really makes this book for everyone—and aids the learner in best practices while working with example data sets that mimic data sets they might use in real life. *Pandas for Everyone, Second Edition*, is a quick but detailed foray for new data scientists, instructors, and more to experience best practices and the massive potential of Pandas in a clear-cut format.

–Anne M. Brown, PhD (she/her)
Assistant Professor
Data Services—University Libraries
Department of Biochemistry
Virginia Tech, Blacksburg, VA 24061

Foreword to First Edition

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

Preface

My foray into teaching was in 2013 when I attended my first Software-Carpentry workshop, and I've been involved in teaching ever since. In 2019, I was lucky enough to be one of the RStudio (now Posit, PBC) interns with the education group. By then, data science education has already gained a tremendous amount of momentum. When I finished my internship, I needed a dissertation topic for my degree, and wanted to combine teaching with medicine. Luckily, I knew a librarian at the university, Andi Ogier, who connected me with Anne Brown, who was also interested in teaching data literacy skills in the health sciences. The rest is history. Anne became my PhD chair, and with the rest of my committee, Dave Higdon, Alex Hanlon, and Nikki Lewis, I got to do research on data science education in the medical and biomedical sciences.[1] The first edition of the book became a foundation for what data science topics were taught for the workshop component of the dissertation. The second edition of *Pandas for Everyone* incorporates many of the things I've learned while studying education and pedagogy.

1. You can learn more about my dissertation around data science education here: `https://github.com/chendaniely/dissertation`

Long story short, befriend a librarian. Their profession revolves around data.

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

Methods course (essentially a combination of a first-semester epidemiology and biostatistics course). I've been learning and teaching ever since.

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

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

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

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

Breakdown of the Book

This book is organized into multiple parts plus a set of appendices.

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.
- Chapter 4: Discusses Hadley Wickham's "Tidy Data" paper, which deals with reshaping and cleaning common data problems.
- Chapter 5: 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.

Part II

Part II focuses on what happens after you load data and need to further process your data.

- Chapter 6: Focuses on combining data sets, either by concatenating them together or by merging disparate data.
- Chapter 7: Normalizes data for more robust data storage.
- Chapter 8: Describes `'.groupby()'` operations (i.e., split-apply-combine). These powerful concepts, like `'.apply()'`, are often needed to scale data. They are also great ways to efficiently aggregate, transform, or filter your data.

Part III

Part III covers the types of data stored in columns.

- Chapter 9: 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 10: Deals with data types and how to convert from different types within `'DataFrame'` columns.
- Chapter 11: Introduces string manipulation, which is frequently needed as part of the data cleaning task because data are often encoded as text.

- Chapter 12: 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 13: Linear models are the simpler models to fit. This chapter covers fitting these models using the `'statsmodels'` and `'sklean'` libraries.
- Chapter 14: 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.
- Chapter 15: Covers survival models, which is what you use when you have data censoring.
- Chapter 16: 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 17: Regularization is a technique used when the models we are fitting are too complex or overfit our data.
- Chapter 18: 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 19: Quickly summarizes the computation stack in Python, and starts down the path to code performance and scaling.

- Chapter 20: Provides some links and references on learning beyond the book.

Appendices

The appendices 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.

- Appendix A: Provides concept maps for the introductory chapters to help breakdown and relate concepts to one another.
- Appendixes B–J: These appendices 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 K–Y: These appendices cover general programming concepts that are relevant to Python and Pandas. They are supplemental references to the main part of the book.
- Appendix Z: Replicates some of the modeling code in R as a reference to compare similar results.

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 Appendix A - Appendix J 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 maps and learning 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 I, 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.

The concept maps in Appendix A and the learning objectives provided in the earlier chapters should help contextualize how concepts are related to one another.

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:

Click here to view code image

```
https://github.com/chendaniely/pandas_for_every
one
```

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

Get the Data

The easiest way to get all the data to code along the book is to download the ZIP file of the book's repository here:

Click here to view code image

```
https://github.com/chendaniely/pandas_for_every
one
```

The book's repository will have the latest instructors on how to download the book's data, and more detailed instructors for how to get the book can be found in Appendix B.3.

Setup Python

Appendix G and Appendix H cover environments and installing packages, respectively. There you will find the URLs and commands on how to setup Python to code along the book. Again, the book's repository will always contain the latest set of instructions.

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` or `P4E` 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, Second Edition*, 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 (9780137891153) 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

So many people have made this book happen, in addition to the folks from the first edition (see additional acknowledgments below).

The people who helped with the book logistics: Mary Roth and Debra Williams Cauley with the book production, Cody Huddleston and Gloria W with copy editing.

My PhD committee: Anne Brown, Dave Higdoen, Alex Hanlon, and Nikki Lewis, for getting me to think about teaching and pedagogy over the years and improving the book to make it better suited for teaching and learning.

All my students who gave me an opportunity to teach you and hone how to present materials.

Everyone who has sent me corrections over the years. In particular, Sam Johnson, who gave me a cover-to-cover set of improvements.

The creators, maintainers, and contributors to the Quarto scientific and technical publishing system for creating a tool to make the book much more maintainable.[2]

2. Quarto: https://quarto.org/docs/books

Finally, my friends and family who have helped me get through graduate school and provided feedback during the book writing.

Acknowledgments from the First Edition

Introduction to Data Science: The three people who paved the way for this book were my instructors in the “Introduction to Data Science” course at Columbia—Rachel Schutt, Kayur Patel, and Jared Lander. Without them, I wouldn’t even know what the term “data science” means. I learned

so much about the field through their lectures and labs; everything I know and do today can be traced back to this class. The instructors were only part of the learning process. The people in my study group, where we fumbled through our homework assignments and applied our skills to the final project of summarizing scientific articles, made learning the material and passing the class possible. They were Niels Bantilan, Thomas Vo, Vivian Peng, and Sabrina Cheng (depicted in the figure here). Perhaps unsurprisingly, they also got me through my master's program (more on that later).

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

Software-Carpentry: As part of the "Introduction to Data Science" course, I attended a Software-Carpentry workshop, where I was first introduced to Pandas. My first instructors were Justin Ely and David Warde-Farley. Since then I've been involved in the community, thanks to Greg Wilson, and still remember the first class I helped teach, led by Aron

Ahmadia and Randal S. Olson. The many workshops that I've taught since then, and the fellow instructors whom I've met, gave me the opportunity to master the knowledge and skills I know and practice today, and to disseminate them to new learners, which has cumulated into this book.

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

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

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

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

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

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

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

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

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

Even more thanks go to Gloria Kang, Jacquie Cheung, and Jared Lander for their feedback during the writing process. I also want to thank Chris Zahn for all the work in reviewing the book from cover to cover, and Kaz Sakamoto and Madison Arnsbarger for providing feedback and reviews.

Through their many conversations with me, M Pacer, Sebastian Raschka, Andreas Müller, and Tom Augspurger helped me make sure I covered my bases, and did things "properly."

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

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

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

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

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

About the Author

Daniel Y. Chen, PhD, MPH, completed his PhD at Virginia Tech in Genetics, Bioinformatics, and Computational Biology (GBCB). His dissertation was on data science education in the medical and biomedical sciences. He completed a Master's of Public Health in Epidemiology at Columbia University Mailman School of Public Health, where he studied how attitudes toward behaviors diffuse and spread in social networks. In a past life, he studied psychology and neuroscience at the Macaulay Honors College at CUNY Hunter College and worked in a bench laboratory doing microscopy work looking at proteins in the brain associated with learning and memory.

Daniel currently works as a Postdoctoral Research and Teaching Fellow at the University of British Columbia and as a Data Science Educator at Posit, PBC (formerly, RStudio, PBC). He has been involved with The Carpentries as an instructor, instructor trainer, and community maintainer lead.

Changes in the Second Edition

The second edition mainly updates all the code and libraries to the latest versions at the time of writing. Most of the code form the first edition was unaffected. Bits of the plotting code and machine learning data modeling code ended up changing over the years and were updated.

From a pedagogical perspective, the main Pandas chapters have also been updated with proper learning objectives, and the introductory chapters have accompanying concept maps to help educators plan a learning path, and for learners to visualize how concepts are related to one another. These were all topics I've learned about while doing my dissertation, and I hope they become useful for learners and educators. The book also includes access to online bonus chapters on geopandas, Dask, and creating interactive graphics with Altair.

I've also rearranged the chapters in the second edition based on my experiences when I teach workshops. Part I of the book contains the most important bits of information that I aim to cover in my workshops. The rest of the book can be thought of as data processing details after the more fundamental topics are covered. The chapters that have big changes from the first edition have a section in the chapter's introduction on the details of what has changed.

Many of the libraries and tools mentioned in the conclusion chapters of the book will also have freely available chapters to accompany this book to help you extend your learning.

Part I

Introduction

This book begins with an introduction to the Pandas Python library for data analytics. It first covers the very basics of using the `pandas` library, loading your first data set and doing basic filtering and subsetting commands with your data (Chapter 1). It then goes into more detail about the `DataFrame` and `Series` objects, where we cover more of the attributes and methods these objects can do, including how to save data sets for storage (Chapter 2). It then pivots into data visualization with `matplotlib` and `seaborn` plotting libraries as well as the built-in `pandas` plotting methods (Chapter 3). Next, this part covers one of the fundamental concepts in data literacy, tidy data principles. Where it discusses what a "clean" and "tidy" data set looks like so you can process data with a goal and target in mind (Chapter 4). Finally, this part covers writing functions and applying them to your data, and lays down the foundation for any custom data processing steps in the future (Chapter 5).

Think of this part of the book as the core data literacy knowledge on how to work and think about your data. It also aims to teach you the relevant bits of the Python programming language by using the Pandas library as the motivational use case.

1

Pandas DataFrame Basics

1.1 Introduction

Pandas is an open-source Python library for data analysis. It gives Python the ability to work with spreadsheet-like data for fast data loading, manipulating, aligning, merging, etc. To give Python these enhanced features, Pandas introduces two new data types to Python: `Series` and `DataFrame`. The `DataFrame` will represent your entire spreadsheet or rectangular data, whereas the Series is a single column of the `DataFrame`. A Pandas `DataFrame` can also be thought of as a dictionary or collection of Series.

Why should you use a programming language like Python and a tool like Pandas to work with data? It boils down to automation and reproducibility. If there is a particular set of analyses that needs to be performed on multiple data sets, a programming language can automate the analysis on the data sets. Although many spreadsheet programs have their own macro programming languages, many users do not use them. Furthermore, not all spreadsheet programs are available on all operating systems. Performing data tasks using a programming language forces the user to have a running record of all steps performed on the data. I, like many people, have accidentally hit a key while viewing data in a spreadsheet program, only to find out that my results do not make any sense anymore due to bad data. This is not to say spreadsheet programs are bad or do not have their place in the data workflow. They do, but there are better and more reliable tools out there. These better tools can work in tandem with spreadsheet programs while providing more reliable data manipulation, and introduce the possibility of incorporating data from other data sets and databases.

Learning Objectives

The concept map for this chapter can be found in Figure A.1.

- Use Pandas functions to load a simple delimited data file
- Calculate how many rows and columns were loaded
- Identify the type of data that were loaded
- Name differences between functions, methods, and attributes
- Use methods and attributes to subset rows and columns
- Calculate basic grouped and aggregated statistics from data
- Use methods and attributes to create a simple figure from data

1.2 Load Your First Data Set

When given a data set, we first load it and begin looking at its structure and contents. The simplest way of looking at a data set is to look at and subset specific rows and columns. We can see what type of information is stored in each column, and can start looking for patterns by aggregating descriptive statistics.

Since Pandas is not part of the Python standard library, we have to first tell Python to load (i.e., `import`) the library. If you have not installed data and packages needed to go through the book please see Appendix B.

```
import pandas
```

With the library loaded we can use the `read_csv()` function to load a CSV data file. In order to access the `read_csv()` function from pandas, we use something called "dot notation." More on dot notations can be found in Appendix L, Appendix P, and Appendix E. We write `pandas.read_csv()` to say: within the `pandas` library we just loaded, look inside for the `read_csv()` function.

About the Gapminder Data Set

The Gapminder data set originally comes from https://www.gapminder.org/. This particular version of the book is using Gapminder data prepared by Jennifer Bryan from the

University of British Columbia (now at Posit, PBC, formerly RStudio, PBC). The repository can be found at https://github.com/jennybc/gapminder/.

Click here to view code image

```
# by default read_csv() will read a comma
separated file,
# our gapminder data set is separated by a tab
# we can use the sep parameter and indicate a
tab with \t
df  = pandas.read_csv('./data/gapminder.tsv',
sep='\t')
# print out the data
print(df)
```

```
           country continent  year  lifeExp
pop      gdpPercap
0      Afghanistan      Asia  1952   28.801
8425333     779.445314
1      Afghanistan      Asia  1957   30.332
9240934     820.853030
2      Afghanistan      Asia  1962   31.997
10267083     853.100710
3      Afghanistan      Asia  1967   34.020
11537966     836.197138
4      Afghanistan      Asia  1972   36.088
13079460     739.981106
...            ...       ...   ...      ...
...            ...
1699      Zimbabwe    Africa  1987   62.351
9216418     706.157306
1700      Zimbabwe    Africa  1992   60.377
```

```
10704340     693.420786
1701       Zimbabwe    Africa  1997   46.809
11404948     792.449960
1702       Zimbabwe    Africa  2002   39.989
11926563     672.038623
1703       Zimbabwe    Africa  2007   43.487
12311143     469.709298

[1704 rows x 6 columns]
```

Since we will be using Pandas functions many times throughout the book as well as in your own programming. It is common to give `pandas` the alias `pd`. The above code will be the same as below:

Click here to view code image

```
import pandas as pd
df = pd.read_csv('./data/gapminder.tsv',
sep='\t')
```

We can check to see if we are working with a Pandas `Dataframe` by using the built-in `type()` function (i.e., it comes directly from Python, not a separate library such as Pandas).

Click here to view code image

```
print(type(df))
```

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

The `type()` function is handy when you begin working with many different types of Python objects and need to know what object you are currently working on.

The data set we loaded is currently saved as a Pandas `DataFrame` object (`pandas.core.frame.DataFrame`) and is relatively small.

Every `DataFrame` object has a `.shape` attribute that will give us the number of rows and columns of the `DataFrame`.

Click here to view code image

```
# get the number of rows and columns
print(df.shape)
```

```
(1704, 6)
```

The shape attribute returns a tuple (Appendix G) where the first value is the number of rows and the second value is the number of columns.

From the results above, we see our gapminder data set has 1704 rows and 6 columns.

Since `.shape` is an attribute of the `DataFrame` object, and not a function or method of the `DataFrame` object, it does not have round parentheses after the period (i.e., it's written as `df.shape` and not `df.shape()`). If you made the mistake of putting parentheses after the `.shape` attribute, it would return an error.

Click here to view code image

```
# shape is an attribute, not a method
# this will cause an error
print(df.shape())
```

```
TypeError: 'tuple' object is not callable
```

Typically, when first looking at a data set, we want to know how many rows and columns there are (we just did that). To get a gist of what information the data set contains, we look at the column names. The column names, like `.shape`, are given using the `.column` attribute of the `DataFrame` object.

Click here to view code image

```
# get column names
print(df.columns)
```

```
Index(['country', 'continent', 'year', 'lifeExp',
'pop',
       'gdpPercap'],
      dtype='object')
```

Question

What is the `type` of the column names?

The Pandas `DataFrame` object is similar to other languages that have DataFrame-like objects (e.g., Julia and R). Each column (i.e., `Series`) has to be the same type, whereas each row can contain mixed types. In our current example, we can expect the `country` column to be all strings, and the `year` to be integers. However, it's best to make sure that is the case by using the `.dtypes` attribute or the `.info()` method. Table 1.1 shows what the type in Pandas is relative to native Python.

Click here to view code image

```
# get the dtype of each column
print(df.dtypes)
```

```
country        object
continent      object
year            int64
lifeExp       float64
pop             int64
gdpPercap     float64
dtype: object
```

Click here to view code image

```
# get more information about our data
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
 #    Column     Non-Null Count  Dtype
---   ------     --------------  -----
 0    country    1704 non-null   object
 1    continent  1704 non-null   object
 2    year       1704 non-null   int64
 3    lifeExp    1704 non-null   float64
 4    pop        1704 non-null   int64
 5    gdpPercap  1704 non-null   float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
None
```

1.3 Look at Columns, Rows, and Cells

Now that we're able to load up a simple data file, we want to be able to inspect its contents. We could `print()` out the contents of the `DataFrame`, but with today's data, there are too many cells to make sense of all the printed information. Instead, the best way to look at our data is to inspect it by looking at various subsets of the data. We can use the `.head()` method of a `DataFrame` to look at the first 5 rows of our data.

Table 1.1 Table of Pandas `dtypes` and Python Types

Pandas	Python	Description
object	string	most common data type

Pandas	Python	Description
int64	int	whole numbers
float64	float	numbers with decimals
datetime64	datetime	datetime is found in the Python standard library (i.e., it is not loaded by default and needs to be imported)

Click here to view code image

```
# show the first 5 observations
print(df.head())
```

```
       country continent  year  lifeExp       pop
gdpPercap
0  Afghanistan      Asia  1952   28.801   8425333
779.445314
1  Afghanistan      Asia  1957   30.332   9240934
820.853030
2  Afghanistan      Asia  1962   31.997  10267083
853.100710
3  Afghanistan      Asia  1967   34.020  11537966
836.197138
4  Afghanistan      Asia  1972   36.088  13079460
739.981106
```

This is useful to see if our data loaded properly, and to get a better sense of the columns and contents. However, there are going to be times when we only want particular rows, columns, or values from our data.

Before continuing, make sure you are familiar with Python containers (Appendix F, Appendix H).

1.3.1 Select and Subset Columns by Name

If we want only a specific column from our data, we can access the data using square brackets, `[ ]`.

Click here to view code image

```
# just get the country column and save it to its
own variable
country_df = df['country']
```

Click here to view code image

```
# show the first 5 observations
print(country_df.head())
```

```
0     Afghanistan
1     Afghanistan
2     Afghanistan
3     Afghanistan
4     Afghanistan
Name: country, dtype: object
```

Click here to view code image

```
# show the last 5 observations
print(country_df.tail())
```

```
1699    Zimbabwe
1700    Zimbabwe
1701    Zimbabwe
1702    Zimbabwe
```

```
1703    Zimbabwe
Name: country, dtype: object
```

In order to specify multiple columns by the column name, we need to pass in a Python `list` between the square brackets. This may look a bit strange since there will be 2 sets of square brackets, `[[ ]]`.

The outer set of square brackets tells us that we are subsetting our `DataFrame` by columns. The inner set of square brackets tells us the list of columns we want to use. That is, Python also uses square brackets, `[ ]`, to "list" multiple things as a single object.

Click here to view code image

```
# Looking at country, continent, and year
subset = df[['country', 'continent', 'year']]
```

Click here to view code image

```
print(subset)
```

```
          country continent  year
0     Afghanistan      Asia  1952
1     Afghanistan      Asia  1957
2     Afghanistan      Asia  1962
3     Afghanistan      Asia  1967
4     Afghanistan      Asia  1972
...           ...       ...   ...
1699     Zimbabwe    Africa  1987
1700     Zimbabwe    Africa  1992
1701     Zimbabwe    Africa  1997
1702     Zimbabwe    Africa  2002
1703     Zimbabwe    Africa  2007

[1704 rows x 3 columns]
```

Using the square bracket notation, `[ ]`, you cannot pass an index position to subset a `DataFrame` based on the position of the columns. If you want to do this, look down for the `.iloc[]` notation.

Click here to view code image

```
# subset the first column based on its position.
df[0]
```

```
KeyError: 0
```

1.3.1.1 Single Value Returns `DataFrame` or `Series`

When we first selected a single column we were given a `Series` object back.

Click here to view code image

```
country_df = df['country']
print(type(country_df))
```

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

We can also tell it's a `Series` because it prints out slightly differently from the `DataFrame`.

Click here to view code image

```
print(country_df)
```

```
0        Afghanistan
1        Afghanistan
2        Afghanistan
3        Afghanistan
4        Afghanistan
             ...
```

```
1699        Zimbabwe
1700        Zimbabwe
1701        Zimbabwe
1702        Zimbabwe
1703        Zimbabwe
Name: country, Length: 1704, dtype: object
```

Compare those results to passing in a single element list (note the double square bracket, `[[ ]]`):

Click here to view code image

```
country_df_list = df[['country']] # note the
double square bracket
print(type(country_df_list))
```

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

If we use a list to subset, we will *always* get a `DataFrame` object back.

Click here to view code image

```
print(country_df_list)
```

```
          country
0     Afghanistan
1     Afghanistan
2     Afghanistan
3     Afghanistan
4     Afghanistan
...           ...
1699     Zimbabwe
1700     Zimbabwe
1701     Zimbabwe
```

```
1702     Zimbabwe
1703     Zimbabwe

[1704 rows x 1 columns]
```

Depending on what you need, sometimes you only need a single `Series` (sometimes called a vector), other times for consistency, you will want a `DataFrame` object.

1.3.1.2 Using Dot Notation to Pull a Column of Values

When all you need is a single column (i.e., `Series` or vector) of values and typing `df['column']` will be very tedious. There is a shorthand notation where you can pull the column vector by treating it as a `DataFrame` attribute.

For example, below are two ways of returning the same single column `Series`.

Click here to view code image

```
# using square bracket notation
print(df['country'])
```

```
0        Afghanistan
1        Afghanistan
2        Afghanistan
3        Afghanistan
4        Afghanistan
            ...
1699        Zimbabwe
1700        Zimbabwe
1701        Zimbabwe
1702        Zimbabwe
1703        Zimbabwe
Name: country, Length: 1704, dtype: object
```

```
# using dot notation
print(df.country)
```

```
0       Afghanistan
1       Afghanistan
2       Afghanistan
3       Afghanistan
4       Afghanistan
           ...
1699       Zimbabwe
1700       Zimbabwe
1701       Zimbabwe
1702       Zimbabwe
1703       Zimbabwe
Name: country, Length: 1704, dtype: object
```

There are subtle differences if you want to do other operations (e.g., deleting a column), but for now, you can treat those 2 ways of getting a single column of values as the same. You do have to be mindful of what your columns are named if you want to use the dot notation. That is, if there is a column named `shape`, the `df.shape` will return the number of rows and columns from the `.shape` attribute, not the intended shape column. Also, if your column name has spaces or special characters, you will not be able to use the dot notation to select that column of values, and will have to use the square bracket notation.

1.3.2 Subset Rows

Rows can be subset in multiple ways, by row name or row index. Table 1.2 gives a quick overview of the various methods.

Table 1.2 **Different Methods of Indexing Rows (and/or Columns)**[a]

Subset attribute	Description

Subset attribute	Description
`.loc[]`	Subset based on index label (row name)
`.iloc[]`	Subset based on row index (row number)
~~`.ix[]` (no longer works in `Pandas v0.20`)~~	~~Subset based on index label or row index~~

[a] Subsetting data with `.ix[]` is no longer supported in Pandas. The reason why `.ix[]` was removed is because it would first match on the index label, and if the value was not found, it would match on the index position. This dual subsetting behavior was not explicit and could be problematic since you did not always know how it was subsetting your rows.

1.3.2.1 Subset Rows by `index` Label - `.loc[]`

If we take a look at our gapminder data:

Click here to view code image

```
print(df)
```

```
         country continent  year  lifeExp
pop    gdpPercap
0     Afghanistan      Asia  1952   28.801
8425333   779.445314
1     Afghanistan      Asia  1957   30.332
9240934   820.853030
2     Afghanistan      Asia  1962   31.997
10267083   853.100710
3     Afghanistan      Asia  1967   34.020
11537966   836.197138
4     Afghanistan      Asia  1972   36.088
13079460   739.981106
...            ...         ...   ...       ...
```

```
...            ...
1699    Zimbabwe    Africa  1987   62.351
9216418   706.157306
1700    Zimbabwe    Africa  1992   60.377
10704340   693.420786
1701    Zimbabwe    Africa  1997   46.809
11404948   792.449960
1702    Zimbabwe    Africa  2002   39.989
11926563   672.038623
1703    Zimbabwe    Africa  2007   43.487
12311143   469.709298

[1704 rows x 6 columns]
```

We can see on the left side of the printed `DataFrame`, what appear to be row numbers. This column-less row of values is the "index" label of the `DataFrame`. Think of it like column names, but, for rows. By default, Pandas will fill in the index labels with the row numbers (note that it starts counting from `0`). A common example where the row index labels are not the row number is when we work with time series data. In that case, the index label will be a timestamp, but for now, we will keep the default row number values.

We can use the `.loc[]` accessor attribute on the `DataFrame` to subset rows based on the index label.

Click here to view code image

```
# get the first row
# python counts from 0
print(df.loc[0])
```

```
country       Afghanistan
continent            Asia
year                 1952
```

```
lifeExp              28.801
pop                 8425333
gdpPercap        779.445314
Name: 0, dtype: object
```

Click here to view code image

```
# get the 100th row
# python counts from 0
print(df.loc[99])
```

```
country      Bangladesh
continent          Asia
year               1967
lifeExp          43.453
pop            62821884
gdpPercap    721.186086
Name: 99, dtype: object
```

```
# get the last row
# this will cause an error
print(df.loc[-1])
```

```
KeyError: -1
```

Note that passing `-1` as the `.loc[]` will cause an error because it is actually looking for the row index label (i.e., row number) `-1`, which does not exist in our example `DataFrame`. Instead, we can use a bit of Python to calculate the total number of rows, and then pass *that* value into `.loc[]`.

Click here to view code image

```
# get the last row (correctly)

# use the first value given from shape to get
the number of rows
number_of_rows = df.shape[0]

# subtract 1 from the value since we want the
last index value
last_row_index = number_of_rows - 1

# finally do the subset using the index of the
last row
print(df.loc[last_row_index])
```

```
country         Zimbabwe
continent         Africa
year                2007
lifeExp           43.487
pop             12311143
gdpPercap     469.709298
Name: 1703, dtype: object
```

Or use the `.tail()` method to return the last `n=1` row, instead of the default `5`.

Click here to view code image

```
# there are many ways of doing what you want
print(df.tail(n=1))
```

```
       country continent  year  lifeExp       pop
gdpPercap
1703  Zimbabwe    Africa  2007   43.487  12311143
469.709298
```

Notice that using `.tail()` and `.loc[]` printed out the results differently. Let's look at what type is returned when we use these methods.

Click here to view code image

```
# get the last row of data in different ways
subset_loc = df.loc[0]
subset_head = df.head(n=1)
```

```
# type using loc of 1 row
print(type(subset_loc))
```

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

```
# type of using head of 1 row
print(type(subset_head))
```

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

At the beginning of this chapter, we mentioned that Pandas introduces two new data types into Python: `Series` and `DataFrame`. Depending on which method we use and how many rows we return, Pandas will return a different object. The way an object gets printed to the screen can be an indicator of the type, but it's always best to use the `type()` function to be sure. We go into more detail about these objects in Chapter 2.

1.3.2.2 Subsetting Multiple Rows

As with columns, we can filter multiple rows.

Click here to view code image

```
print(df.loc[[0, 99, 999]])
```

```
         country continent   year   lifeExp
pop      gdpPercap
0    Afghanistan      Asia   1952    28.801
8425333     779.445314
99    Bangladesh      Asia   1967    43.453
62821884     721.186086
999     Mongolia      Asia   1967    51.253
1149500    1226.041130
```

1.3.3 Subset Rows by Row Number: `.iloc[]`

`.iloc[]` does the same thing as `.loc[]`, but is used to subset by the row index number. In our current example, `.iloc[]` and `.locp[]` will behave exactly the same way since the index labels are the row numbers. However, keep in mind that the index labels do not necessarily have to be row numbers.

Click here to view code image

```
# get the 2nd row
print(df.iloc[1])
```

```
country      Afghanistan
continent           Asia
year                1957
lifeExp           30.332
pop              9240934
gdpPercap      820.85303
Name: 1, dtype: object
```

```
## get the 100th row
print(df.iloc[99])
```

```
country         Bangladesh
continent             Asia
year                  1967
lifeExp             43.453
pop               62821884
gdpPercap       721.186086
Name: 99, dtype: object
```

Note that when we subset on `1`, we actually get the second row, rather than the first row. This follows Python's zero-indexed behavior, meaning that the first item of a container is index `0` (i.e., `0`th item of the container). More details about this kind of behavior are found in Appendix F, Appendix I, and Appendix M.

With `.iloc[]`, we can pass in the `-1` to get the last row — something we couldn't do with `.loc[]`.

Click here to view code image

```
# using -1 to get the last row
print(df.iloc[-1])
```

```
country           Zimbabwe
continent           Africa
year                  2007
lifeExp             43.487
pop               12311143
gdpPercap       469.709298
Name: 1703, dtype: object
```

Just as before, we can pass in a list of integers to get multiple rows.

Click here to view code image

```
## get the first, 100th, and 1000th row
print(df.iloc[[0, 99, 999]])
```

```
         country continent  year  lifeExp
pop      gdpPercap
0    Afghanistan      Asia  1952   28.801
8425333     779.445314
99    Bangladesh      Asia  1967   43.453
62821884     721.186086
999     Mongolia      Asia  1967   51.253
1149500    1226.041130
```

1.3.4 Mix It Up

We can use `.loc[]` and `.iloc[]` to obtain subsets rows, columns, or both. The general syntax for `.loc[]` and `.iloc[]` uses square brackets with a comma. The part to the left of the comma is the row values to subset; the part to the right of the comma is the column values to subset. That is, `df.loc[[rows], [columns]]` or `df.iloc[[rows], [columns]]`.

1.3.4.1 Selecting Columns

If we want to use these techniques to just subset columns, we must use Python's slicing syntax (Appendix I). We need to do this because if we are subsetting columns, we are getting all the rows for the specified column. So, we need a method to capture all the rows.

The Python slicing syntax uses a colon, `:`. If we have just a colon, it "slices" (i.e., gets) all the values in that axis. So, if we just want to get the first column using the `.loc[]` or `.iloc[]` syntax, we can write `df.loc[:, [columns]]` to subset the column(s).

Click here to view code image

```
# subset columns with loc
# note the position of the colon
# it is used to select all rows
```

```
subset = df.loc[:, ['year', 'pop']]
print(subset)
```

```
      year       pop
0     1952   8425333
1     1957   9240934
2     1962  10267083
3     1967  11537966
4     1972  13079460
...    ...       ...
1699  1987   9216418
1700  1992  10704340
1701  1997  11404948
1702  2002  11926563
1703  2007  12311143

[1704 rows x 2 columns]
```

```
# subset columns with iloc
# iloc will allow us to use integers
# -1 will select the last column
subset = df.iloc[:, [2, 4, -1]]
print(subset)
```

```
      year       pop   gdpPercap
0     1952   8425333  779.445314
1     1957   9240934  820.853030
2     1962  10267083  853.100710
3     1967  11537966  836.197138
4     1972  13079460  739.981106
...      ...         ...         ...

1699  1987   9216418  706.157306
```

```
1700  1992   10704340  693.420786
1701  1997   11404948  792.449960
1702  2002   11926563  672.038623
1703  2007   12311143  469.709298

[1704 rows x 3 columns]
```

We will get an error if we don’t specify `.loc[]` or `iloc[]` correctly.

Click here to view code image

```
# subset columns with loc
# but pass in integer values
# this will cause an error
subset = df.loc[:, [2, 4, -1]]
print(subset)
```

```
KeyError: "None of [Int64Index([2, 4, -1],
dtype='int64')]
are in the [columns]"
```

```
# subset columns with iloc
# but pass in index names
# this will cause an error
subset = df.iloc[:, ['year', 'pop']]
print(subset)
```

```
IndexError: .iloc requires numeric indexers, got
['year' 'pop']
```

1.3.4.2 Subsetting with `range()`

You can use the built-in `range()` function to create a range of values in Python. This way you can specify beginning and end values, and Python will automatically create a range of values in between. By default, every value between the beginning and the end (inclusive left, exclusive right; see Appendix I) will be created, unless you specify a step (Appendix I and Appendix M). In Python 3, the `range()` function returns a generator. A generator is like a single-use list; it disappears after you use it once. This is mainly to save system resources. See Appendix M for more information about generators.

We just saw in Section 1.3.4.1 how we can select columns using a list of integers. Since `range()` returns a generator, we have to first convert the generator to a list.

Click here to view code image

```
# create a range of integers from 0 - 4
inclusive
small_range = list(range(5))
print(small_range)
```

```
[0, 1, 2, 3, 4]
```

```
# subset the dataframe with the range
subset = df.iloc[:, small_range]
print(subset)
```

```
          country continent  year  lifeExp
pop
0     Afghanistan      Asia  1952   28.801
8425333
1     Afghanistan      Asia  1957   30.332
9240934
2     Afghanistan      Asia  1962   31.997
10267083
3     Afghanistan      Asia  1967   34.020
```

```
11537966
4    Afghanistan      Asia  1972   36.088
13079460
...           ...         ...   ...       ...
...
1699    Zimbabwe    Africa  1987   62.351
9216418
1700    Zimbabwe    Africa  1992   60.377
10704340
1701    Zimbabwe    Africa  1997   46.809
11404948
1702    Zimbabwe    Africa  2002   39.989
11926563
1703    Zimbabwe    Africa  2007   43.487
12311143

[1704 rows x 5 columns]
```

Note that when `list(range(5))` is called, five integers are returned: `0` – `4`.

Click here to view code image

```
# create a range from 3 - 5 inclusive
small_range = list(range(3, 6))
print(small_range)
```

```
[3, 4, 5]
```

```
subset = df.iloc[:, small_range]
print(subset)
```

```
      lifeExp       pop   gdpPercap
0      28.801   8425333  779.445314
```

```
1       30.332   9240934  820.853030
2       31.997  10267083  853.100710
3       34.020  11537966  836.197138
4       36.088  13079460  739.981106
...        ...       ...         ...
1699    62.351   9216418  706.157306
1700    60.377  10704340  693.420786
1701    46.809  11404948  792.449960
1702    39.989  11926563  672.038623
1703    43.487  12311143  469.709298

[1704 rows x 3 columns]
```

> **Question**
>
> What happens when you specify a `range()` that's beyond the number of columns you have?

Again, note that the values are specified in a way such that the range is inclusive on the left, and exclusive on the right.

We can also pass in a 3rd parameter into range, `step`, that allows us to change how to increment between the start and stop values (defaults to `step=1`).

Click here to view code image

```
# create a range from 0 - 5 inclusive, every
other integer
small_range = list(range(0, 6, 2))
subset = df.iloc[:, small_range]
print(subset)
```

```
        country   year       pop
0     Afghanistan  1952   8425333
```

```
1       Afghanistan  1957   9240934
2       Afghanistan  1962  10267083
3       Afghanistan  1967  11537966
4       Afghanistan  1972  13079460
...             ...   ...       ...
1699       Zimbabwe  1987   9216418
1700       Zimbabwe  1992  10704340
1701       Zimbabwe  1997  11404948
1702       Zimbabwe  2002  11926563
1703       Zimbabwe  2007  12311143

[1704 rows x 3 columns]
```

Converting a generator to a list is a bit awkward; we can use the Python slicing syntax to fix this.

1.3.4.3 Subsetting with Slicing `:`

Python's slicing syntax, `:`, is similar to the `range()` function. Instead of a function that specifies `start`, `stop`, and `step` values delimited by a comma, we separate the values with the colon, `:`.

If you understand what was going on with the `range()` function earlier, then slicing can be seen as a shorthand for the same thing.

The `range()` function can be used to create a generator that can also be converted to a list of values. The colon syntax, `:`, only has meaning within the square bracket, `[ ]` slicing and subsetting context; it has no inherent meaning on its own.

Here are the columns of our data set.

Click here to view code image

```
print(df.columns)
```

```
Index(['country', 'continent', 'year', 'lifeExp',
'pop',
```

```
       'gdpPercap'],
     dtype='object')
```

See how `range()` and `:` are used to slice our data.

Click here to view code image

```
small_range = list(range(3))
subset = df.iloc[:, small_range]
print(subset)
```

```
         country continent  year
0    Afghanistan      Asia  1952
1    Afghanistan      Asia  1957
2    Afghanistan      Asia  1962
3    Afghanistan      Asia  1967
4    Afghanistan      Asia  1972
...          ...       ...   ...
1699    Zimbabwe    Africa  1987
1700    Zimbabwe    Africa  1992
1701    Zimbabwe    Africa  1997
1702    Zimbabwe    Africa  2002
1703    Zimbabwe    Africa  2007

[1704 rows x 3 columns]
```

```
# slice the first 3 columns
subset = df.iloc[:, :3]
print(subset)
```

```
         country continent  year
0    Afghanistan      Asia  1952
```

```
1    Afghanistan      Asia  1957
2    Afghanistan      Asia  1962
3    Afghanistan      Asia  1967
4    Afghanistan      Asia  1972
...          ...       ...   ...
1699    Zimbabwe    Africa  1987
1700    Zimbabwe    Africa  1992
1701    Zimbabwe    Africa  1997
1702    Zimbabwe    Africa  2002
1703    Zimbabwe    Africa  2007

[1704 rows x 3 columns]
```

```
small_range = list(range(3, 6))
subset = df.iloc[:, small_range]
print(subset)
```

```
      lifeExp       pop   gdpPercap
0      28.801   8425333  779.445314
1      30.332   9240934  820.853030
2      31.997  10267083  853.100710
3      34.020  11537966  836.197138
4      36.088  13079460  739.981106
...       ...       ...         ...
1699   62.351   9216418  706.157306
1700   60.377  10704340  693.420786
1701   46.809  11404948  792.449960
1702   39.989  11926563  672.038623

1703 43.487 12311143 469.709298

[1704 rows x 3 columns]
```

```python
# slice columns 3 to 5 inclusive
subset = df.iloc[:, 3:6]
print(subset)
```

```
     lifeExp       pop   gdpPercap
0     28.801   8425333  779.445314
1     30.332   9240934  820.853030
2     31.997  10267083  853.100710
3     34.020  11537966  836.197138
4     36.088  13079460  739.981106
...      ...       ...         ...
1699  62.351   9216418  706.157306
1700  60.377  10704340  693.420786
1701  46.809  11404948  792.449960
1702  39.989  11926563  672.038623
1703  43.487  12311143  469.709298

[1704 rows x 3 columns]
```

```python
small_range = list(range(0, 6, 2))
subset = df.iloc[:, small_range]
print(subset)
```

```
         country  year       pop
0    Afghanistan  1952   8425333
1    Afghanistan  1957   9240934
2    Afghanistan  1962  10267083
3    Afghanistan  1967  11537966
4    Afghanistan  1972  13079460
...          ...   ...       ...
1699    Zimbabwe  1987   9216418
1700    Zimbabwe  1992  10704340
```

```
1701     Zimbabwe    1997  11404948
1702     Zimbabwe    2002  11926563
1703     Zimbabwe    2007  12311143

[1704 rows x 3 columns]
```

```python
# slice every other columns
subset = df.iloc[:, 0:6:2]
print(subset)
```

```
          country    year       pop
0     Afghanistan    1952   8425333
1     Afghanistan    1957   9240934
2     Afghanistan    1962  10267083
3     Afghanistan    1967  11537966
4     Afghanistan    1972  13079460
...           ...     ...       ...
1699     Zimbabwe    1987   9216418
1700     Zimbabwe    1992  10704340
1701     Zimbabwe    1997  11404948
1702     Zimbabwe    2002  11926563
1703     Zimbabwe    2007  12311143

[1704 rows x 3 columns]
```

Question

What happens if you use the slicing method with 2 colons, but leave a value out? For example:

- `df.iloc[:, 0:6:]`
- `df.iloc[:, 0::2]`

- `df.iloc[:, :6:2]`
- `df.iloc[:, ::2]`
- `df.iloc[:, ::]`

1.3.5 Subsetting Rows and Columns

When only using the colon, `:`, in `.loc[]` and `.iloc[]` to the left of the comma, we select all the rows in our dataframe (i.e., we slice all the values in the first axis of our `DataFrame`). However, we can choose to put values to the left of the comma if we want to select specific rows along with specific columns.

```
# using loc
print(df.loc[42, 'country'])
```

```
Angola
```

```
# using iloc
print(df.iloc[42, 0])
```

```
Angola
```

Just make sure you don't confuse the differences between `.loc[]` and `.iloc[]`.

```
# will cause an error
print(df.loc[42, 0])
```

```
KeyError: 0
```

1.3.5.1 Subsetting Multiple Rows and Columns

We can combine the row and column subsetting syntax with the multiple-row and multiple-column subsetting syntax to get various slices of our data.

Click here to view code image

```
# get the 1st, 100th, and 1000th rows
# from the 1st, 4th, and 6th column
# note the columns we are hoping to get are:
# country, lifeExp, and gdpPercap
print(df.iloc[[0, 99, 999], [0, 3, 5]])
```

```
        country  lifeExp    gdpPercap
0   Afghanistan   28.801   779.445314
99   Bangladesh   43.453   721.186086
999    Mongolia   51.253  1226.041130
```

In my own work, I try to pass in the actual column names when subsetting data whenever possible (i.e., I try to use `.loc[]` as much as I can). That approach makes the code more readable since you do not need to look at the column name vector to know which index is being called. Additionally, using absolute indexes can lead to problems if the column order gets changed. This is just a general rule of thumb, as there will be exceptions where using the index position is a better option (e.g., concatenating data in Chapter 6).

Click here to view code image

```
# if we use the column names directly,
# it makes the code a bit easier to read
# note now we have to use loc, instead of iloc
print(df.loc[[0, 99, 999], ['country',
'lifeExp', 'gdpPercap']])
```

```
        country  lifeExp   gdpPercap
0   Afghanistan   28.801  779.445314
```

```
99     Bangladesh    43.453   721.186086
999      Mongolia    51.253  1226.041130
```

Important

Remember, you can use the slicing syntax on the row portion of the `.loc[]` and `.iloc[]` attributes. Pay attention to the differences in how those two attributes select values: `.loc[]` matches on the named value, and `.iloc[]` slices by position.

The results below are slightly different for the very reason.

Click here to view code image

```
print(df.loc[10:13, :])
```

```
         country continent  year  lifeExp
pop    gdpPercap
10   Afghanistan      Asia  2002   42.129
25268405   726.734055
11   Afghanistan      Asia  2007   43.828
31889923   974.580338
12       Albania    Europe  1952   55.230
1282697  1601.056136
13       Albania    Europe  1957   59.280
1476505  1942.284244
```

```
print(df.iloc[10:13, :])
```

```
        country continent  year  lifeExp
pop      gdpPercap
10  Afghanistan      Asia  2002   42.129
25268405    726.734055
11  Afghanistan      Asia  2007   43.828
31889923    974.580338
```

```
12       Albania    Europe  1952   55.230
1282697   1601.056136
```

More detail about how slicing works in Python is described in Appendix I.

1.4 Grouped and Aggregated Calculations

If you've worked with other Python libraries or programming languages, you know that many basic statistical calculations either come with the library or are built into the language. Let's look at our Gapminder data again.

Click here to view code image

```
print(df)
```

```
         country continent  year  lifeExp
pop   gdpPercap
0     Afghanistan      Asia  1952   28.801
8425333  779.445314
1     Afghanistan      Asia  1957   30.332
9240934  820.853030
2     Afghanistan      Asia  1962   31.997
10267083  853.100710
3     Afghanistan      Asia  1967   34.020
11537966  836.197138
4     Afghanistan      Asia  1972   36.088
13079460  739.981106
...           ...       ...   ...      ...
...          ...
1699     Zimbabwe    Africa  1987   62.351
9216418  706.157306
1700     Zimbabwe    Africa  1992   60.377
```

```
10704340  693.420786
1701     Zimbabwe    Africa  1997   46.809
11404948  792.449960
1702     Zimbabwe    Africa  2002   39.989
11926563  672.038623
1703     Zimbabwe    Africa  2007   43.487
12311143  469.709298

[1704 rows x 6 columns]
```

There are several initial questions that we can ask ourselves:

- For each year in our data, what was the average life expectancy? What is the average life expectancy, population, and GDP?
- What if we stratify the data by continent and perform the same calculations?
- How many countries are listed in each continent?

1.4.1 Grouped Means

To answer the questions just posed, we need to perform a grouped (i.e., aggregate) calculation. In other words, we need to perform a calculation, be it an average or a frequency count, but apply it to each subset of a variable. Another way to think about grouped calculations is as a split–apply–combine process. We first split our data into various parts, then apply a function (or calculation) of our choosing to each of the split parts, and finally combine all the individual split calculations into a single dataframe. We accomplish grouped (i.e., aggregate) computations by using the `.groupby()` method on `DataFrames`. Grouped calculations are further discussed in Chapter 8.

Click here to view code image

```
# For each year in our data, what was the
average life expectancy?
```

```
# To answer this question, we need to:
# 1. split our data into parts by year
# 2. get the 'lifeExp' column
# 3. calculate the mean
print(df.groupby('year')['lifeExp'].mean())
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
          ...
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, Length: 12, dtype: float64
```

Let's unpack the statement we used in this example. We first create a grouped object.

Click here to view code image

```
# create grouped object by year
grouped_year_df = df.groupby('year')
print(type(grouped_year_df))
```

```
<class
'pandas.core.groupby.generic.DataFrameGroupBy'>
```

If we printed the grouped `DataFrame` Pandas would return only the memory location.

Click here to view code image

```
print(grouped_year_df)
```

```
<pandas.core.groupby.generic.DataFrameGroupBy
object at 0x15fdb7df0>
```

From the grouped data, we can subset the columns of interest on which we want to perform our calculations. To our question, `lifeExp` column. We can use the subsetting methods described in Section 1.3.1.

Click here to view code image

```
grouped_year_df_lifeExp =
grouped_year_df['lifeExp']
print(type(grouped_year_df_lifeExp))
```

```
<class
'pandas.core.groupby.generic.SeriesGroupBy'>
```

```
print(grouped_year_df_lifeExp)
```

```
<pandas.core.groupby.generic.SeriesGroupBy object
at 0x106c55ae0>
```

Notice that we now are given a series (because we asked for only one column) and the contents of the series are grouped (in our example by year).

Finally, we know the `lifeExp` column is of type `float64`. An operation we can perform on a vector of numbers is to calculate the mean to get our final desired result.

Click here to view code image

```
mean_lifeExp_by_year =
grouped_year_df_lifeExp.mean()
```

```
print(mean_lifeExp_by_year)
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
          ...
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, Length: 12, dtype: float64
```

We can perform a similar set of calculations for the population and GDP since they are of types `int64` and `float64`, respectively. But what if we want to group and stratify the data by more than one variable? And what if we want to perform the same calculation on multiple columns? We can build on the material earlier in this chapter by using a list!

Click here to view code image

```
# the backslash allows us to break up 1 long
line of python code
# into multiple lines
# df.groupby(['year', 'continent'])[['lifeExp',
'gdpPercap']].mean()
# is the same as
multi_group_var = df\
  .groupby(['year', 'continent'])\
  [['lifeExp', 'gdpPercap']]\
  .mean()
```

```
# look at the first 10 rows
print(multi_group_var)
```

```
                  lifeExp     gdpPercap
year continent
1952 Africa      39.135500    1252.572466
     Americas    53.279840   4079.062552
     Asia        46.314394   5195.484004
     Europe      64.408500   5661.057435
     Oceania     69.255000  10298.085650
...                    ...           ...
2007 Africa      54.806038   3089.032605


     Americas    73.608120  11003.031625
     Asia        70.728485  12473.026870
     Europe      77.648600  25054.481636
     Oceania     80.719500  29810.188275

[60 rows x 2 columns]
```

We can also use round parentheses, `( )` for "method chaining" (more about this notation in Appendix D.1).

Click here to view code image

```
# we can also wrap the entire statement
# around round parentheses
# with each .method() on a new line
# this is the preferred style for writing
"method chaining"
multi_group_var = (
  df
  .groupby(['year', 'continent'])
```

```
    [['lifeExp', 'gdpPercap']]
    .mean()
)
```

The output data is grouped by year and continent. For each year–continent pair, we calculated the average life expectancy and average GDP. The data is also printed out a little differently. Notice the year and continent column names are not on the same line as the life expectancy and GPD column names. There is some hierarchal structure between the year and continent row indices. We'll discuss working with these types of data in more detail in Section 8.5.

If you need to "flatten" the `DataFrame`, you can use the `.reset_index()` method.

Click here to view code image

```
flat = multi_group_var.reset_index()
print(flat)
```

```
    year continent    lifeExp     gdpPercap
0   1952    Africa  39.135500   1252.572466
1   1952  Americas  53.279840   4079.062552
2   1952      Asia  46.314394   5195.484004
3   1952    Europe  64.408500   5661.057435
4   1952   Oceania  69.255000  10298.085650
..   ...       ...        ...           ...
55  2007    Africa  54.806038   3089.032605
56  2007  Americas  73.608120  11003.031625
57  2007      Asia  70.728485  12473.026870
58  2007    Europe  77.648600  25054.481636
59  2007   Oceania  80.719500  29810.188275

[60 rows x 4 columns]
```

Question

Does the order of the list we used to group the data matter?

1.4.2 Grouped Frequency Counts

Another common data-related task is to calculate frequencies. We can use the `.nunique()` and `.value_counts()` methods, respectively, to get counts of unique values and frequency counts on a Pandas `Series`.

Click here to view code image

```
# use the nunique (number unique)
# to calculate the number of unique values in a
series
print(df.groupby('continent')
['country'].nunique())
```

```
continent
Africa      52
Americas    25
Asia        33
Europe      30
Oceania      2
Name: country, dtype: int64
```

Question

What do you get if you use `.value_counts()` instead of `.nunique()`?

1.5 Basic Plot

Visualizations are extremely important in almost every step of the data process. They help us identify trends in data when we are trying to understand and clean the data, and they help us convey our final findings. More information about visualization and plotting is described in Chapter 3.

Let's look at the yearly life expectancies for the world population again.

Click here to view code image

```
global_yearly_life_expectancy =
df.groupby('year')['lifeExp'].mean()
print(global_yearly_life_expectancy)
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
          ...
1987    63.212613
1992    64.160338

1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, Length: 12, dtype: float64
```

We can use Pandas to create some basic plots as shown in Figure 1.1. More about plotting is covered in Chapter 3.

Click here to view code image

```
# matplotlib is the default plotting library
# we need to import first
import matplotlib.pyplot as plt

# use the .plot() DataFrame method
global_yearly_life_expectancy.plot()

# show the plot
plt.show()
```

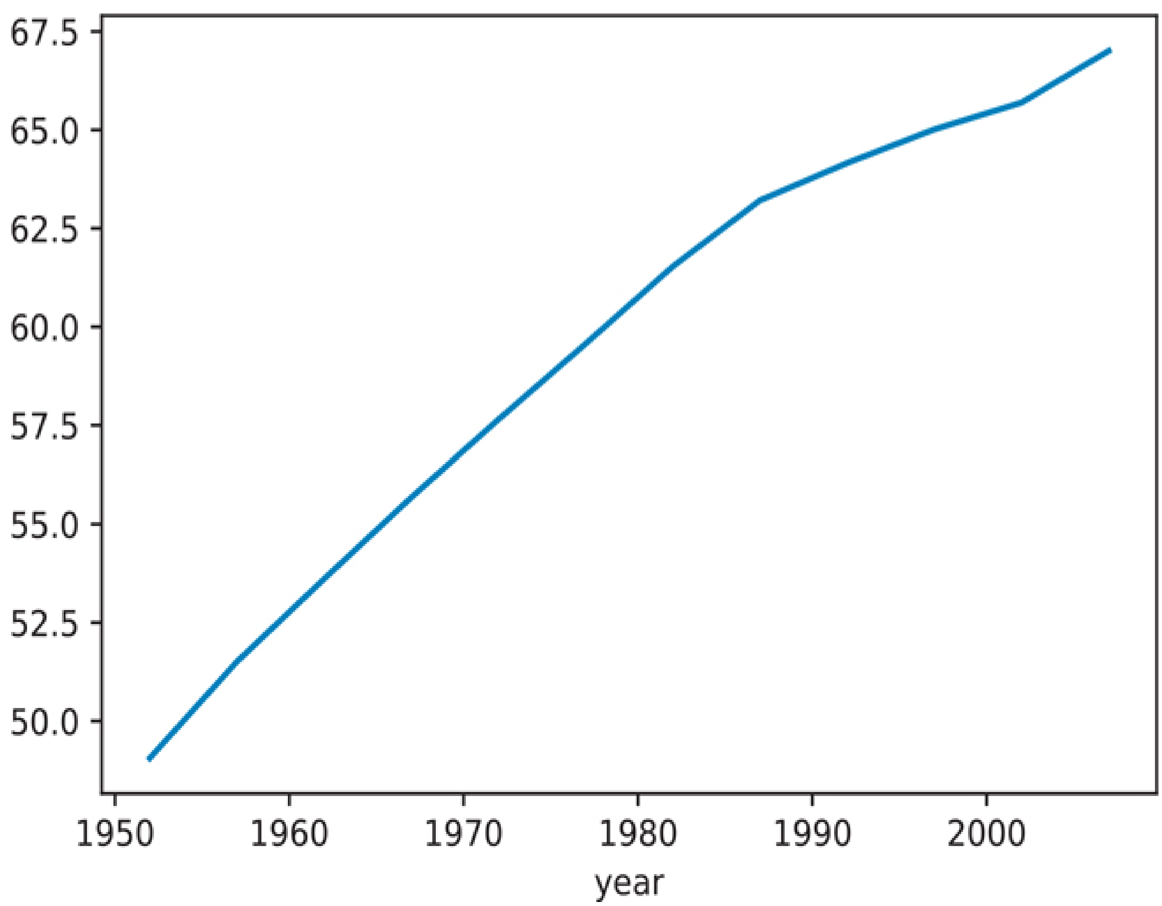


Figure 1.1 Basic plot in Pandas showing average life expectancy over time

Conclusion

This chapter explained how to load up a simple data set and start looking at specific observations. It may seem tedious at first to look at observations this way, especially if you are already familiar with the use of a spreadsheet program. Keep in mind that when doing data analytics, the goal is to produce reproducible results, not repeat repetitive tasks, and be able to combine multiple data sources as needed. Scripting languages give you that ability and flexibility.

Along the way, you learned about some of the fundamental programming abilities and data structures that Python has to offer. You also encountered a quick way to obtain aggregated statistics and plots. The next chapter goes into more detail about the Pandas `DataFrame` and `Series` objects, as well as other ways you can subset and visualize your data.

As you work your way through this book, if there is a concept or data structure that is foreign to you, check the various appendices for more information. Many fundamental programming features of Python are covered in the appendices.

2

Pandas Data Structures Basics

Chapter 1 introduced the Pandas `DataFrame` and `Series` objects. These data structures resemble the primitive Python data containers (lists and dictionaries) for indexing and labeling, but have additional features that make working with data easier.

Learning Objectives

The concept map for this chapter can be found in Figure A.2.

- Use functions to create and load manual data
- Describe the `Series` object
- Describe the `DataFrame` object
- Identify basic operations on `Series` objects
- Identify basic operations on `DataFrame` objects
- Perform conditional subsetting, fancy slicing, and indexing
- Use methods to save data

2.1 Create Your Own Data

Whether you are manually inputting data or creating a small test example, knowing how to create `DataFrames` without loading data from a file is a useful skill. It is especially helpful when you are asking a question about a StackOverflow error.

2.1.1 Create a Series

The Pandas `Series` is a one-dimensional container (i.e., Python `Iterable`), similar to the built-in Python `list`. It is the data type that represents each column of the `DataFrame`. Table 1.1 lists the possible `dtypes` for Pandas `DataFrame` columns. Each value in a `DataFrame` column must be stored as the same `dtype`. For example, if a column contains the number `1` and the sequence of letters (i.e., string) `"pizza"`, the entire `dtype` of the column will be a string (Pandas will call this an `object dtype`).

Since a `DataFrame` can be thought of as a dictionary of `Series` objects, where each `key` is the column name and the `value` is the `Series`, we can conclude that a `Series` is very similar to a Python `list`, except that each element must be the same `dtype`. Those who have used the `numpy` library will realize this is the same behavior as demonstrated by the `ndarray`.

The easiest way to create a `Series` is to pass in a Python `list`. If we pass in a list of mixed types, the most common representation of both will be used. Typically the `dtype` will be `object`.

Click here to view code image

```
import pandas as pd

s = pd.Series(['banana', 42])
print(s)
```

```
0    banana
1        42
dtype: object
```

Notice that the "row number" is shown on the left of the `Series`. This is actually the `index` for the series. It is similar to the row name and row index we saw in Section 1.3.2 for `DataFrames`. It implies that we can actually assign a "name" to values in our series.

Click here to view code image

```
# manually assign index values to a series
# by passing a Python list
s = pd.Series(
data=["Wes McKinney", "Creator of Pandas"],
index=["Person", "Who"],
)

print(s)
```

```
Person           Wes McKinney
Who         Creator of Pandas
dtype: object
```

Question

- What happens if you use other Python containers such as `list`, `tuple`, `dict`, or even the `ndarray` from the `numpy` library?
- What happens if you pass an `index` along with the containers?
- Does passing in an `index` when you use a `dict` overwrite the index? Or does it sort the values?

2.1.2 Create a DataFrame

As mentioned in Chapter 1, a `DataFrame` can be thought of as a dictionary of `Series` objects. This is why dictionaries are the most common way of creating a `DataFrame`. The `key` represents the column name, and the `values` are the contents of the column.

Click here to view code image

```
scientists = pd.DataFrame(
  {
    "Name": ["Rosaline Franklin", "William
```

```
Gosset"],
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
    "Age": [37, 61],
  }
)

print(scientists)
```

```
                Name    Occupation        Born
Died  Age
0  Rosaline Franklin       Chemist  1920-07-25
1958-04-16   37
1     William Gosset  Statistician  1876-06-13
1937-10-16   61
```

If we look at the documentation for `DataFrame`[1], we see that we can use the `columns` parameter or specify the column order. If we want to use the `name` column for the row index, we can use the `index` parameter.

1. `DataFrame` documentation: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

Click here to view code image

```
scientists = pd.DataFrame(
  data={
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
    "Age": [37, 61],
  },
  index=["Rosaline Franklin", "William Gosset"],
```

```
    columns=["Occupation", "Born", "Died", "Age"],
)
print(scientists)
```

```
                        Occupation        Born
Died   Age
Rosaline Franklin      Chemist  1920-07-25  1958-
04-16   37
William Gosset    Statistician  1876-06-13  1937-
10-16   61
```

2.2 The Series

In Section 1.3.2.1, we saw how the slicing method affects the `type` of the result. If we use `.loc[]` to subset the first row of our `scientists` `DataFrame`, we will get a `Series` object back.

First, let's re-create our example `DataFrame`.

Click here to view code image

```
# create our example dataframe
# with a row index label
scientists = pd.DataFrame(

  data={
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
    "Age": [37, 61],
  },
  index=["Rosaline Franklin", "William Gosset"],
  columns=["Occupation", "Born", "Died", "Age"],
)
```

```
print(scientists)
```

```
                    Occupation        Born
Died   Age
Rosaline Franklin      Chemist  1920-07-25  1958-
04-16   37
William Gosset    Statistician  1876-06-13  1937-
10-16   61
```

Select a scientist by the row index label.

Click here to view code image

```
# select by row index label
first_row = scientists.loc['William Gosset']
print(type(first_row))
```

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

```
print(first_row)
```

```
Occupation    Statistician
Born            1876-06-13
Died            1937-10-16
Age                     61
Name: William Gosset, dtype: object
```

When a series is printed (i.e., the string representation), the index is printed as the first "column," and the values are printed as the second "column." There are many attributes and methods associated with a `Series` object.[2] Two examples of attributes are `.index` and `.values`.

2. `Series` documentation: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html

Click here to view code image

```
print(first_row.index)
```

```
Index(['Occupation', 'Born', 'Died', 'Age'],
dtype='object')
```

```
print(first_row.values)
```

```
['Statistician' '1876-06-13' '1937-10-16' 61]
```

Table 2.1 Some of the Attributes Within a Series

Series attributes	**Description**
.loc	Subset using index value
.iloc	Subset using index position
.dtype or dtypes	The type of the `Series` contents
.T	Transpose of the series
.shape	Dimensions of the data
.size	Number of elements in the `Series`
.values	`ndarray` or `ndarray`-like of the `Series`

An example of a `Series` method is `.keys()`, which is an alias for the `.index` attribute.

Click here to view code image

```
print(first_row.keys())
```

```
Index(['Occupation', 'Born', 'Died', 'Age'],
dtype='object')
```

By now, you might have questions about the syntax for `.index`, `.values`, and `.keys()`. More information about attributes and methods is found in Appendix P on classes. Attributes can be thought of as features of an object (in this example, our object is a `Series`). Methods can be thought of as some calculation or operation that is performed on an object. The subsetting syntax for `.loc[]` and `.iloc[]` (from Section 1.3.2) consists of all attributes. This is why the syntax does not rely on a set of round parentheses, `( )`, but rather a set of square brackets, `[ ]`, for subsetting. Since `.keys()` is a method, if we wanted to get the first key (which is also the first index), we would use the square brackets *after* the method call. Some attributes for the series are listed in Table 2.1.

Click here to view code image

```
# get the first index using an attribute
print(first_row.index[0])
```

```
Occupation
```

```
# get the first index using a method
print(first_row.keys()[0])
```

```
Occupation
```

2.2.1 The Series Is ndarray-like

The Pandas data structure known as `Series` is very similar to the `numpy.ndarray` (Appendix O). In turn, many methods and functions that operate on a `ndarray` will also operate on a `Series`. A `Series` may sometimes be referred to as a "vector."

2.2.1.1 Series Methods

Let's first get a series of the `Age` column from our `scientists` dataframe.

```
# get the 'Age' column
ages = scientists['Age']
print(ages)
```

```
Rosaline Franklin    37
William Gosset       61
Name: Age, dtype: int64
```

NumPy is a scientific computing library that typically deals with numeric vectors. Since a `Series` can be thought of as an extension to the `numpy.ndarray`, there is an overlap of attributes and methods. When we have a vector of numbers, there are common calculations we can perform.[3]

3. Descriptive statistics: https://pandas.pydata.org/pandas-docs/stable/user_guide/basics.html#descriptive-statistics

Click here to view code image

```
# calculate the mean
print(ages.mean())
```

```
49.0
```

```
# calculate the minimum
print(ages.min())
```

```
37
```

```
# calculate the maximum
print(ages.max())
```

```
61
```

```
# calculate the standard deviation
print(ages.std())
```

```
16.97056274847714
```

The `.mean()`, `.min()`, `.max()`, and `.std()` are also methods in the `numpy.ndarray`.[4] Some `Series` methods are listed in Table 2.2.

4. NumPy `ndarrary` documentation:
https://numpy.org/doc/stable/reference/arrays.ndarray.html

2.2.2 Boolean Subsetting: Series

Chapter 1 showed how we can use specific indices to subset our data. Only rarely, however, will we know the exact row or column index to subset the data. Typically you are looking for values that meet (or don't meet) a particular calculation or observation.

To explore this process, let's use a larger data set.

Click here to view code image

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

We just saw how we can calculate basic descriptive metrics of vectors. The `.describe()` method will calculate multiple descriptive statistics in a single method call.

Click here to view code image

```
        ages = scientists['Age']
        print(ages)
```

Table 2.2 Some of the Methods That Can Be Performed on a Series

Series Methods	Description

Series Methods	Description
.append()	Concatenates two or more Series
.corr()	Calculate a correlation with another Series[5]
.cov()	Calculate a covariance with another Series[6]
.describe()	Calculate summary statistics[7]
.drop_duplicates()	Returns a Series without duplicates
.equals()	Determines whether a Series has the same elements
.get_values()	Get values of the Series; same as the values attribute
.hist()	Draw a histogram
.isin()	Checks whether values are contained in a Series
.min()	Returns the minimum value
.max()	Returns the maximum value
.mean()	Returns the arithmetic mean
.median()	Returns the median
.mode()	Returns the mode(s)
.quantile()	Returns the value at a given quantile
.replace()	Replaces values in the Series with a specified value
.sample()	Returns a random sample of values from the Series
.sort_values()	Sorts values
.to_frame()	Converts a Series to a DataFrame
.transpose()	Returns the transpose
.unique()	Returns a numpy.ndarray of unique values

5. Missing values will be automatically dropped.

6. Missing values will be automatically dropped.

7. Missing values will be automatically dropped.

```
0    37
1    61
2    90
3    66
4    56
5    45
6    41
7    77
Name: Age, dtype: int64
```

```
# get basic stats
print(ages.describe())
```

```
count     8.000000
mean     59.125000
std      18.325918

min      37.000000
25%      44.000000
50%      58.500000
75%      68.750000
max      90.000000
Name: Age, dtype: float64
```

```
# mean of all ages
print(ages.mean())
```

```
59.125
```

What if we wanted to subset our ages by identifying those above the mean?

Click here to view code image

```
print(ages[ages > ages.mean()])
```

```
1    61
2    90
3    66
7    77
Name: Age, dtype: int64
```

Let's tease out this statement and look at what `ages > ages.mean()` returns.

Click here to view code image

```
print(ages > ages.mean())
```

```
0    False
1     True
2     True
3     True
4    False
5    False
6    False
7     True
Name: Age, dtype: bool
```

```
print(type(ages > ages.mean()))
```

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

This statement returns a `Series` with a `.dtype` of `bool`. In other words, we can not only subset values using labels and indices, but also supply a vector of boolean values. Python has many functions and methods. Depending on how they are implemented, they may return labels, indices, or booleans. Keep this point in mind as you learn new methods and seek to piece together various parts for your work.

If we liked, we could manually supply a vector of `bool`s to subset our data.

Click here to view code image

```
# get index 0, 1, 4, 5, and 7
manual_bool_values = [

  True,  # 0
  True,  # 1
  False, # 2
  False, # 3
  True,  # 4
  True,  # 5
  False, # 6
  True,  # 7
]
print(ages[manual_bool_values])
```

```
0    37
1    61
4    56
5    45
7    77
Name: Age, dtype: int64
```

2.2.3 Operations Are Automatically Aligned and Vectorized (Broadcasting)

If you're familiar with programming, you would find it strange that `ages > ages.mean()` returns a vector without any `for` loops (Appendix J). Many of the methods that work on `Series` (and also `DataFrames`) are "vectorized," meaning that they work on the entire vector simultaneously. This approach makes the code easier to read, and typically, optimizations are available to make calculations faster.

2.2.3.1 Vectors of the Same Length

If you perform an operation between two vectors of the same length, the resulting vector will be an element-by-element calculation of the vectors.

```
print(ages + ages)
```

```
0     74
1    122
2    180
3    132
4    112
5     90
6     82
7    154
Name: Age, dtype: int64
```

```
print(ages * ages)
```

```
0    1369
1    3721
2    8100
3    4356
```

```
4    3136
5    2025
6    1681
7    5929
Name: Age, dtype: int64
```

2.2.3.2 Vectors With Integers (Scalars)

When you perform an operation on a vector using a scalar, the scalar will be recycled across all the elements in the vector.

```
print(ages + 100)
```

```
0    137
1    161
2    190
3    166
4    156
5    145
6    141
7    177
Name: Age, dtype: int64
```

```
print(ages * 2)
```

```
0     74
1    122
2    180
3    132
4    112
5     90
6     82
```

```
7    154
Name: Age, dtype: int64
```

2.2.3.3 Vectors With Different Lengths

When you are working with vectors of different lengths, the behavior will depend on the `type()` of the vectors. With a `Series`, the vectors will perform an operation matched by the index. The rest of the resulting vector will be filled with a "missing" value, denoted with `NaN`, signifying "not a number" (Chapter 9).

This type of behavior, which is called broadcasting, differs between languages. Broadcasting in Pandas refers to how operations are calculated between arrays with different shapes.

Click here to view code image

```
print(ages + pd.Series([1, 100]))
```

```
0     38.0
1    161.0
2      NaN
3      NaN

4      NaN
5      NaN
6      NaN
7      NaN
dtype: float64
```

With other `types()`, the shapes must match.

Click here to view code image

```
import numpy as np
```

```
# this will cause an error
print(ages + np.array([1, 100]))
```

```
ValueError: operands could not be broadcast
together with shapes (8,) (2,)
```

2.2.3.4 Vectors With Common Index Labels (Automatic Alignment)

What's convenient in Pandas is how data alignment is almost always automatic. If possible, things will always align themselves with the index label when actions are performed.

Click here to view code image

```
# ages as they appear in the data
print(ages)
```

```
0    37
1    61
2    90
3    66
4    56
5    45
6    41
7    77
Name: Age, dtype: int64
```

```
rev_ages = ages.sort_index(ascending=False)
print(rev_ages)
```

```
7    77
6    41
5    45
```

```
4    56
3    66
2    90
1    61
0    37
Name: Age, dtype: int64
```

If we perform an operation using `ages` and `rev_ages`, it will still be conducted on an element-by-element basis, but the vectors will be aligned first before the operation is carried out.

Click here to view code image

```
# reference output to show index label alignment
print(ages * 2)
```

```
0     74
1    122
2    180
3    132
4    112
5     90
6     82
7    154
Name: Age, dtype: int64
```

```
# note how we get the same values
# even though the vector is reversed
print(ages + rev_ages)
```

```
0     74
1    122
2    180
3    132
```

```
4    112
5     90
6     82
7    154
Name: Age, dtype: int64
```

2.3 The DataFrame

The `DataFrame` is the most common Pandas object. It can be thought of as Python's way of storing spreadsheet-like data. Many of the features of the `Series` data structure carry over into the `DataFrame`.

2.3.1 Parts of a DataFrame

There are 3 main parts to a Pandas `DataFrame` object the `.index`, `.columns`, and `.values`. These refer to the row name, column names, and data values, respectively.

Click here to view code image

```
scientists.index
```

```
RangeIndex(start=0, stop=8, step=1)
```

```
scientists.columns
```

```
Index(['Name', 'Born', 'Died', 'Age',
'Occupation'], dtype='object')
```

```
scientists.values
```

```
array([['Rosaline Franklin', '1920-07-25', '1958-
04-16', 37, 'Chemist'],
```

```
       ['William Gosset', '1876-06-13', '1937-10-
16', 61, 'Statistician'],
       ['Florence Nightingale', '1820-05-12',
'1910-08-13', 90, 'Nurse'],
       ['Marie Curie', '1867-11-07', '1934-07-04',
66, 'Chemist'],
       ['Rachel Carson', '1907-05-27', '1964-04-
14', 56, 'Biologist'],
       ['John Snow', '1813-03-15', '1858-06-16',
45, 'Physician'],
       ['Alan Turing', '1912-06-23', '1954-06-07',
41,
        'Computer Scientist'],
       ['Johann Gauss', '1777-04-30', '1855-02-
23', 77, 'Mathematician']],
      dtype=object)
```

The `.values` comes in handy when you don't want all the row index label information, and really just want the base `numpy` representation of the data.

2.3.2 Boolean Subsetting: DataFrames

Just as we were able to subset a `Series` with a boolean vector, so can we subset a `DataFrame` with a `bool`.

Click here to view code image

```
# boolean vectors will subset rows
print(scientists.loc[scientists['Age'] >
scientists['Age'].mean()])
```

```
                   Name           Born            Died
```

```
Age      Occupation
1         William Gosset  1876-06-13  1937-10-16
61   Statistician
2  Florence Nightingale  1820-05-12  1910-08-13
90            Nurse
3            Marie Curie  1867-11-07  1934-07-04
66          Chemist
7           Johann Gauss  1777-04-30  1855-02-23
77  Mathematician
```

Table 2.3 summarizes the various subsetting methods.

Table 2.3 Table of DataFrame Subsetting Methods

Syntax	Selection Result
`df[column_name]`	`Series`
`df[[column1, column2, ... ]]`	`DataFrame`
`df.loc[row_label]`	Row by row index label (row name)
`df.loc[[label1, label2, ...]]`	Multiple rows by index label
`df.iloc[row_number]`	Row by row number
`df.iloc[[row1, row2, ...]]`	Multiple rows by row number
`df[bool]`	Row based on `bool`
`df[[bool1, bool2, ...]]`	Multiple rows based on `bool`
`df[start:stop:step]`	Rows based on slicing notation

2.3.3 Operations Are Automatically Aligned and Vectorized (Broadcasting)

Pandas supports *broadcasting* because the `Series` and `DataFrame` objects are built on top of the `numpy` library.[8] Broadcasting describes what happens when performing operations between array-like objects. These behaviors depend on the type of object, its length, and any labels associated with the object.

8. `NumPy` Library: `http://www.numpy.org/`

First, let's create a subset of our dataframes.

Click here to view code image

```
first_half = scientists[:4]
second_half = scientists[4:]
```

```
print(first_half)
```

```
                    Name        Born        Died
Age    Occupation
0     Rosaline Franklin  1920-07-25  1958-04-16
37        Chemist
1        William Gosset  1876-06-13  1937-10-16
61  Statistician
2   Florence Nightingale  1820-05-12  1910-08-13
90          Nurse
3           Marie Curie  1867-11-07  1934-07-04
66        Chemist
```

```
print(second_half)
```

```
            Name        Born        Died  Age
Occupation
4  Rachel Carson  1907-05-27  1964-04-14   56
Biologist
5      John Snow  1813-03-15  1858-06-16   45
Physician
```

```
6    Alan Turing  1912-06-23  1954-06-07   41
Computer Scientist
7   Johann Gauss  1777-04-30  1855-02-23   77
Mathematician
```

When we perform an action on a dataframe with a scalar, it will try to apply the operation on each cell of the dataframe. In this example, numbers will be multiplied by 2, and strings will be doubled (this is Python's normal behavior with strings).

Click here to view code image

```
# multiply by a scalar
print(scientists * 2)
```

```
                                       Name
Born  \
0        Rosaline FranklinRosaline Franklin  1920-
07-251920-07-25
1              William GossetWilliam Gosset  1876-
06-131876-06-13
2  Florence NightingaleFlorence Nightingale  1820-
05-121820-05-12
3                    Marie CurieMarie Curie  1867-
11-071867-11-07
4                Rachel CarsonRachel Carson  1907-
05-271907-05-27
5                      John SnowJohn Snow  1813-
03-151813-03-15
6                    Alan TuringAlan Turing  1912-
06-231912-06-23
7                  Johann GaussJohann Gauss  1777-
04-301777-04-30
```

```
                  Died  Age
Occupation
0  1958-04-161958-04-16   74
ChemistChemist
1  1937-10-161937-10-16  122
StatisticianStatistician
2  1910-08-131910-08-13  180
NurseNurse
3  1934-07-041934-07-04  132
ChemistChemist
4  1964-04-141964-04-14  112
BiologistBiologist
5  1858-06-161858-06-16   90
PhysicianPhysician
6  1954-06-071954-06-07   82  Computer
ScientistComputer Scientist
7  1855-02-231855-02-23  154
MathematicianMathematician
```

If your dataframes are all numeric values and you want to "add" the values on a cell-by-cell basis, you can use the `.add()` method. The automatic alignment can be better seen in Chapter 6, when we concatenate dataframes together.

2.4 Making Changes to Series and DataFrames

Now that we know various ways of subsetting and slicing our data (see Table 2.3), we should be able to alter our data objects.

2.4.1 Add Additional Columns

The `type` of the `Born` and `Died` columns is `object`, meaning they are strings or a sequence of characters.

```
print(scientists.dtypes)
```

```
Name           object
Born           object
Died           object
Age             int64
Occupation     object
dtype: object
```

We can convert the strings to a proper `datetime` type so we can perform common date and time operations (e.g., take differences between dates or calculate a person's age). You can provide your own `format` if you have a date that has a specific format. A list of `format` variables can be found in the Python `datetime` module documentation.[9] More examples with datetimes can be found in Chapter 12. The format of our date looks like "YYYY-MM-DD," so we can use the `%Y-%m-%d` format.

9. `datetime` module documentation: https://docs.python.org/3.10/library/datetime.html#strftime-and-strptime-behavior

Click here to view code image

```
# format the 'Born' column as a datetime
born_datetime =
pd.to_datetime(scientists['Born'], format='%Y-
%m-%d')
print(born_datetime)
```

```
0    1920-07-25
1    1876-06-13
2    1820-05-12
3    1867-11-07
4    1907-05-27
```

```
5    1813-03-15
6    1912-06-23
7    1777-04-30
Name: Born, dtype: datetime64[ns]
```

```
# format the 'Died' column as a datetime
died_datetime =
pd.to_datetime(scientists['Died'], format='%Y-
%m-%d')
```

If we wanted, we could create a new set of columns that contain the `datetime` representations of the `object` (string) dates. The below example uses python's multiple assignment syntax (Appendix N).

Click here to view code image

```
scientists['born_dt'], scientists['died_dt'] = (
  born_datetime,
  died_datetime
)

print(scientists.head())
```

```
                   Name        Born        Died
Age     Occupation  \
0     Rosaline Franklin  1920-07-25  1958-04-16
37       Chemist
1        William Gosset  1876-06-13  1937-10-16
61   Statistician
2  Florence Nightingale  1820-05-12  1910-08-13
90         Nurse
3           Marie Curie  1867-11-07  1934-07-04
66       Chemist
```

```
4          Rachel Carson  1907-05-27  1964-04-14
56     Biologist

     born_dt     died_dt
0 1920-07-25  1958-04-16
1 1876-06-13  1937-10-16
2 1820-05-12  1910-08-13
3 1867-11-07  1934-07-04
4 1907-05-27  1964-04-14
```

```
print(scientists.shape)
```

```
(8, 7)
```

```
print(scientists.dtypes)
```

```
Name                      object
Born                      object
Died                      object
Age                        int64

Occupation                object
born_dt           datetime64[ns]
died_dt           datetime64[ns]
dtype: object
```

2.4.2 Directly Change a Column

We can also assign a new value directly to the existing column. The example in this section shows how to randomize the contents of a column. More complex calculations that involve multiple columns can be seen in Chapter 5, in the discussion of the `.apply()` method.

First, let's look at the original `Age` values.

```
print(scientists['Age'])
```

```
0    37
1    61
2    90
3    66
4    56
5    45
6    41
7    77
Name: Age, dtype: int64
```

Now let’s shuffle the values.

Click here to view code image

```
# the frac=1 tells pandas to randomly select
100% of the values
# the random_state makes the randomization the
same each time
scientists["Age"] =
scientists["Age"].sample(frac=1,
random_state=42)
```

Note

We set a `random_state` as a way to make sure it randomly picks the same values on each run of the code. This way the code stats consistent when the code from the book is generated. But this technique is also useful when you are programming to make sure your values are not constantly fluctuating when you are trying to do something randomly. You can always remove it to make it completely random every time the code runs.

For long bits of code we can wrap the code around round parentheses `( )` to break up the code into multiple lines. We will be using this convention for longer bits of code in this book (Appendix D.1).

Click here to view code image

```
# the previous line of code is equivalent to
scientists['Age'] = (
  scientists['Age']
  .sample(frac=1, random_state=42)
)
```

```
print(scientists['Age'])
```

```
0    37
1    61
2    90
3    66
4    56
5    45
6    41
7    77
Name: Age, dtype: int64
```

If you notice, that we tried to randomly shuffle the column, but when we assigned the values back into the dataframe, it reverted back to the original order. That's because Pandas will try to automatically join on the `.index` values on many operations, for this example to get around this problem we need to remove that `.index` information. One way of doing that, is to assign just the `.values` of the shuffled values that does not have any `.index` value associated with it.

Click here to view code image

```
scientists['Age'] = (
scientists['Age']
  .sample(frac=1, random_state=42)
  .values # remove the index so it doesn't auto
align the values
)

print(scientists['Age'])
```

```
0    61
1    45
2    37
3    77
4    90
5    56
6    66
7    41
Name: Age, dtype: int64
```

We can recalculate the “real” age using `datetime` arithmetic. More information about `datetime` can be found in Chapter 12.

Click here to view code image

```
# subtracting dates will give us number of days
scientists['age_days'] = (
  scientists['died_dt'] - scientists['born_dt']
)

print(scientists)
```

```
                   Name        Born        Died
Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16
```

```
61
1        William Gosset  1876-06-13  1937-10-16
45
2  Florence Nightingale  1820-05-12  1910-08-13
37

3           Marie Curie  1867-11-07  1934-07-04
77
4         Rachel Carson  1907-05-27  1964-04-14
90
5             John Snow  1813-03-15  1858-06-16
56
6           Alan Turing  1912-06-23  1954-06-07
66
7          Johann Gauss  1777-04-30  1855-02-23
41

            Occupation    born_dt    died_dt
age_days
0              Chemist 1920-07-25 1958-04-16 13779
days
1         Statistician 1876-06-13 1937-10-16 22404
days
2                Nurse 1820-05-12 1910-08-13 32964
days
3              Chemist 1867-11-07 1934-07-04 24345
days
4            Biologist 1907-05-27 1964-04-14 20777
days
5            Physician 1813-03-15 1858-06-16 16529
days
6   Computer Scientist 1912-06-23 1954-06-07 15324
days
```

```
7        Mathematician 1777-04-30 1855-02-23 28422
days
```

```
# we can convert the value to just the year
# using the astype method
scientists['age_years'] = (
  scientists['age_days']
  .astype('timedelta64[Y]')
)
print(scientists)
```

```
                   Name        Born        Died
Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16
61
1        William Gosset  1876-06-13  1937-10-16
45
2  Florence Nightingale  1820-05-12  1910-08-13
37
3           Marie Curie  1867-11-07  1934-07-04
77
4         Rachel Carson  1907-05-27  1964-04-14
90
5             John Snow  1813-03-15  1858-06-16
56
6           Alan Turing  1912-06-23  1954-06-07
66
7          Johann Gauss  1777-04-30  1855-02-23
41

           Occupation    born_dt    died_dt
age_days  age_years
0             Chemist 1920-07-25 1958-04-16 13779
```

```
days        37.0
1        Statistician 1876-06-13 1937-10-16 22404
days        61.0
2               Nurse 1820-05-12 1910-08-13 32964
days        90.0
3             Chemist 1867-11-07 1934-07-04 24345
days        66.0
4           Biologist 1907-05-27 1964-04-14 20777
days        56.0
5           Physician 1813-03-15 1858-06-16 16529
days        45.0
6  Computer Scientist 1912-06-23 1954-06-07 15324
days        41.0
7       Mathematician 1777-04-30 1855-02-23 28422
days        77.0
```

Important

Many functions and methods in the `pandas` library will have an `inplace` parameter that you can set to the value `True`. When this is set, the function or method will return `None` instead of the modified dataframe. Generally, you do not want to use this parameter.

Contrary to popular belief, this does not make things go faster, and the parameter may be deprecated in the future: `https://github.com/pandas-dev/pandas/issues/16529`

2.4.3 Modifying Columns with `.assign()`

Another way you can assign and modify columns is with the `.assign()` method. This has the benefit of using method chaining (Appendix R). Let's redo the `age_years` column creation, but this time using `'.assign()`.

Click here to view code image

```
scientists = scientists.assign(
  # new columns on the left of the equal sign
  # how to calculate values on the right of the
equal sign
  # separate new columns with a comma
  age_days_assign=scientists['died_dt'] -
scientists['born_dt'],

age_year_assign=scientists['age_days'].astype('t
imedelta64[Y]')
)

print(scientists)
```

```
                   Name        Born        Died
Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16
61
1        William Gosset  1876-06-13  1937-10-16
45
2  Florence Nightingale  1820-05-12  1910-08-13
37
3           Marie Curie  1867-11-07  1934-07-04
77
4         Rachel Carson  1907-05-27  1964-04-14
90
5             John Snow  1813-03-15  1858-06-16
56
6           Alan Turing  1912-06-23  1954-06-07
66
7          Johann Gauss  1777-04-30  1855-02-23
41
```

```
            Occupation    born_dt    died_dt
age_days  age_years  \
0            Chemist 1920-07-25 1958-04-16 13779
days        37.0
1       Statistician 1876-06-13 1937-10-16 22404
days        61.0
2              Nurse 1820-05-12 1910-08-13 32964
days        90.0
3            Chemist 1867-11-07 1934-07-04 24345
days        66.0
4          Biologist 1907-05-27 1964-04-14 20777
days        56.0
5          Physician 1813-03-15 1858-06-16 16529
days        45.0
6  Computer Scientist 1912-06-23 1954-06-07 15324
days        41.0
7      Mathematician 1777-04-30 1855-02-23 28422
days        77.0

  age_days_assign  age_year_assign
0      13779 days             37.0
1      22404 days             61.0
2      32964 days             90.0
3      24345 days             66.0
4      20777 days             56.0
5      16529 days             45.0
6      15324 days             41.0
7      28422 days             77.0
```

You can look into the `.assign()` documentation for more examples.[10] Since this is only showing a simple example of how to use the method to assign new values. Effectively using `.assign()` will require you to know about `lambda` functions, which we will cover in Chapter 5.

10. `.assign()` documentation: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.assign.html

Note

In the example we just did with `.assign()`, we did not use the first new value, `age_days_assign`, in the calculation for the second new value, `age_year_assign`. We would have to know how to write a `lambda` functions to know how the following code works.

Click here to view code image

```
scientists = scientists.assign(
    age_days_assign=scientists["died_dt"] -
scientists["born_dt"],
    age_year_assign=lambda df_:
df_["age_days_assign"].astype(
        "timedelta64[Y]"
    ),
)
print(scientists)
```

```
                   Name        Born        Died
Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16
61
1        William Gosset  1876-06-13  1937-10-16
45
2  Florence Nightingale  1820-05-12  1910-08-13
37
3           Marie Curie  1867-11-07  1934-07-04
77
4         Rachel Carson  1907-05-27  1964-04-14
90
```

```
5              John Snow  1813-03-15  1858-06-16
56
6            Alan Turing  1912-06-23  1954-06-07
66
7           Johann Gauss  1777-04-30  1855-02-23
41

           Occupation    born_dt    died_dt
age_days  age_years  \
0             Chemist 1920-07-25 1958-04-16
13779 days       37.0
1        Statistician 1876-06-13 1937-10-16
22404 days       61.0
2               Nurse 1820-05-12 1910-08-13
32964 days       90.0
3             Chemist 1867-11-07 1934-07-04
24345 days       66.0
4           Biologist 1907-05-27 1964-04-14
20777 days       56.0
5           Physician 1813-03-15 1858-06-16
16529 days       45.0
6  Computer Scientist 1912-06-23 1954-06-07
15324 days       41.0
7       Mathematician 1777-04-30 1855-02-23
28422 days       77.0

  age_days_assign  age_year_assign
0      13779 days             37.0
1      22404 days             61.0
2      32964 days             90.0
3      24345 days             66.0
```

```
4       20777 days                 56.0
5       16529 days                 45.0
6       15324 days                 41.0
7       28422 days                 77.0
```

2.4.4 Dropping Values

To drop a column, we can either select all the columns we want to by using the column subsetting techniques (Section 1.3.1), or select columns to drop with the `.drop()` method on our dataframe.[11]

11. `DataFrame .drop()` method: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.drop.html

Click here to view code image

```
# all the current columns in our data
print(scientists.columns)
```

```
Index(['Name', 'Born', 'Died', 'Age',
'Occupation', 'born_dt',
       'died_dt', 'age_days', 'age_years',
'age_days_assign',
       'age_year_assign'], dtype='object')
```

```
# drop the shuffled age column
# you provide the axis=1 argument to drop
column-wise
scientists_dropped = scientists.drop(['Age'],
axis="columns")
```

```
# columns after dropping our column
print(scientists_dropped.columns)
```

```
Index(['Name', 'Born', 'Died', 'Occupation',
'born_dt', 'died_dt',
       'age_days', 'age_years', 'age_days_assign',
       'age_year_assign'],
      dtype='object')
```

2.5 Exporting and Importing Data

In our examples so far, we have been importing data. It is also common practice to export or save data sets while processing them. Data sets are either saved out as final cleaned versions of data or in intermediate steps. Both of these outputs can be used for analysis or as input to another part of the data processing pipeline.

> **Tip**
>
> It's okay to save intermediate data set files as you work. You do not need to process *all* your data and analysis in one giant code script.
>
> Saving the data output from one script that gets imported from another is the basis of creating data pipelines.

2.5.1 Pickle

Python has a way to `pickle` data. This is Python's way of serializing and saving data in a binary format. Reading pickle data is also backwards compatible. `pickle` files are usually saved with an extension of `.p`, `.pkl`, or `.pickle`. We will see how to save and load `pickle` data below.

2.5.1.1 Series

Many of the export methods for a `Series` are also available for a `DataFrame`. Those readers who have experience with `numpy` will know that a `.save()` method is available for `ndarrays`. This method has been deprecated, and the replacement is to use the `.to_pickle` method.

Click here to view code image

```
names = scientists['Name']
print(names)
```

```
0       Rosaline Franklin
1          William Gosset
2    Florence Nightingale
3             Marie Curie
4           Rachel Carson
5               John Snow
6             Alan Turing
7            Johann Gauss
Name: Name, dtype: object
```

```
# pass in a string to the path you want to save
names.to_pickle('output/scientists_names_series.
pickle')
```

The `pickle` output is in a binary format. If you try to open it in a text editor, you will see a bunch of garbled characters.

If the object you are saving is an intermediate step in a set of calculations that you want to save, or if you know that your data will stay in the Python world, saving objects to a `pickle` will be optimized for Python and disk storage space. However, this approach means that people who do not use Python will not be able to read the data.

2.5.1.2 DataFrame

The same method can be used on `DataFrame` objects.

Click here to view code image

```
scientists.to_pickle('output/scientists_df.pickl
e')
```

2.5.1.3 Read `pickle` data

To read `pickle` data, we can use the `pd.read_pickle()` function.

Click here to view code image

```
# for a Series
series_pickle = pd.read_pickle(
    "output/scientists_names_series.pickle"
)
print(series_pickle)
```

```
0        Rosaline Franklin
1           William Gosset
2     Florence Nightingale
3              Marie Curie
4            Rachel Carson
5                John Snow
6              Alan Turing
7             Johann Gauss
Name: Name, dtype: object
```

```
# for a DataFrame
dataframe_pickle =
pd.read_pickle('output/scientists_df.pickle')
print(dataframe_pickle)
```

```
                   Name        Born        Died
Age \
0     Rosaline Franklin  1920-07-25  1958-04-16
61
```

```
1         William Gosset  1876-06-13  1937-10-16
45
2  Florence Nightingale  1820-05-12  1910-08-13
37
3            Marie Curie  1867-11-07  1934-07-04
77
4          Rachel Carson  1907-05-27  1964-04-14
90
5              John Snow  1813-03-15  1858-06-16
56
6            Alan Turing  1912-06-23  1954-06-07
66
7           Johann Gauss  1777-04-30  1855-02-23
41

           Occupation    born_dt    died_dt
age_days  age_years  \
0             Chemist 1920-07-25 1958-04-16 13779
days       37.0
1        Statistician 1876-06-13 1937-10-16 22404
days       61.0
2               Nurse 1820-05-12 1910-08-13 32964
days       90.0
3             Chemist 1867-11-07 1934-07-04 24345
days       66.0
4           Biologist 1907-05-27 1964-04-14 20777
days       56.0
5           Physician 1813-03-15 1858-06-16 16529
days       45.0
6  Computer Scientist 1912-06-23 1954-06-07 15324
days       41.0
7       Mathematician 1777-04-30 1855-02-23 28422
days       77.0
```

```
  age_days_assign  age_year_assign
0      13779 days             37.0
1      22404 days             61.0
2      32964 days             90.0
3      24345 days             66.0
4      20777 days             56.0
5      16529 days             45.0
6      15324 days             41.0
7      28422 days             77.0
```

Again, the `pickle` files are saved with an extension of `.p`, `.pkl`, or `.pickle`.

2.5.2 Comma-Separated Values (CSV)

Comma-separated values (CSV) are the most flexible data storage type. For each row, the column information is separated with a comma. The comma is not the only type of delimiter, however. Some files are delimited by a tab (TSV) or even a semicolon. The main reason why CSVs are a preferred data format when collaborating and sharing data is because any program can open this kind of data structure. It can even be opened in a text editor. However, the universal storage format does come at a price. CSV files are usually slower and take up more disk space when compared to other binary formats.

The `Series` and `DataFrame` have a `.to_csv()` method to write a CSV file. The documentation for `Series`[12] and `DataFrame`[13] identifies many different ways you can modify the resulting CSV file. For example, if you wanted to save a TSV file because there are commas in your data, you can change the `sep` parameter (Appendix O).

12. Saving a `Series` to CSV: https://pandas.pydata.org/docs/reference/api/pandas.Series.to_csv.html

13. Saving a `DataFrame` to CSV: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_csv.html

By default, the `.index` of a DataFrame gets written to the CSV file. This creates a file where the first column does not have a name, and only holds the row numbers of the dataframe being saved. This extraneous column in the CSV becomes problematic when you try to read the CSV back into Pandas. So we typically put in the `index=False` parameter when saving CSV files to avoid this problem.

Click here to view code image

```
# do not write the row names in the CSV output
scientists.to_csv('output/scientists_df_no_index
.csv', index=False)
```

2.5.2.1 Import CSV Data

Importing CSV files was illustrated in Section 1.2. This operation uses the `pd.read_csv()` function. In the documentation, you can see there are various ways to read in a `CSV`.[14] Look at Appendix O if you need more information on using function parameters.

14. `pd.read_csv()` documentation: https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html

2.5.3 Excel

Excel, which is probably the most commonly used data type (or the second most commonly used, after CSVs), has a bad reputation within the data science community, mainly because colors and other superfluous information can easily find its way into the data set, not to mention one-off calculations that ruin the rectangular structure of a data set. Some other reasons are listed at the very beginning of this chapter. The goal of this book isn't to bash Excel, but to teach you about a reasonable alternative tool for data analytics. In short, the more of your work you can do in a scripting language, the easier it will be to scale up to larger projects, catch and fix mistakes, and collaborate. However, Excel's popularity and market share are unrivaled. Excel has its own scripting language if you absolutely have to work in it. This will allow you to work with data in a more predictable and reproducible manner.

2.5.3.1 Series

The `Series` data structure does not have an explicit `.to_excel()` method. If you have a `Series` that needs to be exported to an Excel file, one option is to convert the `Series` into a one-column `DataFrame`.

Before saving and reading Excel files, make sure you have the `openpyxl` library installed (using `pip install openpyxl` See Appendix B).

Click here to view code image

```
print(names)
```

```
0       Rosaline Franklin
1          William Gosset
2    Florence Nightingale
3             Marie Curie
4           Rachel Carson
5               John Snow
6             Alan Turing
7            Johann Gauss
Name: Name, dtype: object
```

```
# convert the Series into a DataFrame
# before saving it to an Excel file
names_df = names.to_frame()

# save to an excel file
names_df.to_excel(
```

```
  'output/scientists_names_series_df.xls',
engine='openpyxl'
)
```

2.5.3.2 DataFrames

From the preceding example, you can see how to export a `DataFrame` to an Excel file. The documentation shows several ways to further fine-tune the output.[15] For example, you can output data to a specific "sheet" using the `sheet_name` parameter.

15. `.to_excel()` documentation: `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_excel.html`

Click here to view code image

```
# saving a DataFrame into Excel format
scientists.to_excel(
  "output/scientists_df.xlsx",
  sheet_name="scientists",
  index=False
)
```

2.5.4 Feather

The format called "feather" is used to save `DataFrames` into a binary object that can also be loaded into other languages (e.g., R). The main benefit of this approach is that it is faster than writing and reading a CSV file between the languages. See the Pandas `.to_feather()`[16] and feather file format documentation[17] for more information on storing for backwards compatibility.

16. Pandas `to_feather()` documentation: `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_feather.html`

17. Feather file format documentation:
https://arrow.apache.org/docs/python/feather.html

The `feather` formatter is installed via `conda install -c conda-forge pyarrow` or `pip install pyarrow`. More on installing packages are described in Appendix B.

You can use the `.to_feather()` method on a dataframe to save the `feather` objects.

Click here to view code image

```
# save to feather file
scientists.to_feather('output/scientists.feather
')

# read feather file
sci_feather =
pd.read_feather('output/scientists.feather')

print(sci_feather)
```

```
                   Name        Born        Died
Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16
61
1        William Gosset  1876-06-13  1937-10-16
45
2  Florence Nightingale  1820-05-12  1910-08-13
37
3           Marie Curie  1867-11-07  1934-07-04
77
4         Rachel Carson  1907-05-27  1964-04-14
90
5             John Snow  1813-03-15  1858-06-16
56
```

```
6           Alan Turing  1912-06-23  1954-06-07
66
7          Johann Gauss  1777-04-30  1855-02-23
41

          Occupation    born_dt    died_dt
age_days  age_years  \
0            Chemist 1920-07-25 1958-04-16 13779
days       37.0
1       Statistician 1876-06-13 1937-10-16 22404
days       61.0
2              Nurse 1820-05-12 1910-08-13 32964
days       90.0
3            Chemist 1867-11-07 1934-07-04 24345
days       66.0
4          Biologist 1907-05-27 1964-04-14 20777
days       56.0
5          Physician 1813-03-15 1858-06-16 16529
days       45.0
6  Computer Scientist 1912-06-23 1954-06-07 15324
days       41.0
7      Mathematician 1777-04-30 1855-02-23 28422
days       77.0

  age_days_assign  age_year_assign
0      13779 days             37.0
1      22404 days             61.0
2      32964 days             90.0
3      24345 days             66.0
4      20777 days             56.0
5      16529 days             45.0
```

```
6       15324 days                41.0
7       28422 days                77.0
```

2.5.5 Arrow

Feather files are part of the Apache Arrow project.[18] One of the main goals of Arrow is to have a memory storage format for dataframe objects that work across multiple programming languages without having to convert types for each of them.

18. Apache Arrow: `https://arrow.apache.org/docs/index.html`]

> **Note**
>
> The Apache Arrow project is separate from the Python Arrow library, which is used for Dates and Times: `https://arrow.readthedocs.io/en/latest/`

Arrow has its own Pandas integration[19] to convert Pandas `DataFrame` objects to Arrow objects (`from_pandas()`[20]) and from Arrow objects to Pandas `DataFrame` objects (`to_pandas()`[21]). Once the data is in an Arrow format, it can much more efficiently be used in other programming languages.

19. Arrow Pandas integration: `https://arrow.apache.org/docs/python/pandas.html`

20. Arrow `from_pandas()`: `https://arrow.apache.org/docs/python/generated/pyarrow.Table.html#pyarrow.Table.from_pandas`

21. Arrow `to_pandas()`: `https://arrow.apache.org/docs/python/generated/pyarrow.Table.html#pyarrow.Table.to_pandas`

2.5.6 Dictionary

The Pandas `Series` and `DataFrame` objects also have a `.to_dict()` method. This converts the object into a Python dictionary object. This format is particularly useful if you have a `DataFrame` or `Series` and you want to use the data from outside Pandas.

Let's create a smaller subset of the `scientist` data so all the dictionary data will display properly

Click here to view code image

```
# first 2 rows of data
sci_sub_dict = scientists.head(2)

# convert the dataframe into a dictionary
sci_dict = sci_sub_dict.to_dict()

# using the pretty print library to print the
dictionary
import pprint
pprint.pprint(sci_dict)
```

```
{'Age': {0: 61, 1: 45},
 'Born': {0: '1920-07-25', 1: '1876-06-13'},
 'Died': {0: '1958-04-16', 1: '1937-10-16'},
 'Name': {0: 'Rosaline Franklin', 1: 'William
Gosset'},
 'Occupation': {0: 'Chemist', 1: 'Statistician'},
 'age_days': {0: Timedelta('13779 days 00:00:00'),
              1: Timedelta('22404 days
00:00:00')},
 'age_days_assign': {0: Timedelta('13779 days
00:00:00'),
                     1: Timedelta('22404 days
00:00:00')},
```

```
 'age_year_assign': {0: 37.0, 1: 61.0},
 'age_years': {0: 37.0, 1: 61.0},
 'born_dt': {0: Timestamp('1920-07-25 00:00:00'),
             1: Timestamp('1876-06-13 00:00:00')},
 'died_dt': {0: Timestamp('1958-04-16 00:00:00'),
             1: Timestamp('1937-10-16 00:00:00')}}
```

Once the dictionary output is created, we can read it back into Pandas.

Click here to view code image

```
# read in the dictionary object back into a
dataframe
sci_dict_df = pd.DataFrame.from_dict(sci_dict)
print(sci_dict_df)
```

```
                Name        Born        Died  Age
Occupation  \
0  Rosaline Franklin  1920-07-25  1958-04-16   61
Chemist
1     William Gosset  1876-06-13  1937-10-16   45
Statistician

     born_dt    died_dt   age_days  age_years
age_days_assign  \
0 1920-07-25 1958-04-16 13779 days       37.0
13779 days
1 1876-06-13 1937-10-16 22404 days       61.0
22404 days

  age_year_assign
0            37.0
1            61.0
```

Danger

Because the `scientists` data set we are working with includes dates and times, we cannot simply copy and paste the dictionary as a string into the `pd.DataFrame.from_dict()` function. You will get a `NameError: name 'Timedelta' is not defined` error.

Dates and times are stored in a different format from what gets printed to the screen. Depending on the `dtype` stored in the columns, your ability to simply copy and paste the `.to_dict()` output may or may not return the same exact dataframe back.

If you need a way to work with dates, you will actually need to **convert** it into a more general format and convert the value back into a date.

2.5.7 JSON (JavaScript Objectd Notation)

JSON data is another common plain text file format. The benefit of using the `.to_jsion()` is that it can convert dates and times for you to read back into Pandas. By using `orient='records'` we can either pass in the variable or copy and paste from the output to load it back into Pandas. The `indent=2` allows the output to print a bit nicer to the screen (and book).

Click here to view code image

```
# convert the dataframe into a dictionary
sci_json = sci_sub_dict.to_json(
  orient='records', indent=2, date_format="iso"
)
```

Click here to view code image

```
pprint.pprint(sci_json)
```

```
('[\n'
 '  {\n'
 '    "Name":"Rosaline Franklin",\n'
 '    "Born":"1920-07-25",\n'
 '    "Died":"1958-04-16",\n'
 '    "Age":61,\n'
 '    "Occupation":"Chemist",\n'
 '    "born_dt":"1920-07-25T00:00:00.000Z",\n'
 '    "died_dt":"1958-04-16T00:00:00.000Z",\n'
 '    "age_days":"P13779DT0H0M0S",\n'
 '    "age_years":37.0,\n'
 '    "age_days_assign":"P13779DT0H0M0S",\n'
 '    "age_year_assign":37.0\n'
 '  },\n'
 '  {\n'
 '    "Name":"William Gosset",\n'
 '    "Born":"1876-06-13",\n'
 '    "Died":"1937-10-16",\n'
 '    "Age":45,\n'
 '    "Occupation":"Statistician",\n'
 '    "born_dt":"1876-06-13T00:00:00.000Z",\n'
 '    "died_dt":"1937-10-16T00:00:00.000Z",\n'
 '    "age_days":"P22404DT0H0M0S",\n'
 '    "age_years":61.0,\n'
 '    "age_days_assign":"P22404DT0H0M0S",\n'
 '    "age_year_assign":61.0\n'
 '  }\n'
 ']')
```

```
# copy the string to re-create the dataframe
sci_json_df = pd.read_json(
  ('[\n'
 '  {\n'
```

```
'     "Name":"Rosaline Franklin",\n'
'     "Born":"1920-07-25",\n'
'     "Died":"1958-04-16",\n'
'     "Age":61,\n'
'     "Occupation":"Chemist",\n'
'     "born_dt":"1920-07-25T00:00:00.000Z",\n'
'      "died_dt":"1958-04-16T00:00:00.000Z",\n'
'     "age_days":"P13779DT0H0M0S",\n'
'     "age_years":37.0,\n'
'     "age_days_assign":"P13779DT0H0M0S",\n'
'     "age_year_assign":37.0\n'
'   },\n'
'   {\n'

'     "Name":"William Gosset",\n'
'     "Born":"1876-06-13",\n'
'     "Died":"1937-10-16",\n'
'     "Age":45,\n'
'     "Occupation":"Statistician",\n'
'     "born_dt":"1876-06-13T00:00:00.000Z",\n'
'     "died_dt":"1937-10-16T00:00:00.000Z",\n'
'     "age_days":"P22404DT0H0M0S",\n'
'     "age_years":61.0,\n'
'     "age_days_assign":"P22404DT0H0M0S",\n'
'     "age_year_assign":61.0\n'
'   }\n'
']'),
  orient="records"
)
print(sci_json_df)
```

```
                    Name             Born             Died     Age
Occupation \
```

```
0  Rosaline Franklin  1920-07-25  1958-04-16    61
Chemist
1     William Gosset  1876-06-13  1937-10-16    45
Statistician

                    born_dt
died_dt  \
0  1920-07-25T00:00:00.000Z  1958-04-
16T00:00:00.000Z
1  1876-06-13T00:00:00.000Z  1937-10-
16T00:00:00.000Z

         age_days  age_years age_days_assign
age_year_assign
0  P13779DT0H0M0S         37  P13779DT0H0M0S
37
1  P22404DT0H0M0S         61  P22404DT0H0M0S
61
```

Notice how the dates are all different from the original values? That's because we choose to convert the dates into ISO 8601 string format.

```
print(sci_json_df.dtypes)
```

```
Name                   object
Born                   object
Died                   object
Age                     int64
Occupation             object
born_dt                object
died_dt                object
age_days               object
age_years               int64
```

```
age_days_assign    object
age_year_assign     int64
dtype: object
```

If we want the original datetime object back, we need to convert that representation back into a date.

Click here to view code image

```
sci_json_df["died_dt_json"] =
pd.to_datetime(sci_json_df["died_dt"])

print(sci_json_df)
```

```
                Name        Born        Died  Age
Occupation  \
0  Rosaline Franklin  1920-07-25  1958-04-16   61
Chemist
1     William Gosset  1876-06-13  1937-10-16   45
Statistician

                    born_dt
died_dt  \
0  1920-07-25T00:00:00.000Z  1958-04-
16T00:00:00.000Z
1  1876-06-13T00:00:00.000Z  1937-10-
16T00:00:00.000Z

         age_days  age_years  age_days_assign
age_year_assign  \
0  P13779DT0H0M0S         37   P13779DT0H0M0S
37
1  P22404DT0H0M0S         61   P22404DT0H0M0S
61
```

```
                 died_dt_json
0  1958-04-16 00:00:00+00:00
1  1937-10-16 00:00:00+00:00
```

```
print(sci_json_df.dtypes)
```

```
Name                                  object
Born                                  object
Died                                  object
Age                                    int64
Occupation                            object
born_dt                               object
died_dt                               object
age_days                              object
age_years                              int64
age_days_assign                       object
age_year_assign                        int64
died_dt_json            datetime64[ns, UTC]
dtype: object
```

Working with dates and times is always tricky. We talk more about them in Chapter 12.

2.5.8 Other Data Output Types

There are many ways Pandas can export and import data. Indeed, `.to_pickle()`, `.to_csv()`, `.to_excel()`, `.to_feather()`, `.to_dict()` are only some of the data formats that can make their way into Pandas `DataFrames`. Table 2.4 lists some of these other output formats.

Table 2.4 `DataFrame` Export Methods

Export Method	Description
`.to_clipboard()`	Save data into the system clipboard for pasting
`.to_dense()`	Convert data into a regular “dense” `DataFrame`
`.to_dict()`	Convert data into a Python `dict`
`.to_gbq()`	Convert data into a Google BigQuery table
`.to_hdf()`	Save data into a hierarchal data format (HDF)
`.to_msgpack()`	Save data into a portable JSON-like binary
`.to_html()`	Convert data into a HTML table
`.to_json()`	Convert data into a JSON string
`.to_latex()`	Convert data into a LATEX tabular environment
`.to_records()`	Convert data into a record array
`.to_string()`	Show `DataFrame` as a string for `stdout`
`.to_sparse()`	Convert data into a `SparceDataFrame`
`.to_sql()`	Save data into a SQL database
`.to_stata()`	Convert data into a Stata `dta` file

Conclusion

This chapter went into a little more detail about how the Pandas `Series` and `DataFrame` objects work in Python. There were some simpler examples of data cleaning shown, along with a few common ways to export data to share with others. Chapter 1 and Chapter 2 should give you a good basis on how Pandas works as a library.

The next chapter covers the basics of plotting in Python and Pandas. Data visualization is not only used at the end of an analysis to plot results, but also is heavily utilized throughout the entire data pipeline.

3

Plotting Basics

Data visualization is as much a part of the data processing step as the data presentation step. It is much easier to compare plotted values than to compare numerical values. By visualizing data we can get a better intuitive sense of the data than would be possible by looking at tables of values alone. Additionally, visualizations can bring to light hidden patterns in data, that you, the analyst, can use for model selection.

Learning Objectives

The concept map for this chapter can be found in Figure A.3.

- Explain why visualizing data is important
- Create various statistical plots for exploratory data analysis
- Use plotting functions from the `matplotlib`, `seaborn`, and `pandas` libraries
- Identify when to use univariate, bivariate, and multivariate plots
- Use different color palettes to make plots more accessible

3.1 Why Visualize Data?

The quintessential example for creating visualizations of data is Anscombe's quartet. This data set was created by English statistician Frank Anscombe to show the importance of statistical graphs.

The Anscombe data set contains four sets of data, each of which contains two continuous variables. Each set has the same mean, variance, correlation, and regression line. However, only when the data are visualized does it become obvious that each set does not follow the same pattern. This goes to show the benefits of visualizations and the pitfalls of looking at only summary statistics.

Click here to view code image

```
# the anscombe data set can be found in the
seaborn library
import seaborn as sns
anscombe = sns.load_data set("anscombe")
print(anscombe)
```

```
 data set      x      y
0        I   10.0   8.04
1        I    8.0   6.95
2        I   13.0   7.58
3        I    9.0   8.81
4        I   11.0   8.33
..     ...    ...    ...
39      IV    8.0   5.25
40      IV   19.0  12.50
41      IV    8.0   5.56
42      IV    8.0   7.91
43      IV    8.0   6.89

[44 rows x 3 columns]
```

3.2 Matplotlib Basics

`matplotlib` is Python's fundamental plotting library. It is extremely flexible and gives the user full control over all elements of the plot.

Importing the `matplotlib` plotting features is a little different from our previous package imports. You can think of it as importing the package `matplotlib`, with all of the plotting utilities stored under a subfolder (or subpackage) called `pyplot`. Just as we imported a package and gave it an abbreviated name, we can do the same with `matplotlib.pyplot`.

Click here to view code image

```
import matplotlib.pyplot as plt
```

The names of most of the basic plots will start with `plt.plot()`. In our example, the plotting feature takes one vector for the `x`-values, and a corresponding vector for the `y`-values (Figure 3.1).

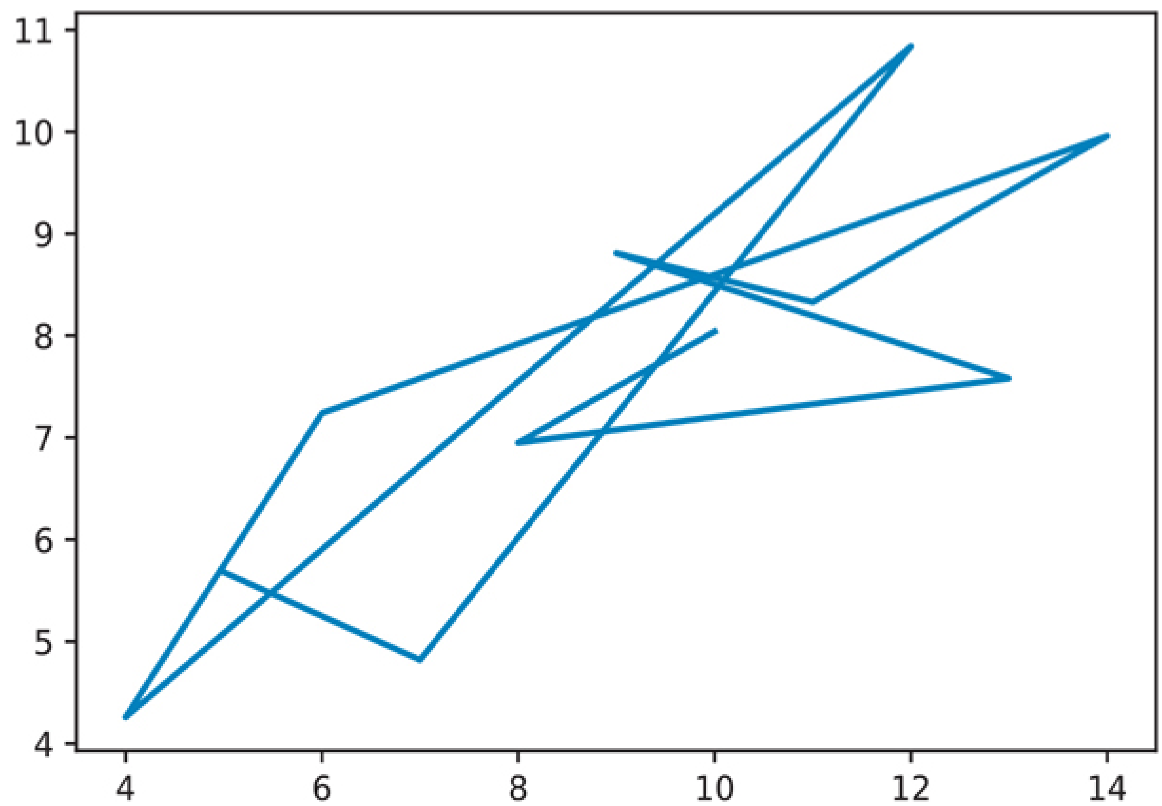


Figure 3.1 Anscombe data set I

Click here to view code image

```
# create a subset of the data
# contains only data set 1 from anscombe
```

```
data set_1 = anscombe[anscombe['data set'] ==
'I']

plt.plot(data set_1['x'], data set_1['y'])
plt.show() # will need this to show explicitly
show the plot
```

By default, `plt.plot()` will draw lines. If we want it to draw points instead, we can pass an `'o'` parameter to tell `plt.plot()` to use points (Figure 3.2).

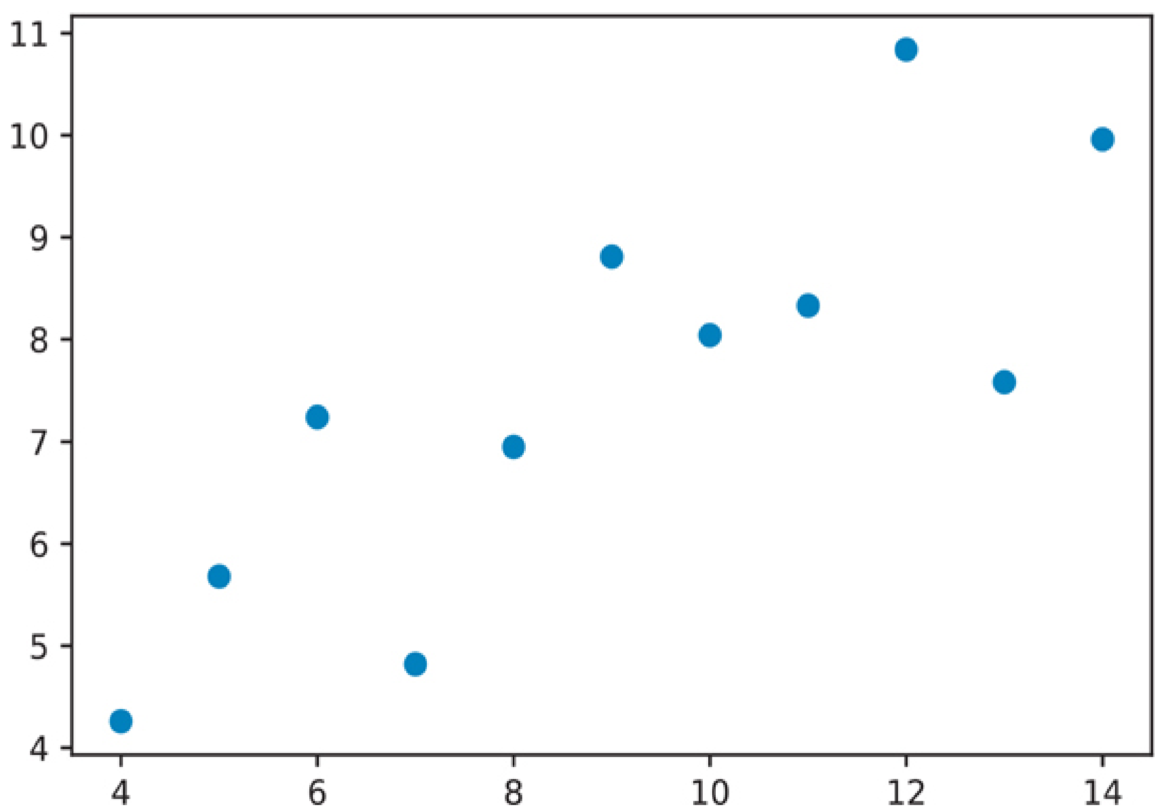


Figure 3.2 Anscombe data set I using points

Click here to view code image

```
plt.plot(data set_1['x'], data set_1['y'], 'o')
plt.show()
```

We can repeat this process for the rest of the `data sets` in our anscombe data.

Click here to view code image

```
# create subsets of the anscombe data
data_set_2 = anscombe[anscombe['data set'] ==
'II']
data_set_3 = anscombe[anscombe['data set'] ==
'III']
data_set_4 = anscombe[anscombe['data set'] ==
'IV']
```

3.2.1 Figure Objects and Axes Subplots

At this point, we could make these plots individually, but `matplotlib` offers a much handier way to create subplots. You can specify the dimensions of your final figure, and put in smaller plots to fit the specified dimensions. This way, you can present your results in a single figure.

The `subplot` syntax takes three parameters:

- Number of rows in the figure for subplots
- Number of columns in the figure for subplots
- Subplot location

The subplot location is sequentially numbered, and plots are placed first in a left-to-right direction, then from top to bottom. If we try to plot this now (by running the following code), we will get an empty figure (Figure 3.3). All we have done so far is create a figure and split it into a 2 x 2 grid where plots can be placed. Since no plots were created and inserted, nothing will show up.

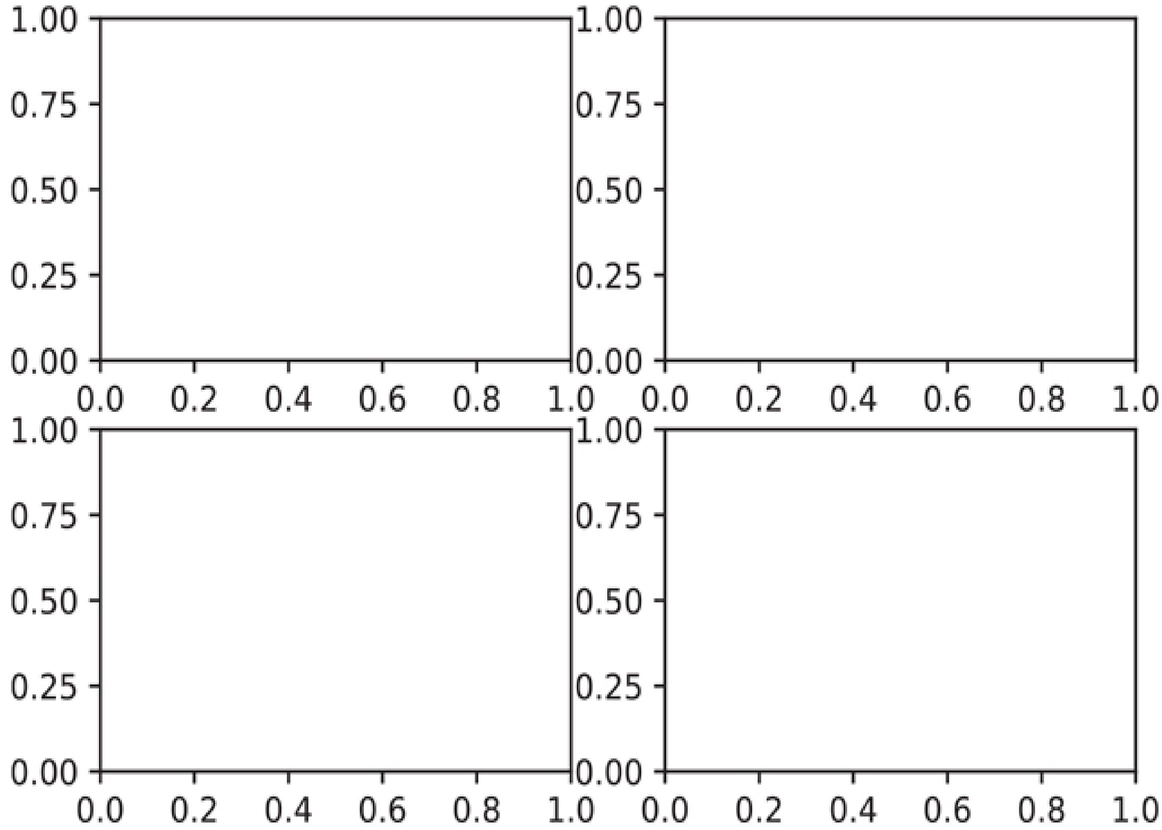


Figure 3.3 Matplotlib figure with four empty axes in a 2x2 grid

Click here to view code image

```
# create the entire figure where our subplots
will go
fig = plt.figure()

# tell the figure how the subplots should be
laid out
# in the example, we will have
# 2 row of plots, and each row will have 2 plots

# subplot has 2 rows and 2 columns, plot
location 1
axes1 = fig.add_subplot(2, 2, 1)
```

```
# subplot has 2 rows and 2 columns, plot
location 2
axes2 = fig.add_subplot(2, 2, 2)

# subplot has 2 rows and 2 columns, plot
location 3
axes3 = fig.add_subplot(2, 2, 3)
```

Click here to view code image

```
# subplot has 2 rows and 2 columns, plot
location 4
axes4 = fig.add_subplot(2, 2, 4)

plt.show()
```

We can use the `.plot()` method on each axis to create our plot (Figure 3.4).

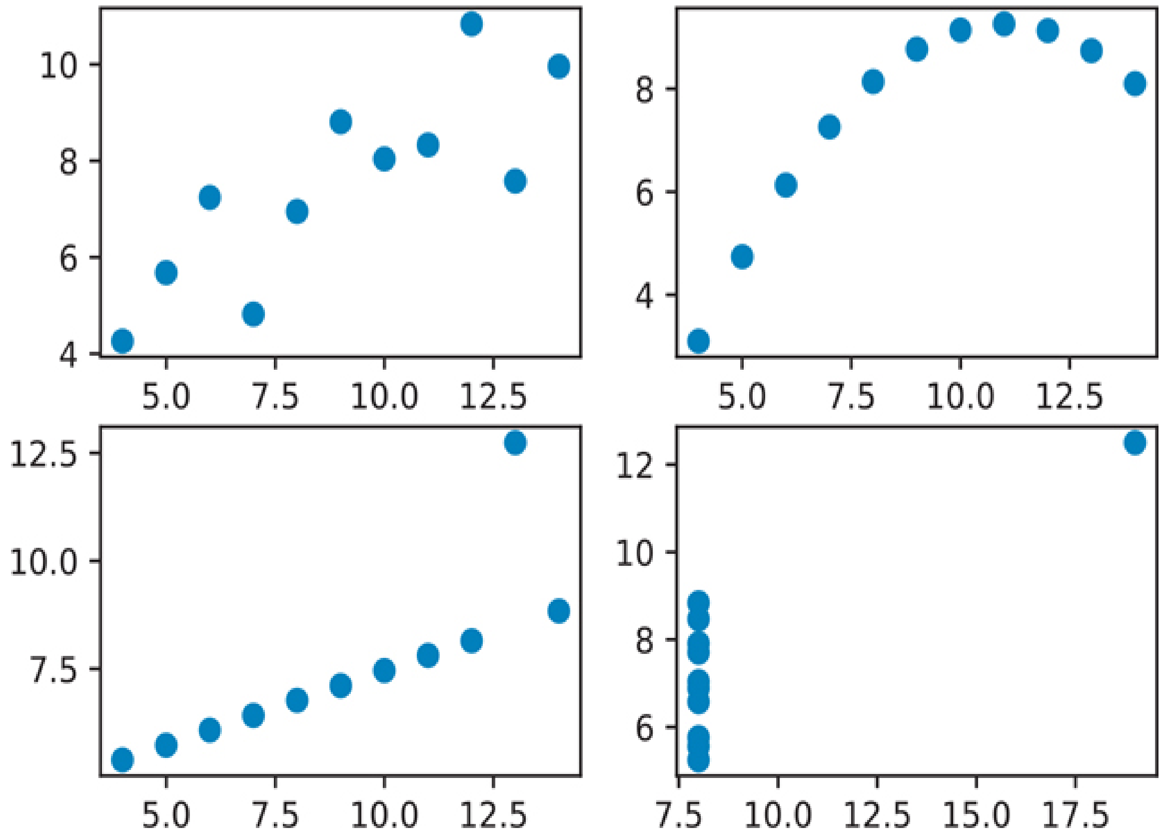


Figure 3.4 Matplotlib figure with four scatter plots

Important

With a lot of plotting code, you need to run *all* the code together. Usually, running parts of it as you attempt to build on a figure will not return anything.

Click here to view code image

```
# you need to run all the plotting code
together, same as above
fig = plt.figure()
axes1 = fig.add_subplot(2, 2, 1)
axes2 = fig.add_subplot(2, 2, 2)
axes3 = fig.add_subplot(2, 2, 3)
```

```
axes4 = fig.add_subplot(2, 2, 4)

# add a plot to each of the axes created above
axes1.plot(data set_1['x'], data set_1['y'],
'o')
axes2.plot(data set_2['x'], data set_2['y'],
'o')
axes3.plot(data set_3['x'], data set_3['y'],
'o')
axes4.plot(data set_4['x'], data set_4['y'],
'o')

plt.show()
```

Finally, we can add a label to our subplots, and improve the subplot spacing with `fig.tight_layout()`, but `fig.set_tight_layout()` is preferred (Figure 3.5).

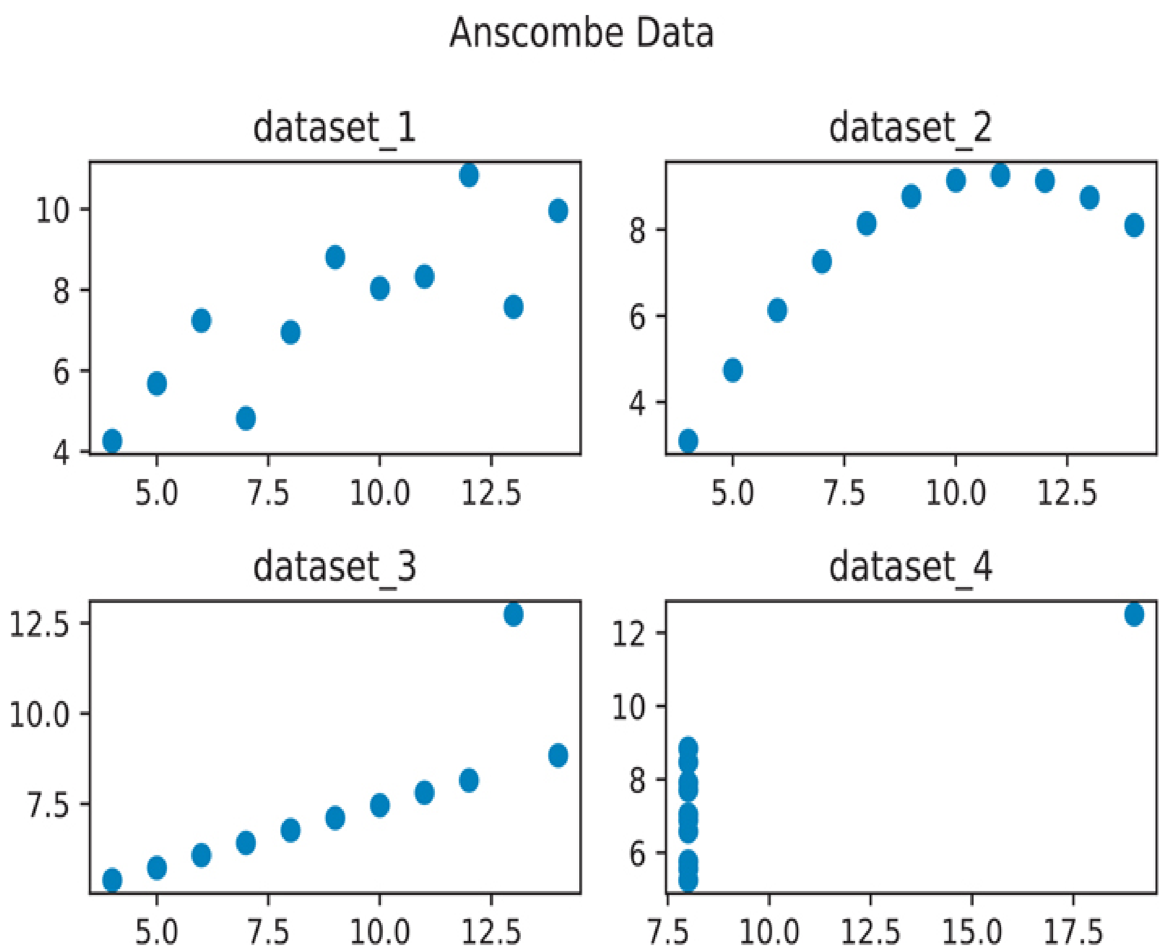


Figure 3.5 Anscombe data visualization

Click here to view code image

```
# you need to run all the plotting code
together, same as above
fig = plt.figure()
axes1 = fig.add_subplot(2, 2, 1)
axes2 = fig.add_subplot(2, 2, 2)
axes3 = fig.add_subplot(2, 2, 3)
axes4 = fig.add_subplot(2, 2, 4)
axes1.plot(data set_1['x'], data set_1['y'],
'o')
axes2.plot(data set_2['x'], data set_2['y'],
```

```
'o')
axes3.plot(data set_3['x'], data set_3['y'],
'o')
axes4.plot(data set_4['x'], data set_4['y'],
'o')

# add a small title to each subplot
axes1.set_title("data set_1")
axes2.set_title("data set_2")
axes3.set_title("data set_3")
axes4.set_title("data set_4")

# add a title for the entire figure (title above
the title)
fig.suptitle("Anscombe Data") # note spelling of
"suptitle"

# use a tight layout so the plots and titles
don't overlap
fig.set_tight_layout(True)

# show the figure
plt.show()
```

The Anscombe data visualizations illustrate why just looking at summary statistical values can be misleading. The moment the points are visualized, it becomes clearer that even though each data set has the same summary statistical values, the relationships between points vastly differ across the data sets.

To finish off the Anscombe example, we can add `.set_xlabel()` and `.set_ylabel()` to each of the subplots to add `x`- and `y`-axis labels, just as we added a title to the figure.

3.2.2 Anatomy of a Figure

Before we move on and learn how to create more statistical plots, you should become familiar with the `matplotlib` documentation on "Anatomy of a Figure."[1] I have reproduced its older figure in Figure 3.6, and the newer figure in Figure 3.7.

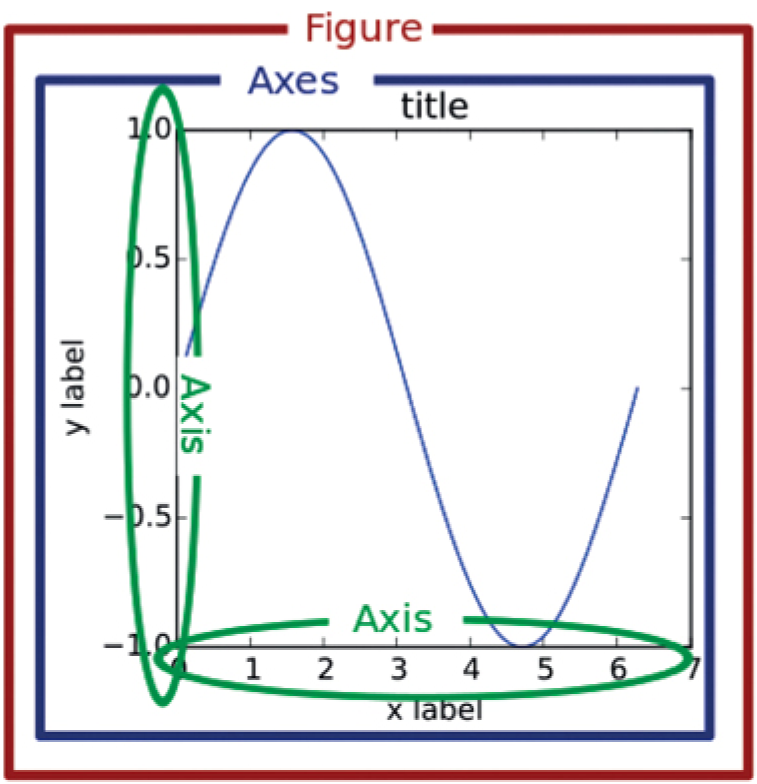


Figure 3.6 Matplotlib anatomy of a figure (old version)

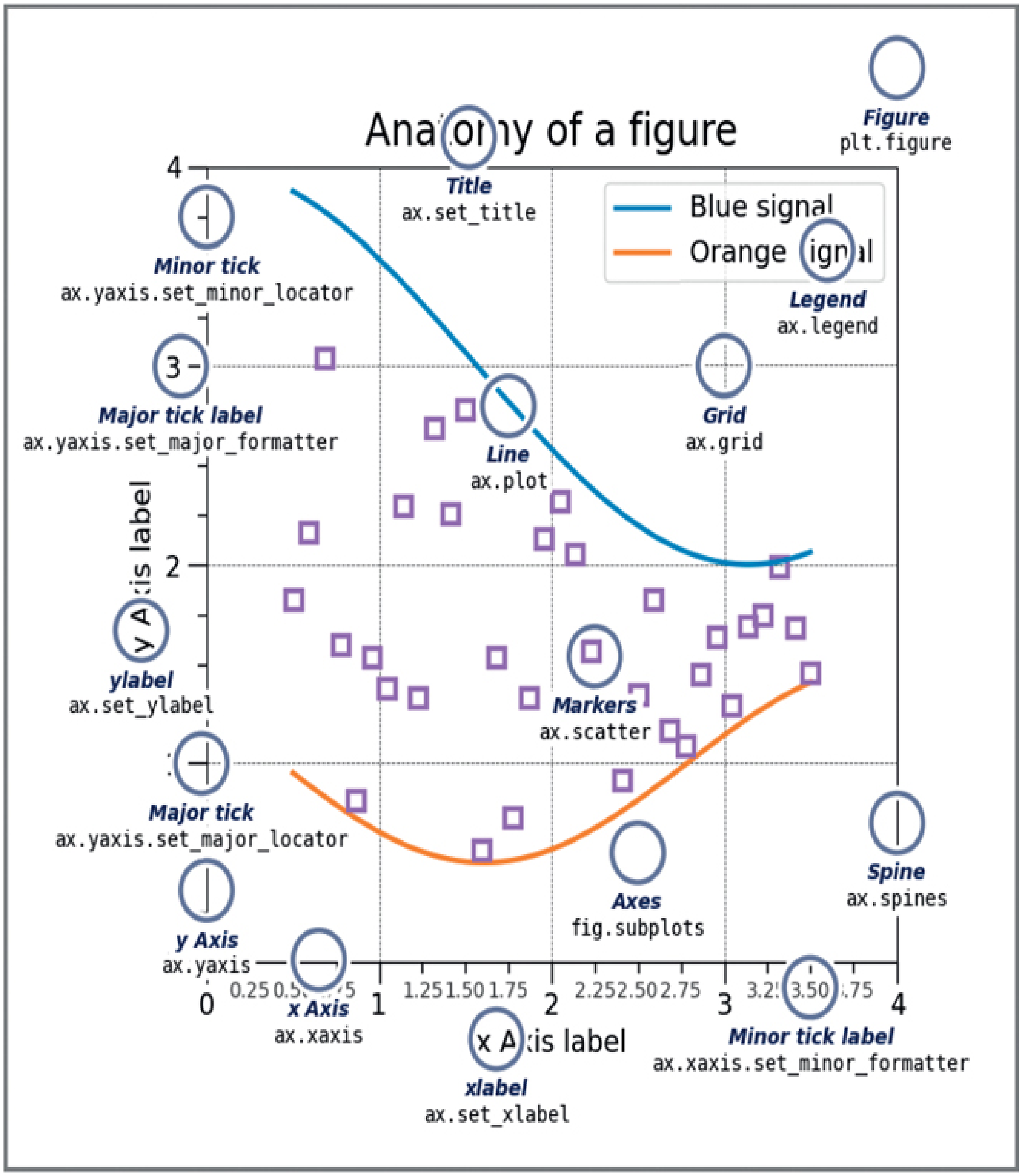


Figure 3.7 Matplotlib anatomy of a figure (new version)

1. Anatomy of a `matplotlib` figure:
https://matplotlib.org/stable/gallery/showcase/anatomy.html

One of the most confusing parts of plotting in Python is the use of the terms “axis” and “axes” especially when trying to verbally describe the different parts (since they are pronounced similarly). In the Anscombe example, each individual subplot plot has axes. The axes contain both an x-axis and a y-axis. All four subplots together make the figure.

The remainder of the chapter shows you how to create statistical plots, first with matplotlib and later using a higher-level plotting library that is based on matplotlib and specifically made for statistical graphics, seaborn.

> **Important**
>
> Knowing whether or not a plotting function returns one or more axes or a figure will be important to know when plotting. For example, you can't put a figure inside another figure as you can with one or more axes.

3.3 Statistical Graphics Using matplotlib

The tips data we will be using for the next series of visualizations come from the seaborn library. This data set contains the amount of the tips that people leave for various variables. For example, the total cost of the bill, the size of the party, the day of the week, and the time of day.

We can load this data set just as we did the Anscombe data set.

Click here to view code image

```
tips = sns.load_data set("tips")
print(tips)
```

```
    total_bill  tip     sex smoker  day   time size
0        16.99 1.01  Female     No  Sun Dinner    2
1        10.34 1.66    Male     No  Sun Dinner    3
2        21.01 3.50    Male     No  Sun Dinner    3
3        23.68 3.31    Male     No  Sun Dinner    2
4        24.59 3.61  Female     No  Sun Dinner    4
..         ...  ...     ...    ...  ...    ...  ...
```

```
239       29.03 5.92   Male     No  Sat Dinner    3
240       27.18 2.00 Female    Yes  Sat Dinner    2
241       22.67 2.00   Male    Yes  Sat Dinner    2
242       17.82 1.75   Male     No  Sat Dinner    2
243       18.78 3.00 Female     No Thur Dinner    2

[244 rows x 7 columns]
```

3.3.1 Univariate (Single Variable)

In statistics jargon, the term “univariate” refers to a single variable.

3.3.1.1 Histograms

Histograms are the most common means of looking at a single variable. The values are “binned”, meaning they are grouped together and plotted to show the distribution of the variable (Figure 3.8).

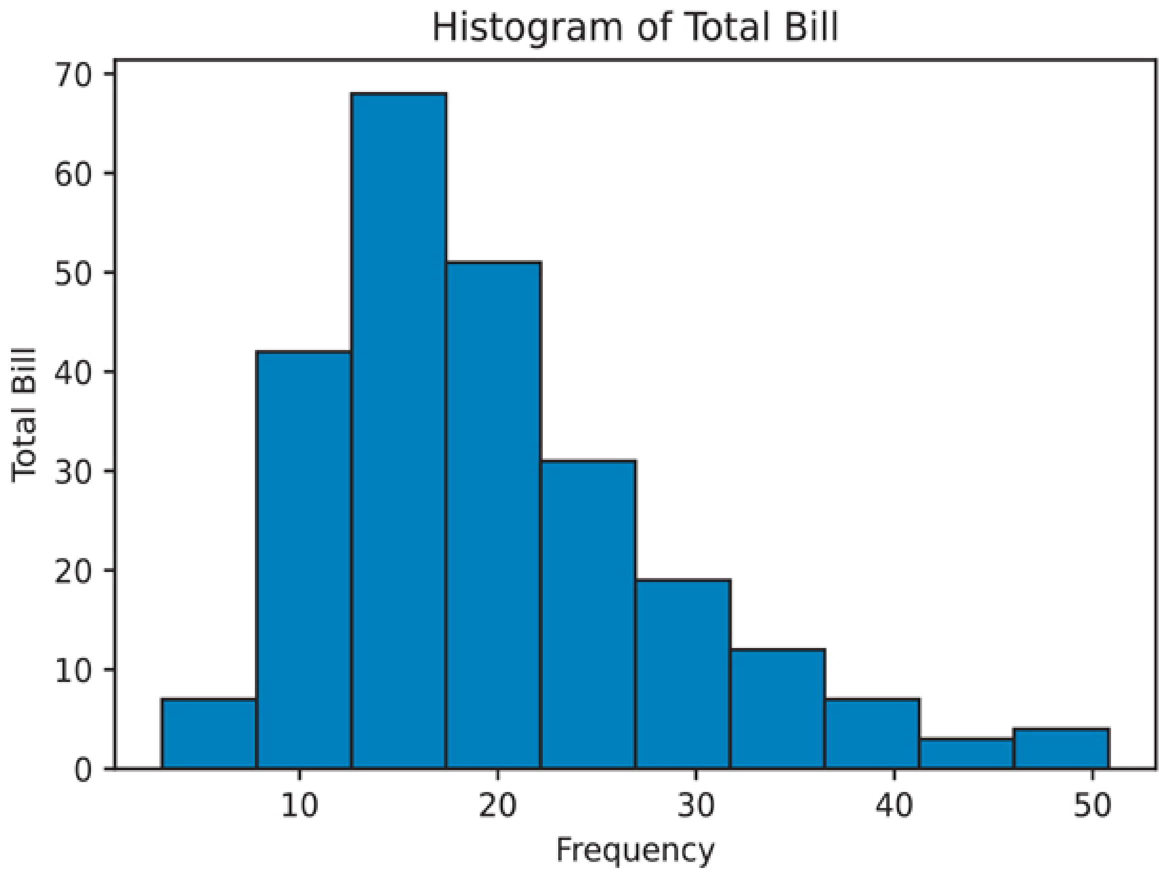


Figure 3.8 Histogram using matplotlib

Click here to view code image

```
# create the figure object
fig = plt.figure()

# subplot has 1 row, 1 column, plot location 1
axes1 = fig.add_subplot(1, 1, 1)

# make the actual histogram
axes1.hist(data=tips, x='total_bill', bins=10)

# add labels
axes1.set_title('Histogram of Total Bill')
axes1.set_xlabel('Frequency')
```

```
axes1.set_ylabel('Total Bill')

plt.show()
```

3.3.2 Bivariate (Two Variables)

In statistics jargon, the term “bivariate” refers to two variables.

3.3.2.1 Scatter Plot

Scatter plots are used when a continuous variable is plotted against another continuous variable (Figure 3.9).

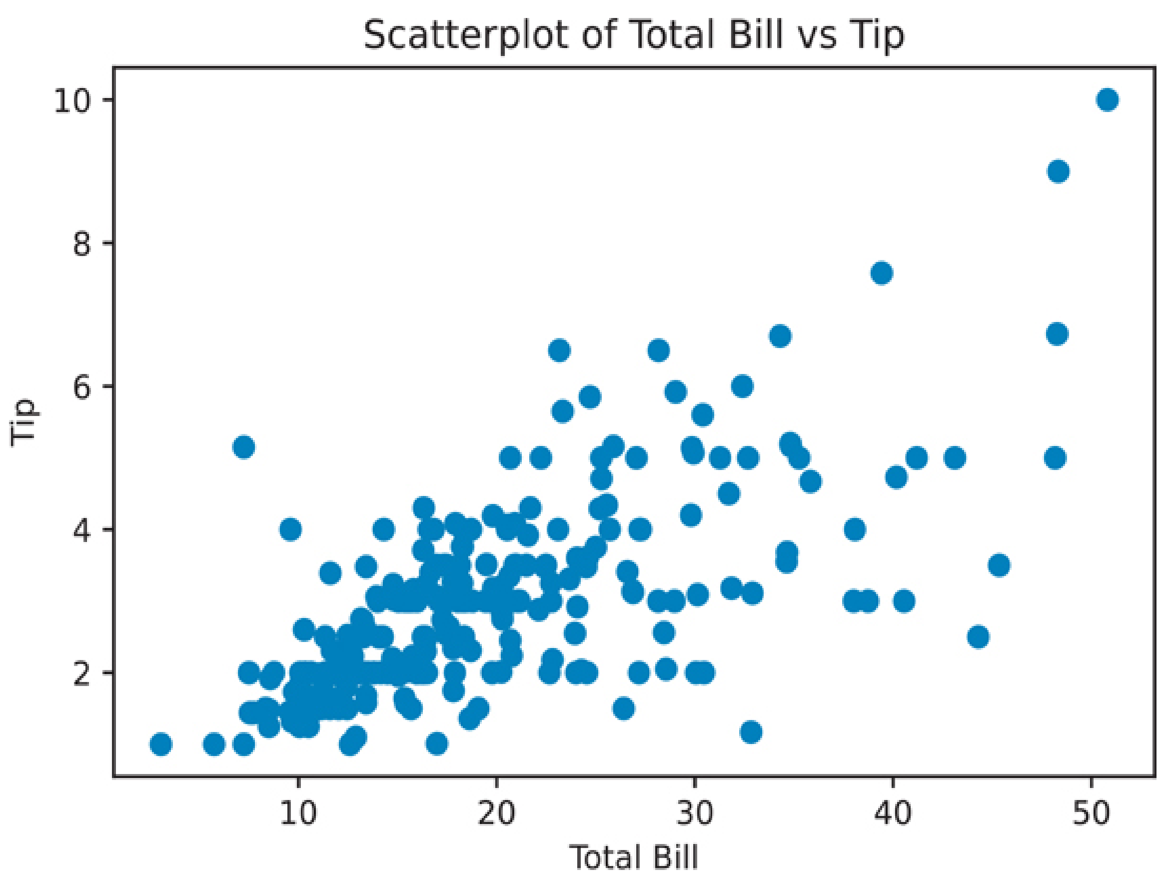


Figure 3.9 Scatter plot using matplotlib

Click here to view code image

```
# create the figure object
scatter_plot = plt.figure()
axes1 = scatter_plot.add_subplot(1, 1, 1)

# make the actual scatter plot
axes1.scatter(data=tips, x='total_bill',
y='tip')

# add labels
axes1.set_title('Scatterplot of Total Bill vs
Tip')
axes1.set_xlabel('Total Bill')
axes1.set_ylabel('Tip')

plt.show()
```

3.3.2.2 Box Plot

Box plots are used when a discrete variable is plotted against a continuous variable (Figure 3.10).

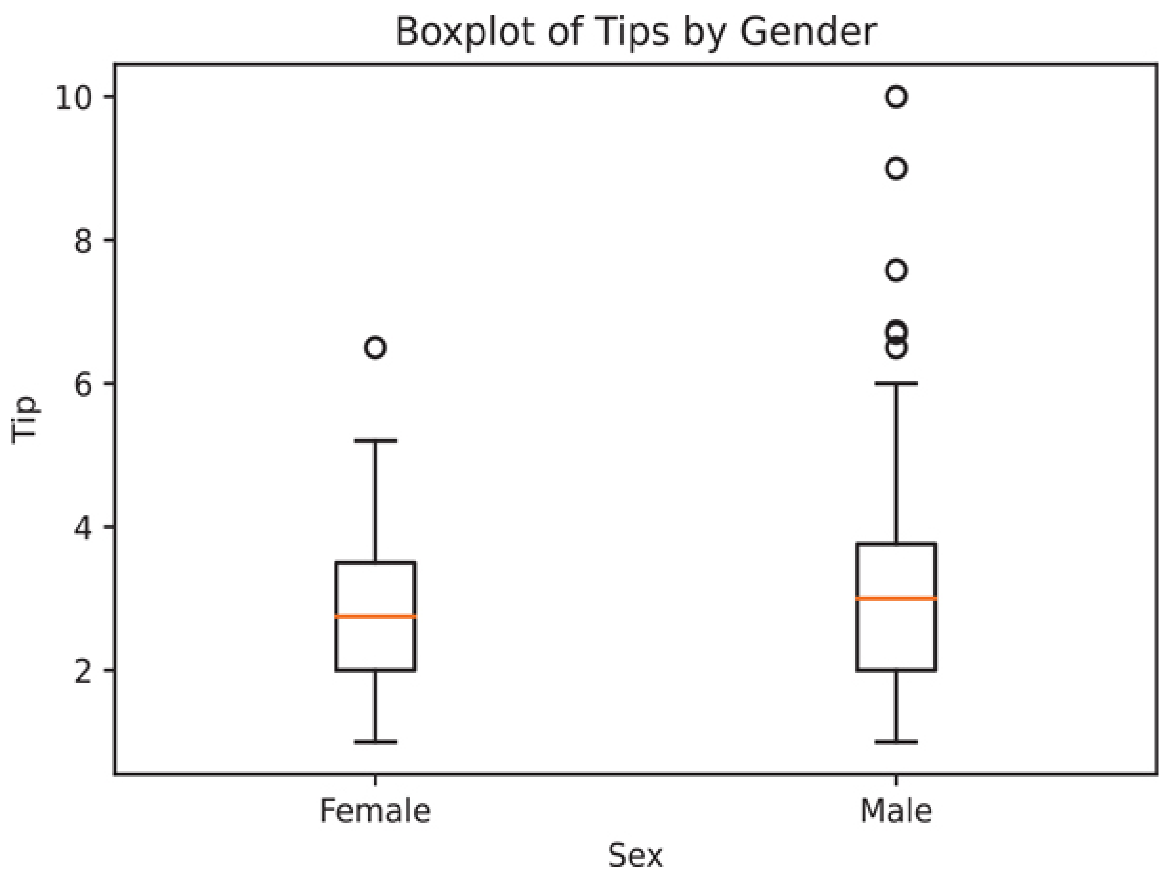


Figure 3.10 Box plot using matplotlib

> **Note**
>
> A discrete variable is usually something that is countable (using whole numbers). A continuous variable is usually a something that is measured and can have a decimal or fractional value.

Click here to view code image

```
# create the figure object
boxplot = plt.figure()
axes1 = boxplot.add_subplot(1, 1, 1)
```

Click here to view code image

```
# make the actual box plot
axes1.boxplot(
  # first argument of box plot is the data
  # since we are plotting multiple pieces of
data
  # we have to put each piece of data into a
list
  x=[
      tips.loc[tips["sex"] == "Female", "tip"],
      tips.loc[tips["sex"] == "Male", "tip"],
  ],
  # we can then pass in an optional labels
parameter
  # to label the data we passed
  labels=["Female", "Male"],
)


# add labels
axes1.set_xlabel('Sex')
axes1.set_ylabel('Tip')
axes1.set_title('Boxplot of Tips by Gender')

plt.show()
```

3.3.3 Multivariate Data

Plotting multivariate data is tricky because there is not a panacea or template that can be used for every case. To illustrate the process of plotting multivariate data, let's build on our earlier scatter plot.

If we wanted to add another variable, say `sex`, one option would be to color the points based on the value of the third variable. If we wanted to add a fourth variable, we could add size to the dots. The only caveat with using size as a variable is that humans are not very good at visually differentiating areas. Sure, if there's an enormous dot next to a tiny one, the relationship will be conveyed. But smaller differences are difficult to distinguish and may add clutter to your visualization. One way to reduce clutter is to add some value of transparency to the individual points, such that many overlapping points will show a darker region of a plot than less crowded areas.

A general convention is that different colors are much easier to distinguish than changes in size. If you have to use areas to convey differences in values, be sure that you are actually plotting relative areas. A common pitfall is to map a value to the radius of a circle for plots, but since the formula for a circle is πr^2, your areas are actually based on a squared scale. That is not only misleading but wrong.

Colors are also difficult to pick. Humans do not perceive hues on a linear scale, so you need to think carefully when picking color palettes. Luckily `matplotlib`[2] and `seaborn`[3] come with their own set of color palettes. Tools like `colorbrewer`[4] can help you pick good color palettes.

2. `matplotlib` colormaps: `https://matplotlib.org/stable/tutorials/colors/colormaps.html`

3. `seaborn` color palettes: `https://seaborn.pydata.org/tutorial/color_palettes.html`

4. `colorbrewer` color palettes: `http://colorbrewer2.org/`

Figure 3.11 uses color to add a third variable, `sex`, to our scatter plot. Since our values for `sex` only contain 2 values: `Male` and `Female`, we need to "map" the values to a color.

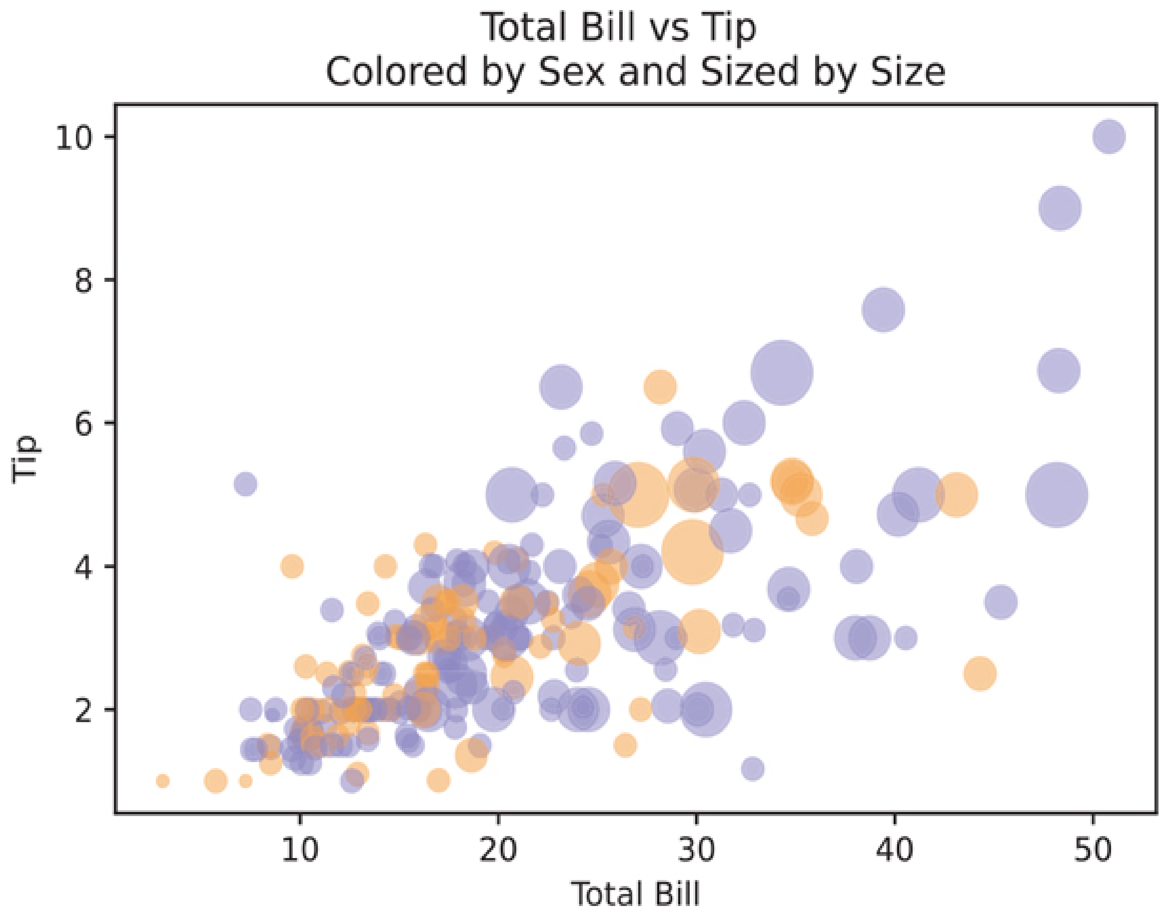


Figure 3.11 Matplotlib scatter plot with sex for the point color and size as point size

Click here to view code image

```
# assign color values
colors = {
    "Female": "#f1a340",  # orange
    "Male": "#998ec3",    # purple
}

scatter_plot = plt.figure()
axes1 = scatter_plot.add_subplot(1, 1, 1)

axes1.scatter(
```

```
  data=tips,
  x='total_bill',
  y='tip',

  # set the size of the dots based on party size
  # we multiply the values by 10 to make the
points bigger
  # and also to emphasize the difference
  s=tips["size"] ** 2 * 10,


  # set the color for the sex using our color
values above
  c=tips['sex'].map(colors),

  # set the alpha so points are more transparent
  # this helps with overlapping points
  alpha=0.5
)

# label the axes
axes1.set_title('Colored by Sex and Sized by
Size')
axes1.set_xlabel('Total Bill')
axes1.set_ylabel('Tip')

# figure title on top
scatter_plot.suptitle("Total Bill vs Tip")

plt.show()
```

`matplotlib` is an imperative plotting library. We'll see how other declarative plotting libraries allow us to make exploratory plots.

3.4 Seaborn

matplotlib is a core plotting tool in Python. seaborn builds on matplotlib by providing a higher-level declarative interface for statistical graphics. It gives us the ability to create more complex visualizations with fewer lines of code. The seaborn library is tightly integrated with the pandas library and the rest of the PyData stack (numpy, scipy, statsmodels, etc.), making visualizations from any part of the data analysis easier. Since seaborn is built on top of matplotlib, the user can still fine-tune the visualizations.

We've already loaded the seaborn library to access its data sets.

Click here to view code image

```
# load seaborn if you have not done so already
import seaborn as sns

tips = sns.load_data set("tips")
```

You will be able to look up all the seaborn plotting function documentation from the official seaborn site and then going to the API reference.[5]

5. seaborn website: https://seaborn.pydata.org/

For print, we are also going to set the "paper" context, to change some of the default font size, line width, axis tics, etc.

Click here to view code image

```
# set the default seaborn context optimized for
paper print
# the default is "notebook"
sns.set_context("paper")
```

3.4.1 Univariate

Just like we did with the `matplotlib` examples, we will make a series of univariate plots.

3.4.1.1 Histogram

Histograms are created using `sns.histplot()` (Figure 3.12).

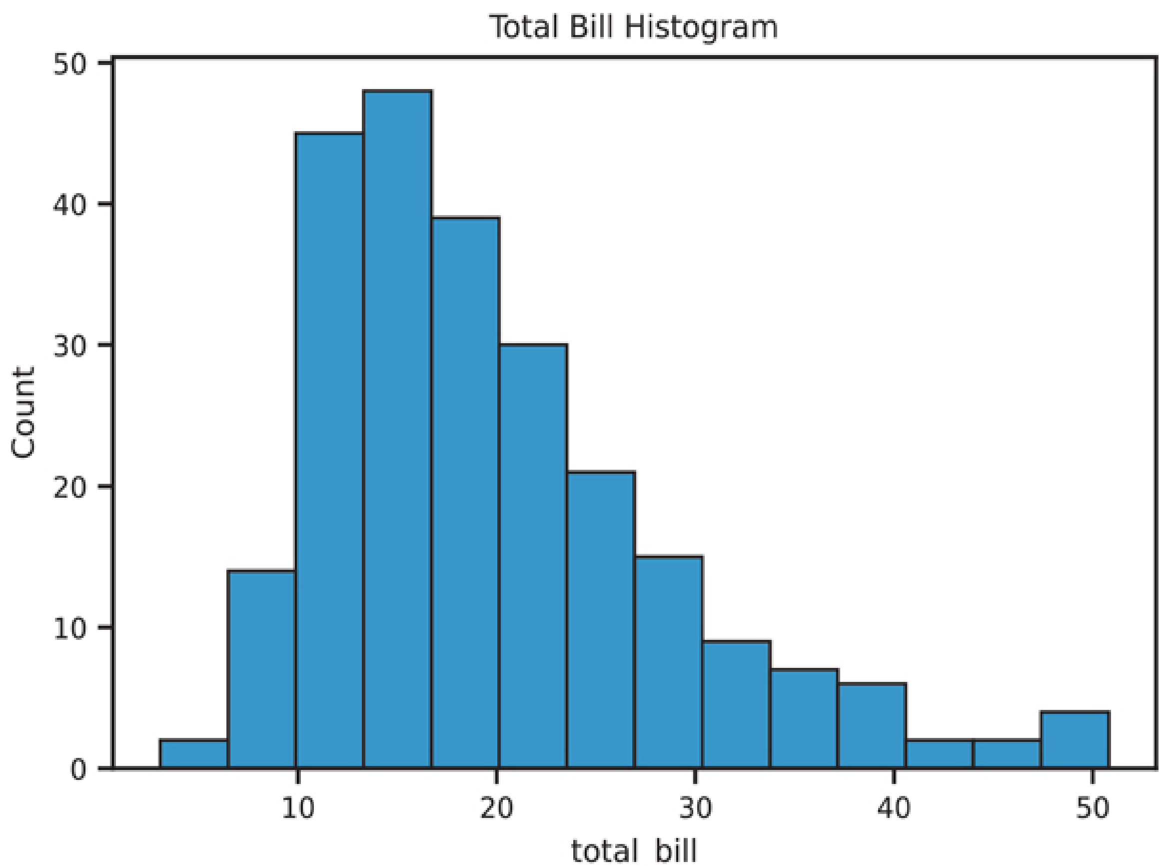


Figure 3.12 Seaborn histplot

Instead of two separate steps of creating an empty figure, and then specifying the individual axes subplots, We can create the figure with all the axes in a single step with the `subplots()` function. By default it will return two things back. The first thing will be the figure object, the second will be all the axes objects. We can then use the Python multiple assignment syntax to assign the parts to variables in a single step (Appendix Q).

From there we can use the `Figure` and `axes` objects just like before.

Click here to view code image

```
# the subplots function is a shortcut for
# creating separate figure objects and
# adding individual subplots (axes) to the
figure
hist, ax = plt.subplots()

# use seaborn to draw a histogram into the axes
sns.histplot(data=tips, x="total_bill", ax=ax)

# use matplotlib notation to set a title
ax.set_title('Total Bill Histogram')

# use matplotlib to show the figure
plt.show()
```

3.4.1.2 Density Plot (Kernel Density Estimation)

Density plots are another way to visualize a univariate distribution (Figure 3.13). In essence, they are created by drawing a normal distribution centered at each data point, then smoothing out the overlapping plots so that the area under the curve is 1.

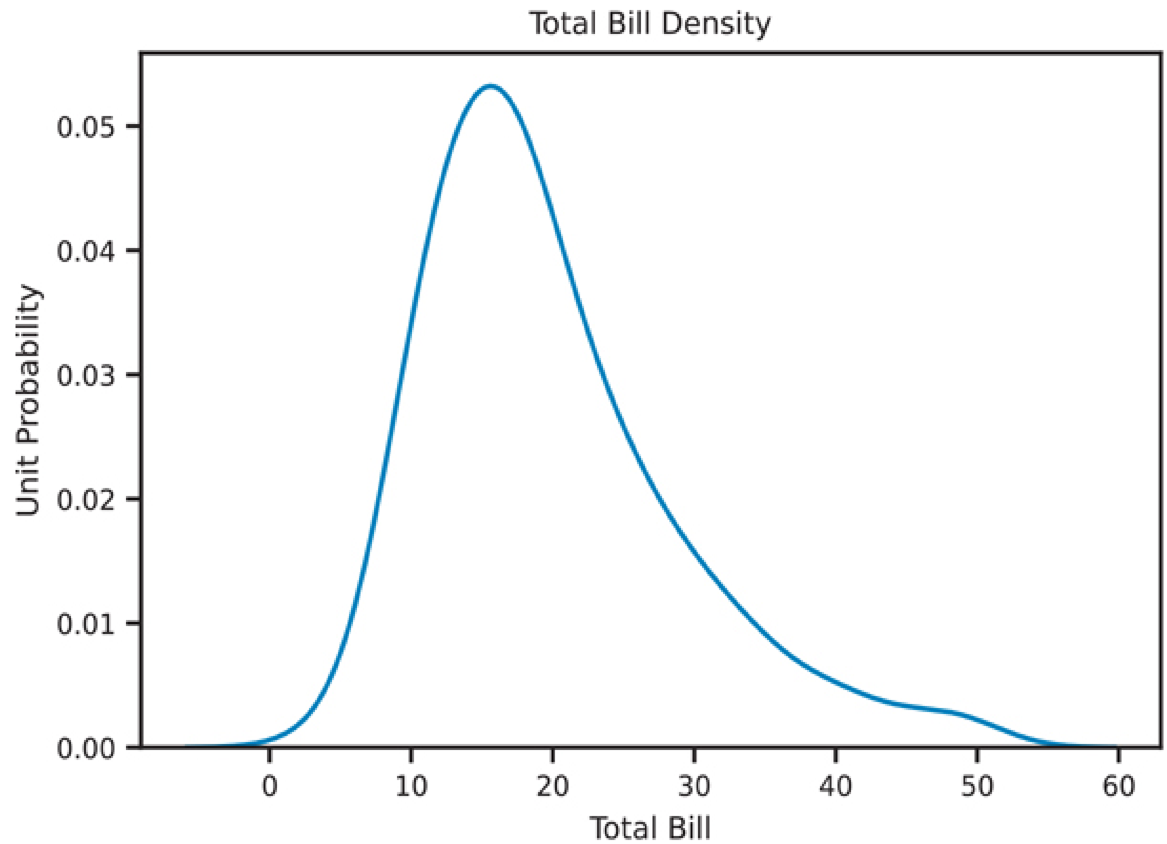


Figure 3.13 Seaborn kde plot

Click here to view code image

```
den, ax = plt.subplots()

sns.kdeplot(data=tips, x="total_bill", ax=ax)

ax.set_title('Total Bill Density')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Unit Probability')

plt.show()
```

3.4.1.3 Rug Plot

Rug plots are a one-dimensional representation of a variable's distribution. They are typically used with other plots to enhance a visualization. Figure 3.14 shows a histogram overlaid with a density plot and a rug plot on the bottom.

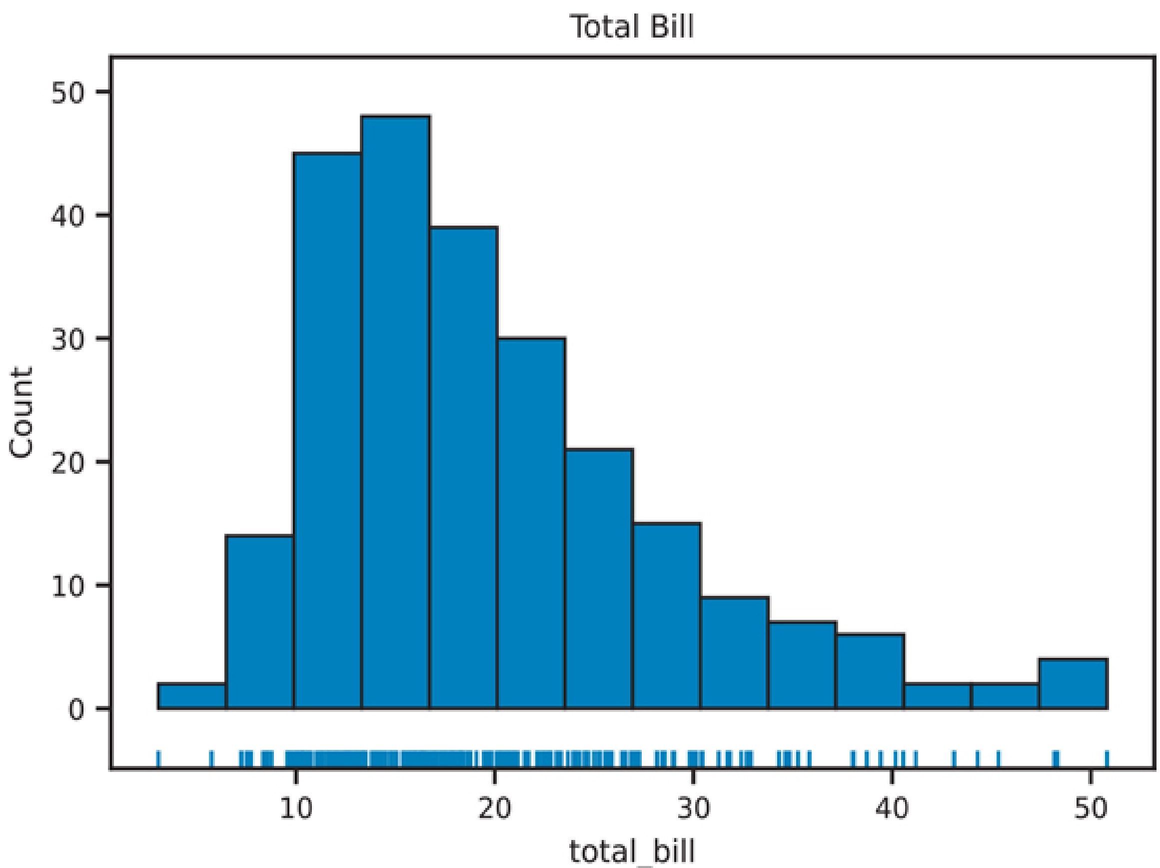


Figure 3.14 Seaborn rug plot with histogram

Click here to view code image

```
rug, ax = plt.subplots()

# plot 2 things into the axes we created
sns.rugplot(data=tips, x="total_bill", ax=ax)
sns.histplot(data=tips, x="total_bill", ax=ax)
```

Click here to view code image

```
ax.set_title("Rug Plot and Histogram of Total
Bill")
ax.set_title("Total Bill")

plt.show()
```

3.4.1.4 Distribution Plots

The newer `sns.displot()` function allows us to put together many of the univariate plots together into a single plot. This is the successor to the older `sns.distplot()` function (note the very subtle difference in spelling).

The `sns.displot()` function returns a `FacetGrid` object, not an `axes`, so the way we have been creating a figure and plotting the axes does not apply to this particular function. The benefit of it returning a more complex object is how it can plot multiple things at the same time. Figure 3.15 shows how we can combine many of the distribution figures into a single figure.

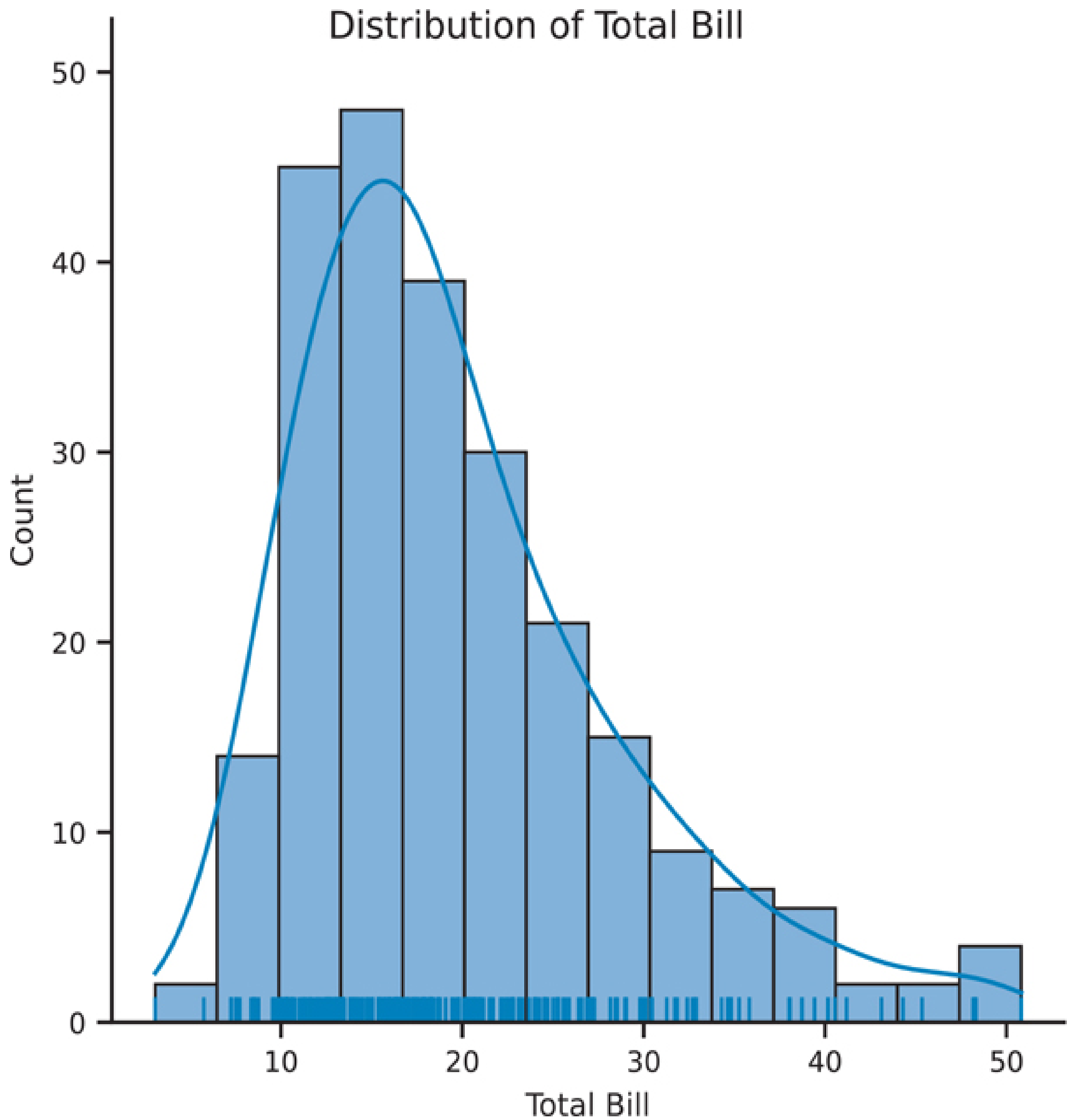


Figure 3.15 Seaborn distribution plot showing histogram, kde, and rug plots

Click here to view code image

```
# the FacetGrid object creates the figure and
axes for us
fig = sns.displot(data=tips, x="total_bill",
kde=True, rug=True)
```

```
fig.set_axis_labels(x_var="Total Bill",
y_var="Count")
fig.figure.suptitle('Distribution of Total
Bill')

plt.show()
```

3.4.1.5 Count Plot (Bar Plot)

Bar plots are very similar to histograms, but instead of binning values to produce a distribution, bar plots can be used to count discrete variables. Seaborn calls this a count plot (Figure 3.16).

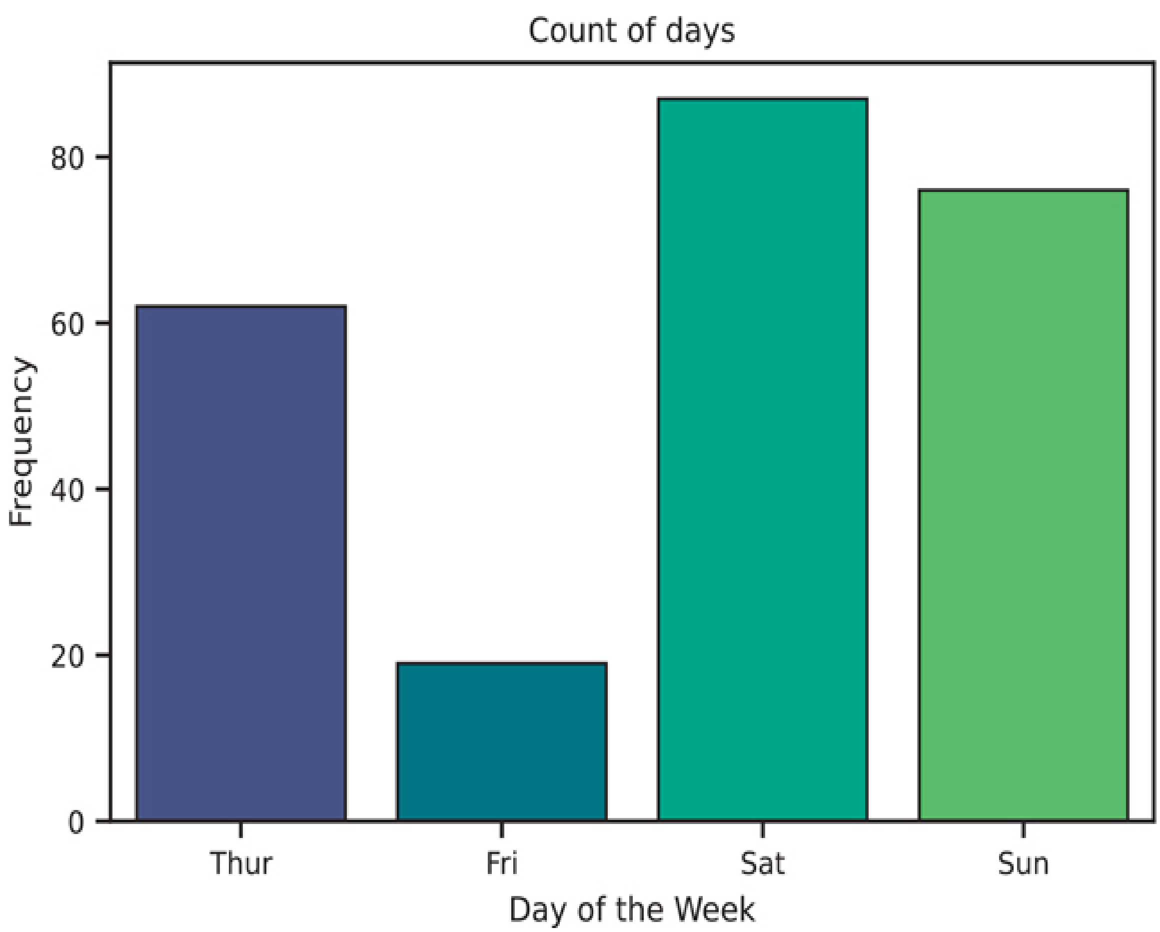


Figure 3.16 Seaborn count plot (i.e., bar plot) using the viridis color palette

Click here to view code image

```
count, ax = plt.subplots()

# we can use the viridis palette to help
distinguish the colors
sns.countplot(data=tips, x='day',
palette="viridis", ax=ax)

ax.set_title('Count of days')
ax.set_xlabel('Day of the Week')
ax.set_ylabel('Frequency')

plt.show()
```

Note

The viridis color palette was designed by Stéfan van der Walt and Nathaniel Smith to be colorblind friendly, and also be distinguishable in greyscale. They presented this color palette at the SciPy 2015 Conference, "A Better Default Colormap for Matplotlib" `https://www.youtube.com/watch?v=xAoljeRJ3lU`

3.4.2 Bivariate Data

We will now use the `seaborn` library to plot two variables.

3.4.2.1 Scatter Plot

There are a few ways to create a scatter plot in `seaborn`. The main difference is the type of object that gets created, an `Axes` or `FacetGrid` (i.e., type of `Figure`). `sns.scatterplot()` returns an `Axes` object (Figure 3.17).

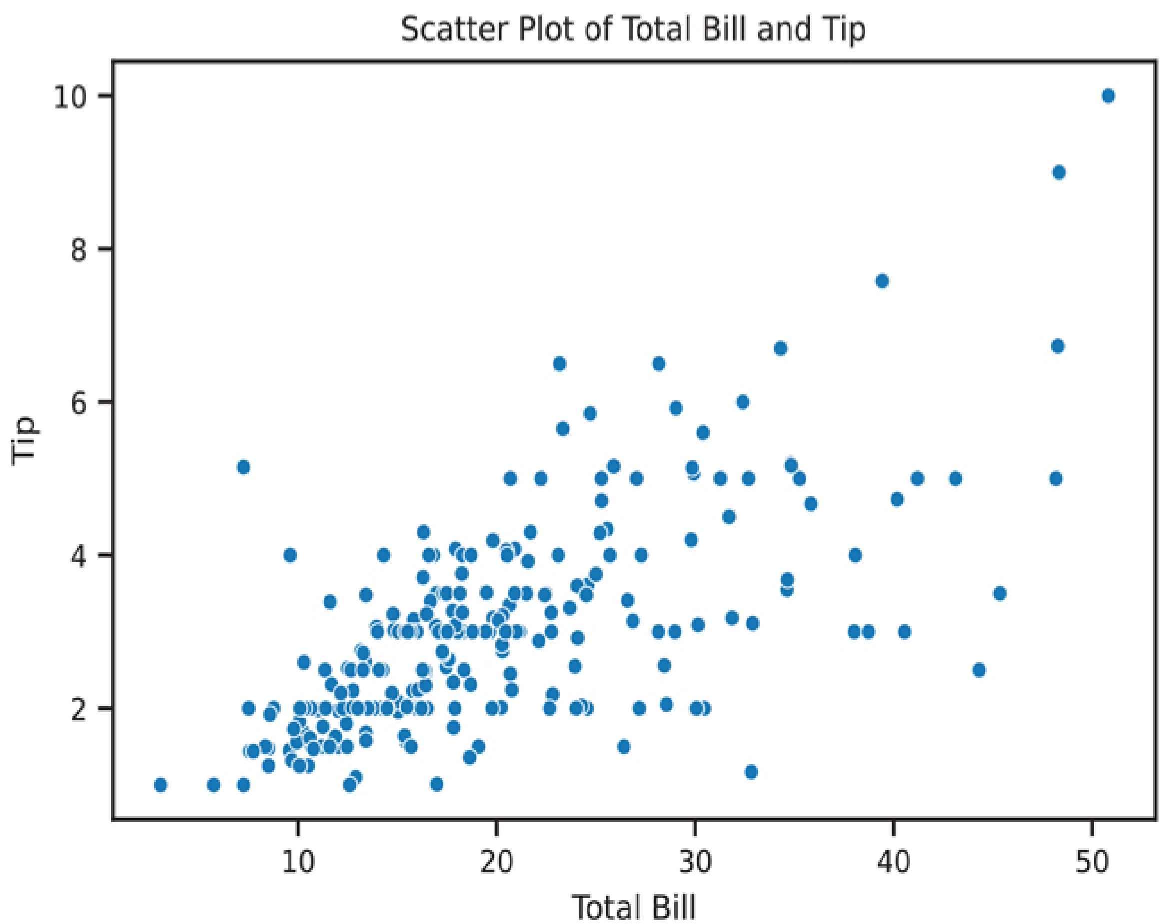


Figure 3.17 Seaborn scatter plot using `sns.scatterplot()`

Click here to view code image

```
scatter, ax = plt.subplots()

# use fit_reg=False if you do not want the
regression line
sns.scatterplot(data=tips, x='total_bill',
y='tip', ax=ax)
```

```
ax.set_title('Scatter Plot of Total Bill and
Tip')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Tip')

plt.show()
```

We can also use `sns.regplot()` to create a scatter plot and also draw a regression line (Figure 3.18).

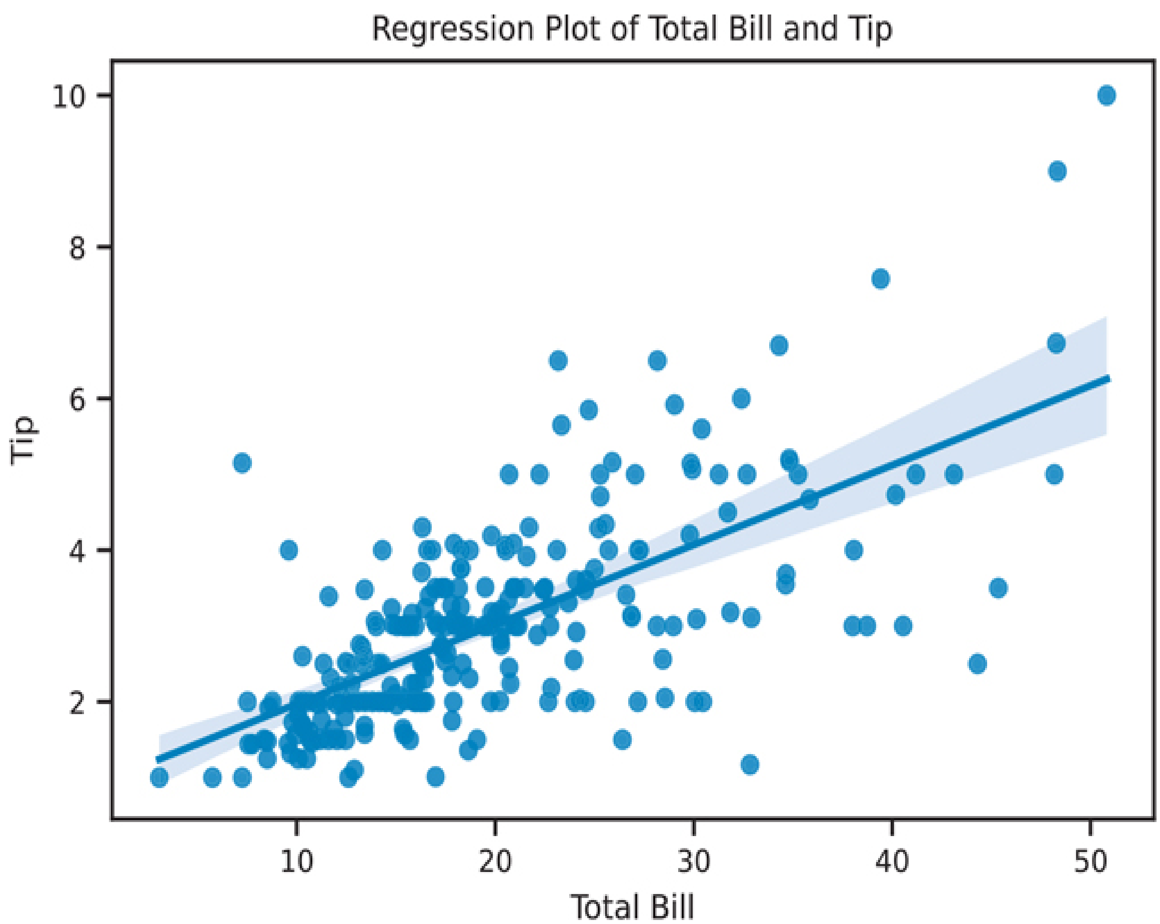


Figure 3.18 Seaborn scatter plot using `sns.regplot()`

Click here to view code image

```
reg, ax = plt.subplots()

# use fit_reg=False if you do not want the
regression line
sns.regplot(data=tips, x='total_bill', y='tip',
ax=ax)

ax.set_title('Regression Plot of Total Bill and
Tip')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Tip')

plt.show()
```

A similar function, `sns.lmplot()`, can also create scatter plots. Internally, `sns.lmplot()` calls `sns.regplot()`, so `sns.regplot()` is a more general plotting function. The main difference is that `sns.regplot()` creates an `axes` object whereas `sns.lmplot()` creates a `figure` object (See Section 3.2.2 for the parts of a figure). Figure 3.19 creates a scatter plot with a regression line, but creates the `figure` object directly, similar to the `FacetGrid` from `sns.displot()` in Section 3.4.1.4.

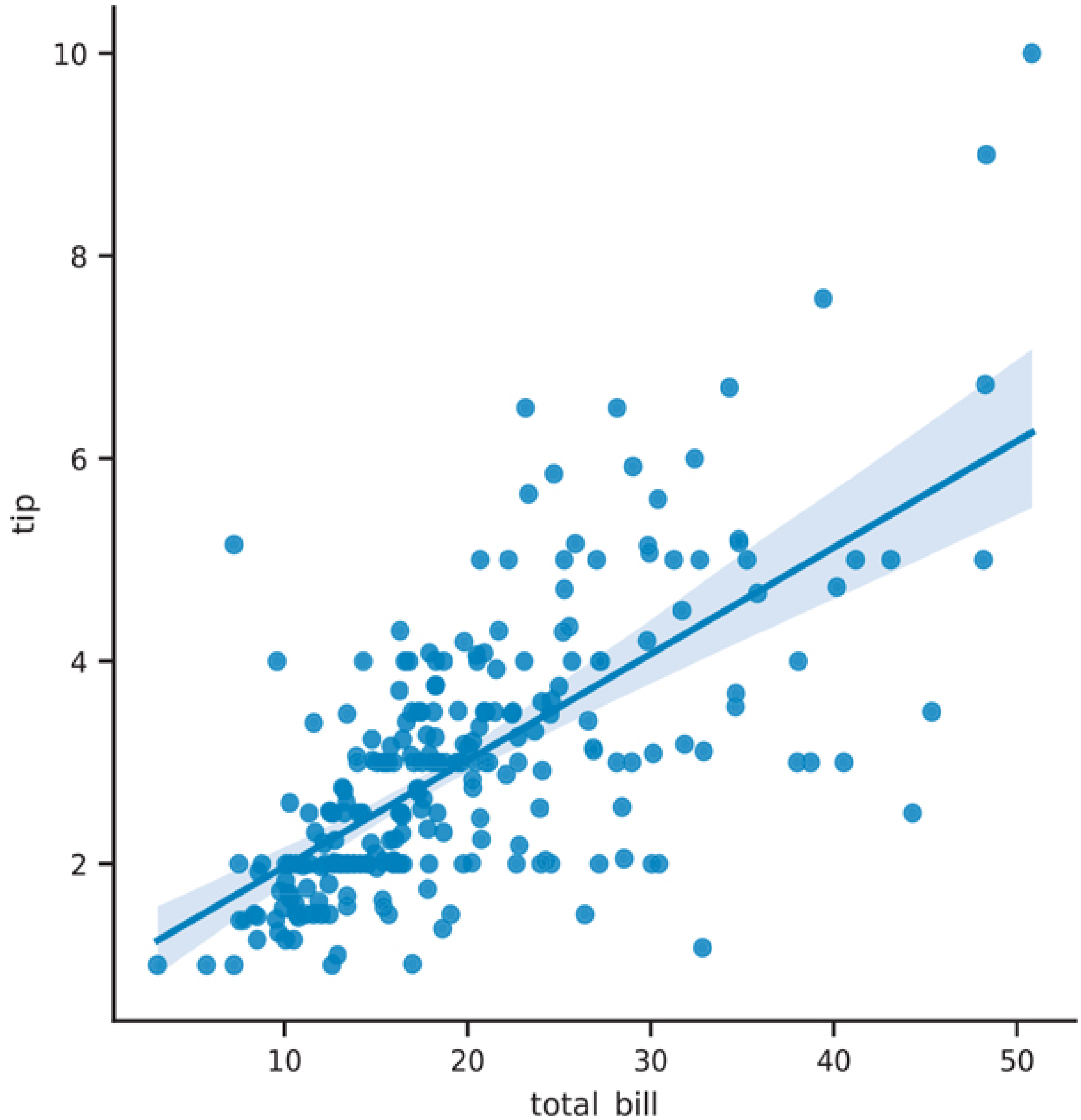


Figure 3.19 Seaborn scatter plot using `sns.lmplot()`

Click here to view code image

```
# use if you do not want the regression line
fig = sns.lmplot(data=tips, x='total_bill',
y='tip')

plt.show()
```

3.4.2.2 Joint Plot

We can also create a scatter plot that includes a univariate plot on each axis using `sns.jointplot()` (Figure 3.20). One major difference is that `sns.jointplot()` does not return axes, so we do not need to create a figure with axes on which to place our plot. Instead, this function creates a `JointGrid` object. If we need access to the base `matplotlib Figure` object, we use the `.figure` attribute.

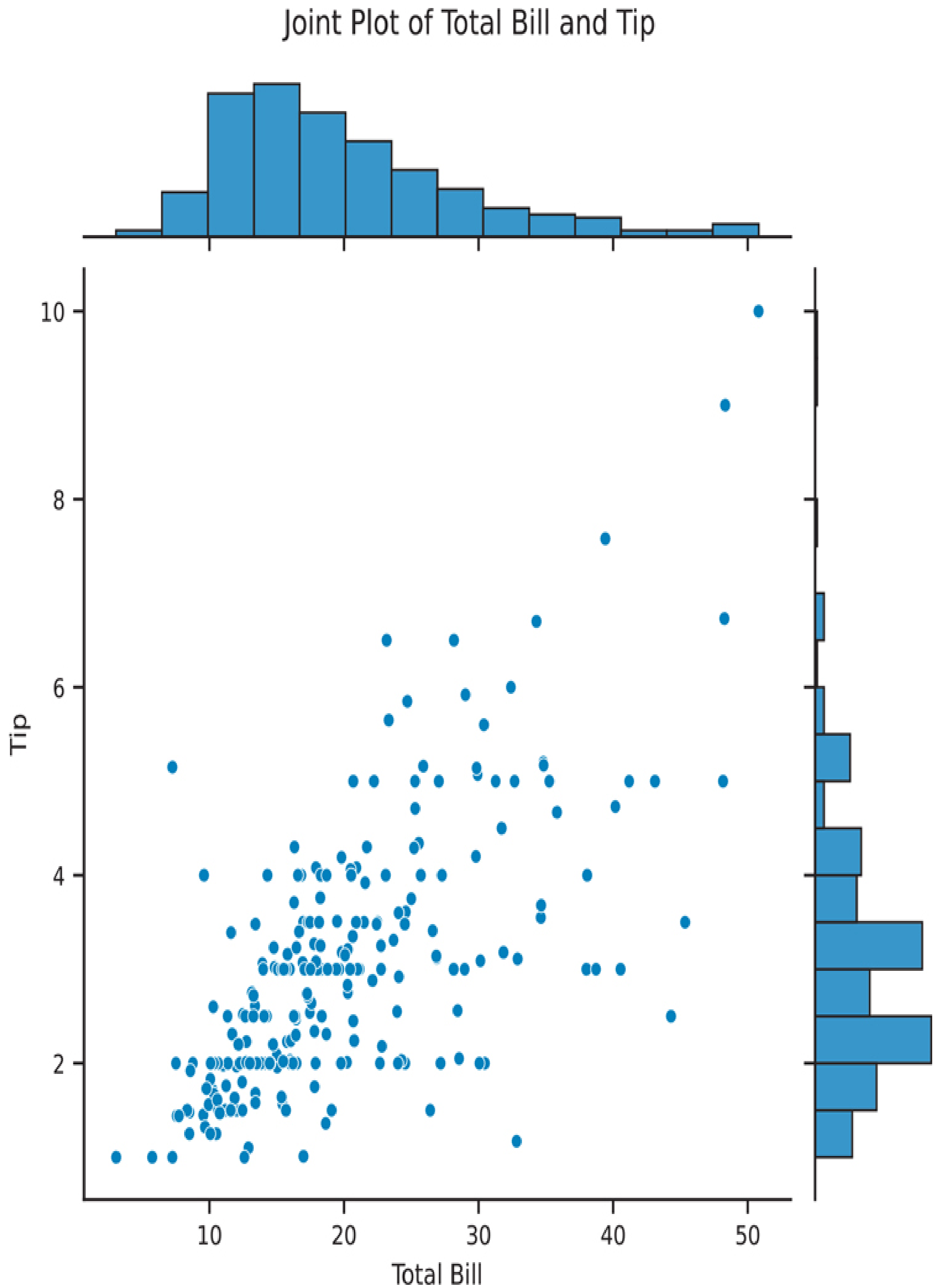


Figure 3.20 Seaborn scatter plot using `sns.jointplot()`

Click here to view code image

```
# jointplot creates the figure and axes for us
joint = sns.jointplot(data=tips, x='total_bill',
y='tip')

joint.set_axis_labels(xlabel='Total Bill',
ylabel='Tip')

# add a title and move the text up so it doesn't
clash with histogram
joint.figure.suptitle('Joint Plot of Total Bill
and Tip', y=1.03)

plt.show()
```

3.4.2.3 Hexbin Plot

Scatter plots are great for comparing two variables. However, sometimes there are too many points for a scatter plot to be meaningful. One way to get around this issue is to bin and aggregate nearby points on the plot together. Just as histograms can bin a variable to create a bar, hexbin plots can bin two variables (Figure 3.21). A hexagon is used for this purpose because it is the most efficient shape to cover an arbitrary 2D surface. This is an example of `seaborn` building on top of `matplotlib`, as `hexbin()` is a `matplotlib` function.

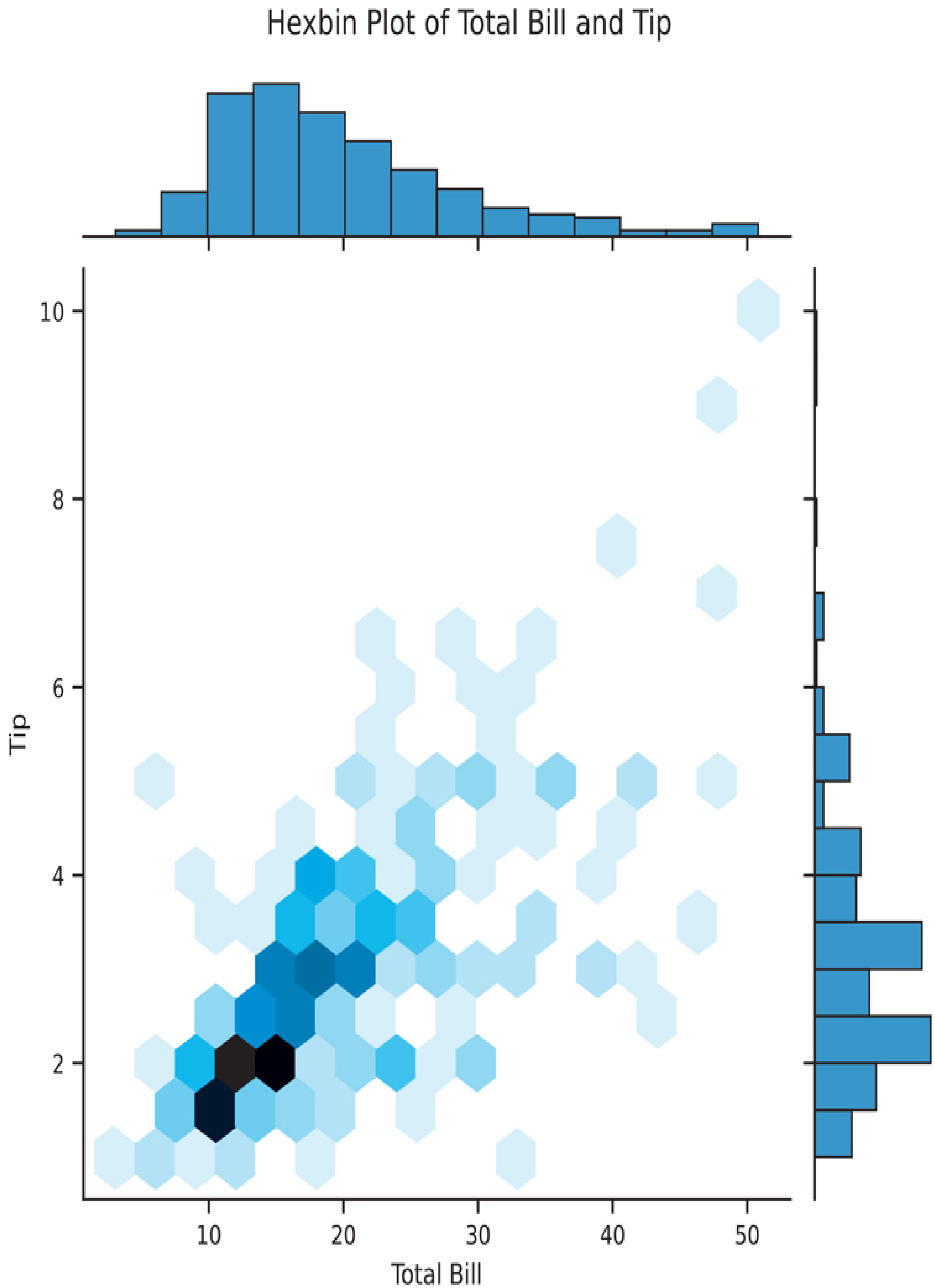


Figure 3.21 Seaborn hexbin plot using `sns.jointplot()`

Click here to view code image

```
# we can use jointplot with kind="hex" for a
hexbin plot
hexbin = sns.jointplot(
  data=tips, x="total_bill", y="tip", kind="hex"
)

hexbin.set_axis_labels(xlabel='Total Bill',
ylabel='Tip')
hexbin.figure.suptitle('Hexbin Plot of Total
Bill and Tip', y=1.03)

plt.show()
```

3.4.2.4 2D Density Plot

You can also create a 2D kernel density plot. This kind of process is similar to how `sns.kdeplot()` works, except it creates a density plot across two variables. The bivariate plot can be shown on its own (Figure 3.22).

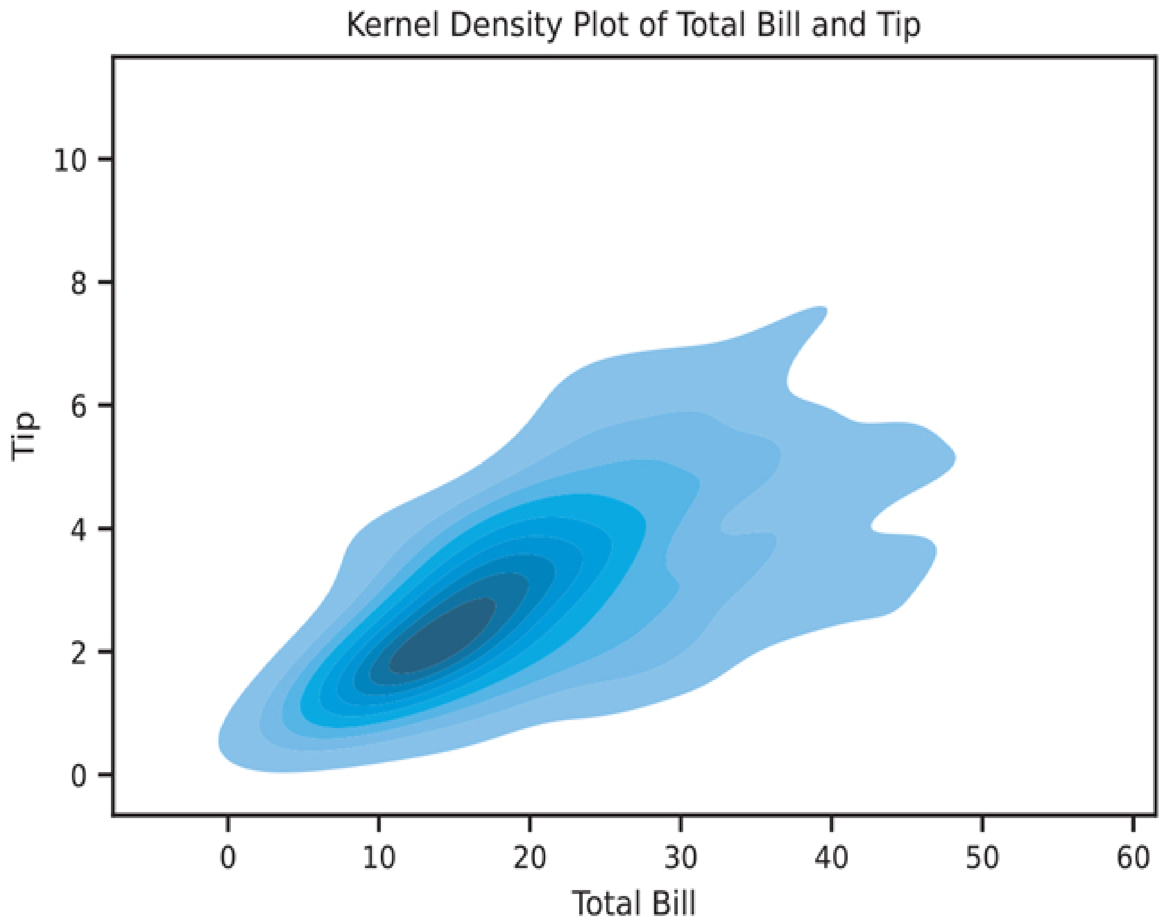


Figure 3.22 Seaborn KDE plot using `sns.kdeplot()`

Click here to view code image

```
kde, ax = plt.subplots()

# shade will fill in the contours
sns.kdeplot(data=tips, x="total_bill", y="tip",
shade=True, ax=ax)

ax.set_title('Kernel Density Plot of Total Bill
and Tip')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Tip')
```

```
plt.show()
```

`sns.jointplot()` will also allow us to create KDE plots (Figure 3.23).

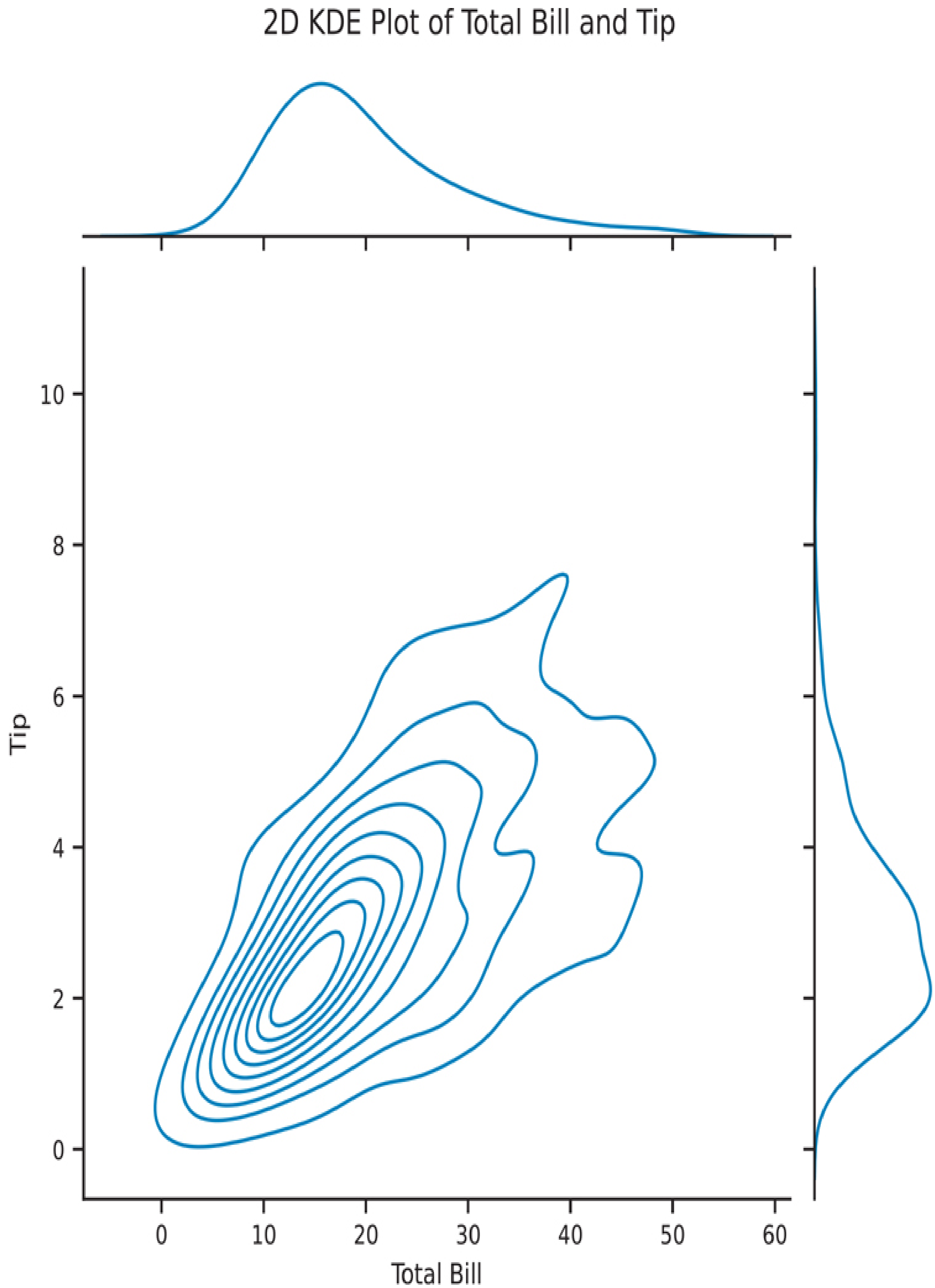


Figure 3.23 Seaborn KDE plot using `sns.jointplot()`

Click here to view code image

```
kde2d = sns.jointplot(data=tips, x="total_bill",
y="tip", kind="kde")

kde2d.set_axis_labels(xlabel='Total Bill',
ylabel='Tip')
kde2d.fig.suptitle('2D KDE Plot of Total Bill
and Tip', y=1.03)

plt.show()
```

3.4.2.5 Bar Plot

Bar plots can also be used to show multiple variables. By default, `sns.barplot()` will calculate a mean (Figure 3.24), but you can pass any function into the `estimator` parameter. For example, you could pass in the `np.mean()` function to calculate the mean using the version from the `numpy` library.

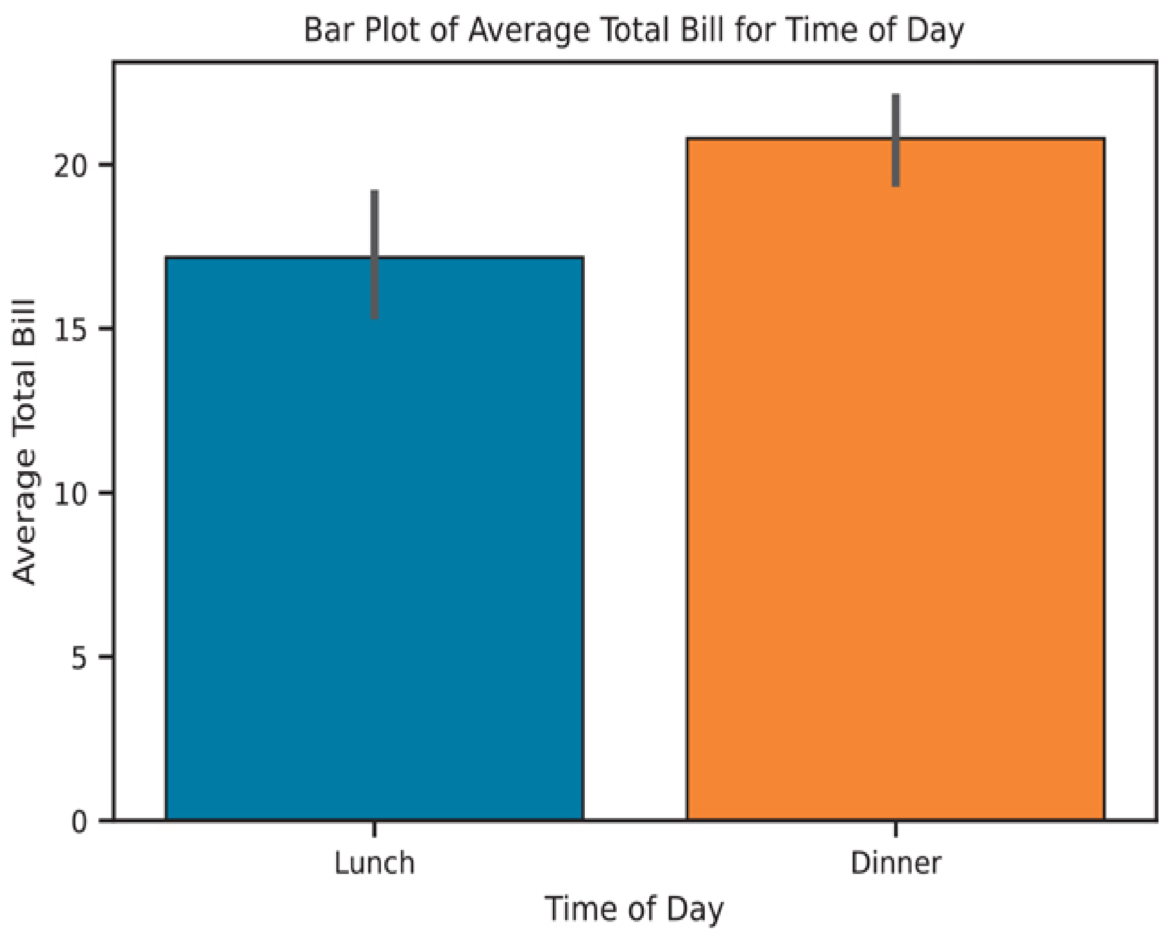


Figure 3.24 Seaborn bar plot using the `np.mean()` function

Click here to view code image

```
import numpy as np

bar, ax = plt.subplots()

# plot the average total bill for each value of
time
# mean is calculated using numpy
sns.barplot(
  data=tips, x="time", y="total_bill",
estimator=np.mean, ax=ax
)
```

```
ax.set_title('Bar Plot of Average Total Bill for
Time of Day')
ax.set_xlabel('Time of Day')
ax.set_ylabel('Average Total Bill')

plt.show()
```

3.4.2.6 Box Plot

Unlike the previously mentioned plots, a box plot (Figure 3.25) shows multiple statistics: the minimum, first quartile, median, third quartile, maximum, and, if applicable, outliers based on the interquartile range.

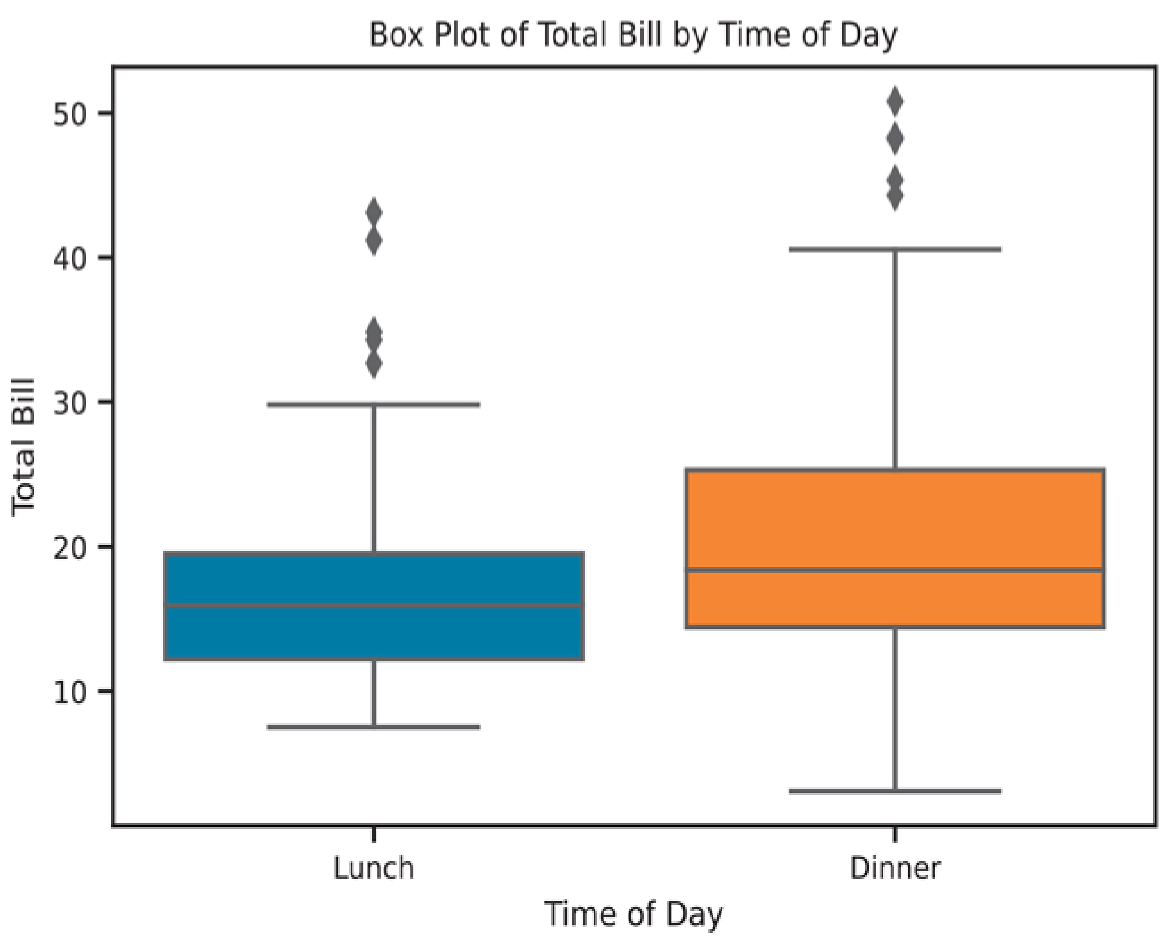


Figure 3.25 Seaborn box plot of total bill by time of day

The `y` parameter in `sns.boxplot()` is optional. If it is omitted, the plotting function will create a single box in the plot.

Click here to view code image

```
box, ax = plt.subplots()

# the y is optional, but x would have to be a
numeric variable
sns.boxplot(data=tips, x='time', y='total_bill',
ax=ax)

ax.set_title('Box Plot of Total Bill by Time of
Day')
ax.set_xlabel('Time of Day')
ax.set_ylabel('Total Bill')

plt.show()
```

3.4.2.7 Violin Plot

Box plots are a classical statistical visualization, but they can obscure the underlying distribution of the data. Violin plots (Figure 3.26) can show the same values as a box plot, but plot the "boxes" as a kernel density estimation. This can help retain more visual information about your data since only plotting summary statistics can be misleading, as seen by the Anscombe quartet (Section 3.2.1).

Figure 3.26 Seaborn violin plot of total bill by time of day

Click here to view code image

```
violin, ax = plt.subplots()

sns.violinplot(data=tips, x='time',
y='total_bill', ax=ax)

ax.set_title('Violin plot of total bill by time
of day')
ax.set_xlabel('Time of day')
ax.set_ylabel('Total Bill')

plt.show()
```

We can now see how the violin plot is related to the box plot. In Figure 3.27, we will create a single figure with 2 axes (i.e., subplots).

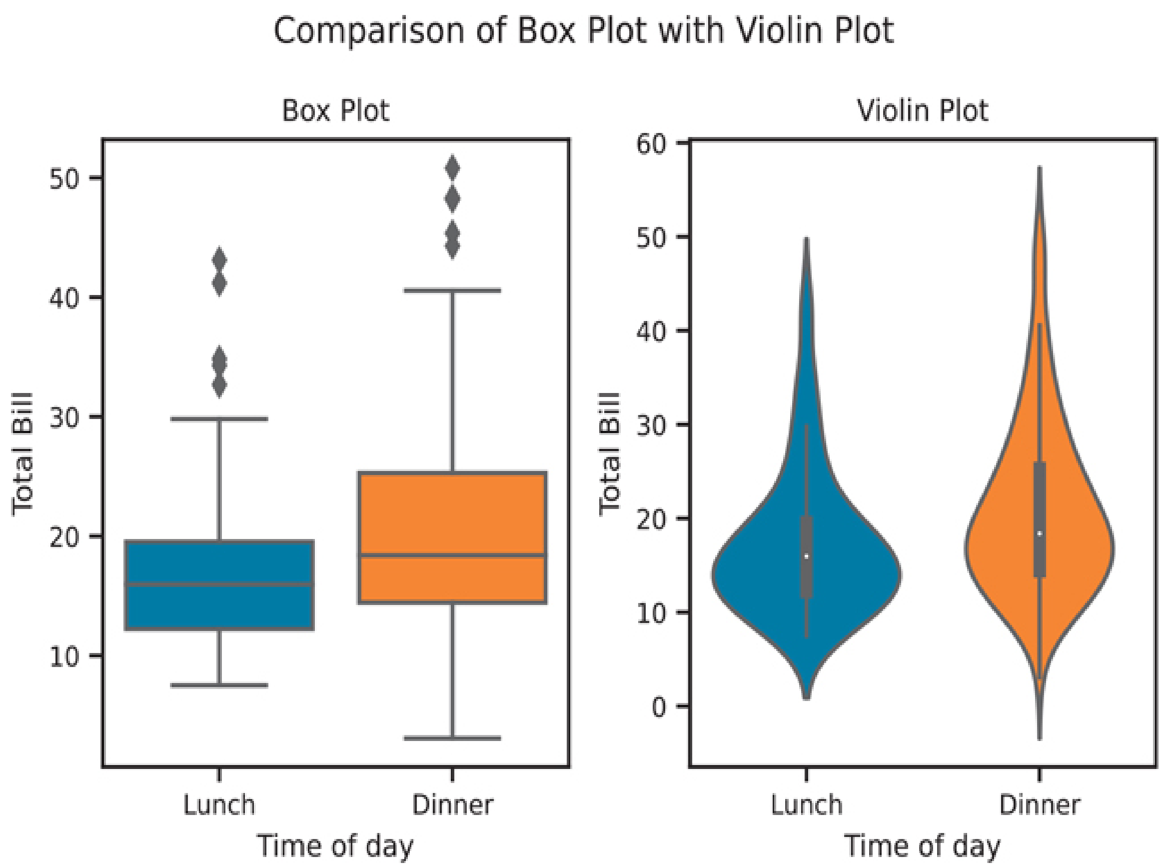


Figure 3.27 Comparing box plots with violin plots

Click here to view code image

```
# create the figure with 2 subplots
box_violin, (ax1, ax2) = plt.subplots(nrows=1,
ncols=2)

sns.boxplot(data=tips, x='time', y='total_bill',
ax=ax1)
sns.violinplot(data=tips, x='time',
y='total_bill', ax=ax2)
```

```
# set the titles
ax1.set_title('Box Plot')
ax1.set_xlabel('Time of day')
ax1.set_ylabel('Total Bill')

ax2.set_title('Violin Plot')
ax2.set_xlabel('Time of day')
ax2.set_ylabel('Total Bill')

box_violin.suptitle("Comparison of Box Plot with
Violin Plot")

# space out the figure so labels do not overlap
box_violin.set_tight_layout(True)

plt.show()
```

3.4.2.8 Pairwise Relationships

When you have mostly numeric data, visualizing all of the pairwise relationships can be performed using `sns.pairplot()`. This function will plot a scatter plot between each pair of variables, and a histogram for the univariate data (Figure 3.28).

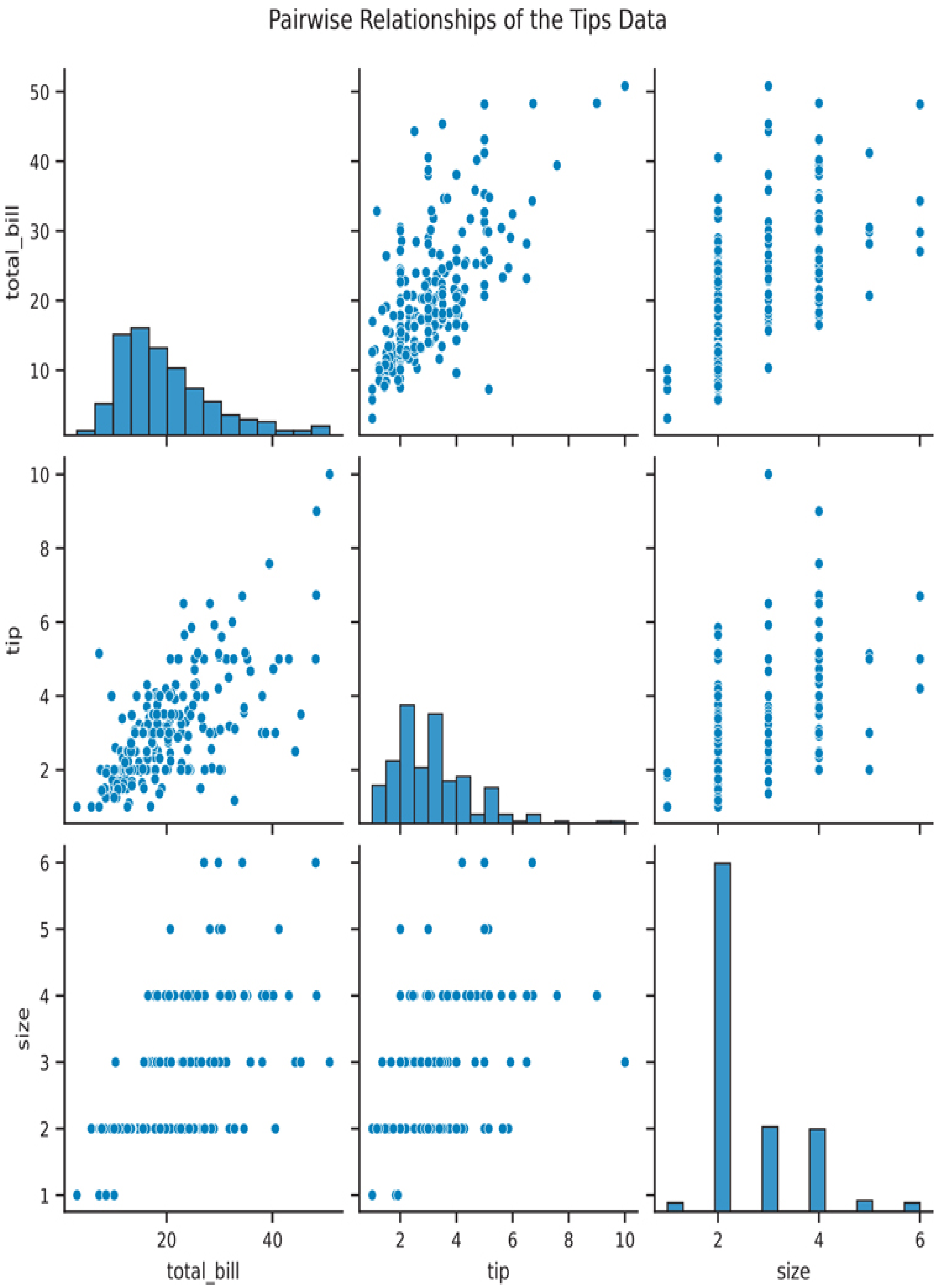


Figure 3.28 Seaborn pair plot

Click here to view code image

```
fig = sns.pairplot(data=tips)

fig.figure.suptitle(
  'Pairwise Relationships of the Tips Data',
y=1.03
)

plt.show()
```

One drawback when using `sns.pairplot()` is that there is redundant information; that is, the top half of the visualization is the same as the bottom half. We can use `sns.PairGrid()` to manually assign the plots for the top half and bottom half. This plot is shown in Figure 3.29.

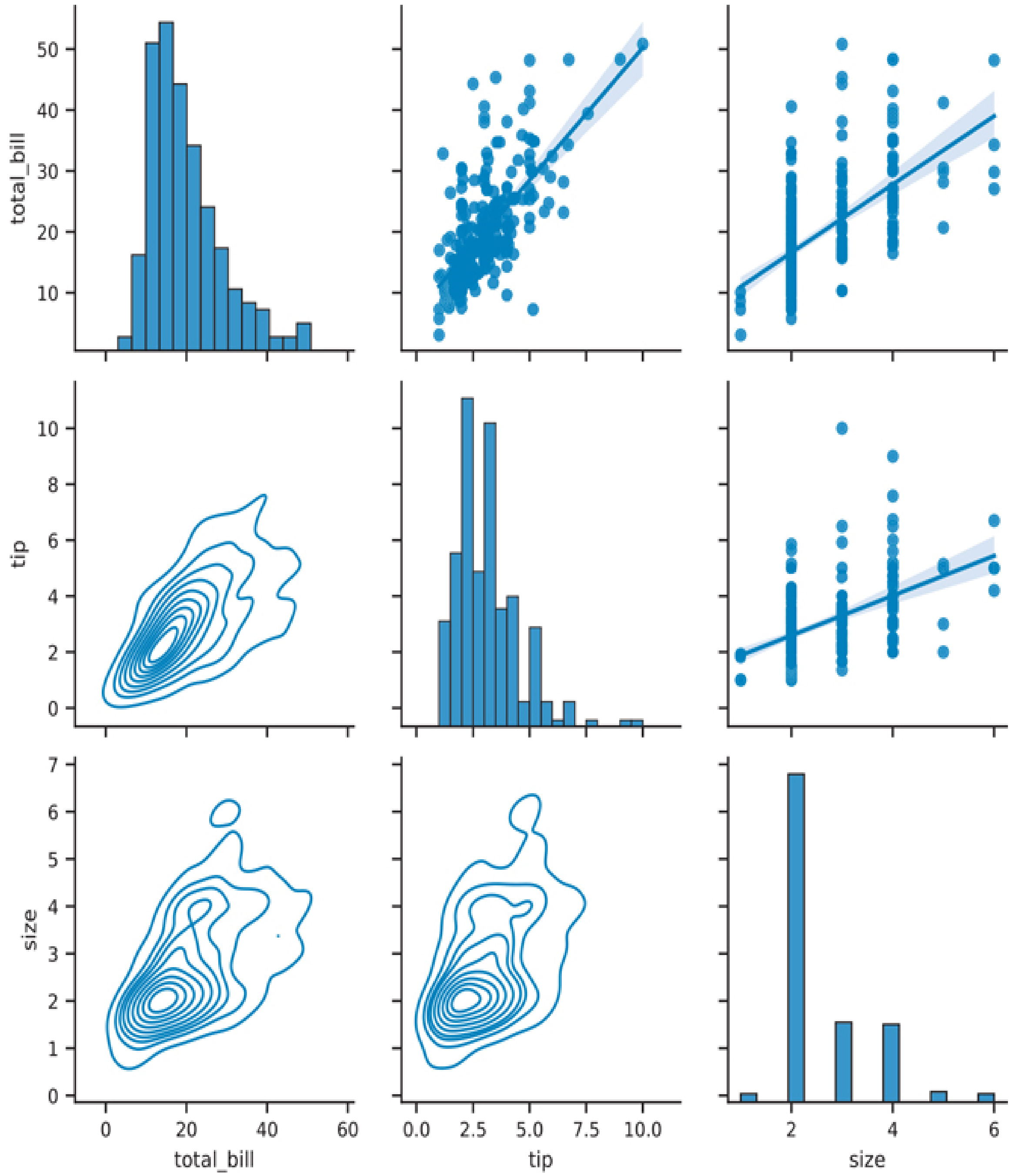


Figure 3.29 Seaborn pair plot with different plots on the upper and lower halves

Click here to view code image

```
# create a PairGrid, make the diagonal plots on
a different scale
pair_grid = sns.PairGrid(tips,
diag_sharey=False)

# set a separate function to plot the upper,
bottom, and diagonal
# functions need to return an axes, not a figure


# we can use plt.scatter instead of sns.regplot
pair_grid = pair_grid.map_upper(sns.regplot)
pair_grid = pair_grid.map_lower(sns.kdeplot)
pair_grid = pair_grid.map_diag(sns.histplot)

plt.show()
```

3.4.3 Multivariate Data

As mentioned in Section 3.3.3, there is no de facto template for plotting multivariate data. Possible ways to include more information are to use color, size, or shape to distinguish data within the plot.

3.4.3.1 Colors

When we are using `sns.violinplot()`, we can pass the `hue` parameter to color the plot by `sex`. We can reduce the redundant information by having each half of the violins represent a different `sex`, as shown in Figure 3.30. Try the following code with and without the `split` parameter.

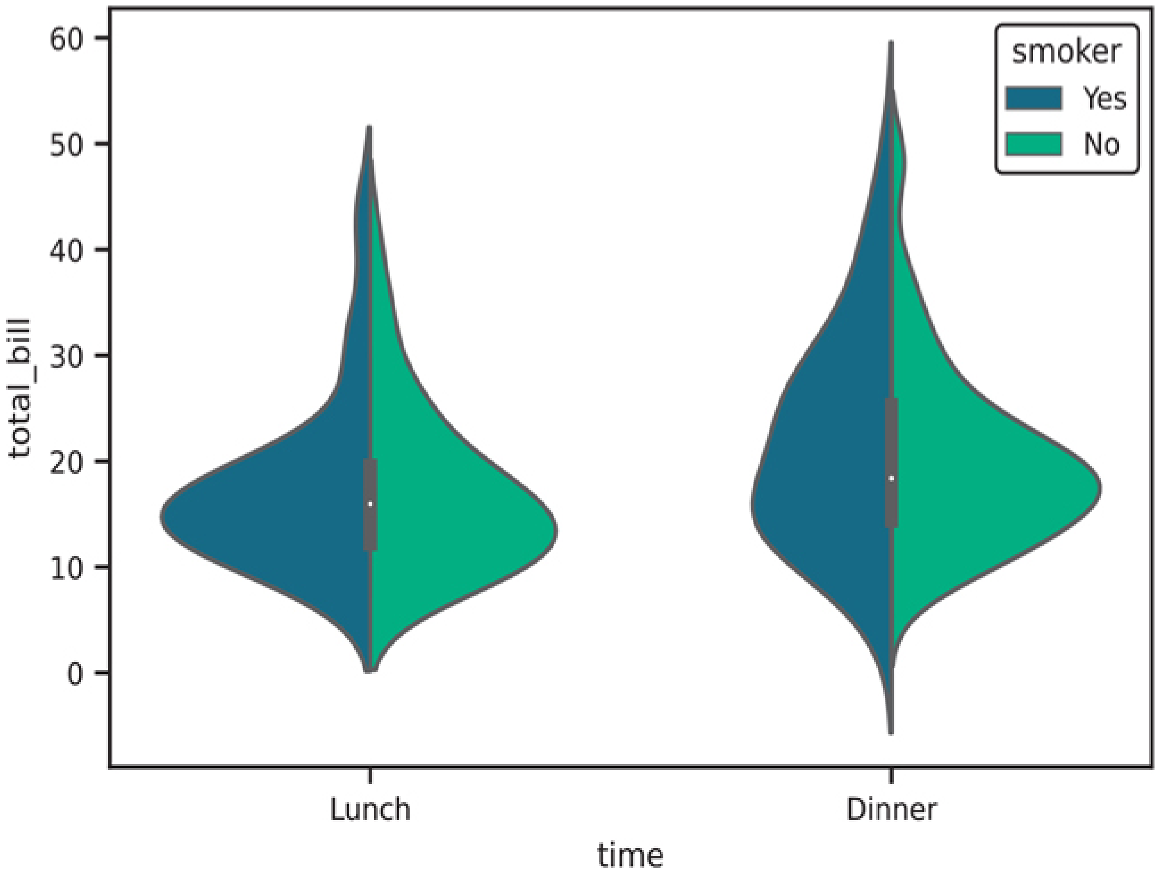


Figure 3.30 Seaborn violin plot with hue parameter

Click here to view code image

```
violin, ax = plt.subplots()

sns.violinplot(
  data=tips,
  x="time",
  y="total_bill",
  hue="smoker", # set color based on smoker
variable
  split=True,
  palette="viridis", # palette specifies the
colors for hue
```

```
    ax=ax,
)

plt.show()
```

The `hue` parameter can be passed into various other plotting functions as well. Figure 3.31 shows its use in a `sns.lmplot()`.

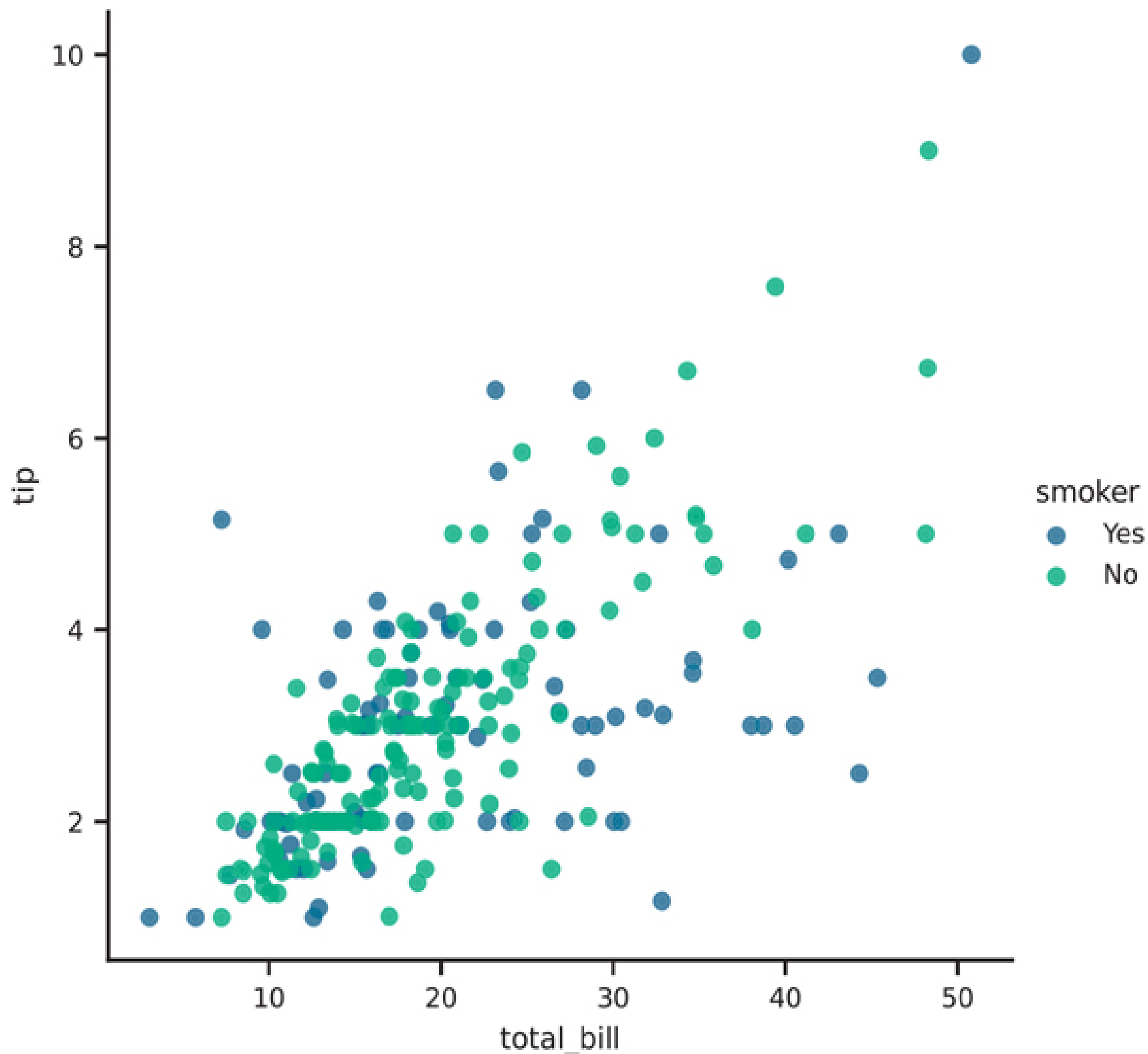


Figure 3.31 Seaborn lmplot plot with hue parameter

Click here to view code image

```
# note the use of lmplot instead of regplot to
return a figure
scatter = sns.lmplot(
  data=tips,
  x="total_bill",
  y="tip",
  hue="smoker",
  fit_reg=False,
  palette="viridis",
)

plt.show()
```

We can make our pairwise plots a little more meaningful by passing one of the categorical variables as the `hue` parameter. Figure 3.32 shows this approach in our `sns.pairplot()`.

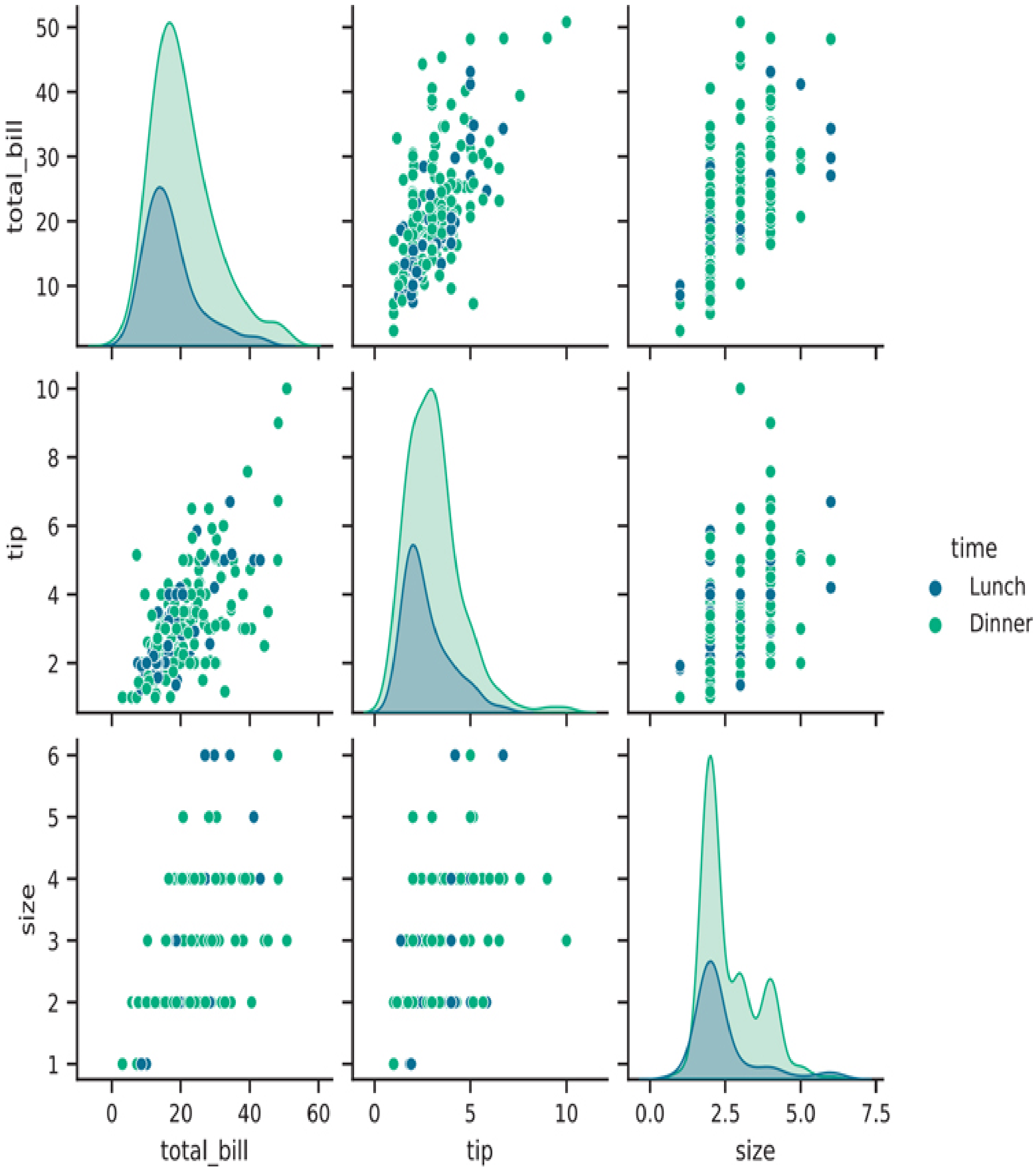


Figure 3.32 Seaborn pair plot with hue parameter

Click here to view code image

```
fig = sns.pairplot(
  tips,
  hue="time",
```

```
  palette="viridis",

  height=2, # facet height to make the entire
figure smaller
)

plt.show()
```

3.4.3.2 Size and Shape

Working with point sizes can be another means of adding more information to a plot. However, this option should be used sparingly, since the human eye is not very good at comparing areas. Figure 3.33 shows using the `hue` for color and `size` for point sizes in the `sns.scatterplot()` function.

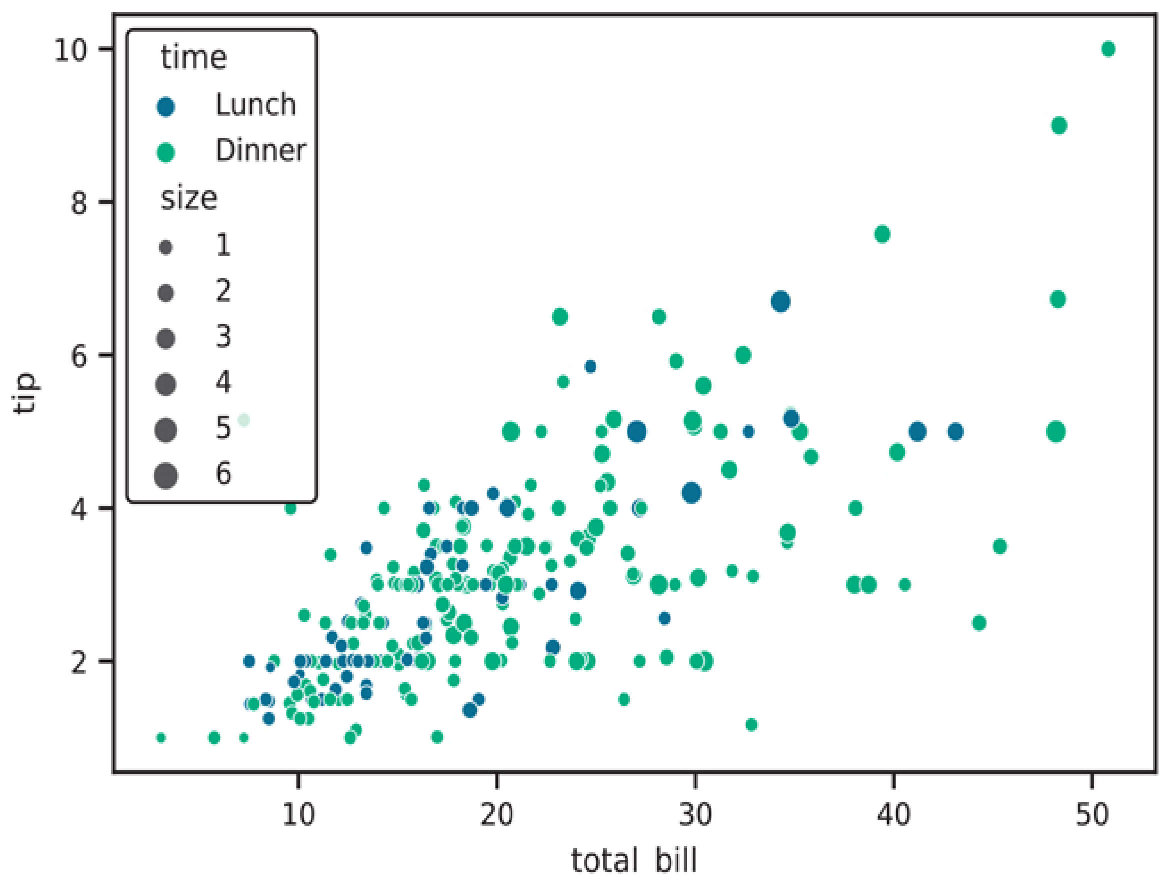


Figure 3.33 Scatter plot of tip vs total bill, colored by time of day, and sized by table size

Click here to view code image

```
fig, ax = plt.subplots()

sns.scatterplot(
  data=tips,
  x="total_bill",
  y="tip",
  hue="time",
  size="size",
  palette="viridis",
  ax=ax,
)

plt.show()
```

3.4.4 Facets

What if we want to show more variables? Or if we know which plot we want for our visualization, but we want to make multiple plots over a categorical variable? Facets are designed to meet these needs. Instead of individually subsetting data and lay out the axes in a figure (as we did in Figure 3.5), facets in `seaborn` can handle this work for you.

To use facets, your data needs to be what Hadley Wickham[6] calls "Tidy Data,"[7] where each row represents an observation in the data, and each column is a variable. More about tidy data is discussed in Chapter 4.

6. Hadley Wickham, PhD: http://hadley.nz

7. Tidy Data paper: http://vita.had.co.nz/papers/tidy-data.pdf

3.4.4.1 One Facet Variable

Figure 3.34 shows a re-creation of the Anscombe quartet data from Figure 3.5 in `seaborn`. The trick to faceted plots in `seaborn` is to look for the `col` or `row` parameter in the plotting function. Here, we use `sns.relplot()` to make our faceted scatter plot (the `sns.scatterplot()` documentation also points to use `sns.relplot()` for facets).

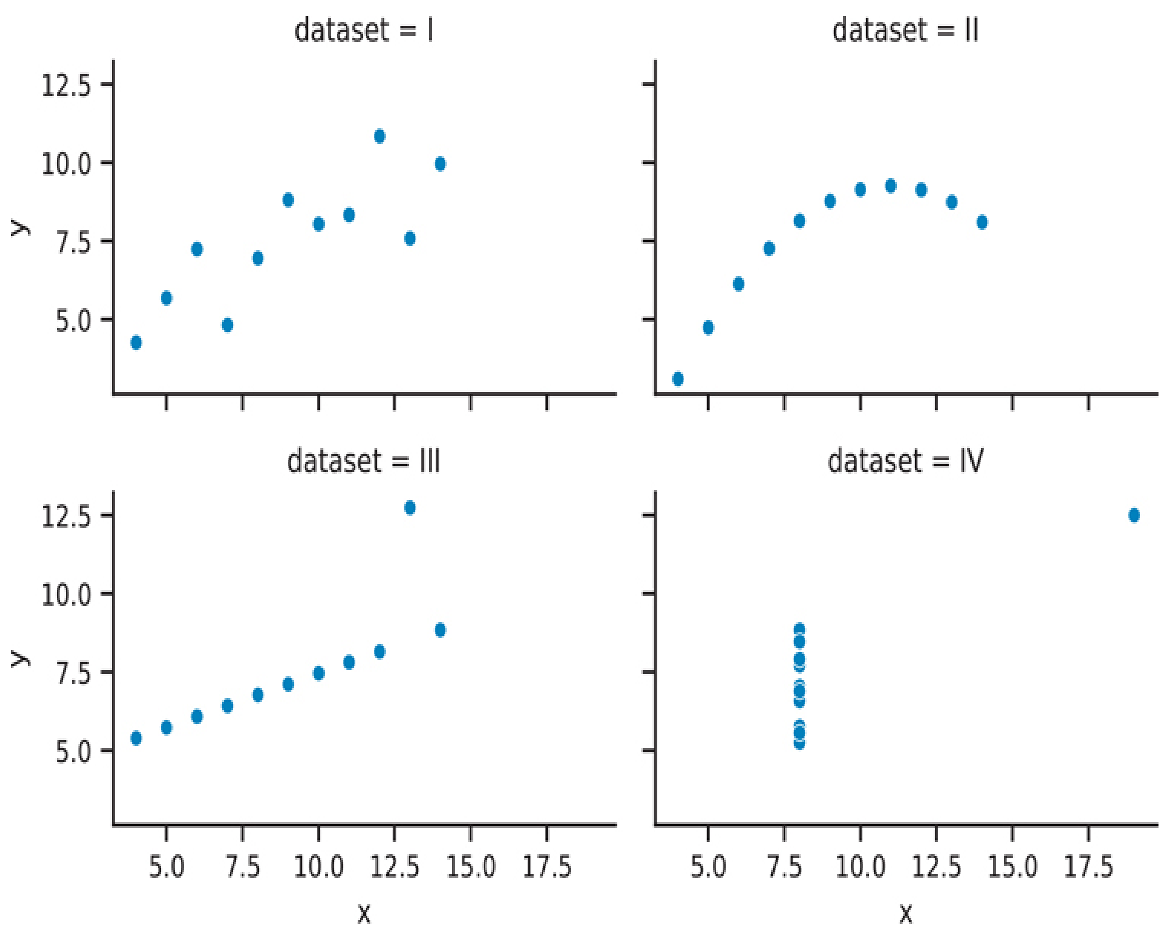


Figure 3.34 Seaborn Anscombe plot with facets

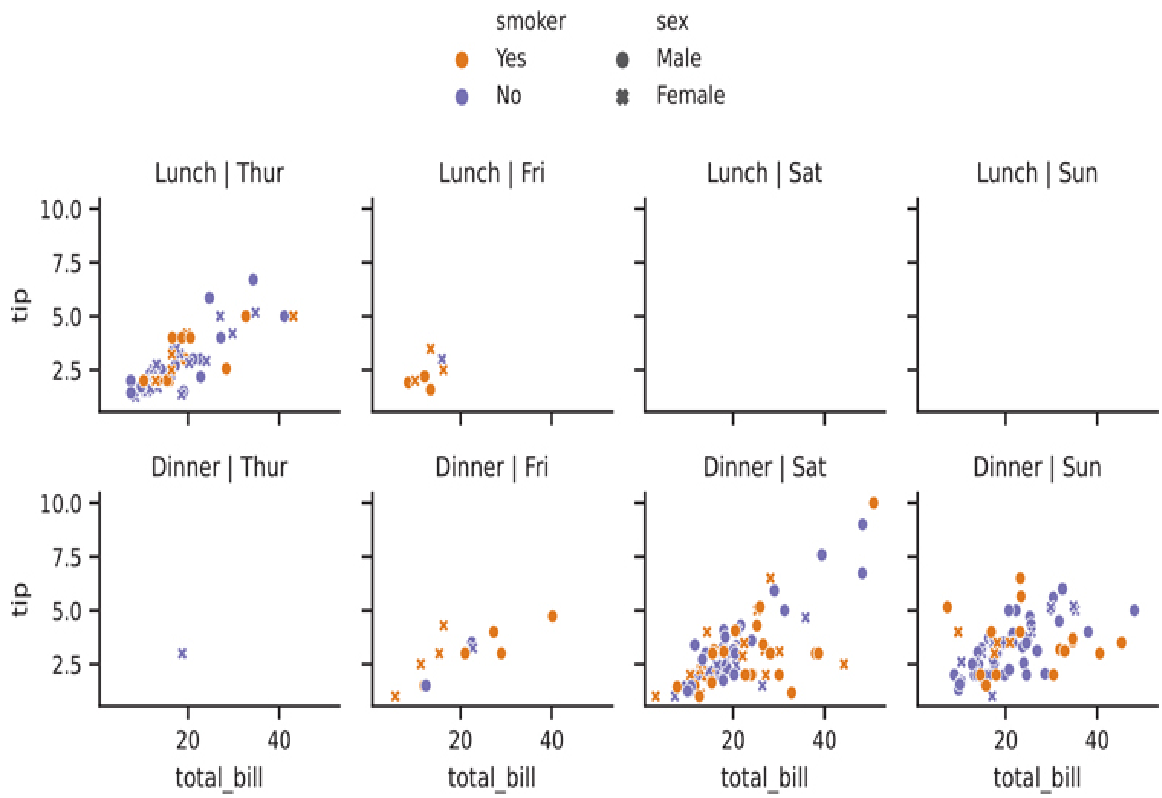


Figure 3.35 Seaborn tips scatter plot with hue, style, and facets

Click here to view code image

```
anscombe_plot = sns.relplot(
  data=anscombe,
  x="x",
  y="y",
  kind="scatter",
  col="data set",

  col_wrap=2,
  height=2,
  aspect=1.6, # aspect ratio of each facet
)

anscombe_plot.figure.set_tight_layout(True)
```

```
plt.show()
```

The col parameter is the variable that the plot will facet by, and the col_wrap parameter creates a figure that has two columns. If we do not use the col_wrap parameter, all four plots will be plotted in the same row.

3.4.4.2 Two Facet Variables

We can build on this to incorporate two categorical variables into our faceted plot. Additional categorical variables can be passed into the hue, style, etc. parameters.

Click here to view code image

```
'''python
colors = {
  "Yes": "#f1a340", # orange
  "No" : "#998ec3", # purple
}
# make the faceted scatter plot
# this is the only part that is needed to draw
the figure
facet2 = sns.relplot(
  data=tips,
  x="total_bill",
  y="tip",
  hue="smoker",
  style="sex",

  kind="scatter",
  col="day",
  row="time",
  palette=colors,
```

```
  height=1.7, # adjusted to fit figure on page
)

# below is to make the plot pretty
# adjust facet titles
facet2.set_titles(
  row_template="{row_name}",
  col_template="{col_name}"
)

# adjust the legend to not have it overlap the
figure
sns.move_legend(
  facet2,
  loc="lower center",
  bbox_to_anchor=(0.5, 1),
  ncol=2,   #number legend columns
  title=None,   #legend title
  frameon=False, #remove frame (i.e., border
box) around legend
)

facet2.figure.set_tight_layout(True)

plt.show()'''
```

3.4.4.3 Manually Create Facets

Many of the plots we created in `seaborn` are axes-level functions. What this means is that not every plotting function will have `col` and `col_wrap` parameters for faceting. Instead, we must create a `FacetGrid` that knows which variable to facet on, and then supply the individual plot code for each facet. Figure 3.36 shows our manually created facet plot.

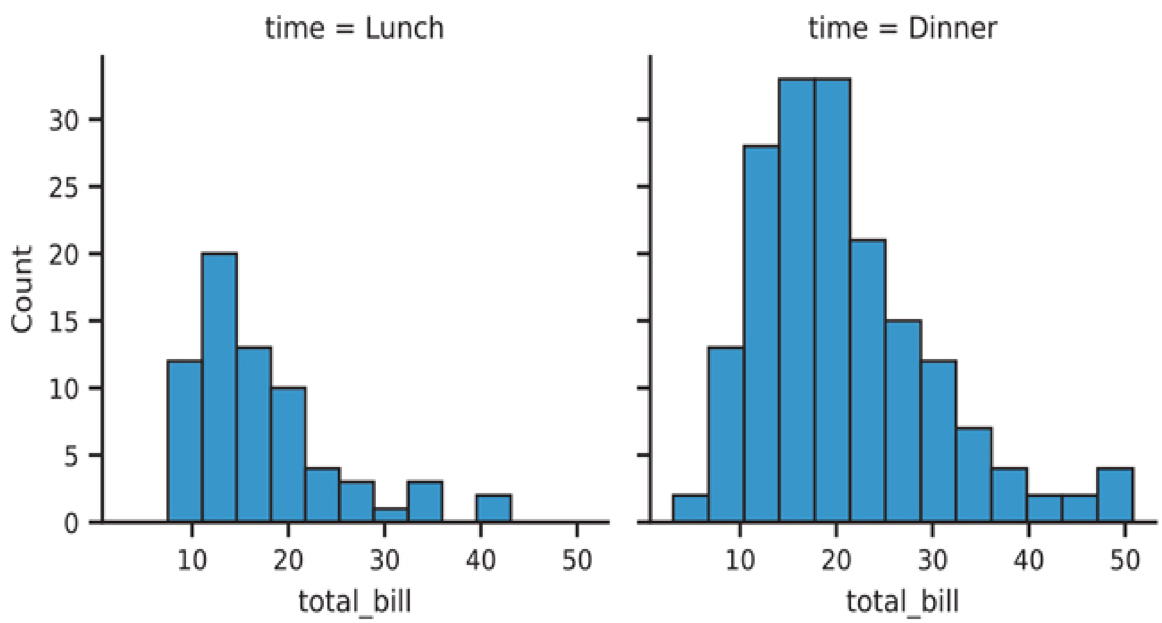


Figure 3.36 Seaborn plot with manually created facets

> **Danger**
>
> If you can, use one of the `seaborn` plotting functions that returns a figure object with `row` and `col` parameters to facet (e.g., `sns.relplot()` or `sns.catplot()`). You should opt to use those functions instead of manually creating a `FacetGrid` object. Many of the `seaborn` plotting functions will point to a different `seaborn` function if you want to facet.

Click here to view code image

```
# create the FacetGrid
facet = sns.FacetGrid(tips, col='time')

# for each value in time, plot a histogram of
total bill
# you pass in parameters as if you were passing
them directly
# into sns.histplot()
```

```
facet.map(sns.histplot, 'total_bill')
plt.show()
```

The individual facets need not be univariate plots, as seen in Figure 3.37.

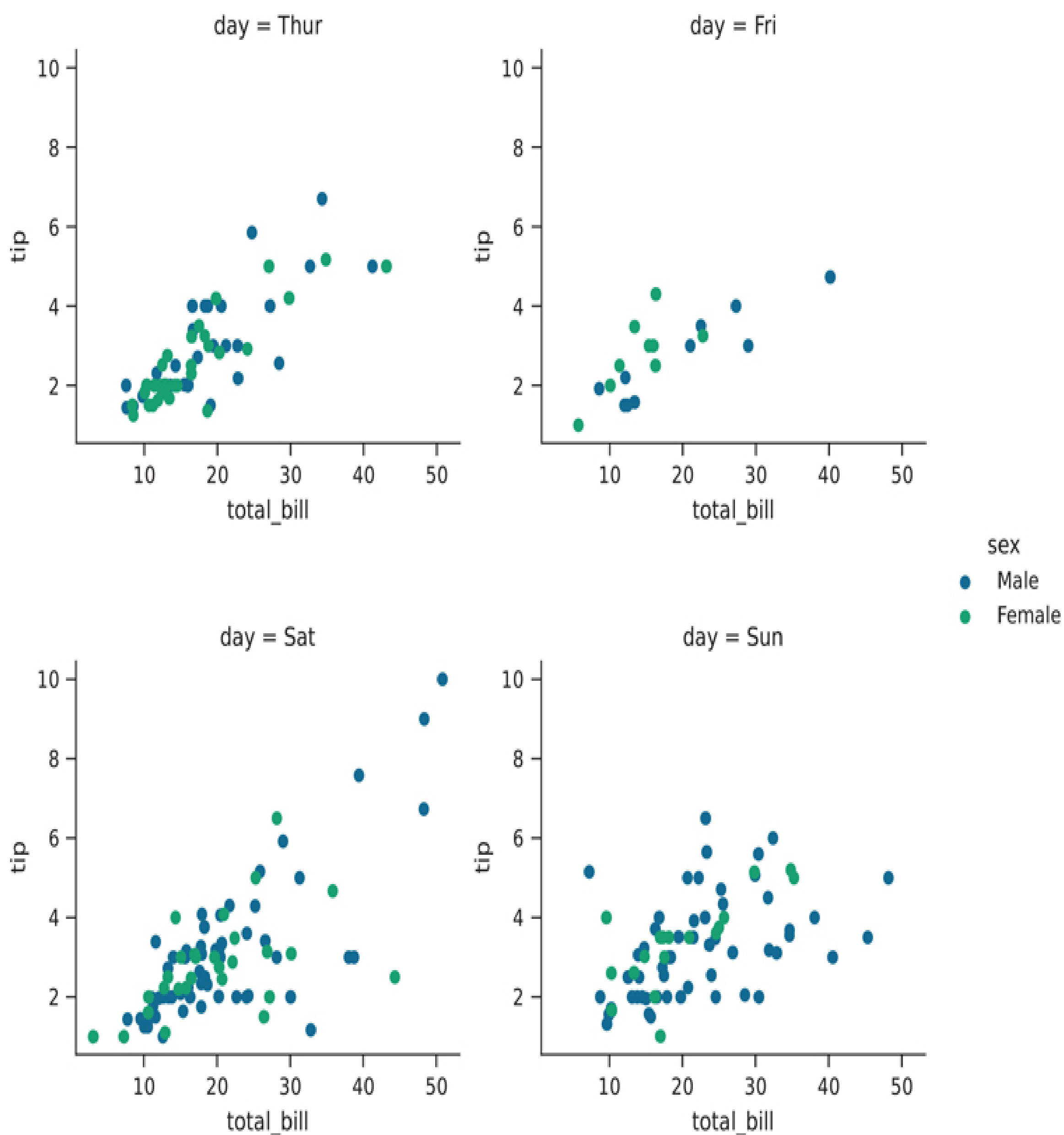


Figure 3.37 Seaborn plot with manually created facets that contain multiple variables

Click here to view code image

```
facet = sns.FacetGrid(
  tips, col='day', hue='sex', palette="viridis"
)
facet.map(plt.scatter, 'total_bill', 'tip')
facet.add_legend()
plt.show()
```

Another thing you can do with facets is to have one variable be faceted on the *x*-axis, and another variable faceted on the *y*-axis. We accomplish this by passing a `row` parameter. The result is shown in Figure 3.38.

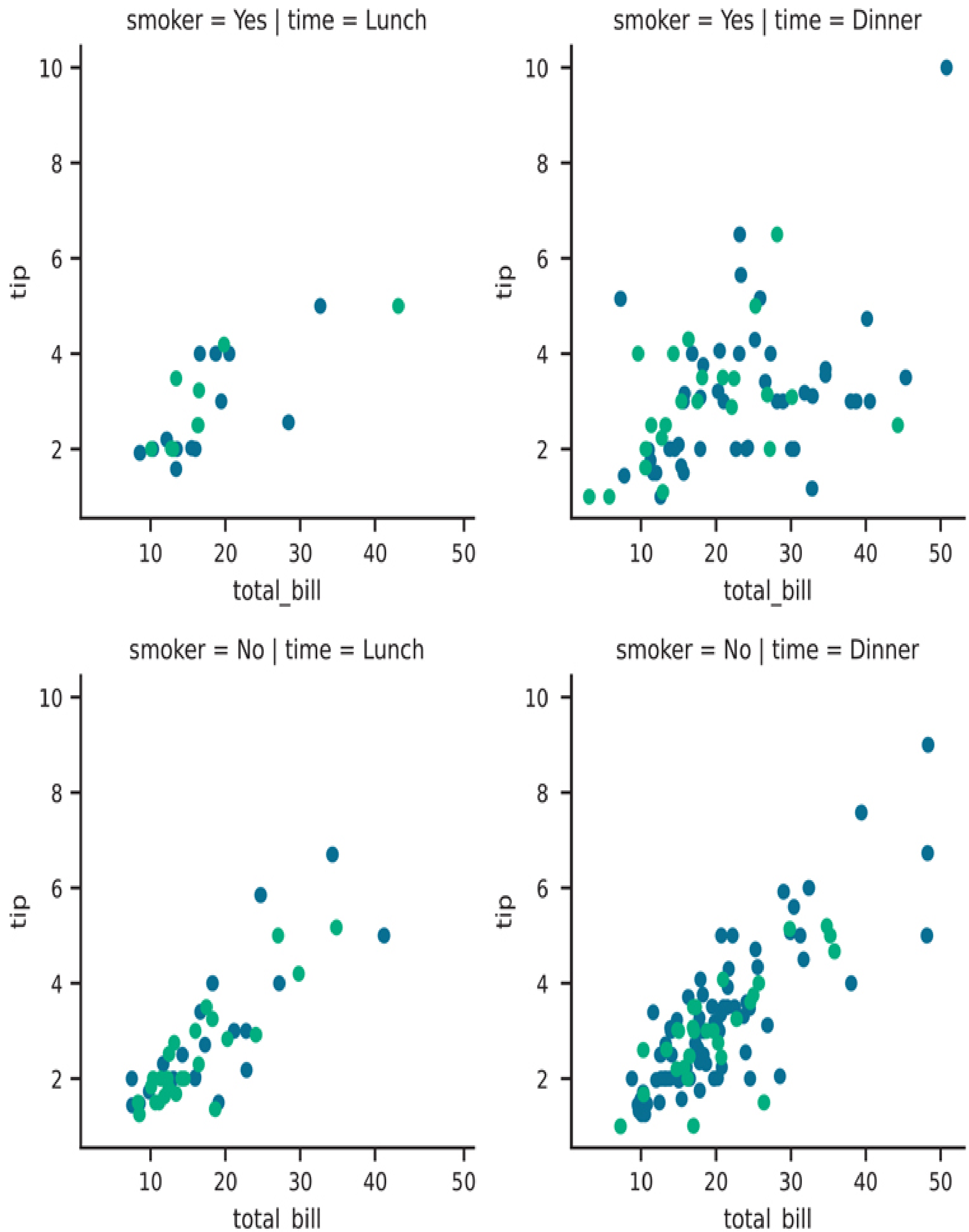


Figure 3.38 Seaborn plot with manually created facets with two variables

Click here to view code image

```
facet = sns.FacetGrid(
  tips, col='time', row='smoker', hue='sex',
palette="viridis"
)
facet.map(plt.scatter, 'total_bill', 'tip')
plt.show()
```

If you do not want all of the `hue` elements to overlap (i.e., you want this behavior in scatter plots, but not violin plots), you can use the `sns.catplot()` function. The result is shown in Figure 3.39.

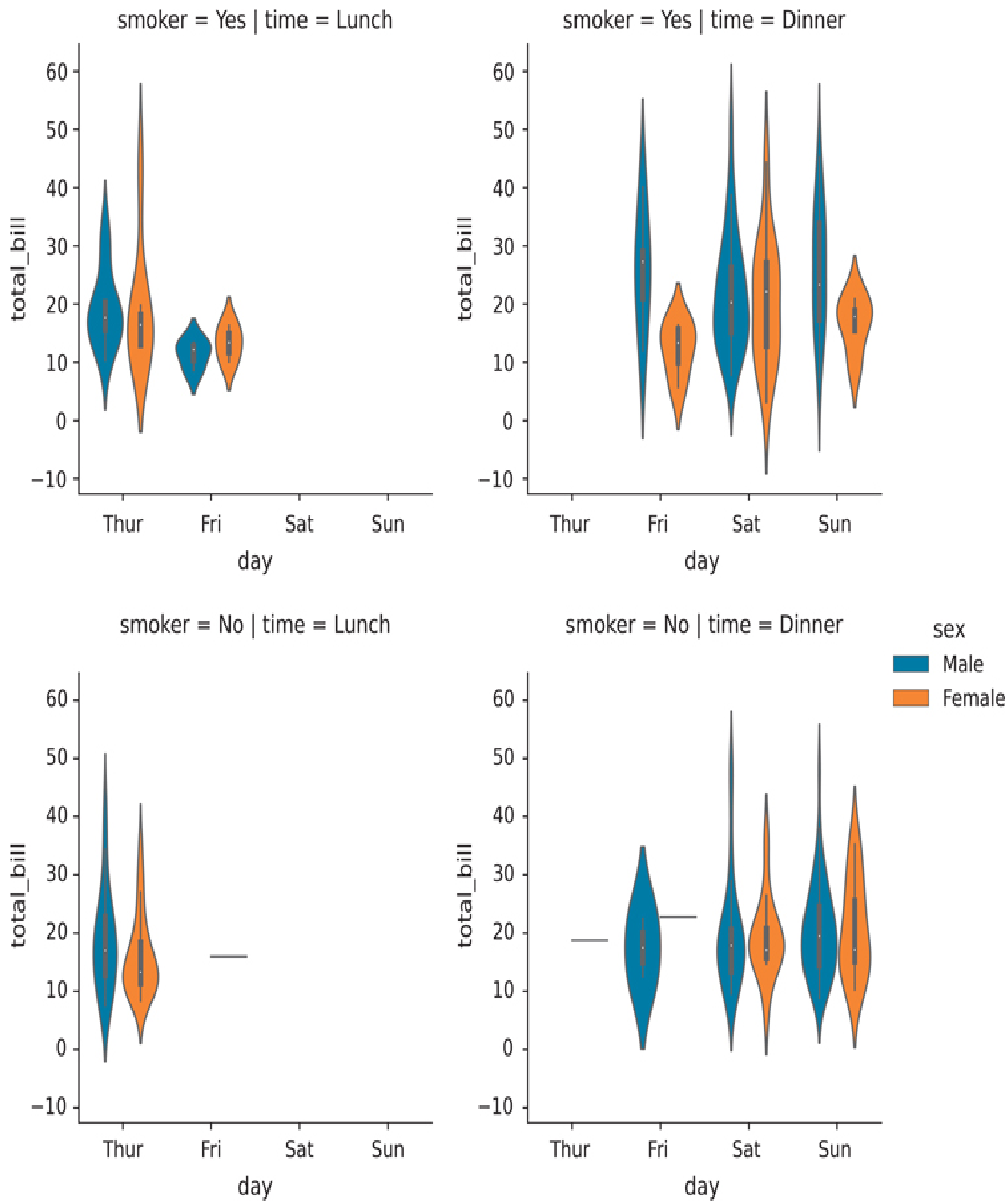


Figure 3.39 Seaborn plot with manually created facets with two non-overlapping variables

```
facet = sns.catplot(
    x="day",
```

```
    y="total_bill",
    hue="sex",
    data=tips,
    row="smoker",
    col="time",
    kind="violin",
)
plt.show()
```

3.4.5 Seaborn Styles and Themes

The seaborn plots shown in this chapter have all used the default plot styles. We can change the plot style with the sns.set_style function. Typically, this function is run just once at the top of your code; all subsequent plots will use the same style set.

3.4.5.1 Styles

The styles that come with seaborn are darkgrid, whitegrid, dark, white, and ticks. Figure 3.40 shows a base plot, and Figure 3.41 shows a plot with the whitegrid style.

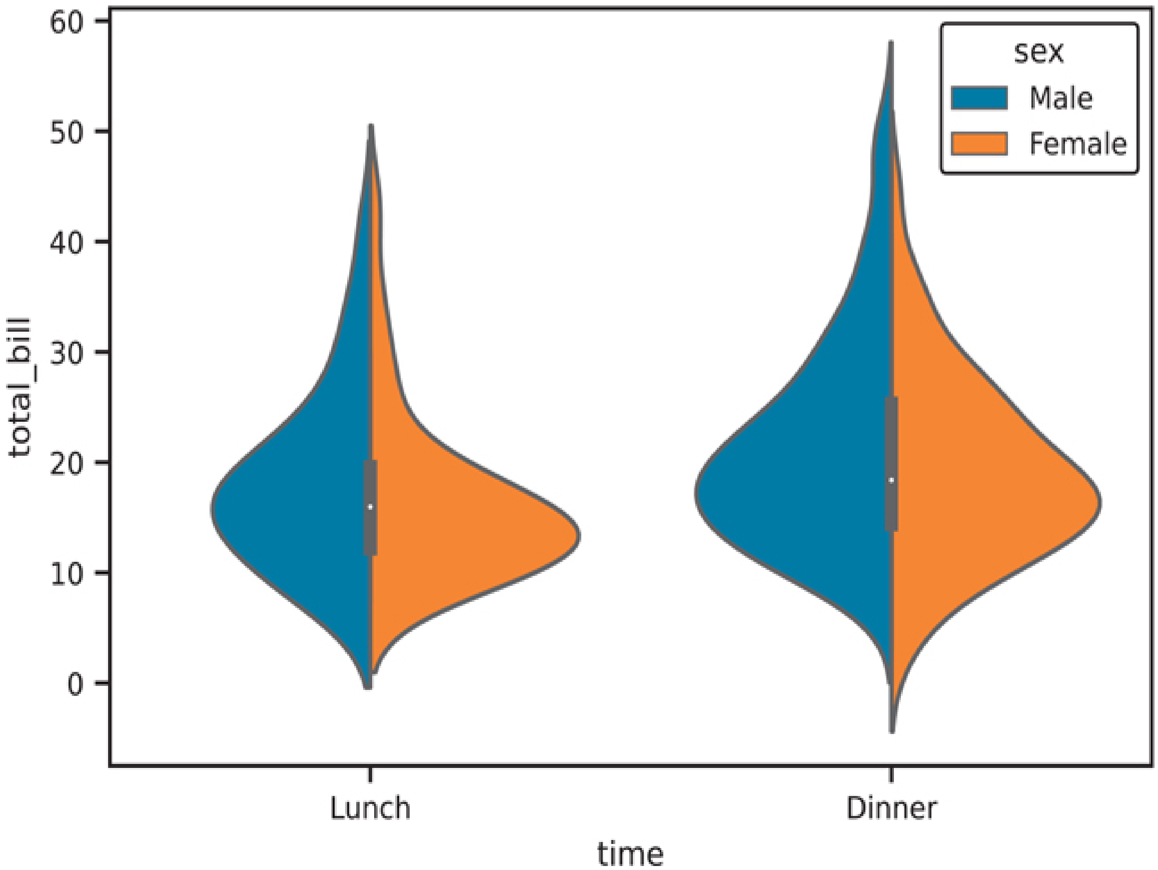


Figure 3.40 Baseline violin plot with default seaborn style

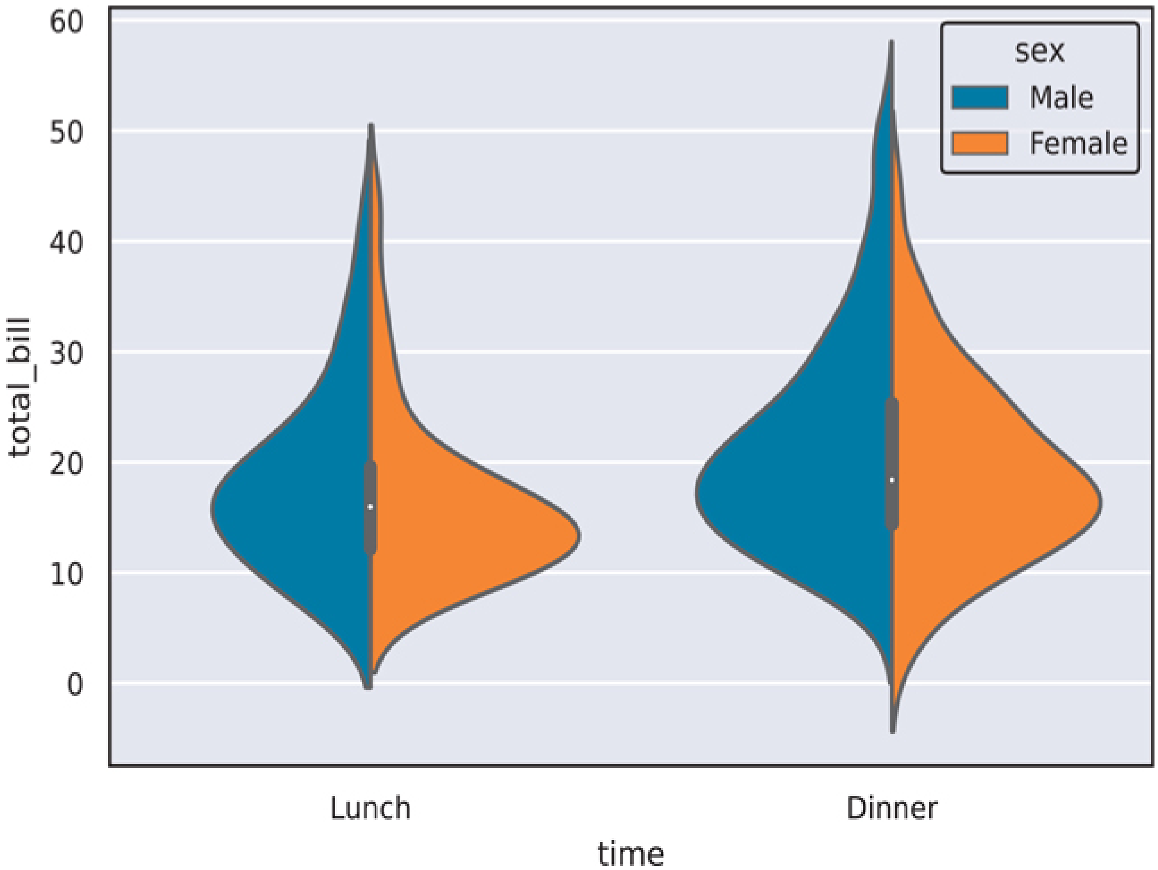


Figure 3.41 Violin plot with "darkgrid" seaborn style

The `with` block allow us to temporarily use a style without setting it as a default for all subsequent plots. If you want to set the style as a default you would use `sns.set_style("whitegrid")` instead of the `with` block.

Click here to view code image

```
# initial plot for comparison
fig, ax = plt.subplots()
sns.violinplot(
  data=tips, x="time", y="total_bill",
hue="sex", split=True, ax=ax
)
```

```
plt.show()
```

```
# Use this to set a global default style
# sns.set_style("whitegrid")
```

```
# temporarily set style and plot
# remove the with line + indentation if using
sns.set_style()
with sns.axes_style("darkgrid"):
```

```
  fig, ax = plt.subplots()
  sns.violinplot(
    data=tips, x="time", y="total_bill",
hue="sex", split=True, ax=ax
  )

plt.show()
```

The following code shows what all the styles look like (Figure 3.42).

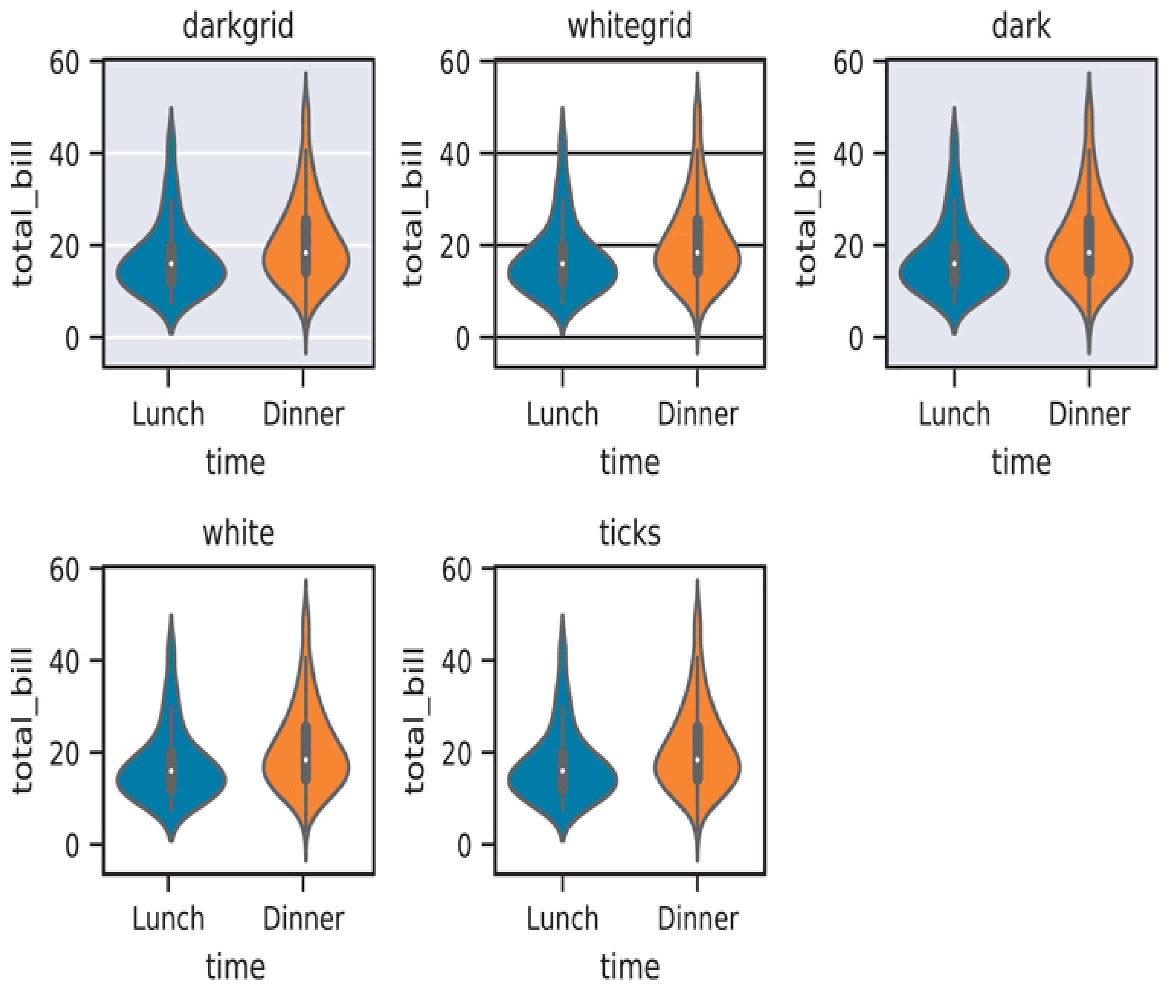


Figure 3.42 All seaborn styles

Click here to view code image

```
seaborn_styles = ["darkgrid", "whitegrid",
"dark", "white", "ticks"]

fig = plt.figure()
for idx, style in enumerate(seaborn_styles):
  plot_position = idx + 1
  with sns.axes_style(style):
    ax = fig.add_subplot(2, 3, plot_position)
    violin = sns.violinplot(
```

```
        data=tips, x="time", y="total_bill", ax=ax
    )
    violin.set_title(style)
fig.set_tight_layout(True)
plt.show()
```

3.4.5.2 Plotting Contexts

The `seaborn` library comes with a set of contexts that quickly tweak various parts of the figure (text size, line width, axis tick size, etc.) for different “contexts.” This chapter uses the `"paper"` context since it is made for printed text, but the default context is `"notebook"`. Below you will see the various parameters set for each context, and Figure 3.43 shows a quick preview of each context.

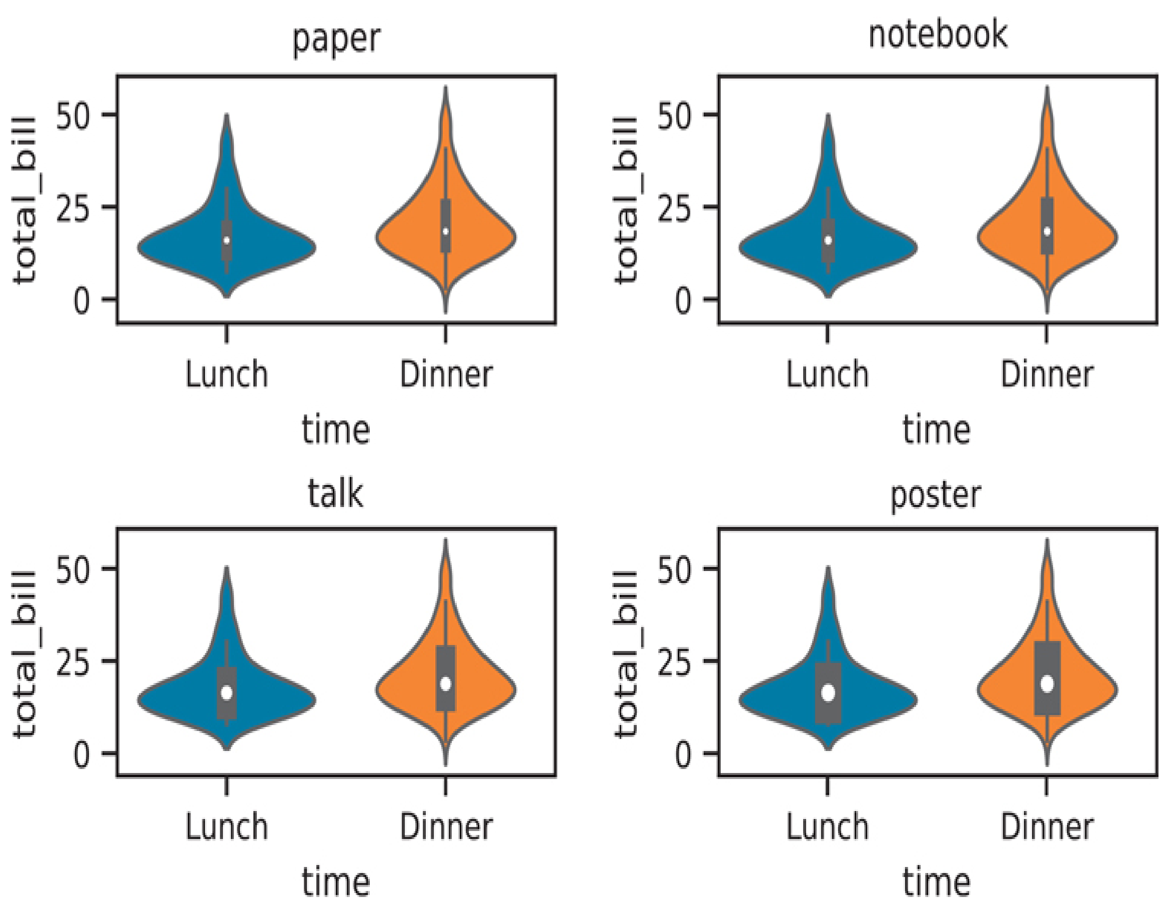


Figure 3.43 Example of seaborn figure contexts

Click here to view code image

```
contexts = pd.DataFrame(
    {
        "paper": sns.plotting_context("paper"),
        "notebook":
sns.plotting_context("notebook"),
        "talk": sns.plotting_context("talk"),
        "poster": sns.plotting_context("poster"),
    }
)
print(contexts)
```

```
                    paper  notebook    talk  poster
axes.linewidth        1.0      1.25   1.875     2.5
grid.linewidth        0.8      1.00   1.500     2.0
lines.linewidth       1.2      1.50   2.250     3.0
lines.markersize      4.8      6.00   9.000    12.0
patch.linewidth       0.8      1.00   1.500     2.0
xtick.major.width     1.0      1.25   1.875     2.5
ytick.major.width     1.0      1.25   1.875     2.5
xtick.minor.width     0.8      1.00   1.500     2.0
ytick.minor.width     0.8      1.00   1.500     2.0

xtick.major.size      4.8      6.00   9.000    12.0
ytick.major.size      4.8      6.00   9.000    12.0
xtick.minor.size      3.2      4.00   6.000     8.0
ytick.minor.size      3.2      4.00   6.000     8.0
font.size             9.6     12.00  18.000    24.0
axes.labelsize        9.6     12.00  18.000    24.0
axes.titlesize        9.6     12.00  18.000    24.0
xtick.labelsize       8.8     11.00  16.500    22.0
ytick.labelsize       8.8     11.00  16.500    22.0
```

```
legend.fontsize          8.8    11.00  16.500   22.0
legend.title_fontsize    9.6    12.00  18.000   24.0
```

```
context_styles = contexts.columns

fig = plt.figure()
for idx, context in enumerate(context_styles):
  plot_position = idx + 1
  with sns.plotting_context(context):
    ax = fig.add_subplot(2, 2, plot_position)
    violin = sns.violinplot(
      data=tips, x="time", y="total_bill", ax=ax
    )
    violin.set_title(context)
fig.set_tight_layout(True)
plt.show()
```

3.4.6 How to Go Through Seaborn Documentation

Throughout this chapter discussing `seaborn` plotting, we've talked about different plotting objects that come out of the `matplotlib` library, mainly the `Axes` and `Figure` objects. For all plotting libraries that build on top of `matplotlib`, it's important to know how to read aspects of the documentation, so you can customize your plots to your liking.

Let's use the violin plot (Figure 3.27) and pair plot (Figure 3.28) in Section 3.4.2.7 and Section 3.4.2.8 as examples of how to walk through object documentation.

3.4.6.1 Matplotlib Axes Objects

A snippet of the code for Figure 3.27 is below:

Click here to view code image

```
box_violin, (ax1, ax2) = plt.subplots(nrows=1,
ncols=2)

sns.boxplot(data=tips, x='time', y='total_bill',
ax=ax1)
sns.violinplot(data=tips, x='time',
y='total_bill', ax=ax2)

ax1.set_title('Box Plot')
ax1.set_xlabel('Time of day')
ax1.set_ylabel('Total Bill')

ax2.set_title('Violin Plot')
ax2.set_xlabel('Time of day')
ax2.set_ylabel('Total Bill')

box_violin.suptitle("Comparison of Box Plot with
Violin Plot")

box_violin.set_tight_layout(True)
plt.show()
```

In this particular example, if we look up the documentation for the `sns.violinplot()`, we will see that the function returns a `matplotlib Axes` object.

> Returns ax : matplotlib Axes

Click here to view code image

```
Returns the Axes object with the plot drawn onto
it.
```

We can also confirm that the `ax2` object we created is an `Axes` object:

Click here to view code image

```
print(type(ax2))
```

```
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

Since the `Axes` object is from `matplotlib`, if we want to make additional tweaks to the figure outside of the `sns.violinplot()` function, we would need to look into the `matplotlib.axes` documentation.[8] This is where you would find the documentation for the `.set_title()` method that was used to create the figure title.

8. `Axes` API docs: https://matplotlib.org/stable/api/axes_api.html#module-matplotlib.axes

3.4.6.2 Matplotlib Figure Objects

Using the same reproduced code for Figure 3.27 above, we can see the `type()` of the `box_violin` object we created and go to the `Figure` documentation.[9]

9. `Figure` API docs: https://matplotlib.org/stable/api/figure_api.html#module-matplotlib.figure

Click here to view code image

```
print(type(box_violin))
```

```
<class 'matplotlib.figure.Figure'>
```

This is where we can find the `.suptitle()` method used to add the overall title to the figure.

3.4.6.3 Custom Seaborn Objects

The code for Figure 3.28 is reproduced below:

Click here to view code image

```
fig = sns.pairplot(data=tips)
fig.figure.suptitle(
  'Pairwise Relationships of the Tips Data',
y=1.03
)
plt.show()
```

This is an example of an object specific to `seaborn`, the `PairGrid` object.[10]

10. `seaborn.PairGrid` docs: `https://seaborn.pydata.org/generated/seaborn.PairGrid.html`

Click here to view code image

```
print(type(fig))
```

```
<class 'seaborn.axisgrid.PairGrid'>
```

If we scroll down to the bottom of the documentation page, we can see all the attributes and methods for the `PairGrid` object. However, we know that `.suptitle()` is a `matplotlib.Figure` method. From the API documentation at the bottom of the page, we can see how we can access the underlying `Figure` object by using the `.figure` attribute. This is why we needed to write `.figure.suptitle()` to take the `sns.FacetGrid` object, access the `matplotlib.Figure` object, then the `.subtitle()` method.

3.4.7 Next-Generation Seaborn Interface

There is a new `seaborn` interface in the works.[11] However, at the time of writing, the next-gen interface is not official yet. When the official change occurs and the API is stable, the book's website will provide the updated code for the `seaborn` section.[12]

11. Next-generation seaborn interface:
https://seaborn.pydata.org/nextgen/

12. *Pandas for Everyone* GitHub Page:
https://github.com/chendaniely/pandas_for_everyone/

3.5 Pandas Plotting Method

Pandas objects also come equipped with their own plotting functions. Just as in `seaborn`, the plotting functions built into Pandas are just wrappers around `matplotlib` with preset values. In general, plotting using Pandas follows the `DataFrame.plot.<PLOT_TYPE>` or `Series.plot.<PLOT_TYPE>` methods.

3.5.1 Histogram

Histograms can be created using the `Series.plot.hist()` (Figure 3.44) or `DataFrame.plot.hist()` (Figure 3.45) function.

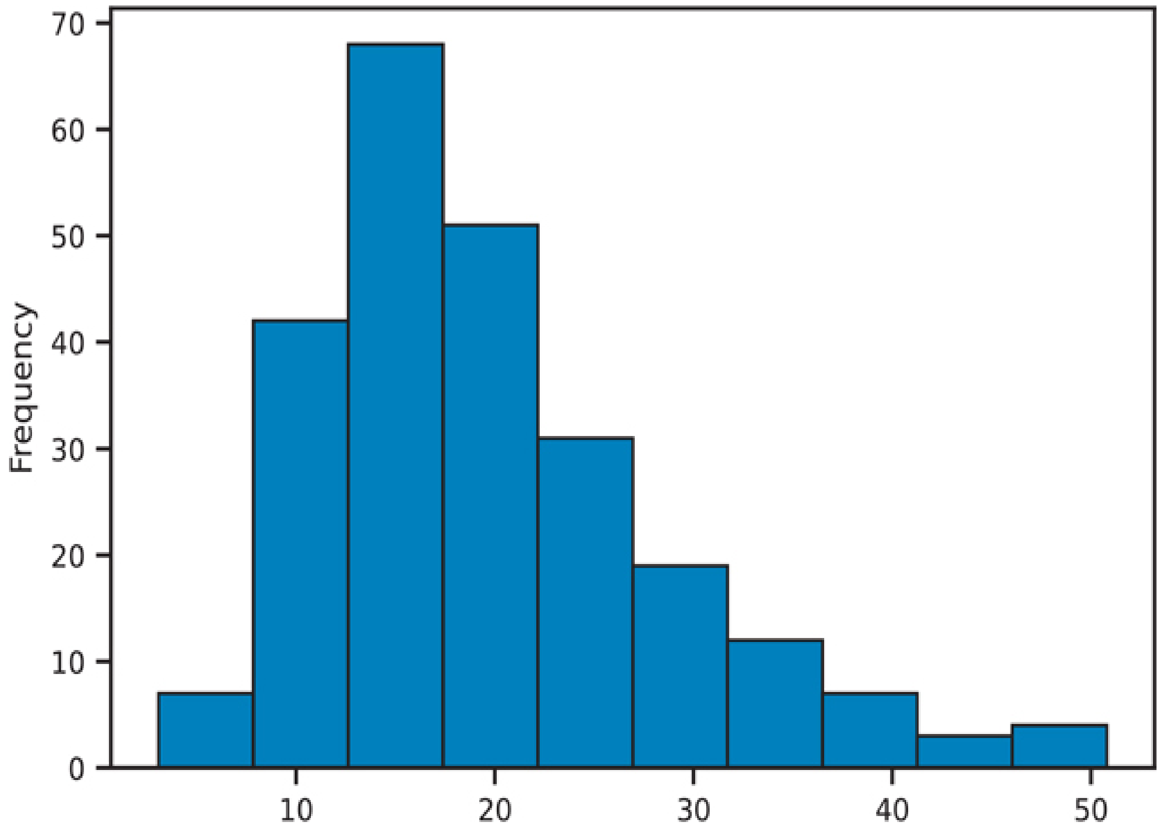


Figure 3.44 Histogram of a Pandas Series

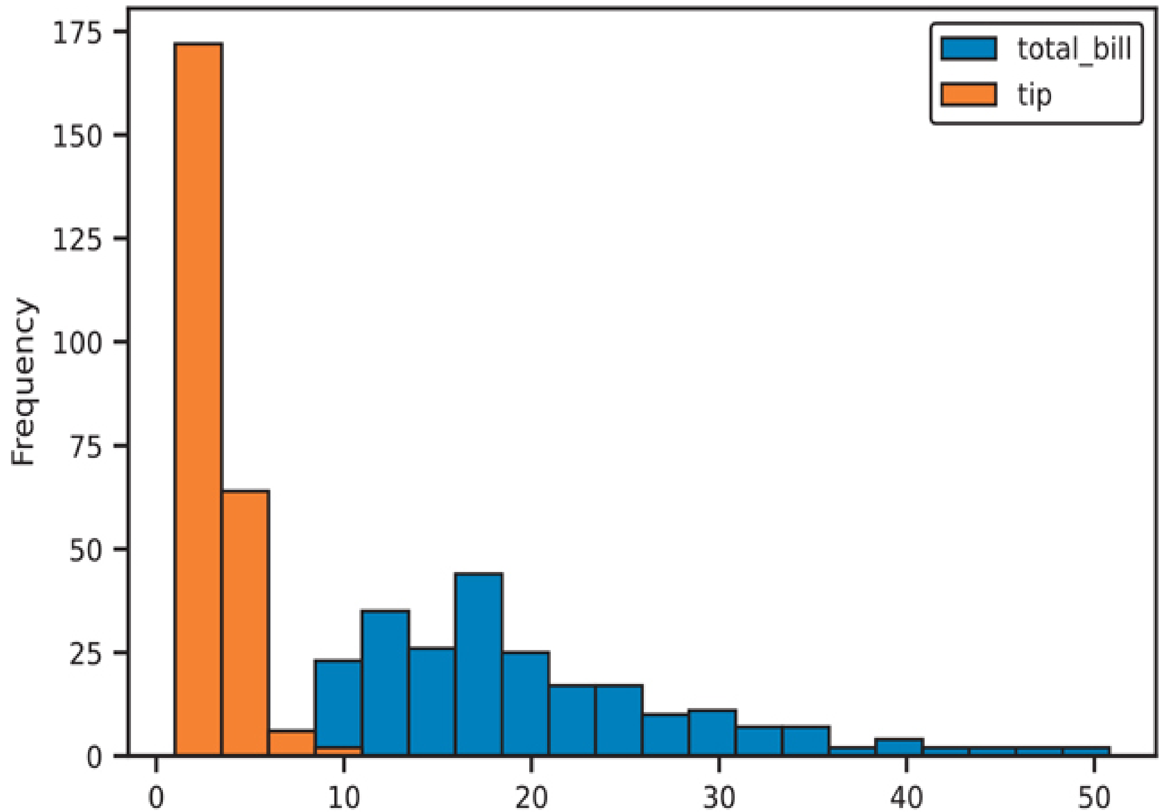


Figure 3.45 Histogram of a Pandas DataFrame

Click here to view code image

```
# on a series
fig, ax = plt.subplots()
tips['total_bill'].plot.hist(ax=ax)
plt.show()
```

```
# on a dataframe
# set alpha channel transparency to see through
the overlapping bars
fig, ax = plt.subplots()
tips[['total_bill', 'tip']].plot.hist(alpha=0.5,
bins=20, ax=ax)
plt.show()
```

3.5.2 Density Plot

The kernel density estimation (density) plot can be created with the `DataFrame.plot. kde()` function (Figure 3.46).

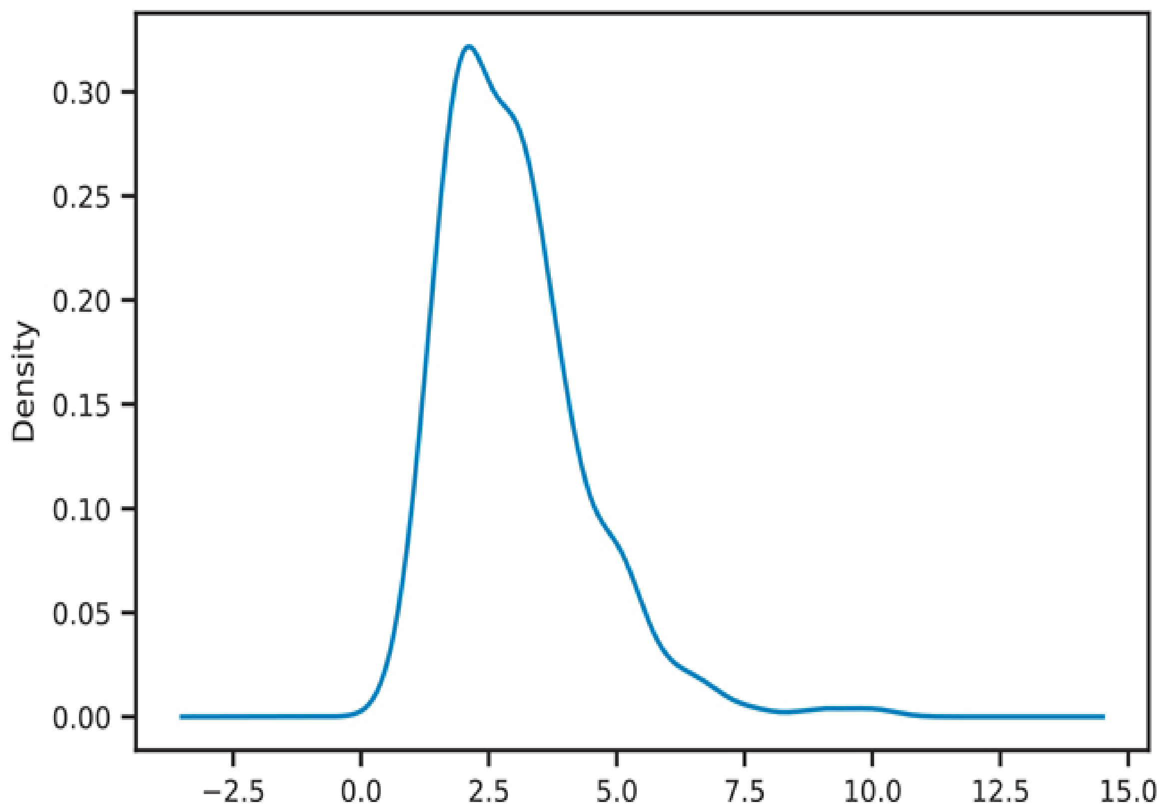


Figure 3.46 Pandas KDE plot

```
fig, ax = plt.subplots()
tips['tip'].plot.kde(ax=ax)
plt.show()
```

3.5.3 Scatter Plot

Scatter plots are created by using the `DataFrame.plot.scatter()` function (Figure 3.47).

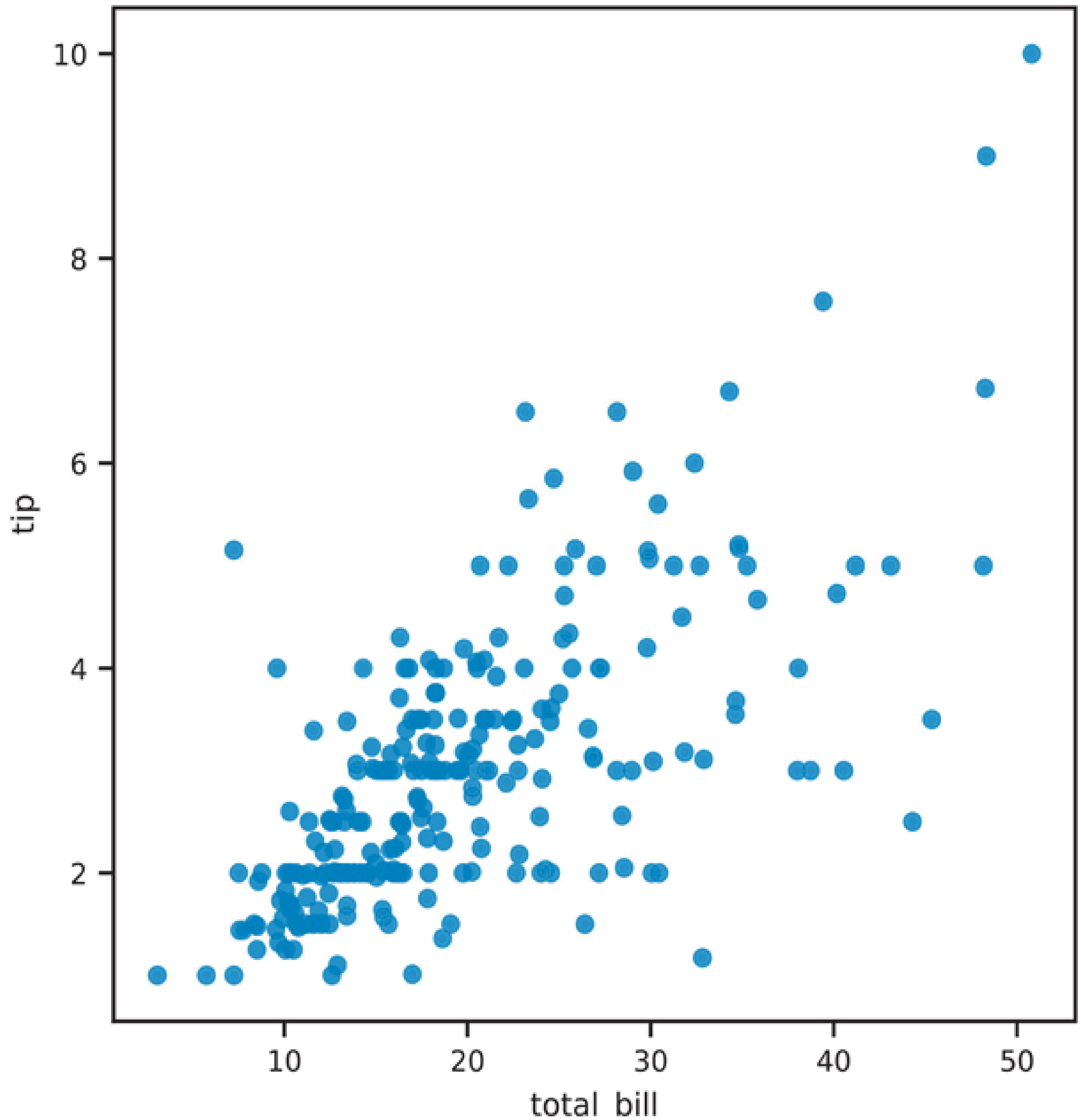


Figure 3.47 Pandas scatter plot

Click here to view code image

```
fig, ax = plt.subplots()
tips.plot.scatter(x='total_bill', y='tip',
ax=ax)
plt.show()
```

3.5.4 Hexbin Plot

Hexbin plots are created using the `Dataframe.plt.hexbin()` function (Figure 3.48).

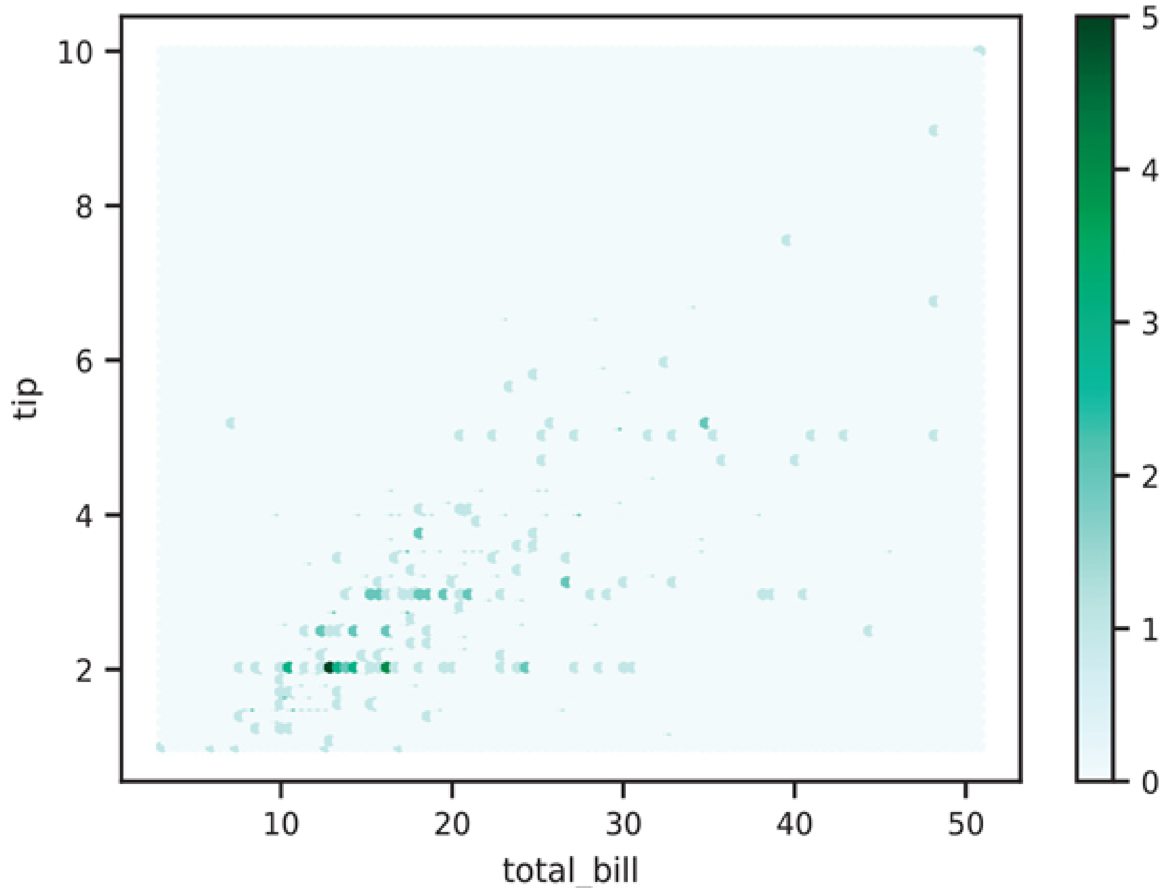


Figure 3.48 Pandas hexbin plot

Click here to view code image

```
fig, ax = plt.subplots()
tips.plot.hexbin(x='total_bill', y='tip', ax=ax)
plt.show()
```

Grid size can be adjusted with the `gridsize` parameter (Figure 3.49).

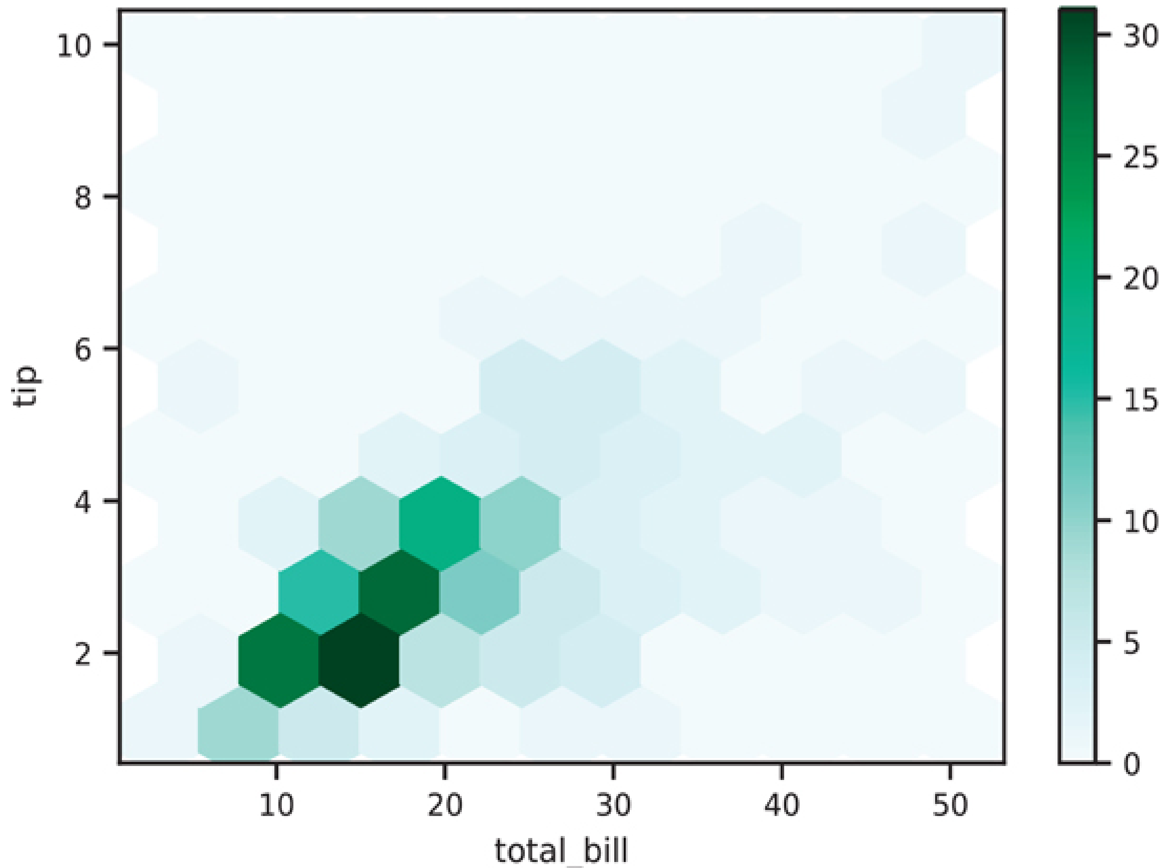


Figure 3.49 Pandas hexbin plot with modified grid size

Click here to view code image

```
fig, ax = plt.subplots()
tips.plot.hexbin(x='total_bill', y='tip',
gridsize=10, ax=ax)
plt.show()
```

3.5.5 Box Plot

Box plots are created with the `DataFrame.plot.box()` function (Figure 3.50).

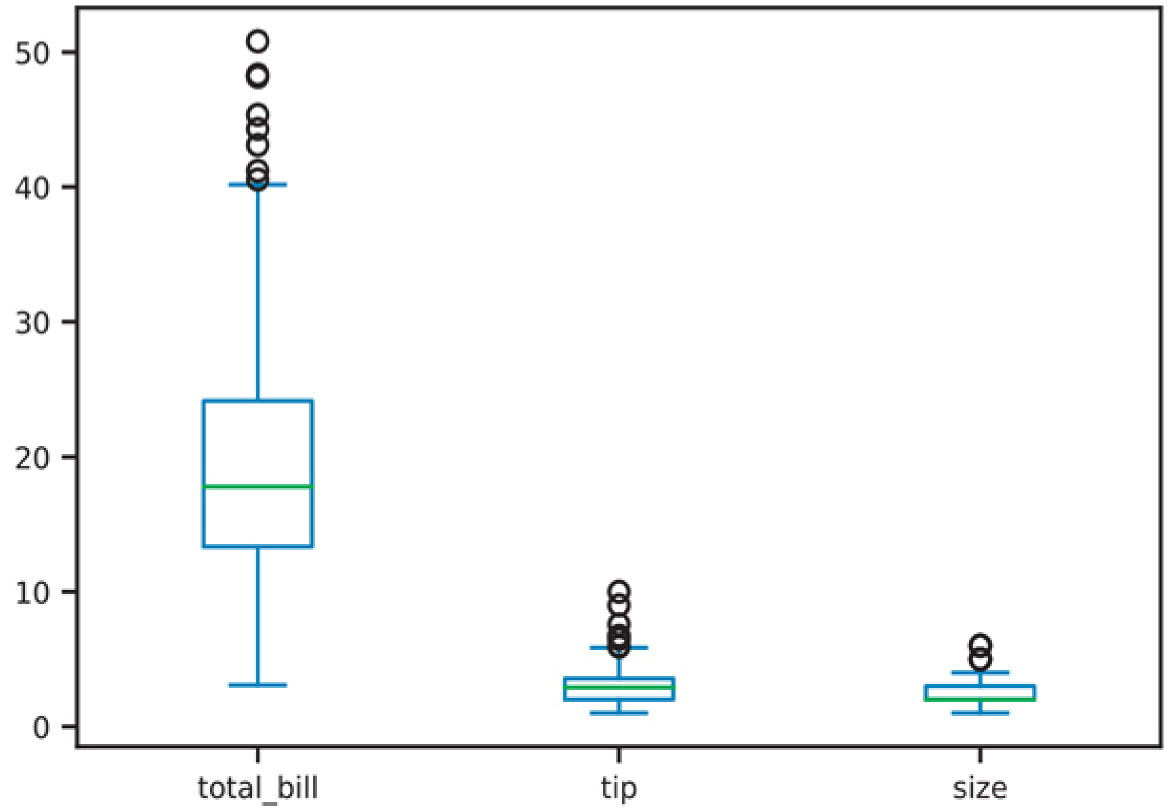


Figure 3.50 Pandas box plot

```
fig, ax = plt.subplots()
ax = tips.plot.box(ax=ax)
plt.show()
```

Conclusion

Data visualization is an integral part of exploratory data analysis and data presentation. This chapter provided an introduction to the various ways to explore and present your data. As we continue through the book, we will learn about more complex visualizations.

There are myriad plotting and visualization resources available on the Internet. The `seaborn` documentation, Pandas visualization documentation, and `matplotlib` documentation all provide ways to further tweak your plots (e.g., colors, line thickness, legend placement, figure annotations). Other resources include `colorbrewer` to help pick good color schemes. The plotting libraries mentioned in this chapter also have various color schemes that can be used to highlight the content of your visualizations.

4

Tidy Data

Hadley Wickham, PhD,[1] one of the more prominent members of the R community, introduced the concept of **tidy data** in a *Journal of Statistical Software* paper.[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.

1. Hadley Wickham, PhD: `http://hadley.nz`

2. Tidy Data paper: `http://vita.had.co.nz/papers/tidy-data.pdf`

What is tidy data? Hadley Wickham's paper defines it as meeting the following criteria: (1) Each row is an observation, (2) Each column is a variable, and (3) Each type of observational unit forms a table.

The newer definition from the R4DS book[3] focuses on an individual data set (i.e., table):

3. R For Data Science Book: `https://r4ds.had.co.nz/tidy-data.html`

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

This chapter goes through the various ways to tidy data using examples from Wickham's paper.

Learning Objectives

The concept map for this chapter can be found in Figure A.4.

- Identify the components of tidy data
- Identify common data errors
- Use functions and methods to process and tidy data

Note About This Chapter

Data used in this chapter will have `NaN` missing values when they are loaded into Pandas (Chapter 9). In the raw CSV files, they will appear as empty values. I typically try to avoid forward referencing in the book, but I felt that the concept of tidy data warranted a much earlier place in the book because it is so fundamental to how we should be thinking about data technically (as opposed to ethically), that the chapter was moved toward the front of the book without having to cover more detailed data processing steps first. I could have changed the data sets such that there were no missing values, but opted not to do so because (1) it would no longer follow the data used in Wickam's "Tidy Data" paper, and (2) it would be a less realistic data set.

4.1 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.

4.1.1 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.

Click here to view code image

```
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 should be reshaped so that we have `religion`, `income`, and `count` variables.

Click here to view code image

```
# show only the first few columns
print(pew.iloc[:,  0:5])
```

```
                religion  <$10k  $10-20k $20-30k
$30-40k
0               Agnostic     27       34      60
81
1                Atheist     12       27      37
52
2               Buddhist     27       21      30
34
3               Catholic    418      617     732
670
4     Don’t know/refused     15       14      15
11
..                   ...    ...      ...     ...
...
13             Orthodox      13       17      23
32
14       Other Christian      9        7      11
13
15          Other Faiths     20       33      40
46
16  Other World Religions     5        2       3
4
17          Unaffiliated    217      299     374
```

```
365

[18 rows x 5 columns]
```

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.

> **Note**
>
> I usually use the terminology from the R world of using "pivot" to refer to going from wide data to long data and vice versa. I usually will specify the direction with "pivot longer" to go from wide data to long data, and "pivot wider" to go from long data to wide data.
>
> In this chapter "pivot longer" will refer to the dataframe `.melt()` method, and "pivot wider" will refer to the dataframe `.pivot()` method.

Pandas `DataFrames` have a method called `.melt()` that will reshape the dataframe into a tidy format and it 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`.

Click here to view code image

```
# we do not need to specify a value_vars since
we want to pivot
```

```python
# all the columns except for the 'religion' column
pew_long = pew.melt(id_vars='religion')
```

```python
print(pew_long)
```

```
                 religion             variable
value
0                Agnostic                <$10k
27
1                 Atheist                <$10k
12
2                Buddhist                <$10k
27
3                Catholic                <$10k
418
4      Don't know/refused                <$10k
15
..                    ...                  ...
...
175              Orthodox  Don't know/refused
73
176       Other Christian  Don't know/refused
18
177          Other Faiths  Don't know/refused
71
178 Other World Religions  Don't know/refused
8
179          Unaffiliated  Don't know/refused
597

[180 rows x 3 columns]
```

Note

The `.melt()` method also exists as a pandas function, `pd.melt()`

The below two lines of code are equivalent:

Click here to view code image

```
# melt method
pew_long = pew.melt(id_vars='religion')

# melt function
pew_long = pd.melt(pew, id_vars='religion')
```

Internally, the `.melt()` method redirects the function call to the Pandas `pd.melt()` function. The `.melt()` method notation is there to make the Pandas API more consistent, and also allows us to method-chain (Appendix U).

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

Click here to view code image

```
pew_long = pew.melt(
  id_vars="religion", var_name="income",
value_name="count"
)
```

```
print(pew_long)
```

```
                  religion              income
count
0                 Agnostic               <$10k
27
1                  Atheist               <$10k
```

```
12
2                   Buddhist              <$10k
27
3                   Catholic              <$10k
418
4         Don't know/refused              <$10k
15
..                       ...                   ...
...
175                 Orthodox  Don't know/refused
73
176          Other Christian  Don't know/refused
18
177             Other Faiths  Don't know/refused
71
178  Other World Religions  Don't know/refused
8
179             Unaffiliated  Don't know/refused
597

[180 rows x 3 columns]
```

4.1.2 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.

Click here to view code image

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

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

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

3  2000  3 Doors Down                          Loser
4:24    2000-10-21
4  2000      504 Boyz                  Wobble Wobble
3:35    2000-04-15

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

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

Click here to view code image

```
# use a list to reference more than 1 variable
billboard_long = billboard.melt(
  id_vars=["year", "artist", "track", "time",
"date.entered"],
  var_name="week",
  value_name="rating",
)

print(billboard_long)
```

```
       year            artist
track  time  \
0      2000             2 Pac  Baby Don't Cry
(Keep... 4:22
1      2000           2Ge+her  The Hardest Part Of
... 3:15
2      2000      3 Doors Down
Kryptonite 3:53
3      2000      3 Doors Down
Loser 4:24
4      2000          504 Boyz            Wobble
Wobble 3:35
...     ...               ...
... ...
24087  2000       Yankee Grey     Another Nine
Minutes 3:10
24088  2000  Yearwood, Trisha          Real Live
Woman 3:55
24089  2000   Ying Yang Twins  Whistle While You
Tw... 4:19
24090  2000     Zombie Nation            Kernkraft
```

```
400 3:30
24091  2000   matchbox twenty
Bent 4:12

      date.entered  week  rating
0       2000-02-26   wk1    87.0
1       2000-09-02   wk1    91.0
2       2000-04-08   wk1    81.0
3       2000-10-21   wk1    76.0
4       2000-04-15   wk1    57.0
...            ...   ...     ...
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

[24092 rows x 7 columns]
```

4.2 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.

Click here to view code image

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

```
Index(['Date', 'Day', 'Cases_Guinea',
'Cases_Liberia',
```

```
       'Cases_SierraLeone', 'Cases_Nigeria',
'Cases_Senegal',
       'Cases_UnitedStates', 'Cases_Spain',
'Cases_Mali',
       'Deaths_Guinea', 'Deaths_Liberia',
'Deaths_SierraLeone',
       'Deaths_Nigeria', 'Deaths_Senegal',
'Deaths_UnitedStates',
       'Deaths_Spain', 'Deaths_Mali'],
      dtype='object')
```

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

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

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

First, let's fix the problem we know how to fix, by melting the data into long format.

Click here to view code image

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

```
print(ebola_long)
```

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


1947  3/27/2014    5   Deaths_Mali     NaN
1948  3/26/2014    4   Deaths_Mali     NaN
1949  3/25/2014    3   Deaths_Mali     NaN
1950  3/24/2014    2   Deaths_Mali     NaN
1951  3/22/2014    0   Deaths_Mali     NaN

[1952 rows x 4 columns]
```

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 11). 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 `DataFrames` 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" it 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.` attribute. `.str.` is a special type of attribute that Pandas calls an "accessor" because it can "access" string methods (see Chapter 11 for more on strings). Access to the Python string methods and allow us to work across the entire column. This will be the key to parting out the multiple bits of information stored in each value.

4.2.1 Split and Add Columns Individually

We can use the `.str` accessor to make a call to the `.split()` method and pass in the _ understore.

Click here to view code image

```
# 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
```

After we split on the underscore, the values are returned in a list. We can tell it's a list by:

1. Knowing about the `.split()` method on base Python string objects[4]
2. Visually seeing the square brackets in the output, `[ ]`
3. Getting the `type()` of one of the items in the `Series`

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

Click here to view code image

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

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

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

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

Now that the column has been split into 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.

Click here to view code image

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

```
print(status_values)
```

```
0        Cases
1        Cases
2        Cases
3        Cases
4        Cases
          ...
1947    Deaths
1948    Deaths
1949    Deaths
1950    Deaths
```

```
1951    Deaths
Name: variable, Length: 1952, dtype: object
```

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

Click here to view code image

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

```
print(ebola_long)
```

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


...         ...  ...            ...     ...
...       ...
1947  3/27/2014    5    Deaths_Mali     NaN
Deaths       Mali
1948  3/26/2014    4    Deaths_Mali     NaN
Deaths       Mali
1949  3/25/2014    3    Deaths_Mali     NaN
Deaths       Mali
```

```
1950   3/24/2014     2     Deaths_Mali      NaN
Deaths       Mali
1951   3/22/2014     0     Deaths_Mali      NaN
Deaths       Mali

[1952 rows x 6 columns]
```

4.2.2 Split and Combine in a Single Step

We can actually do the above steps in a single step. If we look at the `.str.split()` method documentation (you can find this by looking by going to the Pandas API documentation > `Series` > String Handling (`.str.`) > `.split()` method[5]), there is a parameter named `expand` that defaults to `False`, but when we set it to `True`, it will return a `DataFrame` where each result of the split is in a separate column, instead of a `Series` of `list` containers.

5. `Series.str.split()` method documentation: https://pandas.pydata.org/docs/reference/api/pandas.Series.str.split.html#pandas.Series.str.split

Click here to view code image

```
# reset our ebola_long data
ebola_long = ebola.melt(id_vars=['Date', 'Day'])

# split the column by _ into a dataframe using
expand
variable_split =
ebola_long.variable.str.split('_', expand=True)

print(variable_split)
```

```
           0        1
0      Cases   Guinea
```

```
1       Cases   Guinea
2       Cases   Guinea
3       Cases   Guinea
4       Cases   Guinea
...       ...      ...
1947   Deaths     Mali
1948   Deaths     Mali
1949   Deaths     Mali
1950   Deaths     Mali
1951   Deaths     Mali

[1952 rows x 2 columns]
```

From here, we can actually use the Python and Pandas multiple assignment feature (Appendix Q), to directly assign the newly split columns into the original `DataFrame`. Since our output `variable_split` returned a `DataFrame` with two columns, we can assign two new columns to our `ebola_long DataFrame`.

Click here to view code image

```
ebola_long[['status', 'country']] =
variable_split
```

```
print(ebola_long)
```

```
           Date  Day      variable    value
status country
0       1/5/2015  289  Cases_Guinea   2776.0
Cases   Guinea
1       1/4/2015  288  Cases_Guinea   2775.0
Cases   Guinea
2       1/3/2015  287  Cases_Guinea   2769.0
Cases   Guinea
```

```
3       1/2/2015   286   Cases_Guinea       NaN
Cases   Guinea
4     12/31/2014   284   Cases_Guinea    2730.0
Cases   Guinea
...          ...   ...            ...       ...
...      ...
1947   3/27/2014     5    Deaths_Mali       NaN
Deaths     Mali
1948   3/26/2014     4    Deaths_Mali       NaN
Deaths     Mali
1949   3/25/2014     3    Deaths_Mali       NaN
Deaths     Mali
1950   3/24/2014     2    Deaths_Mali       NaN
Deaths     Mali
1951   3/22/2014     0    Deaths_Mali       NaN
Deaths     Mali

[1952 rows x 6 columns]
```

You can also opt to do this as a concatenation (`pd.concat()`) function call as well (Chapter 6).

4.3 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 (`.melt()` and some string parsing with the `.str.` accessor attribute). 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” the variable into separate columns, i.e., go from long data to wide data.

Click here to view code image

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

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

The weather data include minimum (`tmin`) and maximum (`tmax`) temperatures recorded for each day (`d1`, `d2`, …, `d31`) of the month (month). The `element` column contains variables that need to be pivoted wider 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 fix the `day` values.

Click here to view code image

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

```
print(weather_melt)
```

```
          id  year  month element  day  temp
0    MX17004  2010      1    tmax   d1   NaN
1    MX17004  2010      1    tmin   d1   NaN
2    MX17004  2010      2    tmax   d1   NaN
3    MX17004  2010      2    tmin   d1   NaN
4    MX17004  2010      3    tmax   d1   NaN
..       ...   ...    ...     ...  ...   ...
677  MX17004  2010     10    tmin  d31   NaN
678  MX17004  2010     11    tmax  d31   NaN
679  MX17004  2010     11    tmin  d31   NaN
680  MX17004  2010     12    tmax  d31   NaN
681  MX17004  2010     12    tmin  d31   NaN

[682 rows x 6 columns]
```

Next, we need to pivot up the variables stored in the element column.

Click here to view code image

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

```
print(weather_tidy)
```

```
element                   tmax  tmin
id      year month day
MX17004 2010 1     d30    27.8  14.5
             2     d11    29.7  13.4
                    d2    27.3  14.4
```

```
                d23   29.9   10.7
                 d3   24.1   14.4
...                    ...    ...

           11   d27   27.7   14.2
                d26   28.1   12.1
                 d4   27.2   12.0
           12    d1   29.9   13.8
                 d6   27.8   10.5

[33 rows x 2 columns]
```

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

Click here to view code image

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

```
element        id  year  month  day  tmax  tmin
0         MX17004  2010      1  d30  27.8  14.5
1         MX17004  2010      2  d11  29.7  13.4
2         MX17004  2010      2   d2  27.3  14.4
3         MX17004  2010      2  d23  29.9  10.7
4         MX17004  2010      2   d3  24.1  14.4
..            ...   ...    ...  ...   ...   ...
28        MX17004  2010     11  d27  27.7  14.2
29        MX17004  2010     11  d26  28.1  12.1
30        MX17004  2010     11   d4  27.2  12.0
31        MX17004  2010     12   d1  29.9  13.8
32        MX17004  2010     12   d6  27.8  10.5
```

```
[33 rows x 6 columns]
```

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

Click here to view code image

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

print(weather_tidy)
```

```
element        id  year  month  day  tmax  tmin
0         MX17004  2010      1  d30  27.8  14.5
1         MX17004  2010      2  d11  29.7  13.4
2         MX17004  2010      2   d2  27.3  14.4
3         MX17004  2010      2  d23  29.9  10.7
4         MX17004  2010      2   d3  24.1  14.4

..            ...   ...    ...  ...   ...   ...
28        MX17004  2010     11  d27  27.7  14.2
29        MX17004  2010     11  d26  28.1  12.1
30        MX17004  2010     11   d4  27.2  12.0
31        MX17004  2010     12   d1  29.9  13.8
32        MX17004  2010     12   d6  27.8  10.5

[33 rows x 6 columns]
```

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.

5

Apply Functions

Learning about `.apply()` is fundamental in the data cleaning process. It also encapsulates key concepts in programming, mainly writing functions. The `.apply()` method takes a function and applies it (i.e., runs it) across each row or column of a `DataFrame` without having you write the code for each element separately.

If you've programmed before, then the concept of an apply should be familiar. It is similar to writing a `for` loop across each row or column and calling the function, or making a `map()` call to a function. In general, this is the preferred way to apply functions across dataframes, because it typically is much faster than writing a `for` loop in Python.

If you haven't programmed before, then prepare to see how we can easily incorporate custom calculations that can be easily repeated across our data.

Learning Objectives

The concept map for this chapter can be found in Figure A.1.

- Create and use functions
- Use the `.apply()` method to iteratively perform a calculation across `Series` and `DataFrames`
- Identify what parts of a `Series` and `DataFrame` are passed into `.apply()`
- Create vectorized functions using Python decorators

Note About This Chapter

This chapter was also moved up from a later chapter for the second edition. This is one of the few parts of the book that relies on a completely toy example to simplify what is going on. Later on, we will be able to build on the skills taught in this chapter.

5.1 Primer on Functions

Functions are core elements of using the `.apply()` method. There's a lot more information about functions in Appendix O, but here's a quick introduction.

Functions are a way to group and reuse Python code. If you are ever in a situation where you are copying/pasting code and changing a few parts of the code, then chances are, the copied code can be written into a function. To create a function, we need to define it (with the `def` keyword). The body of a function is indented.

The PEP8 Style Guide for Python Code says to use four spaces for an indentation. This book uses two spaces for an indentation because of horizontal space limitations, but I am a new convert to using tabs for indentation because it creates more accessible code and is friendlier for people using Braille readers.[1]

1. Tabs for accessibility: https://alexandersandberg.com/articles/default-to-tabs-instead-of-spaces-for-an-accessible-first-environment/

The basic function skeleton looks like this:

Click here to view code image

```
def my_function(): # define a new function
called my_function
  # indentation for
  # function code
  pass # this statement is here to make a valid
empty function
```

Since Pandas is used for data analysis, let's write some more "useful" functions:

- squares a given value
- takes two numbers and calculates their average

Click here to view code image

```
def my_sq(x):
  """Squares a given value
  """
  return x ** 2


def avg_2(x, y):
  """Calculates the average of 2 numbers
  """
  return (x + y) / 2
```

The text within the triple quotes `"""` is a "docstring." It is the text that appears when you look up the help documentation about a function. You can such docstrings to create your own documentation for functions you write as well.

We've been using functions (and methods) throughout this book. If we want to use functions that we've created ourselves, we can call them just like functions we've loaded from a library.

```
my_calc_1 = my_sq(4)
print(my_calc_1)
```

```
16
```

```
my_calc_2 = avg_2(10, 20)
print(my_calc_2)
```

```
15.0
```

5.2 Apply (Basics)

Now that we know how to write functions, how do we use them in Pandas? When working with `DataFrames`, it's more likely that you want to use a function across rows or columns of your data.

Here's a mock dataframe of two columns.

Click here to view code image

```
import pandas as pd

df = pd.DataFrame({"a": [10, 20, 30], "b": [20,
30, 40]})
print(df)
```

```
    a   b
0  10  20
1  20  30
2  30  40
```

We can `.apply()` our functions over a `Series` (i.e., an individual column or row).

For didactic purposes, let's use the function we wrote to square the `'a'` column. In this overly-simplified example, we could have directly squared the column.

```
print(df['a'] ** 2)
```

```
0    100
1    400
2    900
Name: a, dtype: int64
```

Of course, that would not allow us to use a function we wrote ourselves.

5.2.1 Apply Over a Series

In our example, if we subset a single column or row using a single pair of square brackets, `[ ]`, the `type()` of the object we get back is a Pandas `Series`.

Click here to view code image

```
# get the first column
print(type(df['a']))
```

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

```
# get the first row
print(type(df.iloc[0]))
```

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

The `Series` has a method called `.apply()`.[2] To use the `.apply()` method, we give it the function we want to use across each element in the `Series`.

2. `Series apply` documentation: https://pandas.pydata.org/docs/reference/api/pandas.Series.apply.html

For example, if we want to square each value in column `a`, we can do the following:

Click here to view code image

```
# apply our square function on the 'a' column
sq = df['a'].apply(my_sq)
print(sq)
```

```
0    100
1    400
2    900
Name: a, dtype: int64
```

> **Note**
>
> We do not need the round parentheses, `( )`, when we pass the function into `.apply()`, we pass in `my_sq` instead of `my_sq()`.
>
> In more technical terms, this is called a "function factory," where we are giving `.apply()` a reference to the function we want to use, but we are not invoking the function at this moment.

Let's build on this example by writing a function that takes two parameters. The first parameter will be a value, and the second parameter will be the exponent to which we'll raise the value. So far in our `my_sq()` function, we've "hard-coded" the exponent, `2`, to raise our value.

```
def my_exp(x, e):
  return x ** e
```

Now, if we want to use our function, we have to provide two parameters to it.

Click here to view code image

```
# pass in the exponent, 3
cubed = my_exp(2, 3)
print(cubed)
```

```
8
```

```
# if we don't pass in all the parameters
my_exp(2)
```

```
TypeError: my_exp() missing 1 required positional
argument: 'e'
```

However, if we want to apply the function on our series, we will need to pass in the second parameter. To do this, we pass the second argument as a **keyword argument** into `.apply()`.

Click here to view code image

```
# the exponent, e, to 2
ex = df['a'].apply(my_exp, e=2)
print(ex)
```

```
0    100
1    400
2    900
Name: a, dtype: int64
```

```
# exponent, e, to 3
ex = df['a'].apply(my_exp, e=3)
print(ex)
```

```
0     1000
1     8000
2    27000
Name: a, dtype: int64
```

5.2.2 Apply Over a DataFrame

Now that we've seen how to apply functions over a one-dimensional `Series`, let's see how the syntax changes when we are working with `DataFrames`. Here is the example `DataFrame` from earlier:

Click here to view code image

```python
df = pd.DataFrame({"a": [10, 20, 30], "b": [20,
30, 40]})
print(df)
```

```
    a   b
0  10  20
1  20  30
2  30  40
```

`DataFrames` typically have at least two dimensions. Thus, when we apply a function over a dataframe, we first need to specify which axis to apply the function over–for example, column-by-column or row-by-row.

Let's first write a function that takes a single value and prints out the given value. The function below does not have a `return` statement, All it is doing is displaying on the screen whatever we pass it.

```python
def print_me(x):
  print(x)
```

Let's `.apply()` this function on our dataframe, The syntax is similar to using the `.apply()` method on a `Series`, but this time we need to specify whether we want the function to be applied column-wise or row-wise.

If we want the function to work column-wise, we can pass the `axis=0` or `axis="index"` parameter into `.apply()`. If we want the function to work row-wise, we can pass the `axis=1` or `axis="columns"` parameter into `.apply()`.[3]

3. I find the "index" and "column" text specification for the `axis` parameter counter-intuitive, so I will typically specify using the `0/1` notation with a comment. In practice, you will almost never set `axis=1` or `axis="columns"` for performance reasons.

5.2.2.1 Column-Wise Operations

Use the `axis=0` parameter (the default value) in `.apply()` when working with functions in a column-wise manner (i.e., for each column).

```
df.apply(print_me, axis=0)
```

```
0    10
1    20
2    30
Name: a, dtype: int64
0    20
1    30
2    40
Name: b, dtype: int64
```

	0
a	None
b	None

Compare this output to the following:

```
print(df['a'])
```

```
0    10
1    20
2    30
Name: a, dtype: int64
```

```
print(df['b'])
```

```
0    20
1    30
2    40
Name: b, dtype: int64
```

You can see that the outputs are exactly the same. When you apply a function across a `DataFrame` (in this case, column-wise with `axis=0`), the entire axis (e.g., column) is passed into the first argument of the function. To illustrate this further, let's write a function that calculates the mean (average) of three numbers (each column in our data set contains values).

```
def avg_3(x, y, z):
  return (x + y + z) / 3
```

If we try to apply this function across our columns, we get an error.

Click here to view code image

```
# will cause an error
print(df.apply(avg_3))
```

```
TypeError: avg_3() missing 2 required positional
arguments: 'y' and 'z'
```

From the (last line of the) error message, you can see that the function takes three arguments (`x`, `y`, and `z`), but we failed to pass in the `y` and `z` (i.e., the second and third) arguments. Again, when we use `.apply()`, the **entire** column is passed into the **first** argument. For this function to work with the `.apply()` method, we will have to rewrite parts of it.

Click here to view code image

```
def avg_3_apply(col):
  """The avg_3 function but apply compatible
  by taking in all the values as the first
argument
  and parsing out the values within the function
  """
  x = col[0]
  y = col[1]
```

```
    z = col[2]
    return (x + y + z) / 3
```

```
print(df.apply(avg_3_apply))
```

```
a    20.0
b    30.0
dtype: float64
```

Now that we've rewritten our function to take in all the column values, we get two values back after we apply (one for each column of our `DataFrame`) and each value represents the average of the three values.

5.2.2.2 Row-Wise Operations

Row-wise operations work just like column-wise operations. The part that differs is the axis we use. We will now use `axis=1` in the `.apply()` method. Instead of the entire column being passed into the first argument of the function, the **entire row** is used as the first argument.

Since our example dataframe has two columns and three rows, the `avg_3\apply()` function we just wrote will not work for row-wise operations.

Click here to view code image

```
# will cause an error
print(df.apply(avg_3_apply, axis=1))
```

```
IndexError: index 2 is out of bounds for axis 0
with size 2
```

The main issue here is the `'index out of bounds'`. We passed the row of data in as the first argument, but in our function we begin indexing out of range (i.e., we have only two values in each row, but we tried to get index `2`, which means the third element, and it does not exist).

If we wanted to calculate our averages row-wise, we would have to write a new function to work with two values.

Click here to view code image

```
def avg_2_apply(row):
  """Taking the average of row value.
  Assuming that there are only 2 values in a
row.
  """
  x = row[0]
  y = row[1]
  return (x + y) / 2

print(df.apply(avg_2_apply, axis=0))
```

```
a    15.0
b    25.0
dtype: float64
```

5.3 Vectorized Functions

When we use `.apply()`, we are able to make a function work on a column-by-column or row-by-row basis. In the previous section, Section 5.2, we had to rewrite our function when we wanted to apply it because the entire column or row was passed into the first parameter of the function. However, there might be times when it is not feasible to rewrite a function in this way. We can leverage the `.vectorize()` function and decorator to vectorize any function. Vectorizing your code can also lead to performance gains (Appendix V).

Here's our toy dataframe:

Click here to view code image

```
df = pd.DataFrame({"a": [10, 20, 30], "b": [20,
30, 40]})
print(df)
```

```
    a   b
0  10  20
1  20  30
2  30  40
```

And here's our average function, which we can apply on a row-by-row basis:

```
def avg_2(x, y):
  return (x + y) / 2
```

For a vectorized function, we'd like to be able to pass in a vector of values for `x` and a vector of values for `y`, and the results should be the average of the given `x` and `y` values in the same order. In other words, we want to be able to write `avg_2(df['a'], df['y'])` and get `[15, 25, 35]` as a result.

Click here to view code image

```
print(avg_2(df['a'], df['b']))
```

```
0    15.0
1    25.0
2    35.0
dtype: float64
```

This approach works because the actual calculations within our function are inherently vectorized. That is, if we add two numeric columns together, Pandas (and the NumPy library) will automatically perform element-wise addition. Likewise, when we divide by a scalar, it will "broadcast" the scalar, and divide each element by the scalar.

Let's change our function and perform a non-vectorizable calculation.

Click here to view code image

```
import numpy as np

def avg_2_mod(x, y):
    """Calculate the average, unless x is 20
    If the value is 20, return a missing value
    """
    if (x == 20):
        return(np.NaN)
    else:
        return (x + y) / 2
```

If we run this function, it will cause an error.

Click here to view code image

```
# will cause an error
print(avg_2_mod(df['a'], df['b']))
```

```
ValueError: The truth value of a Series is
ambiguous. Use a.empty,
a.bool(), a.item(), a.any() or a.all().
```

However, if we give it individual numbers instead of a vector, it will work as expected.

```
print(avg_2_mod(10, 20))
```

```
15.0
```

```
print(avg_2_mod(20, 30))
```

```
nan
```

5.3.1 Vectorize with NumPy

We want to change our function so that when it is given a vector of values, it will perform the calculations in an element-wise manner. We can do this by using the `vectorize()` function from `numpy`. We pass `np.vectorize()` to the **function** we want to vectorize, to create a new function.

Click here to view code image

```
import numpy as np

# np.vectorize actually creates a new function
avg_2_mod_vec = np.vectorize(avg_2_mod)

# use the newly vectorized function
print(avg_2_mod_vec(df['a'], df['b']))
```

```
[15. nan 35.]
```

This method works well if you do not have the source code for an existing function. However, if you are writing your own function, you can use a Python decorator to automatically vectorize the function without having to create a new function. A decorator is a function that takes another function as input, and modifies how that function's output behaves.

Click here to view code image

```
# to use the vectorize decorator
# we use the @ symbol before our function
definition
@np.vectorize
def v_avg_2_mod(x, y):
  """Calculate the average, unless x is 20
```

```
  Same as before, but we are using the vectorize
decorator
  """
  if (x == 20):
    return(np.NaN)
  else:
    return (x + y) / 2

# we can then directly use the vectorized
function
# without having to create a new function
print(v_avg_2_mod(df['a'], df['b']))
```

```
[15. nan 35.]
```

5.3.2 Vectorize with Numba

The `numba` library[4] is designed to optimize Python code, especially calculations on arrays performing mathematical calculations. Just like `numpy`, it also has a `vectorize` decorator.

4. numba: https://numba.pydata.org/

Click here to view code image

```
import numba

@numba.vectorize
def v_avg_2_numba(x, y):
  """Calculate the average, unless x is 20
  Using the numba decorator.
  """
  # we now have to add type information to our
function
```

```
  if (int(x) == 20):
    return(np.NaN)
  else:
    return (x + y) / 2
```

The numba library is so optimized that it does not understand Pandas objects.

Click here to view code image

```
print(v_avg_2_numba(df['a'], df['b']))
```

```
ValueError: Cannot determine Numba type of
<class 'pandas.core.series.Series'>
```

We actually have to pass in the numpy array representation of our data using the .values attribute of our Series objects (Chapter R).

Click here to view code image

```
# passing in the numpy array
print(v_avg_2_numba(df['a'].values,
df['b'].values))
```

```
[15. nan 35.]
```

5.4 Lambda Functions (Anonymous Functions)

Sometimes the function used in the .apply() method is simple enough that there is no need to create a separate function.

Let's look at our simple DataFrame example and our squaring function again.

Click here to view code image

```
df = pd.DataFrame({'a': [10, 20, 30],
                   'b': [20, 30, 40]})
print(df)
```

```
    a   b
0  10  20
1  20  30
2  30  40
```

```
def my_sq(x):
  return x ** 2

df['a_sq'] = df['a'].apply(my_sq)
print(df)
```

```
    a   b  a_sq
0  10  20   100
1  20  30   400
2  30  40   900
```

You can see that the actual function is a simple one-liner. Usually when this happens, people will opt to write the one-liner directly in the `apply` method. This method is called using **lambda functions**. We can perform the same operation as shown earlier in the following manner.

Click here to view code image

```
df['a_sq_lamb'] = df['a'].apply(lambda x: x **
2)
print(df)
```

```
    a   b  a_sq  a_sq_lamb
0  10  20   100        100
```

```
1   20   30    400            400
2   30   40    900            900
```

To write the lambda function, we use the `lambda` keyword. Since apply functions will pass the entire axis as the first argument, our `lambda` function example takes only one parameter, `x`. The `x` in `lambda x` is analogous to the `x` in `def my_sq(x)`, each value in the `'a'` column will be individually passed into our `lambda` function. We can then write our function directly, without having to define it. The calculated result is automatically returned.

Although you can write complex multiple-line lambda functions, typically people will use the lambda function approach when small one-liner calculations are needed. The code can become hard to read if the lambda function tries to do too much at once.

Conclusion

This chapter covered an important concept – namely, creating functions that can be used on our data. Not all data cleaning steps or manipulations can be done using built-in functions. There will be many times when you will have to write your own custom functions to process and analyze data.

This chapter uses oversimplified examples to create and use functions, but that means we can go into more complex examples as we learn more about the `pandas` library.

Part II

Data Processing

Now that we know the basics of working with our data, we can go into more detail on how to process it. Data does not always come in one part. We begin with combining multiple data sets, by either concatenating it together or joining them by values (Chapter 6). Combining data is usually something we do in the tidying process (Chapter 4), but normalizing data is the process of splitting it up into separate parts. It seems counterintuitive to split data up, but this is something that is typically done for data storage, especially for databases (Chapter 7). Finally, we go into more detail into grouped operations (Chapter 8) that were first introduced in Chapter 1.

6

Data Assembly

By now, you should be able to load data into `pandas` and do some basic visualizations. This part of the book focuses on various data cleaning tasks. We begin with assembling a data set for analysis by combining various data sets together.

Learning Objectives

- Identify when needs to be combined
- Identify whether data needs to be concatenated or joined together
- Use the appropriate function or methods to combine multiple data sets
- Produce a single data set from multiple files
- Assess whether data was joined properly

6.1 Combine Data Sets

We first talked about tidy data principles in Chapter 4. This chapter will cover the third criterion in the original "Tidy Data" paper[1]: "each type of observational unit forms a table."

1. Tidy Data paper: `http://vita.had.co.nz/papers/tidy-data.pdf`

When data is tidy, you need to combine various tables together to answer a question. For example, there may be a separate table holding company information and another table holding stock prices. If we want to look at all the stock prices within the tech industry, we may first have to find all the tech companies from the company information table, and then combine that data with the stock price data to get the data we need for our question. The data may have been split up into separate tables to reduce the amount of redundant information (we don't need to store the company information with each stock price entry), but this arrangement means we as data analysts must combine the relevant data ourselves to answer our question.

Other times, a single data set may be split into multiple parts. For example, with timeseries data, each date may be in a separate file. In another case, a file may have been split into parts to make the individual files smaller. You may also need to combine data from multiple sources to answer a question (e.g., combine latitudes and longitudes with zip codes). In both cases, you will need to combine data into a single dataframe for analysis.

6.2 Concatenation

One of the (conceptually) easier ways to combine data is with concatenation. Concatenation can be thought of as appending a row or column to your data. This approach is possible if your data was split into parts or if you performed a calculation that you want to append to your existing data set.

Let's begin with some example data sets so you can see what is actually happening.

Click here to view code image

```
import pandas as pd

df1 = pd.read_csv('data/concat_1.csv')
df2 = pd.read_csv('data/concat_2.csv')
df3 = pd.read_csv('data/concat_3.csv')
```

```
print(df1)
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
```

```
2   a2   b2   c2   d2
3   a3   b3   c3   d3
```

```
print(df2)
```

```
    A    B    C    D
0   a4   b4   c4   d4
1   a5   b5   c5   d5
2   a6   b6   c6   d6
3   a7   b7   c7   d7
```

```
print(df3)
```

```
      A     B     C     D
0    a8    b8    c8    d8
1    a9    b9    c9    d9
2   a10   b10   c10   d10
3   a11   b11   c11   d11
```

Concatenation is accomplished by using the `concat()` function from Pandas.

6.2.1 Review Parts of a DataFrame

Section 2.3.1 talked about the three parts of a dataframe: `.index`, `.columns`, and `.values`. We will be working with `.index` and `.columns` a lot in this chapter.

The `.index` refers to the labels on the left of the dataframe, by default they will be numbered starting from `0`.

Click here to view code image

```
print(df1.index)
```

```
RangeIndex(start=0, stop=4, step=1)
```

The “index” is an “axis” of a dataframe. These terms are important because pandas will try to automatically align by axis. The other axis is the “columns,” which we can get with `.columns`.

Click here to view code image

```
print(df1.columns)
```

```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

This refers to the column names of the dataframe.

Finally, just to be complete, the body of the dataframe can be represented as an `numpy` array with `.values`.

```
print(df1.values)
```

```
[['a0'  'b0'  'c0'  'd0']
 ['a1'  'b1'  'c1'  'd1']
 ['a2'  'b2'  'c2'  'd2']
 ['a3'  'b3'  'c3'  'd3']]
```

6.2.2 Add Rows

Stacking (i.e., concatenating) the dataframes on top of each other uses the `concat()` function in `pandas`. All of the dataframes to be concatenated are passed in a `list`.

Click here to view code image

```
row_concat = pd.concat([df1, df2, df3])
print(row_concat)
```

```
      A    B    C    D
0    a0   b0   c0   d0
1    a1   b1   c1   d1
2    a2   b2   c2   d2
3    a3   b3   c3   d3
0    a4   b4   c4   d4
..  ...   ... ...  ...
3    a7   b7   c7   d7
0    a8   b8   c8   d8
1    a9   b9   c9   d9
2   a10  b10  c10  d10
3   a11  b11  c11  d11

[12 rows x 4 columns]
```

As you can see, `concat()` blindly stacks the dataframes together. If you look at the row names (i.e., the row indices), they are also simply a stacked version of the original row indices. If we apply the various subsetting methods (Table 2.3), the table will be subsetted as expected.

Click here to view code image

```
# subset the fourth row of the concatenated
dataframe
print(row_concat.iloc[3, :])
```

```
A    a3
B    b3
C    c3
D    d3
Name: 3, dtype: object
```

Question

What happens when you use `.loc[]` to subset the new dataframe?

Section 2.1.1 showed the process for creating a `Series`. However, if we create a new series to append to a dataframe, it does not append correctly.

Click here to view code image

```
# create a new row of data
new_row_series = pd.Series(['n1', 'n2', 'n3',
'n4'])
print(new_row_series)
```

```
0    n1
1    n2
2    n3
3    n4
dtype: object
```

```
# attempt to add the new row to a dataframe
print(pd.concat([df1, new_row_series]))
```

```
     A    B    C    D    0
0   a0   b0   c0   d0  NaN
1   a1   b1   c1   d1  NaN
2   a2   b2   c2   d2  NaN
3   a3   b3   c3   d3  NaN
0  NaN  NaN  NaN  NaN   n1
1  NaN  NaN  NaN  NaN   n2
2  NaN  NaN  NaN  NaN   n3
3  NaN  NaN  NaN  NaN   n4
```

The first things you may notice are the `NaN` missing values. This is simply Python's way of representing a "missing value" (more about missing values in Chapter 9). We were hoping to append our new values as a row, but that didn't happen. In fact, not only did our code not append the values as a row, but it also created a new column completely misaligned with everything else.

Let's think about what is happening here. First, our series did not have a matching column, so our `new_row` was added to a new column. The rest of the values were concatenated to the bottom of the dataframe, and the original index values were retained.

To fix this problem, we need turn our series into a dataframe. This data frame contains one row of data, and the column names are the ones the data will bind to.

Click here to view code image

```
new_row_df = pd.DataFrame(
  # note the double brackets to create a "row"
of data
  data=[["n1", "n2", "n3", "n4"]],
  columns=["A", "B", "C", "D"],
)

print(new_row_df)
```

```
    A    B    C    D
0  n1   n2   n3   n4
```

```
# concatenate the row of data
print(pd.concat([df1, new_row_df]))
```

```
    A    B    C    D
0  a0   b0   c0   d0
1  a1   b1   c1   d1
2  a2   b2   c2   d2
```

```
3   a3    b3    c3    d3
0   n1    n2    n3    n4
```

`concat()` is a general function that can concatenate multiple things at once.

6.2.2.1 Ignore the Index

In the last example, when we added a `dict` to a dataframe, we had to use the `ignore_index` parameter. If we look closer, you can see that the row index was also incremented by 1, and did not repeat a previous index value.

If we simply want to concatenate or append data together, we can use the `ignore_index` parameter to reset the row index after the concatenation.

Click here to view code image

```
row_concat_i = pd.concat([df1, df2, df3],
ignore_index=True)
print(row_concat_i)
```

```
       A     B     C     D
0     a0    b0    c0    d0
1     a1    b1    c1    d1
2     a2    b2    c2    d2
3     a3    b3    c3    d3
4     a4    b4    c4    d4
..   ...   ...   ...   ...
7     a7    b7    c7    d7
8     a8    b8    c8    d8
9     a9    b9    c9    d9
10   a10   b10   c10   d10
11   a11   b11   c11   d11
```

```
[12 rows x 4 columns]
```

6.2.3 Add Columns

Concatenating columns is very similar to concatenating rows. The main difference is the `axis` parameter in the `concat` function. The default value of `axis` is `0` (or `"index"`), so it will concatenate data in a row-wise fashion. However, if we pass `axis=1` (or `axis="columns"`) to the function, it will concatenate data in a column-wise manner.

Click here to view code image

```
col_concat = pd.concat([df1, df2, df3],
axis="columns")
print(col_concat)
```

```
    A   B   C   D   A   B   C   D    A    B    C
D
0   a0  b0  c0  d0  a4  b4  c4  d4   a8   b8   c8
d8
1   a1  b1  c1  d1  a5  b5  c5  d5   a9   b9   c9
d9
2   a2  b2  c2  d2  a6  b6  c6  d6  a10  b10  c10
d10
3   a3  b3  c3  d3  a7  b7  c7  d7  a11  b11  c11
d11
```

If we try to subset data based on column names, we will get a similar result when we concatenated row-wise and subset by row index.

```
print(col_concat['A'])
```

```
    A   A    A
0  a0  a4   a8
1  a1  a5   a9
2  a2  a6  a10
3  a3  a7  a11
```

Adding a single column to a dataframe can be done directly without using any specific Pandas function (We saw this in Section 2.4.1). Simply pass a new column name for the vector you want assigned to the new column.

Click here to view code image

```
col_concat['new_col_list'] = ['n1', 'n2', 'n3',
'n4']
print(col_concat)
```

```
    A   B   C   D   A   B   C   D    A    B    C
D  new_col_list
0  a0  b0  c0  d0  a4  b4  c4  d4   a8   b8   c8
d8           n1
1  a1  b1  c1  d1  a5  b5  c5  d5   a9   b9   c9
d9           n2
2  a2  b2  c2  d2  a6  b6  c6  d6  a10  b10  c10
d10           n3
3  a3  b3  c3  d3  a7  b7  c7  d7  a11  b11  c11
d11           n4
```

Click here to view code image

```
col_concat['new_col_series'] = pd.Series(['n1',
'n2', 'n3', 'n4'])
print(col_concat)
```

```
    A    B    C    D    A    B    C    D     A     B     C
D  new_col_list \
0  a0   b0   c0   d0   a4   b4   c4   d4    a8    b8    c8
d8              n1
1  a1   b1   c1   d1   a5   b5   c5   d5    a9    b9    c9
d9              n2
2  a2   b2   c2   d2   a6   b6   c6   d6   a10   b10   c10
d10              n3
3  a3   b3   c3   d3   a7   b7   c7   d7   a11   b11   c11
d11              n4

  new_col_series
0              n1
1              n2
2              n3
3              n4
```

Using the `concat()` function still works, as long as you give it a dataframe. However this approach requires more code.

Finally, we can reset the column indices so we do not have duplicated column names.

Click here to view code image

```
print(pd.concat([df1, df2, df3], axis="columns",
ignore_index=True))
```

```
    0    1    2    3    4    5    6    7     8     9    10
11
0  a0   b0   c0   d0   a4   b4   c4   d4    a8    b8    c8
d8
1  a1   b1   c1   d1   a5   b5   c5   d5    a9    b9    c9
d9
2  a2   b2   c2   d2   a6   b6   c6   d6   a10   b10   c10
```

```
d10
3   a3   b3   c3   d3   a7   b7   c7   d7   a11   b11   c11
d11
```

6.2.4 Concatenate with Different Indices

The examples shown so far have assumed we are performing a row or column concatenation. They also assume that the new row(s) had the same column names or the column(s) had the same row indices.

This section addresses what happens when the row and column indices are not aligned.

6.2.4.1 Concatenate Rows with Different Columns

Let's modify our dataframes for the next few examples.

Click here to view code image

```
# rename the columns of our dataframes
df1.columns = ['A', 'B', 'C', 'D']
df2.columns = ['E', 'F', 'G', 'H']
df3.columns = ['A', 'C', 'F', 'H']

print(df1)
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

```
print(df2)
```

```
    E   F   G   H
0  a4  b4  c4  d4
1  a5  b5  c5  d5
2  a6  b6  c6  d6
3  a7  b7  c7  d7
```

```
print(df3)
```

```
     A    C    F    H
0   a8   b8   c8   d8
1   a9   b9   c9   d9
2  a10  b10  c10  d10
3  a11  b11  c11  d11
```

If we try to concatenate these dataframes as we did in Section 6.2.2, the dataframes now do much more than simply stack one on top of the other. The columns align themselves, and `NaN` fills in any missing areas.

Click here to view code image

```
row_concat = pd.concat([df1, df2, df3])
print(row_concat)
```

```
     A    B    C    D    E    F    G    H
0   a0   b0   c0   d0  NaN  NaN  NaN  NaN
1   a1   b1   c1   d1  NaN  NaN  NaN  NaN
2   a2   b2   c2   d2  NaN  NaN  NaN  NaN
3   a3   b3   c3   d3  NaN  NaN  NaN  NaN
0  NaN  NaN  NaN  NaN   a4   b4   c4   d4
..  ...  ...  ...  ...  ...  ...  ...  ...
3  NaN  NaN  NaN  NaN   a7   b7   c7   d7
0   a8  NaN   b8  NaN  NaN   c8  NaN   d8
1   a9  NaN   b9  NaN  NaN   c9  NaN   d9
```

```
2   a10    NaN    b10    NaN    NaN    c10    NaN    d10
3   a11    NaN    b11    NaN    NaN    c11    NaN    d11

[12 rows x 8 columns]
```

One way to avoid the inclusion of `NaN` values is to keep only those columns that are shared in common by the list of objects to be concatenated. A parameter named `join` accomplishes this. By default, it has a value of `'outer'`, meaning it will keep all the columns. However, we can set `join='inner'` to keep only the columns that are shared among the data sets.

If we try to keep only the columns from all three dataframes, we will get an empty dataframe, since there are no columns in common.

Click here to view code image

```
print(pd.concat([df1, df2, df3], join='inner'))
```

```
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]

[12 rows x 0 columns]
```

If we use the dataframes that have columns in common, only the columns that all of them share will be returned.

Click here to view code image

```
print(pd.concat([df1,df3], ignore_index=False,
join='inner'))
```

```
    A    C
0   a0   c0
```

```
1   a1   c1
2   a2   c2
3   a3   c3
0   a8   b8
1   a9   b9
2  a10  b10
3  a11  b11
```

6.2.4.2 Concatenate Columns with Different Rows

Let's take our dataframes and modify them again so that they have different row indices. Here, we are building on the same dataframe modifications from Section 6.2.4.1.

```
df1.index = [0, 1, 2, 3]
df2.index = [4, 5, 6, 7]
df3.index = [0, 2, 5, 7]
```

```
print(df1)
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

```
print(df2)
```

```
    E   F   G   H
4  a4  b4  c4  d4
5  a5  b5  c5  d5
6  a6  b6  c6  d6
7  a7  b7  c7  d7
```

```
print(df3)
```

```
      A    C    F    H
0    a8   b8   c8   d8
2    a9   b9   c9   d9
5   a10  b10  c10  d10
7   a11  b11  c11  d11
```

When we concatenate along `axis="columns"` (`axis=1`), the new dataframes will be added in a column-wise fashion and matched against their respective row indices. Missing values indicators appear in the areas where the indices did not align.

Click here to view code image

```
col_concat = pd.concat([df1, df2, df3],
axis="columns")
print(col_concat)
```

```
      A    B    C    D    E    F    G    H    A   C
F   H
0    a0   b0   c0   d0  NaN  NaN  NaN  NaN   a8
b8   c8   d8
1    a1   b1   c1   d1  NaN  NaN  NaN  NaN  NaN
NaN  NaN  NaN
2    a2   b2   c2   d2  NaN  NaN  NaN  NaN   a9
b9   c9   d9
3    a3   b3   c3   d3  NaN  NaN  NaN  NaN  NaN
NaN  NaN  NaN
4   NaN  NaN  NaN  NaN   a4   b4   c4   d4  NaN
NaN  NaN  NaN
5   NaN  NaN  NaN  NaN   a5   b5   c5   d5  a10
b10  c10  d10
6   NaN  NaN  NaN  NaN   a6   b6   c6   d6  NaN
```

```
NaN   NaN   NaN
7   NaN   NaN   NaN   NaN    a7    b7    c7    d7   a11
b11   c11   d11
```

Just as we did when we concatenated in a row-wise manner, we can choose to keep the results only when there are matching indices by using `join="inner"`.

Click here to view code image

```
print(pd.concat([df1, df3], axis="columns",
join='inner'))
```

```
    A    B    C    D    A    C    F    H
0   a0   b0   c0   d0   a8   b8   c8   d8
2   a2   b2   c2   d2   a9   b9   c9   d9
```

6.3 Observational Units Across Multiple Tables

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.

In this example, all of the billboard ratings data have a pattern.

Click here to view code image

```
data/billboard-by_week/billboard-XX.csv
```

Where `XX` represents the week (e.g., 03). We can use the a pattern matching function from the built-in `pathlib` module in Python to get a list of all the filenames that match a particular pattern.

Click here to view code image

```
from pathlib import Path

# from my current directory fine (glob) the this
pattern

billboard_data_files = (
    Path(".")
    .glob("data/billboard-by_week/billboard-
*.csv")
)

# this line is optional if you want to see the
full list of files
billboard_data_files =
sorted(list(billboard_data_files))

print(billboard_data_files)
```

```
[PosixPath('data/billboard-by_week/billboard-
01.csv'),
PosixPath('data/billboard-by_week/billboard-
02.csv'),
PosixPath('data/billboard-by_week/billboard-
03.csv'),
PosixPath('data/billboard-by_week/billboard-
04.csv'),
PosixPath('data/billboard-by_week/billboard-
05.csv'),
```

```
..    ...                    ...                     ...
...
PosixPath('data/billboard-by_week/billboard-
72.csv'),
PosixPath('data/billboard-by_week/billboard-
73.csv'),
PosixPath('data/billboard-by_week/billboard-
74.csv'),
PosixPath('data/billboard-by_week/billboard-
75.csv'),
PosixPath('data/billboard-by_week/billboard-
76.csv')]
```

The `type()` of `billboard_data_files` is a generator object, so if you "use it" you will lose its contents. If you want to see the full list, you would need to run:

Click here to view code image

```
billboard_data_files = list(billboard_data_files)
```

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.

Click here to view code image

```
billboard01 =
pd.read_csv(billboard_data_files[0])
billboard02 =
pd.read_csv(billboard_data_files[1])
billboard03 =
pd.read_csv(billboard_data_files[2])
```

```
# just look at one of the data sets we loaded
print(billboard01)
```

```
     year            artist
track   time  \
0    2000             2 Pac  Baby Don't Cry
(Keep...   4:22
1    2000           2Ge+her  The Hardest Part Of
...   3:15
2    2000      3 Doors Down
Kryptonite   3:53
3    2000      3 Doors Down
Loser   4:24
4    2000          504 Boyz             Wobble
Wobble  3:35
..    ...               ...
...   ...
312  2000       Yankee Grey      Another Nine
Minutes  3:10
313  2000  Yearwood, Trisha           Real Live
Woman  3:55
314  2000   Ying Yang Twins   Whistle While You
Tw...  4:19
315  2000     Zombie Nation             Kernkraft
400  3:30
316  2000   matchbox twenty
Bent  4:12

   date.entered week  rating
0    2000-02-26  wk1    87.0
1    2000-09-02  wk1    91.0
2    2000-04-08  wk1    81.0
3    2000-10-21  wk1    76.0
```

```
4    2000-04-15  wk1    57.0
..          ...  ...     ...
312  2000-04-29  wk1    86.0
313  2000-04-01  wk1    85.0
314  2000-03-18  wk1    95.0
315  2000-09-02  wk1    99.0
316  2000-04-29  wk1    60.0

[317 rows x 7 columns]
```

We can concatenate them just as we did in Chapter 6.

Click here to view code image

```
# shape of each dataframe
print(billboard01.shape)
print(billboard02.shape)
print(billboard03.shape)
```

```
(317, 7)
(317, 7)
(317, 7)
```

```
# concatenate the dataframes together
billboard = pd.concat([billboard01, billboard02,
billboard03])

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

```
(951, 7)
```

Let's write a check to make sure the number of rows were concatenated correctly

```
assert (
    billboard01.shape[0]
    + billboard02.shape[0]
    + billboard03.shape[0]
    == billboard.shape[0]
)
```

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.

6.3.1 Load Multiple Files Using a Loop

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

Click here to view code image

```
# this part was the same as earlier
from pathlib import Path
billboard_data_files = (
    Path(".")
    .glob("data/billboard-by_week/billboard-
*.csv")
)

# create an empty list to append to
list_billboard_df = []

# loop though each CSV filename
for csv_filename in billboard_data_files:
```

```
    # 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_billboard_df.append(df)

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

```
76
```

> **Important**
>
> The `Path.glob()` method returns a generator (Appendix P). This means that when we go through each element of the "list," the item gets "used up," so it won't exist again. This saves a lot of compute resources since Python does not need to store everything in memory all at once. The downside is you will need to re-create the generator if you plan on using it multiple times. You can opt to turn the generator into a regular python list so all the elements are stored perpetually by using the `list()` function, e.g., `list(billboard_data_files)`.

Click here to view code image

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

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

Click here to view code image

```
# look at the first dataframe
print(list_billboard_df[0])
```

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

..    ...               ...
...   ...
312  2000       Yankee Grey     Another Nine
Minutes  3:10
313  2000  Yearwood, Trisha          Real Live
Woman  3:55
314  2000   Ying Yang Twins  Whistle While You
Tw...  4:19
315  2000     Zombie Nation            Kernkraft
400  3:30
316  2000   matchbox twenty
Bent  4:12

   date.entered  week  rating
```

```
0    2000-02-26  wk15     NaN
1    2000-09-02  wk15     NaN
2    2000-04-08  wk15    38.0
3    2000-10-21  wk15    72.0
4    2000-04-15  wk15    78.0
..          ...   ...     ...
312  2000-04-29  wk15     NaN
313  2000-04-01  wk15     NaN
314  2000-03-18  wk15     NaN
315  2000-09-02  wk15     NaN
316  2000-04-29  wk15     3.0

[317 rows x 7 columns]
```

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

Click here to view code image

```
billboard_loop_concat =
pd.concat(list_billboard_df)
print(billboard_loop_concat.shape)
```

```
(24092, 7)
```

6.3.2 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 K).

Click here to view code image

```
# we have to re-create the generator because we
# "used it up" in the previous example
billboard_data_files = (
    Path(".")
    .glob("data/billboard-by_week/billboard-
*.csv")
)
# the loop code without comments
list_billboard_df = []
for csv_filename in billboard_data_files:
    df = pd.read_csv(csv_filename)
    list_billboard_df.append(df)

billboard_data_files = (
    Path(".")
    .glob("data/billboard-by_week/billboard-
*.csv")
)

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

Warning

If you get a `ValueError: No objects to concatenate` message, it means you did not re-create the `billboard_data_files` generator.

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

```
print(type(billboard_dfs))
```

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

```
print(len(billboard_dfs))
```

```
76
```

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

Click here to view code image

```
billboard_concat_comp = pd.concat(billboard_dfs)
```

```
print(billboard_concat_comp)
```

```
     year              artist
track  time  \
0    2000               2 Pac  Baby Don't Cry
(Keep...  4:22
1    2000             2Ge+her  The Hardest Part Of
...  3:15
2    2000       3 Doors Down
Kryptonite  3:53
3    2000       3 Doors Down
Loser  4:24
4    2000            504 Boyz            Wobble
Wobble  3:35
..    ...                 ...
...   ...
312  2000         Yankee Grey     Another Nine
Minutes  3:10
313  2000   Yearwood, Trisha          Real Live
Woman  3:55
314  2000    Ying Yang Twins  Whistle While You
Tw...  4:19
```

```
315  2000     Zombie Nation             Kernkraft
400  3:30
316  2000   matchbox twenty
Bent  4:12

   date.entered  week  rating
0    2000-02-26  wk15     NaN
1    2000-09-02  wk15     NaN
2    2000-04-08  wk15    38.0
3    2000-10-21  wk15    72.0
4    2000-04-15  wk15    78.0
..          ...   ...     ...
312  2000-04-29  wk18    NaN
313  2000-04-01  wk18    NaN
314  2000-03-18  wk18    NaN
315  2000-09-02  wk18    NaN
316  2000-04-29  wk18    3.0

[24092 rows x 7 columns]
```

6.4 Merge Multiple Data Sets

The previous section alluded to a few database concepts. The `join="inner"` and the default `join="outer"` parameters come from working with databases when we want to merge tables.

Instead of simply having a row or column index that you want to use to concatenate values, sometimes you may have two or more dataframes that you want to combine based on common data values. This task is known in the database world as performing a "join."

Pandas has a `.join()` method that uses `.merge()` under the hood. `.join()` will merge dataframe objects based on an index, but the `.merge()` function is much more explicit and flexible.

If you are planning to merge dataframes by the row index, for example, you might want to look into the `.join()` method.[2]

2. Pandas `DataFrame.join()` method:
`https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.join.html`

We will be using the set of survey data in the following examples.

Click here to view code image

```
person = pd.read_csv('data/survey_person.csv')
site = pd.read_csv('data/survey_site.csv')
survey = pd.read_csv('data/survey_survey.csv')
visited = pd.read_csv('data/survey_visited.csv')
```

```
print(person)
```

```
      ident   personal   family
0      dyer    William     Dyer
1        pb      Frank  Pabodie
2      lake   Anderson     Lake
3       roe  Valentina  Roerich
4  danforth      Frank Danforth
```

```
print(site)
```

```
    name    lat    long
0   DR-1 -49.85 -128.57
1   DR-3 -47.15 -126.72
2  MSK-4 -48.87 -123.40
```

```
print(visited)
```

```
  ident  site       dated
0   619  DR-1  1927-02-08
1   622  DR-1  1927-02-10
```

```
2   734  DR-3  1939-01-07
3   735  DR-3  1930-01-12
4   751  DR-3  1930-02-26
5   752  DR-3         NaN
6   837 MSK-4  1932-01-14
7   844  DR-1  1932-03-22
```

```
print(survey)
```

```
   taken person  quant reading
0    619   dyer    rad    9.82
1    619   dyer    sal    0.13
2    622   dyer    rad    7.80
3    622   dyer    sal    0.09
4    734     pb    rad    8.41
..   ...    ...    ...     ...
16   752    roe    sal   41.60
17   837   lake    rad    1.46
18   837   lake    sal    0.21
19   837    roe    sal   22.50
20   844    roe    rad   11.25

[21 rows x 4 columns]
```

Currently, our data is split into multiple parts, where each part is an observational unit. If we wanted to look at the dates at each site along with the latitude and longitude information for that site, we would have to combine (and merge) multiple dataframes. We can do this with the `.merge()` method in Pandas.

When we call this method, the dataframe that is called will be referred to as the one on the "left." Within the `.merge()` method, the first parameter is the "right" dataframe (i.e., `left.merge(right)`). The next parameter is `how` the final merged result looks.

Table 6.1 provides more details. Next, we set the `on` parameter. This specifies which columns to match on. If the left and right columns do not have the same name, we can use the `left_on` and `right_on` parameters instead.

Table 6.1 How the Pandas how Parameter Relates to SQL

Pandas	SQL	Description
left	left outer	Keep all the keys from the left
right	right outer	Keep all the keys from the right
outer	full outer	Keep all the keys from both left and right
inner	inner	Keep only the keys that exist in both left and right

6.4.1 One-to-One Merge

In the simplest type of merge, we have two dataframes where we want to join one column to another column, and where the columns we want to join do not contain any duplicate values.

For this example, we will modify the `visited` dataframe so there are no duplicated `site` values.

Click here to view code image

```
visited_subset = visited.loc[[0, 2, 6], :]
print(visited_subset)
```

```
 ident   site       dated
0  619   DR-1  1927-02-08
2  734   DR-3  1939-01-07
6  837  MSK-4  1932-01-14
```

```
# get a count of the values in the site column
print(
```

```
    visited_subset["site"].value_counts()
)
```

```
DR-1     1
DR-3     1
MSK-4    1
Name: site, dtype: int64
```

We can perform our one-to-one merge as follows:

Click here to view code image

```
# the default value for 'how' is 'inner'
# so it doesn't need to be specified
o2o_merge = site.merge(
    visited_subset, left_on="name",
right_on="site"
)
print(o2o_merge)
```

```
    name    lat    long  ident   site       dated
0   DR-1 -49.85 -128.57    619   DR-1  1927-02-08
1   DR-3 -47.15 -126.72    734   DR-3  1939-01-07
2  MSK-4 -48.87 -123.40    837  MSK-4  1932-01-14
```

As you can see, we have now created a new dataframe from two separate dataframes where the rows were matched based on a particular set of columns. In SQL-speak, the columns used to match are called "keys."

6.4.2 Many-to-One Merge

If we choose to do the same merge, but this time without using the subsetted `visited` dataframe, we would perform a many-to-one merge. In this kind of merge, one of the dataframes has key values that repeat.

Click here to view code image

```
# get a count of the values in the site column
print(
  visited["site"].value_counts()
)
```

```
DR-3     4
DR-1     3
MSK-4    1
Name: site, dtype: int64
```

The dataframes that contain the single observations will then be duplicated in the merge.

Click here to view code image

```
m2o_merge = site.merge(visited, left_on='name',
right_on='site')
print(m2o_merge)
```

```
    name    lat    long  ident   site       dated
0   DR-1 -49.85 -128.57    619   DR-1  1927-02-08
1   DR-1 -49.85 -128.57    622   DR-1  1927-02-10
2   DR-1 -49.85 -128.57    844   DR-1  1932-03-22
3   DR-3 -47.15 -126.72    734   DR-3  1939-01-07
4   DR-3 -47.15 -126.72    735   DR-3  1930-01-12
5   DR-3 -47.15 -126.72    751   DR-3  1930-02-26
6   DR-3 -47.15 -126.72    752   DR-3         NaN
7  MSK-4 -48.87 -123.40    837  MSK-4  1932-01-14
```

The site information (name, lat, and long) were duplicated and matched to the visited data.

6.4.3 Many-to-Many Merge

Lastly, there will be times when we want to perform a match based on multiple columns. As an example, suppose we have two dataframes that come from person merged with survey, and another dataframe that comes from visited merged with survey.

> **Danger**
>
> All the code for performing a merge uses the same method, .merge(). The only thing that makes the results differ is whether or not the left and/or right dataframe has duplicate keys.
>
> In practice, you usually do not want a many-to-many merge. Since that means a cartesian product of the keys were joined together. That is, every combination of duplicated values were combined.

Click here to view code image

```
ps = person.merge(survey, left_on='ident',
right_on='person')
vs = visited.merge(survey, left_on='ident',
right_on='taken')
```

```
print(ps)
```

```
   ident   personal   family taken person quant
reading
0    dyer    William     Dyer    619    dyer    rad
9.82
1    dyer    William     Dyer    619    dyer    sal
0.13
```

```
2   dyer    William     Dyer   622   dyer   rad
7.80
3   dyer    William     Dyer   622   dyer   sal
0.09
4     pb      Frank  Pabodie   734     pb   rad
8.41
..   ...        ...      ...   ...    ...   ...
...
14  lake   Anderson     Lake   837   lake   rad
1.46
15  lake   Anderson     Lake   837   lake   sal
0.21
16   roe  Valentina  Roerich   752    roe   sal
41.60
17   roe  Valentina  Roerich   837    roe   sal
22.50
18   roe  Valentina  Roerich   844    roe   rad
11.25

[19 rows x 7 columns]
```

```
print(vs)
```

```
  ident   site       dated  taken person quant
reading
0   619   DR-1  1927-02-08    619   dyer   rad
9.82
1   619   DR-1  1927-02-08    619   dyer   sal
0.13
2   622   DR-1  1927-02-10    622   dyer   rad
7.80
3   622   DR-1  1927-02-10    622   dyer   sal
0.09
```

```
4   734   DR-3  1939-01-07    734     pb   rad
8.41
..  ...    ...          ...    ...    ...   ...
...
16  752   DR-3         NaN    752    roe   sal
41.60
17  837  MSK-4  1932-01-14    837   lake   rad
1.46
18  837  MSK-4  1932-01-14    837   lake   sal
0.21
19  837  MSK-4  1932-01-14    837    roe   sal
22.50
20  844   DR-1  1932-03-22    844    roe   rad
11.25

[21 rows x 7 columns]
```

We know there is a many-to-many merge happening because there are duplicate values in the keys for **both** the left and right dataframe.

Click here to view code image

```
print(
  ps["quant"].value_counts()
)
```

```
rad     8
sal     8
temp    3
Name: quant, dtype: int64
```

```
print(
  vs["quant"].value_counts()
)
```

```
sal      9
rad      8
temp     4
Name: quant, dtype: int64
```

We can perform a many-to-many merge by passing the multiple columns to match on in a Python list.

```
ps_vs = ps.merge(
    vs,
    left_on=["quant"],
    right_on=["quant"],
)
```

Let’s look at just the first row of data.

Click here to view code image

```
print(ps_vs.loc[0, :])
```

```
ident_x              dyer
personal          William
family               Dyer
taken_x               619
person_x             dyer
                      ...
site                 DR-1
dated          1927-02-08
taken_y               619
person_y             dyer
reading_y            9.82
Name: 0, Length: 13, dtype: object
```

Pandas will automatically add a suffix to a column name if there are collisions in the name. In the output, the `_x` refers to values from the left dataframe, and the `_y` suffix comes from values in the right dataframe.

6.4.4 Check Your Work with Assert

A simple way to check your work before and after a merge is by looking at the number of rows of our data before and after the merge. If you end up with **more** rows than either of the dataframes you are merging together, that means a many-to-many merge occurred, and that is usually situation you do not want.

Click here to view code image

```
print(ps.shape) # left dataframe
```

```
(19, 7)
```

```
print(vs.shape) # right dataframe
```

```
(21, 7)
```

```
print(ps_vs.shape) # after merge
```

```
(148, 13)
```

One way you can check your work is by having your code fail when you know a bad condition exists. You can achieve this by using the Python `assert` statement. When an expression evaluates to `True`, `assert` will not return anything, and your code will continue on to the next expression.

```
# expect this to be true
# note there is no output
assert vs.shape[0] == 21
```

However, if the expression to `assert` evaluates to `False`, it will throw an `AssertionError`, and your code will stop.

Click here to view code image

```
assert ps_vs.shape[0] <= vs.shape[0]
```

```
AssertionError:
```

Using `assert` is a good technique to build in checks into your code without having to run it and visually inspecting the result. This is also the basis for creating "unit tests" for functions.

Conclusion

Sometimes, you may need to combine various parts or data or multiple data sets depending on the question you are trying to answer. Keep in mind, however, that the data you need for analysis does not necessarily equate to the best shape of data for storage.

The survey data used in the last example came in four separate parts that needed to be merged together. After we merged the tables, a lot of redundant information appeared across the rows. From a data storage and data entry point of view, each of these duplications can lead to errors and data inconsistency. This is what Hadley meant by saying that in tidy data, "each type of observational unit forms a table."

7

Data Normalization

The final point in the original "Tidy Data" paper stated that for data to be tidy "… each type of observational unit forms a table." However, usually we need to combine multiple data sets together so we can do an analysis (Chapter 6). But when we think about how to store and manage data in a way where we reduce the amount of duplication and potential for errors, we should try to normalize our data into separate tables so a single fix can propagate when we combine the data together again.

Learning Objectives

- Identify the differences between tidy data and data normalization
- Apply data subsetting to split data into normalized parts

7.1 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, and in other data sets that track a value over time.

Let's look again at the Billboard data we cleaned in Section 4.1.2.

Click here to view code image

```
import pandas as pd
```

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

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

print(billboard_long)
```

```
       year              artist
track  time  \
0      2000               2 Pac  Baby Don't Cry
(Keep...  4:22
1      2000             2Ge+her  The Hardest Part Of
...  3:15
2      2000        3 Doors Down
Kryptonite  3:53
3      2000        3 Doors Down
Loser  4:24
4      2000            504 Boyz              Wobble
Wobble  3:35
...     ...                 ...
...  ...
24087  2000         Yankee Grey     Another Nine
Minutes  3:10
24088  2000   Yearwood, Trisha          Real Live
Woman  3:55
24089  2000    Ying Yang Twins  Whistle While You
Tw...  4:19
24090  2000      Zombie Nation              Kernkraft
```

```
400  3:30
24091  2000   matchbox twenty
Bent  4:12

      date.entered  week  rating
0       2000-02-26   wk1    87.0
1       2000-09-02   wk1    91.0
2       2000-04-08   wk1    81.0
3       2000-10-21   wk1    76.0
4       2000-04-15   wk1    57.0
...            ...   ...     ...
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

[24092 rows x 7 columns]
```

Suppose we subset the data based on a particular track:

Click here to view code image

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

```
       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
...      ...              ...   ...      ...
...    ...      ...
22510   2000   3 Doors Down   Loser   4:24   2000-10-
21    wk72     NaN
22827   2000   3 Doors Down   Loser   4:24   2000-10-
21    wk73     NaN
23144   2000   3 Doors Down   Loser   4:24   2000-10-
21    wk74     NaN
23461   2000   3 Doors Down   Loser   4:24   2000-10-
21    wk75     NaN
23778   2000   3 Doors Down   Loser   4:24   2000-10-
21    wk76     NaN

[76 rows x 7 columns]
```

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.

We can place the `year`, `artist`, `track`, and `time` 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 date entered, 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 6.

Click here to view code image

```
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.

Click here to view code image

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

```
(317, 4)
```

We can then assign a unique value to each row of data. There are many ways you could do this, there we take the index value and add `1` so it doesn't start with `0`.

Click here to view code image

```
billboard_songs['id'] = billboard_songs.index +
1
print(billboard_songs)
```

```
     year                artist
track  time    id
0     2000                2 Pac   Baby Don't Cry
(Keep...   4:22     1
1     2000              2Ge+her   The Hardest Part Of
...   3:15     2
2     2000             3 Doors            Down
```

```
Kryptonite  3:53    3
3    2000            3 Doors              Down
Loser  4:24    4
4    2000           504 Boyz            Wobble
Wobble  3:35    5
..    ...               ...
...   ...  ...
312  2000       Yankee Grey     Another Nine
Minutes  3:10  313
313  2000  Yearwood, Trisha          Real Live
Woman  3:55  314
314  2000   Ying Yang Twins  Whistle While You
Tw...  4:19  315
315  2000     Zombie Nation            Kernkraft
400  3:30  316
316  2000   matchbox twenty
Bent  4:12  317

[317 rows x 5 columns]
```

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.

Click here to view code image

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

```
(24092, 8)
```

```
print(billboard_ratings)
```

```
       year           artist
track  time  \
0      2000            2 Pac  Baby Don't Cry
(Keep...  4:22
1      2000            2 Pac  Baby Don't Cry
(Keep...  4:22
2      2000            2 Pac  Baby Don't Cry
(Keep...  4:22
3      2000            2 Pac  Baby Don't Cry
(Keep...  4:22
4      2000            2 Pac  Baby Don't Cry
(Keep...  4:22
...     ...              ...
...   ...
24087  2000  matchbox twenty
Bent  4:12
24088  2000  matchbox twenty
Bent  4:12
24089  2000  matchbox twenty
Bent  4:12
24090  2000  matchbox twenty
Bent  4:12
24091  2000  matchbox twenty
Bent  4:12

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

```
4        2000-02-26   wk5  87.0     1
...            ...    ...   ...   ...
24087  2000-04-29  wk72   NaN   317
24088  2000-04-29  wk73   NaN   317
24089  2000-04-29  wk74   NaN   317
24090  2000-04-29  wk75   NaN   317
24091  2000-04-29  wk76   NaN   317

[24092 rows x 8 columns]
```

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

Click here to view code image

```
billboard_ratings = billboard_ratings[
      ["id", "date.entered", "week", "rating"]
]
print(billboard_ratings)
```

```
        id date.entered  week  rating
0        1   2000-02-26   wk1    87.0
1        1   2000-02-26   wk2    82.0
2        1   2000-02-26   wk3    72.0
3        1   2000-02-26   wk4    77.0
4        1   2000-02-26   wk5    87.0
...    ...          ...   ...     ...
24087  317   2000-04-29  wk72     NaN
24088  317   2000-04-29  wk73     NaN
24089  317   2000-04-29  wk74     NaN
24090  317   2000-04-29  wk75     NaN
24091  317   2000-04-29  wk76     NaN

[24092 rows x 4 columns]
```

Conclusion

This chapter explored how we can reduce the amount of duplicate information in data for efficient data storage. Data normalization can be thought of as the opposite process of preparing data for analysis, visualization, and model fitting. But typically you will need to combine multiple normalized data sets together into a tidy data set.

8

Groupby Operations: Split-Apply-Combine

Grouped operations are a powerful way to aggregate, transform, and filter data. They rely on the mantra of "split–apply–combine":

1. Data is split into separate parts based on key(s).
2. A function is applied to each part of the data.
3. The results from each part are combined to create a new data set.

This is a powerful concept because parts of your original data can be split up into independent parts to perform a calculation. If you worked with databases in the past, then you should recognize that the Pandas `.groupby()` works just like the SQL `GROUP BY`. The split–apply–combine concept is also heavily used in "big data" systems that use distributed computing, with the data being split into independent parts and dispatched to a separate server where a function is applied, and the results are then combined.

The techniques shown in this chapter can all be done without using the `.groupby()` method. For example:

- Aggregation can be done by using conditional subsetting on a dataframe
- Transformation can be done by passing a column into a separate function
- Filtering can be done with conditional subsetting

However, when you work with your data using `.groupby()` statements, your code can be faster, you have greater flexibility when you want to create multiple groups, and you can more readily work with larger data sets on distributed or parallel systems.

Learning Objectives

- Understand what grouped data is
- Calculate summaries of data using `.groupby()` operations
- Perform aggregation, transformation, and filtering operations on grouped data
- Separate data by groups for separate calculations

8.1 Aggregate

Aggregation is the process of taking multiple values and returning a **single value**. Calculating an arithmetic mean is an example, as multiple values are averaged to produce a single value.

8.1.1 Basic One-Variable Grouped Aggregation

Section 1.4.1 showed how to calculate grouped means using the `gapminder` data set. We calculated the average life expectancy for each year of the data and plotted it. This is an example of using group-by operations for data aggregation; that is, we used the `.groupby()` method to calculate a summary statistic, the mean, for all the values in each year.

Aggregation may sometimes be referred to as summarization. Both terms mean that some form of data reduction is involved. For example, when you calculate a summary statistic, such as the mean, you are taking multiple values and replacing them with a single value. The amount of data is now smaller.

Click here to view code image

```
import pandas as pd
df = pd.read_csv('data/gapminder.tsv', sep='\t')
```

```
# calculate the average life expectancy for each
year
avg_life_exp_by_year = df.groupby('year')
["lifeExp"].mean()

print(avg_life_exp_by_year)
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
          ...
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, Length: 12, dtype: float64
```

Groupby statements can be thought of as creating a subset of each unique value of a column (or unique pairs from columns). For example, we could get a list of unique values in the column.

Click here to view code image

```
# get a list of unique years in the data
years = df.year.unique()
print(years)
```

```
[1952 1957 1962 1967 1972 1977 1982 1987 1992 1997
2002 2007]
```

We can go through each of the years and subset the data.

Click here to view code image

```
# subset the data for the year 1952
y1952 = df.loc[df.year == 1952, :]
print(y1952)
```

```
                    country continent  year  lifeExp
pop  \
0               Afghanistan      Asia  1952   28.801
8425333
12                  Albania    Europe  1952   55.230
1282697
24                  Algeria    Africa  1952   43.077
9279525
36                   Angola    Africa  1952   30.015
4232095
48                Argentina  Americas  1952   62.485
17876956
...                     ...       ...   ...      ...
...
1644                Vietnam      Asia  1952   40.412
26246839
1656     West Bank and Gaza      Asia  1952   43.160
1030585
1668            Yemen, Rep.      Asia  1952   32.548
4963829
1680                 Zambia    Africa  1952   42.038
2672000
1692               Zimbabwe    Africa  1952   48.451
3080907
```

```
        gdpPercap
0      779.445314
12    1601.056136
24    2449.008185
36    3520.610273
48    5911.315053
...           ...
1644   605.066492
1656  1515.592329
1668   781.717576
1680  1147.388831
1692   406.884115

[142 rows x 6 columns]
```

Finally, we can perform a function on the subset data. Here we take the mean of the `lifeExp` values.

Click here to view code image

```
y1952_mean = y1952["lifeExp"].mean()
print(y1952_mean)
```

```
49.057619718309866
```

The `.groupby()` method essentially repeats this process for every year column (i.e., splits the data), calculates the mean value (i.e., applies a function), and conveniently returns all the results in a single dataframe (i.e., combines the values together).

Of course, mean is not the only type of aggregation function you can use. There are many built-in methods in Pandas you can use with the `.groupby()` method.

8.1.2 Built-In Aggregation Methods

Table 8.1 provides a non-exclusive list of built-in Pandas methods you can use to aggregate your data.

Table 8.1 Methods and Functions That Can Be Used With `.groupby()`

Pandas Method	Numpy/Scipy Function	Description
`.count()`	`np.count_nonzero()`	Frequency count **not** including NaN values
`.size()`		Frequency count **with** NaN values
`.mean()`	`np.mean()`	Mean of the values
`.std()`	`np.std()`	Sample standard deviation
`.min()`	`np.min()`	Minimum values
`.quantile(q=0.25)`	`np.percentile(q=0.25)`	25th percentile of the values
`.quantile(q=0.50)`	`np.percentile(q=0.50)`	50th percentile of the values
`.quantile(q=0.75)`	`np.percentile(q=0.75)`	75th percentile of the values
`.max()`	`np.max()`	Maximum value
`.sum()`	`np.sum()`	Sum of the values
`.var()`	`np.var()`	Unbiased variance
`.sem()`	`scipy.stats.sem()`	Unbiased standard error of the mean
`.describe()`	`scipy.stats.describe()`	Count, mean, standard deviation, minimum, 25%, 50%, 75%, and maximum
`.first()`		Returns the first row

Pandas Method	Numpy/Scipy Function	Description
`.last()`		Returns the last row
`.nth()`		Returns the *n*th row (Python starts counting from 0)

For example, we can calculate multiple summary statistics simultaneously with `.describe()`.

Click here to view code image

```
# group by continent and describe each group
continent_describe = df.groupby('continent')
["lifeExp"].describe()
print(continent_describe)
```

```
           count       mean        std     min
25%       50%  \
continent
Africa     624.0  48.865330   9.150210  23.599
42.37250  47.7920
Americas   300.0  64.658737   9.345088  37.579
58.41000  67.0480
Asia       396.0  60.064903  11.864532  28.801
51.42625  61.7915
Europe     360.0  71.903686   5.433178  43.585
69.57000  72.2410
Oceania     24.0  74.326208   3.795611  69.120
71.20500  73.6650

                75%     max
continent
```

```
Africa       54.41150  76.442
Americas     71.69950  80.653
Asia         69.50525  82.603
Europe       75.45050  81.757
Oceania      77.55250  81.235
```

8.1.3 Aggregation Functions

You can also use an aggregation function that is not listed in the "Pandas Method" column in Table 8.1. Instead of directly calling the aggregation method, you can call the `.agg()` or `.aggregate()` method, and pass the aggregation function you want in there. When using `.agg()` or `.aggregate()`, you will use the functions listed in the "Numpy/Scipy Function" column in Table 8.1.

> **Note**
>
> The `.agg()` method is an alias for `.aggregate()`. The Pandas documentation suggests you use the alias, `.agg()`, over the fully spelled out method.

8.1.3.1 Functions From Other Libraries

We can use the `mean()` function from the `numpy` library by passing the function into the `.agg()` method.

Click here to view code image

```
import numpy as np

# calculate the average life expectancy by
continent
# but use the np.mean function
cont_le_agg = df.groupby('continent')
```

```
["lifeExp"].agg(np.mean)

print(cont_le_agg)
```

```
continent
Africa       48.865330
Americas     64.658737
Asia         60.064903
Europe       71.903686
Oceania      74.326208
Name: lifeExp, dtype: float64
```

Note

When we pass in the function into `.agg()`, we only need the actual function object, we do not need to "call" the function. That's why we write `np.mean` and not `np.mean()`. This is similar to when we called `.apply()` in Chapter 5.

8.1.3.2 Custom User Functions

Sometimes we may want to perform a calculation that is not provided by Pandas or another library. We can write our own function that performs the calculation we want and use it in `.agg()` as well.

Let's create our own mean function. Recall the mean function:

$$\text{mean} = x = \frac{1}{n}\sum_{i=1}^{n} x_i \quad (8.1)$$

Click here to view code image

```
def my_mean(values):
  """My version of calculating a mean"""
  # get the total number of numbers for the
```

```
denominator
  n = len(values)

  # start the sum at 0
  sum = 0
  for value in values:
      # add each value to the running sum
      sum += value

  # return the summed values divided by the
number of values
  return sum / n
```

Note that the function we wrote takes only one parameter, `values`. What gets passed into the function, however, is the entire series of values. This is why we need to iterate through the values to take the sum.

Also, we could have calculated the sum in the function by using `values.sum()`, which can actually handle missing values better than the way the `for` loop is currently written. See Chapter 5 for a review of these concepts.

We can pass our custom function straight into the `.agg()` or `.aggregate()` method with `my_mean`.

Click here to view code image

```
# use our custom function into agg
agg_my_mean = df.groupby('year')
["lifeExp"].agg(my_mean)

print(agg_my_mean)
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
```

```
1967    55.678290
1972    57.647386
           ...
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, Length: 12, dtype: float64
```

Finally, we can write functions that take multiple parameters. As long as the first parameter takes the series of values from the dataframe, you can pass the other arguments as keywords into `.agg()` or `.aggregate()`.

In the following example, we will calculate the global average life expectancy, `diff_value`, and subtract it from each grouped value.

Click here to view code image

```
def my_mean_diff(values, diff_value):
    """Difference between the mean and
diff_value
    """
    n = len(values)
    sum = 0
    for value in values:
        sum += value
    mean = sum / n
    return(mean - diff_value)

# calculate the global average life expectancy
mean
global_mean = df["lifeExp"].mean()
print(global_mean)
```

```
59.474439366197174
```

```
# custom aggregation function with multiple
parameters
agg_mean_diff = (
  df
  .groupby("year")
  ["lifeExp"]
  .agg(my_mean_diff, diff_value=global_mean)
)

print(agg_mean_diff)
```

```
year
1952   -10.416820
1957    -7.967038
1962    -5.865190
1967    -3.796150
1972    -1.827053
          ...
1987     3.738173
1992     4.685899
1997     5.540237
2002     6.220483
2007     7.532983
Name: lifeExp, Length: 12, dtype: float64
```

8.1.4 Multiple Functions Simultaneously

When we want to calculate multiple aggregation functions, we can pass the individual functions into `.agg()` or `.aggregate()` as a Python `list`. Examples of functions you can use here are listed in the "Numpy/Scipy Function" column in Table 8.1.

Click here to view code image

```
# calculate the count, mean, std of the lifeExp
by continent
gdf = (
  df
  .groupby("year")
  ["lifeExp"]
  .agg([np.count_nonzero, np.mean, np.std])
)

print(gdf)
```

```
      count_nonzero       mean        std
year
1952            142  49.057620  12.225956
1957            142  51.507401  12.231286
1962            142  53.609249  12.097245
1967            142  55.678290  11.718858
1972            142  57.647386  11.381953
...             ...        ...        ...
1987            142  63.212613  10.556285
1992            142  64.160338  11.227380
1997            142  65.014676  11.559439
2002            142  65.694923  12.279823
2007            142  67.007423  12.073021

[12 rows x 3 columns]
```

8.1.5 Use a dict in `.agg()` / `.aggregate()`

There are some other ways you can apply functions in the `.agg()` and `.aggregate()` methods. For example, you can pass `.agg()` a Python dictionary. However, the results will differ depending on whether you are aggregating directly on a `DataFrame` or on a `Series` object.

8.1.5.1 On a DataFrame

When specifying a `dict` on a grouped `DataFrame`, the keys are the columns of the `DataFrame`, and the values are the functions used in the aggregated calculation. This approach allows you to group one or more variables and use a different aggregation function on different columns simultaneously.

Click here to view code image

```
# use a dictionary on a dataframe to agg
different columns
# for each year, calculate the
# average lifeExp, median pop, and median
gdpPercap
gdf_dict = df.groupby("year").agg(

  {
    "lifeExp": "mean",
    "pop": "median",
    "gdpPercap": "median"
  }
)

print(gdf_dict)
```

```
        lifeExp         pop    gdpPercap
year
1952  49.057620   3943953.0  1968.528344
1957  51.507401   4282942.0  2173.220291
1962  53.609249   4686039.5  2335.439533
1967  55.678290   5170175.5  2678.334740
```

```
1972  57.647386   5877996.5  3339.129407
...         ...         ...          ...
1987  63.212613   7774861.5  4280.300366
1992  64.160338   8688686.5  4386.085502
1997  65.014676   9735063.5  4781.825478
2002  65.694923  10372918.5  5319.804524
2007  67.007423  10517531.0  6124.371108

[12 rows x 3 columns]
```

8.1.5.2 On a Series

In the past, passing a `dict` into a `Series` after a `.groupby()` allowed you to directly calculate aggregate statistics as the returned value, with the key of the `dict` being the new column name. However, this notation is not consistent with the behavior when `dict`s are passed into grouped `DataFrames`, as shown in the example in Section 8.1.5.1. To have user-defined column names in the output of a grouped series calculation, you need to rename those columns after the fact.

Click here to view code image

```
gdf = (
  df
  .groupby("year")
  ["lifeExp"]
  .agg(
    [
      np.count_nonzero,
      np.mean,
      np.std,
    ]
  )
  .rename(
```

```
    columns={
      "count_nonzero": "count",
      "mean": "avg",
      "std": "std_dev",
    }
  )
  .reset_index() # return a flat dataframe
)

print(gdf)
```

```
    year  count        avg    std_dev
0   1952    142  49.057620  12.225956
1   1957    142  51.507401  12.231286
2   1962    142  53.609249  12.097245
3   1967    142  55.678290  11.718858
4   1972    142  57.647386  11.381953
..   ...    ...        ...        ...
7   1987    142  63.212613  10.556285
8   1992    142  64.160338  11.227380
9   1997    142  65.014676  11.559439
10  2002    142  65.694923  12.279823
11  2007    142  67.007423  12.073021

[12 rows x 4 columns]
```

8.2 Transform

When we transform data, we pass values from our dataframe into a function. The function then "transforms" the data. Unlike `.agg()`, which can take multiple values and return a single (aggregated) value, `.transform()` takes multiple values and returns a one-to-one transformation of the values. That is, it does not reduce the amount of data.

8.2.1 Z-Score Example

Let's calculate the z-score of our life expectancy data by year. The z-score identifies the number of standard deviations from the mean of our data. It centers our data around 0, with a standard deviation of 1. This technique standardizes our data and makes it easier to compare different variables with different units to each other.

Here's the formula for calculating z-score:

$$z = \frac{x-\mu}{\sigma} \quad (8.2)$$

- x is a data point in our data set
- μ is the average of our data set, as calculated by Equation 8.1
- σ is the standard deviation, as calculated by Equation 8.3

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i-\mu)^2} \quad (8.3)$$

Let's write a Python function that calculates a z-score.

Click here to view code image

```
def my_zscore(x):
  '''Calculates the z-score of provided data
  'x' is a vector or series of values
  '''
  return((x - x.mean()) / x.std())
```

Now we can use this function to `.transform()` our data by group.

Click here to view code image

```
transform_z = df.groupby('year')
["lifeExp"].transform(my_zscore)

print(transform_z)
```

```
0       -1.656854
1       -1.731249
2       -1.786543
3       -1.848157
4       -1.894173
           ...
1699    -0.081621
1700    -0.336974
1701    -1.574962
1702    -2.093346
1703    -1.948180
Name: lifeExp, Length: 1704, dtype: float64
```

Note the shape of our original dataframe, and that of the `transform_z` value. Both have the same number of rows and data.

Click here to view code image

```
# note the number of rows in our data
print(df.shape)
```

```
(1704, 6)
```

```
# note the number of values in our
transformation
print(transform_z.shape)
```

```
(1704,)
```

The `scipy` library has its own `zscore()` function. Let's use its `zscore()` function in a `.groupby()` `.transform()` and compare it to what happens when we do not use `.groupby()`.

Click here to view code image

```
from scipy.stats import zscore

# calculate a grouped zscore

sp_z_grouped = df.groupby('year')
["lifeExp"].transform(zscore)

# calculate a nongrouped zscore
sp_z_nogroup = zscore(df["lifeExp"])
```

Notice that not all of the `zscore()` values are the same.

```
# grouped z-score
print(transform_z.head())
```

```
0   -1.656854
1   -1.731249
2   -1.786543
3   -1.848157
4   -1.894173
Name: lifeExp, dtype: float64
```

```
# grouped z-score using scipy
print(sp_z_grouped.head())
```

```
0   -1.662719
1   -1.737377
2   -1.792867
```

```
3    -1.854699
4    -1.900878
Name: lifeExp, dtype: float64
```

```
# nongrouped z-score
print(sp_z_nogroup[:5])
```

```
0    -2.375334
1    -2.256774
2    -2.127837
3    -1.971178
4    -1.811033
Name: lifeExp, dtype: float64
```

Our grouped results are similar. However, when we calculate the *z*-score outside the `.groupby()`, we get the *z*-score calculated on the entire data set, not broken out by group.

8.2.2 Missing Value Example

Chapter 9 covers missing values and explored how we can fill in missing values. In the Ebola data set example in that chapter, it made more sense to fill in the missing data using the `.interpolate()` method, or forward/backward filling our data.

In certain data sets, filling the missing values with the mean of the column could also make sense. At other times, however, it may make more sense to fill in missing data based on a particular group. Let's work with the `tips` data set that comes from the `seaborn` library.

Click here to view code image

```
import seaborn as sns
import numpy as np

# set the seed so results are deterministic
```

```python
np.random.seed(42)

# sample 10 rows from tips
tips_10 = sns.load_dataset("tips").sample(10)

# randomly pick 4 'total_bill' values and turn
them into missing
tips_10.loc[
    np.random.permutation(tips_10.index)[:4],
    "total_bill"
] = np.NaN

print(tips_10)
```

```
    total_bill   tip     sex smoker   day    time
size
24       19.82  3.18    Male     No   Sat  Dinner
2
6         8.77  2.00    Male     No   Sun  Dinner
2
153        NaN  2.00    Male     No   Sun  Dinner
4
211        NaN  5.16    Male    Yes   Sat  Dinner
4
198        NaN  2.00  Female    Yes  Thur   Lunch
2
176        NaN  2.00    Male    Yes   Sun  Dinner
2
192      28.44  2.56    Male    Yes  Thur   Lunch
2
124      12.48  2.52  Female     No  Thur   Lunch
2
9        14.78  3.23    Male     No   Sun  Dinner
```

```
2
101       15.38  3.00  Female    Yes    Fri   Dinner
2
```

Chapter 9 also shows how you can use the `.fillna()` method to fill in the missing values. However, we may not want to simply fill the missing values with the `mean` of `total_bill`. Perhaps the `Male` and `Female` values in the `sex` column have different spending habits, or perhaps the `total_bill` values differ between time of day (`time`), or and `size` of the table. These are all valid concerns when processing our data.

We can use the `.groupby()` method to calculate a statistic to fill in missing values. Instead of using `.agg()`, we use the `.transform()` method. First, let's count the non-missing values by `sex`.

Click here to view code image

```
count_sex = tips_10.groupby('sex').count()
print(count_sex)
```

```
        total_bill tip smoker day time size
sex
Male                 4   7       7   7    7    7
Female               2   3       3   3    3    3
```

This result gives us the number of non-missing values for each value of `sex` in each column. We have three missing values for `Male`, and one missing value for `Female`. Now let's calculate a grouped average, and use the grouped average to fill in the missing values.

Click here to view code image

```
def fill_na_mean(x):
  """Returns the average of a given vector"""
  avg = x.mean()
  return x.fillna(avg)
```

```
# calculate a mean 'total_bill' by 'sex'
total_bill_group_mean = (
  tips_10
  .groupby("sex")
  .total_bill
  .transform(fill_na_mean)
)

# assign to a new column in the original data
# you can also replace the original column by
using 'total_bill'
tips_10["fill_total_bill"] =
total_bill_group_mean
```

If we just look at the two `total_bill` columns, we see that different values were filled in for the `NaN` missing values.

Click here to view code image

```
print(tips_10[['sex', 'total_bill',
'fill_total_bill']])
```

```
        sex  total_bill  fill_total_bill
24     Male       19.82          19.8200
6      Male        8.77           8.7700
153    Male         NaN          17.9525
211    Male         NaN          17.9525
198  Female         NaN          13.9300
176    Male         NaN          17.9525
192    Male       28.44          28.4400
124  Female       12.48          12.4800
```

```
9       Male         14.78              14.7800
101   Female         15.38              15.3800
```

8.3 Filter

The last type of action you can perform with the `.groupby()` method is `.filter()`. This allows you to split your data by keys, and then perform some kind of boolean subsetting on the data. As with all the examples for `.groupby()`, you can accomplish the same thing by using regular subsetting, as described in Section 1.3 and Section 2.4.1. Let's use the full tips data set and look at the number of observations for the various `size` values.

Click here to view code image

```
# load the tips data set
tips = sns.load_dataset('tips')

# note the number of rows in the original data
print(tips.shape)
```

```
(244, 7)
```

```
# look at the frequency counts for the table
size
print(tips['size'].value_counts())
```

```
2    156
3     38
4     37
5      5
1      4
6      4
Name: size, dtype: int64
```

The output shows that table sizes of `1`, `5`, and `6` are infrequent. Depending on your needs, you may want to filter those data points out. In this example, we want each group to consist of 30 or more observations.

To accomplish this goal, we can use the `.filter()` method on a grouped operation.

Click here to view code image

```
# filter the data such that each group has more
than 30 observations
tips_filtered = (
  tips
  .groupby("size")
  .filter(lambda x: x["size"].count() >= 30)
)
```

The output shows that our data set was filtered down.

Click here to view code image

```
print(tips_filtered.shape)
```

```
(231, 7)
```

```
print(tips_filtered['size'].value_counts())
```

```
2    156
3     38
4     37
Name: size, dtype: int64
```

8.4 The `pandas.core.groupby. DataFrameGroupBy` object

The `.aggregate()`, `.transform()`, and `.filter()` methods are commonly used ways of working with grouped objects in Pandas. In this section, we will investigate some of the inner workings of grouped objects. The `.groupby()` documentation is an excellent resource for some of the more nuanced features of `.groupby()`.[1]

1. `groupby()` documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html

8.4.1 Groups

Throughout this chapter, we've directly chained `.agg()`, `.transform()`, or `.filter()` after the `.groupby()`. However, we can actually save the results of `.groupby()` before we perform those other methods. We will start with the subsetted `tips` data set.

Click here to view code image

```
tips_10 = sns.load_dataset('tips').sample(10,
random_state=42)
print(tips_10)
```

```
    total_bill   tip     sex  smoker   day    time
size
24       19.82  3.18    Male      No   Sat  Dinner
2
6         8.77  2.00    Male      No   Sun  Dinner
2
153      24.55  2.00    Male      No   Sun  Dinner
4
211      25.89  5.16    Male     Yes   Sat  Dinner
4
198      13.00  2.00  Female     Yes  Thur   Lunch
2
```

```
176        17.89  2.00    Male      Yes    Sun   Dinner
2
192        28.44  2.56    Male      Yes   Thur    Lunch
2
124        12.48  2.52  Female       No   Thur    Lunch
2
9          14.78  3.23    Male       No    Sun   Dinner
2
101        15.38  3.00  Female      Yes    Fri   Dinner
2
```

We can choose to save just the `groupby` object without running any other `.agg()`, `.transform()`, or `.filter()` method on it.

Click here to view code image

```
# save just the grouped object
grouped = tips_10.groupby('sex')

# note that we just get back the object and its
memory location
print(grouped)
```

```
<pandas.core.groupby.generic.DataFrameGroupBy
object at 0x15ed37880>
```

When we try to print out the `grouped` result, we get a memory reference back and the data type is a Pandas `DataFrameGroupBy` object. Under the hood, nothing has been actually calculated yet, because we never performed an action that requires a calculation. If we want to actually see the calculated groups, we can call the `groups` attribute.

Click here to view code image

```
# see the actual groups of the groupby
# it returns only the index
print(grouped.groups)
```

```
{'Male': [24, 6, 153, 211, 176, 192, 9], 'Female':
[198, 124, 101]}
```

Even when we ask for the `groups` from our `grouped` object, we get only the `index` of the dataframe back. Think of this index as indicating the row numbers. It is intended mainly to optimize performance. Again, we haven't calculated anything yet.

This approach does allow you to save just the grouped result. You could then perform multiple `.agg()`, `.transform()`, or `.filter()` operations without having to process the `.groupby()` statement again.

8.4.2 Group Calculations Involving Multiple Variables

One of the nice things about Python is that it follows the EAFP mantra: It is "easier to ask for forgiveness than for permission." Throughout the chapter, we have been performing `.groupby()` calculations on a single column. If we specify the calculation we want right after the `.groupby()`, however, Python will perform the calculation on all the columns it can and silently drop the rest.

Here's an example of a grouped mean on all the columns by `sex`.

Click here to view code image

```
# calculate the mean on relevant columns
avgs = grouped.mean()
print(avgs)
```

```
        total_bill       tip      size
sex
```

```
Male           20.02  2.875714  2.571429
Female         13.62  2.506667  2.000000
```

As you can see, not all the columns reported a mean.

Click here to view code image

```
# list all the columns
print(tips_10.columns)
```

```
Index(['total_bill', 'tip', 'sex', 'smoker',
'day', 'time', 'size'],
 dtype='object')
```

The `smoker`, `day`, and `time` columns were not returned in the results those columns do not contain numeric values, rather, they contain categorical values. To use the `time` column as an example, there is no arithmetic mean for the terms `Dinner` and `Lunch`.

8.4.3 Selecting a Group

If we want to extract a particular group, we can use the `.get_group()` method, and pass in the group that we want. For example, if we wanted the `Female` values:

Click here to view code image

```
# get the 'Female' group
female = grouped.get_group('Female')
print(female)
```

```
     total_bill   tip     sex  smoker   day    time
size
198       13.00  2.00  Female     Yes  Thur   Lunch
2
```

```
124         12.48  2.52  Female     No  Thur   Lunch
2
101         15.38  3.00  Female    Yes   Fri  Dinner
2
```

8.4.4 Iterating Through Groups

Another benefit of saving just the `groupby` object is that you can then iterate through the groups individually. There might be times when it's easier to conceptualize a question using a `for` loop, rather than trying to formulate an `.agg()`, `.transform()`, or `.filter()` method. Sometimes this might be the only way to do the task. Other times, it might be the way to get the task done for now, and you can work on optimizing the solution later.

We can iterate through our `grouped` values just like any other container in Python using a `for` loop.

Click here to view code image

```
for sex_group in grouped:
    print(sex_group)
```

```
('Male',      total_bill   tip  sex  smoker  day
time  size
24          19.82  3.18  Male     No    Sat
Dinner     2
6            8.77  2.00  Male     No    Sun
Dinner     2
153         24.55  2.00  Male     No    Sun
Dinner     4
211         25.89  5.16  Male     Yes   Sat
Dinner     4
176         17.89  2.00  Male     Yes   Sun
Dinner     2
```

```
192        28.44  2.56  Male     Yes  Thur    Lunch
2
9          14.78  3.23  Male     No   Sun
Dinner     2)
('Female',      total_bill   tip      sex smoker
day   time  size
198        13.00  2.00  Female   Yes  Thur    Lunch
2
124        12.48  2.52  Female   No   Thur    Lunch
2
101        15.38  3.00  Female   Yes  Fri
Dinner     2)
```

If you try to get just the first index from the `grouped` object, you will get an error message. This object is still a `pandas.core.groupby.DataFrameGroupBy` object, rather than a real Pandas container.

Click here to view code image

```
# you can't really get the 0 element from the
grouped object
print(grouped[0])
```

```
KeyError: 'Column not found: 0'
```

For now, let's modify the `for` loop to just show the first element, along with some of the things we get when we loop over the `grouped` object.

Click here to view code image

```
for sex_group in grouped:
    # get the type of the object (tuple)
    print(f'the type is: {type(sex_group)}\n')
```

```
    # get the length of the object (2 elements)
    print(f'the length is: {len(sex_group)}\n')

    # get the first element
    first_element = sex_group[0]
    print(f'the first element is:
{first_element}\n')

    # the type of the first element (string)
    print(f'it has a type of:
{type(sex_group[0])}\n')

    # get the second element
    second_element = sex_group[1]
    print(f'the second element
is:\n{second_element}\n')

    # get the type of the second element
(dataframe)
    print(f'it has a type of:
{type(second_element)}\n')

    # print what we have
    print(f'what we have:')
    print(sex_group)

    # stop after first iteration
    break
```

```
the type is: <class 'tuple'>

the length is: 2
```

```
the first element is: Male

it has a type of: <class 'str'>

the second element is:
     total_bill   tip   sex smoker   day    time
size
24        19.82  3.18  Male     No   Sat  Dinner
2
6          8.77  2.00  Male     No   Sun  Dinner
2
153       24.55  2.00  Male     No   Sun  Dinner
4
211       25.89  5.16  Male    Yes   Sat  Dinner
4
176       17.89  2.00  Male    Yes   Sun  Dinner
2
192       28.44  2.56  Male    Yes  Thur  Lunch
2
9         14.78  3.23  Male     No   Sun  Dinner
2

it has a type of: <class
'pandas.core.frame.DataFrame'>


what we have:
('Male',      total_bill   tip  sex  smoker  day
time  size
24        19.82  3.18  Male     No   Sat
Dinner  2
6          8.77  2.00  Male     No   Sun
Dinner  2
```

```
153         24.55  2.00  Male     No   Sun
Dinner  4
211         25.89  5.16  Male     Yes  Sat
Dinner  4
176         17.89  2.00  Male     Yes  Sun
Dinner  2
192         28.44  2.56  Male     Yes  Thur    Lunch
2
9           14.78  3.23  Male     No   Sun
Dinner  2)
```

We have a two-element `tuple` in which the first element is a `str` (string) that represents the `Male` key, and the second element is a `DataFrame` of the `Male` data.

If you prefer, you can forgo all the techniques introduced in this chapter and iterate through your grouped values in this manner to perform your calculations. Again, there may be times when this is the only way to get something done. Perhaps you have a complicated condition you want to check for each group, or you want to write out each group into separate files. This option is available to you if you need to iterate through the groups one at a time.

8.4.5 Multiple Groups

So far in this chapter, we have included one variable in the `.groupby()` method. In fact, we can add multiple variables during the `.groupby()` process. Section 1.4.1 briefly showed such a case.

Let's say we want to calculate the `mean` of our `tips` data by `sex`, time of day (`time`), and day of week (`day`). We can pass in `['sex', 'time']` as a Python `list` instead of the single string we have been using.

Click here to view code image

```
# mean by sex and time
bill_sex_time = tips_10.groupby(['sex', 'time'])

group_avg = bill_sex_time.mean()
```

8.4.6 Flattening the Results (`.reset_index()`)

The final topic that will be covered in this section is the results from the `.groupby()` statement. Let's look at the type of the `group_avg` we just calculated.

Click here to view code image

```
# type of the group_avg
print(type(group_avg))
```

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

We have a `DataFrame`, but the results look a little strange: We have what appear to be empty cells in the dataframe.

If we look at the `columns`, we get what we expect.

Click here to view code image

```
print(group_avg.columns)
```

```
Index(['total_bill', 'tip', 'size'],
dtype='object')
```

However, more interesting things happen when we look at the `index`.

Click here to view code image

```
print(group_avg.index)
```

```
MultiIndex([(  'Male',  'Lunch'),
            (  'Male', 'Dinner'),
            ('Female',  'Lunch'),
            ('Female', 'Dinner')],
           names=['sex', 'time'])
```

If we like, we can use a `MultiIndex`. If we want to get a regular flat dataframe back, we can call the `.reset_index()` method on the results.

Click here to view code image

```
group_method = tips_10.groupby(['sex',
'time']).mean().reset_index()
print(group_method)
```

```
      sex    time  total_bill       tip      size
0    Male   Lunch   28.440000  2.560000  2.000000
1    Male  Dinner   18.616667  2.928333  2.666667
2  Female   Lunch   12.740000  2.260000  2.000000
3  Female  Dinner   15.380000  3.000000  2.000000
```

Alternatively, we can use the `as_index=False` parameter in the `.groupby()` method (it is `True` by default).

Click here to view code image

```
group_param = tips_10.groupby(['sex', 'time'],
as_index=False).mean()
print(group_param)
```

```
      sex    time  total_bill       tip      size
0    Male   Lunch   28.440000  2.560000  2.000000
1    Male  Dinner   18.616667  2.928333  2.666667
```

```
2   Female    Lunch   12.740000  2.260000  2.000000
3   Female   Dinner   15.380000  3.000000  2.000000
```

8.5 Working With a MultiIndex

Sometimes, you may want to chain calculations after a `.groupby()` method. You can always “flatten” the results and then execute another `.groupby()` statement, but that may not always be the most efficient way of performing the calculation.

We begin with epidemiological simulation data on influenza cases in Chicago (this is a fairly large data set).

Click here to view code image

```
# notice that we can even read a compressed zip
file of a csv
intv_df = pd.read_csv('data/epi_sim.zip')

print(intv_df)
```

```
         ig_type  intervened        pid  rep  sid
tr
0              3          40  294524448    1  201
0.000135
1              3          40  294571037    1  201
0.000135
2              3          40  290699504    1  201
0.000135
3              3          40  288354895    1  201
0.000135
4              3          40  292271290    1  201
0.000135
...          ...         ...        ...  ...  ...
...
```

```
9434648         2           87  345636694    2  201
0.000166
9434649         3           87  295125214    2  201
0.000166
9434650         2           89  292571119    2  201
0.000166
9434651         3           89  292528142    2  201
0.000166
9434652         2           95  291956763    2  201
0.000166

[9434653 rows x 6 columns]
```

About the Epidemiological Simulation Data Set

This data set comes from a simulation which was run using a program called Indemics. It was developed by the Network Dynamics and Simulation Science Laboratory at Virginia Tech.

The references for the program are:

- Bisset KR, Chen J, Deodhar S, Feng X, Ma Y, Marathe MV. Indemics: An interactive high-performance computing framework for data intensive epidemic modeling. *ACM Transactions on Modeling and Computer Simulation*. 2014; 24(1):10. 1145/2501602. doi:10.1145/2501602.
- Deodhar S, Bisset K, Chen J, Ma Y, Marathe MV. Enhancing software capability through integration of distinct software in epidemiological systems. 2nd ACM SIGHIT International Health Informatics Symposium, 2012.
- Bisset KR, Chen J, Feng X, Ma Y, Marathe MV. Indemics: An interactive data intensive framework for high performance epidemic simulation. In *Proceedings the 24rd International Conference on Conference on Supercomputing*. 2010; 233-242.

The data set includes six columns:

- `ig_type`: edge type (type of relationship between two nodes in the network, such as “school” and “work”)
- `intervened`: time in the simulation at which an intervention occurred for a given person (`pid`)
- `pid`: simulated person’s ID number
- `rep`: replication run (each set of simulation parameters was run multiple times)
- `sid`: simulation ID
- `tr`: transmissibility value of the influenza virus

Let’s count the number of interventions for each replicate, intervention time, and treatment value. Here, we are counting the `ig_type` arbitrarily. We just need a value to get a count of observations for the groups.

Click here to view code image

```
count_only = (
  intv_df
  .groupby(["rep", "intervened", "tr"])
  ["ig_type"]
  .count()
)

print(count_only)
```

```
rep  intervened  tr
0    8           0.000166    1
     9           0.000152    3
                 0.000166    1
     10          0.000152    1
                 0.000166    1
                                ..
2    193         0.000135    1
```

```
                 0.000152    1
     195         0.000135    1
     198         0.000166    1
     199         0.000135    1
Name: ig_type, Length: 1196, dtype: int64
```

Now that we've done a `.groupby()` `.count()`, we can perform an additional `.groupby()` that calculates the average value. However, our initial `.groupby()` method does not return a regular flat dataframe.

Click here to view code image

```
print(type(count_only))
```

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

Instead, the results take the form of a multi-index series. If we want to do another `.groupby()` operation, we have to pass in the `levels` parameter to refer to the multi-index levels. Here we pass in `[0, 1, 2]` for the first, second, and third index levels, respectively.

Click here to view code image

```
count_mean = count_only.groupby(level=[0, 1,
2]).mean()
print(count_mean.head())
```

```
rep  intervened  tr
0    8           0.000166     1.0
     9           0.000152     3.0
                 0.000166     1.0
     10          0.000152     1.0
                 0.000166     1.0
Name: ig_type, dtype: float64
```

We can combine all of these operations in a single command.

Click here to view code image

```
count_mean = (
    intv_df
    .groupby(["rep", "intervened", "tr"])
["ig_type"]
    .count()
    .groupby(level=[0, 1, 2])
    .mean()
)
```

Figure 8.1 shows our results.

Click here to view code image

```
import seaborn as sns
import matplotlib.pyplot as plt


fig = sns.lmplot(
   data=count_mean.reset_index(),
   x="intervened",
   y="ig_type",
   hue="rep",
   col="tr",
   fit_reg=False,
   palette="viridis"
)

plt.show()
```

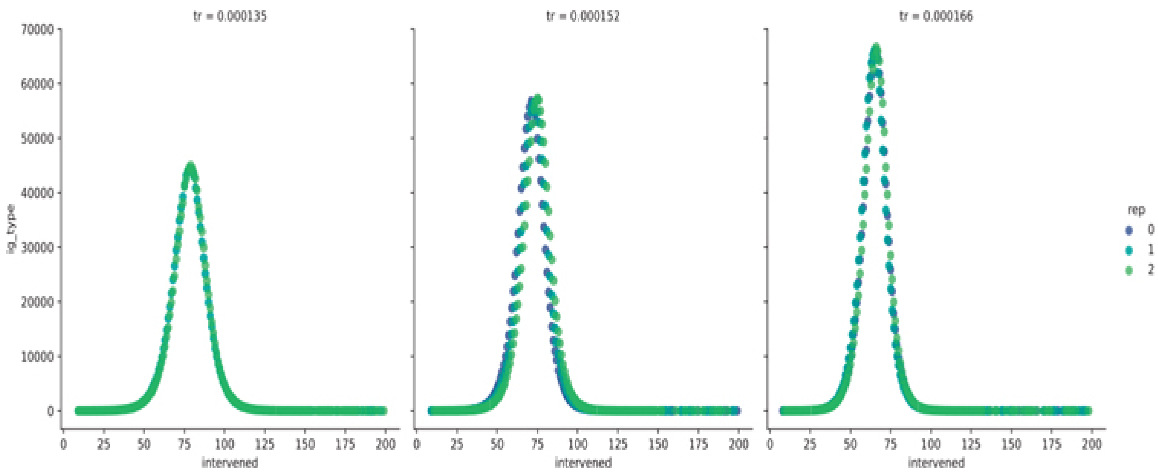


Figure 8.1 Grouped counts and mean

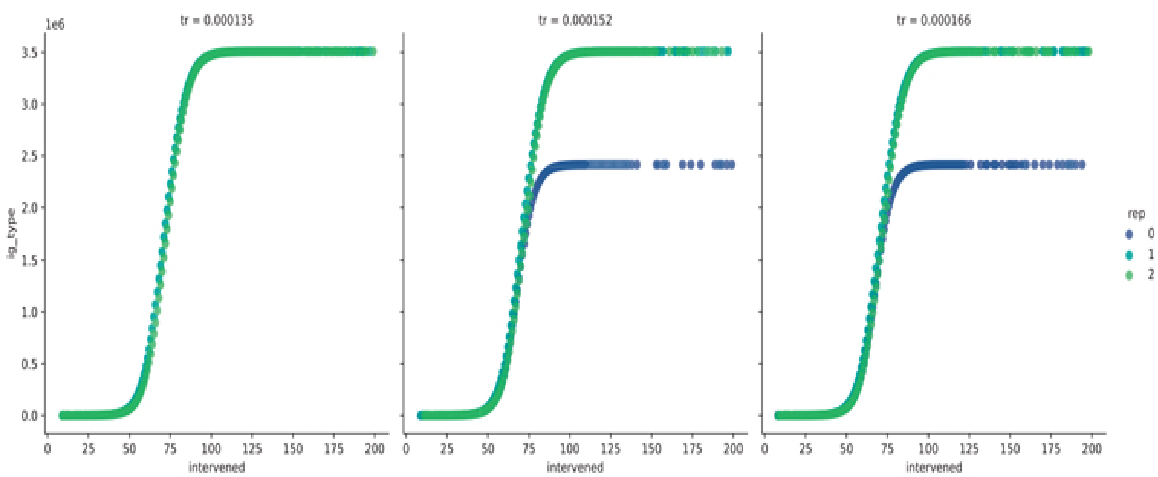


Figure 8.2 Grouped cumulative counts. The plot shows that one of the replicates did not run in our simulation.

The previous example showed how we can pass in a `level` to perform an additional `.groupby()` calculation. It used integer positions, but we can also pass in the string of the level to make our code a bit more readable.

Here, instead of looking at the `.mean()`, we will be using `.cumsum()` for the cumulative sum.

Figure 8.2 shows our results.

Click here to view code image

```
cumulative_count = (
  intv_df
  .groupby(["rep", "intervened", "tr"])
  ["ig_type"]
  .count()
  .groupby(level=["rep"])
  .cumsum()
  .reset_index()
)

fig = sns.lmplot(
  data=cumulative_count,
  x="intervened",
  y="ig_type",
  hue="rep",
  col="tr",
  fit_reg=False,
  palette="viridis"
)
plt.show()
```

Conclusion

The `.groupby()` statement follows the pattern of “split–apply–combine.”” It is a powerful concept that is not necessarily new to data analytics, but can help you think about your data and pipelines in a different way that will scale more readily to “big data” and “distributed” systems.

I urge you to check out the documentation for the `.groupby()` method and the general documentation for `.groupby()`, as there are many more complex things you can do with `groupby` statements. The material covered in this chapter should suffice for the vast majority of needs and use cases.

Part III

Data Types

After we have all the data we want, we can go into processing different parts of it. Working with missing data (Chapter 9), changing the data type stored in columns (Chapter 10), and working with string (Chapter 11) and date-time (Chapter 12) data are all common data types we need to be able to work with while cleaning and munging our data.

9

Missing Data

Rarely will you be given a data set without any missing values. There are many representations of missing data. In databases, they are `NULL` values; certain programming languages use `NA`; and depending on where you get your data, missing values can be an empty string, `''`, or even numeric values such as `88` or `99`. Pandas displays missing values as `NaN`.

Learning Objectives

- Identify how missing values are represented in pandas
- Recognize potential ways data can go missing in data processing
- Use different functions to fill in missing values

9.1 What Is a NaN Value?

The `NaN` value in Pandas comes from `numpy`. Missing values may be used or displayed in a few ways in Pandas — `NaN`, `NAN`, or `nan`— they are all the same in terms of how you specify a missing (floating point) number, but they are not the same in terms of equality. Appendix I describes how these missing values are imported.

Click here to view code image

```
# Just import the numpy missing values
from numpy import NaN, NAN, nan
```

Missing values are different than other types of data in that they don't really equal anything, not even to themselves. The data is missing, so there is no concept of equality. `NaN` is not equivalent to `0` or an empty string, ''. This is known as "three-valued logic."

```
print(NaN == True)
```

```
False
```

```
print(NaN == 0)
```

```
False
```

```
print(NaN == '')
```

```
False
```

```
print(NaN == NaN)
```

```
False
```

```
print(NaN == NAN)
```

```
False
```

```
print(NaN == nan)
```

```
False
```

```
print(nan == NAN)
```

```
False
```

Pandas has functions to test for missing values, `isnull()`.

```
import pandas as pd
```

```
print(pd.isnull(NaN))
```

```
True
```

```
print(pd.isnull(nan))
```

```
True
```

```
print(pd.isnull(NAN))
```

```
True
```

Pandas also has functions for testing non-missing values, `notnull()`.

```
print(pd.notnull(NaN))
```

```
False
```

```
print(pd.notnull(42))
```

```
True
```

```
print(pd.notnull('missing'))
```

```
True
```

9.2 Where Do Missing Values Come From?

We can get missing values when we load in a data set with missing values, or from the data munging process.

9.2.1 Load Data

The survey data we used in Chapter 6 included a data set, `visited`, that contained missing data. When we loaded the data, Pandas automatically found the missing data cell and gave us a dataframe with the `NaN` value in the appropriate cell. In the `read_csv()` function, three parameters relate to reading missing values: `na_values`, `keep_default_na`, and `na_filter`.

The `na_values` parameter allows you to specify additional missing or `NaN` values. You can pass in either a Python `str` (i.e., string) or a list-like object to be automatically coded as missing values when the file is read. Of course, default missing values, such as `NA`, `NaN`, or `nan`, are already available, which is why this parameter is not always used. Some health data may code `99` as a missing value; to specify the use of this value, you would set `na_values=[99]`.

The `keep_default_na` parameter is a `bool` (i.e., `True` or `False` boolean) that allows you to specify whether any additional values need to be considered as missing. This parameter is `True` by default, meaning any additional missing values specified with the `na_values` parameter will be appended to the list of missing values. However, `keep_default_na` can also be set to `keep_default_na=False`, which will **only** use the missing values specified in `na_values`.

Lastly, `na_filter` is a `bool` that will specify whether any values will be read as missing. The default value of `na_filter=True` means that missing values will be coded as `NaN`. If we assign `na_filter=False`, then nothing will be recoded as missing. This parameter can be thought of as a means to turn off all the parameters set for `na_values` and `keep_default_na`, but it is more likely to be used when you want to achieve a performance boost by loading in data without missing values.

Click here to view code image

```
# set the location for data
visited_file = 'data/survey_visited.csv'
```

```
print(pd.read_csv(visited_file))
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3         NaN
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

```
print(pd.read_csv(visited_file,
keep_default_na=False))
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

```
print(
  pd.read_csv(visited_file, na_values=[""],
keep_default_na=False)
)
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
```

```
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3         NaN
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

9.2.2 Merged Data

Chapter 6 showed you how to combine data sets. Some of the examples in that chapter included missing values in the output. If we recreate the merged table from Section 6.4.3, we will see missing values in the merged output.

Click here to view code image

```
visited = pd.read_csv('data/survey_visited.csv')
survey = pd.read_csv('data/survey_survey.csv')
```

```
print(visited)
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3         NaN
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

```
print(survey)
```

```
    taken person quant  reading
0     619   dyer   rad     9.82
```

```
1     619   dyer   sal      0.13
2     622   dyer   rad      7.80
3     622   dyer   sal      0.09
4     734     pb   rad      8.41
..    ...    ...   ...       ...
16    752    roe   sal     41.60
17    837   lake   rad      1.46
18    837   lake   sal      0.21
19    837    roe   sal     22.50
20    844    roe   rad     11.25

[21 rows x 4 columns]
```

```
vs = visited.merge(survey, left_on='ident',
right_on='taken')
print(vs)
```

```
    ident   site       dated  taken person quant
reading
0     619   DR-1  1927-02-08    619   dyer   rad
9.82
1     619   DR-1  1927-02-08    619   dyer   sal
0.13
2     622   DR-1  1927-02-10    622   dyer   rad
7.80
3     622   DR-1  1927-02-10    622   dyer   sal
0.09
4     734   DR-3  1939-01-07    734     pb   rad
8.41
..    ...    ...         ...    ...    ...   ...
...
16    752   DR-3         NaN    752    roe   sal
41.60
```

```
17     837   MSK-4   1932-01-14    837    lake    rad
1.46
18     837   MSK-4   1932-01-14    837    lake    sal
0.21
19     837   MSK-4   1932-01-14    837     roe    sal
22.50
20     844    DR-1   1932-03-22    844     roe    rad
11.25

[21 rows x 7 columns]
```

9.2.3 User Input Values

The user can also create missing values—for example, by creating a vector of values from a calculation or a manually curated vector. To build on the examples from Section 2.1, we will create our own data with missing values. `NaN` values are valid for both `Series` and `DataFrame` objects.

Click here to view code image

```
# missing value in a series
num_legs = pd.Series({'goat': 4, 'amoeba': nan})
print(num_legs)
```

```
goat       4.0
amoeba     NaN
dtype: float64
```

```
# missing value in a dataframe
scientists = pd.DataFrame(
  {
    "Name": ["Rosaline Franklin", "William
Gosset"],
```

```
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
    "missing": [NaN, nan],
  }
)
print(scientists)
```

```
                Name   Occupation        Born
Died  missing
0  Rosaline Franklin      Chemist  1920-07-25
1958-04-16      NaN
1     William Gosset  Statistician  1876-06-13
1937-10-16      NaN
```

You will notice the `dtype` of the `missing` column will be a `float64`. This is because the `NaN` missing value from `numpy` is a floating point value.

```
print(scientists.dtypes)
```

```
Name           object
Occupation     object
Born           object
Died           object
missing       float64
dtype: object
```

You can also assign a column of missing values to a dataframe directly.

Click here to view code image

```
# create a new dataframe
scientists = pd.DataFrame(
```

```
    {
      "Name": ["Rosaline Franklin", "William
Gosset"],
      "Occupation": ["Chemist", "Statistician"],
      "Born": ["1920-07-25", "1876-06-13"],
      "Died": ["1958-04-16", "1937-10-16"],
    }
)

# assign a column of missing values
scientists["missing"] = nan

print(scientists)
```

```
                Name    Occupation        Born
Died  missing
0  Rosaline Franklin       Chemist  1920-07-25
1958-04-16      NaN
1     William Gosset  Statistician  1876-06-13
1937-10-16      NaN
```

9.2.4 Reindexing

Another way to introduce missing values into your data is to reindex your dataframe. This is useful when you want to add new indices to your dataframe, but still want to retain its original values. A common usage is when the index represents some time interval, and you want to add more dates.

If we wanted to look at only the years from 2000 to 2010 from the Gapminder data plot in Section 1.5, we could perform the same grouped operations, subset the data, and then reindex it.

Click here to view code image

```
gapminder = pd.read_csv('data/gapminder.tsv',
sep='\t')

life_exp = gapminder.groupby(['year'])
['lifeExp'].mean()
print(life_exp)
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
          ...
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, Length: 12, dtype: float64
```

We can reindex by subsetting the data and use the `.reindex()` method.

Click here to view code image

```
# subset
y2000 = life_exp[life_exp.index > 2000]
print(y2000)
```

```
year
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

```
# reindex
print(y2000.reindex(range(2000, 2010)))
```

```
year
2000          NaN
2001          NaN
2002    65.694923
2003          NaN
2004          NaN
2005          NaN
2006          NaN
2007    67.007423
2008          NaN
2009          NaN
Name: lifeExp, dtype: float64
```

9.3 Working With Missing Data

Now that we know how missing values can be created, let's see how they behave when we are working with data.

9.3.1 Find and Count Missing Data

One way to look at the number of missing values is to `count()` them.

Click here to view code image

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

```
# count the number of non-missing values
print(ebola.count())
```

```
Date                       122
Day                        122
Cases_Guinea                93
Cases_Liberia               83
Cases_SierraLeone           87
                          ...
Deaths_Nigeria              38
Deaths_Senegal              22
Deaths_UnitedStates         18
Deaths_Spain                16
Deaths_Mali                 12
Length: 18, dtype: int64
```

You can also subtract the number of non-missing rows from the total number of rows.

Click here to view code image

```
num_rows = ebola.shape[0]
num_missing = num_rows - ebola.count()
print(num_missing)
```

```
Date                         0
Day                          0
Cases_Guinea                29
Cases_Liberia               39
Cases_SierraLeone           35
                          ...
Deaths_Nigeria              84
Deaths_Senegal             100
Deaths_UnitedStates        104
Deaths_Spain               106
Deaths_Mali                110
Length: 18, dtype: int64
```

If you want to count the total number of missing values in your data, or count the number of missing values for a particular column, you can use the `count_nonzero()` function from `numpy` in conjunction with the `.isnull()` method.

Click here to view code image

```
import numpy as np

print(np.count_nonzero(ebola.isnull()))
```

```
1214
```

```
print(np.count_nonzero(ebola['Cases_Guinea'].isn
ull()))
```

```
29
```

Another way to get missing data counts is to use the `.value_counts()` method on a series. This will print a frequency table of values. If you use the `dropna` parameter, you can also get a missing value count.

Click here to view code image

```
# value counts from the Cases_Guinea column
cnts =
ebola.Cases_Guinea.value_counts(dropna=False)
print(cnts)
```

```
NaN       29
86.0       3
495.0      2
112.0      2
```

```
390.0       2
           ..
1199.0      1
1298.0      1
1350.0      1
1472.0      1
49.0        1
Name: Cases_Guinea, Length: 89, dtype: int64
```

The results are sorted so you can subset the count vector to just look at the missing values.

Click here to view code image

```
# select the values in the Series where the
index is a NaN value
print(cnts.loc[pd.isnull(cnts.index)])
```

```
NaN    29
Name: Cases_Guinea, dtype: int64
```

In Python, `True` values equate to the integer value `1`, and `False` values equate to the integer value `0`. We can use this behavior to get the number of missing values by summing up a boolean vector with the `.sum()` method.

Click here to view code image

```
# check if the value is missing, and sum up the
results
print(ebola.Cases_Guinea.isnull().sum())
```

```
29
```

9.3.2 Clean Missing Data

There are many different ways we can deal with missing data. For example, we can replace the missing data with another value, fill in the missing data using existing data, or drop the data from our data set.

9.3.2.1 Recode or Replace

We can use the `.fillna()` method to recode the missing values to another value. For example, suppose we wanted the missing values to be recoded as a 0. When we use `.fillna()`, we can recode the values to a specific value.

Click here to view code image

```
# fill the missing values to 0 and only look at
the first 5 columns
print(ebola.fillna(0).iloc[:, 0:5])
```

```
           Date  Day  Cases_Guinea  Cases_Liberia
Cases_SierraLeone
0      1/5/2015  289        2776.0            0.0
10030.0
1      1/4/2015  288        2775.0            0.0
9780.0
2      1/3/2015  287        2769.0         8166.0
9722.0
3      1/2/2015  286           0.0         8157.0
0.0
4    12/31/2014  284        2730.0         8115.0
9633.0
..          ...  ...           ...            ...
...
117   3/27/2014    5         103.0            8.0
6.0
```

```
118   3/26/2014    4          86.0             0.0
0.0
119   3/25/2014    3          86.0             0.0
0.0
120   3/24/2014    2          86.0             0.0
0.0
121   3/22/2014    0          49.0             0.0
0.0

[122 rows x 5 columns]
```

9.3.2.2 Forward Fill

We can use built-in methods to fill forward or backward. When we fill data forward, the last known value (from top to bottom) is used for the next missing value. In this way, missing values are replaced with the last known and recorded value.

Click here to view code image

```
print(ebola.fillna(method='ffill').iloc[:, 0:5])
```

```
           Date  Day  Cases_Guinea  Cases_Liberia
Cases_SierraLeone
0      1/5/2015  289        2776.0            NaN
10030.0
1      1/4/2015  288        2775.0            NaN
9780.0
2      1/3/2015  287        2769.0         8166.0
9722.0
3      1/2/2015  286        2769.0         8157.0
9722.0
4    12/31/2014  284        2730.0         8115.0
9633.0
```

```
..              ...   ...              ...              ...
...
117   3/27/2014    5           103.0              8.0
6.0
118   3/26/2014    4            86.0              8.0
6.0
119   3/25/2014    3            86.0              8.0
6.0
120   3/24/2014    2            86.0              8.0
6.0
121   3/22/2014    0            49.0              8.0
6.0

[122 rows x 5 columns]
```

If a column begins with a missing value, then that data will remain missing because there is no previous value to fill in.

9.3.2.3 Backward Fill

We can also have Pandas fill data backward. When we fill data backward, the newest value (from top to bottom) is used to replace the missing data. In this way, missing values are replaced with the newest value.

Click here to view code image

```
print(ebola.fillna(method='bfill').iloc[:, 0:5])
```

```
           Date  Day  Cases_Guinea  Cases_Liberia
Cases_SierraLeone
0      1/5/2015  289        2776.0         8166.0
10030.0
1      1/4/2015  288        2775.0         8166.0
9780.0
2      1/3/2015  287        2769.0         8166.0
```

```
9722.0
3       1/2/2015  286         2730.0          8157.0
9633.0
4     12/31/2014  284         2730.0          8115.0
9633.0
..           ...  ...            ...             ...
...
117    3/27/2014    5          103.0             8.0
6.0
118    3/26/2014    4           86.0             NaN
NaN
119    3/25/2014    3           86.0             NaN
NaN
120    3/24/2014    2           86.0             NaN
NaN
121    3/22/2014    0           49.0             NaN
NaN

[122 rows x 5 columns]
```

If a column ends with a missing value, then it will remain missing because there is no new value to fill in.

9.3.2.4 Interpolate

Interpolation uses existing values to fill in missing values. There are many ways to fill in missing values, the interpolation in Pandas fills in missing values linearly. Specifically, it treats the missing values as if they should be equally spaced apart.

Click here to view code image

```
print(ebola.interpolate().iloc[:, 0:5])
```

```
           Date  Day  Cases_Guinea  Cases_Liberia
Cases_SierraLeone
0      1/5/2015  289        2776.0            NaN
10030.0
1      1/4/2015  288        2775.0            NaN
9780.0
2      1/3/2015  287        2769.0         8166.0
9722.0
3      1/2/2015  286        2749.5         8157.0
9677.5
4    12/31/2014  284        2730.0         8115.0
9633.0
..          ...  ...           ...            ...
...
117   3/27/2014    5         103.0            8.0
6.0
118   3/26/2014    4          86.0            8.0
6.0
119   3/25/2014    3          86.0            8.0
6.0
120   3/24/2014    2          86.0            8.0
6.0
121   3/22/2014    0          49.0            8.0
6.0

[122 rows x 5 columns]
```

Notice how it behaves kind of in a forward fill fashion, but instead of passing on the last known value, it will fill in the differences between values.

The `.interpolate()` method has a `method` parameter that can change the interpolation method.[1] Possible values at the time of writing have been reproduced in Table 9.1.

1. `Series.interpolate()` documentation: https://pandas.pydata.org/docs/reference/api/pandas.Series.interpolate.html

Table 9.1 Possible Values (at the Time of Writing) to Pass Into the `method` Parameter in the `.interpolate()` Method

	Technique	Description
1	linear	Ignore the index and treat the values as equally spaced. This is the only method supported on Multi-Indexes
2	time	Works on daily and higher resolution data to interpolate given length of interval
3	index, values	Use the actual numerical values of the index
4	pad	Fill in NaNs using existing values
5	nearest, zero, slinear, quadratic, cubic, spline, barycentric, polynomial	Passed to `scipy.interpolate.interp1d`; these methods use the numerical values of the index
6	krogh, piecewise_polynomial, spline, pchip, akima, cubicspline	Wrappers around the SciPy interpolation methods of similar names
7	from_derivatives	Refers to `scipy.interpolate.BPoly`

9.3.2.5 Drop Missing Values

The last way to work with missing data is to drop observations or variables with missing data. Depending on how much data is missing, keeping only complete case data can leave you with a useless data set. Perhaps the missing data is not random, so that dropping missing values will leave you with a biased data set, or perhaps keeping only complete data will leave you with insufficient data to run your analysis.

We can use the `.dropna()` method to drop missing data, and specify parameters to this method that control how data are dropped. For instance, the `how` parameter lets you specify whether a row (or column) is dropped when `'any'` or `'all'` of the data is missing. The `thresh` parameter lets you specify how many non-`NaN` values you have before dropping the row or column.

```
print(ebola.shape)
```

```
(122, 18)
```

If we keep only complete cases in our Ebola data set, we are left with just one row of data.

Click here to view code image

```
ebola_dropna = ebola.dropna()
print(ebola_dropna.shape)
```

```
(1, 18)
```

```
print(ebola_dropna)
```

```
          Date  Day  Cases_Guinea  Cases_Liberia
Cases_SierraLeone  \
19  11/18/2014  241        2047.0         7082.0
6190.0

    Cases_Nigeria  Cases_Senegal
Cases_UnitedStates  Cases_Spain  \
19           20.0             1.0
4.0          1.0

    Cases_Mali  Deaths_Guinea  Deaths_Liberia
Deaths_SierraLeone  \
```

```
19         6.0         1214.0          2963.0
1267.0

    Deaths_Nigeria  Deaths_Senegal
Deaths_UnitedStates  \
19             8.0             0.0
1.0

    Deaths_Spain  Deaths_Mali
19           0.0          6.0
```

9.3.3 Calculations With Missing Data

Suppose we wanted to look at the case counts for multiple regions. We can add multiple regions together to get a new column holding the case counts.

```
ebola["Cases_multiple"] = (
  ebola["Cases_Guinea"]
  + ebola["Cases_Liberia"]
  + ebola["Cases_SierraLeone"]
)
```

Let's look at the first 10 lines of the calculation.

Click here to view code image

```
ebola_subset = ebola.loc[
    :,
    [
        "Cases_Guinea",
        "Cases_Liberia",
        "Cases_SierraLeone",
        "Cases_multiple",
    ],
```

```
]
print(ebola_subset.head(n=10))
```

```
   Cases_Guinea  Cases_Liberia  Cases_SierraLeone
Cases_multiple
0        2776.0            NaN            10030.0
NaN
1        2775.0            NaN             9780.0
NaN
2        2769.0         8166.0             9722.0
20657.0
3           NaN         8157.0                NaN
NaN
4        2730.0         8115.0             9633.0
20478.0
5        2706.0         8018.0             9446.0
20170.0
6        2695.0            NaN             9409.0
NaN
7        2630.0         7977.0             9203.0
19810.0
8        2597.0      NaN 9004.0                NaN
9        2571.0         7862.0             8939.0
19372.0
```

You can see that a value for `Cases_multiple` was calculated only when there was no missing value for `Cases_Guinea`, `Cases_Liberia`, and `Cases_SierraLeone`. Calculations with missing values will typically return a missing value, unless the function or method called has a means to ignore missing values in its calculations.

Examples of built-in methods that can ignore missing values include `.mean()` and `.sum()`. These functions will typically have a `skipna` parameter that will still calculate a value by skipping over the missing values.

Click here to view code image

```
# skipping missing values is True by default
print(ebola.Cases_Guinea.sum(skipna = True))
```

```
84729.0
```

```
print(ebola.Cases_Guinea.sum(skipna = False))
```

```
nan
```

9.4 Pandas Built-In NA Missing

Pandas 1.0 introduced a built-in `NA` value (`pd.NA`). At the time of writing this feature is still "experimental."[2] The main goal of this feature is to provide a missing value that works across different data types.

2. Pandas experimental `NA`: https://pandas.pydata.org/docs/user_guide/missing_data.html#experimental-na-scalar-to-denote-missing-values

Let's use our previous `scientists` data set from earlier and look at the `.dtypes`.

Click here to view code image

```
scientists = pd.DataFrame(
  {
    "Name": ["Rosaline Franklin", "William
Gosset"],
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
    "Age": [37, 61]
  }
```

```python
)

print(scientists)
```

```
                  Name    Occupation        Born
Died  Age
0  Rosaline Franklin       Chemist  1920-07-25
1958-04-16   37
1     William Gosset  Statistician  1876-06-13
1937-10-16   61
```

```python
print(scientists.dtypes)
```

```
Name             object
Occupation       object
Born             object
Died             object
Age               int64
dtype: object
```

```python
scientists.loc[1, "Name"] = pd.NA
scientists.loc[1, "Age"] = pd.NA

print(scientists)
```

```
                  Name    Occupation        Born
Died   Age
0  Rosaline Franklin        Chemist  1920-07-25
1958-04-16    37
1                <NA>  Statistician  1876-06-13
1937-10-16  <NA>
```

```python
print(scientists.dtypes)
```

```
Name            object
Occupation      object
Born            object
Died            object
Age             object
dtype: object
```

Compare the `.dtypes` from `pd.NA` and `np.NaN` from earlier in this chapter.

Click here to view code image

```
scientists = pd.DataFrame(
  {
    "Name": ["Rosaline Franklin", "William
Gosset"],
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
    "Age": [37, 61]
  }
)

scientists.loc[1, "Name"] = np.NaN
scientists.loc[1, "Age"] = np.NaN

print(scientists.dtypes)
```

```
Name             object
Occupation       object
Born             object
Died             object
Age             float64
dtype: object
```

Since `pd.NA` is still experimental, best follow up with its behavior in the official documentation.

Conclusion

It is rare to have a data set without any missing values. It is important to know how to work with missing values because, even when you are working with data that is complete, missing values can still arise from your own data munging. In this chapter, we examined some of the basic methods used in the data analysis process that pertain to data validity. By looking at your data and tabulating missing values, you can start the process of assessing whether the data is of sufficiently high quality for making decisions and drawing inferences.

10

Data Types

Data types determine what can and cannot be done to a variable (i.e., column). For example, when numeric data types are added together, the result will be a sum of the values; in contrast, if strings (in Pandas they are `object` or `string` types) are added, the strings will be concatenated together.

This chapter presents a quick overview of the various data types you may encounter in Pandas, and means to convert from one data type to another.

Learning Objectives

- Recognize columns in data store the same data type
- Identify what kind of data type is stored in a column
- Use functions to change the type of a column
- Modify categorical columns

10.1 Data Types

In this chapter, we'll use the built-in `tips` data set from the `seaborn` library.

Click here to view code image

```
import pandas as pd
import seaborn as sns

tips = sns.load_data set("tips")
```

To get a list of the data types stored in each column of our dataframe, we call the dtypes attribute (Section 1.2).

```
print(tips.dtypes)
```

```
total_bill     float64
tip            float64
sex           category
smoker        category
day           category
time          category
size             int64
dtype: object
```

Table 1.1 listed the various types of data that can be stored in a Pandas column. Our data set includes data of types int64, float64, and category. The int64 and float64 types represent numeric values without and with decimal points, respectively. The number following the numeric data type represents the number of bits of information that will be stored for that particular number.

The category data type represents categorical variables. It differs from the generic object data type that stores arbitrary Python objects (usually strings). We will explore these differences later in this chapter. Since the tips data set is a fully prepared and cleaned data set, variables that store strings were saved as a category.

10.2 Converting Types

The data type that is stored in a column will govern which kinds of functions and calculations you can perform on the data found in that column. Clearly, then, it's important to know how to convert between data types.

This section focuses on how to convert from one data type to another. Keep in mind that you need not do all your data type conversions at once, when you first get your data. Data analytics is not a linear process, and you can choose to convert types on the fly as needed. We saw an example of this in Section 2.4.2, when we converted a date value into just the number of years.

10.2.1 Converting to String Objects

In our `tips` data, the `sex`, `smoker`, `day`, and `time` variables are stored as a `category`. In general, it's much easier to work with string `object` types when the variable is not a numeric number. There are performance benefits from using a `category` data type, however.

Some data sets may have an `id` column in which the `id` is stored as a number, but has no meaning if you perform a calculation on it (e.g., if you try to find the mean). Unique identifiers or `id` numbers are typically coded this way, and you may want to convert them to string `object` types depending on what you need.

To convert values into strings, we use the `.astype()` method on the column (i.e., `Series`).[1] The `.astype()` method takes a parameter, `dtype`, which will be the new data type the column will take on. In this case, we want to convert the `sex` variable to a string `object`, `str`.

1. `Series.astype()` method documentation: https://pandas.pydata.org/pandas-docs/version/0.23/generated/pandas.Series.astype.html

Click here to view code image

```
# convert the category sex column into a string
dtype
tips['sex_str'] = tips['sex'].astype(str)
```

Python has built-in `str`, `float`, `int`, `complex`, and `bool` types. However, you can also specify any `dtype` from the `numpy` library. If we look at the `dtypes` now, you will see the `sex_str` now has a `dtype` of `object`.

```
print(tips.dtypes)
```

```
total_bill        float64
tip               float64
sex              category
smoker           category
day              category
time             category
size                int64
sex_str            object
dtype: object
```

10.2.2 Converting to Numeric Values

The `.astype()` method is a generic function that can be used to convert any column in a dataframe to another `dtype`.

Recall that each column in a `DataFrame` is a Pandas `Series` object. That's why the `.astype()` documentation is listed under `pandas.Series.astype`. The example here shows how to change the type of a `DataFrame` column, but if you are working with a `Series` object, you can use the same `.astype()` method to convert the `Series` as well.

We can provide any built-in or `numpy` type to the `.astype()` method to convert the `dtype` of the column. For example, if we wanted to convert the `total_bill` column first to a string `object` and then back to its original `float64`, we can pass in `str` and `float` into `astype`, respectively.

Click here to view code image

```
# convert total_bill into a string
tips['total_bill'] =
tips['total_bill'].astype(str)
print(tips.dtypes)
```

```
total_bill     object
tip           float64
sex          category
smoker       category
day          category
time         category
size            int64
sex_str        object
dtype: object
```

```
# convert it back to a float
tips['total_bill'] =
tips['total_bill'].astype(float)
print(tips.dtypes)
```

```
total_bill        float64
tip               float64
sex              category
smoker           category
day              category
time             category
size                int64
sex_str            object
dtype: object
```

10.2.2.1 The `.to_numeric()` Method

When converting variables into numeric values (e.g., `int`, `float`), you can also use the Pandas `to_numeric()` function, which handles non-numeric values better.

Since each column in a dataframe has to have the same `dtype`, there will be times when a numeric column contains strings as some of its values. For example, instead of the NaN value that represents a missing value in Pandas, a numeric column might use the string 'missing' or 'null' for this purpose instead. This would make the entire column a string object type instead of a numeric type.

Let's subset our tips dataframe and also put in a 'missing' value in the total_bill column to illustrate how the to_numeric() function works.

> **Note**
>
> We use the .copy() method here to avoid the SettingWithCopyWarning message when we modify the subsetted data set (Appendix T).

Click here to view code image

```
# subset the tips data
tips_sub_miss = tips.head(10).copy()
```

```
# assign some 'missing' values
tips_sub_miss.loc[[1, 3, 5, 7], 'total_bill'] =
'missing'
```

```
print(tips_sub_miss)
```

```
  total_bill  tip    sex smoker day   time size
sex_str
0      16.99 1.01 Female     No Sun Dinner    2
Female
1    missing 1.66   Male     No Sun Dinner    3
Male
2      21.01 3.50   Male     No Sun Dinner    3
Male
3    missing 3.31   Male     No Sun Dinner    2
```

```
Male
4       24.59 3.61 Female     No Sun Dinner    4
Female
5     missing 4.71   Male     No Sun Dinner    4
Male
6        8.77 2.00   Male     No Sun Dinner    2
Male
7     missing 3.12   Male     No Sun Dinner    4
Male
8       15.04 1.96   Male     No Sun Dinner    2
Male
9       14.78 3.23   Male     No Sun Dinner    2
Male
```

Looking at the `dtypes`, you will see that the `total_bill` column is now a string `object`.

```
print(tips_sub_miss.dtypes)
```

```
total_bill          object
tip                float64
sex               category
smoker            category
day               category
time              category
size                 int64
sex_str             object
dtype: object
```

If we now try to use the `.astype()` method to convert the column back to a `float`, we will get an error: Pandas does not know how to convert `'missing'` into a `float`.

Click here to view code image

```
# this will cause an error
tips_sub_miss['total_bill'].astype(float)
```

```
ValueError: could not convert string to float:
'missing'
```

If we use the `to_numeric()` function from the `pandas` library, we get a similar error.

Click here to view code image

```
# this will cause an error
pd.to_numeric(tips_sub_miss['total_bill'])
```

```
ValueError: Unable to parse string "missing" at
position 1
```

The `to_numeric()` function has a parameter called `errors` that governs what happens when the function encounters a value that it is unable to convert to a numeric value. By default, this value is set to `'raise'`; that is, if it does encounter a value it is unable to convert to a numeric value, it will `'raise'` an error.

Based on the documentation:[2]

2. `to_numeric()` function documentation: `https://pandas.pydata.org/docs/reference/api/pandas.to_numeric.html`

- 'raise', then invalid parsing will raise an exception
- 'coerce', then invalid parsing will be set as NaN
- 'ignore', then invalid parsing will return the input

Going out of order from the documentation, if we pass `errors` the `'ignore'` value, nothing will change in our column. But we also do not get an error message, which may not always be the behavior we want.

Click here to view code image

```python
tips_sub_miss["total_bill"] = pd.to_numeric(
    tips_sub_miss["total_bill"], errors="ignore"
)

print(tips_sub_miss)
```

```
  total_bill  tip    sex smoker day   time size
sex_str
0       16.99 1.01 Female     No Sun Dinner    2
Female
1     missing 1.66   Male     No Sun Dinner    3
Male
2       21.01 3.50   Male     No Sun Dinner    3
Male
3     missing 3.31   Male     No Sun Dinner    2
Male
4       24.59 3.61 Female     No Sun Dinner    4
Female
5     missing 4.71   Male     No Sun Dinner    4
Male
6        8.77 2.00   Male     No Sun Dinner    2
Male
7     missing 3.12   Male     No Sun Dinner    4
Male
8       15.04 1.96   Male     No Sun Dinner    2
Male
9       14.78 3.23   Male     No Sun Dinner    2
Male
```

```python
print(tips_sub_miss.dtypes)
```

```
total_bill      object
tip            float64
sex           category
smoker        category
day           category
time          category
size             int64
sex_str         object
dtype: object
```

In contrast, if we pass in the `'coerce'` value, we will get `NaN` values for the `'missing'` string.

Click here to view code image

```
tips_sub_miss["total_bill"] = pd.to_numeric(
    tips_sub_miss["total_bill"], errors="coerce"
)
```

```
print(tips_sub_miss)
```

```
   total_bill  tip    sex smoker day   time size
sex_str
0        16.99 1.01 Female     No Sun Dinner    2
Female
1          NaN 1.66   Male     No Sun Dinner    3
Male
2        21.01 3.50   Male     No Sun Dinner    3
Male
3          NaN 3.31   Male     No Sun Dinner    2
Male
4        24.59 3.61 Female     No Sun Dinner    4
Female
5          NaN 4.71   Male     No Sun Dinner    4
```

```
Male
6         8.77 2.00   Male      No Sun Dinner    2
Male
7          NaN 3.12   Male      No Sun Dinner    4
Male
8        15.04 1.96   Male      No Sun Dinner    2
Male
9        14.78 3.23   Male      No Sun Dinner    2
Male
```

```
print(tips_sub_miss.dtypes)
```

```
total_bill     float64
tip            float64
sex           category
smoker        category
day           category
time          category
size             int64
sex_str         object
dtype: object
```

This is a useful trick when you know a column must contain numeric values, but for some reason the data include non-numeric values.

10.3 Categorical Data

Not all data values are numeric. Pandas has a `category dtype` that can encode categorical values.[3] Here are a few use cases for categorical data:

3. Categorical data: `https://pandas.pydata.org/docs/user_guide/categorical.html`

- It can be memory and speed efficient to store data in this manner, especially if the data set includes many repeated string values

- Categorical data may be appropriate when a column of values has an order (e.g., a Likert scale)
- Some Python libraries understand how to deal with categorical data (e.g., when fitting statistical models)

10.3.1 Convert to Category

To convert a column into a categorical type, we pass `category` into the `.astype()` method.

Click here to view code image

```
# convert the sex column into a string object
first
tips['sex'] = tips['sex'].astype('str')
print(tips.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 8 columns):

 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   total_bill  244 non-null    float64
 1   tip         244 non-null    float64
 2   sex         244 non-null    object
 3   smoker      244 non-null    category
 4   day         244 non-null    category
 5   time        244 non-null    category
 6   size        244 non-null    int64
 7   sex_str     244 non-null    object
dtypes: category(3), float64(2), int64(1),
object(2)
```

```
memory usage: 10.8+ KB
None
```

Click here to view code image

```
# convert the sex column back into categorical
data
tips['sex'] = tips['sex'].astype('category')
print(tips.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 8 columns):

 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   total_bill   244 non-null    float64
 1   tip          244 non-null    float64
 2   sex          244 non-null    category
 3   smoker       244 non-null    category
 4   day          244 non-null    category
 5   time         244 non-null    category
 6   size         244 non-null    int64
 7   sex_str      244 non-null    object
dtypes: category(4), float64(2), int64(1),
object(1)
memory usage: 9.3+ KB
None
```

You can also see the difference in memory usage from the string and category storage.

10.3.2 Manipulating Categorical Data

The API reference has a list of which operations can be performed on a categorical `Series`.[4] The `.cat.` accessor is an attribute that allows you to access the category information in the `Series`. This list has been reproduced in Table 10.1.

4. The `.cat.` accessor: `https://pandas.pydata.org/docs/reference/series.html#categorical-accessor`

Table 10.1 Categorical Accessor Attributes and Methods

Attribute or Method	**Description**
`Series.cat.categories`	The categories
`Series.cat.ordered`	Whether the categories are ordered
`Series.cat.codes`	Return the integer code of the category
`Series.cat.rename_categories()`	Rename categories
`Series.cat.reorder_categories()`	Reorder categories
`Series.cat.add_categories()`	Add new categories
`Series.cat.remove_categories()`	Remove categories
`Series.cat.remove_unused_categories()`	Remove unused categories
`Series.cat.set_categories())`	Set new categories
`Series.cat.as_ordered()`	Make the category ordered
`Series.cat.as_unordered()`	Make the category unordered

Conclusion

This chapter covered how to convert from one data type to another. `dtypes` govern which operations can and cannot be performed on a column. While this chapter is relatively short, converting types is an important skill when you are working with data and when you are using other Pandas methods.

11

Strings and Text Data

Introduction

Most data in the world can be stored as text and strings. Even values that may eventually be numeric data may initially come in the form of text. It's important to be able to work with text data. This chapter won't be specific to Pandas. That is, we will mainly explore how you manipulate strings within Python without Pandas. The following chapters will cover some more Pandas materials. Then we will come back to strings and see how it all ties back with Pandas. As an aside, some of the string examples in this chapter come from *Monty Python and the Holy Grail*.

Learning Objectives

- Recall how to subset containers and sequences
- Recognize strings are a type of container object
- Modify strings based on use case
- Create regular expression patterns to match strings
- Combine pose text with code output into a single sentence

11.1 Strings

In Python, a `string` is simply a series of characters. They are created by a set of opening and matching single or double quotes. Below are two strings, `grail` and `a scratch`. These strings are assigned to the variables `word` and `sent`, respectively.

```
word = 'grail'
sent = 'a scratch'
```

So far in this book, we have seen strings in a column represented as the `object dtype`.

11.1.1 Subset and Slice Strings

A string can be thought of as a container of characters. You can subset a string like any other Python container (e.g., `list` or `Series`).

Table 11.1 and Table 11.2 show the strings with their associated index. This information will help you understand the examples in which we slice values using the index.

Table 11.1 Index Positions for the String "grail"

index	0	1	2	3	4
string	g	r	a	i	l
neg index	–5	–4	–3	–2	–1

Table 11.2 Index Positions for the String "a scratch"

index	0	1	2	3	4	5	6	7	8
string	a		s	c	r	a	t	c	h
neg index	–9	–8	–7	–6	–5	–4	–3	–2	–1

11.1.1.1 Single Letter

To get the first letter of our strings, we can use the square bracket notation, `[ ]`. This notation is the same method we used in Section 1.3 when we looked at various slices of data.

```
print(word[0])
```

```
g
```

```
print(sent[3])
```

```
c
```

11.1.1.2 Slice Multiple Letters

Alternatively, we can use slicing notation (Appendix L) to get ranges from our strings.

Click here to view code image

```
# get the first 3 characters
# note index 3 is really the 4th character
print(word[0:3])
```

```
gra
```

Recall that when using slicing notation in Python, it is left-side inclusive, right-side exclusive. In other words, it will include the index value specified first, but it will not include the index value specified second.

For example, the notation `[0:3]` will include the characters from `0` to `3`, but not index `3`. Another way to say this is to state that `[0:3]` will include the indices from `0` to `2`, inclusive.

11.1.1.3 Negative Numbers

Recall that in Python, passing in a negative index actually starts the count from the **end** of a container.

Click here to view code image

```
# get the last letter from "a scratch"
print(sent[ -1])
```

```
h
```

The negative index refers to the index position as well, so you can also use it to slice values.

```
# get 'a' from "a scratch"
print(sent[ -9: -8])
```

```
a
```

You can combine non-negative numbers with negative numbers.

```
# get 'a'
print(sent[0: -8])
```

```
a
```

Note that you can't actually get the last letter when using a negative index for the second value.

```
# scratch
print(sent[2: -1])
```

```
scratc
```

```
# scratch
print(sent[ -7: -1])
```

```
scratc
```

11.1.2 Get the Last Character in a String

Just getting the last element in a string (or any container) can be done with the negative index, `-1`. However, it becomes problematic when we want to use slicing notation and also include the last character. For example, if we tried to use slicing notation to get the word "scratch" from the `sent` variable, the result returned would be one letter short.

Since Python is right-side exclusive, we need to specify an index position that is one greater than the last index. To do this, we can get the `len` (length) of the string and then pass that value into the slicing notation.

Click here to view code image

```
# note that the last index is one position is
smaller than
# the number returned for len
s_len = len(sent)
print(s_len)
```

```
9
```

```
print(sent[2:s_len])
```

```
scratch
```

11.1.2.1 Slice from the Beginning or to the End

A very common task is to slice a value from the beginning to a certain point in the string (or container). The first element will always be `0`, so we can always write something like `word[0:3]` to get the first three elements, or `word[-3:len(word)]` to get the last three elements.

Another shortcut for this task is to leave out the data on the left or right side of the `:`. If the left side of the `:` is empty, then the slice will start from the beginning and end at the index on the right (non-inclusive). If the right side of the `:` is empty, then the slice will start from the index on the left, and end at the end of the string. For example, these slices are equivalent:

```
print(word[0:3])
```

```
gra
```

```
# left the left side empty
print(word[ :3])
```

```
gra
```

```
print(sent[2:len(sent)])
```

```
scratch
```

```
# leave the right side empty
print(sent[2: ])
```

```
scratch
```

Another way to specify the entire string is to leave both values empty.

```
print(sent[:])
```

```
a scratch
```

11.1.2.2 Slice Increments (Steps)

The final notation while slicing allows you to slice in increments. To do this, you use a second colon, `:`, to provide a third number. The third number allows you to specify the increment to pull values out.

For example, you can get every other string by passing in `2` for every second character.

Click here to view code image

```
# step by 2, to get every other character
print(sent[::2])
```

```
asrth
```

Any integer can be passed here, so if you wanted every third character (or value in a container), you could pass in `3`.

```
# get every third character
print(sent[::3])
```

```
act
```

11.2 String Methods

Many methods are also used when processing data in Python. A list of all the string methods can be found on the "String Methods" documentation page.[1] Table 11.3 and Table 11.4 summarize some string methods that are commonly used in Python.

1. String methods: `https://docs.python.org/3/library/stdtypes.html#string-methods`

Table 11.3 Python String Methods

Method	Description
`.capitalize()`	Capitalizes the first character
`.count()`	Counts the number of occurrences of a string
`.startswith()`	`True` if the string begins with specified value
`.endswith()`	`True` if the string ends with specified value

Method	Description
`.find()`	Smallest index of where the string matched, `-1` if no match
`.index()`	Same as find but returns `ValueError` if no match
`.isalpha()`	`True` if all characters are alphabetic
`.isdecimal()`	`True` if all characters are decimal numbers (see documentation as well as `.isdigit()`, `.isnumeric()`, and `.isalnum()`)
`.isalnum()`	`True` if all characters are alphanumeric (alphabetic or numeric)
`.lower()`	Copy of a string with all lowercase letters
`.upper()`	Copy of string with all uppercase letters
`.replace()`	Copy of a string with the `old` values replaced with `new`
`.strip()`	Removes leading and trailing whitespace; also see `lstrip` and `rstrip`
`.split()`	Returns a list of values split by the delimiter (separator)
`.partition()`	Similar to `split(maxsplit=1)` but also returns the separator
`.center()`	Centers the string to a given width
`.zfill()`	Copy of string left filled with `'0'`

Table 11.4 Examples of Using Python String Methods

Code	Results
"black Knight".capitalize()	'Black knight'
"It's just a flesh wound!".count('u')	2
"Halt! Who goes there?".startswith('Halt')	True
"coconut".endswith('nut')	True
"It's just a flesh wound!".find('u')	7
"It's just a flesh wound!".index('scratch')	ValueError
"old woman".isalpha()	False (there is a whitespace)
"37".isdecimal()	True
"I'm 37".isalnum()	False (apostrophe and space)
"Black Knight".lower()	'black knight'
"Black Knight".upper()	'BLACK KNIGHT'
"flesh wound!".replace('flesh wound', 'scratch')	'scratch!'
" I'm not dead. ".strip()	"I'm not dead."
"NI! NI! NI! NI!".split(sep=' ')	['NI!', 'NI!', 'NI!', 'NI!']
"3,4.partition(',')	('3', ',', '4')
"nine".center(width=10)	' nine '
"9".zfill(with=5)	'00009'

11.3 More String Methods

There are a few more string methods that are useful, but hard to convey in a table.

11.3.1 Join

The `.join()` method takes a container (e.g., a `list`) and returns a new string that combines each element in the list. For example, suppose we wanted to combine coordinates in the degrees, minutes, seconds (DMS) notation.

```
d1 = '40°'
m1 = "46'"
s1 = '52.837"'
u1 = 'N'

d2 = '73°'
m2 = "58'"
s2 = '26.302"'
u2 = 'W'
```

We can join all the values with a space, `' '`, by using the `.join()` method on the space string.

Click here to view code image

```
coords = ' '.join([d1, m1, s1, u1, d2, m2, s2,
u2])
print(coords)
```

```
40° 46' 52.837" N 73° 58' 26.302" W
```

This method is also useful if you have a list of strings that you want to separate using your own delimiter (e.g., tabs with `\t` and commas with `,`). If we wanted, we could now `.split()` on a space, `" "`, and get the individual parts from `coords`.

Click here to view code image

```
coords.split(" ")
```

```
['40°', "46'", '52.837"', 'N', '73°', "58'",
'26.302"', 'W']
```

11.3.2 Splitlines

The `.splitlines()` method is similar to the `.split()` method. It is typically used on strings that are multiple lines long and will return a list in which each element of the list is a line in the multiple-line string.

> **Note**
>
> You can create a multi-line string in Python by beginning and ending the string with a triple-quote, `'''` or `"""`.

Click here to view code image

```
multi_str = """Guard: What? Ridden on a horse?
King Arthur: Yes!
Guard: You're using coconuts!
King Arthur: What?
Guard: You've got ... coconut[s] and you're
bangin' 'em together.
"""

print(multi_str)
```

```
Guard: What? Ridden on a horse?
King Arthur: Yes!
Guard: You're using coconuts!
King Arthur: What?
```

```
Guard: You've got ... coconut[s] and you're
bangin' 'em together.
```

We can get every line as a separate element in a list using `.splitlines()`.

Click here to view code image

```
multi_str_split = multi_str.splitlines()
```

```
print(multi_str_split)
```

```
[
  "Guard: What? Ridden on a horse?",
  "King Arthur: Yes!",
  "Guard: You're using coconuts!",
  "King Arthur: What?",
  "Guard: You've got ... coconut[s] and you're
bangin' 'em together."
]
```

Finally, suppose we just wanted the text from the “Guard”. This is a two-person conversation, so the “Guard” speaks every other line.

Click here to view code image

```
guard = multi_str_split[::2]
```

```
print(guard)
```

```
[
  "Guard: What? Ridden on a horse?",
  "Guard: You're using coconuts!",
  "Guard: You've got ... coconut[s] and you're
```

```
bangin' 'em together."
]
```

There are a few ways to just get the lines from the “Guard”. One way would be to use the `.replace()` method on the string and `.replace()` the `Guard:` string with an empty string `''`. We could then use the `.splitlines()` method.

Click here to view code image

```
guard = multi_str.replace("Guard:
",""").splitlines()[::2]
```

```
print(guard)
```

```
[
  "What? Ridden on a horse?",
  "You're using coconuts!",
  "You've got ... coconut[s] and you're bangin'
'em together."
]
```

11.4 String Formatting (F-Strings)

Formatting strings allows you to specify a generic template for a string, and insert variables into the pattern. It can also handle various ways to visually represent strings—for example, showing two decimal values in a `float`, or showing a number as a percentage instead of a decimal value.

String formatting can even help when you want to print something to the console. Instead of just printing out the variable, you can print a string that provides hints about the value that is printed.

This chapter will only talk about “formatted literal strings”, also known as f-strings, which were introduced in Python 3.6. Older C-Style formatting and the `.format()` method have been moved to Appendix W.1 and Appendix W.2, respectively.

To create an f-string, we will write our strings as `f""`:

```
s = f"hello"
print(s)
```

```
hello
```

This tells the string that it is an f-string. This now allows us to use `{ }` within the string to put in Python variables or calculations.

Click here to view code image

```
num = 7
s = f"I only know {num} digits of pi."
print(s)
```

```
I only know 7 digits of pi.
```

This allows us to create readable strings using Python variables. You can put in different types of objects within a f-string.

Click here to view code image

```
const = "e"
value = 2.718
s = f"Some digits of {const}: {value}"
print(s)
```

```
Some digits of e: 2.718
```

```
lat = "40.7815° N"
lon = "73.9733° W"
s = f"Hayden Planetarium Coordinates: {lat},
{lon}"
print(s)
```

```
Hayden Planetarium Coordinates: 40.7815° N,
73.9733° W
```

Variables can be reused within a f-string.

Click here to view code image

```
word = "scratch"

s = f"""Black Knight:  'Tis but a {word}.
King Arthur: A {word}? Your arm's off!
"""
print(s)
```

```
Black Knight: 'Tis but a scratch.
King Arthur: A scratch? Your arm's off!
```

11.4.1 Formatting Numbers

Numbers can also be formatted.

Click here to view code image

```
p = 3.14159265359
print(f"Some digits of pi: {p}")
```

```
Some digits of pi: 3.14159265359
```

You can specify how to format a placeholder by using the optional colon character, :, and use the format specification mini-language[2] to change how it outputs in the string. Here is an example of formatting numbers and use thousands-place comma separators.

2. String formatting mini-language: https://docs.python.org/3.4/library/string.html#format-string-syntax

Click here to view code image

```
digits = 67890
s = f"In 2005, Lu Chao of China recited
{67890:,} digits of pi."
print(s)
```

```
In 2005, Lu Chao of China recited 67,890 digits of
pi.
```

The formatting mini-language also supports how many decimal values are displayed.

Click here to view code image

```
prop = 7 / 67890
s = f"I remember {prop:.4} or {prop:.4%} of what
Lu Chao recited."
print(s)
```

```
I remember 0.0001031 or 0.0103% of what Lu Chao
recited.
```

We can also use the formatting mini-language to left pad a digit with `0`.

Click here to view code image

```
id = 42
print(f"My ID number is {id:05d}")
```

```
My ID number is 00042
```

In the `:05d`, the colon tells us we are going to provide a formatting pattern, the `0` is the character we will use to pad, and the `5d` tells us to pad with 5 digits.

Sometimes we can use the formatting mini-language, but we can also use a lot of the built-in string methods as well.

Click here to view code image

```
id_zfill = "42".zfill(5)
print(f"My ID number is {id_zfill}")
```

```
My ID number is 00042
```

Or we can put in a python expression directly in the f-string.

Click here to view code image

```
print(f"My ID number is {'42'.zfill(5)}")
```

```
My ID number is 00042
```

It is usually better to do all the function calls *before* creating the f-string, so all you are passing into the f-string is a variable. This just makes the code easier to read.

11.5 Regular Expressions (RegEx)

When the base Python string methods that search for patterns aren't enough, you can throw the kitchen sink at the problem by using regular expressions (regex). The extremely powerful regular expressions provide a (nontrivial) way to find and match patterns in strings. The downside is that after you finish writing a complex regular expression, it becomes difficult to figure out what the pattern does by looking at it. That is, the syntax is difficult to read.

For many data tasks, such as matching a telephone number or address field validation, it's almost easier to Google which type of pattern you are trying to match, and paste what someone has already written into your own code (don't forget to document where you got the pattern from).

Before continuing, you might want to visit regex101.[3] It's a great place and reference for regular expressions and testing patterns on test strings. It even has a Python mode, so you can directly copy/paste a pattern from the site into your own Python code.

Regular expressions in Python use the `re` module.[4] This module also has a great How To[5] that can be used as an additional resource.

3. Regex101 website: https://regex101.com/

4. `re` module documentation: https://docs.python.org/3/library/re.html

5. Regular Expression HOWTO: https://docs.python.org/3/howto/regex.html#regex-howto

Table 11.5 and Table 11.6 show some RegEx syntax and special characters that will be used in this section.

Table 11.5 Basic RegEx Syntax

Syntax	Description
`.`	Matches any one character
`^`	Matches from the beginning of a string
`$`	Matches from the end of a string
`*`	Matches zero or more repetitions of the previous character
`+`	Matches one or more repetitions of the previous character
`?`	Matches zero or one repetition of the previous character
`{m}`	Matches `m` repetitions of the previous character
`{m,n}`	Matches any number from `m` to `n` of the previous character
`\`	Escape character
`[ ]`	A set of characters (e.g., `[a-z]` will match all letters from a to z)
`|`	OR; `A | B` will match `A` or `B`

Syntax	Description
`( )`	Matches the pattern specified within the parentheses exactly

Table 11.6 **RegEx Special Characters**

Sequence	Description
`\d`	A digit
`\D`	Any character NOT a digit (opposite of `\d`)
`\s`	Any whitespace character
`\S`	Any character NOT a whitespace (opposite of `\s`)
`\w`	Word characters
`\W`	Any character NOT a word character (opposite of `\w`)

To use regular expressions, we write a string that contains the RegEx pattern, and provide a string for the pattern to match. Various functions within `re` can be used to handle specific needs. Some common tasks are provided in Table 11.7.

Table 11.7 **Common RegEx Functions in `re`**

Function	Description
`search()`	Find the first occurrence of a string
`match()`	Match from the beginning of a string
`fullmatch()`	Match the entire string
`split()`	Split string by the pattern
`findall()`	Find all non-overlapping matches of a string
`finditer()`	Similar to `findall` but returns a Python iterator
`sub()`	Substitute the matched pattern with the provided string

11.5.1 Match a Pattern

We will be using the `re` module to write the regular expression pattern we want to match in a string. Let's write a pattern that will match 10 digits (the digits for a U.S. telephone number).

```
import re

tele_num = '1234567890'
```

There are many ways we can match 10 consecutive digits. We can use the `match()` function to see if the pattern matches a string. The output of many `re` functions is a `match` object.

Click here to view code image

```
m = re.match(pattern='\d\d\d\d\d\d\d\d\d\d',
string=tele_num)
print(type(m))
```

```
<class 're.Match'>
```

```
print(m)
```

```
<re.Match object; span=(0, 10),
match='1234567890'>
```

If we look at the printed match object, we see that, if there was a match, the `span` identifies the index of the string where the matches occurred, and the `match` identifies the exact string that got matched.

Many times when we are matching a pattern to a string, we simply want a `True` or `False` value indicating whether there was a match. If you just need a `True/False` value returned, you can run the built-in `bool()` function to get the boolean value of the match object.

```
print(bool(m))
```

```
True
```

At other times, a regular expression match will be part of an `if` statement (Appendix X), so this kind of `bool()` casting is unnecessary.

```
# should print match
if m:
  print('match')
else:
  print('no match')
```

```
match
```

If we wanted to extract some of the match object values, such as the index positions or the actual string that matched, we can use a few methods on the match object.

Click here to view code image

```
# get the first index of the string match
print(m.start())
```

```
0
```

```
# get the last index of the string match
print(m.end())
```

```
10
```

```
# get the first and last index of the string
match
print(m.span())
```

```
(0, 10)
```

```
# the string that matched the pattern
print(m.group())
```

```
1234567890
```

Telephone numbers can be a little more complex than a series of 10 consecutive digits. Here's another common representation.

Click here to view code image

```
tele_num_spaces = '123 456 7890'
```

Suppose we use the previous pattern in this example.

Click here to view code image

```
# we can simplify the previous pattern
m = re.match(pattern='\d{10}',
string=tele_num_spaces)
print(m)
```

```
None
```

You can tell the pattern did not match because the match object returned `None`. If we run our `if` statement again, it will print `'no match'`.

```
if m:
    print('match')
else:
    print('no match')
```

```
no match
```

Let's modify our pattern this time, by assuming the new string has three digits, a space, another three digits, and another space, followed by four digits. If we want to make it general to the original example, the spaces can be matched zero or one time. The new RegEx pattern will look like the following code:

Click here to view code image

```
# you may see the RegEx pattern as a separate
variable
# because it can get long and
# make the actual match function call hard to
read
p = '\d{3}\s?\d{3}\s?\d{4}'
m = re.match(pattern=p, string=tele_num_spaces)
print(m)
```

```
<re.Match object; span=(0, 12), match='123 456
7890'>
```

Area codes can also be surrounded by parentheses and a dash between the seven main digits.

Click here to view code image

```
tele_num_space_paren_dash = '(123) 456-7890'
p = '\(?\d{3}\)?\s?\d{3}\s?-?\d{4}'
m = re.match(pattern=p,
string=tele_num_space_paren_dash)
print(m)
```

```
<re.Match object; span=(0, 14), match='(123) 456-
7890'>
```

Finally, there could be a country code before the number.

Click here to view code image

```
cnty_tele_num_space_paren_dash = '+1 (123) 456-
7890'
p = '\+?1\s?\(?\d{3}\)?\s?\d{3}\s?-?\d{4}'
m = re.match(pattern=p,
string=cnty_tele_num_space_paren_dash)
print(m)
```

```
<re.Match object; span=(0, 17), match='+1 (123)
456-7890'>
```

As these examples suggest, although powerful, regular expressions can easily become unwieldy. Even something as simple as a telephone number can lead to a daunting series of symbols and numbers. Even so, sometimes regular expressions are the only way to get something done.

11.5.2 Remember What Your RegEx Patterns Are

The last regular expression of a phone number had many complex components. Chances are you forget what most of the pattern means after you write it, let alone trying to figure out what it means when you eventually review back your code.

Let's see how we can re-write the last example in a more maintainable way, by utilizing one of the quirks of the Python language.

In Python 2 strings next to each other will be concatenated and joined together into a single string.

Click here to view code image

```
"multiple" "strings" "next" "to" "each" "other"
```

```
'multiplestringsnexttoeachother'
```

Note that no extra delimiter, space, or character is added between subsequent strings, they are just concatenated together.

Tip

You can also use this trick with really long URLs that you want to split across multiple lines.

That also means that we could break up our long pattern string across multiple lines. We can tell python to treat all the separate strings as a single value that we can assign to a variable by wrapping the statement around a pair of round parentheses, `( )`.

Click here to view code image

```
p = (
    '\+?'
    '1'
    '\s?'
    '\(?'
    '\d{3}'
     '\)?'
    '\s?'
    '\d{3}'
    '\s?'
    '-?'
    '\d{4}'
)
print(p)
```

```
\+?1\s?\(?\d{3}\)?\s?\d{3}\s?-?\d{4}
```

Now that we have our code across multiple lines, we can add comments to our string, as if it was regular Python code.

Click here to view code image

```
p = (
  '\+?'      # maybe starts with a +
  '1'        # the number 1
  '\s?'      # maybe there's a whitespace
  '\(?'      # maybe there's an open round
parenthesis (
  '\d{3}'    # 3 numbers
  '\)?'      # maybe there's a closing round
parenthesis )
  '\s?'      # maybe there's a whitespace
  '\d{3}'    # 3 numbers
  '\s?'      # maybe there's a whitespace
  '-?'       # maybe there's a dash character
  '\d{4}'    # 4 numbers
)
print(p)
```

```
\+?1\s?\(?\d{3}\)?\s?\d{3}\s?-?\d{4}
```

This technique allows you to write your regular expressions in a manner that you can understand later on, and make it easier to debug the pattern if something is not matching as you expect.

Click here to view code image

```
cnty_tele_num_space_paren_dash = '+1 (123) 456-
7890'
m = re.match(pattern=p,
string=cnty_tele_num_space_paren_dash)
print(m)
```

```
<re.Match object; span=(0, 17), match='+1 (123)
456-7890'>
```

11.5.3 Find a Pattern

We can use the `findall()` function to find all matches within a pattern. Let's write a pattern that matches digits and uses it to find all the digits from a string.

Click here to view code image

```
# python will concatenate 2 strings next to each
other
s = (
  "14 Ncuti Gatwa, "
  "13 Jodie Whittaker, war John Hurt, 12 Peter
Capaldi, "
  "11 Matt Smith, 10 David Tennant, 9
Christopher Eccleston"
)

print(s)
```

```
14 Ncuti Gatwa, 13 Jodie Whittaker, war John Hurt,
12 Peter Capaldi,
11 Matt Smith, 10 David Tennant, 9 Christopher
Eccleston
```

Click here to view code image

```
# pattern to match 1 or more digits
p = "\d+"

m = re.findall(pattern=p, string=s)
print(m)
```

```
['14', '13', '12', '11', '10', '9']
```

11.5.4 Substitute a Pattern

In our `str.replace()` example (Section 11.3.2), we wanted to get all the lines from the `Guard`, so we ended up doing a direct string replacement on the script. However, using regular expressions, we can generalize the pattern so we can get either the line from the `Guard` or the line from `King Arthur`.

Click here to view code image

```
multi_str = """Guard: What? Ridden on a horse?
King Arthur: Yes!
Guard: You're using coconuts!
King Arthur: What?
Guard: You've got ... coconut[s] and you're
bangin' 'em together.
"""

p = '\w+\s?\w+:\s?'

s = re.sub(pattern=p, string=multi_str, repl='')
print(s)
```

```
What? Ridden on a horse?
Yes!
You're using coconuts!
What?
You've got ... coconut[s] and you're bangin' 'em
together.
```

Now we can get either party's line by using string slicing with increments.

Click here to view code image

```python
guard = s.splitlines()[ ::2]
kinga = s.splitlines()[1::2] # skip the first
element
```

```python
print(guard)
```

```
[
  "What? Ridden on a horse?",
  "You're using coconuts!",
  "You've got ... coconut[s] and you're bangin'
'em together."
]
```

```python
print(kinga)
```

```
[
  "Yes!",
  "What?"
]
```

Don't be afraid to mix and match regular expressions with the simpler pattern match and string methods.

11.5.5 Compile a Pattern

When we work with data, typically many operations will occur on a column-by-column or row-by-row basis. Python's `re` module allows you to `compile()` a pattern so it can be reused. This can lead to performance benefits, especially if your data set is large. Here we will see how to compile a pattern and use it just as we did in the previous examples in this section.

The syntax is almost the same. We write our regular expression pattern, but this time, instead of saving it to a variable directly, we pass the string into the `compile()` function and save that result. We can then use the other `re` functions on the compiled pattern. Also, since the pattern is already compiled, you no longer need to specify the pattern parameter in the method.

Here is the `match()` example:

Click here to view code image

```
# pattern to match 10 digits
p = re.compile('\d{10}')
s = '1234567890'

# note: calling match on the compiled pattern
# not using the re.match function
m = p.match(s)
print(m)
```

```
<re.Match object; span=(0, 10),
match='1234567890'>
```

The `findall()` example:

Click here to view code image

```
p = re.compile('\d+')
s = (
  "14 Ncuti Gatwa, "
  "13 Jodie Whittaker, war John Hurt, 12 Peter
Capaldi, "
  "11 Matt Smith, 10 David Tennant, 9
Christopher Eccleston"
)

m = p.findall(s)
print(m)
```

```
['14', '13', '12', '11', '10', '9']
```

The `sub()` or substitution example:

Click here to view code image

```
p = re.compile('\w+\s?\w+:\s?')
s = "Guard: You're using coconuts!"

m = p.sub(string=s, repl='')
print(m)
```

```
You're using coconuts!
```

11.6 The regex Library

The `re` library is popular because it comes with the Python installation. However, seasoned regular expression writers may find the `regex` library to have more comprehensive features. It is backward compatible with the `re` library, so all the code from the `re` RegEx section (Section 11.5) will still work with the `regex` library. The documentation for this library can be found on the PyPI page.[6]

6. `regex` documentation: `https://pypi.python.org/pypi/regex`

Click here to view code image

```
import regex

# a re example using the regex library
p = regex.compile('\d+')
s = (
  "14 Ncuti Gatwa, "
  "13 Jodie Whittaker, war John Hurt, 12 Peter
Capaldi, "
  "11 Matt Smith, 10 David Tennant, 9
```

```
Christopher Eccleston"
)

m = p.findall(s)
print(m)
```

```
['14', '13', '12', '11', '10', '9']
```

I will defer to the examples and explanations on `http://www.rexegg.com/` for more details:

- `www.rexegg.com/regex-python.html`
- `www.rexegg.com/regex-best-trick.html`

Conclusion

The world is filled with data stored as text. Understanding how to manipulate text strings is a fundamental skill for the data scientist. Python has many built-in string methods and libraries that can make string and text manipulation easier. This chapter covered some of the fundamental methods of string manipulations that we can build on when working with data.

12

Dates and Times

One of the bigger reasons for using Pandas is its ability to work with timeseries data. We observed some of this capability earlier, when we concatenated data in Chapter 6 and saw how the indices automatically aligned themselves. This chapter focuses on the more common tasks encountered when working with data that involve dates and times.

Learning Objectives

- Create date objects with the `datetime` library
- Use functions to convert strings into a date
- Use functions to format dates
- Perform date calculations
- Use functions to resample dates
- Use functions to work with and convert time zones

12.1 Python's `datetime` Object

Python has a built-in `datetime` object that is found in the `datetime` library.

Click here to view code image

```
from datetime import datetime
```

We can use `datetime` to get the current date and time.

Click here to view code image

```
now = datetime.now()
print(f"Last time this chapter was rendered for
print: {now}")
```

```
Last time this chapter was rendered for print:
2022-09-01 01:55:41.496795
```

We can also create our own `datetime` manually.

```
t1 = datetime.now()
t2 = datetime(1970, 1,1)
```

And we can do `datetime` math.

```
diff = t1 - t2
print(diff)
```

```
19236 days, 1:55:41.499914
```

The data type of a date calculation is a `timedelta`.

```
print(type(diff))
```

```
<class 'datetime.timedelta'>
```

We can perform these types of actions when working within a Pandas dataframe.

12.2 Converting to datetime

Converting an object type into a `datetime` type is done with the `to_datatime` function. Let's load up our Ebola data set and convert the `Date` column into a proper `datetime` object.

Click here to view code image

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

# top left corner of the data
print(ebola.iloc[:5, :5])
```

```
        Date Day Cases_Guinea Cases_Liberia
Cases_SierraLeone
0   1/5/2015 289       2776.0           NaN
10030.0
1   1/4/2015 288       2775.0           NaN
9780.0
2   1/3/2015 287       2769.0        8166.0
9722.0
3   1/2/2015 286          NaN        8157.0
NaN
4 12/31/2014 284       2730.0        8115.0
9633.0
```

The first `Date` column contains date information, but the `.info()` attribute tells us it is actually encoded as a generic string `object` in Pandas.

Click here to view code image

```
print(ebola.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 18 columns):

 #   Column              Non-Null Count Dtype
---  ------              -------------- -----
```

```
 0   Date                 122 non-null   object
 1   Day                  122 non-null   int64
 2   Cases_Guinea         93 non-null    float64
 3   Cases_Liberia        83 non-null    float64
 4   Cases_SierraLeone    87 non-null    float64
 5   Cases_Nigeria        38 non-null    float64
 6   Cases_Senegal        25 non-null    float64
 7   Cases_UnitedStates   18 non-null    float64
 8   Cases_Spain          16 non-null    float64
 9   Cases_Mali           12 non-null    float64
 10  Deaths_Guinea        92 non-null    float64
 11  Deaths_Liberia       81 non-null    float64
 12  Deaths_SierraLeone   87 non-null    float64
 13  Deaths_Nigeria       38 non-null    float64
 14  Deaths_Senegal       22 non-null    float64
 15  Deaths_UnitedStates  18 non-null    float64
 16  Deaths_Spain         16 non-null    float64
 17  Deaths_Mali          12 non-null    float64
dtypes: float64(16), int64(1), object(1)
memory usage: 17.3+ KB
None
```

We can create a new column, `date_dt`, that converts the `Date` column into a `datetime`.

Click here to view code image

```
ebola['date_dt'] = pd.to_datetime(ebola['Date'])
```

We can also be a little more explicit with how we convert data into a `datetime` object.

The `to_datetime()` function has a parameter called `format` that allows you to manually specify the `format` of the date you are hoping to parse. Since our date is in a `month/day/year` format, we can pass in the string `%m/%d/%Y`.

Click here to view code image

```
ebola['date_dt'] = pd.to_datetime(ebola['Date'],
format='%m/%d/%Y')
```

In both cases, we end up with a new column with a `datetime` type.

Click here to view code image

```
print(ebola.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 21 columns):

 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Date                 122 non-null    object
 1   Day                  122 non-null    int64
 2   Cases_Guinea         93 non-null    float64
 3   Cases_Liberia        83 non-null    float64
 4   Cases_SierraLeone    87 non-null    float64
 5   Cases_Nigeria        38 non-null    float64
 6   Cases_Senegal        25 non-null    float64
 7   Cases_UnitedStates   18 non-null    float64
 8   Cases_Spain          16 non-null    float64
 9   Cases_Mali           12 non-null    float64
 10  Deaths_Guinea        92 non-null    float64
 11  Deaths_Liberia       81 non-null    float64
 12  Deaths_SierraLeone   87 non-null    float64
 13  Deaths_Nigeria       38 non-null    float64
 14  Deaths_Senegal       22 non-null    float64
 15  Deaths_UnitedStates 18 non-null    float64
 16  Deaths_Spain         16 non-null    float64
```

```
 17 Deaths_Mali           12 non-null    float64
 18 date_dt               122 non-null
datetime64[ns]
 19 date_dt_a             122 non-null
datetime64[ns]
 20 date_dt_al            122 non-null
datetime64[ns]
dtypes: datetime64[ns](3), float64(16), int64(1),
object(1)
memory usage: 20.1+ KB
None
```

The `to_datetime()` function includes convenient built-in options. For example, you can set the `dayfirst` or `yearfirst` options to `True` if the date format begins with a day (e.g., `31-03-2014`) or if the date begins with a year (e.g., `2014-03-31`), respectively.

For other date formats, you can manually specify how they are represented using the syntax specified by python's `strptime`.[1] This syntax is replicated in Table 12.1 from the official Python documentation.

1. `strftime` (string format time) and `strptime` (string parse time) behavior: `https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior`

Table 12.1 Python strftime and strptime Behavior (reproduced from the official Python documentation[2])

Directive	Meaning	Example
%a	Weekday abbreviated name	Sun, Mon, …, Sat
%A	Weekday full name	Sunday, Monday, …, Saturday
%w	Weekday as a number, where 0 is Sunday	0, 1, …, 6

Directive	Meaning	Example
%d	Day of the month as a two-digit number	01, 02, …, 31
%b	Month abbreviated name	Jan, Feb, …, Dec
%B	Month full name	January, February, …, December
%m	Month as a two-digit number	01, 02, …, 12
%y	Year as a two-digit number	00, 01, …, 99
%Y	Year as a four-digit number	0001, 0002, …, 2013, 2014, …, 9999
%H	Hour (24-hour clock) as a two-digit number	00, 01, …, 23
%I	Hour (12-hour clock) as a two-digit number	01, 02, …, 12
%p	AM or PM	AM, PM
%M	Minute as a two-digit number	00, 01, …, 59
%S	Second as a two-digit number	00, 01, …, 59
%f	Microsecond as a number	000000, 000001, …, 999999
%z	UTC offset in the form of +HHMM or \hbox{--HHMM}	(empty), +0000, -0400, +1030
%Z	Time zone name	(empty), UTC, EST, CST
%j	Day of the year as a three-digit number	001, 002, …, 366
%U	Week number of the year (Sunday first)	00, 01, …, 53
%W	Week number of the year (Monday first)	00, 01, …, 53
%c	Date and time representation	Tue Aug 16 21:30:00 1988

Directive	Meaning	Example
%x	Date representation	08/16/88 (None);08/16/1988
%X	Time representation	21:30:00
%%	Literal % character	%
%G	ISO 8601 year	0001, 0002, …, 2013, 2014, …, 9999
%u	ISO 8601 weekday	1, 2, …, 7
%V	ISO 8601 week	01, 02, …, 53

2. `strftime` (string format time) and `strptime` (string parse time) behavior: `https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior`

12.3 Loading Data That Include Dates

Many of the data sets used in this book are in a CSV format, or else they come from the `seaborn` library. The `gapminder` data set was an exception: It was a tab-separated file (TSV). The `read_csv()` function has a lot of parameters – for example, `parse_dates`, `inher_datetime_format`, `keep_date_col`, `date_parser`, `dayfirst`, and `cache_dates`. We can parse the `Date` column directly by specifying the column we want in the `parse_dates` parameter.

Click here to view code image

```
ebola =
pd.read_csv('data/country_timeseries.csv',
parse_dates=["Date"])
print(ebola.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 18 columns):

 #   Column               Non-Null Count Dtype
---  ------               -------------- -----
 0   Date                 122 non-null
datetime64[ns]
 1   Day                  122 non-null   int64
 2   Cases_Guinea         93 non-null   float64
 3   Cases_Liberia        83 non-null   float64
 4   Cases_SierraLeone    87 non-null   float64
 5   Cases_Nigeria        38 non-null   float64
 6   Cases_Senegal        25 non-null   float64
 7   Cases_UnitedStates   18 non-null   float64
 8   Cases_Spain          16 non-null   float64
 9   Cases_Mali           12 non-null   float64
 10  Deaths_Guinea        92 non-null   float64
 11  Deaths_Liberia       81 non-null   float64
 12  Deaths_SierraLeone   87 non-null   float64
 13  Deaths_Nigeria       38 non-null   float64
 14  Deaths_Senegal       22 non-null   float64
 15  Deaths_UnitedStates 18 non-null   float64
 16  Deaths_Spain         16 non-null   float64
 17  Deaths_Mali          12 non-null   float64
dtypes: datetime64[ns](1), float64(16), int64(1)
memory usage: 17.3 KB
None
```

This example shows how we can automatically convert columns into dates directly when the data are loaded.

12.4 Extracting Date Components

Now that we have a `datetime` object, we can extract various parts of the date, such as year, month, or day. Here's an example `datetime` object.

Click here to view code image

```
d = pd.to_datetime('2021-12-14')
print(d)
```

```
2021-12-14 00:00:00
```

If we pass in a single string, we get a `Timestamp`.

Click here to view code image

```
print(type(d))
```

```
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

Now that we have a proper `datetime`, we can access various date components as attributes.

```
print(d.year)
```

```
2021
```

```
print(d.month)
```

```
12
```

```
print(d.day)
```

```
14
```

In Chapter 4, we tidied our data when we needed to parse a column that stored multiple bits of information and used the `.str.` accessor to use string methods like `.split()`. We can do something similar here with datetime objects by accessing datetime methods using the .dt. accessor.[3] Let's first re-create our date_dt column.

3. Datetime-like properties: https://pandas.pydata.org/docs/reference/series.html#datetimelike-properties

Click here to view code image

```
ebola['date_dt'] = pd.to_datetime(ebola['Date'])
```

We know we can get date components such as the year, month, and day by using the year, month, and day attributes, respectively, on a column basis; we saw how this works when we parsed strings in a column using .str.. Here's the Date and date_dt columns we just created.

Click here to view code image

```
print(ebola[['Date', 'date_dt']])
```

```
           Date     date_dt
0    2015-01-05  2015-01-05
1    2015-01-04  2015-01-04
2    2015-01-03  2015-01-03
3    2015-01-02  2015-01-02
4    2014-12-31  2014-12-31
..          ...         ...
117  2014-03-27  2014-03-27
118  2014-03-26  2014-03-26
119  2014-03-25  2014-03-25
120  2014-03-24  2014-03-24
121  2014-03-22  2014-03-22

[122 rows x 2 columns]
```

We can create a new year column based on the Date column.

Click here to view code image

```
ebola['year'] = ebola['date_dt'].dt.year
print(ebola[['Date', 'date_dt', 'year']])
```

```
           Date    date_dt  year
0    2015-01-05 2015-01-05  2015
1    2015-01-04 2015-01-04  2015
2    2015-01-03 2015-01-03  2015
3    2015-01-02 2015-01-02  2015
4    2014-12-31 2014-12-31  2014
..          ...        ...   ...
117  2014-03-27 2014-03-27  2014
118  2014-03-26 2014-03-26  2014
119  2014-03-25 2014-03-25  2014
120  2014-03-24 2014-03-24  2014
121  2014-03-22 2014-03-22  2014

[122 rows x 3 columns]
```

Let's finish parsing our date.

Click here to view code image

```
ebola = ebola.assign(
    month=ebola["date_dt"].dt.month,
    day=ebola["date_dt"].dt.day
)
```

```
print(ebola[['Date', 'date_dt', 'year', 'month',
'day']])
```

```
           Date    date_dt  year  month  day
0    2015-01-05 2015-01-05  2015      1    5
```

```
1   2015-01-04 2015-01-04 2015     1   4
2   2015-01-03 2015-01-03 2015     1   3
3   2015-01-02 2015-01-02 2015     1   2
4   2014-12-31 2014-12-31 2014    12  31
..         ...        ...  ...   ... ...
117 2014-03-27 2014-03-27 2014     3  27
118 2014-03-26 2014-03-26 2014     3  26
119 2014-03-25 2014-03-25 2014     3  25
120 2014-03-24 2014-03-24 2014     3  24
121 2014-03-22 2014-03-22 2014     3  22

[122 rows x 5 columns]
```

When we parsed out our dates, the data type was not preserved.

Click here to view code image

```
print(ebola.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 22 columns):

 #   Column               Non-Null Count Dtype
---  ------               -------------- -----
 0   Date                 122 non-null
datetime64[ns]
 1   Day                  122 non-null   int64
 2   Cases_Guinea         93 non-null    float64
 3   Cases_Liberia        83 non-null    float64
 4   Cases_SierraLeone    87 non-null    float64
 5   Cases_Nigeria        38 non-null    float64
 6   Cases_Senegal        25 non-null    float64
```

```
 7  Cases_UnitedStates  18 non-null    float64
 8  Cases_Spain         16 non-null    float64
 9  Cases_Mali          12 non-null    float64
 10 Deaths_Guinea       92 non-null    float64
 11 Deaths_Liberia      81 non-null    float64
 12 Deaths_SierraLeone  87 non-null    float64
 13 Deaths_Nigeria      38 non-null    float64
 14 Deaths_Senegal      22 non-null    float64
 15 Deaths_UnitedStates 18 non-null    float64
 16 Deaths_Spain        16 non-null    float64
 17 Deaths_Mali         12 non-null    float64
 18 date_dt             122 non-null
datetime64[ns]
 19 year                122 non-null   int64
 20 month               122 non-null   int64
 21 day                 122 non-null   int64
dtypes: datetime64[ns](2), float64(16), int64(4)
memory usage: 21.1 KB
None
```

12.5 Date Calculations and Timedeltas

One of the benefits of having date objects is being able to do date calculations. Our Ebola data set includes a column named `Day` that indicates how many days into an Ebola outbreak a country is. We can recreate this column using date arithmetic. Here's the bottom left corner of our data.

Click here to view code image

```
print(ebola.iloc[-5:, :5])
```

```
          Date   Day   Cases_Guinea   Cases_Liberia
Cases_SierraLeone
```

```
117 2014-03-27    5          103.0                8.0
6.0
118 2014-03-26    4           86.0                NaN
NaN
119 2014-03-25    3           86.0                NaN
NaN
120 2014-03-24    2           86.0                NaN
NaN
121 2014-03-22    0           49.0                NaN
NaN
```

The first day of the outbreak (the earliest date in this data set) is `2015-03-22`. So, if we want to calculate the number of days into the outbreak, we can subtract this date from each date by using the `.min()` method of the column.

```
print(ebola['date_dt'].min())
```

```
2014-03-22 00:00:00
```

We can use this date in our calculation.

Click here to view code image

```
ebola['outbreak_d'] = ebola['date_dt'] -
ebola['date_dt'].min()
```

```
print(ebola[['Date', 'Day', 'outbreak_d']])
```

```
          Date Day  outbreak_d
0   2015-01-05 289    289 days
1   2015-01-04 288    288 days
2   2015-01-03 287    287 days
3   2015-01-02 286    286 days
4   2014-12-31 284    284 days
```

```
..          ...  ...        ...
117 2014-03-27   5       5 days
118 2014-03-26   4       4 days
119 2014-03-25   3       3 days
120 2014-03-24   2       2 days
121 2014-03-22   0       0 days

[122 rows x 3 columns]
```

When we perform this kind of date calculation, we actually end up with a `timedelta` object.

Click here to view code image

```
print(ebola.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 23 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Date                122 non-null
datetime64[ns]
 1   Day                 122 non-null    int64
 2   Cases_Guinea        93 non-null     float64
 3   Cases_Liberia       83 non-null     float64
 4   Cases_SierraLeone   87 non-null     float64
 5   Cases_Nigeria       38 non-null     float64
 6   Cases_Senegal       25 non-null     float64
 7   Cases_UnitedStates  18 non-null     float64
 8   Cases_Spain         16 non-null     float64
 9   Cases_Mali          12 non-null     float64
 10  Deaths_Guinea       92 non-null     float64
 11  Deaths_Liberia      81 non-null     float64
```

```
 12 Deaths_SierraLeone  87 non-null     float64
 13 Deaths_Nigeria      38 non-null     float64
 14 Deaths_Senegal      22 non-null     float64
 15 Deaths_UnitedStates 18 non-null     float64
 16 Deaths_Spain        16 non-null     float64
 17 Deaths_Mali         12 non-null     float64
 18 date_dt             122 non-null
datetime64[ns]
 19 year                122 non-null    int64
 20 month               122 non-null    int64
 21 day                 122 non-null    int64
 22 outbreak_d          122 non-null
timedelta64[ns]
dtypes: datetime64[ns](2), float64(16), int64(4),
timedelta64[ns](1)
memory usage: 22.0 KB
None
```

We get `timedelta` objects as results when we perform calculations with `datetime` objects.

12.6 Datetime Methods

Let's look at another data set. This one deals with bank failures.

Click here to view code image

```
banks = pd.read_csv('data/banklist.csv')
print(banks.head())
```

```
                                             Bank
Name  \
0                                  Fayette County
Bank
```

```
1    Guaranty Bank, (d/b/a BestBank in Georgia &
Mi...
2                                        First NBC
Bank
3                                        Proficio
Bank
4                          Seaway Bank and Trust
Company

                   City     ST   CERT  \
0           Saint Elmo     IL   1802
1            Milwaukee     WI  30003
2          New Orleans     LA  58302
3    Cottonwood Heights    UT  35495
4              Chicago     IL  19328

                   Acquiring Institution
Closing Date Updated Date
0             United Fidelity  Bank, fsb
26-May-17    26-Jul-17
1    First-Citizens Bank & Trust Company
5-May-17    26-Jul-17
2                           Whitney Bank
28-Apr-17    26-Jul-17
3                      Cache Valley Bank
3-Mar-17    18-May-17
4                    State Bank of Texas
27-Jan-17    18-May-17
```

Again, we can import our data with the dates directly parsed.

Click here to view code image

```
banks = pd.read_csv(
  "data/banklist.csv", parse_dates=["Closing
Date", "Updated Date"]
)

print(banks.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 553 entries, 0 to 552
Data columns (total 7 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Bank Name              553 non-null    object
 1   City                   553 non-null    object
 2   ST                     553 non-null    object
 3   CERT                   553 non-null    int64
 4   Acquiring Institution  553 non-null    object
 5   Closing Date           553 non-null
datetime64[ns]
 6   Updated Date           553 non-null
datetime64[ns]
dtypes: datetime64[ns](2), int64(1), object(4)
memory usage: 30.4+ KB
None
```

We can parse out the date by obtaining the quarter and year in which the bank closed.

Click here to view code image

```
banks = banks.assign(
  closing_quarter=banks['Closing
Date'].dt.quarter,
```

```
  closing_year=banks['Closing Date'].dt.year
)
```

```
closing_year =
banks.groupby(['closing_year']).size()
```

Alternatively, we can calculate how many banks closed in each quarter of each year.

Click here to view code image

```
closing_year_q = (
  banks
  .groupby(['closing_year', 'closing_quarter'])
  .size()
)
```

We can then plot these results as shown in Figure 12.1 and Figure 12.2.

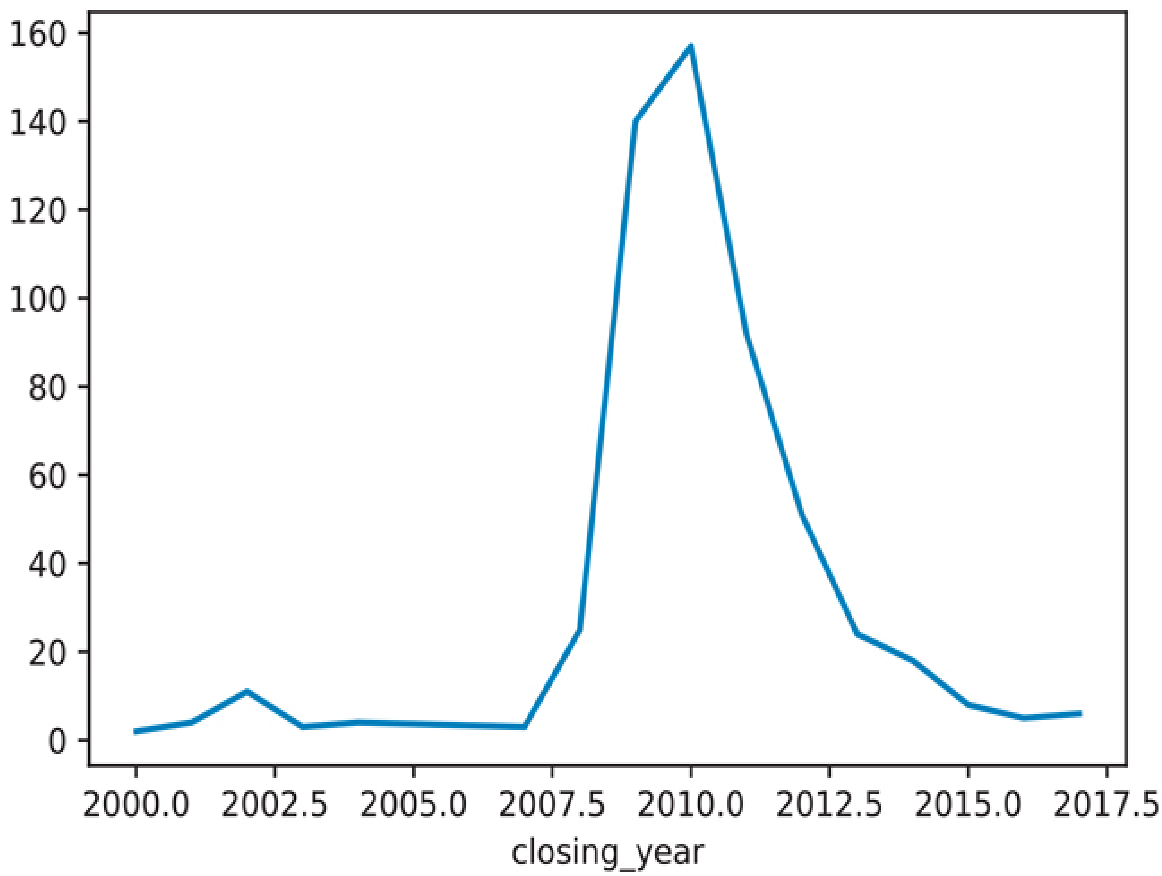


Figure 12.1 Number of banks closing each year

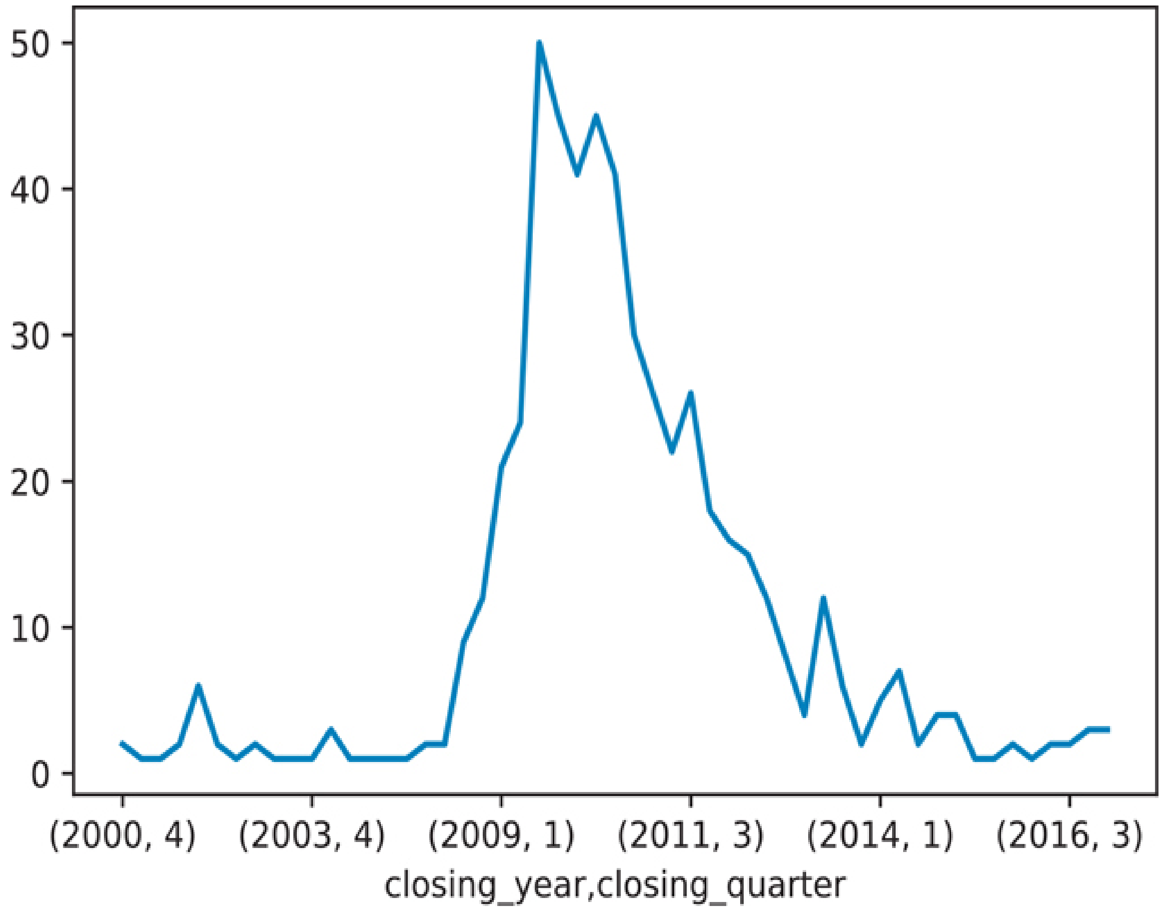


Figure 12.2 Number of banks closing each year by quarter

Click here to view code image

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax = closing_year.plot()
plt.show()
```

```
fig, ax = plt.subplots()
ax = closing_year_q.plot()
plt.show()
```

12.7 Getting Stock Data

One commonly encountered type of data that contains dates is stock prices. Luckily Python has a way of getting this type of data programmatically with the `pandas-datareader` library.[4]

4. `pandas-datareader` library: https://pandas-datareader.readthedocs.io/

Click here to view code image

```
# we can install and use the pandas_datafreader
# to get data from the Internet
import pandas_datareader.data as web

# in this example we are getting stock
information about Tesla
tesla = web.DataReader('TSLA', 'yahoo')

print(tesla)
```

```
      Date        High         Low        Open
Close \
2017-09-05 23.699333 23.059334 23.586666
23.306000
2017-09-06 23.398666 22.770666 23.299999
22.968666
2017-09-07 23.498667 22.896667 23.065332
23.374001
2017-09-08 23.318666 22.820000 23.266001
22.893333
2017-09-11 24.247334 23.333332 23.423332
24.246000
...         ...         ...         ...
...
```

```
2022-08-25 302.959991 291.600006 302.359985
296.070007
2022-08-26 302.000000 287.470001 297.429993
288.089996
2022-08-29 287.739990 280.700012 282.829987
284.820007
2022-08-30 288.480011 272.649994 287.869995
277.700012
2022-08-31 281.250000 271.809998 280.619995
275.609985

      Date      Volume   Adj Close
2017-09-05  57526500.0   23.306000
2017-09-06  61371000.0   22.968666
2017-09-07  63588000.0   23.374001
2017-09-08  48952500.0   22.893333
2017-09-11 115006500.0   24.246000
       ...         ...         ...
2022-08-25  53230000.0  296.070007
2022-08-26  56905800.0  288.089996
2022-08-29  41864700.0  284.820007
2022-08-30  50541800.0  277.700012
2022-08-31  51788900.0  275.609985

[1257 rows x 6 columns]
```

```
# the stock data was saved
# so we do not need to rely on the Internet
again
# instead we can load the same data set as a
file
tesla = pd.read_csv(
```

```
    'data/tesla_stock_yahoo.csv', parse_dates=
 ["Date"]
 )

print(tesla)
```

```
           Date       Open       High         Low
Close \
0    2010-06-29  19.000000  25.000000   17.540001
23.889999
1    2010-06-30  25.790001  30.420000   23.299999
23.830000
2    2010-07-01  25.000000  25.920000   20.270000
21.959999
3    2010-07-02  23.000000  23.100000   18.709999
19.200001
4    2010-07-06  20.000000  20.000000   15.830000
16.110001
...         ...        ...        ...         ...
...
1786 2017-08-02 318.940002 327.119995  311.220001
325.890015
1787 2017-08-03 345.329987 350.000000  343.149994
347.089996
1788 2017-08-04 347.000000 357.269989  343.299988
356.910004
1789 2017-08-07 357.350006 359.480011  352.750000
355.170013
1790 2017-08-08 357.529999 368.579987  357.399994
365.220001
```

```
     Adj Close    Volume
0    23.889999  18766300
1    23.830000  17187100
2    21.959999   8218800
3    19.200001   5139800
4    16.110001   6866900
...        ...       ...
1786 325.890015 13091500
1787 347.089996 13535000
1788 356.910004  9198400
1789 355.170013  6276900
1790 365.220001  7449837

[1791 rows x 7 columns]
```

12.8 Subsetting Data Based on Dates

Since we now know how to extract parts of a date out of a column (Section 12.4), we can incorporate these methods to subset our data without having to parse out the individual components manually.

For example, if we want only data for June 2010 from our stock price data set, we can use boolean subsetting.

Click here to view code image

```
print(
  tesla.loc[
    (tesla.Date.dt.year == 2010) &
(tesla.Date.dt.month == 6)
  ]
)
```

```
        Date       Open    High        Low
Close Adj Close \
```

```
0 2010-06-29  19.000000  25.00  17.540001
23.889999 23.889999
1 2010-06-30  25.790001  30.42  23.299999
23.830000 23.830000

    Volume
0 18766300
1 17187100
```

12.8.1 The DatetimeIndex Object

When we are working with `datetime` data, we often need to set the `datetime` object to be the dataframe's index. To this point, we've mainly left the dataframe row index to be the row number. We have also seen some side effects that arise because the row index may not always be the row number, such as when we were concatenating dataframes in Chapter 6.

First, let's assign the `Date` column as the index.

Click here to view code image

```
tesla.index = tesla['Date']
print(tesla.index)
```

```
DatetimeIndex(['2010-06-29', '2010-06-30', '2010-
07-01',
               '2010-07-02', '2010-07-06', '2010-
07-07',
               '2010-07-08', '2010-07-09', '2010-
07-12',
               '2010-07-13',
               ...
               '2017-07-26', '2017-07-27', '2017-
07-28',
```

```
               '2017-07-31', '2017-08-01', '2017-
08-02',
               '2017-08-03', '2017-08-04', '2017-
08-07',
               '2017-08-08'],
              dtype='datetime64[ns]', name='Date',
length=1791, freq=None)
```

With the index set as a date object, we can now use the date directly to subset rows. For example, we can subset our data based on the year.

Click here to view code image

```
print(tesla['2015'])
```

```
                 Date       Open       High
Low \
Date
2015-01-02 2015-01-02 222.869995 223.250000
213.259995
2015-01-05 2015-01-05 214.550003 216.500000
207.160004
2015-01-06 2015-01-06 210.059998 214.199997
204.210007
2015-01-07 2015-01-07 213.350006 214.779999
209.779999
2015-01-08 2015-01-08 212.809998 213.800003
210.009995
       ...        ...        ...        ...
...
2015-12-24 2015-12-24 230.559998 231.880005
228.279999
2015-12-28 2015-12-28 231.490005 231.979996
225.539993
```

```
2015-12-29 2015-12-29 230.059998 237.720001
229.550003
2015-12-30 2015-12-30 236.600006 243.630005
235.669998
2015-12-31 2015-12-31 238.509995 243.449997
238.369995

                Close  Adj Close  Volume
Date
2015-01-02 219.309998 219.309998 4764400
2015-01-05 210.089996 210.089996 5368500
2015-01-06 211.279999 211.279999 6261900
2015-01-07 210.949997 210.949997 2968400
2015-01-08 210.619995 210.619995 3442500
...               ...        ...     ...
2015-12-24 230.570007 230.570007  708000
2015-12-28 228.949997 228.949997 1901300
2015-12-29 237.190002 237.190002 2406300
2015-12-30 238.089996 238.089996 3697900
2015-12-31 240.009995 240.009995 2683200

[252 rows x 7 columns]
```

```
print(tesla.loc['2015'])
```

Alternatively, we can subset the data based on the year and month.

Click here to view code image

```
print(tesla['2010-06'])
```

```
                  Date        Open  High        Low
Close \
Date
2010-06-29 2010-06-29  19.000000  25.00  17.540001
23.889999
2010-06-30 2010-06-30  25.790001  30.42  23.299999
23.830000

           Adj Close   Volume
Date
2010-06-29 23.889999 18766300
2010-06-30 23.830000 17187100
```

```
print(tesla.loc['2010-06'])
```

12.8.2 The TimedeltaIndex Object

Just as we set the index of a dataframe to a `datetime` to create a `DatetimeIndex`, so we can do the same thing with a `timedelta` to create a `TimedeltaIndex`.

Let's create a `timedelta`.

Click here to view code image

```
tesla['ref_date'] = tesla['Date'] -
tesla['Date'].min()
```

Now we can assign the `timedelta` to the index.

Click here to view code image

```
tesla.index = tesla['ref_date']
```

```
print(tesla)
```

```
                  Date      Open        High
Low \
ref_date
0 days    2010-06-29 19.000000   25.000000
17.540001
1 days    2010-06-30 25.790001   30.420000
23.299999
2 days    2010-07-01 25.000000   25.920000
20.270000
3 days    2010-07-02 23.000000   23.100000
18.709999
7 days    2010-07-06 20.000000   20.000000
15.830000
...              ...        ...         ...
...
2591 days  2017-08-02 318.940002 327.119995
311.220001
2592 days  2017-08-03 345.329987 350.000000
343.149994
2593 days  2017-08-04 347.000000 357.269989
343.299988
2596 days  2017-08-07 357.350006 359.480011
352.750000
2597 days  2017-08-08 357.529999 368.579987
357.399994

           Close Adj Close Volume   ref_date
ref_date
0 days     23.889999  23.889999  18766300    0
days
1 days     23.830000  23.830000  17187100    1
days
2 days     21.959999  21.959999   8218800    2
```

```
days
3 days     19.200001  19.200001   5139800    3
days
7 days     16.110001  16.110001   6866900    7
days
...              ...        ...        ...        ...
2591 days 325.890015 325.890015  13091500 2591
days
2592 days 347.089996 347.089996  13535000 2592
days
2593 days 356.910004 356.910004   9198400 2593
days
2596 days 355.170013 355.170013   6276900 2596
days
2597 days 365.220001 365.220001   7449837 2597
days

[1791 rows x 8 columns]
```

We can now select our data based on these deltas.

Click here to view code image

```
print(tesla['0 day': '10 day'])
```

```
              Date       Open       High        Low
Close \
ref_date
0 days  2010-06-29 19.000000 25.000000 17.540001
23.889999
1 days  2010-06-30 25.790001 30.420000 23.299999
23.830000
2 days  2010-07-01 25.000000 25.920000 20.270000
```

```
21.959999
3 days  2010-07-02 23.000000 23.100000 18.709999
19.200001
7 days  2010-07-06 20.000000 20.000000 15.830000
16.110001
8 days  2010-07-07 16.400000 16.629999 14.980000
15.800000
9 days  2010-07-08 16.139999 17.520000 15.570000
17.459999
10 days 2010-07-09 17.580000 17.900000 16.549999
17.400000

          Adj Close   Volume ref_date
ref_date
0 days    23.889999 18766300    0 days
1 days    23.830000 17187100    1 days
2 days    21.959999  8218800    2 days
3 days    19.200001  5139800    3 days
7 days    16.110001  6866900    7 days
8 days    15.800000  6921700    8 days
9 days    17.459999  7711400    9 days
10 days   17.400000  4050600   10 days
```

12.9 Date Ranges

Not every data set will have a fixed frequency of values. For example, in our Ebola data set, we do not have an observation for every day in the date range.

Click here to view code image

```
ebola = pd.read_csv(
'data/country_timeseries.csv', parse_dates=
```

```
["Date"]
)
```

Here, `2015-01-01` is missing from the `.head()` of the data.

Click here to view code image

```
print(ebola.iloc[:, :5])
```

```
             Date Day  Cases_Guinea  Cases_Liberia
Cases_SierraLeone
0      2015-01-05 289        2776.0            NaN
10030.0
1      2015-01-04 288        2775.0            NaN
9780.0
2      2015-01-03 287        2769.0         8166.0
9722.0
3      2015-01-02 286           NaN         8157.0
NaN
4      2014-12-31 284        2730.0         8115.0
9633.0
..            ... ...           ...            ...
...
117    2014-03-27   5         103.0            8.0
6.0
118    2014-03-26   4          86.0            NaN
NaN
119    2014-03-25   3          86.0            NaN
NaN
120    2014-03-24   2          86.0            NaN
NaN
121    2014-03-22   0          49.0            NaN
NaN
```

```
[122 rows x 5 columns]
```

It's common practice to create a date range to `.reindex()` a data set. We can use the `date_range()`

Click here to view code image

```
head_range = pd.date_range(start='2014-12-31',
end='2015-01-05')
print(head_range)
```

```
DatetimeIndex(['2014-12-31', '2015-01-01', '2015-
01-02',
               '2015-01-03', '2015-01-04', '2015-
01-05'],
             dtype='datetime64[ns]', freq='D')
```

We'll just work with the first five rows in this example.

```
ebola_5 = ebola.head()
```

If we want to set this date range as the index, we need to first set the date as the index.

Click here to view code image

```
ebola_5.index = ebola_5['Date']
```

Next we can `.reindex()` our data.

Click here to view code image

```
ebola_5 = ebola_5.reindex(head_range)

print(ebola_5.iloc[:, :5])
```

```
                 Date            Day  Cases_Guinea
Cases_Liberia \
2014-12-31 2014-12-31        284.0         2730.0
8115.0
2015-01-01        NaT          NaN            NaN
NaN
2015-01-02 2015-01-02        286.0            NaN
8157.0
2015-01-03 2015-01-03        287.0         2769.0
8166.0
2015-01-04 2015-01-04        288.0         2775.0
NaN
2015-01-05 2015-01-05        289.0         2776.0
NaN


      Cases_SierraLeone
2014-12-31       9633.0
2015-01-01          NaN
2015-01-02          NaN
2015-01-03       9722.0
2015-01-04       9780.0
2015-01-05      10030.0
```

12.9.1 Frequencies

When we created our `head_range`, the print statement included a parameter called `freq`. In that example, `freq` was `'D'` for “day.” That is, the values in our date range were stepped through using a day-by-day increment. The possible frequencies are reproduced from the Pandas timeseries documentation that is listed in Table 12.2.[5]

5. Frequency offset aliases:
`https://pandas.pydata.org/docs/user_guide/timeseri`

es.html#offset-aliases

Table 12.2 **Possible Frequencies**

Alias	Description
B	Business day frequency
C	Custom business day frequency (experimental)
D	Calendar day frequency
W	Weekly frequency
M	Month end frequency
SM	Semi-month end frequency (15th and end of month)
BM	Business month end frequency
CBM	Custom business month end frequency
MS	Month start frequency
SMS	Semi-month start frequency (1st and 15th)
BMS	Business month start frequency
CBMS	Custom business month start frequency
Q	Quarter end frequency
BQ	Business quarter end frequency
QS	Quarter start frequency
BQS	Business quarter start frequency
A	Year end frequency
BA	Business year end frequency
AS	Year start frequency
BAS	Business year start frequency
BH	Business hour frequency
H	Hour frequency

Alias	Description
T	Minute frequency
S	Second frequency
L	Millisecond frequency
U	Microsecond frequency
N	Nanosecond frequency

These values can be passed into the `freq` parameter when calling `date_range`. For example, January 2, 2022, was a Sunday, and we can create a range consisting of the business days in that week.

Click here to view code image

```
# business days during the week of Jan 1, 2022
print(pd.date_range('2022-01-01', '2022-01-07',
freq='B'))
```

```
DatetimeIndex(['2022-01-03', '2022-01-04', '2022-
01-05',
               '2022-01-06', '2022-01-07'],
              dtype='datetime64[ns]', freq='B')
```

12.9.2 Offsets

Offsets are variations on a base frequency. For example, we can take the business days range that we just created and add an offset such that instead of *every* business day, data are included for *every other* business day.

Click here to view code image

```
# every other business day during the week of
Jan 1, 2022
```

```
print(pd.date_range('2022-01-01', '2017-01-07',
freq='2B'))
```

```
DatetimeIndex([], dtype='datetime64[ns]',
freq='2B')
```

We created this offset by putting a multiplying value before the base frequency. This kind of offset can be combined with other base frequencies as well. For example, we can specify the first Thursday of each month in the year 2022.

Click here to view code image

```
print(pd.date_range('2022-01-01', '2022-12-31',
freq='WOM-1THU'))
```

```
DatetimeIndex(['2022-01-06', '2022-02-03', '2022-
03-03',
               '2022-04-07', '2022-05-05', '2022-
06-02',
               '2022-07-07', '2022-08-04', '2022-
09-01',
               '2022-10-06', '2022-11-03', '2022-
12-01'],
             dtype='datetime64[ns]', freq='WOM-
1THU')
```

We can also specify the third Friday of each month.

Click here to view code image

```
print(pd.date_range('2022-01-01', '2022-12-31',
freq='WOM-3FRI'))
```

```
DatetimeIndex(['2022-01-21', '2022-02-18', '2022-
03-18',
               '2022-04-15', '2022-05-20', '2022-
06-17',
               '2022-07-15', '2022-08-19', '2022-
09-16',
               '2022-10-21', '2022-11-18', '2022-
12-16'],
             dtype='datetime64[ns]', freq='WOM-
3FRI')
```

12.10 Shifting Values

There are a few reasons why you might want to shift your dates by a certain value. For example, you might need to correct some kind of measurement error in your data. Alternatively, you might want to standardize the start dates for your data so you can compare trends.

Even though our Ebola data isn't "tidy," one of the benefits of the data in its current format is that it allows us to plot the outbreak. This plot is shown in Figure 12.3.

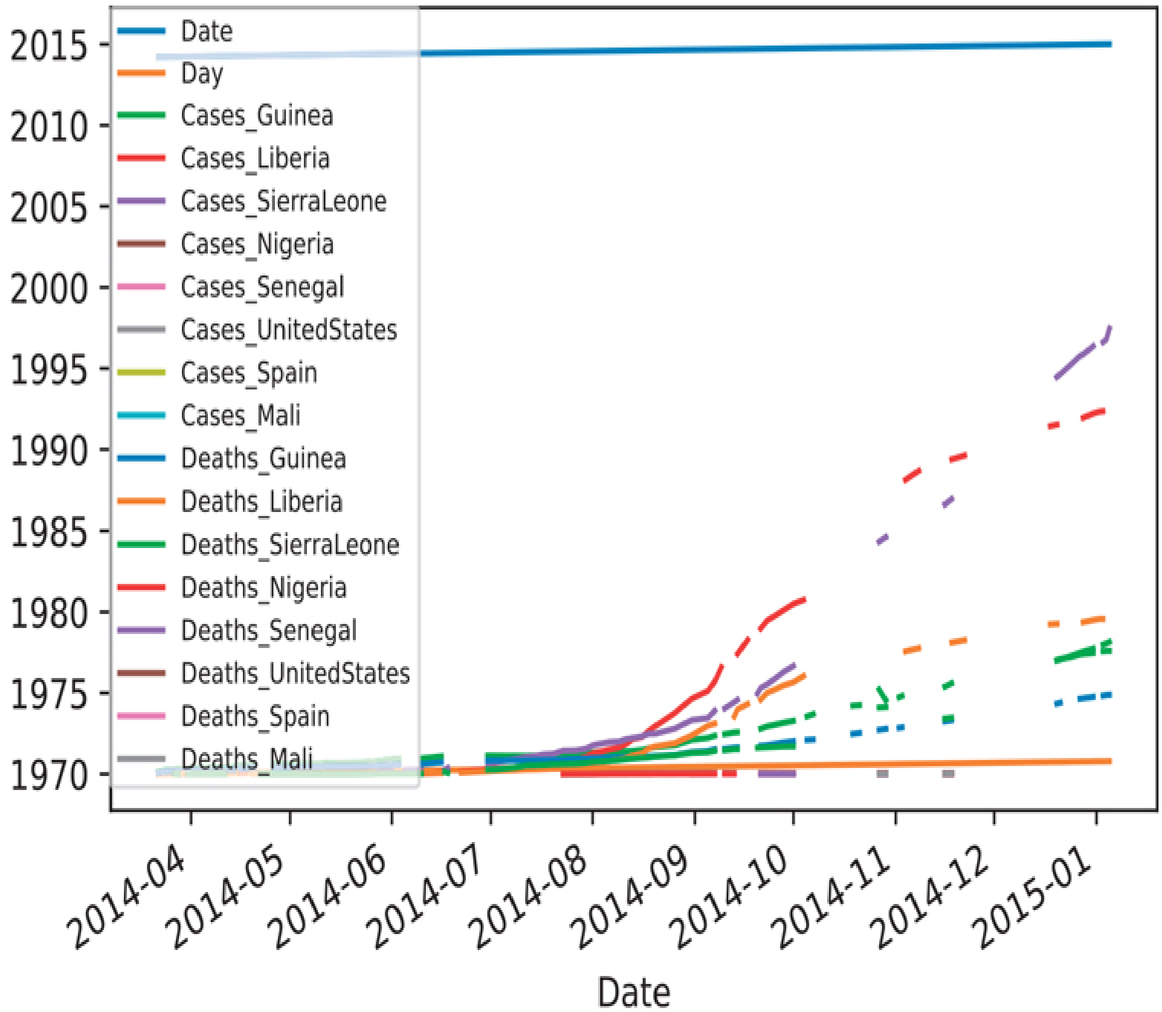


Figure 12.3 Ebola plot of cases and deaths (unshifted dates)

Click here to view code image

```
import matplotlib.pyplot as plt

ebola.index = ebola['Date']
```

```
fig, ax = plt.subplots()
ax = ebola.plot(ax=ax)
ax.legend(fontsize=7, loc=2, borderaxespad=0.0)
plt.show()
```

When we're looking at an outbreak, one useful piece of information is how fast an outbreak is spreading relative to other countries. Let's look at just a few columns from our Ebola data set.

Click here to view code image

```
ebola_sub = ebola[['Day', 'Cases_Guinea',
'Cases_Liberia']]
print(ebola_sub.tail(10))
```

```
           Day Cases_Guinea  Cases_Liberia
Date
2014-04-04 13         143.0           18.0
2014-04-01 10         127.0            8.0
2014-03-31  9         122.0            8.0
2014-03-29  7         112.0            7.0
2014-03-28  6         112.0            3.0
2014-03-27  5         103.0            8.0
2014-03-26  4          86.0            NaN
2014-03-25  3          86.0            NaN
2014-03-24  2          86.0            NaN
2014-03-22  0          49.0            NaN
```

You can see that each country's starting date is different, which makes it difficult to compare the actual slopes between countries when a new outbreak occurs later in time.

In this example, we want all our dates to start from a common `0` day. There are multiple steps to this process.

- Since not every date is listed, we need to create a date range of all the dates in our data set.
- We need to calculate the difference between the earliest date in our data set, and the earliest valid (non `NaN`) date in each column.
- We can then shift each of the columns by this calculated value.

Before we begin, let's start with a fresh copy of the Ebola data set. We'll parse the `Date` column as a proper `date` object, and assign this date to the `.index`. In this example, we are parsing the date and setting it as the index directly.

Click here to view code image

```
ebola = pd.read_csv(
  "data/country_timeseries.csv",
  index_col="Date",
  parse_dates=["Date"],
)

print(ebola.iloc[:, :4])
```

```
           Day Cases_Guinea Cases_Liberia
Cases_SierraLeone
Date
2015-01-05 289       2776.0            NaN
10030.0
2015-01-04 288       2775.0            NaN
9780.0
2015-01-03 287       2769.0         8166.0
9722.0
2015-01-02 286          NaN         8157.0
NaN
2014-12-31 284       2730.0         8115.0
9633.0
...        ...          ...            ...
...
2014-03-27   5        103.0            8.0
6.0
2014-03-26   4         86.0            NaN
NaN
```

```
2014-03-25    3          86.0           NaN
NaN
2014-03-24    2          86.0           NaN
NaN
2014-03-22    0          49.0           NaN
NaN

[122 rows x 4 columns]
```

First, we need to create the date range to fill in all the missing dates in our data. Then, when we shift our date values downward, the number of days that the data will shift will be the same as the number of rows that will be shifted.

Click here to view code image

```
new_idx = pd.date_range(ebola.index.min(),
ebola.index.max())

print(new_idx)
```

```
DatetimeIndex(['2014-03-22', '2014-03-23', '2014-
03-24',
              '2014-03-25', '2014-03-26', '2014-
03-27',
              '2014-03-28', '2014-03-29', '2014-
03-30',
              '2014-03-31',

              ...
              '2014-12-27', '2014-12-28', '2014-
12-29',
              '2014-12-30', '2014-12-31', '2015-
01-01',
```

```
                 '2015-01-02', '2015-01-03', '2015-
01-04',
                 '2015-01-05'],
                dtype='datetime64[ns]', length=290,
freq='D')
```

Looking at our `new_idx`, we see that the dates are not in the order that we want. To fix this, we can reverse the order of the index.

Click here to view code image

```
new_idx = reversed(new_idx)
print(new_idx)
```

```
<reversed object at 0x105aedfc0>
```

Now we can properly `.reindex()` our data. This will create rows of `NaN` values if the index does not exist already in our data set.

Click here to view code image

```
ebola = ebola.reindex(new_idx)
```

If we look at the `.head()` and `.tail()` of the resulting data, we see that dates that were originally not listed have been added into the data set, along with a row of `NaN` missing values. Additionally, the `Date` column is filled with the `NaT` value, which is an internal Pandas representation for missing time value (similar to how `NaN` is used for numeric missing values).

Click here to view code image

```
print(ebola.iloc[:, :4])
```

```
                Day  Cases_Guinea  Cases_Liberia
```

```
Cases_SierraLeone
Date
2015-01-05  289.0       2776.0           NaN
10030.0
2015-01-04  288.0       2775.0           NaN
9780.0
2015-01-03  287.0       2769.0        8166.0
9722.0
2015-01-02  286.0          NaN        8157.0
NaN
2015-01-01    NaN          NaN           NaN
NaN
...           ...          ...           ...
...
2014-03-26    4.0         86.0           NaN
NaN
2014-03-25    3.0         86.0           NaN
NaN
2014-03-24    2.0         86.0           NaN
NaN
2014-03-23    NaN          NaN           NaN
NaN
2014-03-22    0.0         49.0           NaN
NaN

[290 rows x 4 columns]
```

Now that we've created our date range and assigned it to the `index`, our next step is to calculate the difference between the earliest date in our data set and the earliest valid (non-missing) date in each column. To perform this calculation, we can use the `Series` method called `.last_valid_index()`, which returns the label (`index`) of the last non-missing or non-null value. An analogous method called `.first_valid_index()` returns the first non-missing or non-null value. Since we want to perform this calculation across all the columns, we can use the `.apply()` method.

Click here to view code image

```
last_valid =
ebola.apply(pd.Series.last_valid_index)
print(last_valid)
```

```
Day                    2014-03-22
Cases_Guinea           2014-03-22
Cases_Liberia          2014-03-27
Cases_SierraLeone      2014-03-27
Cases_Nigeria          2014-07-23
                          ...
Deaths_Nigeria         2014-07-23
Deaths_Senegal         2014-09-07
Deaths_UnitedStates    2014-10-01
Deaths_Spain           2014-10-08
Deaths_Mali            2014-10-22
Length: 17, dtype: datetime64[ns]
```

Next, we want to get the earliest date in our data set.

Click here to view code image

```
earliest_date = ebola.index.min()
print(earliest_date)
```

```
2014-03-22 00:00:00
```

We then subtract this date from each of our `last_valid` dates.

Click here to view code image

```
shift_values = last_valid - earliest_date
print(shift_values)
```

```
Day                       0 days
Cases_Guinea              0 days
Cases_Liberia             5 days
Cases_SierraLeone       5 days
Cases_Nigeria           123 days
                          ...
Deaths_Nigeria          123 days
Deaths_Senegal          169 days
Deaths_UnitedStates     193 days
Deaths_Spain            200 days
Deaths_Mali             214 days
Length: 17, dtype: timedelta64[ns]
```

Finally, we can iterate through each column, using the `.shift()` method to shift the columns down by the corresponding value in `shift_values`. Note that the values in `shift_values` are all positive. If they were negative (if we flipped the order of our subtraction), this operation would shift the values up.

Click here to view code image

```
ebola_dict = {}

for idx, col in enumerate(ebola):
    d = shift_values[idx].days
    shifted = ebola[col].shift(d)
    ebola_dict[col] = shifted

#print(ebola_dict)
```

Since we have a `dict` of values, we can convert it to a dataframe using the Pandas `DataFrame` function.

Click here to view code image

```
ebola_shift = pd.DataFrame(ebola_dict)
```

The last row in each column now has a value; that is, the columns have been shifted down appropriately.

Click here to view code image

```
print(ebola_shift.tail())
```

```
           Day Cases_Guinea Cases_Liberia
Cases_SierraLeone \
Date
2014-03-26 4.0         86.0            8.0
2.0
2014-03-25 3.0         86.0            NaN
NaN
2014-03-24 2.0         86.0            7.0
NaN
2014-03-23 NaN          NaN            3.0
2.0
2014-03-22 0.0         49.0            8.0
6.0



           Cases_Nigeria Cases_Senegal
Cases_UnitedStates \
Date
2014-03-26           1.0           NaN
1.0
2014-03-25           NaN           NaN
```

```
NaN
2014-03-24             NaN            NaN
NaN
2014-03-23             NaN            NaN
NaN
2014-03-22             0.0            1.0
1.0

              Cases_Spain Cases_Mali Deaths_Guinea
Deaths_Liberia \
Date
2014-03-26             1.0        NaN          62.0
4.0
2014-03-25             NaN        NaN          60.0
NaN
2014-03-24             NaN        NaN          59.0
2.0
2014-03-23             NaN        NaN           NaN
3.0
2014-03-22             1.0        1.0          29.0
6.0

            Deaths_SierraLeone Deaths_Nigeria
Deaths_Senegal \
Date
2014-03-26                 2.0            1.0
NaN
2014-03-25                 NaN            NaN
NaN
2014-03-24                 NaN            NaN
NaN
2014-03-23                 2.0            NaN
```

```
NaN
2014-03-22                         5.0              0.0
0.0

            Deaths_UnitedStates Deaths_Spain
Deaths_Mali
Date
2014-03-26                          0.0             1.0
NaN
2014-03-25                          NaN             NaN
NaN
2014-03-24                          NaN             NaN
NaN
2014-03-23                          NaN             NaN
NaN
2014-03-22                          0.0             1.0
1.0
```

Finally, since the indices are no longer valid across each row, we can remove them, and then assign the correct index, which is the `Day`. Note that `Day` no longer represents the first day of the entire outbreak, but rather the first day of an outbreak for the given country.

Click here to view code image

```
ebola_shift.index = ebola_shift['Day']
ebola_shift = ebola_shift.drop(['Day'],
axis="columns")
```

```
print(ebola_shift.tail())
```

```
      Cases_Guinea Cases_Liberia Cases_SierraLeone
Cases_Nigeria \
```

```
Day
4.0           86.0          8.0               2.0
1.0
3.0           86.0          NaN               NaN
NaN
2.0           86.0          7.0               NaN
NaN
NaN            NaN          3.0               2.0
NaN
0.0           49.0          8.0               6.0
0.0

    Cases_Senegal Cases_UnitedStates Cases_Spain
Cases_Mali \
Day
4.0           NaN                1.0         1.0
NaN
3.0           NaN                NaN         NaN
NaN
2.0           NaN                NaN         NaN
NaN
NaN           NaN                NaN         NaN
NaN
0.0           1.0                1.0         1.0
1.0

     Deaths_Guinea Deaths_Liberia
Deaths_SierraLeone \
Day
4.0           62.0            4.0
2.0
3.0           60.0            NaN
```

```
NaN
2.0            59.0            2.0
NaN
NaN             NaN            3.0
2.0
0.0            29.0            6.0
5.0

    Deaths_Nigeria Deaths_Senegal
Deaths_UnitedStates \
Day
4.0             1.0            NaN
0.0
3.0             NaN            NaN
NaN
2.0             NaN            NaN
NaN
NaN             NaN            NaN
NaN
0.0             0.0            0.0
0.0

   Deaths_Spain Deaths_Mali
Day
4.0          1.0         NaN
3.0          NaN         NaN
2.0          NaN         NaN
NaN          NaN         NaN
0.0          1.0         1.0
```

12.11 Resampling

Resampling converts a `datetime` from one frequency to another frequency. Three types of resampling can occur:

- Downsampling: from a higher frequency to a lower frequency (e.g., daily to monthly)
- Upsampling: from a lower frequency to a higher frequency (e.g., monthly to daily)
- No change: frequency does not change (e.g., every first Thursday of the month to the last Friday of the month)

The values we can pass into `.resample()` are listed in Table 12.2.

Click here to view code image

```
# downsample daily values to monthly values
# since we have multiple values, we need to
aggregate the results
# here we will use the mean
down = ebola.resample('M').mean()
print(down.iloc[:,  :5])
```

```
                  Day Cases_Guinea   Cases_Liberia
\
Date
2014-03-31   4.500000    94.500000        6.500000
2014-04-30  24.333333   177.818182       24.555556
2014-05-31  51.888889   248.777778       12.555556
2014-06-30  84.636364   373.428571       35.500000
2014-07-31 115.700000   423.000000      212.300000
...               ...          ...             ...
2014-09-30 177.500000   967.888889     2815.625000
2014-10-31 207.470588  1500.444444     4758.750000
2014-11-30 237.214286  1950.500000     7039.000000
2014-12-31 271.181818  2579.625000     7902.571429
2015-01-31 287.500000  2773.333333     8161.500000
```

```
     Cases_SierraLeone  Cases_Nigeria
Date
2014-03-31    3.333333            NaN
2014-04-30    2.200000            NaN
2014-05-31    7.333333            NaN
2014-06-30  125.571429            NaN
2014-07-31  420.500000       1.333333
...                ...            ...
2014-09-30 1726.000000      20.714286
2014-10-31 3668.111111      20.000000
2014-11-30 5843.625000      20.000000
2014-12-31 8985.875000      20.000000
2015-01-31 9844.000000            NaN

[11 rows x 5 columns]
```

```
# here we will upsample our downsampled value
# notice how missing dates are populated,
# but they are filled in with missing values
up = down.resample('D').mean()
print(up.iloc[:, :5])
```

```
             Day Cases_Guinea Cases_Liberia
Cases_SierraLeone \
Date
2014-03-31   4.5    94.500000           6.5
3.333333
2014-04-01   NaN          NaN           NaN
NaN
2014-04-02   NaN          NaN           NaN
NaN
2014-04-03   NaN          NaN           NaN
NaN
```

```
2014-04-04   NaN          NaN            NaN
NaN
...          ...          ...            ...
...
2015-01-27   NaN          NaN            NaN
NaN
2015-01-28   NaN          NaN            NaN
NaN
2015-01-29   NaN          NaN            NaN
NaN
2015-01-30   NaN          NaN            NaN
NaN
2015-01-31 287.5  2773.333333         8161.5
9844.000000

     Cases_Nigeria
Date
2014-03-31     NaN
2014-04-01     NaN
2014-04-02     NaN
2014-04-03     NaN
2014-04-04     NaN
...            ...
2015-01-27     NaN
2015-01-28     NaN
2015-01-29     NaN
2015-01-30     NaN
2015-01-31     NaN

[307 rows x 5 columns]
```

12.12 Time Zones

Don't try to write your own time zone converter. As Tom Scott explains in a "Computerphile" video, "That way lies madness."[6] There are many things you probably did not even think to consider when working with different time zones. For example, not every country implements daylight savings time, and even those that do, may not necessarily change the clocks on the same day of the year. And don't forget about leap years and **leap seconds**! Luckily Python has a library specifically designed to work with time zones[7], Pandas also wraps this library when working with time zones.

6. The problem with time and time zones: Computerphile: www.youtube.com/watch?v=-5wpm-gesOY

7. Documentation for pytz:a https://pythonhosted.org/pytz/

```
import pytz
```

There are many time zones available in the library.

Click here to view code image

```
print(len(pytz.all_timezones))
```

```
594
```

Here are the U.S. time zones:

Click here to view code image

```
import re
regex = re.compile(r'^US')
selected_files = filter(regex.search,
pytz.common_timezones)
print(list(selected_files))
```

```
['US/Alaska', 'US/Arizona', 'US/Central',
'US/Eastern', 'US/Hawaii',
' US/Mountain', 'US/Pacific']
```

The easiest way to interact with time zones in Pandas is to use the string names given in `pytz.all_timezones()`.

One way to illustrate time zones is to create two timestamps using the Pandas `Timestamp` function. For example, if there was a flight between the JFK and LAX airports that departed at 7:00 AM from New York and landed at 9:57 AM in Los Angeles. We can encode these times with the proper time zone.

Click here to view code image

```
# 7AM Eastern
depart = pd.Timestamp('2017-08-29 07:00',
tz='US/Eastern')
print(depart)
```

```
2017-08-29 07:00:00-04:00
```

```
arrive = pd.Timestamp('2017-08-29 09:57')
print(arrive)
```

```
2017-08-29 09:57:00
```

Another way we can encode a time zone is by using the `.tz_localize()` method on an “empty” timestamp.

Click here to view code image

```
arrive = arrive.tz_localize('US/Pacific')
print(arrive)
```

```
2017-08-29 09:57:00-07:00
```

We can convert the arrival time back to the Eastern time zone to see what the time would be on the East Coast when the flight arrives.

Click here to view code image

```
print(arrive.tz_convert('US/Eastern'))
```

```
2017-08-29 12:57:00-04:00
```

We can also perform operations on time zones. Here we look at the difference between the times to get the flight duration.

```
duration = arrive - depart
print(duration)
```

```
0 days 05:57:00
```

12.13 Arrow for Better Dates and Times

If you do end up working with date and time columns often, I would suggest looking into the `arrow` library. You can find the documentation page here: `https://arrow.readthedocs.io/en/latest/` Do not confuse this Arrow library with the Apache Arrow project for language-independent dataframe formats.

Arrow is a separate library that needs to be installed, but works slightly different from the methods shown in this chapter. However, it does do a better job handling time zones. See this post by Paul Ganssle for more information about the benefits of `arrow` over `pytz`: `https://blog.ganssle.io/articles/2018/03/pytz-fastest-footgun.html`

Conclusion

Pandas provides a series of convenient methods and functions when we are working with dates and times because these types of data are used so often with time-series data. A common example of time-series data is stock prices, but other examples include observational and simulated data. These convenient Pandas functions and methods allow you to easily work with date objects without having to resort to string manipulation and parsing.

Part IV

Data Modeling

This part of the book follows the methods described in Jared Lander's *R for Everyone*. The rationale is that since you have learned the methods of data manipulation in Python using Pandas, you can save out the cleaned data set if you need to use a method from another analytics language.

This part covers many of the basic modeling techniques and serves as an introduction to data analytics and machine learning. Other great references are:

- Andreas Müller and Sarah Guido's *Introduction to Machine Learning with Python*
- Sebastian Raschka and Vahid Mirjalili's *Python Machine Learning*
- Mark Fenner's *Machine Learning with Python for Everyone*
- Andrew Kelleher and Adam Kelleher's *Machine Learning in Production: Developing and Optimizing Data Science Workflows and Applications*

Many of the techniques covered so far in the book apply to figuring out what kind of information is stored in our columns, in particular, the variable we are trying to model or predict. If our data has an outcome variable, we can use supervised modeling techniques. If our variable of interest is continuous, we would use a linear regression model (Chapter 13). If our outcome variable is binary we would use a logistic regression model, if it is count data, we would use a Poisson model (Chapter 14). Survival models are used when we are looking for an outcome of interest, but also have censoring (Chapter 15). When we are fitting models for prediction, we sometimes need to find a way to pick the "best" model, this is when we have to compare model diagnostics (Chapter 16).

If we are solely interested in prediction, and not inference, we can employ regularization techniques to make our model more numerically stable (Chapter 17). If we do not have an outcome variable we can test our model against, we would use some kind of unsupervised modeling technique, such as clustering (Chapter 18).

13

Linear Regression (Continuous Outcome Variable)

13.1 Simple Linear Regression

The goal of linear regression is to draw a straight-line relationship between a response variable (also known as an outcome or dependent variable) and a predictor variable (also known as a feature, covariate, or independent variable).

Let's take another look at our `tips` data set.

Click here to view code image

```
import pandas as pd
import seaborn as sns

tips = sns.load_dataset('tips')
print(tips)
```

```
    total_bill   tip     sex  smoker    day    time
size
0        16.99  1.01  Female      No    Sun  Dinner
2
1        10.34  1.66    Male      No    Sun  Dinner
3
2        21.01  3.50    Male      No    Sun  Dinner
3
3        23.68  3.31    Male      No    Sun  Dinner
```

```
2
4          24.59 3.61  Female     No   Sun  Dinner
4
..           ...  ...      ...    ...   ...     ...
...
239        29.03 5.92    Male     No   Sat  Dinner
3
240        27.18 2.00  Female    Yes   Sat  Dinner
2
241        22.67 2.00    Male    Yes   Sat  Dinner
2
242        17.82 1.75    Male     No   Sat  Dinner
2
243        18.78 3.00  Female     No  Thur  Dinner
2

[244 rows x 7 columns]
```

In our simple linear regression, we'd like to see how the `total_bill` relates to or predicts the `tip`.

13.1.1 With `statsmodels`

We can use the `statsmodels` library to perform our simple linear regression. We will use the formula API (application programming interface) from `statsmodels`. This is a new library we are working with.

Click here to view code image

```
import statsmodels.formula.api as smf
```

To perform this simple linear regression, we use the `ols()` function, which computes the ordinary least squares value; it is one method to estimate parameters in a linear regression. Recall that the formula for a line is $y = mx + b$, where `y` is our response variable, `x` is our predictor, `b` is the intercept, and `m` is the slope, the parameter we are estimating.

The formula notation has two parts, separated by a tilde, ~. To the left of the tilde is the response variable, and to the right of the tilde are the predictor(s).

Click here to view code image

```
model = smf.ols(formula='tip ~ total_bill',
data=tips)
```

Once we have specified our model, we can fit the data to the model by using the `fit` method.

Click here to view code image

```
results = model.fit()
```

To look at our results, we can call the `.summary()` method on the `results`.

Click here to view code image

```
print(results.summary())
```

```
                     OLS Regression Results
==================================================
========================
Dep. Variable:                   tip   R-squared:
0.457
Model:                           OLS   Adj. R-
squared:                 0.454
Method:                Least Squares   F-
statistic:                    203.4
Date:               Thu, 01 Sep 2022   Prob (F-
statistic):          6.69e-34
```

```
Time:                            01:55:45   Log-
Likelihood:                    -350.54
No. Observations:                     244   AIC:
705.1
Df Residuals:                         242   BIC:
712.1
Df Model:                               1
Covariance Type:              nonrobust
==================================================
========================
                  coef    std err          t
P>|t|      [0.025      0.975]
--------------------------------------------------
------------------------
Intercept        0.9203      0.160      5.761
0.000       0.606       1.235
total_bill       0.1050      0.007     14.260
0.000       0.091       0.120
==================================================
========================
Omnibus:                           20.185   Durbin-
Watson:                      2.151
Prob(Omnibus):                      0.000   Jarque-
Bera (JB):                  37.750
Skew:                               0.443   Prob(JB):
6.35e-09
Kurtosis:                           4.711   Cond. No.
53.0
==================================================
========================

Notes:
```

```
[1] Standard Errors assume that the covariance
matrix of the errors is correctly specified.
```

Here we can see the `Intercept` of the model and the `total_bill`. We can use these parameters in our formula for the line, $y = (0.105)x + 0.920$. To interpret these numbers, we say: for every one unit increase in `total_bill` (i.e., every time the bill increases by a dollar), the tip increases by 0.105 (i.e., 10.5 cents).

If we just want the coefficients, we can call the `.params` attribute on the `results`.

Click here to view code image

```
print(results.params)
```

```
Intercept     0.920270
total_bill    0.105025
dtype: float64
```

Depending on your field, you may also need to report a confidence interval, which identifies the possible values the estimated value can take on. The confidence interval includes the values less than `[0.025 0.975]`. We can also extract these values using the `.conf_int()` method.

Click here to view code image

```
print(results.conf_int())
```

```
                   0         1
Intercept   0.605622  1.234918
total_bill  0.090517  0.119532
```

13.1.2 With scikit-learn

We can also use the `sklearn` library to fit various machine learning models. To perform the same analysis we just did, we need to import the `linear_model` module from this library.

Click here to view code image

```
from sklearn import linear_model
```

We can then create our linear regression object.

Click here to view code image

```
# create our LinearRegression object
lr = linear_model.LinearRegression()
```

Next, we need to specify the predictor, `X`, and the response, `y`. To do this, we pass in the columns we want to use for the model.

> **Note**
>
> Note the parameters are upper-case letter `X` and lower-case letter `y`.
>
> This comes from mathematical notation, where the predictors, `X` are a **matrix** of values, and the response, `y`, is a **vector** of values.

Too simple of an example

If we simply pass in a single variable into the `X` parameter, we actually get an error.

Click here to view code image

```
# note it is an uppercase X
# and a lowercase y
# this will fail because our X has only 1
variable
```

```python
predicted = lr.fit(X=tips['total_bill'],
y=tips['tip'])
```

```
ValueError: Expected 2D array, got 1D array
instead:
array=[16.99 10.34 21.01 23.68 24.59 25.29  8.77
26.88 15.04 14.78 10.27 35.26
 15.42 18.43 14.83 21.58 10.33 16.29 16.97 20.65
17.92 20.29 15.77 39.42
 19.82 17.81 13.37 12.69 21.7  19.65  9.55 18.35
15.06 20.69 17.78 24.06
 16.31 16.93 18.69 31.27 16.04 17.46 13.94  9.68
30.4  18.29 22.23 32.4
 28.55 18.04 12.54 10.29 34.81  9.94 25.56 19.49
38.01 26.41 11.24 48.27
 20.29 13.81 11.02 18.29 17.59 20.08 16.45  3.07
20.23 15.01 12.02 17.07
 26.86 25.28 14.73 10.51 17.92 27.2  22.76 17.29
19.44 16.66 10.07 32.68
 15.98 34.83 13.03 18.28 24.71 21.16 28.97 22.49
5.75 16.32 22.75 40.17
 27.28 12.03 21.01 12.46 11.35 15.38 44.3  22.42
20.92 15.36 20.49 25.21
 18.24 14.31 14.    7.25 38.07 23.95 25.71 17.31
29.93 10.65 12.43 24.08
 11.69 13.42 14.26 15.95 12.48 29.8   8.52 14.52
11.38 22.82 19.08 20.27
 11.17 12.26 18.26  8.51 10.33 14.15 16.   13.16
17.47 34.3  41.19 27.05
 16.43  8.35 18.64 11.87  9.78  7.51 14.07 13.13
17.26 24.55 19.77 29.85
 48.17 25.   13.39 16.49 21.5 12.66  16.21 13.81
17.51 24.52 20.76 31.71
```

```
 10.59 10.63 50.81 15.81  7.25 31.85 16.82 32.9
17.89 14.48  9.6  34.63
 34.65 23.33 45.35 23.17 40.55 20.69 20.9  30.46
18.15 23.1  15.69 19.81
 28.44 15.48 16.58  7.56 10.34 43.11 13.   13.51
18.71 12.74 13.   16.4
 20.53 16.47 26.59 38.73 24.27 12.76 30.06 25.89
48.33 13.27 28.17 12.9
 28.15 11.59  7.74 30.14 12.16 13.42  8.58 15.98
13.42 16.27 10.09 20.45
 13.28 22.12 24.01 15.69 11.61 10.77 15.53 10.07
12.6  32.83 35.83 29.03
 27.18 22.67 17.82 18.78].
Reshape your data either using array.reshape(-1,
1) if your data has a
single feature or array.reshape(1, -1) if it
contains a single sample.
```

Since `sklearn` is built to take `numpy` arrays, there will be times when you have to do some data manipulations to pass your dataframe into `sklearn`. The error message in the preceding output essentially tells us the matrix passed is not in the correct shape. We need to reshape our inputs. Depending on whether we have a single feature (which is the case here) or a single sample (i.e., multiple observations), we will specify `reshape(-1, 1)` or `reshape(1, -1)`, respectively.

Calling `.reshape()` directly on the column will raise either a `DeprecationWarning` (Pandas 0.17), a `ValueError` (Pandas 0.19), or an `AttributeError` depending on the version of Pandas being used.

Click here to view code image

```
# this will fail
predicted = lr.fit(
    X=tips["total_bill"].reshape(-1, 1),
```

```
y=tips["tip"]
)
```

```
AttributeError: 'Series' object has no attribute
'reshape'
```

To properly reshape our data, we must use the `.values` attribute (otherwise you may get another error or warning). When we call `.values` on a Pandas dataframe or series, we get the `numpy ndarray` representation of the data.

Click here to view code image

```
# we fix the data by putting it in the correct
shape for sklearn
predicted = lr.fit(
    X=tips["total_bill"].values.reshape(-1, 1),
y=tips["tip"]
)
```

Since `sklearn` works on `numpy ndarrays`, you may see code that explicitly passes in the `numpy` vector into the `X` or `y` parameter: `y=tips['tip'].values`.

Unfortunately, `sklearn` doesn't provide us with the nice summary tables that `statsmodels` does. This reflects differing schools of thought: statistics and computer science in contrast to prediction and machine learning. To obtain the coefficients in `sklearn`, we call the `.coef_` attribute on the fitted model.

```
print(predicted.coef_)
```

```
[0.10502452]
```

To get the intercept, we call the `.intercept_` attribute.

```
print(predicted.intercept_)
```

```
0.920269613554674
```

Notice that we get the same results as we did with `statsmodels`. That is, people in our data set are tipping about 10% of their bill amount.

13.2 Multiple Regression

In simple linear regression, one predictor is regressed on a single response variable. Alternatively, we can use multiple regression to put multiple predictors in a model.

13.2.1 With `statsmodels`

Fitting a multiple regression model to a data set is very similar to fitting a simple linear regression model. Using the formula interface, we add the other covariates to the right-hand side.

Click here to view code image

```
# note the .fit() method chain at the end
model = smf.ols(formula="tip ~ total_bill +
size", data=tips).fit()
```

Click here to view code image

```
print(model.summary())
```

Click here to view code image

```
                          OLS Regression Results
==============================================================
========================
Dep. Variable:                          tip   R-squared:
0.468
Model:                                  OLS   Adj. R-
```

```
squared:                    0.463
Method:                     Least Squares  F-
statistic:                       105.9
Date:                    Thu, 01 Sep 2022  Prob (F-
statistic):         9.67e-34
Time:                            01:55:46  Log-
Likelihood:                 -347.99
No. Observations:                     244  AIC:
702.0
Df Residuals:                         241  BIC:
712.5
Df Model:                               2
Covariance Type:                nonrobust
==================================================
========================
                 coef    std err          t
P>|t|      [0.025  0.975]
--------------------------------------------------
------------------------
Intercept      0.6689      0.194     3.455
0.001       0.288   1.050
total_bill     0.0927      0.009    10.172
0.000       0.075   0.111
size           0.1926      0.085     2.258
0.025       0.025   0.361
==================================================
========================
Omnibus:                       24.753   Durbin-
Watson:                    2.100
Prob(Omnibus):                  0.000   Jarque-Bera
(JB):               46.169
Skew:                           0.545   Prob(JB):
9.43e-11
```

```
Kurtosis:                       4.831   Cond. No.
67.6
==================================================
========================

Notes:
[1] Standard Errors assume that the covariance
matrix of the errors is correctly specified.
```

The interpretations are exactly the same as before, except each parameter is interpreted “with all other variables held constant.” That is, for every one unit increase (dollar) in `total_bill`, the tip increases by `0.09` (i.e., 9 cents) as long as the size of the group does not change.

13.2.2 With scikit-learn

The syntax for multiple regression in `sklearn` is very similar to the syntax for simple linear regression with this library. To add more features to the model, we pass in the columns we want to use.

Click here to view code image

```
lr = linear_model.LinearRegression()

# since we are performing multiple regression
# we no longer need to reshape our X values
predicted = lr.fit (X=tips[["total_bill",
"size"]], y=tips["tip"])

print(predicted.coef_)
```

```
[0.09271334 0.19259779]
```

We can get the intercept from the model just as we did earlier.

Click here to view code image

```
print(predicted.intercept_)
```

```
0.6689447408125035
```

13.3 Models with Categorical Variables

So far, we have used only continuous predictors in our model. If we look at the `.info()` method of our `tips` data set, however, we can see that our data includes categorical variables (you can also use the `.dtypes` attribute).

Click here to view code image

```
print(tips.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   total_bill   244 non-null    float64
 1   tip          244 non-null    float64
 2   sex          244 non-null    category
 3   smoker       244 non-null    category
 4   day          244 non-null    category
 5   time         244 non-null    category
 6   size         244 non-null    int64
dtypes: category(4), float64(2), int64(1)
memory usage: 7.4 KB
None
```

When we want to model a categorical variable, we have to create "dummy variables." That is, each unique value in the category becomes a new binary feature. These are also called “one-hot encoding,” depending on the field you’re in. For example, `sex` in our data can hold one of two values, `Female` or `Male`.

Click here to view code image

```
print(tips.sex.unique())
```

```
['Female', 'Male']
Categories (2, object): ['Male', 'Female']
```

13.3.1 Categorical Variables in `statsmodels`

`statsmodels` will automatically create dummy variables for us. To avoid multicollinearity, we typically drop one of the dummy variables. That is, if we have a column that indicates whether an individual is female, then we know if the person is not female (in our data), that person must be male. In such a case, we can effectively drop the dummy variable that codes for males and still have the same information.

Here’s the model that uses all the variables in our data.

Click here to view code image

```
model = smf.ols(
    formula="tip ~ total_bill + size + sex +
smoker + day + time",
    data=tips,
).fit()
```

We can see from the summary that `statsmodels` automatically creates dummy variables as well as drops the reference variable to avoid multicollinearity.

Click here to view code image

```
print(model.summary())
```

```
                            OLS Regression Results
==================================================
========================
Dep. Variable:                    tip  R-squared:
0.470
Model:                            OLS  Adj. R-
squared:                0.452
Method:                 Least Squares  F-statistic:
26.06
Date:                Thu, 01 Sep 2022  Prob (F-
statistic):          1.20e-28
Time:                        01:55:46  Log-
Likelihood:              -347.48
No. Observations:                 244  AIC:
713.0
Df Residuals:                     235  BIC:
744.4
Df Model:                           8
Covariance Type:            nonrobust
==================================================
========================
                  coef    std err       t      P>|t|
[0.025       0.975]
--------------------------------------------------
------------------------
Intercept       0.5908      0.256   2.310     0.022
0.087        1.095
sex[T.Female]   0.0324      0.142   0.229     0.819
-0.247        0.311
smoker[T.No]    0.0864      0.147   0.589     0.556
-0.202        0.375
day[T.Fri]      0.1623      0.393   0.412     0.680
-0.613        0.937
```

```
day[T.Sat]      0.0408     0.471   0.087    0.931
-0.886       0.968
day[T.Sun]      0.1368     0.472   0.290    0.772
-0.793       1.066
time[T.Dinner] -0.0681     0.445  -0.153    0.878
-0.944       0.808
total_bill      0.0945     0.010   9.841    0.000
0.076       0.113
size            0.1760     0.090   1.966    0.051
-0.000       0.352
==================================================
=====================
Omnibus:                   27.860   Durbin-Watson:
2.096
Prob(Omnibus):              0.000   Jarque-Bera
(JB):                52.555
Skew:                       0.607   Prob(JB):
3.87e-12
Kurtosis:                   4.923   Cond. No.
281.
==================================================
========================

Notes:
[1] Standard Errors assume that the covariance
matrix of the errors is correctly specified.
```

The interpretation of the continuous (i.e., numeric) parameters is the same as before. However, our interpretation of categorical variables must be stated in relation to the reference variable (i.e., the dummy variable that was dropped from the analysis). For example, the coefficient for `sex[T.Female]` is `0.0324`. We interpret this value in relation to the reference value, `Male`; that is, we say that when the `sex` of the server "changes" from `Male` to `Female`, the tip increases by `0.324`. For the `day` variable:

Click here to view code image

```
print(tips.day.unique())
```

```
['Sun', 'Sat', 'Thur', 'Fri']
Categories (4, object): ['Thur', 'Fri', 'Sat',
'Sun']
```

We see that our `.summary()` is missing `Thur`, so that is the reference variable to use to interpret the coefficients.

13.3.2 Categorical Variables in scikit-learn

We have to manually create our dummy variables for `sklearn`. Luckily, Pandas has a function, `.get_dummies()`, that will do this work for us. This function converts all the categorical variables into dummy variables automatically, so we do not need to pass in individual columns one at a time. `sklearn` has a `OneHotEncoder` function that does something similar.

13.3.2.1 Dummy Variables in Pandas

The `get_dummies()` function in Pandas can create dummy variable encoding of a dataframe for us.

Click here to view code image

```
tips_dummy = pd.get_dummies(
    tips[["total_bill", "size", "sex", "smoker",
"day", "time"]]
)

print(tips_dummy)
```

```
   total_bill  size  sex_Male  sex_Female
smoker_Yes  smoker_No  \
0       16.99     2         0           1
0          1
1       10.34     3         1           0
0          1
2       21.01     3         1           0
0          1
3       23.68     2         1           0
0          1
4       24.59     4         0           1
0          1
..        ...   ...       ...         ...
...        ...
239     29.03     3         1           0
0          1
240     27.18     2         0           1
1          0
241     22.67     2         1           0
1          0
242     17.82     2         1           0
0          1
243     18.78     2         0           1
0          1

    day_Thur  day_Fri  day_Sat  day_Sun
time_Lunch  time_Dinner
0          0        0        0        1
0            1
1          0        0        0        1
0            1
2          0        0        0        1
0            1
```

```
3            0          0          0          1
0              1
4            0          0          0          1
0              1
..         ...        ...        ...        ...
...            ...
239          0          0          1          0
0              1
240          0          0          1          0
0              1
241          0          0          1          0
0              1
242          0          0          1          0
0              1
243          1          0          0          0
0              1

[244 rows x 12 columns]
```

To drop the reference variable, we can pass in `drop_first=True`.

Click here to view code image

```
x_tips_dummy_ref = pd.get_dummies(
    tips[["total_bill", "size", "sex", "smoker",
"day", "time"]],
    drop_first=True,
)

print(x_tips_dummy_ref)
```

```
     total_bill  size  sex_Female  smoker_No
day_Fri  day_Sat  \
```

```
0          16.99     2            1           1
0         0
1          10.34     3            0           1
0         0
2          21.01     3            0           1
0         0
3          23.68     2            0           1
0         0
4          24.59     4            1           1
0         0
..           ...   ...          ...         ...
...       ...
239        29.03     3            0           1
0         1
240        27.18     2            1           0
0         1
241        22.67     2            0           0
0         1
242        17.82     2            0           1
0         1
243        18.78     2            1           1
0         0

     day_Sun   time_Dinner
0          1             1
1          1             1
2          1             1
3          1             1
4          1             1
..       ...           ...
239        0             1
240        0             1
241        0             1
```

```
242          0              1
243          0              1

[244 rows x 8 columns]
```

We fit the model just as we did earlier.

Click here to view code image

```
lr = linear_model.LinearRegression()
predicted = lr.fit(X=x_tips_dummy_ref,
y=tips["tip"])
```

We also obtain the coefficients in the same way.

Click here to view code image

```
print(predicted.intercept_)
```

```
0.5908374259513787
```

Click here to view code image

```
print(predicted.coef_)
```

```
[ 0.09448701   0.175992   0.03244094   0.08640832
0.1622592   0.04080082
  0.13677854  -0.0681286 ]
```

13.3.2.2 Keeping Index Labels from sklearn

One of the annoying things when trying to interpret a model from `sklearn` is that the coefficients are not labeled. The labels are omitted because the `numpy ndarray` is unable to store this type of metadata. If we want our output to resemble something from `statsmodels`, we need to manually store the labels and append the coefficients to them.

Click here to view code image

```
import numpy as np

# create and fit the model
lr = linear_model.LinearRegression()
predicted = lr.fit (X=x_tips_dummy_ref,
y=tips["tip"])

# get the intercept along with other
coefficients
values = np.append(predicted.intercept_,
predicted.coef_)

# get the names of the values
names = np.append("intercept",
x_tips_dummy_ref.columns)

# put everything in a labeled dataframe
results = pd.DataFrame({"variable": names,
"coef": values})

print(results)
```

```
     variable      coef
0   intercept  0.590837
1  total_bill  0.094487
2        size  0.175992
3  sex_Female  0.032441
4   smoker_No  0.086408
5     day_Fri  0.162259
6     day_Sat  0.040801
```

```
7       day_Sun  0.136779
8   time_Dinner -0.068129
```

13.4 One-Hot Encoding in scikit-learn with Transformer Pipelines

Scikit-learn has its own way of processing data for analysis using "pipelines." We can use the one-hot encoding transformer in a pipeline to process our data in scikit-learn, instead of pandas, before we fit our model.

Click here to view code image

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
```

We first need to specify which columns we want to process, here we are only looking to work with categorical variables.

Click here to view code image

```
categorical_features = ["sex", "smoker", "day",
"time"]
categorical_transformer =
OneHotEncoder(drop="first")
```

Once we have the columns and the processing step we want, we can then pass the steps into `ColumnTransformer()`. Since we want to still have the numeric variables in the final model, but didn't specify a processing step for them, we pass in `remainder="passthrough"` to make sure those variables not specified in the `transformers` step still make it to the final model.

Click here to view code image

```
preprocessor = ColumnTransformer(
  transformers=[
    ("cat", categorical_transformer,
categorical_features),
  ],
  remainder="passthrough", # keep the numeric
columns
)
```

Finally, we can create a `Pipeline()` with all the preprocessing steps, and then to the model we want.

Click here to view code image

```
pipe = Pipeline(
  steps=[
    ("preprocessor", preprocessor),
    ("lr", linear_model.LinearRegression()),
  ]
)
```

Finally, we can fit our model just like before.

Click here to view code image

```
pipe.fit(
  X=tips[["total_bill", "size", "sex", "smoker",
"day", "time"]],
  y=tips["tip"],
)
```

Click here to view code image

```
Pipeline(steps=[('preprocessor',
```

```
ColumnTransformer(remainder='passthrough',
                                transformers=
[('cat',

OneHotEncoder(drop='first'),

['sex', 'smoker', 'day',

'time'])])),
             ('lr', LinearRegression())])
```

We can't get the `.intercept_` and `coef_` because the `Pipeline()`, is not a `LinearRegression()` object.

Click here to view code image

```
print(type(pipe))
```

```
<class 'sklearn.pipeline.Pipeline'>
```

We need to access the coefficients in an additional step. This is because not all models will have `intercept_` and `coef_` values, the `Pipeline()` is a generic function that works with any model within the `sklearn` library.

Click here to view code image

```
# combine the intercept and coefficients into
single vector
coefficients = np.append(
  pipe.named_steps["lr"].intercept_,
pipe.named_steps["lr"].coef_
)

# combine the intercept text with the other
```

```
feature names
labels = np.append(
  ["intercept"], pipe[
:-1].get_feature_names_out()
)

# create a dataframe of all the results
coefs = pd.DataFrame({"variable": labels,
"coef": coefficients})

print(coefs)
```

```
                variable      coef
0              intercept  0.803817
1          cat__sex_Male -0.032441
2        cat__smoker_Yes -0.086408
3           cat__day_Sat -0.121458
4           cat__day_Sun -0.025481
5          cat__day_Thur -0.162259
6        cat__time_Lunch  0.068129
7  remainder__total_bill  0.094487
8        remainder__size  0.175992
```

Note that here the coefficients are not *exactly* the same as the `statsmodels` values because the reference variable is different.

Conclusion

This chapter introduced the basics of fitting models using the `statsmodels` and `sklearn` libraries. The concepts of adding features to a model and creating dummy variables are constantly used when fitting models. Thus far, we have focused on fitting linear models, where the **response variable** is a **continuous variable**. In later chapters, we'll fit models where the response variable is not a continuous variable.

14

Generalized Linear Models

Not every response variable will be continuous, so a linear regression will not be the correct model in every circumstance. Some outcomes may contain binary data (e.g., sick and not sick), or even count data (e.g., how many heads will I get when I flip a coin). A general class of models called generalized linear models (GLM) can account for these types of data, yet still use a linear combination of predictors.

About This Chapter

This chapter has been improved from its first edition version in a few ways. First, the data set example was changed to use the `titanic` data set from the `seaborn` library. The original code from the New York American Community Survey (ACS) was replaced with a new data set to make the model outputs more comparable across multiple libraries and programming languages (Appendix Z).

Next, the first edition of this book did not emphasize the different parameter options in functions from the scikit-learn library. This was originally a bit misleading as it gave off the impression that the models were doing exactly the same thing when they have different default behaviors. This chapter now gives more code and examples to emphasize the model differences between the modeling libraries. The original ACS modeling code can still be found in Appendix Y.

14.1 Logistic Regression (Binary Outcome Variable)

When you have a **binary response variable** (i.e., two possible outcomes), logistic regression is often used to model the data. We will be using the `titanic` data set that was exported from the `seaborn` library.

About the Titanic Data Set

The titanic data set is coming from the `seaborn` library. It was exported directly from the library to be read in so the exact data set can be reused in this chapter along with the example used in Appendix Z.2.

Below is the code used to create the data set.

Click here to view code image

```
import seaborn as sns

titanic = sns.load_data set("titanic")
titanic.to_csv("data/titanic.csv",
index=False)
```

With our data loaded, let's first subset the dataframe using only the columns we will be using for this model. We will also be dropping rows with missing values in them since models usually ignore observations that are not complete anyway, and we are not showing how to impute missing data in this chapter. Notice that we are dropping the missing values **after** we subsetted the columns we wanted, so we are not artificially dropping observations.

Click here to view code image

```
titanic_sub = (
    titanic[["survived", "sex", "age",
"embarked"]].copy().dropna()
)
```

```
print(titanic_sub)
```

```
          survived     sex        age  embarked
0                0    male       22.0         S
1                1  female       38.0         C
2                1  female       26.0         S
3                1  female       35.0         S
4                0    male       35.0         S
..             ...     ...        ...       ...
885              0  female       39.0         Q
886              0    male       27.0         S
887              1  female       19.0         S
889              1    male       26.0         C
890              0    male       32.0         Q

[712 rows x 4 columns]
```

In this data set, our outcome of interest is the `survived` column, on whether an individual survived (`1`) or died (`0`) during the sinking of the Titanic. The other columns, `sex`, `age`, and `embarked` are going to be the variable we use to see who survived.

Click here to view code image

```
# count of values in the survived column
print(titanic_sub["survived"].value_counts())
```

```
0    424
1    288
Name: survived, dtype: int64
```

The `embarked` column describes where the individual boarded the ship from. There are three values for `embarked`: Southampton (`S`), Cherbourg (`C`), and Queenstown (`Q`).

Click here to view code image

```
# count of values in the embarked column
print(titanic_sub["embarked"].value_counts())
```

```
S    554
C    130
Q     28
Name: embarked, dtype: int64
```

Interpreting results from a logistic regression model is not as straightforward as interpreting a linear regression model. In a logistic regression, as with all generalized linear models, there is a transformation (i.e., link function), that that affects how to interpret the results.

The link function for logistic regression is usually the `logit` link function.

$$logit\,(p) = log\left(\frac{p}{1-p}\right)$$

Where p is the probability of the event, and $\frac{p}{1-p}$ is the odds of the event. This is why logistic regression output is typically interpreted as “odds”, and we do that by undoing the *log* call by exponentiating our results. You can think of the “odds” of something as how many “times likely” the outcome will be. That phrasing should **only be used** as an analogy, however, as it is not technically correct. The value of an odds can only be greater than zero, and can never be negative. However, the “log odds” (i.e., logit), can be negative.

14.1.1 With `statsmodels`

To perform a logistic regression in `statsmodels` we can use the `logit()` function. The syntax for this function is the same as that used for linear regression in Chapter 13.

Click here to view code image

```
import statsmodels.formula.api as smf

# formula for the model
form = 'survived ~ sex + age + embarked'

# fitting the logistic regression model, note
the .fit() at the end
py_logistic_smf = smf.logit(formula=form,
data=titanic_sub).fit()

print(py_logistic_smf.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.509889
         Iterations 6
                           Logit Regression Results
==================================================
==========================
Dep. Variable:              survived   No.
Observations:                 712
Model:                         Logit   Df
Residuals:                       707
Method:                          MLE   Df Model:
4
Date:               Thu, 01 Sep 2022   Pseudo R-
squ.:                   0.2444
Time:                       01:55:49   Log-
Likelihood:                -363.04

converged:                      True   LL-Null:
-480.45
```

```
Covariance Type:            nonrobust   LLR p-
value:                  1.209e-49
==================================================
============================
                    coef      std err         z
P>|z|     [0.025        0.975]
--------------------------------------------------
----------------------------
Intercept           2.2046        0.322     6.851
0.000      1.574          2.835
sex[T.male]        -2.4760        0.191   -12.976
0.000     -2.850         -2.102
embarked[T.Q]      -1.8156        0.535    -3.393
0.001     -2.864         -0.767
embarked[T.S]      -1.0069        0.237    -4.251
0.000     -1.471         -0.543
age                -0.0081        0.007    -1.233
0.217     -0.021          0.005
==================================================
============================
```

We can then get the coefficients of the model, and exponentiate it to calculate the odds of each variable.

Click here to view code image

```
import numpy as np

# get the coefficients into a dataframe
res_sm = pd.DataFrame(py_logistic_smf.params,
columns=["coefs_sm"])

# calculate the odds
res_sm["odds_sm"] = np.exp(res_sm["coefs_sm"])
```

```
# round the decimals
print(res_sm.round(3))
```

```
               coefs_sm  odds_sm
Intercept         2.205    9.066
sex[T.male]      -2.476    0.084
embarked[T.Q]    -1.816    0.163
embarked[T.S]    -1.007    0.365
age              -0.008    0.992
```

An example interpretation of these numbers would be that for every one unit increase in `age`, the **odds** of the `survived` decreases by 0.992 **times**. Since the value is close to 1, it seems that age wasn't too much of a factor in survival. You can also confirm that statement by looking at the p-value for the variable in the summary table (under the `P>|z|` column).

A similar interpretation can be made with categorical variables. Recall that categorical variables are always interpreted in relation to the reference variable.

There are two potential values for `sex` in this data set, `male` and `female`, but only a coefficient for `male` is given. So that means the value is interpreted as "males compared to females", where `female` is the reference variable. The odds for the `male` variable are interpreted as: males were `0.084` times more likely to survive compared to females (the odds for not surviving the tragedy were high for males).

14.1.2 With sklearn

When using `sklearn`, remember that dummy variables need to be created manually.

Click here to view code image

```
titanic_dummy = pd.get_dummies(
    titanic_sub[["survived", "sex", "age",
```

```
"embarked"]],
    drop_first=True
)
```

Click here to view code image

```
# note our outcome variable is the first column
(index 0)
print(titanic_dummy)
```

```
     survived   age  sex_male  embarked_Q  embarked_S
0           0  22.0         1           0           1
1           1  38.0         0           0           0
2           1  26.0         0           0           1
3           1  35.0         0           0           1
4           0  35.0         1           0           1
..        ...   ...       ...         ...         ...
885         0  39.0         0           1           0
886         0  27.0         1           0           1
887         1  19.0         0           0           1
889         1  26.0         1           0           0
890         0  32.0         1           1           0

[712 rows x 5 columns]
```

We can then use the `LogisticRegression()` function from the `linear_model` module to create a logistic regression output to fit our model.

Click here to view code image

```
from sklearn import linear_model

# this is the only part that fits the model
```

```
py_logistic_sklearn1 = (
  linear_model.LogisticRegression().fit(
    X=titanic_dummy.iloc[:, 1:], # all the
columns except first
    y=titanic_dummy.iloc[:, 0]   # just the
first column
  )
)
```

> **Danger**
>
> Please read Section 14.1.3, which emphasizes reading the documentation and being aware of the ramifications of the default scikit-learn `LogisticRegression()` values.

The code below will process the scikit-learn logistic regression fitted model into a single dataframe so we can better compare results.

Click here to view code image

```
# get the names of the dummy variable columns
dummy_names = titanic_dummy.columns.to_list()
# get the intercept and coefficients into a
dataframe
sk1_res1 = pd.DataFrame(
    py_logistic_sklearn1.intercept_,
    index=["Intercept"],
    columns=["coef_sk1"],

)
sk1_res2 = pd.DataFrame(
    py_logistic_sklearn1.coef_.T,
    index=dummy_names[1:],
    columns=["coef_sk1"],
```

```
)

# put the results into a single dataframe to
show the results
res_sklearn_pd_1 = pd.concat([sk1_res1,
sk1_res2])

# calculate the odds
res_sklearn_pd_1["odds_sk1"] =
np.exp(res_sklearn_pd_1["coef_sk1"])

print(res_sklearn_pd_1.round(3))
```

```
            coef_sk1   odds_sk1
Intercept      2.024      7.571
age           -0.008      0.992
sex_male      -2.372      0.093
embarked_Q    -1.369      0.254
embarked_S    -0.887      0.412
```

You will notice here that the coefficient values are different from the ones calculated from the `statsmodels` section we just did. The differences are more than a simple rounding error too!

14.1.3 Be Careful of scikit-learn Defaults

The main reason why the `sklearn` results differ from the `statsmodels` results stems from the domain differences where the two packages come from. Scikit-learn comes more from the machine learning world and is focused on **prediction** so the model defaults are set for numeric stability, and not for inference. However, `statsmodels` functions are implemented in a manner more traditional for statistics.

The `LogisticRegression()` function has a `penalty` parameter that defaults to `'l2'`, which adds an L2 penalty term (more about penalty terms in Chapter 17). If we want `LogisticRegression()` to behave in a manner more traditional for statistics, we need to set `penalty="none"`.

Click here to view code image

```
# fit another logistic regression with no
penalty
py_logistic_sklearn2 =
linear_model.LogisticRegression(
     penalty="none" # this parameter is
important!
).fit(
     X=titanic_dummy.iloc[:, 1:],    # all the
columns except first
     y=titanic_dummy.iloc[:, 0]    # just the
first column
)

# rest of the code is the same as before, except
variable names
sk2_res1 = pd.DataFrame(
    py_logistic_sklearn2.intercept_,
    index=["Intercept"],
    columns=["coef_sk2"],
)
sk2_res2 = pd.DataFrame(
    py_logistic_sklearn2.coef_.T,
    index=dummy_names[1:],
    columns=["coef_sk2"],
)
```

```
res_sklearn_pd_2 = pd.concat([sk2_res1,
sk2_res2])
res_sklearn_pd_2["odds_sk2"] =
np.exp(res_sklearn_pd_2["coef_sk2"])
```

Note

In general, always check the documentation for the functions you are using, and make sure you know what all the parameters are doing.

First, let's look at the original `statsmodels` results

Click here to view code image

```
sm_results = res_sm.round(3)

# sort values to make things easier to compare
sm_results = sm_results.sort_index()

print(sm_results)
```

```
                coefs_sm  odds_sm
Intercept          2.205    9.066
age               -0.008    0.992
embarked[T.Q]     -1.816    0.163
embarked[T.S]     -1.007    0.365
sex[T.male]       -2.476    0.084
```

Now, let's compare them with the two `sklearn` results

Click here to view code image

```
# concatenate the 2 model results
sk_results = pd.concat(
```

```
    [res_sklearn_pd_1.round(3),
res_sklearn_pd_2.round(3)],
    axis="columns",
)

# sort cols and rows to make things easy to
compare
sk_results =
sk_results[sk_results.columns.sort_values()]
sk_results = sk_results.sort_index()

print(sk_results)
```

Click here to view code image

```
             coef_sk1  coef_sk2  odds_sk1
odds_sk2
Intercept       2.024     2.205     7.571
9.066
age            -0.008    -0.008     0.992
0.992
embarked_Q     -1.369    -1.816     0.254
0.163
embarked_S     -0.887    -1.007     0.412
0.365
sex_male       -2.372    -2.476     0.093
0.084
```

The results here can also be compared to the same data and model from the R programming language in Appendix Z.2. You can see how subtle differences between the model parameters can cause differences in the interpretations.

14.2 Poisson Regression (Count Outcome Variable)

Poisson regression is performed when our response variable involves count data.

Click here to view code image

```
acs = pd.read_csv('data/acs_ny.csv')
print(acs.columns)
```

```
Index(['Acres', 'FamilyIncome', 'FamilyType',
'NumBedrooms',
       'NumChildren', 'NumPeople', 'NumRooms',
'NumUnits',
       'NumVehicles', 'NumWorkers', 'OwnRent',
'YearBuilt',
       'HouseCosts', 'ElectricBill', 'FoodStamp',
'HeatingFuel',
       'Insurance', 'Language'],
      dtype='object')
```

For example, in the `acs` data, the `NumChildren` variable is an example of count data.

> **About the ACS Data Set**
>
> The American Community Survey (ACS) data we are using contains information about family and house size in New York.

14.2.1 With `statsmodels`

We can perform a Poisson regression using the `poisson()` function in `statsmodels`. We will use the `NumBedrooms` variable (Figure 14.1).

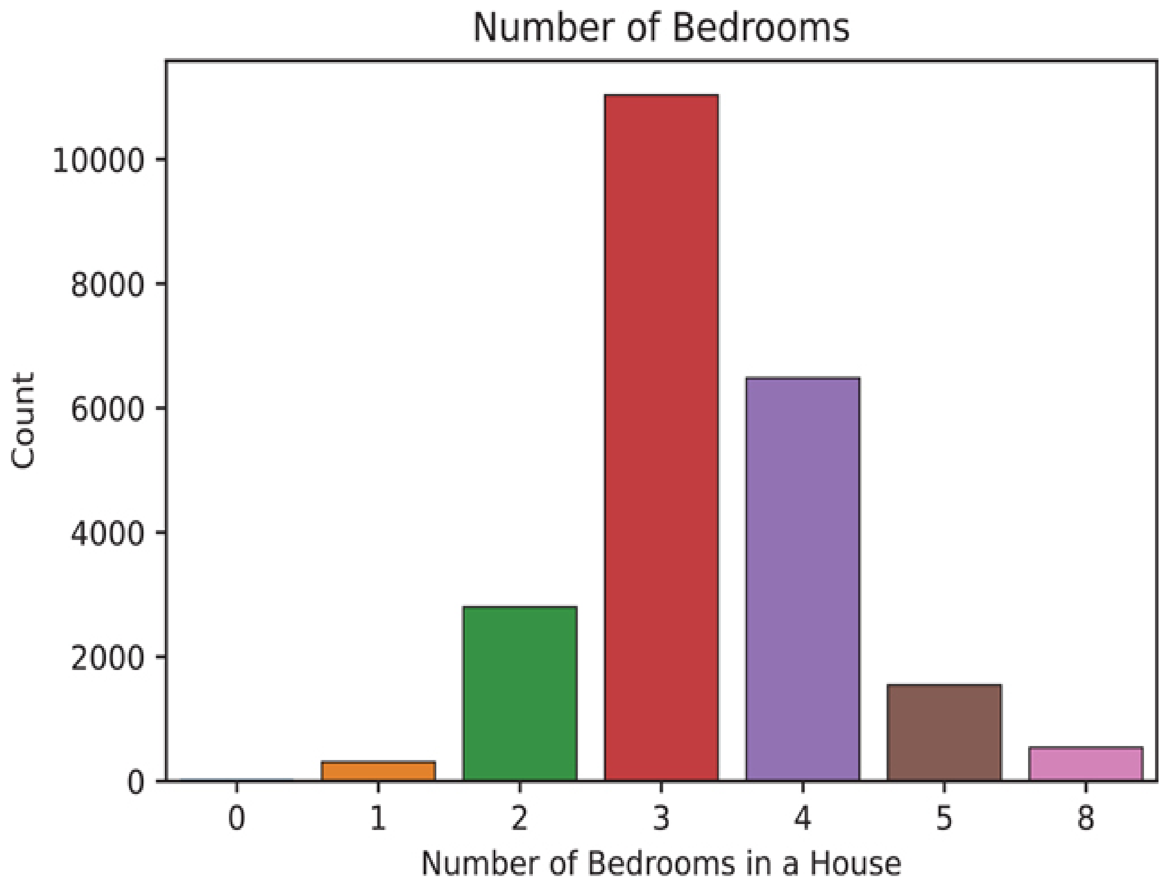


Figure 14.1 Bar plot using the `statsmodels countplot()` function of the `NumBedrooms` variable

Click here to view code image

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
sns.countplot(data = acs, x = "NumBedrooms",
ax=ax)

ax.set_title('Number of Bedrooms')
ax.set_xlabel('Number of Bedrooms in a House')
ax.set_ylabel('Count')

plt.show()
```

.

Click here to view code image

```
model = smf.poisson(
  "NumBedrooms ~ HouseCosts + OwnRent", data=acs
)
results = model.fit()

print(results.summary())
```

```
Optimization terminated successfully.
         Current function value: 1.680998
         Iterations 10

                     Poisson Regression Results
==================================================
============================
Dep. Variable:          NumBedrooms   No.
Observations:              22745
Model:                      Poisson   Df
Residuals:                  22741
Method:                         MLE   Df Model:
3
Date:              Thu, 01 Sep 2022   Pseudo R-
squ.:              0.008309
Time:                      01:55:49   Log-
Likelihood:             -38234.
converged:                     True   LL-Null:
-38555.
Covariance Type:          nonrobust   LLR p-
value:              1.512e-138
==================================================
====================================
```

```
                              coef     std err
z       P>|z|     [0.025        0.975]
--------------------------------------------------
------------------------------------
Intercept                   1.1387       0.006
184.928      0.000      1.127        1.151
OwnRent[T.Outright]    -0.2659       0.051
-5.182       0.000     -0.367       -0.165
OwnRent[T.Rented]      -0.1237       0.012
-9.996       0.000     -0.148       -0.099
HouseCosts           6.217e-05     2.96e-06
21.017       0.000   5.64e-05      6.8e-05
==================================================
====================================
```

The benefit of using a generalized linear model is that the only things that need to be changed are the `family` of the model that needs to be fit, and the `link` function that transforms our data. We can also use the more general `glm()` function to perform all the same calculations.

Click here to view code image

```
import statsmodels.api as sm
import statsmodels.formula.api as smf

model = smf.glm(
   "NumBedrooms ~ HouseCosts + OwnRent",
   data=acs,

family=sm.families.Poisson(sm.genmod.families.li
nks.log()),
).fit()
```

In this example, we are using the `Poisson` family, which comes from `sm.families. Poisson`, and we're passing in the log link function via `sm.genmod.families.links.log()`. We get the same values as we did earlier when we use this method.

Click here to view code image

```
print(results.summary())
```

```
                    Poisson Regression Results
==================================================
============================
Dep. Variable:              NumBedrooms  No.
Observations:               22745
Model:                       Poisson Df
Residuals:                      22741
Method:                          MLE Df  Model:
3
Date:                  Thu, 01 Sep 2022  Pseudo R-
squ.:                0.008309
Time:                          01:55:49  Log-
Likelihood:               -38234.
converged:                         True  LL-Null:
-38555.
Covariance Type:              nonrobust  LLR p-
value:                1.512e-138
==================================================
===================================
                          coef    std err
z   P>|z|     [0.025        0.975]
--------------------------------------------------
-----------------------------------
Intercept               1.1387      0.006
184.928   0.000     1.127         1.151
OwnRent[T.Outright]    -0.2659      0.051
-5.182   0.000    -0.367        -0.165
```

```
OwnRent[T.Rented]      -0.1237      0.012
-9.996   0.000     -0.148         -0.099
HouseCosts            6.217e-05   2.96e-06
21.017   0.000   5.64e-05        6.8e-05
==================================================
===================================
```

14.2.2 Negative Binomial Regression for Overdispersion

If our assumptions for Poisson regression are violated—that is, if our data has overdispersion—we can perform a negative binomial regression instead (Figure 14.2). Overdispersion is the statistics term meaning the numbers have more variance than expected, i.e., the values are too spread out.

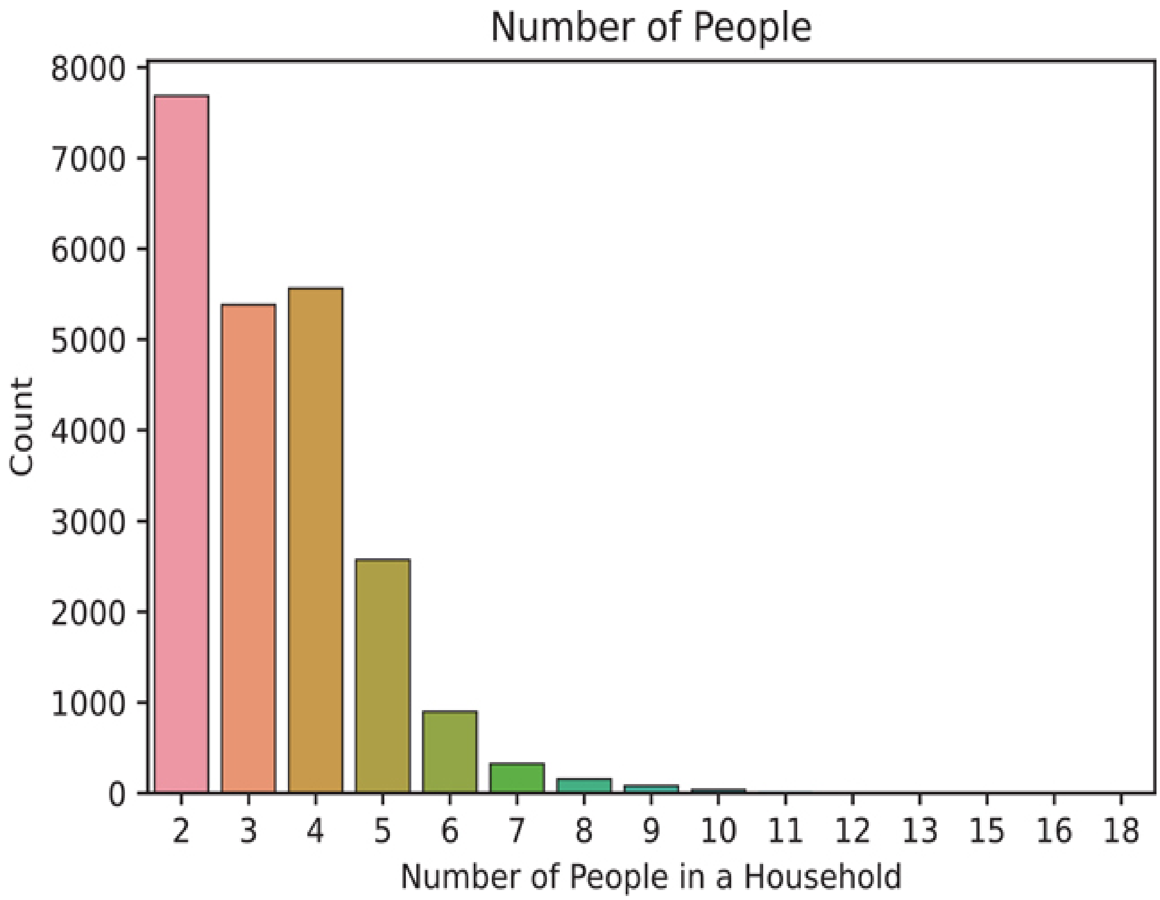


Figure 14.2 Bar plot using the `statsmodels countplot()` function of the `NumPeople` variable

Click here to view code image

```
fig, ax = plt.subplots()

sns.countplot(data = acs, x = "NumPeople",
ax=ax)
ax.set_title('Number of People')
ax.set_xlabel('Number of People in a Household')
ax.set_ylabel('Count')

plt.show()
```

Click here to view code image

```
model = smf.glm(
  "NumPeople ~ Acres + NumVehicles",
  data=acs,
  family=sm.families.NegativeBinomial(
    sm.genmod.families.links.log()
  ),
)

results = model.fit()
```

Click here to view code image

```
print(results.summary())
```

```
              Generalized Linear Model Regression
Results
==================================================
============================
Dep. Variable:          NumPeople   No.
Observations:                 22745
Model:                        GLM Df   Residuals:
22741
Model Family:  NegativeBinomial Df   Model:
3
Link Function:                log   Scale:
1.0000
Method:                      IRLS   Log-
Likelihood:                  -53542.
Date:            Thu, 01 Sep 2022   Deviance:
2605.6
Time:                    01:55:50   Pearson chi2:
2.99e+03
```

```
No. Iterations:                    6   Pseudo R-squ.
(CS):                0.003504
Covariance Type:          nonrobust
==================================================
==============================
                    coef    std err          z
P>|z|      [0.025       0.975]
--------------------------------------------------
------------------------------
Intercept          1.0418      0.025    41.580
0.000       0.993        1.091
Acres[T.10+]      -0.0225      0.040    -0.564
0.573      -0.101        0.056
Acres[T.Sub 1]     0.0509      0.019     2.671
0.008       0.014        0.088
NumVehicles        0.0661      0.008     8.423
0.000       0.051        0.081
==================================================
==============================
```

Look for the reference variable in `Acres`.

Click here to view code image

```
print(acs["Acres"].value_counts())
```

```
Sub 1   17114
1-10     4627
10+      1004
Name: Acres, dtype: int64
```

14.3 More Generalized Linear Models

The documentation page for GLM found in `statsmodels` lists the various families that can be passed into the `glm` parameter.[1] These families can all be found under `sm.families.<FAMILY>`:

- `Binomial`
- `Gamma`
- `Gaussian`
- `InverseGaussian`
- `NegativeBinomial`
- `Poisson`
- `Tweedie`

The link functions are found under `sm.families.family.<FAMILY>.links`. Following is the list of link functions, but note that not all link functions are available for each family:

- `CDFLink`
- `CLogLog`
- `LogLog`
- `Log`
- `Logit`
- `NegativeBinomial`
- `Power`
- `cauchy`
- `cloglog`
- `loglog`
- `identity`
- `inverse_power`
- `inverse_squared`
- `log`
- `logit`

For example, using the all the link functions for the `Binomial` family.

1. https://www.statsmodels.org/dev/glm.html

Click here to view code image

```
sm.families.family.Binomial.links
```

```
[statsmodels.genmod.families.links.Logit,
 statsmodels.genmod.families.links.probit,
 statsmodels.genmod.families.links.cauchy,
 statsmodels.genmod.families.links.Log,
 statsmodels.genmod.families.links.CLogLog,
 statsmodels.genmod.families.links.LogLog,
 statsmodels.genmod.families.links.identity]
```

Conclusion

This chapter covered some of the most basic and common models used in data analysis. These types of models serve as an interpretable baseline for more complex machine learning models. As we cover more complex models, keep in mind that sometimes simple and tried-and-true interpretable models can outperform the fancy newer models.

15

Survival Analysis

Survival analysis is used when we want to model how much time passes before something happens. It is typically used in health contexts when we are looking to see if a drug or intervention prevents an adverse event from occurring. Before we begin with examples of survival analysis, let's define some terms first.

- Event: Outcome, situation, or "event" you are interested in tracking in your study.
- Follow-up: "Lost to follow-up" is a term used in medical data. It means that the patient stopped "following up" to the visits. This can mean that the patient just stopped showing up, or the patient has died Usually, in this context, death is the "event" of interest.
- Censoring: Unsure of the status for a particular observation. This can be right-censored (no more data after this period of time), or left-censored (no data before this period of time). Right-censoring typically occurs from lost to follow up, or the event of interest has occurred (e.g., death).
- Stop time: A point in the data where some censoring event has occurred.

Survival analysis is typically used in medical research when trying to determine whether one treatment prevents a serious adverse event (e.g., death) better than the standard or a different treatment. Survival analysis is also used when data is censored, meaning the exact outcome of an event is not entirely known. For example, patients who follow a treatment regimen may sometimes be lost break to follow-up. The censoring usually occurs at a "stop" event.

Survival analysis is performed using the lifelines library.[1]

1. `lifelines` documentation: `https://lifelines.readthedocs.io/en/latest/`

15.1 Survival Data

Click here to view code image

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

print(bladder)
```

Click here to view code image

```
     id  rx  number  size  stop  event  enum
0     1   1       1     3     1      0     1
1     1   1       1     3     1      0     2
2     1   1       1     3     1      0     3
3     1   1       1     3     1      0     4
4     2   1       2     1     4      0     1
..   ..  ..     ...   ...   ...    ...   ...
335  84   2       2     1    54      0     4
336  85   2       1     3    59      0     1
337  85   2       1     3    59      0     2
338  85   2       1     3    59      0     3
339  85   2       1     3    59      0     4

[340 rows x 7 columns]
```

About the Bladder Data Set

The bladder data set comes from the R `{survival}` package. It contains 85 patients, their cancer recurrence status, and what treatment they were on. Below is a recreation of the code book for the data.

- `id`: Patient ID
- `rx`: Treatment (`1` = placebo, `2` = thiotepa)
- `number`: Initial number of tumors (`8` = 8 or more)
- `size`: Size (cm) of largest initial tumor
- `stop`: Recurrence or censoring time
- `event`: Bladder cancer re-occurrence (`0`: No, `1`: Yes)
- `enum`: Which recurrence (up to 4)

Here are the counts of the different treatments, `rx`.

Click here to view code image

```
print(bladder['rx'].value_counts())
```

```
1    188
2    152
Name: rx, dtype: int64
```

15.2 Kaplan Meier Curves

To perform our survival analysis, we import the `KaplanMeierFitter()` function from the `lifelines` library.

Click here to view code image

```
from lifelines import KaplanMeierFitter
```

Creating the model and fitting the data proceeds similarly to how models are fit using `sklearn`. The `stop` variable indicates when an event occurs, and the `event` variable signals whether the event of interest (bladder cancer re-occurrence) occurred. The `event` value can have a value of `0`, because people can be lost to follow-up. As noted earlier, this type of data is called “censored”.

Click here to view code image

```
kmf = KaplanMeierFitter()
kmf.fit(bladder['stop'],
event_observed=bladder['event'])
```

```
<lifelines.KaplanMeierFitter:"KM_estimate", fitted
with 340 total
observations, 228 right-censored observations>
```

We can plot the survival curve using `matplotlib`, as shown in Figure 15.1.

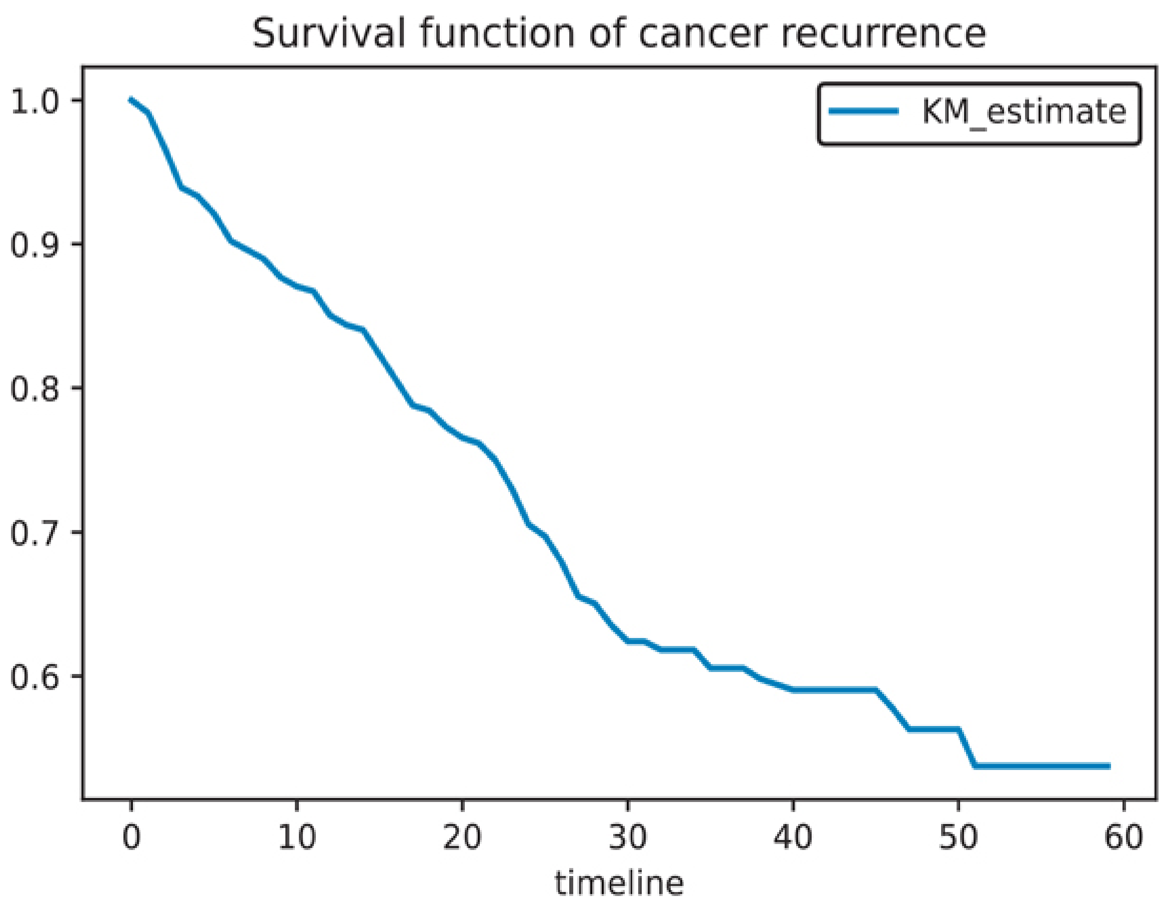


Figure 15.1 Survival function of cancer recurrence using the KaplanMeierFitter

Click here to view code image

```
import matplotlib.pyplot as plt
```

```
fig, ax = plt.subplots()
kmf.survival_function_.plot(ax=ax)
ax.set_title('Survival function of cancer
recurrence')
plt.show()
```

We can also show the confidence interval of our survival curve, as shown in Figure 15.2.

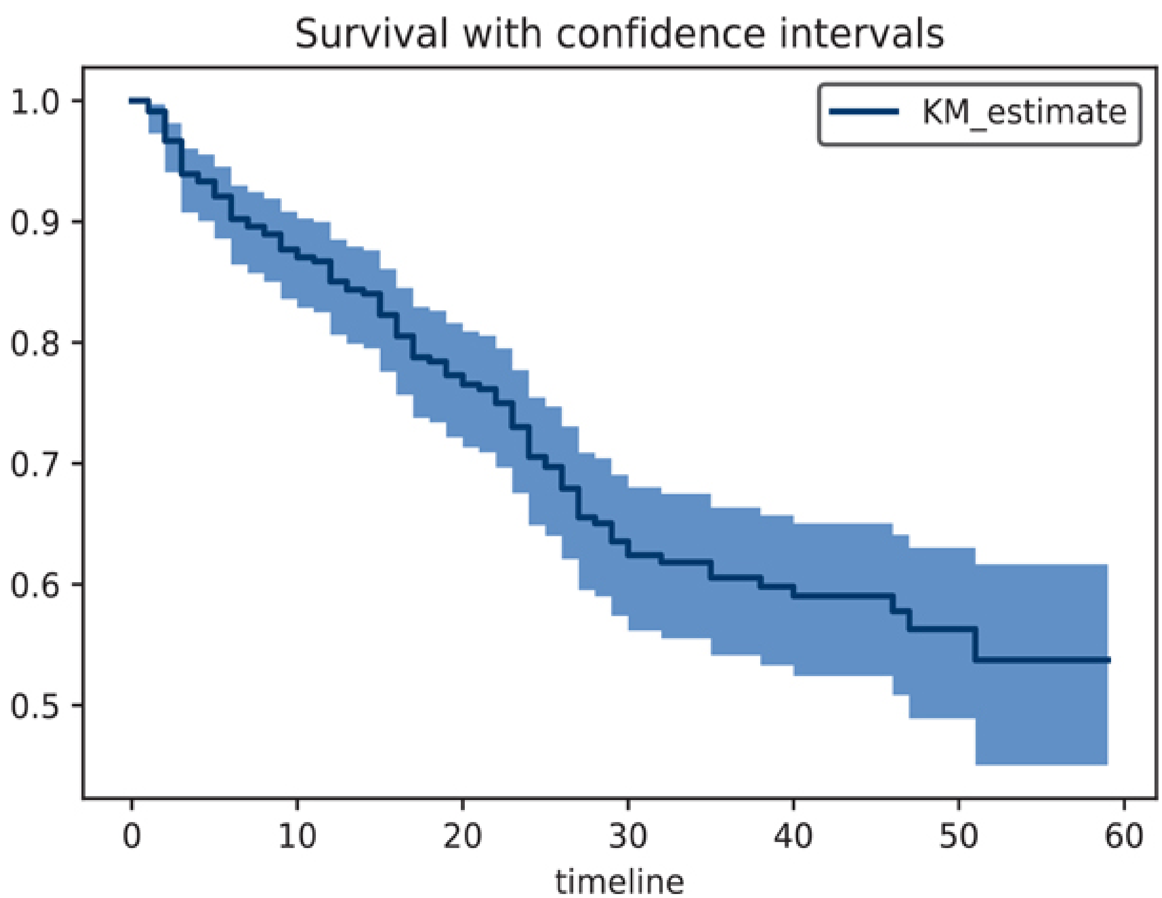


Figure 15.2 Survival function of cancer recurrence with confidence intervals

Click here to view code image

```
fig, ax = plt.subplots()
kmf.plot(ax=ax)
ax.set_title('Survival with confidence
intervals')
plt.show()
```

15.3 Cox Proportional Hazard Model

So far, we've just plotted the survival curve. We can also fit a model to predict survival rate. One such model is called the Cox proportional hazards model. We fit this model using the `CoxPHFitter()` class from `lifelines`.

Click here to view code image

```
from lifelines import CoxPHFitter

cph = CoxPHFitter()
```

We then pass in the columns to be used as predictors.

Click here to view code image

```
cph_bladder_df = bladder[
    ["rx", "number", "size", "enum", "stop",
"event"]
]
cph.fit(cph_bladder_df, duration_col="stop",
event_col="event")
```

```
<lifelines.CoxPHFitter: fitted with 340 total
observations, 228
right-censored observations>
```

Now we can use the `.print_summary()` method to print out the coefficients.

Click here to view code image

```
cph.print_summary()
```

covariate	**coef**	**exp (coef)**	**se (coef)**	**coef lower 95%**	**coef upper 95%**	**exp (coef) lower 95%**	**exp (coef) upper 95%**	**cmp to**	**z**	**p**	**-log2 (p)**
rx	-0.60	0.55	0.20	-0.99	-0.20	0.37	0.82	0.00	-2.97	0.00	8.41
number	0.22	1.24	0.05	0.13	0.31	1.13	1.36	0.00	4.68	0.00	18.38
size	-0.06	0.94	0.07	-0.20	0.08	0.82	1.09	0.00	-0.80	0.42	1.24
enum	-0.60	0.55	0.09	-0.79	-0.42	0.45	0.66	0.00	-6.42	0.00	32.80

We mainly focus on the hazard ratio when looking at CPH models. In the table this is represented by the `exp(coef)` column in the results. Values close to `1` show that there is no change in the survival hazard. Values from `0 -- 1` show a smaller hazard and values greater than `1` show an increase in hazard.

Note

In cancer studies, there is a difference in how the hazard ratios are interpreted.

- Hazard ratio > 1 is a bad prognostic factor

- Hazard ratio < 1 is a good prognostic factor

  That is, hazard ratios < `1` tell us what may be causing cancer.

15.3.1 Testing the Cox Model Assumptions

One way to check the Cox model's assumptions is to plot a separate survival curve by strata. In our example, our strata will be the values of the `rx` column, meaning we will plot a separate curve for each type of treatment. If the `log(-log(survival curve))` versus `log(time)` curves cross each other (Figure 15.3), it signals that the model needs to be stratified by the variable.

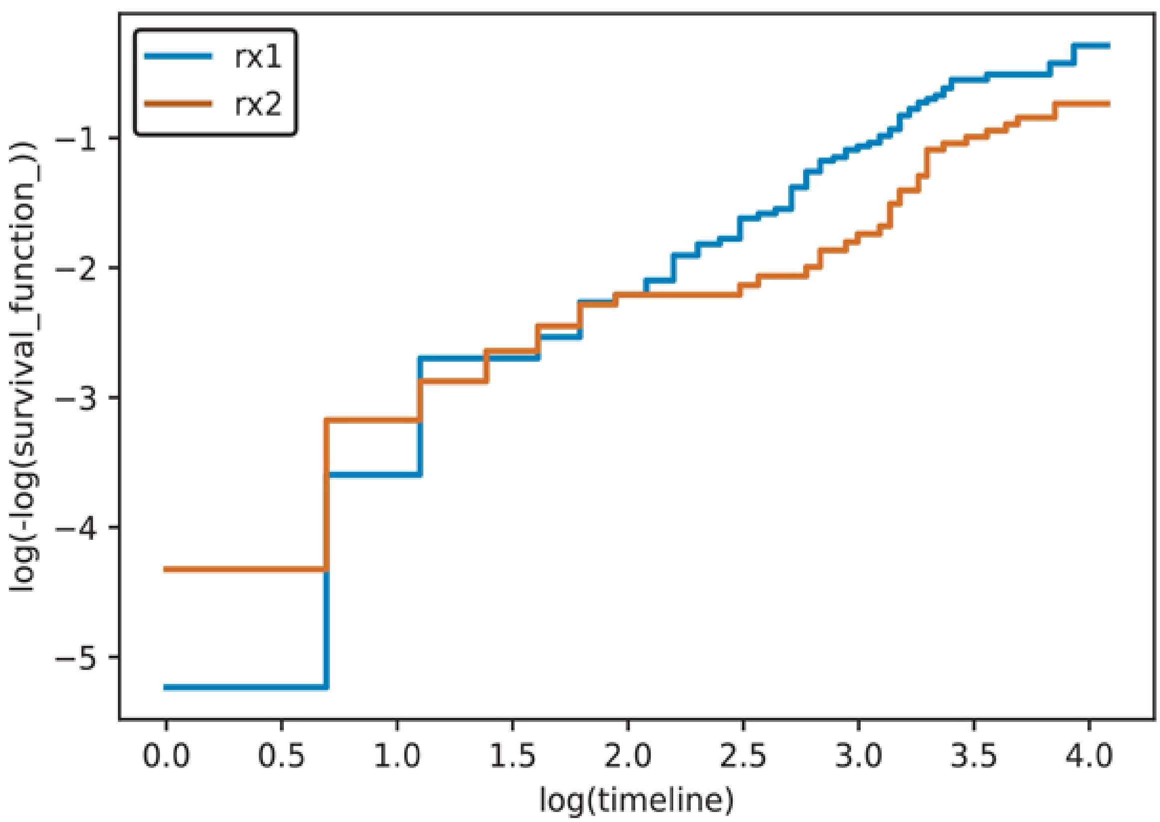


Figure 15.3 Plotting separate survival curves to check the Cox model assumptions

Click here to view code image

```
rx1 = bladder.loc[bladder['rx'] == 1]
rx2 = bladder.loc[bladder['rx'] == 2]

kmf1 = KaplanMeierFitter()
kmf1.fit(rx1['stop'],
event_observed=rx1['event'])

kmf2 = KaplanMeierFitter()
kmf2.fit(rx2['stop'],
event_observed=rx2['event'])

fig, axes = plt.subplots()

# put both plots on the same axes
kmf1.plot_loglogs(ax=axes)
kmf2.plot_loglogs(ax=axes)
```

Click here to view code image

```
axes.legend(['rx1', 'rx2'])

plt.show()
```

Since the lines cross each other, it makes sense to stratify our analysis.

Click here to view code image

```
cph_strat = CoxPHFitter()
cph_strat.fit(
    cph_bladder_df,
    duration_col="stop",
    event_col="event",
    strata=["rx"],
)
```

```
cph_strat.print_summary()
```

covariate	coef	exp (coef)	se (coef)	coef lower 95%	coef upper 95%	exp (coef) lower 95%	exp (coef) upper 95%	cmp to	z	p	-log2 (p)
number	0.21	1.24	0.05	0.12	0.30	1.13	1.36	0.00	4.60	0.000	17.84
size	-0.05	0.95	0.07	-0.19	0.08	0.82	1.09	0.00	-0.77	0.44	1.19
enum	-0.61	0.55	0.09	-0.79	-0.42	0.45	0.66	0.00	-6.45	0.000	33.07

Conclusion

Survival models measure “time to event” with censoring. They are commonly used in a health context but do not have to be solely used in that domain. If you can define some kind of event of interest, e.g., people who come to my website and purchase an item, you can potentially use survival models.

16

Model Diagnostics

Building models is a continuous art. As we start adding and removing variables from our models, we need a means to compare models with one another and a consistent way of measuring model performance. There are many ways we can compare models, and this chapter describes some of these methods.

16.1 Residuals

The residuals of a model compare what the model calculates and the actual values in the data. Let's fit some models on a housing data set.

Click here to view code image

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

print(housing.head())
```

```
  neighborhood            type  units  year_built
sq_ft    income  \
0    FINANCIAL R9-CONDOMINIUM     42      1920.0
36500   1332615
1    FINANCIAL R4-CONDOMINIUM     78      1985.0
126420   6633257
2    FINANCIAL RR-CONDOMINIUM    500         NaN
554174  17310000
```

```
3    FINANCIAL R4-CONDOMINIUM    282       1930.0
249076  11776313
4      TRIBECA R4-CONDOMINIUM    239       1985.0
219495  10004582

    income_per_sq_ft  expense  expense_per_sq_ft
net_income  \
0              36.51   342005               9.37
990610
1              52.47  1762295              13.94
4870962
2              31.24  3543000               6.39
13767000
3              47.28  2784670              11.18
8991643
4              45.58  2783197              12.68
7221385

      value  value_per_sq_ft       boro
0   7300000           200.00  Manhattan
1  30690000           242.76  Manhattan
2  90970000           164.15  Manhattan
3  67556006           271.23  Manhattan
4  54320996           247.48  Manhattan
```

We'll begin with a multiple linear regression model with three covariates.

Click here to view code image

```
import statsmodels
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

```
house1 = smf.glm(
    "value_per_sq_ft ~ units + sq_ft + boro",
data=housing
).fit()

print(house1.summary())
```

```
                 Generalized Linear Model Regression
Results
==================================================
============================
Dep. Variable:       value_per_sq_ft   No.
Observations:                 2626
Model:                          GLM Df   Residuals:
2619
Model Family:              Gaussian Df   Model:
6
Link Function:                identity   Scale:
1879.5
Method:                           IRLS   Log-
Likelihood:                -13621.
Date:               Thu, 01 Sep 2022   Deviance:
4.9224e+06
Time:                         01:55:55   Pearson
chi2:                   4.92e+06
No. Iterations:                      3   Pseudo R-
squ. (CS):               0.7772
Covariance Type:             nonrobust
==================================================
=======================================
                            coef    std err
z       P>|z|      [0.025       0.975]
```

```
---------------------------------------------------
----------------------------------------
Intercept                       43.2909      5.330
8.122      0.000       32.845       53.737
boro[T.Brooklyn]                34.5621      5.535
6.244      0.000       23.714       45.411
boro[T.Manhattan]              130.9924      5.385
24.327      0.000      120.439      141.546
boro[T.Queens]                  32.9937      5.663
5.827      0.000       21.895       44.092
boro[T.Staten Island]           -3.6303      9.993
-0.363      0.716      -23.216       15.956
units                           -0.1881      0.022
-8.511      0.000       -0.231       -0.145
sq_ft                            0.0002   2.09e-05
10.079      0.000        0.000        0.000
===================================================
========================================
```

We can plot the residuals of our model (Figure 16.1). What we are looking for is a plot with a random scattering of points. If a pattern is apparent, then we will need to investigate our data and model to see why this pattern emerged.

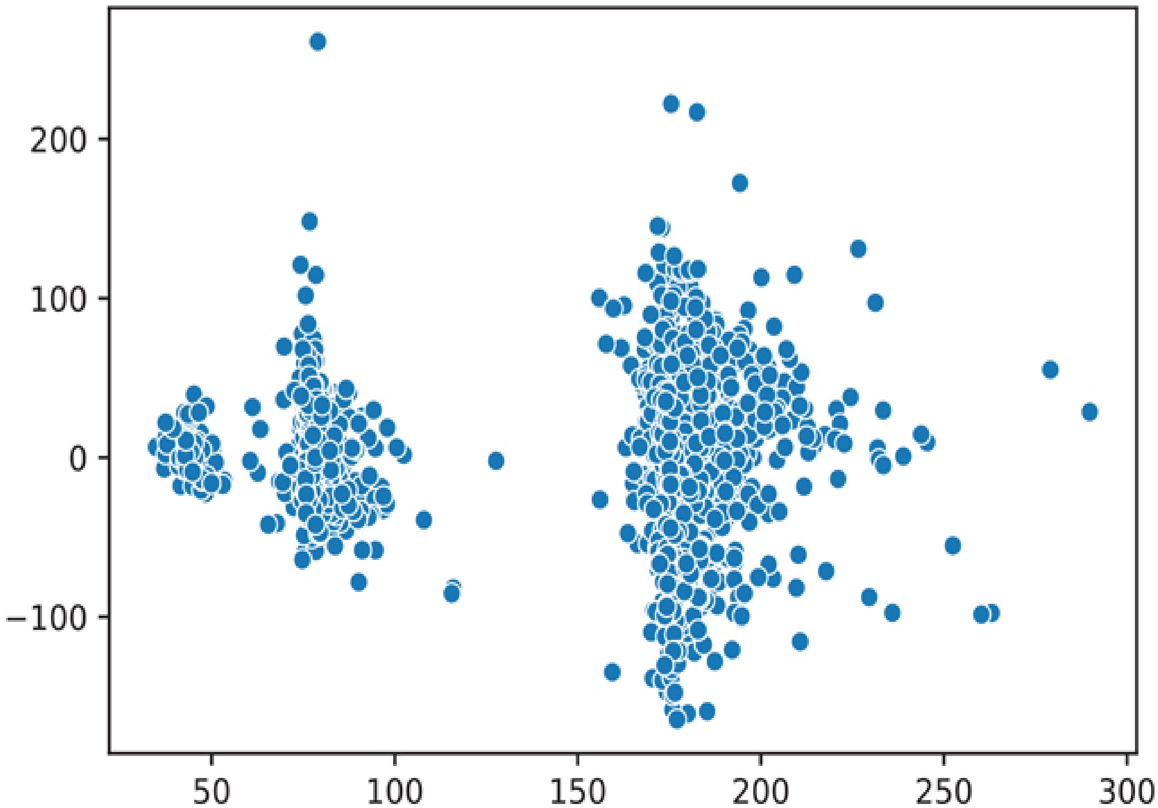


Figure 16.1 Residuals of the house1 model

Click here to view code image

```
import seaborn as sns
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
sns.scatterplot(
  x=house1.fittedvalues,
y=house1.resid_deviance, ax=ax
)

plt.show()
```

This residual plot is concerning because it contains obvious clusters and groups (residual plots are supposed to look random). We can color our plot by the `boro` variable, which indicates the borough of New York where the data apply (Figure 16.2).

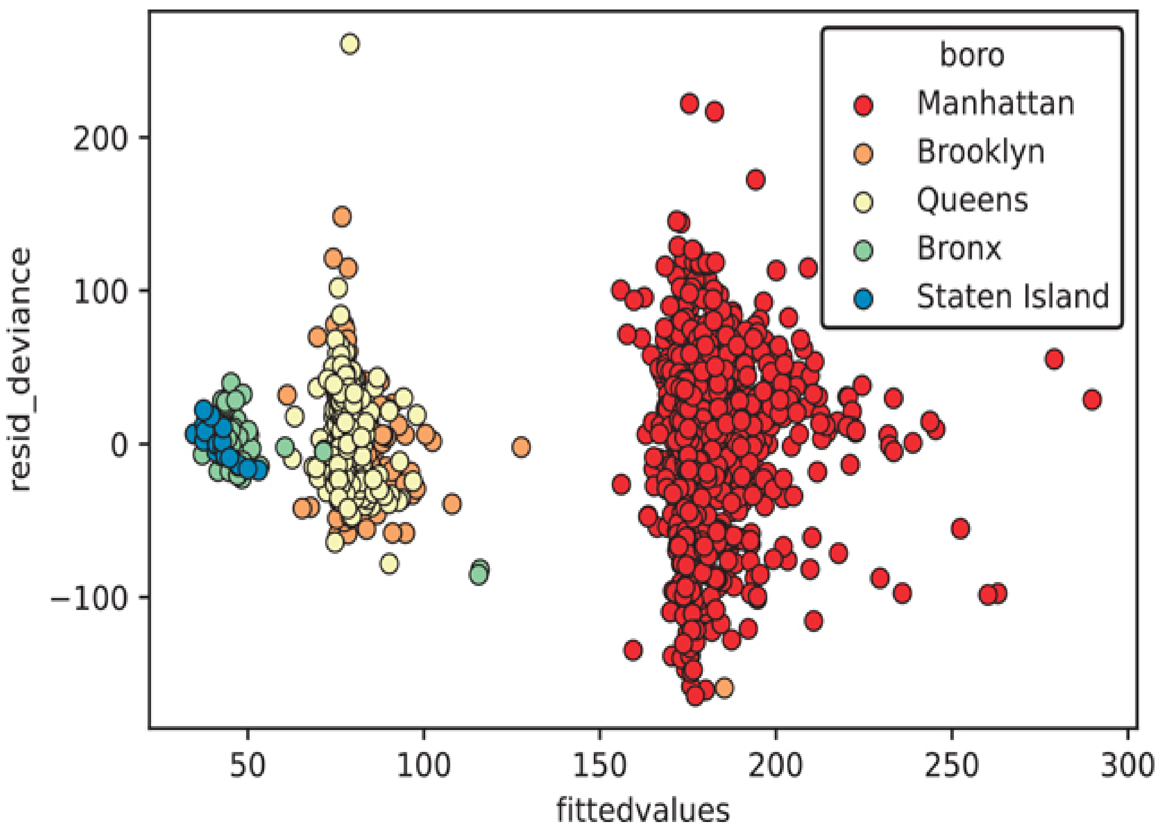


Figure 16.2 Residuals of the house1 model colored by `boro`

Click here to view code image

```
# get the data used for the residual plot and
boro color
res_df = pd.DataFrame(
  {
    "fittedvalues": house1.fittedvalues, # get a
model attribute
    "resid_deviance": house1.resid_deviance,
    "boro": housing["boro"], # get a value from
data column
```

```
    }
)

# greyscale friendly color palette
color_dict = dict(
    {
        "Manhattan": "#d7191c",
        "Brooklyn": "#fdae61",
        "Queens": "#ffffbf",
        "Bronx": "#abdda4",
        "Staten Island": "#2b83ba",
    }
)

fig, ax = plt.subplots()
fig = sns.scatterplot(
    x="fittedvalues",
    y="resid_deviance",
    data=res_df,
    hue="boro",
    ax=ax,
    palette=color_dict,
    edgecolor='black',
)

plt.show()
```

When we color our points based on `boro`, you can see that the clusters are highly governed by the value of this variable.

16.1.1 Q-Q Plots

A q-q plot is a graphical technique that determines whether your data conforms to a reference distribution. Since many models assume the data is normally distributed, a q-q plot is one way to make sure your data really is normal (Figure 16.3).

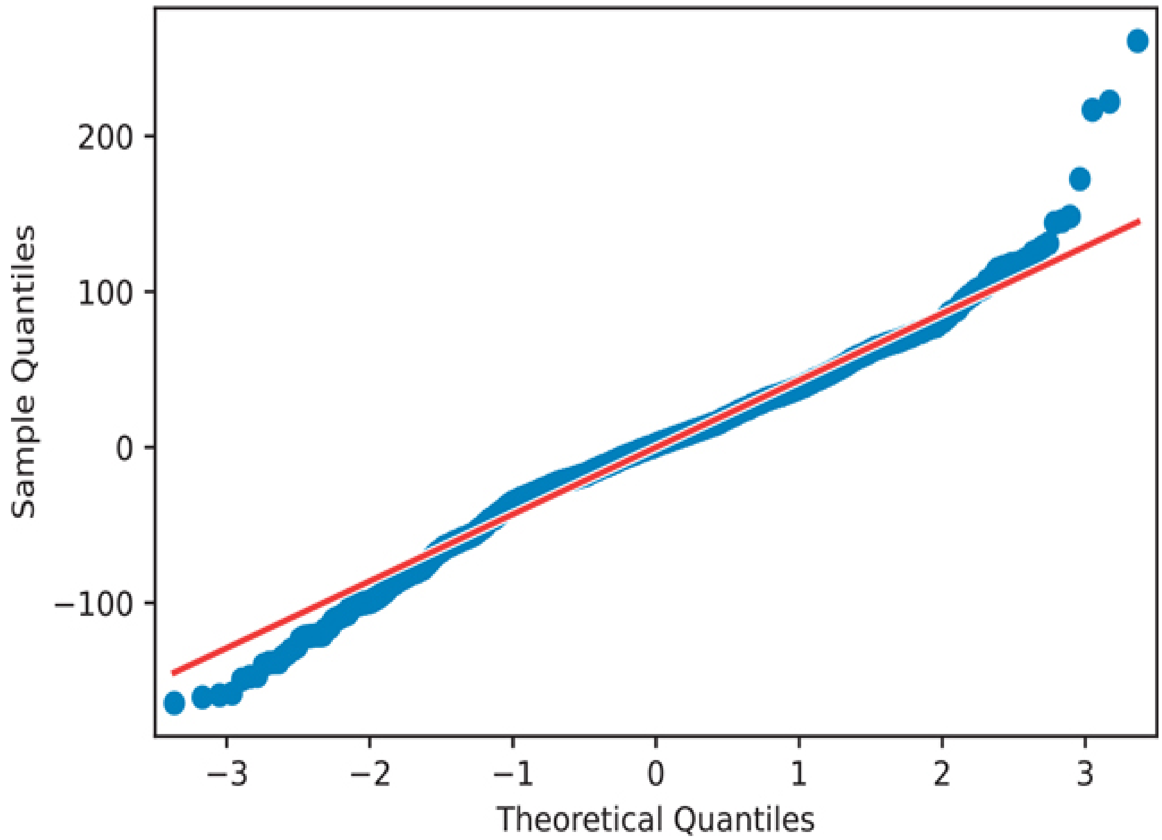


Figure 16.3 The q-q plot of the `house1` model

Click here to view code image

```
from scipy import stats

# make a copy of the variable so we don't need
to keep typing it
resid = house1.resid_deviance.copy()
```

```
fig =
statsmodels.graphics.gofplots.qqplot(resid,
line='r')
plt.show()
```

We can also plot a histogram of the residuals to see if our data is normal (Figure 16.4).

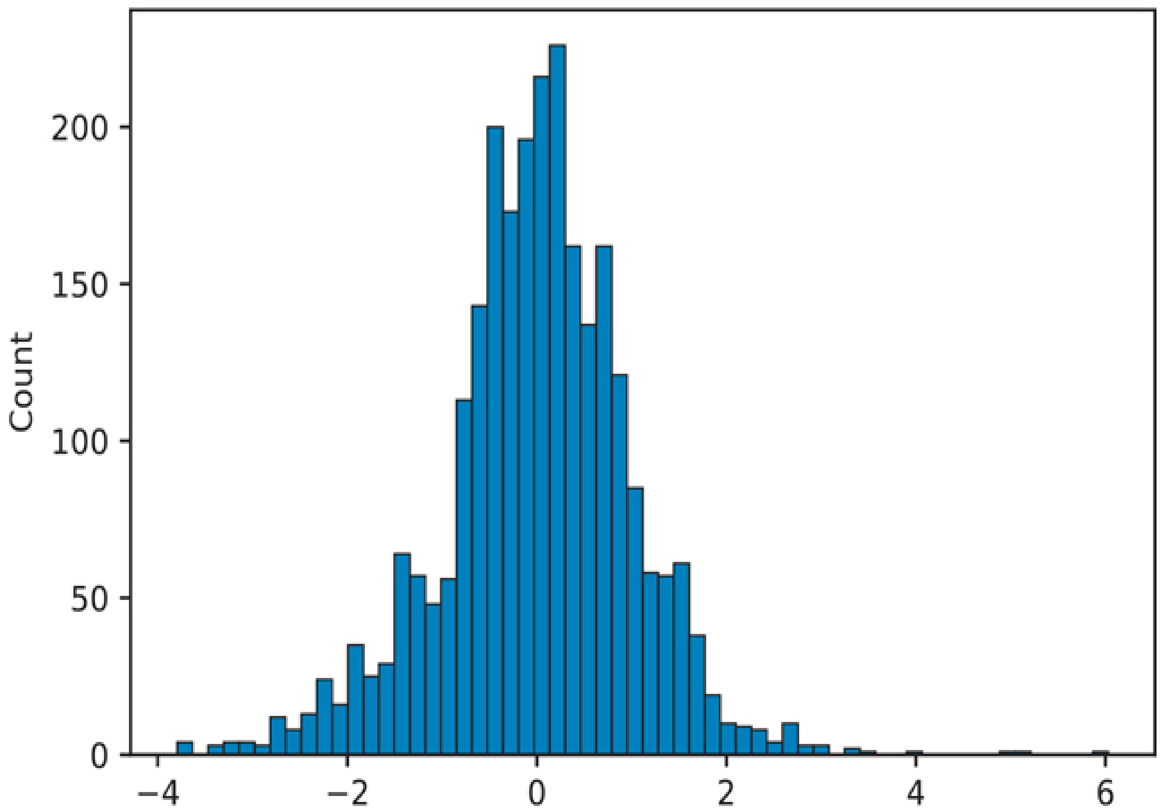


Figure 16.4 Histogram of the `house1` model residuals

Click here to view code image

```
resid_std = stats.zscore(resid)

fig, ax = plt.subplots()
sns.histplot(resid_std, ax=ax)
plt.show()
```

.

If the points on the q-q plot lie on the red line, that means our data match our reference distribution. If the points do not lie on this line, then one thing we can do is apply a transformation to our data. Table 16.1 shows which transformations can be performed on our data. If the q-q plot of points is convex compared to the red reference line, then you can transform your data toward the top of the table. If the q-q plot of points is concave compared to the red reference line, then you can transform your data toward the bottom of the table.

Table 16.1 Transformations

x^p	**Equivalent**	**Description**
x^2	x^2	Square
x^1	x	
$x^{\frac{1}{2}}$	$\sqrt{x}$	Square root
“x”x	$log(x)$	Log
$x^{\frac{-1}{2}}$	$\frac{1}{\sqrt{x}}$	Reciprocal square root
x^{-1}	$\frac{1}{x}$	Reciprocal
x^{-2}	$\frac{1}{x^2}$	Reciprocal square

16.2 Comparing Multiple Models

Now that we know how to assess a single model, we need a means to compare multiple models so that we can pick the “best” one.

16.2.1 Working with Linear Models

We begin by fitting five models. Note that some of the models use the + operator to add covariates to the model, whereas others use the `*` operator. To specify an interaction in our model, we use the `*` operator. That is, the variables that are interacting are behaving in a way that is *not* independent of one another, but in such a way that their values affect one another and are not simply additive.

> **Note**
>
> If the original housing data set had a column named `"class"`, this would cause an error because `"class"` is a Python keyword. Therefore, the column was renamed `"type"`.

Click here to view code image

```
f1 = 'value_per_sq_ft ~ units + sq_ft + boro'
f2 = 'value_per_sq_ft ~ units *  sq_ft + boro'
f3 = 'value_per_sq_ft ~ units + sq_ft * boro +
type'
f4 = 'value_per_sq_ft ~ units + sq_ft * boro +
sq_ft * type'
f5 = 'value_per_sq_ft ~ boro + type'

house1 = smf.ols(f1, data=housing).fit()
house2 = smf.ols(f2, data=housing).fit()
house3 = smf.ols(f3, data=housing).fit()
house4 = smf.ols(f4, data=housing).fit()
house5 = smf.ols(f5, data=housing).fit()
```

With all our models, we can collect all of our coefficients and the model with which they are associated.

Click here to view code image

```
mod_results = (
  pd.concat(
    [
      house1.params,
      house2.params,
```

```
        house3.params,
        house4.params,
        house5.params,
      ],
      axis=1,
    )
    .rename(columns=lambda x: "house" + str(x +
1))
    .reset_index()
    .rename(columns={"index": "param"})
    .melt(id_vars="param", var_name="model",
value_name="estimate")
)
```

```
print(mod_results)
```

```
                            param    model
estimate
0                       Intercept   house1
43.290863
1               boro[T.Brooklyn]   house1
34.562150
2              boro[T.Manhattan]   house1
130.992363
3                 boro[T.Queens]   house1
32.993674
4          boro[T.Staten Island]   house1
-3.630251
..                            ...      ...
...
85          sq_ft:boro[T.Queens]   house5
NaN
86   sq_ft:boro[T.Staten Island]   house5
```

```
NaN
87   sq_ft:type[T.R4-CONDOMINIUM]   house5
NaN
88   sq_ft:type[T.R9-CONDOMINIUM]   house5
NaN
89   sq_ft:type[T.RR-CONDOMINIUM]   house5
NaN

[90 rows x 3 columns]
```

Since it's not very useful to look at a large column of values, we can plot our coefficients to quickly see how the models are estimating parameters in relation to each other (Figure 16.5).

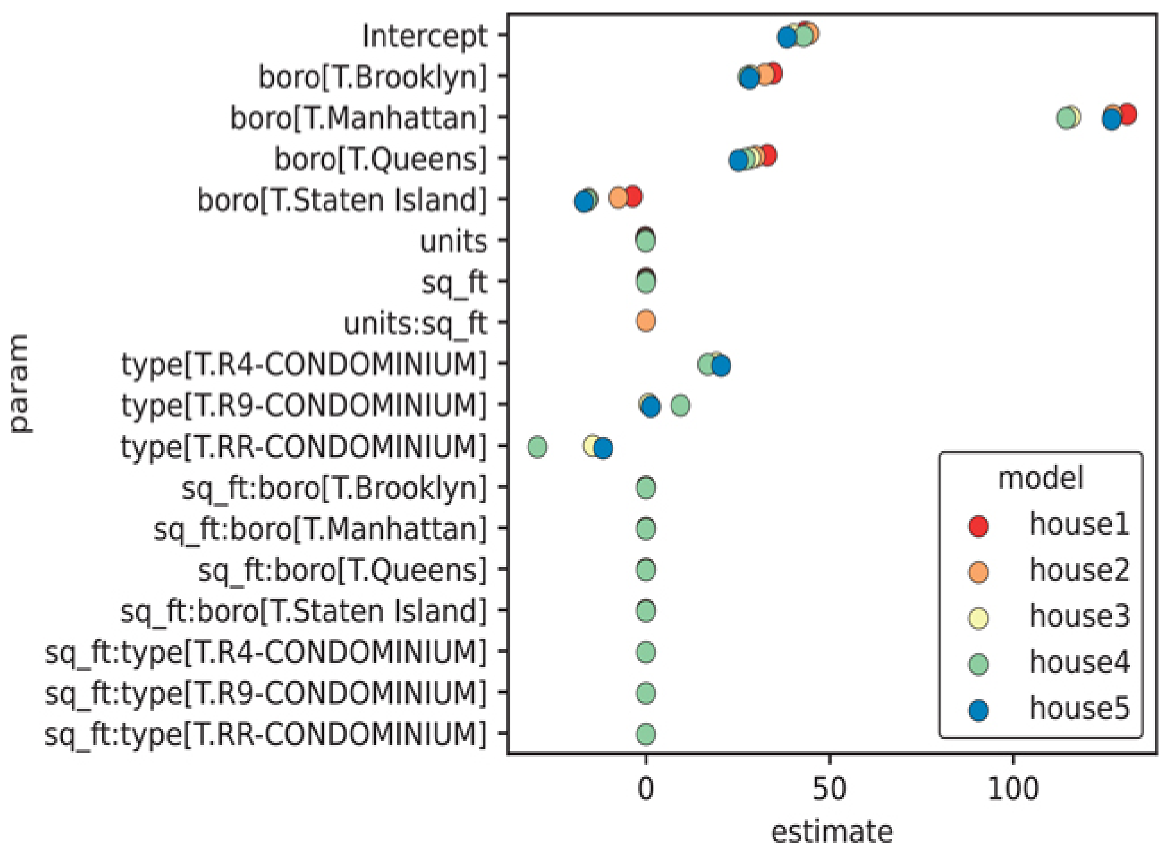


Figure 16.5 Coefficients of the `house1` to `house5` models

Click here to view code image

```
color_dict = dict(
  {
    "house1": "#d7191c",
    "house2": "#fdae61",
    "house3": "#ffffbf",
    "house4": "#abdda4",
    "house5": "#2b83ba",
  }
)
```

Click here to view code image

```
fig, ax = plt.subplots()
ax = sns.pointplot(
  x="estimate",
  y="param",
  hue="model",
  data=mod_results,
  dodge=True, # jitter the points
  join=False, # don't connect the points
  palette=color_dict
)

plt.tight_layout()
plt.show()
```

Now that we have our linear models, we can use the analysis of variance (ANOVA) method to compare them. The ANOVA will give us the residual sum of squares (RSS), which is one way we can measure performance (lower is better).

Click here to view code image

```
model_names = ["house1", "house2", "house3",
"house4", "house5"]
house_anova = statsmodels.stats.anova.anova_lm(
  house1, house2, house3, house4, house5
)

house_anova.index = model_names

print(house_anova)
```

Click here to view code image

```
         df_resid           ssr  df_diff
ss_diff          F  \
house1     2619.0  4.922389e+06      0.0
NaN            NaN
house2     2618.0  4.884872e+06      1.0
37517.437605    20.039049
house3     2612.0  4.619926e+06      6.0
264945.539994    23.585728
house4     2609.0  4.576671e+06      3.0
43255.441192     7.701289
house5     2618.0  4.901463e+06     -9.0
-324791.847907    19.275539

               Pr(>F)
house1            NaN
house2   7.912333e-06
house3   2.754431e-27
house4   4.025581e-05
house5            NaN
```

Another way we can calculate model performance is by using the Akaike information criterion (AIC) and the Bayesian information criterion (BIC). These methods apply a penalty for each feature that is added to the model (lower AIC and BIC value is better). Thus, we should strive to balance performance and parsimony.

Click here to view code image

```
house_models = [house1, house2, house3, house4,
house5]
```

```
abic = pd.DataFrame(
  {
    "model": model_names,
    "aic": [mod.aic for mod in house_models],
    "bic": [mod.bic for mod in house_models],
  }
)

print(abic.sort_values(by=["aic", "bic"]))
```

```
    model           aic           bic
3  house4  27084.800043  27184.644733
2  house3  27103.502577  27185.727615
1  house2  27237.939618  27284.925354
4  house5  27246.843392  27293.829128
0  house1  27256.031113  27297.143632
```

16.2.2 Working with GLM Models

We can perform the same calculations and model diagnostics on generalized linear models (GLMs). We can use the deviance of the model to do model comparisons:

Click here to view code image

```
def deviance_table( *models):
  """Create a table of model diagnostics from
model objects"""
  return pd.DataFrame(
    {
      "df_residuals": [mod.df_resid for mod in
models],
      "resid_stddev": [mod.deviance for mod in
models],
      "df": [mod.df_model for mod in models],
      "deviance": [mod.deviance for mod in
models],
    }
  )
```

Click here to view code image

```
f1 = 'value_per_sq_ft ~ units + sq_ft + boro'
f2 = 'value_per_sq_ft ~ units * sq_ft + boro'
f3 = 'value_per_sq_ft ~ units + sq_ft * boro +
type'
f4 = 'value_per_sq_ft ~ units + sq_ft * boro +
sq_ft * type'
f5 = 'value_per_sq_ft ~ boro + type'

glm1 = smf.glm(f1, data=housing).fit()
glm2 = smf.glm(f2, data=housing).fit()
glm3 = smf.glm(f3, data=housing).fit()
glm4 = smf.glm(f4, data=housing).fit()
glm5 = smf.glm(f5, data=housing).fit()

glm_anova = deviance_table(glm1, glm2, glm3,
```

```
glm4, glm5)
print(glm_anova)
```

```
   df_residuals  resid_stddev    df        deviance
0          2619  4.922389e+06     6    4.922389e+06
1          2618  4.884872e+06     7    4.884872e+06
2          2612  4.619926e+06    13    4.619926e+06
3          2609  4.576671e+06    16    4.576671e+06
4          2618  4.901463e+06     7    4.901463e+06
```

We can do the same set of calculations in a logistic regression.

Click here to view code image

```
# create a binary variable
housing["high"] = (housing["value_per_sq_ft"] >=
150).astype(int)

print(housing["high"].value_counts())
```

```
0      1619
1      1007
Name: high, dtype: int64
```

```
# create and fit our logistic regression using
GLM

f1 = "high ~ units + sq_ft + boro"
f2 = "high ~ units * sq_ft + boro"
f3 = "high ~ units + sq_ft * boro + type"
f4 = "high ~ units + sq_ft * boro + sq_ft *
type"
f5 = "high ~ boro + type"
```

Click here to view code image

```
logistic =
statsmodels.genmod.families.family.Binomial(

link=statsmodels.genmod.families.links.Logit()
)

glm1 = smf.glm(f1, data=housing,
family=logistic).fit()
glm2 = smf.glm(f2, data=housing,
family=logistic).fit()
glm3 = smf.glm(f3, data=housing,
family=logistic).fit()
glm4 = smf.glm(f4, data=housing,
family=logistic).fit()
glm5 = smf.glm(f5, data=housing,
family=logistic).fit()
```

```
# show the deviances from our GLM models
print(deviance_table(glm1, glm2, glm3, glm4,
glm5))
```

```
   df_residuals  resid_stddev  df     deviance
0          2619   1695.631547   6  1695.631547
1          2618   1686.126740   7  1686.126740
2          2612   1636.492830  13  1636.492830
3          2609   1619.431515  16  1619.431515
4          2618   1666.615696   7  1666.615696
```

Finally, we can create a table of AIC and BIC values.

Click here to view code image

```
mods = [glm1, glm2, glm3, glm4, glm5]

abic_glm = pd.DataFrame(
  {
    "model": model_names,
    "aic": [mod.aic for mod in house_models],
    "bic": [mod.bic for mod in house_models],
  }
)

print(abic_glm.sort_values(by=["aic", "bic"]))
```

```
    model           aic           bic
3  house4  27084.800043  27184.644733
2  house3  27103.502577  27185.727615
1  house2  27237.939618  27284.925354
4  house5  27246.843392  27293.829128
0  house1  27256.031113  27297.143632
```

Looking at all these measures, we can say Model 4 is performing the best so far.

16.3 *k*-Fold Cross-Validation

Cross-validation is another technique to compare models. One of the main benefits is that it can account for how well your model performs on new data. It does this by partitioning your data into k parts. It holds one of the parts aside as the "test" set and then fits the model on the remaining $k - 1$ parts, the "training" set. The fitted model is then used on the "test" and an error rate is calculated. This process is repeated until all k parts have been used as a "test" set. The final error of the model is some average across all the models.

Cross-validation can be performed in many different ways. The method just described is called "*k*-fold cross-validation." Alternative ways of performing cross-validation include “leave-one-out cross-validation”, in which the training data consists of all the data except one observation designated as the test set.

Here we will split our data into $k - 1$ testing and training data sets.

Click here to view code image

```
from sklearn.model_selection import
train_test_split
from sklearn.linear_model import
LinearRegression

print(housing.columns)
```

```
Index(['neighborhood', 'type', 'units',
'year_built', 'sq_ft',
       'income', 'income_per_sq_ft', 'expense',
'expense_per_sq_ft',
       'net_income', 'value', 'value_per_sq_ft',
'boro', 'high'],
      dtype='object')
```

```
# get training and test data
X_train, X_test, y_train, y_test =
train_test_split(
    pd.get_dummies(
        housing[["units", "sq_ft", "boro"]],
drop_first=True
    ),

    housing["value_per_sq_ft"],
    test_size=0.20,
    random_state=42,
)
```

Danger

Pay attention to the capitalization of the letter `X` when looking at scikit-learn tutorials and documentation. This is a convention that comes from matrix notation from statistics and mathematics.

We can get a score that indicates how well our model is performing using our test data.

Click here to view code image

```
lr = LinearRegression().fit(X_train, y_train)
print(lr.score(X_test, y_test))
```

```
0.6137125285030868
```

Since `sklearn` relies heavily on the `numpy ndarray`, the `patsy` library allows you to specify a formula just like the formula API in `statsmodels`, and it returns a proper `numpy` array you can use in `sklearn`.

Here is the same code as before, but using the `dmatrices` function in the `patsy` library.

Click here to view code image

```
from patsy import dmatrices

y, X = dmatrices(
    "value_per_sq_ft ~ units + sq_ft + boro",
    housing,
    return_type="dataframe",
)
X_train, X_test, y_train, y_test =
train_test_split(
  X, y, test_size=0.20, random_state=42
```

```
)

lr = LinearRegression().fit(X_train, y_train)
print(lr.score(X_test, y_test))
```

```
0.6137125285030818
```

To perform a *k*-fold cross-validation, we need to import this function from `sklearn`.

Click here to view code image

```
from sklearn.model_selection import KFold,
cross_val_score

# get a fresh new housing data set
housing =
pd.read_csv('data/housing_renamed.csv')
```

We now have to specify how many folds we want. This number depends on how many rows of data you have. If your data does not include too many observations, you may opt to select a smaller *k* (e.g., 2). Otherwise, a *k* between 5 to 10 is fairly common. However, keep in mind that the trade-off with higher *k* values is more computation time.

Click here to view code image

```
kf = KFold(n_splits=5)

y, X = dmatrices('value_per_sq_ft ~ units +
sq_ft + boro', housing)
```

Next we can train and test our model on each fold.

Click here to view code image

```
coefs = []
scores = []
for train, test in kf.split(X):
  X_train, X_test = X[train], X[test]
  y_train, y_test = y[train], y[test]
  lr = LinearRegression().fit(X_train, y_train)
  coefs.append(pd.DataFrame(lr.coef_))
  scores.append(lr.score(X_test, y_test))
```

We can also view the results.

Click here to view code image

```
coefs_df = pd.concat(coefs)
coefs_df.columns = X.design_info.column_names
print(coefs_df)
```

Click here to view code image

```
     Intercept  boro[T.Brooklyn]
boro[T.Manhattan]  boro[T.Queens]  \
0          0.0         33.369037
129.904011       32.103100
0          0.0         32.889925
116.957385       31.295956
0          0.0         30.975560
141.859327       32.043449
0          0.0         41.449196
130.779013       33.050968
0          0.0        -38.511915
56.069855      -17.557939

   boro[T.Staten Island]      units     sq_ft
0          -4.381085e+00  -0.205890  0.000220
```

```
0           -4.919232e+00  -0.146180  0.000155
0           -4.379916e+00  -0.179671  0.000194
0           -3.430209e+00  -0.207904  0.000232
0            3.552714e-15  -0.145829  0.000202
```

We can take a look at the average coefficient across all folds using `.apply()` and the `np.mean()` function.

Click here to view code image

```
import numpy as np
print(coefs_df.apply(np.mean))
```

```
Intercept                 0.000000
boro[T.Brooklyn]         20.034361
boro[T.Manhattan]       115.113918
boro[T.Queens]           22.187107
boro[T.Staten Island]    -3.422088
units                    -0.177095
sq_ft                     0.000201
dtype: float64
```

We can also look at our scores. Each model has a default scoring method. `LinearRegression()`, for example, uses the R^2 (coefficient of determination) regression score function.[1]

Click here to view code image

```
print(scores)
```

```
[0.02731416291043942, -0.5538362212110504,
-0.1563637168806138,
-0.3234202061929452, -1.6929655586752923]
```

We can also use `cross_val_scores` (for cross-validation scores) to calculate our scores.

Click here to view code image

```
# use cross_val_scores to calculate CV scores
model = LinearRegression()
scores = cross_val_score(model, X, y, cv=5)
print(scores)
```

```
[ 0.02731416 -0.55383622 -0.15636372 -0.32342021
-1.69296556]
```

1. Scikit-learn R^2 scoring: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html

When we compare multiple models to one another, we compare the average of the scores.

Click here to view code image

```
print(scores.mean())
```

```
-0.5398543080098925
```

Now we'll refit all our models using *k*-fold cross-validation.

Click here to view code image

```
# create the predictor and response matrices
y1, X1 = dmatrices(
   "value_per_sq_ft ~ units + sq_ft + boro",
housing)

y2, X2 = dmatrices("value_per_sq_ft ~
units*sq_ft + boro", housing)
```

```
y3, X3 = dmatrices(
  "value_per_sq_ft ~ units + sq_ft*boro + type",
housing
)

y4, X4 = dmatrices(
    "value_per_sq_ft ~ units + sq_ft*boro +
sq_ft*type", housing
)

y5, X5 = dmatrices("value_per_sq_ft ~ boro +
type", housing)
```

```
# fit our models
model = LinearRegression()

scores1 = cross_val_score(model, X1, y1, cv=5)
scores2 = cross_val_score(model, X2, y2, cv=5)
scores3 = cross_val_score(model, X3, y3, cv=5)
scores4 = cross_val_score(model, X4, y4, cv=5)
scores5 = cross_val_score(model, X5, y5, cv=5)
```

We can now look at our cross-validation scores.

Click here to view code image

```
scores_df = pd.DataFrame(
    [scores1, scores2, scores3, scores4,
scores5]
)

print(scores_df.apply(np.mean, axis=1))
```

```
0   -5.398543e-01
1   -1.088184e+00
2   -8.668885e+25
3   -7.634198e+25
4   -3.172546e+25
dtype: float64
```

Once again, we see that Model 4 has the best performance.

Conclusion

When we are working with models, it's important to measure their performance. Using ANOVA for linear models, looking at deviance for GLM models, and using cross-validation are all ways we can measure error and performance when trying to pick the best model.

17

Regularization

In Chapter 16, we considered various ways to measure model performance. Section 16.3 described *k*-fold cross-validation, a technique that tries to measure model performance by looking at how it predicts on test data. This chapter explores regularization, one technique to improve performance on test data. Specifically, this method aims to prevent overfitting.

17.1 Why Regularize?

Let's begin with a base case of linear regression. We will be using the ACS data.

Click here to view code image

```
import pandas as pd
acs = pd.read_csv('data/acs_ny.csv')
print(acs.columns)
```

```
Index(['Acres', 'FamilyIncome', 'FamilyType',
'NumBedrooms',
       'NumChildren', 'NumPeople', 'NumRooms',
'NumUnits',
       'NumVehicles', 'NumWorkers', 'OwnRent',
'YearBuilt',
       'HouseCosts', 'ElectricBill', 'FoodStamp',
'HeatingFuel',
```

```
       'Insurance', 'Language'],
      dtype='object')
```

Now, let's create our design matrices using `patsy`.

Click here to view code image

```
from patsy import dmatrices

# sequential strings get concatenated together
in Python
response, predictors = dmatrices(
  "FamilyIncome ~ NumBedrooms + NumChildren +
NumPeople + "
  "NumRooms + NumUnits + NumVehicles +
NumWorkers + OwnRent + "
  "YearBuilt + ElectricBill + FoodStamp +
HeatingFuel + "
  "Insurance + Language",
  data=acs,
)
```

With our predictor and response matrices created, we can use `sklearn` to split our data into training and testing sets.

Click here to view code image

```
from sklearn.model_selection import
train_test_split

X_train, X_test, y_train, y_test =
train_test_split(
  predictors, response, random_state=0
)
```

Now, let's fit our linear model. Here we are normalizing our data so we can compare our coefficients when we use our regularization techniques.

Click here to view code image

```
from sklearn.linear_model import
LinearRegression
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

lr = make_pipeline(
  StandardScaler(with_mean=False),
LinearRegression()
)

lr = lr.fit(X_train, y_train)
print(lr)
```

```
Pipeline(steps=[('standardscaler',
StandardScaler(with_mean=False)),
                ('linearregression',
LinearRegression())])
```

```
model_coefs = pd.DataFrame(
  data=list(
    zip(
      predictors.design_info.column_names,

lr.named_steps["linearregression"].coef_[0],
    )
  ),
  columns=["variable", "coef_lr"],
)
```

```
print(model_coefs)
```

```
                             variable       coef_lr
0                           Intercept  2.697159e-13
1   NumUnits[T.Single attached]  9.661755e+03
2   NumUnits[T.Single detached]  8.345408e+03
3            OwnRent[T.Outright]  2.382740e+03
4              OwnRent[T.Rented]  2.260806e+03
..                            ...           ...
34                       NumRooms  1.340575e+04
35                    NumVehicles  7.228920e+03
36                     NumWorkers  1.877535e+04
37                   ElectricBill  1.000008e+04
38                      Insurance  3.072892e+04

[39 rows x 2 columns]
```

Now, we can look at our model scores.

Click here to view code image

```
# score on the _training_ data
print(lr.score(X_train, y_train))
```

```
0.272614046563856
```

```
# score on the _testing_ data
print(lr.score(X_test, y_test))
```

```
0.2697697956848801
```

In this particular case, our model demonstrates poor performance. In another potential scenario, we might have a high training score and a low test score—a sign of overfitting. Regularization solves this overfitting issue, by putting constraints on the coefficients and variables. This causes the coefficients of our data to be smaller. In the case of LASSO (least absolute shrinkage and selection operator) regression, some coefficients can actually be dropped (i.e., become 0), whereas in ridge regression, coefficients will approach 0, but are never dropped.

17.2 LASSO Regression

The first type of regularization technique is called LASSO, which stands for least absolute shrinkage and selection operator. It is also known as regression with L1 regularization.

We will fit the same model as we did in our linear regression.

Click here to view code image

```
from sklearn.linear_model import Lasso

lasso = make_pipeline(
  StandardScaler(with_mean=False),
  Lasso(max_iter=10000, random_state=42),
)

lasso = lasso.fit(X_test, y_test)
print(lasso)
```

```
Pipeline(steps=[('standardscaler',
StandardScaler(with_mean=False)),
                ('lasso', Lasso(max_iter=10000,
random_state=42))])
```

Now, let's get a dataframe of coefficients, and combine them with our linear regression results.

Click here to view code image

```
coefs_lasso = pd.DataFrame(
  data=list(
    zip(
      predictors.design_info.column_names,
      lasso.named_steps["lasso"].coef_.tolist(),
    )
  ),
  columns=["variable", "coef_lasso"],
)
```

Click here to view code image

```
model_coefs = pd.merge(model_coefs, coefs_lasso,
on='variable')
print(model_coefs)
```

```
                      variable       coef_lr
coef_lasso
0                    Intercept  2.697159e-13
0.000000
1   NumUnits[T.Single attached]  9.661755e+03
7765.482025
2   NumUnits[T.Single detached]  8.345408e+03
7512.067593
3           OwnRent[T.Outright]  2.382740e+03
2431.710977
4             OwnRent[T.Rented]  2.260806e+03
604.186925
..                          ...           ...
...
34                    NumRooms  1.340575e+04
10940.150208
35                 NumVehicles  7.228920e+03
```

```
7724.681161
36                    NumWorkers  1.877535e+04
16911.035390
37                  ElectricBill  1.000008e+04
9516.123582
38                     Insurance  3.072892e+04
32155.544169

[39 rows x 3 columns]
```

Notice that the coefficients are now smaller than their original linear regression values. Additionally, some of the coefficients are now `0`.

Finally, let's look at our training and test data scores.

Click here to view code image

```
print(lasso.score(X_train, y_train))
```

```
0.2669751487716776
```

```
print(lasso.score(X_test, y_test))
```

```
0.2752627973740016
```

There isn't much difference here, but you can see that the test results are now better than the training results. That is, there is an improvement in prediction when using new, unseen data.

17.3 Ridge Regression

Now let's look at another regularization technique, ridge regression. It is also known as regression with L2 regularization.

Most of the code will be very similar to that seen with the previous methods. We will fit the model on our training data, and combine the results with our ongoing dataframe of results.

Click here to view code image

```
from sklearn.linear_model import Ridge

ridge = make_pipeline(
    StandardScaler(with_mean=False),
Ridge(random_state=42)
)

ridge = ridge.fit(X_train, y_train)
print(ridge)
```

```
Pipeline(steps=[('standardscaler',
StandardScaler(with_mean=False)),
                ('ridge',
Ridge(random_state=42))])
```

```
coefs_ridge = pd.DataFrame(
  data=list(
    zip(
      predictors.design_info.column_names,
      ridge.named_steps["ridge"].coef_.tolist()
[0],
    )
  ),
  columns=["variable", "coef_ridge"],
)

model_coefs = pd.merge(model_coefs, coefs_ridge,
on="variable")
print(model_coefs)
```

```
                       variable       coef_lr
coef_lasso \
0                     Intercept  2.697159e-13
0.000000
1   NumUnits[T.Single attached]  9.661755e+03
7765.482025
2   NumUnits[T.Single detached]  8.345408e+03
7512.067593
3           OwnRent[T.Outright]  2.382740e+03
2431.710977
4             OwnRent[T.Rented]  2.260806e+03
604.186925
..                          ...           ...
...
34                     NumRooms  1.340575e+04
10940.150208
35                  NumVehicles  7.228920e+03
7724.681161
36                   NumWorkers  1.877535e+04
16911.035390
37                 ElectricBill  1.000008e+04
9516.123582
38                    Insurance  3.072892e+04
32155.544169

      coef_ridge
0       0.000000
1    9659.413514
2    8342.247690
3    2381.429615
4    2259.526329
..           ...
34  13405.409584
```

```
35    7228.542922
36   18773.079462
37   10000.853603
38   30727.230542

[39 rows x 4 columns]
```

17.4 Elastic Net

The elastic net is a regularization technique that combines the ridge and LASSO regression techniques.

Click here to view code image

```
from sklearn.linear_model import ElasticNet

en = ElasticNet(random_state=42).fit(X_train,
y_train)

coefs_en = pd.DataFrame(

list(zip(predictors.design_info.column_names,
en.coef_)),
    columns=["variable", "coef_en"],
)

model_coefs = pd.merge(model_coefs, coefs_en,
on="variable")
print(model_coefs)
```

```
                      variable       coef_lr
coef_lasso \
0                    Intercept  2.697159e-13
0.000000
```

```
1   NumUnits[T.Single attached]  9.661755e+03
7765.482025
2   NumUnits[T.Single detached]  8.345408e+03
7512.067593
3           OwnRent[T.Outright]  2.382740e+03
2431.710977
4             OwnRent[T.Rented]  2.260806e+03
604.186925
..                          ...           ...
...
34                     NumRooms  1.340575e+04
10940.150208
35                  NumVehicles  7.228920e+03
7724.681161
36                   NumWorkers  1.877535e+04
16911.035390
37                 ElectricBill  1.000008e+04
9516.123582
38                    Insurance  3.072892e+04
32155.544169

      coef_ridge       coef_en
0       0.000000      0.000000
1    9659.413514   1342.291706
2    8342.247690    168.728479
3    2381.429615    445.533238
4    2259.526329   -600.673747
..           ...           ...
34  13405.409584   5685.101939
35   7228.542922   6059.776166
36  18773.079462  12247.547800
37  10000.853603     97.566664
38  30727.230542     32.484207
```

```
[39 rows x 5 columns]
```

The `ElasticNet` object has two parameters, `alpha` and `l1_ratio`, that allow you to control the behavior of the model. The `l1_ratio` parameter specifically controls how much of the L2 or L1 penalty is used. If `l1_ratio = 0`, then the model will behave as described by ridge regression. If `l1_ratio = 1`, then the model will behave as described by LASSO regression. Any value in between will give some combination of the ridge and LASSO regression results.

Since LASSO regression can zero out coefficients, let's just see how the coefficients compare with just the variables where LASSO has turned into a `0`.

Click here to view code image

```
print(model_coefs.loc[model_coefs["coef_lasso"]
== 0])
```

```
                variable        coef_lr  coef_lasso
coef_ridge \
0              Intercept   2.697159e-13         0.0
0.000000
25   HeatingFuel[T.Solar]   1.442204e+02         0.0
142.354045

     coef_en
0   0.000000
25  0.994142
```

17.5 Cross-Validation

Cross-validation (first described in Section 16.3) is a commonly used technique when fitting models. It was mentioned at the beginning of this chapter, as a segue to regularization, but it is also a way to pick optimal parameters for regularization. Since the user must tune certain parameters (also known as hyper-parameters), cross-validation can be used to try out various combinations of these hyper-parameters to pick the "best" model. The `ElasticNet` object has a similar function called `ElasticNetCV` that can iteratively fit the elastic net with various hyper-parameter values.[1]

1. `ElasticNetCV` documentation: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNetCV.html

Click here to view code image

```
from sklearn.linear_model import ElasticNetCV

en_cv = ElasticNetCV(cv=5, random_state=42).fit(
    X_train, y_train.ravel() # ravel is to
remove the 1d warning
)

coefs_en_cv = pd.DataFrame(

list(zip(predictors.design_info.column_names,
en_cv.coef_)),
    columns=["variable", "coef_en_cv"],
)

model_coefs = pd.merge(model_coefs, coefs_en_cv,
on="variable")
print(model_coefs)
```

Click here to view code image

```
                      variable       coef_lr
coef_lasso \
```

```
0                      Intercept  2.697159e-13
0.000000
1   NumUnits[T.Single attached]  9.661755e+03
7765.482025
2   NumUnits[T.Single detached]  8.345408e+03
7512.067593
3           OwnRent[T.Outright]  2.382740e+03
2431.710977
4             OwnRent[T.Rented]  2.260806e+03
604.186925
..                          ...           ...
...
34                     NumRooms  1.340575e+04
10940.150208
35                  NumVehicles  7.228920e+03
7724.681161
36                   NumWorkers  1.877535e+04
16911.035390
37                 ElectricBill  1.000008e+04
9516.123582
38                    Insurance  3.072892e+04
32155.544169

      coef_ridge       coef_en  coef_en_cv
0       0.000000      0.000000    0.000000
1    9659.413514   1342.291706   -0.000000
2    8342.247690    168.728479    0.000000
3    2381.429615    445.533238    0.000000
4    2259.526329   -600.673747   -0.000000
..           ...           ...         ...
34  13405.409584   5685.101939    0.028443
35   7228.542922   6059.776166    0.000000
36  18773.079462  12247.547800    0.000000
```

```
37  10000.853603     97.566664   26.166320
38  30727.230542     32.484207   38.561748

[39 rows x 6 columns]
```

Let's compare which coefficients were turned into `0`.

Click here to view code image

```
print(model_coefs.loc[model_coefs["coef_en_cv"]
== 0])
```

```
                       variable       coef_lr
coef_lasso \
0                     Intercept  2.697159e-13
0.000000
1   NumUnits[T.Single attached]  9.661755e+03
7765.482025
2   NumUnits[T.Single detached]  8.345408e+03
7512.067593
3           OwnRent[T.Outright]  2.382740e+03
2431.710977
4             OwnRent[T.Rented]  2.260806e+03
604.186925
..                          ...           ...
...
31                  NumBedrooms  3.755708e+03
4447.892458
32                  NumChildren  9.524915e+03
6905.672216
33                    NumPeople -1.153672e+04
-8777.265840
35                  NumVehicles  7.228920e+03
7724.681161
```

```
36                   NumWorkers  1.877535e+04
16911.035390

       coef_ridge       coef_en   coef_en_cv
0        0.000000      0.000000          0.0
1     9659.413514   1342.291706         -0.0
2     8342.247690    168.728479          0.0
3     2381.429615    445.533238          0.0
4     2259.526329   -600.673747         -0.0
..            ...           ...          ...
31    3755.521256   2073.910045          0.0
32    9521.180875   2498.719581          0.0
33 -11533.098634  -2562.412933          0.0
35    7228.542922   6059.776166          0.0
36   18773.079462  12247.547800          0.0

[36 rows x 6 columns]
```

Conclusion

Regularization is a technique used to prevent overfitting of data. It achieves this goal by applying some penalty for each feature added to the model. The end result either drops variables from the model or decreases the coefficients of the model. Both techniques try to fit the training data less accurately but hope to provide better predictions with data that has not been seen before. These techniques can be combined (as seen in the elastic net), and can also be iterated over and improved with cross-validation.

18

Clustering

Machine learning methods can generally be classified into two main categories of models: supervised learning and unsupervised learning. Thus far, we have been working on supervised learning models, since we train our models with a target *y* or response variable. In other words, in the training data for our models, we know the "correct" answer. Unsupervised models are modeling techniques in which the "correct" answer is unknown. Many of these methods involve clustering, where the two main methods are *k*-means clustering and hierarchical clustering.

18.1 *k*-Means

The technique known as *k*-means works by first selecting how many clusters, *k*, exist in the data. The algorithm randomly selects *k* points in the data and calculates the distance from every data point to the initially selected *k* points. The closest points to each of the *k* clusters are assigned to the same cluster group. The center of each cluster is then designated as the new cluster centroid. The process is then repeated, with the distance of each point to each cluster centroid being calculated and assigned to a cluster and a new centroid picked. This algorithm is repeated until convergence occurs.

Great visualizations[1] and explanations[2] of how *k*-means works can be found on the Internet. We'll use data about wines for our *k*-means example.

1. Visualizing *k*-means: `http://shabal.in/visuals.html`

2. Visualization and explanation of *k*-means: `https://www.naftaliharris.com/blog/visualizing-k-means-clustering/`

Click here to view code image

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

We will drop the `Cultivar` column since it correlates too closely with the actual clusters in our data.

Click here to view code image

```
wine = wine.drop('Cultivar', axis=1)

# note that the data values are all numeric
print(wine.columns)
```

Click here to view code image

```
Index(['Alcohol', 'Malic acid', 'Ash', 'Alcalinity
of ash ',
        'Magnesium', 'Total phenols',
'Flavanoids',
        'Nonflavanoid phenols', 'Proanthocyanins',
'Color intensity',
        'Hue', 'OD280/OD315 of diluted wines',
'Proline '],
      dtype='object')
```

```
print(wine.head())
```

```
   Alcohol  Malic acid   Ash  Alcalinity of ash
Magnesium \
0    14.23        1.71  2.43               15.6
127
1    13.20        1.78  2.14               11.2
100
```

```
2    13.16        2.36  2.67              18.6
101
3    14.37        1.95  2.50              16.8
113
4    13.24        2.59  2.87              21.0
118

   Total phenols  Flavanoids  Nonflavanoid phenols
Proanthocyanins \
0           2.80        3.06                  0.28
2.29
1           2.65        2.76                  0.26
1.28
2           2.80        3.24                  0.30
2.81
3           3.85        3.49                  0.24
2.18
4           2.80        2.69                  0.39
1.82

   Color intensity   Hue  OD280/OD315 of diluted
wines \
0             5.64  1.04
3.92
1             4.38  1.05
3.40
2             5.68  1.03
3.17
3             7.80  0.86
3.45
4             4.32  1.04
2.93
```

```
   Proline
0                 1065
1                 1050
2                 1185
3                 1480
4                  735
```

`sklearn` has an implementation of the *k*-means algorithm called `KMeans`. Here we will set $k = 3$, and use all the data in our data set.

We will create `k=3` clusters with a random seed of `42`. You can opt to leave out the random_state parameter or use a different value; the 42 will ensure your results are the same as those printed in the book.

Click here to view code image

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3,
random_state=42).fit(wine.values)
```

Here's our `kmeans` object.

Click here to view code image

```
print(kmeans)
```

```
KMeans(n_clusters=3, random_state=42)
```

We can see that since we specified three clusters, there are only three unique labels.

Click here to view code image

```
import numpy as np
print(np.unique(kmeans.labels_,
return_counts=True))
```

```
(array([0, 1, 2], dtype=int32), array([69, 47,
62]))
```

```
kmeans_3 = pd.DataFrame(kmeans.labels_, columns=
['cluster'])
print(kmeans_3)
```

```
     cluster
0          1
1          1
2          1
3          1
4          2
..       ...
173        2
174        2
175        2
176        2
177        0

[178 rows x 1 columns]
```

Finally, we can visualize our clusters. Since humans can visualize things in only three dimensions, we need to reduce the number of dimensions for our data. Our `wine` data set has 13 columns, and we need to reduce this number to three so we can understand what is going on. Furthermore, since we are trying to plot the points in a book (a non-interactive medium), we should reduce the number of dimensions to two, if possible.

18.1.1 Dimension Reduction with PCA

Principal component analysis (PCA) is a projection technique that is used to reduce the number of dimensions for a data set. It works by finding a lower dimension in the data such that the variance is maximized. Imagine a three-dimensional sphere of points. PCA essentially shines a light through these points and casts a shadow in the lower two-dimensional plane. Ideally, the shadows will be spread out as much as possible. While points that are far apart in PCA may not be cause for concern, points that are far apart in the original 3D sphere can have the light shine through them in such a way that the shadows cast are right next to one another. Be careful when trying to interpret points that are close to one another because it is possible that these points could be farther apart in the original space.

We import `PCA` from `sklearn`.

Click here to view code image

```
from sklearn.decomposition import PCA
```

We tell PCA how many dimensions (i.e., principal components) we want to project our data into. Here we are projecting our data down into two components.

Click here to view code image

```
# project our data into 2 components
pca = PCA(n_components=2).fit(wine)
```

Next, we need to transform our data into the new space and add the transformation to our data set.

Click here to view code image

```
# transform our data into the new space
pca_trans = pca.transform(wine)

# give our projections a name
pca_trans_df = pd.DataFrame(pca_trans, columns=
['pca1', 'pca2'])
```

```python
# concatenate our data
kmeans_3 = pd.concat([kmeans_3, pca_trans_df],
axis=1)

print(kmeans_3)
```

```
     cluster        pca1       pca2
0          1  318.562979  21.492131
1          1  303.097420  -5.364718
2          1  438.061133  -6.537309
3          1  733.240139   0.192729
4          2  -11.571428  18.489995
..       ...         ...        ...
173        2   -6.980211  -4.541137
174        2    3.131605   2.335191
175        2   88.458074  18.776285
176        2   93.456242  18.670819
177        0 -186.943190  -0.213331

[178 rows x 3 columns]
```

Finally, we can plot our results (Figure 18.1).

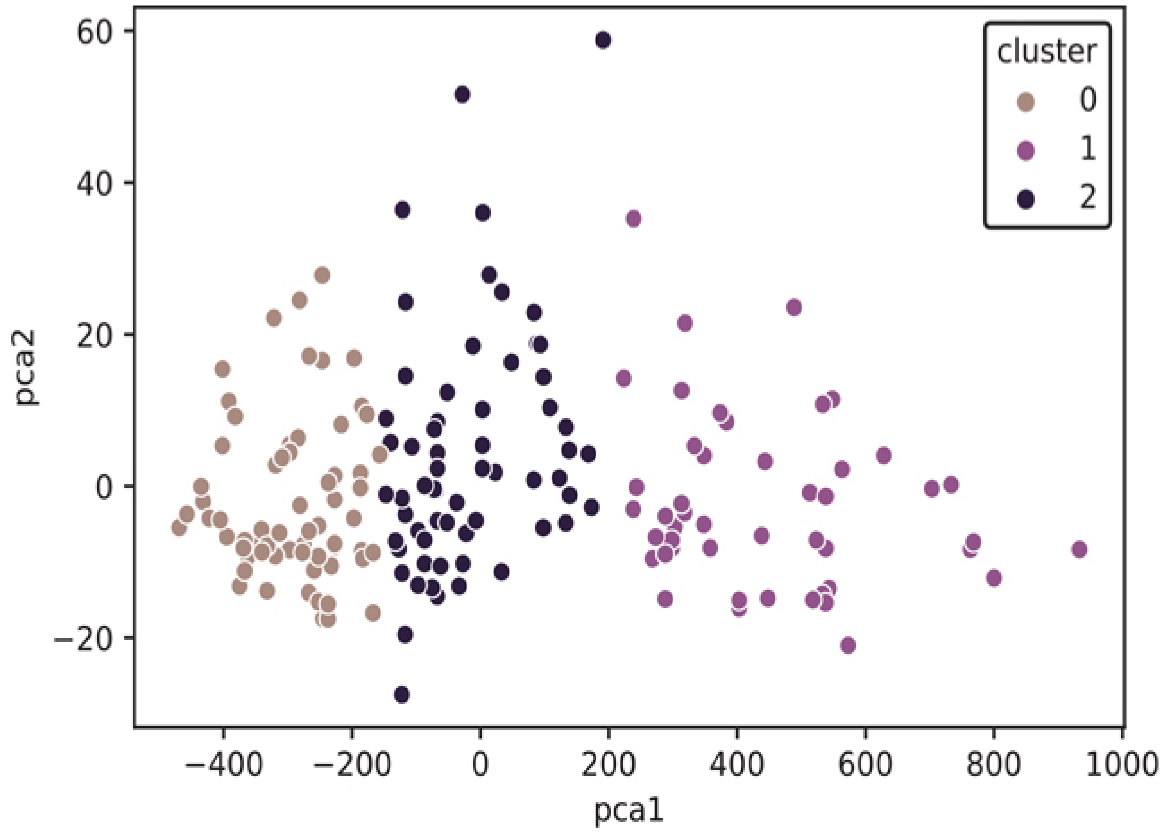


Figure 18.1 *k*-means plot using PCA

Click here to view code image

```
import seaborn as sns
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

sns.scatterplot(
  x="pca1",
  y="pca2",
  data=kmeans_3,
  hue="cluster",
  ax=ax
)
```

```
plt.show()
```

Now that we've seen what *k*-means does to our wine data, let's load the original data set again and keep the `Cultivar` column we dropped.

Click here to view code image

```
wine_all = pd.read_csv('data/wine.csv')
print(wine_all.head())
```

```
   Cultivar  Alcohol  Malic acid   Ash  Alcalinity
of ash \
0         1    14.23        1.71  2.43
15.6
1         1    13.20        1.78  2.14
11.2
2         1    13.16        2.36  2.67
18.6
3         1    14.37        1.95  2.50
16.8
4         1    13.24        2.59  2.87
21.0

   Magnesium  Total phenols  Flavanoids
Nonflavanoid phenols \
0        127           2.80        3.06
0.28
1        100           2.65        2.76
0.26
2        101           2.80        3.24
0.30
3        113           3.85        3.49
0.24
```

```
4        118            2.80         2.69
0.39

   Proanthocyanins  Color intensity   Hue \
0             2.29             5.64  1.04
1             1.28             4.38  1.05
2             2.81             5.68  1.03
3             2.18             7.80  0.86
4             1.82             4.32  1.04
```

Click here to view code image

```
   OD280/OD315 of diluted wines  Proline
0                          3.92     1065
1                          3.40     1050
2                          3.17     1185
3                          3.45     1480
4                          2.93      735
```

We'll run PCA on our data, just as before, and compare the clusters from PCA and the variables from `Cultivar`.

Click here to view code image

```
pca_all = PCA(n_components=2).fit(wine_all)
pca_all_trans = pca_all.transform(wine_all)
pca_all_trans_df = pd.DataFrame(
  pca_all_trans, columns=["pca_all_1",
"pca_all_2"]
)

kmeans_3 = pd.concat(
  [kmeans_3, pca_all_trans_df,
```

```
wine_all["Cultivar"]], axis=1
)
```

We can compare the groupings by faceting our plot (Figure 18.2).

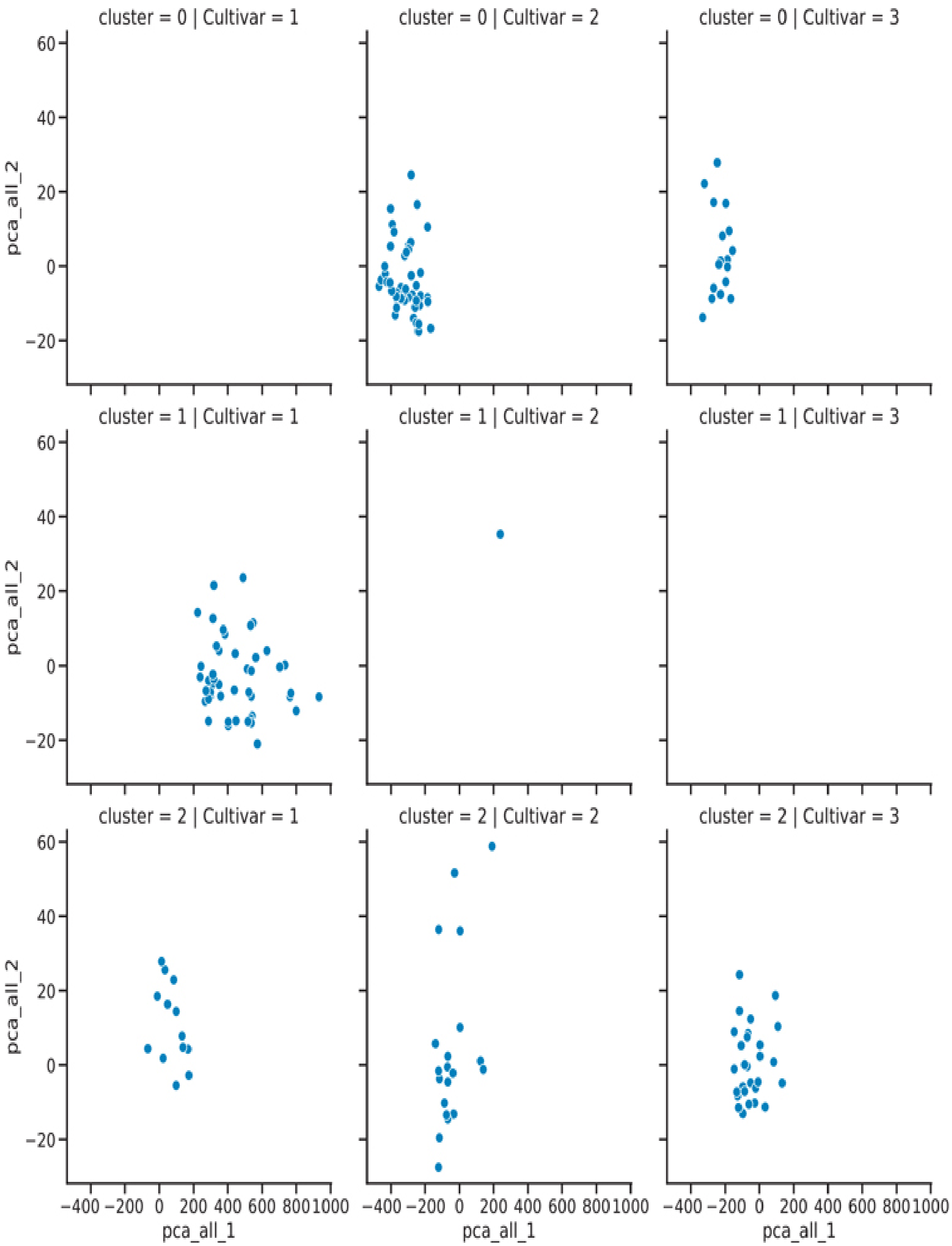


Figure 18.2 Faceted *k*-means plot

Click here to view code image

```
with sns.plotting_context(context="talk"):
  fig = sns.relplot(
    x="pca_all_1",
    y="pca_all_2",
    data=kmeans_3,
    row="cluster",
    col="Cultivar",
  )

fig.figure.set_tight_layout(True)
plt.show()
```

Alternatively, we can look at a cross-tabulated frequency count.

Click here to view code image

```
print(
  pd.crosstab(
    kmeans_3["cluster"], kmeans_3["Cultivar"],
margins=True
  )
)
```

```
Cultivar   1    2   3  All
cluster
0          0   50  19   69
1         46    1   0   47
2         13   20  29   62
All       59   71  48  178
```

18.2 Hierarchical Clustering

As the name suggests, hierarchical clustering aims to build a hierarchy of clusters. It can accomplish this with a bottom-up (agglomerative) or top-town (decisive) approach.

We can perform this type of clustering with the `scipy` library.

Click here to view code image

```
from scipy.cluster import hierarchy
```

We'll load up a clean wine data set again, and drop the `Cultivar` column.

Click here to view code image

```
wine = pd.read_csv('data/wine.csv')
wine = wine.drop('Cultivar', axis=1)
```

Many different formulations of the hierarchical clustering algorithm are possible. We can use `matplotlib` to plot the results.

Click here to view code image

```
import matplotlib.pyplot as plt
```

Below we will cover a few clustering algorithms, they all work slightly differently, but they can lead to different results.

- Complete: Tries to make the clusters as similar to one another as possible
- Single: Creates looser and closer clusters by linking as many of them as possible
- Average and Centroid: Some combination between complete and single
- Ward: Minimizes the distance between the points within each cluster

18.2.1 Complete Clustering

A hierarchical cluster using the complete clustering algorithm is shown in Figure 18.3.

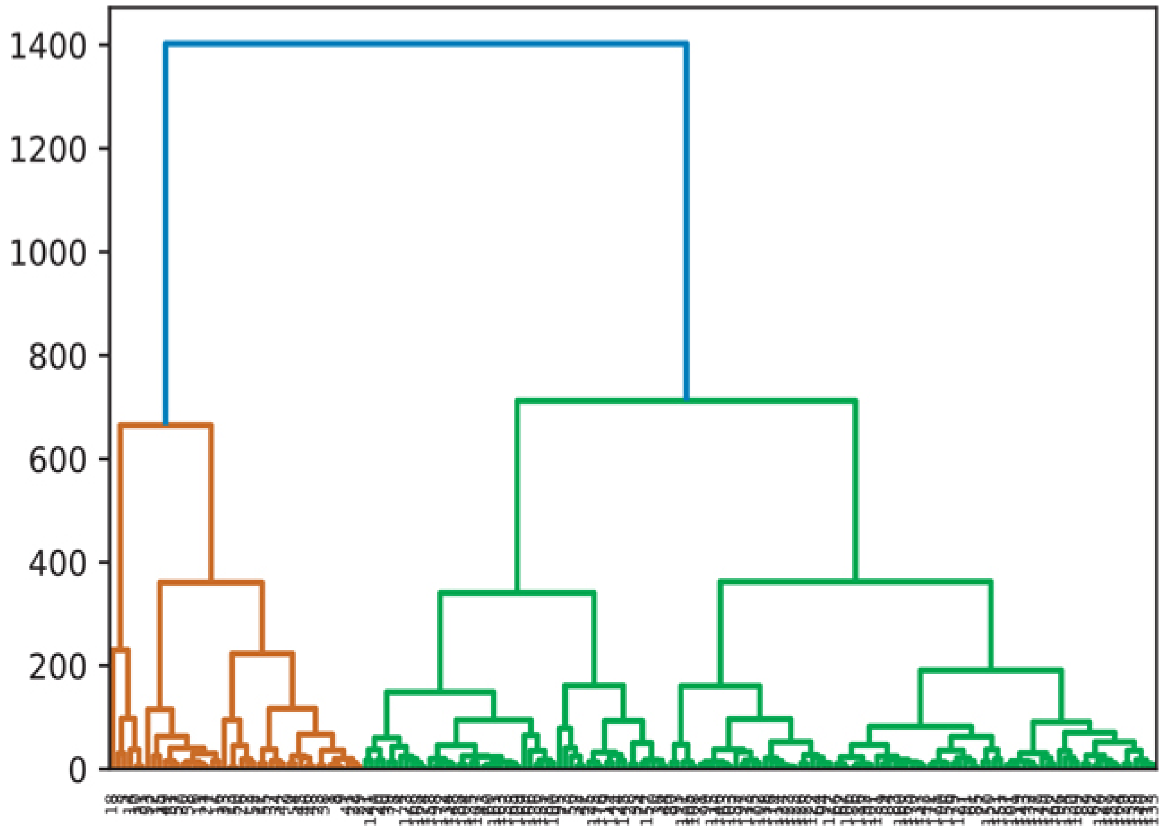


Figure 18.3 Hierarchical clustering: complete

Click here to view code image

```
wine_complete = hierarchy.complete(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_complete)
plt.show()
```

18.2.2 Single Clustering

A hierarchical cluster using the single clustering algorithm is shown in Figure 18.4.

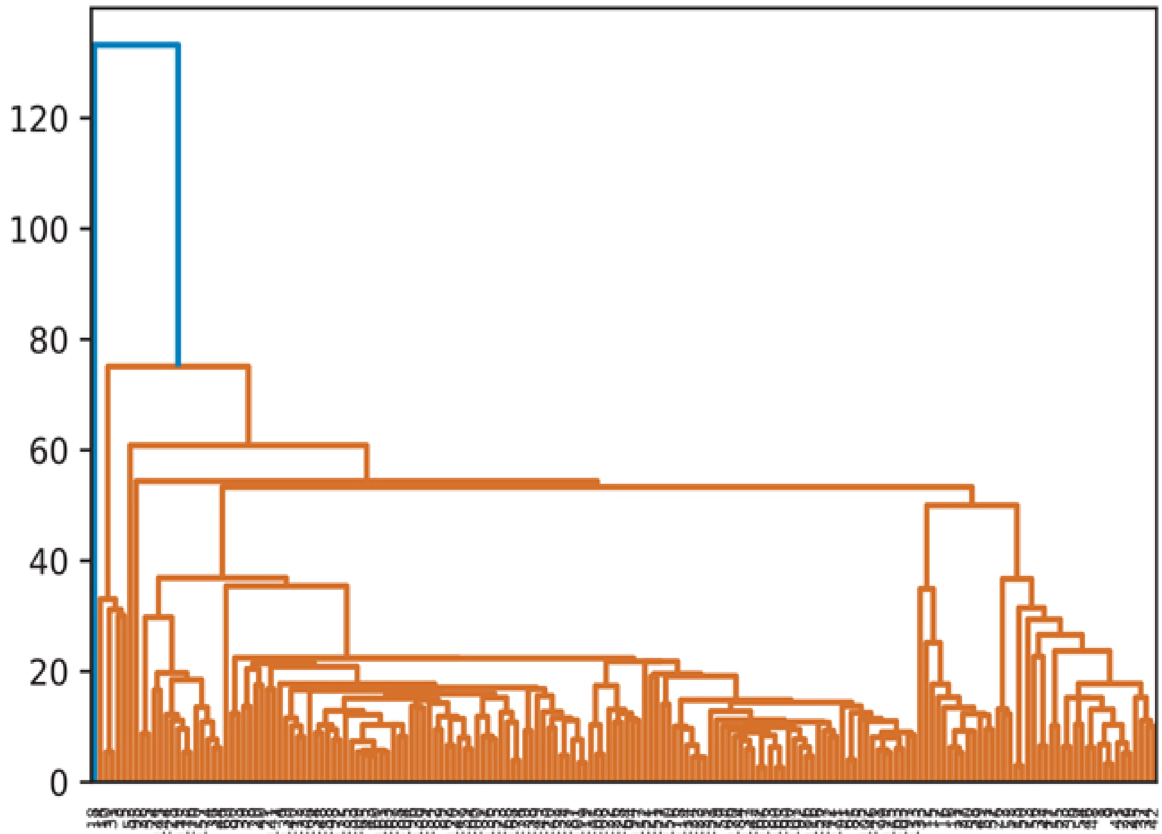


Figure 18.4 Hierarchical clustering: single

Click here to view code image

```
wine_single = hierarchy.single(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_single)
plt.show()
```

18.2.3 Average Clustering

A hierarchical cluster using the average clustering algorithm is shown in Figure 18.5.

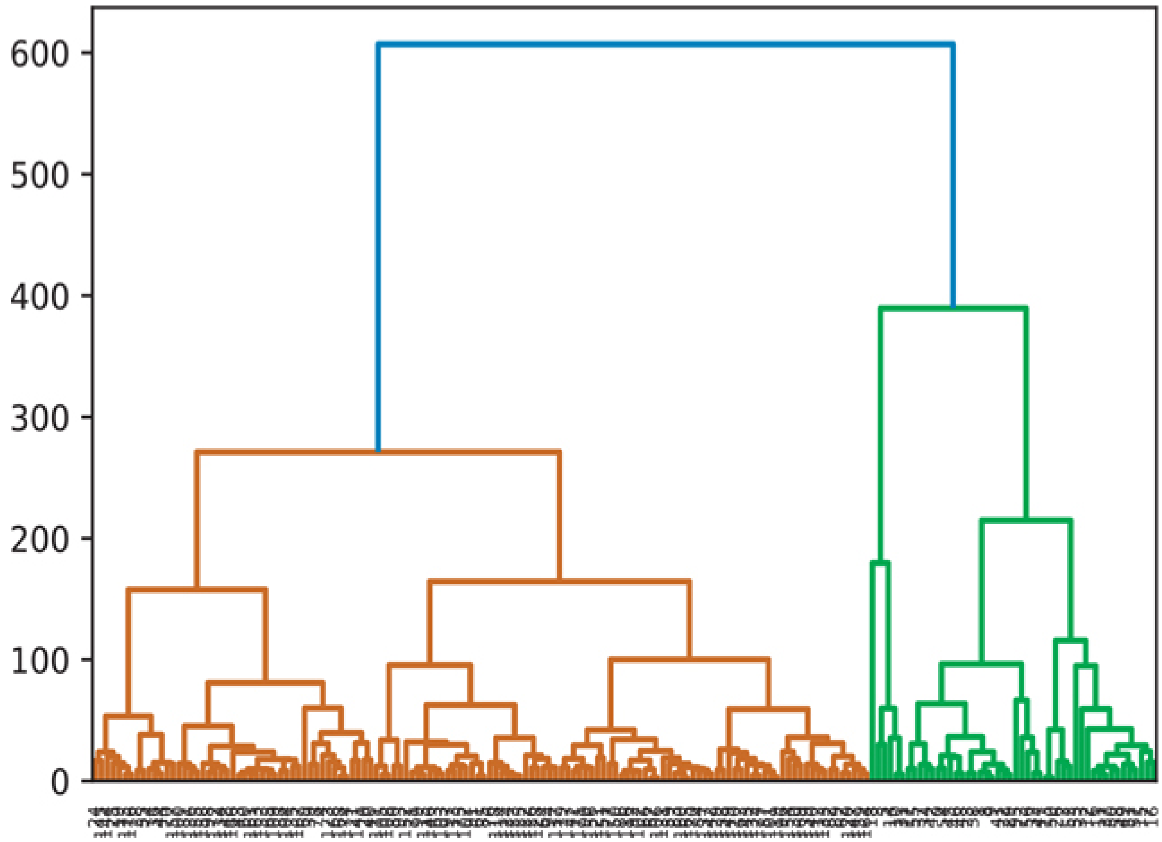


Figure 18.5 Hierarchical clustering: average

Click here to view code image

```
wine_average = hierarchy.average(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_average)
plt.show()
```

18.2.4 Centroid Clustering

A hierarchical cluster using the centroid clustering algorithm is shown in Figure 18.6.

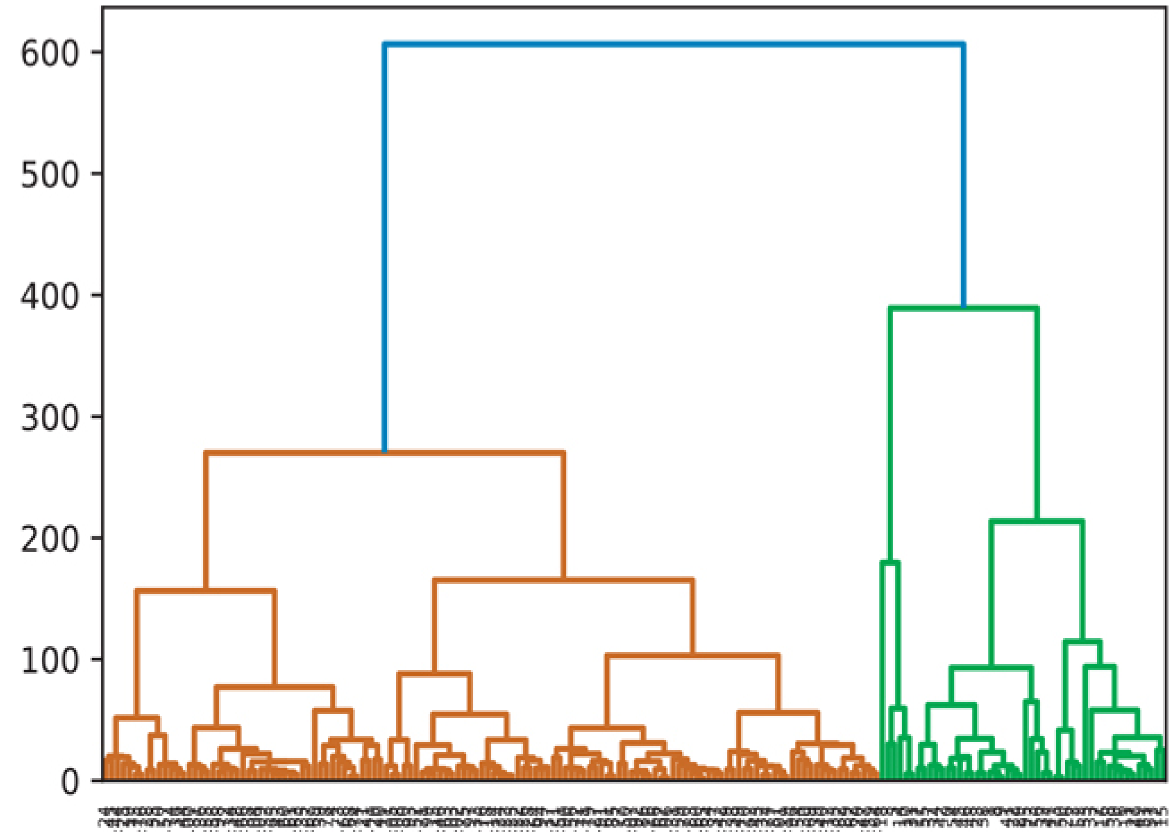


Figure 18.6 Hierarchical clustering: centroid

Click here to view code image

```
wine_centroid = hierarchy.centroid(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_centroid)
plt.show()
```

18.2.5 Ward Clustering

A hierarchical cluster using the ward clustering algorithm is shown in Figure 18.7.

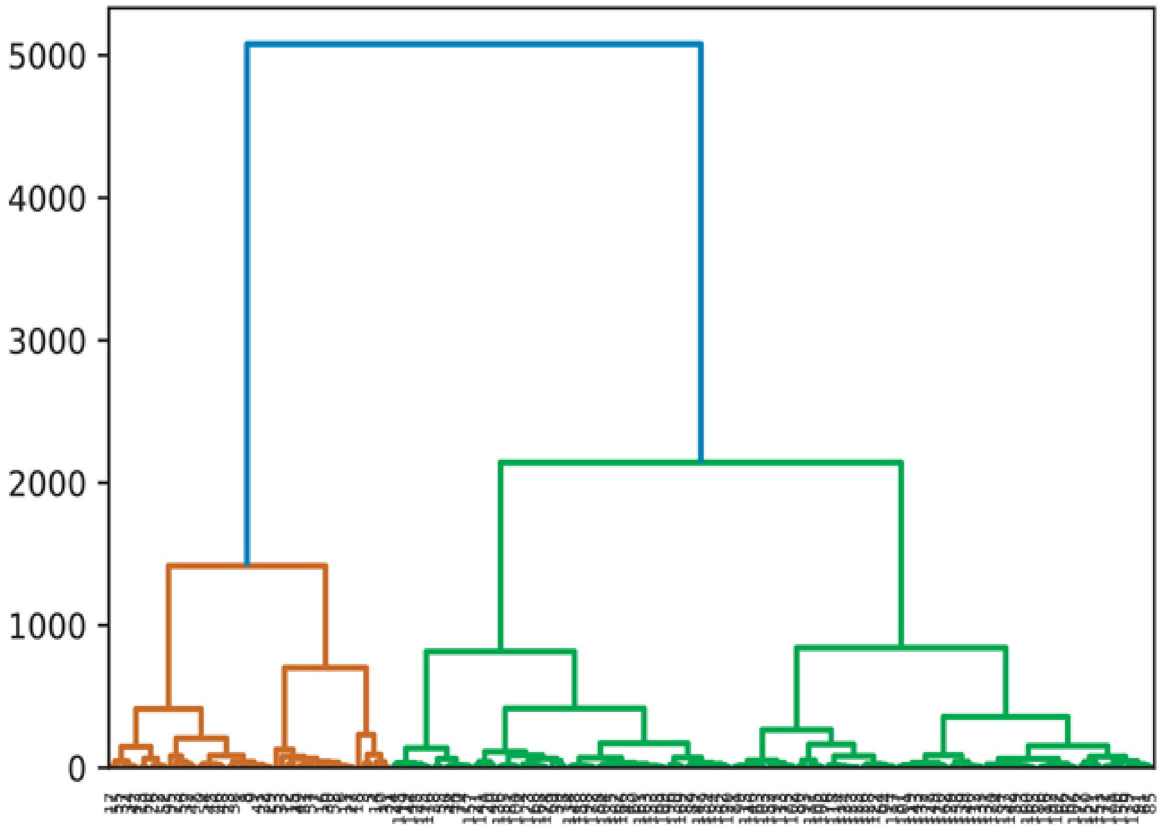


Figure 18.7 Hierarchical clustering: ward

Click here to view code image

```
wine_ward = hierarchy.ward(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_ward)
plt.show()
```

18.2.6 Manually Setting the Threshold

We can pass in a value for `color_threshold` to color the groups based on a specific threshold (Figure 18.8). By default, `scipy` uses the default MATLAB values.

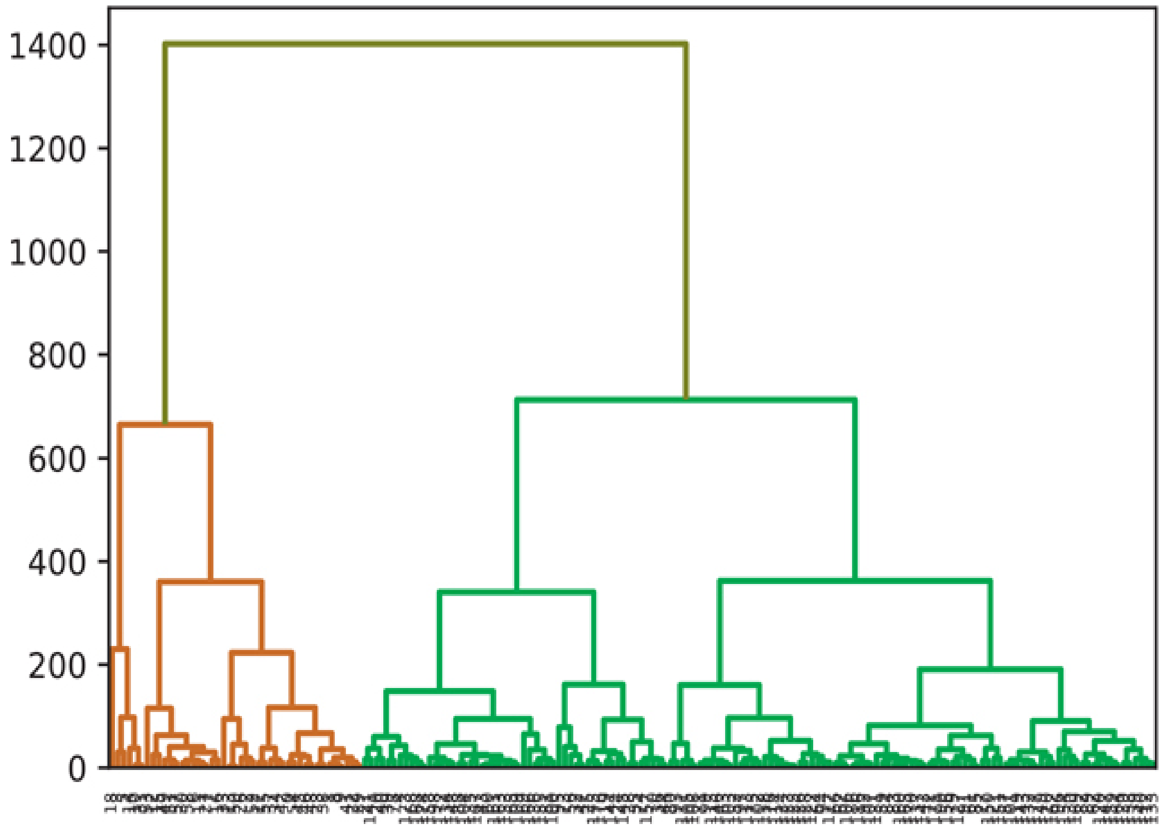


Figure 18.8 Manual hierarchical clustering threshold

Click here to view code image

```
wine_complete = hierarchy.complete(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(
     wine_complete,
     # default MATLAB threshold
     color_threshold=0.7 *
max(wine_complete[:,2]),
     above_threshold_color='y')
plt.show()
```

Conclusion

When you are trying to find the underlying structure in a data set, you will often use unsupervised machine learning methods. *k*-Means and hierarchical clustering are two methods commonly used to solve this problem. The key is to tune your models either by specifying a value for *k* in *k*-means or a threshold value in hierarchical clustering that makes sense for the question you are trying to answer.

It is also common practice to mix multiple types of analysis techniques to solve a problem. For example, you might use an unsupervised learning method to cluster your data and then use these clusters as features in another analysis method.

Part V

Conclusion

If you made it to this part of the book, thank you for reading, and I hope you enjoyed following along and learning the fundamental skills for processing data in Python.

You may hit some of the limitations of Pandas as your data needs grow. Chapter 19 points you to other libraries that expand and parallel Pandas. Finally, Chapter 20 talks about a lot of additional resources for you to continue learning.

19

Life Outside of Pandas

19.1 The (Scientific) Computing Stack

When Jake VanderPlas[1] gave the SciPy[2] 2015 keynote address,[3] he titled his talk "The State of the Stack". Jake described how the community of packages that surround the core Python language developed. Python the language was created in the 1980s. Numerical computing began in 1995 and eventually evolved into the NumPy library in 2006. The NumPy library was the basis of the Pandas `Series` objects that we have worked with throughout this book. The core plotting library, Matplotlib, was created in 2002 and is also used within Pandas in the `plot` method. Pandas' ability to work with heterogeneous data allows the analyst to clean different types of data for subsequent analysis using the scikits, which stemmed from the SciPy package in 2000.

1. Jake VanderPlas: `http://vanderplas.com/`

2. SciPy Conference: `https://conference.scipy.org/`

3. Jake's SciPy 2015 keynote address: `https://speakerdeck.com/jakevdp/the-state-of-the-stack-scipy-2015-keynote`

There have also been advances in how we interface with Python. In 2001, IPython was created to provide more interactivity with the language and the shell. In 2012, Project Jupyter created the interactive notebook for Python, which further solidified the language as a scientific computing platform, as this tool provides an easy and highly extensible way to do literate programming and much more.

However, the Python ecosystem includes more than just these few libraries and tools. SymPy[4] is a fully functional computer algebra system (CAS) in Python that can do symbolic manipulation of mathematical formulas and equations. While Pandas is great for working with rectangular flat files and has support for hierarchical indices, the `xarray` library[5] gives Python the ability to work with *n*-dimensional arrays. Thinking of Pandas as a two-dimensional dataframe—that is, as an array —gives us an *n*-dimensional dataframe. These types of data are frequently encountered within the scientific community.

4. SymPy: `https://www.sympy.org/`

5. Xarray: `http://xarray.pydata.org/`

19.2 Performance

"Premature optimization is the root of all evil". Write your Python code in a way that works first, and that gives you a result which you can test. If it's not fast enough, then you can work on optimizing the code. The SciPy ecosystem has libraries that make Python faster: `cython` and `numba`.

19.2.1 Timing Your Code

Appendix V Gives an example of using the Jupyter `%%timeit` cell magic to time your code. This can be helpful just to compare different methods or implementations, but does not necessarily tell you where to focus your efforts.

19.2.2 Profiling Your Code

Other tools such as cProfile[6] and snakevis[7] can help you time entire scripts and blocks of code and give a line-by-line breakdown of their execution. Additionally, snakevis comes with an IPython `snakevis` extension!

6. cProfile: `https://docs.python.org/3/library/profile.html#module-cProfile`

7. Snakevis: `https://jiffyclub.github.io/snakeviz/`

19.2.3 Concurrent Futures

Many different libraries and frameworks are available to help scale up your computation. `concurrent.futures`[8] allows you to essentially rewrite the function calls into the built-in `map` function.[9]

8. `concurrent.futures`: `https://docs.python.org/3/library/concurrent.futures.html`

9. Python `map()`: `https://docs.python.org/3/library/functions.html#map`

19.3 Dask

Dask is another library that is geared toward working with large data sets.[10] It allows you to create a computational graph, in which only calculations that are out of date need to be recalculated. Dask also parallelizes calculations on your own (single) machine or across multiple machines in a cluster. It creates a system in which you can write code on your laptop, and then quickly scale your code up to larger compute clusters. The nicest part of Dask is that its syntax aims to mimic the syntax from Pandas, which in turn lowers the overhead involved in learning to use this library.

10. Dask: `https://www.dask.org/`

19.4 Siuba

The `tidyverse` set of packages for the R programming language tried to break down each step in the data processing pipeline a single step. This allowed each step to be turned into separate function calls (aka verbs). This is similar to how method chaining works in Pandas. Siuba builds on top of the Pandas library and tries to port the Tidyverse verbs into Pandas.[11]

11. Siuba documentation: `https://siuba.readthedocs.io`

19.5 Ibis

The Ibis project provides a high-level API over tabular data.[12] The main benefits is that it gives the user a consistent way to interact with databases, Dask, and Pandas.

12. Ibis project: `https://ibis-project.org`

19.6 Polars

Polars is a Python (and Rust) dataframe library built on top of Apache Arrow.[13] Its API is similar to Pandas, but relies heavily on method calls. It also removes Pandas indices, something this book has avoided for sake of simplicity. The Polars documentation contains a user's guide that is worth looking into: `https://polars.github.io/polars-book`

13. Polars Library: `https://www.pola.rs/`

19.7 PyJanitor

`pyjanitor` is a Python library that extends Pandas `DataFrame` objects by providing additional `DataFrame` methods to make data processing a little easier.[14] It is modeled after the R package, `janitor`, and has a lot of convenient methods for common data processing steps.

14. `pyjanitor` documentation: `https://pyjanitor-devs.github.io/pyjanitor/`

19.8 Pandera

Many of the steps in data process involve checking and validating data. The `pandera` provides a mechanism for you to test your data.[15] For example, you can use it to make sure there are valid values for a particular column. The tools provided in `pandera` allow you to check your data and have the code fail when it does not meet assumptions before you model the data and make conclusions from it.

15. `pandera` documentation: `https://pandera.readthedocs.io/`

19.9 Machine Learning

This book aimed to lay a foundation to all the parts in the data science process. It's hard to be completely inclusive and cover *everything* that a data scientist might need. Machine learning methods like XGBoost have become extremely popular for its ability to work with a wide variety of data sets and perform well in prediction tasks.[16] We've mentioned a little bit of scikit-learn pipelines in Section 13.4.[17]

16. XGBoost: `https://xgboost.readthedocs.io/`

17. scikit-learn pipelines: `https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html`

To use these machine learning models in production we need to be able to maintain, version control, deploy, and monitor them. This is where MLOps (Machine Learning Operations) come into play, and tools like vetiver can help with that.[18]

18. Vetiver: `https://vetiver.rstudio.com/`

19.10 Publishing

This book was written in a publishing system called Quarto.[19] This allows you to do "literate programming", where you can mix prose text with code and code output. Why I like Quarto is that it is a single program that lets me write reports, books, websites, presentations, etc. It also allows me to work in R and Python simultaneously, which this book also does in Appendix Z.

19. Quarto: `https://quarto.org/docs/books`

JupyterBook is another literate programming platform that builds on Jupyter Notebooks to create a book format.[20]

20. JupyterBook: `https://jupyterbook.org/`

19.11 Dashboards

Over the years many dashboard libraries have been created for Python. Dash,[21] Streamlit,[22] Panel,[23] and Voilà[24] are some of them. I've personally done a lot of my data science result communication work in the R ecosystem, so I'm happy that Shiny for Python[25] was recently announced at the time of writing, since it is similar to what I already know. All the dashboard platforms have pros and cons and have tradeoffs with learning curve, scalability, and flexibility.

21. Dash: `https://plotly.com/dash/`

22. Streamlit: `https://streamlit.io/`

23. Panel: `https://panel.holoviz.org/`

24. Voilà: `https://voila.readthedocs.io`

25. Shiny for Python: `https://shiny.rstudio.com/py/`

Conclusion

Pandas is a popular data science library in Python. Its ubiquity has made it the go-to library when working with data in Python. However, it may not meet everyone's needs and that is why so many other libraries have been built to parallel or extend Pandas. This book mainly focuses around Pandas as the tool to help you think about data processing and give you the foundation to explore other dataframe libraries.

Look out for additional chapters published for free with the book. Many of the libraries mentioned in this part of the book will be expanded upon and released online.

20

It's Dangerous To Go Alone!

Heed this advice! One of the best ways to learn a language is to work on a problem with other people. For example, in pair-programming, two people program together. Alternatively, one person can do the typing while the other person talks through the code. This allows two sets of eyes to look at the code, improves communication between the two colleagues, and gives a sense of ownership. These shared-programming techniques both contribute to higher-quality code and make programming fun, which means you're more likely to improve by doing it more often.

20.1 Local Meetups

Many cities have a Meetup culture in which people can find a common hobby or topic and have a place to "meet up".[1] Python-specific meetups exist, but it's worth going to others that focus on data cleaning, visualization, or machine learning. Even meetups in other languages can be helpful. The more you expose yourself to the community and the field, the more connections you can make with your own work.

1. Meetup: `https://www.meetup.com/`

If there isn't a meetup in your city, create one! You can start with friends and people who are interested, and begin to host regular times to meet and talk. Keep it fun. Talk about topics of interest at a bar. Again, the more enjoyable something is, the more likely you are to do it.

Since the COVID-19 pandemic, many meetups have moved to virtual + online, and hybrid options for meetups are becoming the norm.

20.2 Conferences

Conferences are a great way to learn about the latest libraries and techniques. You also get to meet new people as well as library maintainers. Many conferences sponsor a “sprint day”, during which people are encouraged to work on code and contribute to a library. This is a great way to learn about the library itself, to improve your programming skills, and to contribute to the community.

PyCon is the main Python conference.[2] It includes topics across the entire Python ecosystem, such as Django[3] and Flask[4] for web development. The talks for these conferences are usually recorded and freely available.[5] The SciPy[6] and EuroSciPy[7] conferences focus more on the scientific and analytics stack aspects of Python. I have attended SciPy over the past few years, and I can assure you that the tutorials cover a vast set of topics. The best way to view the conference tutorials and talks is to find the respective YouTube page for the conference.

2. PyCon conference: `https://us.pycon.org`

3. Django: `www.djangoproject.com`

4. Flask: `https://flask.palletsprojects.com`

5. Python 2017 talks: `www.youtube.com/channel/UCrJhliKNQ8g0qoE_zvL8eVg`

6. SciPy Conference: `https://conference.scipy.org`

7. EuroSciPy Conference: `https://www.euroscipy.org/`

AnacondaCon is a newer conference that likewise has videos posted online.[8] Jupyter also hosts its own conferences. Jupyter Days and JupyterCon have videos, and you can hear when the next conference is on the main Jupyter blog.[9] Finally, PyData, the nonprofit that supports many open-source projects, sponsors conferences and provides videos.[10]

8. AnacondaCon Conference: `https://anacondacon.io/`

9. JupyterCon Conference `https://jupytercon.com/`

10. PyData: `https://pydata.org/`

20.3 The Carpentries

The Carpentries is a nonprofit organization that aims to teach all the programming and data skills to researchers. It's where I got my start into data science education. Software-Carpentry, Data Carpentry, and Library Carpentry are sister organizations under The Carpentries.

The Carpentries does a great job sharing their lesson materials. If you ever need a resource to learn or teach out of, I cannot recommend the materials from The Carpentries enough: https://carpentries.org/workshops-curricula/.

20.4 Podcasts

Data science related podcasts are plentiful. Here are some that I listen to (in no particular order):

- Vanishing Gradients: https://vanishinggradients.fireside.fm/
- Data Skeptic: https://dataskeptic.com/
- Talk Python to Me: https://talkpython.fm/
- Python Bytes: https://pythonbytes.fm/
- Super Data Science: https://www.superdatascience.com/podcast
- Shiny Developer Series: https://shinydevseries.com/
- R Weekly Highlights: https://rweekly.fireside.fm/
- Not So Standard Deviations: https://nssdeviations.com/
- Partially Derivative (discontinued): http://partiallyderivative.com/
- Linear Digressions (discontinued): http://lineardigressions.com/
- Becoming a Data Scientist (discontinued): www.becomingadatascientist.com

While this isn't an exhaustive list, these podcasts will give you a good sense of the Python and data science community and the tools, news, and thinking behind many data science methods.

20.5 Other Resources

Instead of trying to create a list of Python resources in a book, I've started a project called "The Big Book of Python" that aims to parallel "The Big Book of R". These resources aim to curate a bunch of free resources into a single page. I hope these resources help you with your future data science journey.

- `https://www.bigbookofpython.com/`
- `https://www.bigbookofr.com/`

Conclusion

This book was intended to provide you with a solid foundation from which to learn more about Pandas and its related libraries. Be sure to check out the accompanying github repository for the book for updates and additional resources:
`https://github.com/chendaniely/pandas_for_everyone`
.

Part VI

Appendices

A

Concept Maps

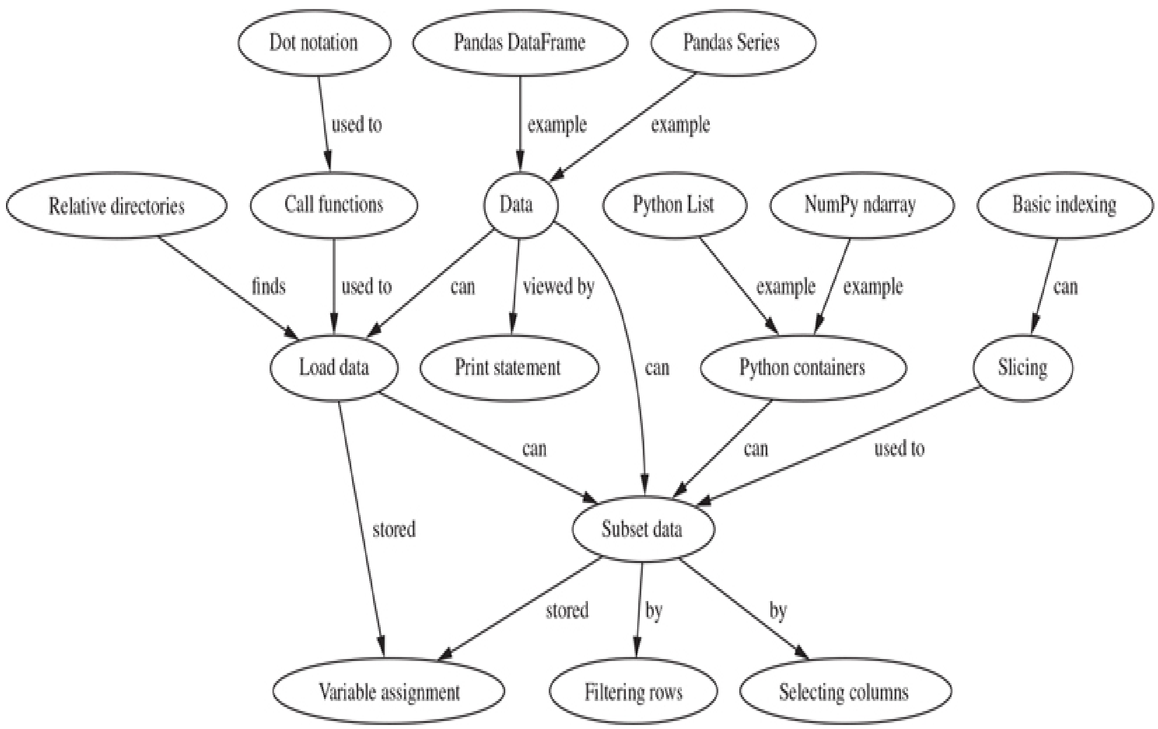


Figure A.1 Concept Map for Pandas DataFrame Basics

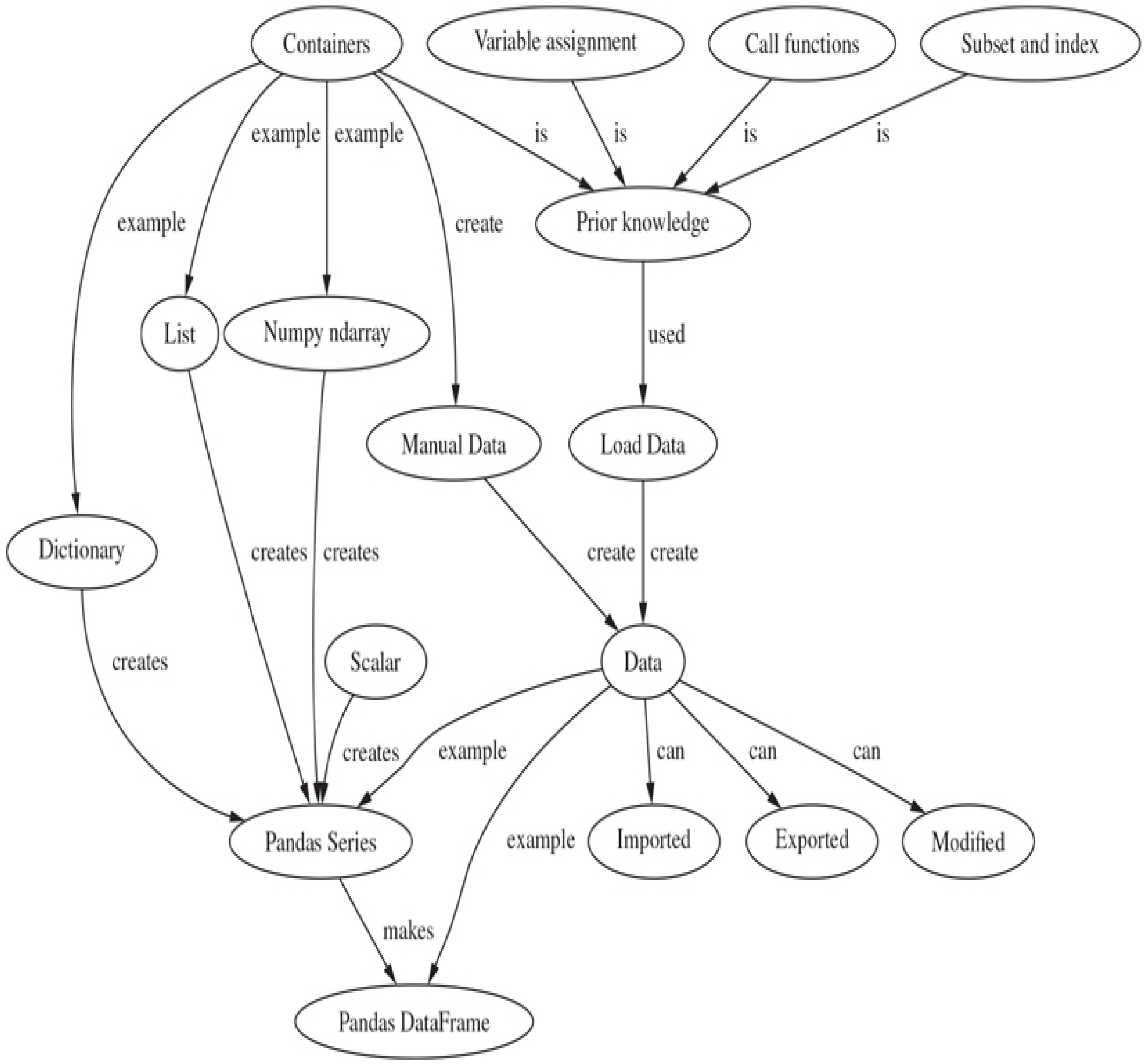


Figure A.2 Concept Map for Pandas Data Structures Basics

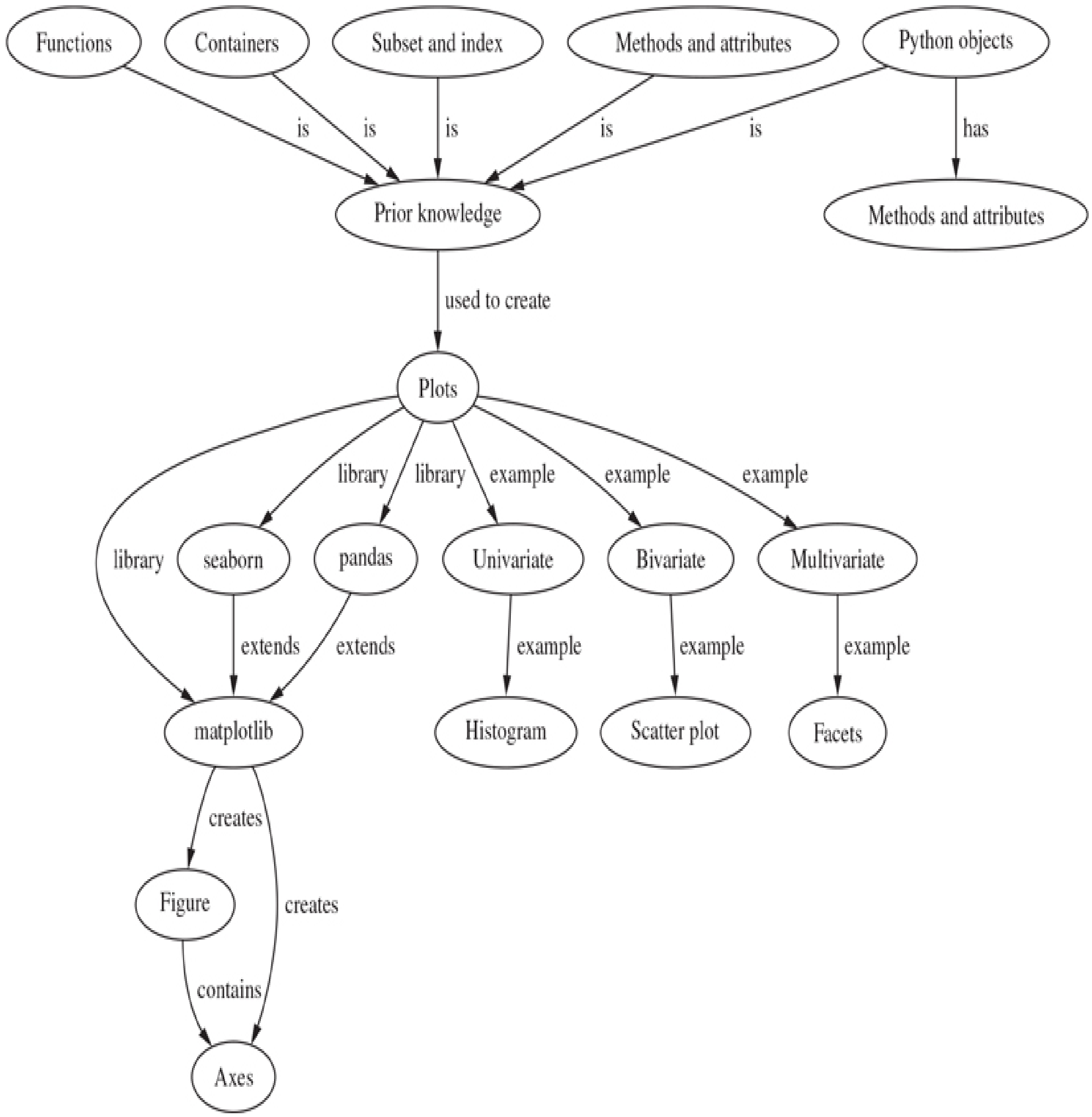


Figure A.3 Concept Map for Plotting Basics

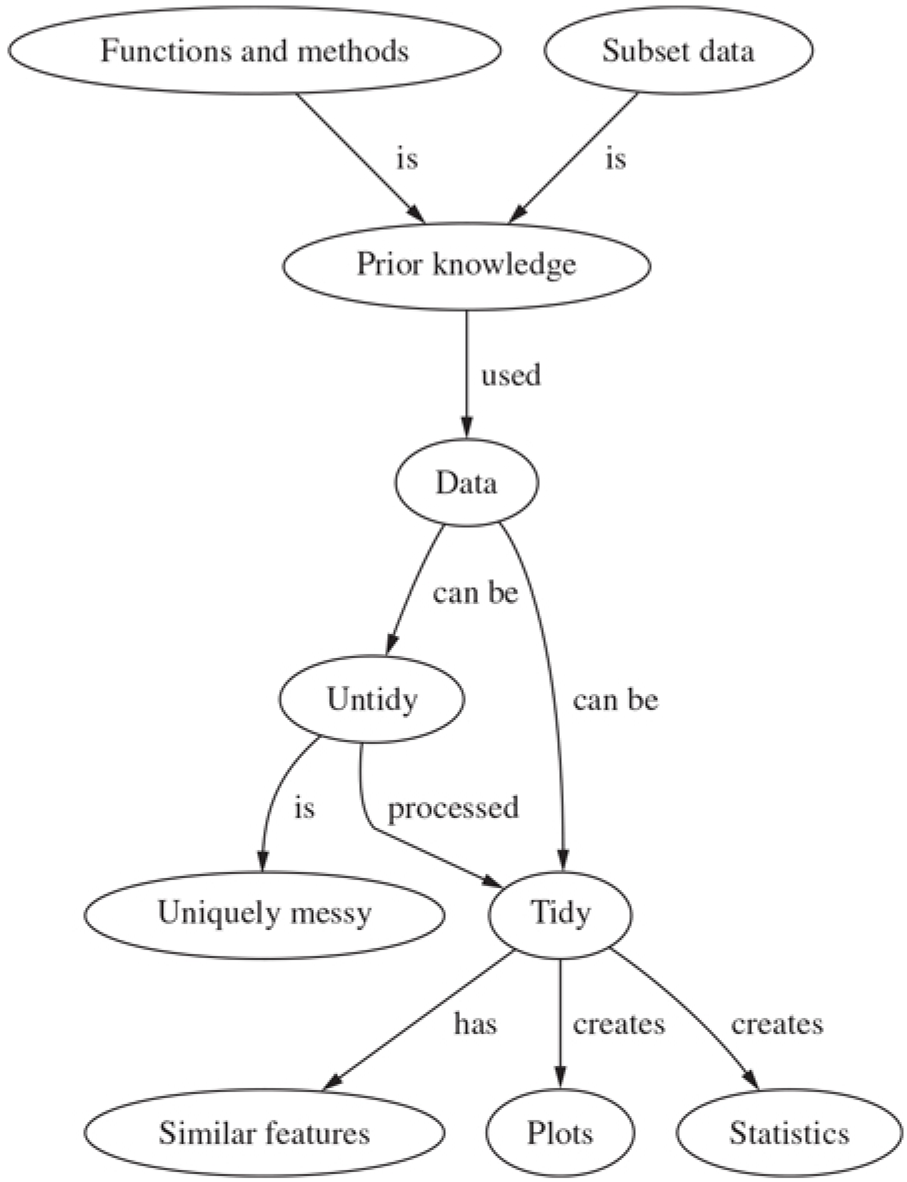


Figure A.4 Concept Map for Tidy Data

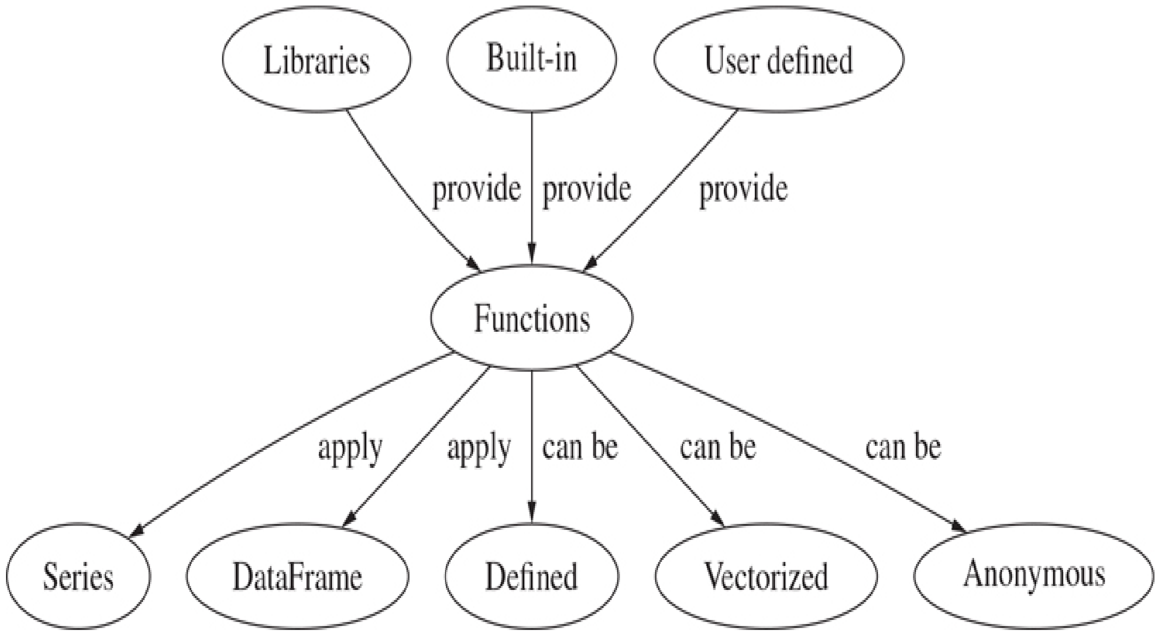


Figure A.5 Concept Map for Apply Functions

B

Installation and Setup

B.1 Install Python

Since Software-Carpentry has been using the Anaconda distribution, I will be using it for the installation instructions described in this appendix. You can also find the generic workshop template installation instructions for Python here:

Click here to view code image

```
https://carpentries.github.io/workshop-
template/#python
```

B.1.1 Anaconda

For the most part, the directions listed on the main Anaconda download site will be the same as the ones listed in this book.[1] You can also look at the Anaconda installation documentation.[2] Be sure to use the Python 3 version. If you also need to have Python 2, follow the instructions in Appendix F on creating Python environments.

1. https://www.anaconda.com/products/distribution

2. https://docs.continuum.io/anaconda/install/

B.1.1.1 Windows

Install Anaconda using the Windows installer with all the default settings. Make sure you check off the box for **Add Anaconda to my PATH environment variable**.

B.1.1.2 Mac

Install Anaconda using the Mac installer with all the default settings.

B.1.1.3 Linux

Installing on Linux involves downloading the `.sh` file and running it from the command line. You can do this by navigating to the Anaconda download site and downloading the `.sh` file there. Alternatively, if you are on a server, for example, you can use the `wget` command. Assuming the `.sh` file is in your `Downloads` folder:

Click here to view code image

```
cd ~/Downloads
bash Anaconda3- * .sh  # your version number
will differ
```

Note that the version of Anaconda will be different by the time this book is published.

Keeping the default options is a good choice. When the installation process asks you to read the license agreement, you can press `q` to exit or accept by typing `yes`.

Type `yes` when the installer asks to prepend Anaconda to the `PATH`. This makes Anaconda the default Python distribution on the system.

When you are done, close the current terminal window. Any new terminal moving forward will default to the Anaconda Python distribution.

B.1.2 Miniconda

Anaconda is a big download because it comes with a lot of packages and dependencies pre-installed. Miniconda is an alternative to the full Anaconda distribution. It only comes with Python installed, and all the other packages need to be installed manually.

B.1.3 Uninstall Anaconda or Miniconda

Since Anaconda will create an `Anaconda3` folder in your home directory, deleting this folder will completely remove anything associated with Anaconda on the machine. This is one of my favorite features of using Anaconda. If I install a bad Python package, I can reset everything back to "normal" by deleting the `Anaconda3` folder.

For Miniconda, you will have a `miniconda3` folder instead.

B.1.4 Pyenv

Pyenv is a tool that lets you manage different versions of Python. It also has a plugin for you to also manage package environments. The benefit that `pyenv` has over `conda` is that it plays a little bit nicer with other tools outside of Python, since it *only* manages the Python version.

Below are some resources to install, setup, and use `pyenv`

- Posit, PBC (formerly RStudio, PBC has a minimal viable python setup instruction for Pyenv: `https://solutions.rstudio.com/python/minimum-viable-python/`
- Calvin Hendryx-Parker gave a great talk at PyCon 2022 on *Bootstrapping Your Local Python Environment* that goes over the Pyenv setup with the pyenv-virtualenv plugin: `https://www.youtube.com/watch?v=-YEUFGFHWgQ`
- Real Python *Managing Multiple Python Versions With pyenv*: `https://realpython.com/intro-to-pyenv/`

The main downside is that Pyenv plugins are not supported on Windows. That means the very useful pyenv-virtualenv plugin isn't usable. For that reason, if you want to go to Pyenv route, I suggest you look into Pipenv for the virtual environment management, and use Pyenv for the Python version management. This way, you have a setup that is OS agnostic.

B.2 Install Python Packages

See Appendix H for how to install the packages needed to code along this book. If you are using a Python setup other than Anaconda (or its derivatives that use `conda`), You need to replace the `conda install` command with `pip install`.

B.3 Download Book Data

You can download the data sets for the book by going to the book's repository and downloading the ZIP file of the repo.

The book's repository can be found here: `https://github.com/chendaniely/pandas_for_everyone`

You can do this by going to the main repository page then clicking Code > Download ZIP (Figure B.1).

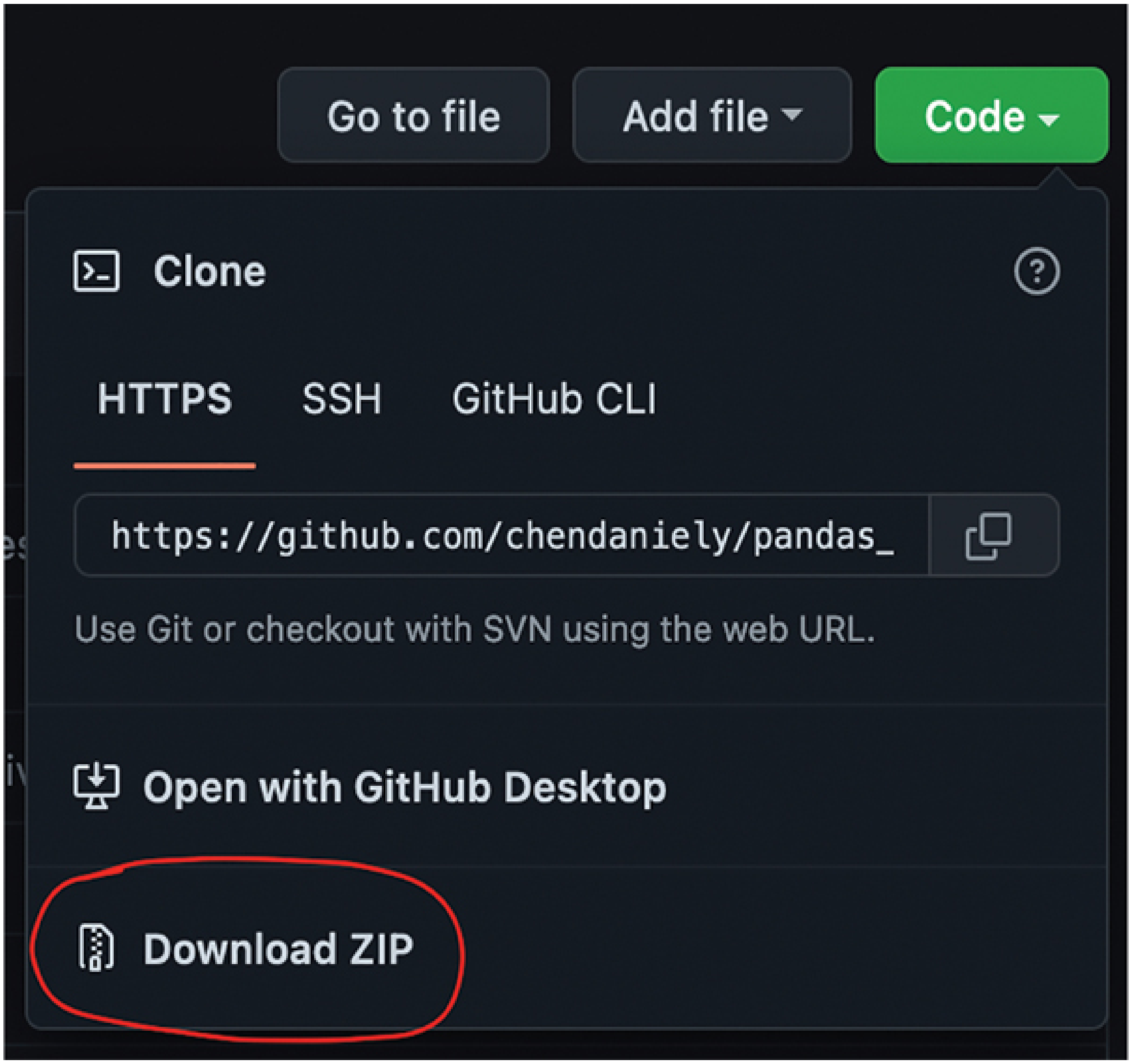


Figure B.1 Clicking on Code > Download ZIP to download the data sets for the book. You can also try the direct URL to the ZIP file here: `https://github.com/chendaniely/pandas_for_everyone/archive/refs/heads/master.zip`

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.

C

Command Line

Having some familiarity with the command line can go a very long way. My main suggestion is to go through the Software-Carpentry Unix Shell lesson.[1] The "Navigating Files and Directories" episode (i.e., lesson) is probably the most important lesson there for this book, but learning about "Shell Scripts" is also important when you are running your Python code from the command line.

1. https://swcarpentry.github.io/shell-novice/

Since this book is mainly a Python book about Pandas, I won't be able to go over all of the topics in learning the Unix Shell. The main takeaway I want to convey in this appendix is the notion of a "working directory".

C.1 Installation

Likely, if you are on a Mac or Linux system, you will already have access to the Bash Shell. By default, Windows does not have it installed.

C.1.1 Windows

In Windows, the best installation approach is to follow the Software-Carpentry Bash Shell instructions.[2] You will be installing Git for Windows,[3] which will also provide the Bash Shell.

2. https://carpentries.github.io/workshop-template/#shell

3. https://gitforwindows.org/

If you do not want to use Git for Windows, Anaconda also comes with its own Anaconda Prompt that you can use to run Python code from the command line. The only difference here is that the Anaconda Prompt will use Windows command line commands, instead of the UNIX-like ones on a Mac or Linux system. However, running your Python scripts from the command line will be the same.

C.1.2 Mac

You can find the `Terminal` application in `Applications / Utilities`. That is, in your main application folder, there will be a folder called `Utilities`, where you can find the `Terminal`.

iTerm2 is a popular alternative to the default Mac `Terminal` application.[4]

4. `https://iterm2.com/`

C.1.3 Linux

The terminal and `bash` are set up on Linux systems by default.

C.2 Basics

At minimum, you should know the following commands:

- Where you currently are in your file system (Windows, Mac, Linux: `pwd`)
- List the contents of the current folder you are in (Windows: `dir`, Mac, Linux: `ls`)
- Change to a different folder (: `cd <folder name>`)
- Run a Python script (Windows, Mac, Linux: `python <python script>.py`)

Another useful "command" is `..` (two dots), which refers to the parent folder of where you are now (Windows, Mac, Linux: `pwd`).

D

Project Templates

It is very easy and convenient to put all the data, code, and outputs in the same folder. However, this convenience is negated by disadvantages of having a messy project folder. That is, putting everything into a single folder can easily lead to a folder on your computer with tens or hundreds of files, which can become unmanageable and confusing for not only others, but yourself.

At minimum, I suggest the following folder structure for any analysis project:

```
my_project/
  |
  |- data/
  |
  |- analysis/
  |
  +- output/
```

I put all my data sets in the `data` folder, any code I write for analysis in the `analysis` folder (sometimes I will name this `code` or `src`), and finally cleaned data sets or other outputs such as figures in the `output` folder. You can adapt this general folder structure as you need.

Here is a paper reference that discusses the theory a bit further:

> Noble WS. (2009). "A Quick Guide to Organizing Computational Biology Projects." *PLoS Comput Biol* 5(7): e1000424. https://doi.org/10.1371/journal.pcbi.1000424

E

Using Python

There are many different ways to use Python. The "simplest" way is to use a text editor and terminal. However, projects like IPython and Jupyter have enhanced Python's REPL (Read–Evaluate–Print–Loop) interface, making it one of the standard interfaces in the data analytics and scientific Python communities.

E.1 Command Line and Text Editor

To use Python from the command line and text editor, you need is a plain text editor and a terminal. Although any plain text editor would work, a "good" one would have a Python feature that will do syntax highlighting and auto-completion. These days VSCode has become a popular text editor that has good extensions for Python support:
https://code.visualstudio.com/

If you are on Windows, be careful not to do too much editing using the default Notepad application, especially if you plan to collaborate with users on other operating systems. Line endings in Notepad are different from those in Windows and on *nix machines (Linux and Macs). If you ever open up a Python file and the indentations and newlines do not appear correctly, it's probably because of how Windows is interpreting the newline endings of the file.

When you work in a text editor, all your Python code will be saved in a `.py` script. You can run the script by executing it from the command line. For example, if your script's name is `my_script.py`, you can execute all the code in the script, line-by-line, with the following command:

```
python my_script.py
```

More information about running Python scripts from the command line is found in Appendix C and Appendix F.

E.2 Python and IPython

Under Windows, Anaconda will provide an "Anaconda command prompt" application. This is just like the regular windows command prompt but is configured to use the Anaconda Python distribution. Typing `python` or `ipython` here will open the `python` or `ipython` command prompt, respectively.

For OSX and Linux, you can run the `python` or `ipython` command prompt by typing the respective command in a terminal.

There are a few differences between the `python` and `ipython` command prompts. The regular `python` prompt takes only Python commands, whereas the `ipython` prompt provides some useful additional commands you can type to enhance your Python experience. My personal suggestion is to use the `ipython` prompt.

You can directly type Python commands into either prompt, or you can save your code in a file and then copy/paste commands into the prompt to run your code.

E.3 Jupyter

Instead of running `python` or `ipython` in the command prompt to run Python, you can run the `jupyter notebook` or `jupyter lab`. This will open another Python interface in a web browser. Even though a web browser is opened, it does not actually need any Internet connection to run, nor is any information sent across the Internet.

The `jupyter notebook` will open in a location on your computer. You can create a new notebook by clicking the "New" button on the top right corner and selecting "python." This will open up a "notebook" where you can type your python commands. Each cell provides a site where you can type your code, and you can run the cell by using the commands in the "Cell" menu bar. Alternatively, you can press `Shift + Enter` to run the cell and create a new cell below it, or press `Ctrl + Enter` to simply run the cell.

An especially useful aspect of the notebook is the ability to interweave your Python code, its output, and regular prose text. Similar to how the text, code, and output is presented in this book.

To change the cell type, make sure you have the cell selected. Then, on the top right below the menu bar, click a drop-down menu that says “Code.” If you change this to “Markdown,” you can write regular prose text that is not Python code to help interpret your results, or record notes about what your code is doing.

E.4 Integrated Development Environments (IDEs)

Anaconda comes with an IDE called Spyder. Those who are familiar with Matlab or RStudio might take comfort in having access to a similar interface.

Other IDEs include the following:

- nteract: `https://nteract.io/`
- PyCharm: `https://www.jetbrains.com/pycharm/`
- VSCode: `https://code.visualstudio.com/`

I suggest exploring the various ways to use Python and seeing which works best for you. IPython/script, Jupyter notebook, and Spyder come pre-installed with Anaconda, so those would be the most accessible, but the other IDEs might work better for your particular circumstances.

F

Working Directories

Building on Appendix C, Appendix D, and Appendix E, this appendix covers working directories, especially when you are working with project templates (Appendix D).

A working directory simply tells the program where the base or reference location is. It's common to place all of your code, data, output, figures, and other project files all in the same folder, because it means the working directory is easy to figure out. However, this practice can easily lead to a messy folder, as mentioned in Appendix D.

We like fully documented project templates that tell us where and how to run our scripts. With this approach, all our scripts have a predictable and consistent working directory.

There are a few ways to figure out what your current working directory is. If you are using IPython, then you can type `pwd` into the `IPython` prompt, and it will return the folder path of your current working directory. This method also works if you are using the Jupyter notebook.

If you are executing your Python code as scripts directly in the command line, then the working directory is the output after you run `cd` on Windows (note there is nothing else after the command), and `pwd` on OSX and Linux.

Here is an example of how working directories affect your code. Suppose you have the following project structure, where the current working directory is denoted by a star (*).

```
my_project/
  |
  |- data/
  |    |
```

```
  |    + data.csv
  |
  |- src/ *
  |    |
  |    + script.py
  |
  +- output/
```

If your `script.py` wants to read in a data set from the `data` folder, it would have to do something like `data = pd.read_csv('../data/data.csv')`. Note that because the current working directory is in the `src` folder, to navigate to the `data.csv`, you need to go up one level `..` to the `my_project` folder and then down into the `data` folder to get to your data set. The benefit of this is that you can run your code by tying it to `python script.py`, though this can lead to some issues discussed later in this appendix.

Let's use a different working directory:

```
my_project/ *
  |
  |- data/
  |   |
  |   + data.csv
  |
  |- src/
  |    |
  |    + script.py
  |
  +- output/
```

Now that the working directory is on the top level, `script.py` can reference the data set with the command `data = pd.read_csv('data/data.csv')`. Note that you no longer need to go up a level to reference your data. However, now if you want to run your code, you have to reference the file as such: `python src/script.py`.

This may be annoying, but it allows you to create any amount of subfolders, and `data` and `output` will always be referenced the same way across all the files.

It also means you as a user have one and only one working directory to execute any script in this project.

G

Environments

Using environments is a great way to work with different versions of Python and/or packages. It also provides an isolated environment to install everything so that if something goes wrong, it won't affect the rest of the system. Python environments are particularly handy when you need different versions of packages installed across different projects. You can also use environments to see all the package dependencies.

G.1 Conda Environments

The Anaconda Python distribution comes with `conda`. The "Getting Started" guide is a useful resource in this case.[1] If you installed Anaconda with Python 3 (Appendix B), this appendix will show you how to create a separate environment that has a different version of Python in it. If we run `python` in the command line, we will begin with Python 3.9. Your exact version will differ from that shown in this book.

1. https://conda.io/projects/conda/en/latest/user-guide/getting-started.html

Click here to view code image

```
% python
Python 3.9.12 (main, Jun  1 2022, 06:34:44)
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

To create a new environment we run the `conda` command from the command line. We use the `create` command within `conda` and specify a `--name` for the environment. Here we are naming our Python environment `py38`. By default, the system will create a Python 3.9 environment, so we have to specify our Python version with `python=3.8`.

Click here to view code image

```
# type this in the (bash) terminal, not in
python
conda create -n py38 python=3.8
```

After running the command, you will see the following output.

Click here to view code image

```
Collecting package metadata
(current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location:
/Users/danielchen/anaconda3/envs/py38

  added / updated specs:
    - python=3.8


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    ca-certificates-2022.07.19 |       hca03da5_0
```

```
124 KB
    certifi-2022.6.15          |   py38hca03da5_0
153 KB
    libffi-3.4.2               |       hc377ac9_4
106 KB
    ncurses-6.3                |       h1a28f6b_3
866 KB
    openssl-1.1.1q             |       h1a28f6b_0
2.2 MB
    pip-22.1.2                 |   py38hca03da5_0
2.5 MB
    python-3.8.13              |       hbdb9e5c_0
10.6 MB
    setuptools-63.4.1          |   py38hca03da5_0
1.1 MB
    sqlite-3.39.2              |       h1058600_0
1.1 MB
    ----------------------------------------------
--------------
                                           Total:
18.6 MB

The following NEW packages will be INSTALLED:

  ca-certificates pkgs/main/osx-arm64::ca-
certificates-2022.07.19-hca03da5_0
  certifi         pkgs/main/osx-arm64::certifi-
2022.6.15-py38hca03da5_0
  libcxx           pkgs/main/osx-arm64::libcxx-
12.0.0-hf6beb65_1
  libffi          pkgs/main/osx-arm64::libffi-
3.4.2-hc377ac9_4
  ncurses         pkgs/main/osx-arm64::ncurses-
```

```
6.3-h1a28f6b_3
  openssl          pkgs/main/osx-arm64::openssl-
1.1.1q-h1a28f6b_0
  pip              pkgs/main/osx-arm64::pip-22.1.2-
py38hca03da5_0
  python           pkgs/main/osx-arm64::python-
3.8.13-hbdb9e5c_0
  readline         pkgs/main/osx-arm64::readline-
8.1.2-h1a28f6b_1
  setuptools       pkgs/main/osx-arm64::setuptools-
63.4.1-py38hca03da5_0
  sqlite           pkgs/main/osx-arm64::sqlite-
3.39.2-h1058600_0
  tk               pkgs/main/osx-arm64::tk-8.6.12-
hb8d0fd4_0
  wheel            pkgs/main/noarch::wheel-0.37.1-
pyhd3eb1b0_0
  xz               pkgs/main/osx-arm64::xz-5.2.5-
h1a28f6b_1
  zlib             pkgs/main/osx-arm64::zlib-
1.2.12-h5a0b063_2


Proceed ([y]/n)? y

Downloading and Extracting Packages
certifi-2022.6.15    | 153 KB    |
########################## | 100%
python-3.8.13        | 10.6 MB   |
########################## | 100%
openssl-1.1.1q       | 2.2 MB    |
########################## | 100%
setuptools-63.4.1    | 1.1 MB    |
```

```
########################## | 100%
ca-certificates-2022 | 124 KB     |
########################## | 100%
pip-22.1.2           | 2.5 MB     |
########################## | 100%
sqlite-3.39.2        | 1.1 MB     |
########################## | 100%
ncurses-6.3          | 866 KB     |
########################## | 100%
libffi-3.4.2         | 106 KB     |
########################## | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate py38
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

The last few lines of the output tell you how you can use your newly created environment. If we run `conda activate py38` from the command line now, our prompt will be prepended with our environment name. If we run `python` in the terminal to launch Python, you will see that a different version of Python is now being used.

Click here to view code image

```
% python

Python 3.8.13 (default, Mar 28 2022, 06:13:39)
```

```
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license"
for more information.
```

To delete an environment, navigate to your `anaconda3` folder. A folder there called `envs` stores all your environments. In this example, if we delete the `py38` folder within `envs`, it's as if we never created our environment, and it will be removed.

Within a given environment, any package or library we install (Appendix H) within it will be specific to that particular environment. Thus, we can have not only different versions of Python between environments but also different versions of libraries. You can create a separate Python environment (`p4e` for "Pandas for Everyone") for this book as well.}

Click here to view code image

```
conda create --name p4e python=3
```

You can install the libraries needed by following the instructions in Appendix H.

G.2 Pyenv + Pipenv

Calvin Hendryx-Parker gave a great talk at PyCon 2022 on *Bootstrapping Your Local Python Environment* that goes over the Pyenv setup with the pyenv-virtualenv plugin: `https://www.youtube.com/watch?v=-YEUFGFHWgQ`

The Hitchhiker's Guide to Python and *Real Python* also have resources on using Pipenv for virtual environments:

- `https://docs.python-guide.org/dev/virtualenvs/#virtualenvironments-ref`
- `https://realpython.com/pipenv-guide/`

H

Install Packages

There will be times when you have to install a Python package that did not come with your distribution. If you used Anaconda to install Python, then you will have a package manager called `conda`.

`conda` has gained popularity over the past few years because of its ability to install Python packages that require non-Python dependencies. You may have heard of other package managers, such as `pip`.

This book uses a few packages that need to be installed. If you installed the entire Anaconda distribution, then libraries like Pandas are already installed. But there's no harm in running the command to reinstall a library. Check the accompanying repository[1] for all the commands to install the relevant libraries for this book.

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

We can use `conda` to install Python libraries. If you created a separate environment for the book (Appendix G), then you can `conda activate p4e` to get into the "Pandas for Everyone" environment.

`conda`'s default repository is maintained by Anaconda, Inc (formerly known as Continuum Analytics). We can install the `pandas` package using `conda`.

Click here to view code image

```
# typed into your terminal, not in Python
conda install pandas
```

For certain packages that are not listed in the default channel, or if the default channel does not have the latest version of a package, we can use the and community-maintained `conda-forge` channel.[2]

2. https://conda-forge.org/

Click here to view code image

```
conda install -c conda-forge pandas
```

Lastly, if the package isn't listed in `conda`, you can also use `pip` to install packages.

```
pip install pandas
```

For example, to install all the libraries used in this book, you can run the following lines:

Click here to view code image

```
conda install -c conda-forge pandas matplotlib
pyarrow openpyxl \
  seaborn numba regex pandas-datareader
statsmodels scikit-learn \
  arrow lifelines
```

Again, it's a good idea to check the accompanying repository for the most recent installation and setup instructions.

H.1 Updating Packages

You can update `conda` itself with the following command:

```
conda update conda
```

Run this command to update all the packages in a given `conda` environment:

```
conda update --all
```

I

Importing Libraries

Libraries provide additional functionality in an organized and packaged way. We mainly work with the Pandas library throughout this book, but there are times when we will import other libraries. You will see many different ways to import a library. The most basic way is to simply import the library by its name.

```
import pandas
```

When we import a library, we can use its functions within Pandas using dot notation.

Click here to view code image

```
print(pandas.read_csv('data/concat_1.csv'))
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

Python gives us a way to alias libraries. This allows us to use an abbreviation for longer library names. To do so, we specify the alias after the `as` statement.

```
import pandas as pd
```

Now, instead of referring to the library as `pandas`, we can use our abbreviation, `pd`.

Click here to view code image

```
print(pd.read_csv('data/concat_1.csv'))
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

Sometimes, if only a few functions are needed from a library, we can import them directly.

```
from pandas import read_csv
```

This will allow us to use the `read_csv()` function directly, without specifying the library it is coming from.

Click here to view code image

```
print(read_csv('data/concat_1.csv'))
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

Finally, there is a method that enables users to import all the functions of a library directly into the namespace.

```
from pandas import *
from numpy import *
from scipy import *
```

This method is not recommended because libraries contain many functions, and a function can mask an existing function. For example, if we import all the functions from `numpy` and from `scipy`, which `mean()` function is used? It's not as clear as saying `numpy.mean()` and `scipy.mean()`.

J

Code Style

The Python Enhancement Proposal 8 (PEP8) discusses the official Python code style guide: `https://peps.python.org/pep-0008/`.

Reading through the style guide is a good way to learn the syntax of a language. Just keep in mind that you do not need to adhere to every single rule.

Tools like Black[1] have been created for Python so your code can be automatically formatted. This is useful so you can have the tool do your formatting for you, and it's one thing less for you to worry about.

1. `https://github.com/psf/black`

While writing this book, I used the online black playground, to format some of the code: `https://black.vercel.app/`. Not every piece of code in the book follows PEP8 or Black. Sometimes, the code puts in additional line breaks to emphasize the code being taught.

J.1 Line Breaks in Code

Writing analysis code does get very wide at times. An additional constraint in the book is that the code needs to be even more narrow compared to the PEP8 rules.

There are two ways you can break up wide lines of code.

1. Using the \ at the end of a line to tell Python that the code continues on the next line
2. Wrapping your entire statement around a pair of round parentheses `( )`

Let's use the example from Section 4.3.

Click here to view code image

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

The first step in tidying up the data set was to call the `.melt()` method.

Click here to view code image

```
# this code is wide and will run off the page
weather_melt = weather.melt(id_vars=["id",
"year", "month", "element"],
var_name="day", value_name="temp")
```

This ends up being a wide line of code. So we can put in line breaks between the round parenthesis of the `.melt()` method call.

Click here to view code image

```
# previous line of code can be rewritten as
weather_melt = weather.melt(
  id_vars=["id", "year", "month", "element"],
  var_name="day",
  value_name="temp",
)
```

In Pandas, many of the methods can be chained together (Appendix U). A common practice is to put each method call on its own line. This way if your eyes look down a straight line, you can get a rough overview of all the steps your data is going through. However, just putting arbitrary line breaks outside of a function call does not work.

Click here to view code image

```
# this will error, putting line break before the
.melt
```

```
# previous line of code can be rewritten as
weather_melt = weather
  .melt(
    id_vars=["id", "year", "month", "element"],
    var_name="day",
    value_name="temp")
```

```
IndentationError: unexpected indent
(3804754158.py, line 4)
```

We can solve this by using one of the techniques listed above

Click here to view code image

```
# use a \ at the end of the line
weather_melt = weather \
  .melt(
    id_vars=["id", "year", "month", "element"],
    var_name="day",
    value_name="temp")
```

```
# wrap the entire statement around ( )
weather_melt = (weather
  .melt(
    id_vars=["id", "year", "month", "element"],
    var_name="day",
    value_name="temp")
)
```

The `( )` method is the style you will see more often reading Pandas code.

K

Containers: Lists, Tuples, and Dictionaries

Python comes with built-in container objects. These objects store data and are also `iterable`, meaning there is a mechanism to iterate through the values stored in the container.

K.1 Lists

Lists are a fundamental data structure in Python. They are used to store heterogeneous data and are created with a pair of square brackets, `[ ]`.

Click here to view code image

```
my_list = ['a', 1, True, 3.14]
print(my_list)
```

```
['a', 1, True, 3.14]
```

We can subset the list using square brackets and provide the index of the item we want.

Click here to view code image

```
# get the first item - index 0
print(my_list[0])
```

```
a
```

We can also pass in a range of values (Appendix P).

```
# get the first 3 values
print(my_list[:3])
```

```
['a', 1, True]
```

We can reassign values when we subset values from the list.

```
# reassign the first value
my_list[0] = 'zzzzz'
print(my_list)
```

```
['zzzzz', 1, True, 3.14]
```

Lists are objects in Python (Appendix S), so they will have methods that they can perform. For example, we can `.append()` values to the list.

Click here to view code image

```
my_list.append('appended a new value!')
print(my_list)
```

```
['zzzzz', 1, True, 3.14, 'appended a new value!']
```

More about lists and their various methods can be found in the documentation.[1]

1. `https://docs.python.org/3/tutorial/datastructures.html#more-on-lists`

K.2 Tuples

A `tuple` is similar to a `list`, in that both can hold heterogeneous bits of information. The main difference is that the contents of a tuple are “immutable”, meaning they cannot be changed. They are created with a pair of round parentheses, `( )`.

Click here to view code image

```
my_tuple  =('a', 1, True, 3.14)
print(my_tuple)
```

```
('a', 1, True, 3.14)
```

Subsetting items can be accomplished in the same ways as for a list (i.e., you use square brackets).

```
# get the first item
print(my_tuple[0])
```

```
a
```

However, if we try to change the contents of an index, we will get an error.

Click here to view code image

```
# this will cause an error
my_tuple[0] = 'zzzzz'
```

```
TypeError: 'tuple' object does not support item
assignment
```

More information about tuples can be found in the documentation.[2]

2. https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences

K.3 Dictionaries

Python dictionaries (`dict`) are efficient ways of storing information. Just as an actual dictionary stores a word and its corresponding definition, a Python `dict` stores some key and a corresponding value. Using dictionaries can make your code more readable because a label is assigned to each value in the dictionary. Contrast this with `list` objects, which are unlabeled. Dictionaries are created by using a set of curly braces, `{ }`.

```
my_dict = {}
print(my_dict)
```

```
{}
```

```
print(type(my_dict))
```

```
<class 'dict'>
```

When we have a `dict`, we can add values to it by using square brackets, `[ ]`. We put the key inside these square brackets. Usually, it is some string, but it can actually be any immutable type (e.g., a Python `tuple`, which is the immutable form of a Python `list`). Here we create two keys, `fname` and `lname`, for a first name and last name, respectively.

```
my_dict['fname'] = 'Daniel'
my_dict['lname'] = 'Chen'
```

We can also create a dictionary directly, with key–value pairs instead of adding them one at a time. To do this, we use our curly braces, `{ }`, with the key–value pairs being specified by a colon.

Click here to view code image

```
my_dict = {'fname': 'Daniel', 'lname': 'Chen'}
print(my_dict)
```

```
{'fname': 'Daniel', 'lname': 'Chen'}
```

To get the values from our keys, we can use the square brackets with the key inside.

```
fn = my_dict['fname']
print(fn)
```

```
Daniel
```

We can also use the `.get()` method.

```
ln = my_dict.get('lname')
print(ln)
```

```
Chen
```

The main difference between these two ways of getting the values from the dictionary is the behavior that occurs when you try to get a nonexistent key. When using the square-bracket notation, trying to get a key that does not exist will return an error.

```
# will return an error
print(my_dict['age'])
```

```
KeyError: 'age'
```

In contrast, the `.get()` method will return `None`.

```
# will return None
print(my_dict.get('age'))
```

```
None
```

To get all the `keys` from the `dict`, we can use the `.keys()` method.

Click here to view code image

```
# get all the keys in the dictionary
print(my_dict.keys())
```

```
dict_keys(['fname', 'lname'])
```

To get all the values from the dict, we can use the .values() method.

Click here to view code image

```
# get all the values in the dictionary
print(my_dict.values())
```

```
dict_values(['Daniel', 'Chen'])
```

To get every key–value pair, you can use the .items() method. This can be useful if you need to loop through a dictionary.

Click here to view code image

```
print(my_dict.items())
```

```
dict_items([('fname', 'Daniel'), ('lname',
'Chen')])
```

Each key–value pair is returned in a form of a tuple, as indicated by the use of round parentheses, ( ).

More on dictionaries can be found in the official documentation on data structures.[3]

3. https://docs.python.org/3/tutorial/datastructures.html#dictionaries

L

Slice Values

Slicing details were also described in Section 11.1.1.

Python is a zero-indexed language (things start counting from zero), and is also left inclusive, right exclusive you are when specifying a range of values. This applies to objects like `lists` and `Series`, where the first element has a position (index) of 0. When creating `ranges` or slicing a range of values from a list-like object, we need to specify both the beginning index and the ending index. This is where the left inclusive, right exclusive terminology comes into play. The left index will be included in the returned range or slice, but the right index will not.

Think of items in a list-like object as being fenced in. The index represents the fence post. When we specify a range or a slice, we are actually referring to the fence posts, so that everything between the posts is returned.

Figure L.1 illustrates why this may be the case. When we slice from 0 to 1, we get only one value back; when we slice from 1 to 3, we get two values back.

```
l = ['one', 'two', 'three']

print(l[0:1])
```

```
['one']
```

```
print(l[1:3])
```

```
['two', 'three']
```

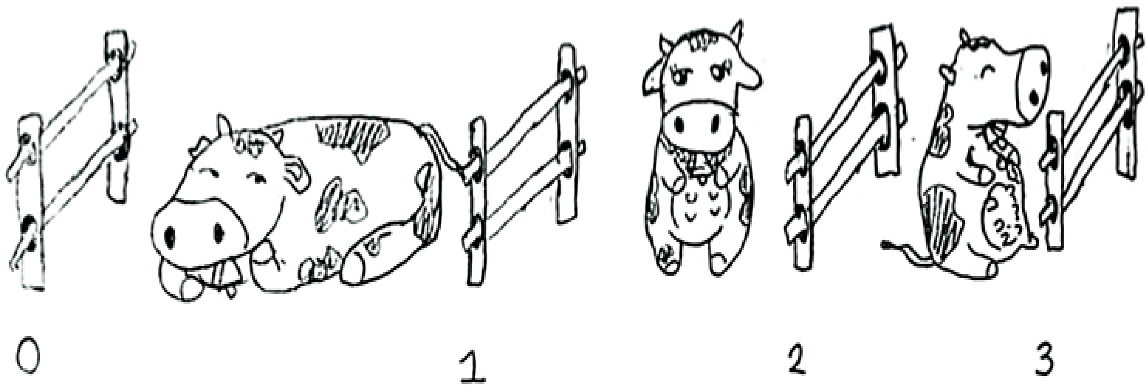


Figure L.1 Think of Slicing Values as Referring to the Fence Posts

The slicing notation used, `:`, comes in two parts. The value on the left denotes the starting value (left inclusive), and the value on the right denotes the ending value (right exclusive). We can leave one of these values blank, and the slicing will start from the beginning (if we leave the left value blank) or go to the end (if we leave the right value blank).

```
print(l[1:])
```

```
['two', 'three']
```

```
print(l[:3])
```

```
['one', 'two', 'three']
```

We can add a second colon, which refers to the “step”. For example, if we have a step value of `2`, then for whatever range we specified using the first colon, the returned value will be every other value from the range.

Click here to view code image

```
# get every other value starting from the first
value
print(l[::2])
```

```
['one', 'three']
```

M

Loops

Loops provide a means to perform the same action across multiple items. Multiple items are typically stored in a Python `list` object. Any list-like object can be iterated over (e.g., tuples, arrays, dataframes, dictionaries). More information on loops can be found in the Software-Carpentry Python lesson on loops.[1]

1. https://swcarpentry.github.io/python-novice-inflammation/05-loop/index.html

To loop over a list. we use a `for` statement. The basic `for` loop looks like this:

```
for item in container:
  # do something
```

The `container` represents some iterable set of values (e.g., a `list`). The `item` represents a temporary variable that represents each item in the iterable. In the `for` statement, the first element of the container is assigned to the temporary variable (in this example, `item`). Everything in the indented block after the colon is then performed. When it gets to the end of the loop, the code assigns the next element in the iterable to the temporary variable and performs the steps over again.

Click here to view code image

```
# an example list of values to iterate over
l = [1, 2, 3]

# write a for loop that prints the value and its
squared value
```

```python
for i in l:
  # print the current value
  print(f"the current value is: {i}")

  # print the square of the value
  print(f"its squared value is: {i*i}")

  # end of the loop, the \n at the end creates a new line
  print("end of loop, going back to the top\n")
```

```
the current value is: 1
its squared value is: 1
end of loop, going back to the top

the current value is: 2
its squared value is: 4
end of loop, going back to the top

the current value is: 3
its squared value is: 9
end of loop, going back to the top
```

N

Comprehensions

A typical task in Python is to iterate over a list, run some function on each value, and save the results into a new list.

Click here to view code image

```
# create a list
l = [1, 2, 3, 4, 5]

# list of newly calculated results
r = []

# iterate over the list
for i in l:
  # square each number and add the new value to
a new list
  r.append(i ** 2)

print(r)
```

```
[1, 4, 9, 16, 25]
```

Unfortunately, this approach requires a few lines of code to do a relatively simple task. One way to rewrite this loop more compactly is by using a Python list comprehension. This shortcut offers a concise way of performing the same action.

Click here to view code image

```
# note the square brackets around on the right-hand side
# this saves the final results as a list
rc = [i ** 2 for i in l]
print(rc)
```

```
[1, 4, 9, 16, 25]
```

```
print(type(rc))
```

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

Our final results will be a list, so the right-hand side will have a pair of square brackets. From there, we write what looks very similar to a `for` loop. Starting from the center and moving toward the right side, we write `for i in l`, which is very similar to the first line of our original `for` loop. On the right side, we write `i ** 2`, which is similar to the body of the `for` loop. Since we are using a list comprehension, we no longer need to specify the list to which we want to append our new values.

O

Functions

Functions are one of the cornerstones of programming. They provide a way to reuse code. If you've ever copy-pasted lines of code just to change a few parameters, then turning those lines of code into a function not only makes your code more readable but also prevents you from making mistakes later on. Every time code is copy-pasted, it adds another place to look if a correction is needed, and puts that burden on the programmer. When you use a function, you need to make a correction only once, and it will be applied every time the function is called.

I highly suggest the Software-Carpentry Python episode on functions for more details.[1] An empty function looks like this:

1. https://swcarpentry.github.io/python-novice-inflammation/08-func/index.html

```
def empty_function():
  pass
```

The function begins with the `def` keyword, then the function name (i.e., how the function will be called and used), a set of round brackets, and a colon. The body of the function is indented (one tab or four spaces). This indentation is *extremely* important. If you omit it, you will get an error. In this example, `pass` is used as a placeholder to do nothing.

Typically functions will have what's called a "docstring"—a multiple-line comment that describes the function's purpose, parameters, and output, and that sometimes contains testing code. When you look up help documentation about a function in Python, the information contained in the function docstring is usually what shows up. This allows the function's documentation and code to travel together, which makes the documentation easier to maintain.

Click here to view code image

```
def empty_function():
  """This is an empty function with a docstring.
  These docstrings are used to help document the
function.
  They can be created by using 3 single quotes
or 3 double quotes.
  The PEP-8 style guide says to use double
quotes.
  """
  pass # this function still does nothing
```

Functions need not have parameters to be called.

Click here to view code image

```
def print_value():
  """Just prints the value 3
  """
  print(3)
```

```
# call our print_value function
print_value()
```

```
3
```

Functions can take parameters as well. We can modify our `print_value()` function so that it prints whatever value we pass into the function.

Click here to view code image

```
def print_value(value):
  """Prints the value passed into the parameter
'value'
```

```
    """
    print(value)
```

```
print_value(3)
```

```
3
```

```
print_value("Hello!")
```

```
Hello!
```

Functions can take multiple values as well.

Click here to view code image

```
def person(fname, lname):
    """A function that takes 3 values, and prints
them
    """
    print(fname)
    print(lname)
```

```
person('Daniel', 'Chen')
```

```
Daniel
Chen
```

The examples thus far have simply created functions that printed values. What makes functions powerful is their ability to take inputs and return an output, not just print values to the screen. To accomplish this, we can use the `return` statement.

Click here to view code image

```
def my_mean_2(x, y):
  """A function that returns the mean of 2
values
  """
  mean_value = (x + y) / 2
  return mean_value

m = my_mean_2(0, 10)
print(m)
```

```
5.0
```

O.1 Default Parameters

Functions can also have default values. In fact, many of the functions found in various libraries have default values. These defaults allow users to type less because users now have to specify just a minimal amount of information for the function, but also give users the flexibility to make changes to the function's behavior if desired. Default values are also useful if you have your own functions and want to add more features without breaking your existing code.

Click here to view code image

```
def my_mean_3(x, y, z=20):
  """A function with a parameter z that has a
default value
  """
  # you can also directly return values without
having to create
  # an intermediate variable
  return (x + y + z) / 3
```

Here we **need** to specify only `x` and `y`.

```
print(my_mean_3(10, 15))
```

```
15.0
```

We can also specify `z` if we want to override its default value.

```
print(my_mean_3(0, 50, 100))
```

```
50.0
```

O.2 Arbitrary Parameters

Sometimes function documentation includes the terms `*args` and `**kwargs`. These stand for “arguments” and “keyword arguments”, respectively. They allow the function author to capture an arbitrary number of arguments into the function. They may also provide a means for the user to pass arguments into another function that is called within the current function.

O.2.1 *args

Let’s write a more generic `mean()` function that can take an arbitrary number of values.

Click here to view code image

```
def my_mean(*args):
  """Calculate the mean for an arbitrary number
of values
  """
  # add up all the values
  sum = 0
  for i in args:
      sum += i
  return sum / len(args)
```

.

```
print(my_mean(0, 10))
```

```
5.0
```

```
print(my_mean(0, 50, 100))
```

```
50.0
```

```
print(my_mean(3, 10, 25, 2))
```

```
10.0
```

O.2.2 **kwargs

`**kwargs` is similar to `*args`, but instead of acting like an arbitrary list of values, they are used like a dictionary—that is, they specify arbitrary pairs of key–value stores.

Click here to view code image

```
def greetings(welcome_word, **kwargs):
  """Prints out a greeting to a person,
  where the person's fname and lname are
provided by the kwargs
  """
  print(welcome_word)
  print(kwargs.get('fname'))
  print(kwargs.get('lname'))
```

```
greetings('Hello!', fname='Daniel',
lname='Chen')
```

Hello!
Daniel
Chen

P

Ranges and Generators

The Python `range()` function allows the user to create a sequence of values by providing a starting value, an ending value, and if needed, a step value. It is very similar to the slicing syntax in Appendix L. By default, if we give `range()` a single number, this function will create a sequence of values starting from `0`.

```
# create a range of 5
r = range(5)
```

However, the `range()` function doesn't just return a list of numbers. In Python 3, it actually returns a generator.

```
print(r)
```

```
range(0, 5)
```

```
print(type(r))
```

```
<class 'range'>
```

If we wanted an actual `list` of the range, we can convert the generator to a list.

```
lr = list(range(5))
print(lr)
```

```
[0, 1, 2, 3, 4]
```

Before you decide to convert a generator, you should think carefully about what you plan to use it for. If you plan to create a generator that will look over a set of data (Appendix M), then there is no need to convert the generator.

```
for i in lr:
  print(i)
```

```
0
1
2
3
4
```

Generators create the next value in the sequence on the fly. As a consequence, the entire contents of the generator do not need to be loaded into memory before using it. Since generators know only the current position and how to calculate the next item in the sequence, you cannot use generators a second time.

The following example comes from the built-in `itertools` library in Python. It creates a Cartesian product of values provided to the function.

Click here to view code image

```
import itertools
prod = itertools.product([1, 2, 3], ['a', 'b',
'c'])

for i in prod:
  print(i)
```

```
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
```

```
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
```

If you need to reuse the Cartesian product again, then you would have to either re-create the generator object or convert the generator into something more static (e.g., a `list`).

Click here to view code image

```
# this will not work because we already used
this generator
for i in prod:
  print(i)
```

```
# create a new generator
prod = itertools.product([1, 2, 3], ['a', 'b',
'c'])
for i in prod:
  print(i)
```

```
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
```

If all you are doing is creating something to iterate over once, it will save you a lot of computer memory if you do not convert it into a `list` object, since Python will just create the object as it goes, instead of trying to store the entire thing at once.

Q

Multiple Assignment

Multiple assignment in Python is a form of syntactic sugar. It provides the programmer with the ability to express something succinctly while making this information easier to express and to be understood by others.

As an example, let's use a list of values.

```
l = [1, 2, 3]
```

If we want to assign a variable to each element of this list, we can subset the list and assign the value.

```
a = l[0]
b = l[1]
c = l[2]
```

```
print(a)
```

```
1
```

```
print(b)
```

```
2
```

```
print(c)
```

```
3
```

With multiple assignment, if the statement to the right is some kind of container, we can directly assign its values to multiple variables on the left. So, the preceding code can be rewritten as follows:

```
a1, b1, c1 = l
```

```
print(a1)
```

```
1
```

```
print(b1)
```

```
2
```

```
print(c1)
```

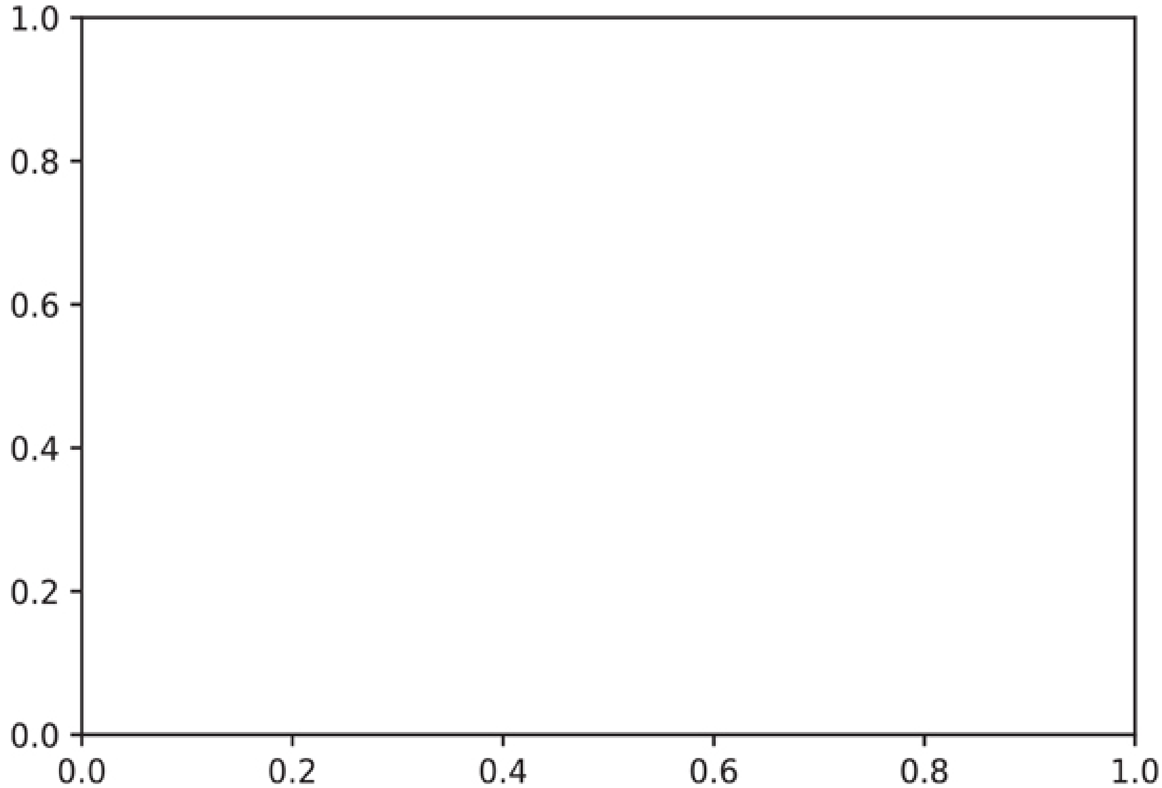
```
3
```

Multiple assignment is often used when generating figures and axes while plotting data.

Click here to view code image

```
import matplotlib.pyplot as plt

f, ax = plt.subplots()
```



This one-line command will create the figure and the axes. Other use cases can be seen in the following Stack Overflow question: https://stackoverflow.com/questions/5182573/multiple-assignment-semantics

R

Numpy ndarray

The `numpy` library[1] gives Python the ability to work with matrices and arrays.

1. https://numpy.org/doc/stable/

```
import numpy as np
```

Pandas started off as an extension to `numpy.ndarray` that provided more features suitable for data analysis. Since then, Pandas has evolved to the point that it shouldn't be thought of as a collection of `numpy` arrays, since the two libraries are different.

Click here to view code image

```
import pandas as pd

df = pd.read_csv('data/concat_1.csv')
print(df)
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

If you do need to get the `numpy.ndarray values` from a `Series` or `DataFrame`, you can use the `values` attribute.

Click here to view code image

```
a = df['A']
print(a)
```

```
0    a0
1    a1
2    a2
3    a3
Name: A, dtype: object
```

```
print(type(a))
```

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

```
print(a.values)
```

```
['a0' 'a1' 'a2' 'a3']
```

```
print(type(a.values))
```

```
<class 'numpy.ndarray'>
```

This is particularly helpful when cleaning data in Pandas. You can then use your newly cleaned data in other Python libraries that do not fully support the `Series` and `DataFrame` objects. The Software-Carpentry Python Inflammation lesson[2] uses `numpy` and can be another good reference to learn about the library and Python as a whole.

2. https://swcarpentry.github.io/python-novice-inflammation/

S

Classes

Python is an object-oriented language, meaning that everything you create or use is a “class”. Classes allow the programmer to group relevant functions and methods together. In Pandas, `Series` and `DataFrame` are classes, and each has its own attributes (e.g., `.shape`) and methods (e.g., `.apply()`). While it’s not this book’s intention to give a lesson on object-oriented programming, I want to very quickly cover classes, with the hope that this information will help you navigate the official documentation and understand why things are the way they are.

What’s nice about classes is that the programmer can define any class for their intended purpose. The following class represents a person. There are a first name (`fname`), a last name (`lname`), and an age (`age`) associated with each person. When the person celebrates their birthday (`celebrate_birthday`), the age increases by 1.

Click here to view code image

```
class Person(object):
  def __init__(self, fname, lname, age):
    self.fname = fname
    self.lname = lname
    self.age = age

  def celebrate_birthday(self):
    self.age += 1
    return(self)
```

With the `Person` class created, we can use it in our code. Let’s create an instance of our `Person`.

Click here to view code image

```
ka = Person(fname='King', lname='Arthur',
age=39)
```

This created a `Person`—King Arthur, age 39—and saved him to a variable named `ka`. We can then get some attributes from `ka` (note that attributes are not functions or methods, so they do not have round brackets).

```
print(ka.fname)
```

```
King
```

```
print(ka.lname)
```

```
Arthur
```

```
print(ka.age)
```

```
39
```

Finally, we can call the method on our class to increment the `age`.

```
ka.celebrate_birthday()
print(ka.age)
```

```
40
```

The Pandas `Series` and `DataFrame` objects are more complex versions of our `Person` class. The general concepts are the same, though. We can instantiate any new class to a variable, and access its attributes or call its methods.

T

SettingWithCopyWarning

The `SettingWithCopyWarning` is just a warning, so your code will still run and produce a result. However, if you do see this warning, it is a "code smell" that maybe you need to re-write something in your code.

Let's work with one of our small example data sets to recreate the warning.

Click here to view code image

```
import pandas as pd

dat = pd.read_csv("data/concat_1.csv")
print(dat)
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

T.1 Modifying a Subset of Data

It's pretty common to subset your data for values you need, and then make changes to that subset.

Click here to view code image

```python
subset = dat[["A", "C"]]
print(subset)
```

```
    A   C
0  a0  c0
1  a1  c1
2  a2  c2
3  a3  c3
```

```python
# this will trigger the warning
subset["new"] = ["bunch", "of", "new", "values"]
print(subset)
```

```
    A   C     new
0  a0  c0   bunch
1  a1  c1      of
2  a2  c2     new
3  a3  c3  values
```

```
/var/folders/2b/qckmp39n7qn1dh0tpcm8g89w0000gn/T/i
pykernel_29772/
4023129152.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice
from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value
instead

See the caveats in the documentation:
https://pandas.pydata.org/
pandas-
docs/stable/user_guide/indexing.html#returning-a-
```

```
view-versus-a-copy
  subset["new"] = ["bunch", "of", "new", "values"]
```

This goes into how Python passes things by reference, so Pandas does not know *for certain* if you are working on a subsetted copy of the original dataframe, or want to make changes to the original dataframe.

The way we fix this is to be explicit when we are working with a subset of the data we plan to modify.

Click here to view code image

```
subset = dat[["A", "C"]].copy() # explicity copy
print(subset)
```

```
   A   C
0  a0  c0
1  a1  c1
2  a2  c2
3  a3  c3
```

```
# no more warning!
subset["new"] = ["bunch", "of", "new", "values"]
print(subset)
```

```
   A   C     new
0  a0  c0   bunch
1  a1  c1      of
2  a2  c2     new
3  a3  c3  values
```

In longer analysis and data processing scripts, the `SettingWithCopyWarning` is not always "close" to where the subsetting happened, so you may need to trace your code back to where you made a copy to your data set. There were a few points in the text book where we made `.copy()` calls. This was to avoid the `SettingWithCopyWarning`.

T.2 Replacing a Value

When you want to replace a particular value in a dataframe, make sure you do the entire replacement in a single `.loc[]` or `.iloc[]` call.

Click here to view code image

```
# reset our data
dat = pd.read_csv("data/concat_1.csv")
print(dat)
```

```
   A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

If you filter your rows and columns in separate steps, you will also run into the `SettingWithCopyWarning`.

Click here to view code image

```
# want to  replace the c2 value
# filter the rows and separately select the
column
dat.loc[dat["C"] == "c2"]["C"] = "new value"

print(dat)
```

```
   A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
```

```
2 a2 b2 c2 d2
3 a3 b3 c3 d3

/var/folders/2b/qckmp39n7qn1dh0tpcm8g89w0000gn/T/i
pykernel_29772/
3306879196.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice
from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value
instead

See the caveats in the documentation:
https://pandas.pydata.org/
pandas-
docs/stable/user_guide/indexing.html#returning-a-
view-versus-a-copy
  dat.loc[dat["C"] == "c2"]["C"] = "new value"
```

Instead, you want to do the entire replacement in a single step.

Click here to view code image

```
dat = pd.read_csv("data/concat_1.csv")
dat.loc[dat["C"] == "c2", ["C"] ] = "new value"
print(dat)
```

```
   A   B          C   D
0  a0  b0        c0  d0
1  a1  b1        c1  d1
2  a2  b2 new value  d2
3  a3  b3        c3  d3
```

T.3 More Resources

For more detail, there is a great blog post by Benjamin Pryke for Dataquest that walks you through this warning: https://www.dataquest.io/blog/settingwithcopywarning/

Kevin Markham from Data School also has a great YouTube video on the topic titled *How do I avoid a SettingWithCopyWarning in pandas*: https://www.youtube.com/watch?v=4R4WsDJ-KVc

U

Method Chaining

Objects in Python usually have methods that modify the existing object. This means that we can call methods sequentially without having to save out our results in intermediate results.

If we use the same `Person` class from Appendix S.

Click here to view code image

```
class Person(object):
    def __init__(self, fname, lname, age):
        self.fname = fname
        self.lname = lname
        self.age = age

    def celebrate_birthday(self):
        self.age += 1
        return(self)
```

We can method chain our results if we wanted our person to have two consecutive birthdays.

Click here to view code image

```
ka = Person(fname='King', lname='Arthur',
age=39)
print(ka.age)
```

```
39
```

```python
# King Arthur has 2 birthdays in a row!
ka.celebrate_birthday().celebrate_birthday()
```

```
<__main__.Person at 0x1039903a0>
```

```python
print(ka.age)
```

```
41
```

We can do something similar in Pandas in Section 4.3 where we tidied up our `weather` data.

Click here to view code image

```python
import pandas as pd

weather = pd.read_csv('data/weather.csv')
print(weather.head())
```

```
       id year month element  d1   d2   d3  d4
d5  d6   ... \
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   ...
3 MX17004 2010     2    tmin NaN 14.4 14.4 NaN
NaN NaN   ...
4 MX17004 2010     3    tmax NaN  NaN  NaN NaN
32.1 NaN   ...
```

```
   d22   d23  d24  d25  d26  d27  d28  d29   d30  d31
0  NaN   NaN  NaN  NaN  NaN  NaN  NaN  NaN  27.8  NaN
1  NaN   NaN  NaN  NaN  NaN  NaN  NaN  NaN  14.5  NaN
2  NaN  29.9  NaN  NaN  NaN  NaN  NaN  NaN   NaN  NaN
3  NaN  10.7  NaN  NaN  NaN  NaN  NaN  NaN   NaN  NaN
4  NaN   NaN  NaN  NaN  NaN  NaN  NaN  NaN   NaN  NaN

[5 rows x 35 columns]
```

We first needed to `.melt()` our date, then `.pivot_table()`, and finally `.reset_index()`. Instead of doing each of the steps in separate parts, we can work as if the results returned them`self`.

Click here to view code image

```
weather_tidy = (
    weather
    .melt(
        id_vars=["id", "year", "month",
"element"],
        var_name="day",
        value_name="temp",
    )
    .pivot_table(
        index=["id", "year", "month", "day"],
        columns="element",
        values="temp",
    )
    .reset_index()
)

print(weather_tidy)
```

```
element       id year month day tmax tmin
0       MX17004 2010     1 d30 27.8 14.5
1       MX17004 2010     2 d11 29.7 13.4
2       MX17004 2010     2  d2 27.3 14.4
3       MX17004 2010     2 d23 29.9 10.7
4       MX17004 2010     2  d3 24.1 14.4
..          ...  ...   ... ...  ...  ...
28      MX17004 2010    11 d27 27.7 14.2
29      MX17004 2010    11 d26 28.1 12.1
30      MX17004 2010    11  d4 27.2 12.0
31      MX17004 2010    12  d1 29.9 13.8
32      MX17004 2010    12  d6 27.8 10.5

[33 rows x 6 columns]
```

V

Timing Code

If you're running Python in an IPython instance (e.g., Jupyter Notebook, Jupyter Lab, or IPython directly), you have access to "magic" commands that allow you to easily perform non-Python tasks.

Magic commands are called with `%` or `%%`. In a Jupyter Notebook the `%timeit` will time a line of code and `%%timeit` will time the entire cell of code.

Let's time the different vectorization methods from Chapter 5.

Click here to view code image

```
import pandas as pd
import numpy as np
import numba


def avg_2(x, y):
  return (x + y) / 2


@np.vectorize
def v_avg_2_mod(x, y):
  """Calculate the average, unless x is 20
  Same as before, but we are using the vectorize
decorator
  """
  if (x == 20):
```

```
    return(np.NaN)
  else:
    return (x + y) / 2

@numba.vectorize
def v_avg_2_numba(x, y):
  """Calculate the average, unless x is 20
  Using the numba decorator.
  """
  # we now have to add type information to our
function
  if (int(x) == 20):
    return(np.NaN)
  else:
    return (x + y) / 2

df = pd.DataFrame({"a": [10, 20, 30], "b": [20,
30, 40]})
print(df)
```

```
    a   b
0  10  20
1  20  30
2  30  40
```

Timing the different methods.

Click here to view code image

```
%%timeit
avg_2(df['a'], df['b'])
```

```
67.1 µs ± 12.7 µs per loop (mean ± std. dev. of 7
runs, 10,000 loops each)
```

```
%%timeit
v_avg_2_mod(df['a'], df['b'])
```

```
16.6 µs ± 1.05 µs per loop (mean ± std. dev. of 7
runs, 100,000 loops each)
```

```
%%timeit
v_avg_2_numba(df['a'].values, df['b'].values)
```

```
3.92 µs ± 632 ns per loop (mean ± std. dev. of 7
runs, 100,000 loops each)
```

The first method isn't even as flexible as the custom functions we created. If you are working with mathematical calculations, you can get performance benefits from changing the library you are using. Otherwise, using `vectorize()` can also help you write more readable `apply` code.

W

String Formatting

W.1 C-Style

An older way to perform string formatting in Python is with the `%` operator. This follows the C `printf` style formatting. The `str.format()` method (Appendix Section W.2) is preferred over the C-style formatting, and if you are using Python 3.6+ you should be using formatted string literals (f-strings) described in Section 11.4. Nonetheless, you may still find code examples that use this formatting style.

We won't go too much into detail about this method, but here are some of the Section 11.4 examples recreated using the C `printf` style formatting.

For digits we can use the `%d` placeholder, here, the `d` represents an integer digit.

Click here to view code image

```
s = 'I only know %d digits of pi' % 7
print(s)
```

```
I only know 7 digits of pi
```

For strings, we can use the `s` placeholder. Note the string pattern uses round parentheses `( )`, instead of curly braces { }. The variable passed is a Python dict, which uses `{ }`.

Click here to view code image

```
print(
    "Some digits of %(cont)s: %(value).2f"
    % {"cont": "e", "value": 2.718}
)
```

```
Some digits of e: 2.72
```

W.2 String Formatting: `.format()` Method

The format string syntax[1] was superseded with formatted string literals (i.e., f-strings) in Python 3.6.

1. https://docs.python.org/3/library/string.html#formatstrings

To format character strings with `.format()`, you essentially write a string with special placeholder characters, `{ }`, and use the `.format()` method on the string to insert values into the placeholder.

```
var = 'flesh wound'
s = "It's just a {}!"
```

```
print(s.format(var))
```

```
It's just a flesh wound!
```

```
print(s.format('scratch'))
```

```
It's just a scratch!
```

The placeholders can also refer to variables multiple times.

Click here to view code image

```
# using variables multiple times by index
s = """Black Knight: 'Tis but a {0}.
King Arthur: A {0}? Your arm's off!
"""
print(s.format('scratch'))
```

```
Black Knight: 'Tis but a scratch.
King Arthur: A scratch? Your arm's off!
```

You can also give the placeholders a variable.

Click here to view code image

```
s = 'Hayden Planetarium Coordinates: {lat},
{lon}'
print(s.format(lat='40.7815° N', lon='73.9733°
W'))
```

```
Hayden Planetarium Coordinates: 40.7815° N,
73.9733° W
```

W.3 Formatting Numbers

Numbers can also be formatted.

Click here to view code image

```
print('Some digits of pi:
{}'.format(3.14159265359))
```

```
Some digits of pi: 3.14159265359
```

You can even format numbers and use thousands-place comma separators.

Click here to view code image

```
print(
  "In 2005, Lu Chao of China recited {:,} digits
of pi".format(67890)
)
```

```
In 2005, Lu Chao of China recited 67,890 digits of
pi
```

Numbers can be used to perform a calculation and formatted to a certain number of decimal values. Here we can calculate a proportion and format it into a percentage.

Click here to view code image

```
# the 0 in {0:.4} and {0:.4%} refer to the 0
index in this format
# the .4 refers to how many decimal values, 4
# if we provide a %, it will format the decimal
as a percentage
print(
  "I remember {0:.4} or {0:.4%} of what Lu Chao
recited".format(
     7 / 67890
  )
)
```

```
I remember 0.0001031 or 0.0103% of what Lu Chao
recited
```

Finally, you can use string formatting to pad a number with zeros, similar to how `zfill` works on strings. When working with data, this method may be useful when working with ID numbers that were read in as numbers but should be strings.

Click here to view code image

```
# the first 0 refers to the index in this format
# the second 0 refers to the character to fill
# the 5 in this case refers to how many
characters in total
# the d signals a digit will be used
# Pad the number with 0s so the entire string
has 5 characters
print("My ID number is {0:05d}".format(42))
```

```
My ID number is 00042
```

X

Conditionals (if-elif-else)

Conditional statements allow your script or program to have "control flow". We have the option of using the `if`, `elif`, and `else` statements.

Let's combine these examples into a simplified version of a popular programming interview problem: Fizz Buzz.

If the number we want to check is a multiple of 2, we want to print `"fizz"`. We can use the modulo operator in Python, `%`, to give us the remainder of a number after division. So, a number is a multiple of `2` if the modulo (i.e., remainder) is `0`. If that statement is true it will run the code in that `if` block (denoted by the indentation).

```
my_num = 4

if my_num % 2 == 0:
  print("fizz")
```

```
fizz
```

If we put multiple `if` statements after one another it will run through each of them in order.

```
my_num = 4

if my_num % 2 == 0:
  print("fizz")
if my_num % 4 == 0:
  print("buzz")
```

```
fizz
buzz
```

```python
my_num = 6

if my_num % 3 == 0:
  print("fizz")
if my_num % 4 == 0:
  print("buzz")
```

```
fizz
```

Sometimes we only want the code to run the first `True` statement. This is useful if we only care about one of the conditions, but also so we are not making unnecessary calculations. We can put subsequent conditions in an `elif` (for "else if") block.

```python
my_num = 4

if my_num % 2 == 0:
  print("fizz")
elif my_num % 4 == 0:
  print("buzz")
```

```
fizz
```

Finally, we can use the `else` block to capture all the results if nothing else before it is `True`.

Click here to view code image

```python
my_num = 7

if my_num % 2 == 0:
  print("fizz")
```

```python
elif my_num % 4 == 0:
  print("buzz")
else:
  print("Not multiple of 2 or 4.")
```

```
Not multiple of 2 or 4.
```

Y

New York ACS Logistic Regression Example

Click here to view code image

```
import pandas as pd

acs = pd.read_csv('data/acs_ny.csv')
print(acs.columns)
```

```
Index(['Acres', 'FamilyIncome', 'FamilyType',
'NumBedrooms', 'NumChildren',
       'NumPeople', 'NumRooms', 'NumUnits',
'NumVehicles', 'NumWorkers',
       'OwnRent', 'YearBuilt', 'HouseCosts',
'ElectricBill', 'FoodStamp',
       'HeatingFuel', 'Insurance', 'Language'],
     dtype='object')
```

```
print(acs.head())
```

```
 Acres FamilyIncome  FamilyType NumBedrooms
NumChildren NumPeople \
0 1-10          150     Married           4
1          3
1 1-10          180 Female Head           3
2          4
```

```
2 1-10         280 Female Head          4
0         2
3 1-10         330 Female Head          2
1         2
4 1-10         330   Male Head          3
1         2

   Num         Num        Num          Num
Own       Year  \
  Rooms        Units      Vehicles   Workers
Rent      Built
0     9 Single detached        1         0
Mortgage   1950-1959
1     6 Single detached        2         0
Rented Before 1939
2     8 Single detached        3         1
Mortgage   2000-2004
3     4 Single detached        1         0
Rented   1950-1959
4     5 Single attached        1         0
Mortgage Before 1939

   House  Electric  Food   Heating Insurance
Language
   Costs  Bill      Stamp   Fuel
0   1800        90      No        Gas      2500
English
1    850        90      No        Oil         0
English
2   2600       260      No        Oil      6600 Other
European
3   1800       140      No        Oil         0
```

```
English
4     860        150       No         Gas          660
Spanish
```

To model these data, we first need to create a binary response variable. Here we split the `FamilyIncome` variable into a binary variable.

Click here to view code image

```
acs["ge150k"] = pd.cut(
     acs["FamilyIncome"],
     [0, 150000, acs["FamilyIncome"].max()],
     labels=[0, 1],
)

acs["ge150k_i"] = acs["ge150k"].astype(int)
print(acs["ge150k_i"].value_counts())
```

```
0    18294
1     4451
Name: ge150k_i, dtype: int64
```

Note

The cutoff values we used to bin our `FamilyIncome` variable with the `.cut()` function is arbitrary.

In so doing, we created a binary (0/1) variable.

Click here to view code image

```
acs.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 22745 entries, 0 to 22744
Data columns (total 20 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Acres          22745 non-null  object
 1   FamilyIncome   22745 non-null  int64
 2   FamilyType     22745 non-null  object
 3   NumBedrooms    22745 non-null  int64
 4   NumChildren    22745 non-null  int64
 5   NumPeople      22745 non-null  int64
 6   NumRooms       22745 non-null  int64
 7   NumUnits       22745 non-null  object
 8   NumVehicles    22745 non-null  int64
 9   NumWorkers     22745 non-null  int64
 10  OwnRent        22745 non-null  object
 11  YearBuilt      22745 non-null  object
 12  HouseCosts     22745 non-null  int64
 13  ElectricBill   22745 non-null  int64
 14  FoodStamp      22745 non-null  object
 15  HeatingFuel    22745 non-null  object
 16  Insurance      22745 non-null  int64
 17  Language       22745 non-null  object
 18  ge150k         22745 non-null  category
 19  ge150k_i       22745 non-null  int64
dtypes: category(1), int64(11), object(8)
memory usage: 3.3+ MB
```

Let's subset our data with just the columns we'll use for the example.

Click here to view code image

```
acs_sub = acs[
  [
```

```
        "ge150k_i",
        "HouseCosts",
        "NumWorkers",
        "OwnRent",
        "NumBedrooms",
        "FamilyType",
    ]
].copy()

print(acs_sub)
```

```
   ge150k_i  HouseCosts  NumWorkers  OwnRent
NumBedrooms  FamilyType
0          0        1800              0 Mortgage
4      Married
1          0         850              0   Rented
3  Female Head
2          0        2600              1 Mortgage
4  Female Head
3          0        1800              0   Rented
2  Female Head
4          0         860              0 Mortgage
3    Male Head
...      ...         ...            ...      ...
...          ...
22740      1        1700              2 Mortgage
5      Married
22741      1        1300              2 Mortgage
4      Married
22742      1         410              3 Mortgage
4      Married
22743      1        1600              3 Mortgage
3      Married
```

```
22744     1        6500              2 Mortgage
4     Married

[22745 rows x 6 columns]
```

```
import statsmodels.formula.api as smf

# we break up the formula string to fit on the
page
model = smf.logit(
    "ge150k_i ~ HouseCosts + NumWorkers +
OwnRent + NumBedrooms
      + FamilyType",
    data=acs_sub,
)

results = model.fit()
```

```
Optimization terminated successfully.
         Current function value: 0.391651
         Iterations 7
```

```
print(results.summary())
```

```
                              Logit Regression
Results
==================================================
============================
Dep. Variable:                ge150k_i No.
Observations:   22745
Model:                           Logit Df Residuals:
22737
Method:                            MLE Df Model:
```

```
7
Date:                Thu, 01 Sep 2022 Pseudo R-
squ.:     0.2078
Time:                        01:57:02 Log-
Likelihood:   -8908.1
converged:                       True LL-Null:
-11244.
Covariance Type:           nonrobust LLR p-value:
0.000
==================================================
=========================================
                             coef   std err
z   P>|z|  [0.025  0.975]
--------------------------------------------------
-----------------------------------------
Intercept                 -5.8081     0.120
-48.456   0.000  -6.043  -5.573
OwnRent[T.Outright]        1.8276     0.208
8.782   0.000   1.420   2.236
OwnRent[T.Rented]         -0.8763     0.101
-8.647   0.000  -1.075  -0.678
FamilyType[T.Male Head]    0.2874     0.150
1.913   0.056  -0.007   0.582
FamilyType[T.Married]      1.3877     0.088
15.781   0.000   1.215   1.560
HouseCosts                 0.0007  1.72e-05
42.453   0.000   0.001   0.001
NumWorkers                 0.5873     0.026
22.393   0.000   0.536   0.639
NumBedrooms                0.2365     0.017
13.985   0.000   0.203   0.270
==================================================
=========================================
```

```
import statsmodels.formula.api as smf

# we break up the formula string to fit on the
page
model = smf.logit(
    "ge150k_i ~ HouseCosts + NumWorkers +
OwnRent + NumBedrooms + FamilyType",
    data=acs_sub,
)

results = model.fit()
```

```
Optimization terminated successfully.
         Current function value: 0.391651
         Iterations 7
```

```
print(results.summary())
```

```
                              Logit Regression
Results
==================================================
============================
Dep. Variable:             ge150k_i No.
Observations:   22745
Model:                        Logit Df Residuals:
22737
Method:                         MLE Df Model:
7
Date:              Thu, 01 Sep 2022 Pseudo R-
squ.:     0.2078
Time:                      01:57:02 Log-
Likelihood:   -8908.1
```

```
converged:                          True LL-Null:
-11244.
Covariance Type:           nonrobust LLR p-value:
0.000
==================================================
==========================================
                             coef    std err
z    P>|z|   [0.025   0.975]
--------------------------------------------------
------------------------------------------
Intercept                 -5.8081      0.120
-48.456   0.000  -6.043  -5.573
OwnRent[T.Outright]        1.8276      0.208
8.782   0.000   1.420   2.236
OwnRent[T.Rented]         -0.8763      0.101
-8.647   0.000  -1.075  -0.678
FamilyType[T.Male Head]    0.2874      0.150
1.913   0.056  -0.007   0.582
FamilyType[T.Married]      1.3877      0.088
15.781   0.000   1.215   1.560
HouseCosts                 0.0007   1.72e-05
42.453   0.000   0.001   0.001
NumWorkers                 0.5873      0.026
22.393   0.000   0.536   0.639
NumBedrooms                0.2365      0.017
13.985   0.000   0.203   0.270
==================================================
==========================================
```

```
import numpy as np

# exponentiate our results
```

```
odds_ratios = np.exp(results.params)
print(odds_ratios)
```

```
Intercept                   0.003003
OwnRent[T.Outright]         6.219147
OwnRent[T.Rented]           0.416310
FamilyType[T.Male Head]     1.332901
FamilyType[T.Married]       4.005636
HouseCosts                  1.000731
NumWorkers                  1.799117
NumBedrooms                 1.266852
dtype: float64
```

```
print(acs.OwnRent.unique())
```

```
['Mortgage' 'Rented' 'Outright']
```

Y.0.1 With sklearn

Click here to view code image

```
predictors = pd.get_dummies(acs_sub.iloc[:, 1:],
drop_first=True)
print(predictors)
```

```
    HouseCosts NumWorkers NumBedrooms
OwnRent_Outright OwnRent_Rented \
0         1800            0           4
0              0
1          850            0           3
0              1
2         2600            1           4
0              0
```

```
3        1800          0            2
0              1
4         860          0            3
0              0
...       ...        ...          ...
...            ...
22740    1700          2            5
0              0
22741    1300          2            4
0              0
22742     410          3            4
0              0
22743    1600          3            3
0              0
22744    6500          2            4
0              0

   FamilyType_Male Head FamilyType_Married
0                     0                  1
1                     0                  0
2                     0                  0
3                     0                  0
4                     1                  0
...                 ...                ...
22740                 0                  1
22741                 0                  1
22742                 0                  1
22743                 0                  1
22744                 0                  1

[22745 rows x 7 columns]
```

```
from sklearn import linear_model
lr = linear_model.LogisticRegression()
```

```
results = lr.fit(X = predictors, y =
acs['ge150k_i'])
```

```
/Users/danielchen/.pyenv/versions/3.10.4/envs/pfe_
book/lib/python3.10/
site-
packages/sklearn/linear_model/_logistic.py:444:
ConvergenceWarning:
lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or
scale the data as shown in:
    https://scikit-
learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for
alternative solver options:
    https://scikit-
learn.org/stable/modules/linear_model.html#logisti
c-
    regression
  n_iter_i = _check_optimize_result(
```

We can also get our coefficients in the same way.

Click here to view code image

```
print(results.coef_)
```

```
[[ 5.83764740e-04  7.29381775e-01 2.82543789e-01
7.03519146e-02
  -2.11748592e+00 -1.02984936e+00 2.50310160e-01]]
```

We can get the intercept as well.

```
print(results.intercept_)
```

```
[-4.82088401]
```

We can print out our results in a more attractive format.

Click here to view code image

```
values = np.append(results.intercept_,
results.coef_)

# get the names of the values
names = np.append("intercept",
predictors.columns)

# put everything in a labeled dataframe
results = pd.DataFrame(
    values,
    index=names,
    columns=["coef"], # you need the square
brackets here
)

print(results)
```

```
                         coef
intercept           -4.820884
HouseCosts           0.000584
```

```
NumWorkers              0.729382
NumBedrooms             0.282544
OwnRent_Outright        0.070352
OwnRent_Rented         -2.117486
FamilyType_Male Head -1.029849
FamilyType_Married      0.250310
```

In order to interpret our coefficients, we still need to exponentiate our values.

Click here to view code image

```
results['or'] = np.exp(results['coef'])
print(results)
```

```
                             coef        or
intercept               -4.820884 0.008060
HouseCosts               0.000584 1.000584
NumWorkers               0.729382 2.073798
NumBedrooms              0.282544 1.326500
OwnRent_Outright         0.070352 1.072886
OwnRent_Rented          -2.117486 0.120334
FamilyType_Male Head    -1.029849 0.357061
FamilyType_Married       0.250310 1.284424
```

Z

Replicating Results in R

Preparing the data used for this section.

Click here to view code image

```
library(MASS)

library(tidyverse)
library(tidymodels)

library(pscl)

# load the tips data
tips <- readr::read_csv("data/tips.csv")

# load the titanic data
titanic <- readr::read_csv("data/titanic.csv")

# subset the columns and drop missing values
titanic_sub <- titanic %>%
  dplyr::select(survived, sex, age, embarked)
%>%
  tidyr::drop_na()

# load the ACS data and fix the data types
acs <- readr::read_csv("data/acs_ny.csv") %>%
  dplyr::mutate( # data gets loaded differently
```

```
from pandas
    NumChildren = as.integer(NumChildren),
    FamilyIncome = as.numeric(FamilyIncome),
    NumBedrooms = as.numeric(NumBedrooms),
    HouseCosts = as.numeric(HouseCosts),
    ElectricBill = as.numeric(ElectricBill),
    NumVehicles = as.numeric(NumVehicles)
  )
```

Z.1 Linear Regression

Click here to view code image

```
r_lm <- lm(tip ~ total_bill, data = tips)
print(summary(r_lm))
```

```
Call:
lm(formula = tip ~ total_bill, data = tips)

Residuals:
    Min      1Q  Median      3Q     Max
-3.1982 -0.5652 -0.0974 0.4863 3.7434

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.920270   0.159735   5.761 2.53e-08
***
total_bill   0.105025   0.007365  14.260  < 2e-16
***
---
Signif. codes:   0 '***' 0.001 '**' 0.01 '*' 0.05
'.' 0.1 ' ' 1
```

```
Residual standard error: 1.022 on 242 degrees of
freedom
Multiple R-squared:  0.4566,    Adjusted R-
squared:  0.4544
F-statistic: 203.4 on 1 and 242 DF,  p-value: <
2.2e-16
```

```
r_lm %>%
broom::tidy()
```

```
# A tibble: 2 x 5
  term        estimate std.error statistic
p.value
  <chr>          <dbl>     <dbl>     <dbl>
<dbl>
1 (Intercept)    0.920   0.160        5.76 2.53e-
8
2 total_bill     0.105   0.00736     14.3  6.69e-
34
```

```
r_lm2 <- lm(tip ~ total_bill + size, data =
tips)
print(summary(r_lm2))
```

```
Call:
lm(formula = tip ~ total_bill + size, data = tips)

Residuals:
    Min      1Q  Median      3Q     Max
-2.9279 -0.5547 -0.0852 0.5095 4.0425
```

```
Coefficients:
               Estimate Std. Error t value
Pr(>|t|)
(Intercept)    0.668945   0.193609   3.455 0.00065
***
total_bill     0.092713   0.009115  10.172 < 2e-16
***
size           0.192598   0.085315   2.258 0.02487
*
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05
'.' 0.1 ' ' 1

Residual standard error: 1.014 on 241 degrees of
freedom
Multiple R-squared:  0.4679,    Adjusted R-
squared:  0.4635
F-statistic: 105.9 on 2 and 241 DF, p-value:  <
2.2e-16
```

```
r_lm2 %>%
broom::tidy()
```

```
# A tibble: 3 x 5
  term        estimate std.error statistic
p.value
  <chr>          <dbl>     <dbl>     <dbl>
<dbl>
1 (Intercept)   0.669    0.194        3.46 6.50e-
4
2 total_bill    0.0927   0.00911     10.2  1.88e-
20
```

```
3 size             0.193    0.0853       2.26 2.49e-
2
```

```
r_lm3 <- lm(
  tip ~ total_bill + size + sex + smoker + day +
time, data = tips
)
print(summary(r_lm3))
```

```
Call:
lm(formula = tip ~ total_bill + size + sex +
smoker + day + time,
    data = tips)

Residuals:
    Min      1Q  Median      3Q     Max
-2.8475 -0.5729 -0.1026 0.4756 4.1076

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 0.803817   0.352702   2.279   0.0236 *
total_bill  0.094487   0.009601   9.841   <2e-16
***
size         0.175992   0.089528   1.966   0.0505 .
sexMale    -0.032441   0.141612  -0.229   0.8190
smokerYes  -0.086408   0.146587  -0.589   0.5561
daySat     -0.121458   0.309742  -0.392   0.6953
daySun     -0.025481   0.321298  -0.079   0.9369
dayThur    -0.162259   0.393405  -0.412   0.6804
timeLunch   0.068129   0.444617   0.153   0.8783
---
```

```
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05
'.' 0.1 ' ' 1

Residual standard error: 1.024 on 235 degrees of
freedom
Multiple R-squared:   0.4701,   Adjusted R-
squared:  0.452
F-statistic: 26.06 on 8 and 235 DF, p-value: <
2.2e-16
```

```
r_lm3 %>%
broom::tidy()
```

```
# A tibble: 9 x 5
  term         estimate std.error statistic p.value
  <chr>           <dbl>     <dbl>     <dbl>   <dbl>
1 (Intercept)   0.804    0.353       2.28   2.36e- 2
2 total_bill    0.0945   0.00960     9.84   2.34e-19
3 size          0.176    0.0895      1.97   5.05e- 2
4 sexMale      -0.0324   0.142      -0.229  8.19e- 1
5 smokerYes    -0.0864   0.147      -0.589  5.56e- 1
6 daySat       -0.121    0.310      -0.392  6.95e- 1
7 daySun       -0.0255   0.321      -0.0793 9.37e- 1
8 dayThur      -0.162    0.393      -0.412  6.80e- 1
9 timeLunch     0.0681   0.445       0.153  8.78e- 1
```

Z.2 Logistic Regression

Click here to view code image

```
# fit a logistic regression model
r_logistic_glm <- glm(
  survived ~ sex + age + embarked,
```

```
  family = binomial (link = "logit"),
  data = titanic_sub
)

summary(r_logistic_glm)
```

```
Call:
glm(formula = survived ~ sex + age + embarked,
family =
binomial(link = "logit"),      data = titanic_sub)

Deviance Residuals:
    Min       1Q   Median      3Q     Max
-2.1185 -0.6498 -0.5972 0.7937 2.1977

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) 2.204585   0.321796   6.851 7.34e-12
***
sexmale    -2.475962   0.190807 -12.976  < 2e-16
***
age        -0.008079   0.006550  -1.233  0.21746
embarkedQ  -1.815592   0.535031  -3.393  0.00069
***
embarkedS  -1.006949   0.236857  -4.251 2.13e-05
***
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05
'.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to
be 1)
```

```
    Null deviance: 960.90 on 711 degrees of
freedom
Residual deviance: 726.08 on 707 degrees of
freedom
AIC: 736.08

Number of Fisher Scoring iterations: 4
```

```
# get the coefficient table and calculate the
odds
res_r_glm <- r_logistic_glm %>%
  broom::tidy() %>%
  dplyr::mutate(odds = exp(estimate) %>%
round(6))

res_r_glm
```

```
# A tibble: 5 x 6
  term          estimate std.error statistic p.value
odds
  <chr>            <dbl>     <dbl>     <dbl>   <dbl>
<dbl>
1 (Intercept)    2.20     0.322         6.85 7.34e-
12  9.07
2 sexmale       -2.48     0.191       -13.0  1.67e-
38  0.0841
3 age           -0.00808  0.00655     -1.23 2.17e-
1  0.992
4 embarkedQ     -1.82     0.535       -3.39 6.90e-
4  0.163
5 embarkedS     -1.01     0.237       -4.25 2.13e-
5  0.365
```

Z.3 Poisson Regression

Click here to view code image

```
poi <- glm(
  NumBedrooms ~ HouseCosts + OwnRent,
  family=poisson(link = "log"),
  data=acs
)
```

```
summary(poi)

Call:
glm(formula = NumBedrooms ~ HouseCosts + OwnRent,
    family = poisson(link = "log"),  data = acs)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.8300  -0.2815  -0.1293  0.2890  2.8142

Coefficients:
                   Estimate Std. Error z value
Pr(>|z|)
(Intercept)       1.139e+00  6.158e-03 184.928  <
2e-16 ***
HouseCosts        6.217e-05  2.958e-06 21.017   <
2e-16 ***
OwnRentOutright  -2.659e-01  5.131e-02 -5.182
2.19e-07 ***
OwnRentRented    -1.237e-01  1.237e-02 -9.996   <
2e-16 ***
---
```

```
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05
'.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to
be 1)

    Null deviance: 7479.9 on 22744 degrees of
freedom
Residual deviance: 6839.2 on 22741 degrees of
freedom
AIC: 76477

Number of Fisher Scoring iterations: 4
```

```
poi %>%
broom::tidy()
```

```
# A tibble: 4 x 5
  term              estimate  std.error statistic
p.value
  <chr>                <dbl>      <dbl>     <dbl>
<dbl>
1 (Intercept)      1.14      0.00616       185.
0
2 HouseCosts       0.0000622 0.00000296     21.0
4.60e-98
3 OwnRentOutright -0.266     0.0513         -5.18
2.19e- 7
4 OwnRentRented   -0.124     0.0124        -10.0
1.58e-23
```

Z.3.1 Negative Binomial Regression for Overdispersion

Click here to view code image

```
od <- MASS::glm.nb(
  NumPeople ~ Acres + NumVehicles,
  data=acs,
  link=log
)
```

```
Warning in theta.ml(Y, mu, sum(w), w, limit =
control$maxit, trace
= control$trace > : iteration limit reached


Warning in theta.ml(Y, mu, sum(w), w, limit =
control$maxit, trace
= control$trace > : iteration limit reached
```

```
summary(od)
```

```
Call:
MASS::glm.nb(formula = NumPeople ~ Acres +
NumVehicles, data = acs,
    link = log, init.theta = 99662.32096)


Deviance Residuals:
    Min       1Q   Median      3Q     Max
-1.3263 -0.7064 -0.1315 0.3153 5.3101


Coefficients:
             Estimate Std. Error z value Pr(>|z|)
```

```
(Intercept)  1.033460   0.012036  85.865  < 2e-16
***
Acres10+    -0.025287   0.019301  -1.310     0.19
AcresSub 1   0.050768   0.009143   5.553 2.81e-08
***
NumVehicles  0.070067   0.003683  19.023  < 2e-16
***
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05
'.' 0.1 ' ' 1

(Dispersion parameter for Negative
Binomial(99662.32) family taken to be 1)

    Null deviance: 12127 on 22744 degrees of
freedom
Residual deviance: 11754 on 22741 degrees of
freedom
AIC: 80879

Number of Fisher Scoring iterations: 1

               Theta: 99662
           Std. Err.: 93669
Warning while fitting theta: iteration limit
reached

2 x log-likelihood: -80869.33
```

```
od %>%
  broom::tidy()
```

```
# A tibble: 4 x 5
  term        estimate std.error statistic  p.value
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept)  1.03     0.0120      85.9  0
2 Acres10+    -0.0253   0.0193      -1.31 1.90e- 1
3 AcresSub 1   0.0508   0.00914      5.55 2.81e- 8
4 NumVehicles  0.0701   0.00368     19.0  1.10e-80
```

```r
pm <- glm(
  NumChildren ~ FamilyIncome + FamilyType +
OwnRent,
  family = poisson(link="log"),
  data = acs
)

pchisq(
  2 * (logLik(od) - logLik(pm)),
  df = 1,
  lower.tail = FALSE
)
```

```
'log Lik.' 1 (df=5)
```


L PANDAS DE
OPEN
10
CEN

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Code Snippets

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

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



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

```
# by default read_csv() will read a comma separated file,
# our gapminder data set is separated by a tab
# we can use the sep parameter and indicate a tab with \t
df = pandas.read_csv('./data/gapminder.tsv', sep='\t')
# print out the data
print(df)
```

```
          country continent  year  lifeExp       pop   gdpPercap
0     Afghanistan      Asia  1952   28.801   8425333  779.445314
1     Afghanistan      Asia  1957   30.332   9240934  820.853030
2     Afghanistan      Asia  1962   31.997  10267083  853.100710
3     Afghanistan      Asia  1967   34.020  11537966  836.197138
4     Afghanistan      Asia  1972   36.088  13079460  739.981106
...           ...       ...   ...      ...       ...         ...
1699     Zimbabwe    Africa  1987   62.351   9216418  706.157306
1700     Zimbabwe    Africa  1992   60.377  10704340  693.420786
1701     Zimbabwe    Africa  1997   46.809  11404948  792.449960
1702     Zimbabwe    Africa  2002   39.989  11926563  672.038623
1703     Zimbabwe    Africa  2007   43.487  12311143  469.709298

[1704 rows x 6 columns]
```

```
import pandas as pd
df = pd.read_csv('./data/gapminder.tsv', sep='\t')
```

```
print(type(df))
```

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

```
# get the number of rows and columns
print(df.shape)
```

```
(1704, 6)
```

```python
# shape is an attribute, not a method
# this will cause an error
print(df.shape())
```

```
TypeError: 'tuple' object is not callable
```

```python
# get column names
print(df.columns)
```

```
Index(['country', 'continent', 'year', 'lifeExp', 'pop',
       'gdpPercap'],
      dtype='object')
```

```python
# get the dtype of each column
print(df.dtypes)
```

```
country       object
continent     object
year           int64
lifeExp      float64
pop            int64
gdpPercap    float64
dtype: object
```

```python
# get more information about our data
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   country    1704 non-null   object
 1   continent  1704 non-null   object
 2   year       1704 non-null   int64
 3   lifeExp    1704 non-null   float64
 4   pop        1704 non-null   int64
 5   gdpPercap  1704 non-null   float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
None
```

```python
# show the first 5 observations
print(df.head())
```

```
       country continent  year  lifeExp       pop   gdpPercap
0  Afghanistan      Asia  1952   28.801   8425333  779.445314
1  Afghanistan      Asia  1957   30.332   9240934  820.853030
2  Afghanistan      Asia  1962   31.997  10267083  853.100710
3  Afghanistan      Asia  1967   34.020  11537966  836.197138
4  Afghanistan      Asia  1972   36.088  13079460  739.981106
```

```python
# just get the country column and save it to its own variable
country_df = df['country']
```

```python
# show the first 5 observations
print(country_df.head())
```

```
0    Afghanistan
1    Afghanistan
2    Afghanistan
3    Afghanistan
4    Afghanistan
Name: country, dtype: object
```

```python
# show the last 5 observations
print(country_df.tail())
```

```
1699    Zimbabwe
1700    Zimbabwe
1701    Zimbabwe
1702    Zimbabwe
1703    Zimbabwe
Name: country, dtype: object
```

```python
# Looking at country, continent, and year
subset = df[['country', 'continent', 'year']]
```

```python
print(subset)
```

```
          country continent  year
0     Afghanistan      Asia  1952
1     Afghanistan      Asia  1957
2     Afghanistan      Asia  1962
3     Afghanistan      Asia  1967
4     Afghanistan      Asia  1972
...           ...       ...   ...
1699     Zimbabwe    Africa  1987
1700     Zimbabwe    Africa  1992
1701     Zimbabwe    Africa  1997
1702     Zimbabwe    Africa  2002
1703     Zimbabwe    Africa  2007

[1704 rows x 3 columns]
```

```python
# subset the first column based on its position.
df[0]
```

```
KeyError: 0
```

```python
country_df = df['country']
print(type(country_df))
```

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

```
print(country_df)
```

```
0       Afghanistan
1       Afghanistan
2       Afghanistan
3       Afghanistan
4       Afghanistan
           ...
1699       Zimbabwe
1700       Zimbabwe
1701       Zimbabwe
1702       Zimbabwe
1703       Zimbabwe
Name: country, Length: 1704, dtype: object
```

```
country_df_list = df[['country']] # note the double square bracket
print(type(country_df_list))
```

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

```
print(country_df_list)
```

```
          country
0     Afghanistan
1     Afghanistan
2     Afghanistan
3     Afghanistan
4     Afghanistan
...           ...
1699     Zimbabwe
1700     Zimbabwe
1701     Zimbabwe
1702     Zimbabwe
1703     Zimbabwe

[1704 rows x 1 columns]
```

```python
# using square bracket notation
print(df['country'])
```

```
0       Afghanistan
1       Afghanistan
2       Afghanistan
3       Afghanistan
4       Afghanistan
           ...
1699       Zimbabwe
1700       Zimbabwe
1701       Zimbabwe
1702       Zimbabwe
1703       Zimbabwe
Name: country, Length: 1704, dtype: object
```

```python
# using dot notation
print(df.country)
```

```
0       Afghanistan
1       Afghanistan
2       Afghanistan
3       Afghanistan
4       Afghanistan
           ...
1699       Zimbabwe
1700       Zimbabwe
1701       Zimbabwe
1702       Zimbabwe
1703       Zimbabwe
Name: country, Length: 1704, dtype: object
```

```python
print(df)
```

```
        country continent  year  lifeExp       pop   gdpPercap
0     Afghanistan      Asia  1952   28.801   8425333  779.445314
1     Afghanistan      Asia  1957   30.332   9240934  820.853030
2     Afghanistan      Asia  1962   31.997  10267083  853.100710
3     Afghanistan      Asia  1967   34.020  11537966  836.197138
4     Afghanistan      Asia  1972   36.088  13079460  739.981106
...           ...       ...   ...      ...       ...         ...
1699     Zimbabwe    Africa  1987   62.351   9216418  706.157306
1700     Zimbabwe    Africa  1992   60.377  10704340  693.420786
1701     Zimbabwe    Africa  1997   46.809  11404948  792.449960
1702     Zimbabwe    Africa  2002   39.989  11926563  672.038623
1703     Zimbabwe    Africa  2007   43.487  12311143  469.709298

[1704 rows x 6 columns]
```

```python
# get the first row
# python counts from 0
print(df.loc[0])
```

```
country      Afghanistan
continent           Asia
year                1952
lifeExp           28.801
pop              8425333
gdpPercap     779.445314
Name: 0, dtype: object
```

```python
# get the 100th row
# python counts from 0
print(df.loc[99])
```

```
country      Bangladesh
continent          Asia
year               1967
lifeExp          43.453
pop            62821884
gdpPercap    721.186086
Name: 99, dtype: object
```

```python
# get the last row
# this will cause an error
print(df.loc[-1])
```

```
KeyError: -1
```

```python
# get the last row (correctly)

# use the first value given from shape to get the number of rows
number_of_rows = df.shape[0]

# subtract 1 from the value since we want the last index value
last_row_index = number_of_rows - 1

# finally do the subset using the index of the last row
print(df.loc[last_row_index])
```

```
country         Zimbabwe
continent         Africa
year                2007
lifeExp           43.487
pop             12311143
gdpPercap     469.709298
Name: 1703, dtype: object
```

```python
# there are many ways of doing what you want
print(df.tail(n=1))
```

```
       country continent  year  lifeExp       pop   gdpPercap
1703  Zimbabwe    Africa  2007   43.487  12311143  469.709298
```

```python
# get the last row of data in different ways
subset_loc = df.loc[0]
subset_head = df.head(n=1)
```

```python
# type using loc of 1 row
print(type(subset_loc))
```

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

```python
# type of using head of 1 row
print(type(subset_head))
```

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

```python
print(df.loc[[0, 99, 999]])
```

```
         country continent  year  lifeExp       pop    gdpPercap
0    Afghanistan      Asia  1952   28.801   8425333   779.445314
99    Bangladesh      Asia  1967   43.453  62821884   721.186086
999     Mongolia      Asia  1967   51.253   1149500  1226.041130
```

```python
# get the 2nd row
print(df.iloc[1])
```

```
country      Afghanistan
continent           Asia
year                1957
lifeExp           30.332
pop              9240934
gdpPercap      820.85303
Name: 1, dtype: object
```

```python
## get the 100th row
print(df.iloc[99])
```

```
country      Bangladesh
continent          Asia
year               1967
lifeExp          43.453
pop            62821884
gdpPercap    721.186086
Name: 99, dtype: object
```

```python
# using -1 to get the last row
print(df.iloc[-1])
```

```
country        Zimbabwe
continent        Africa
year               2007
lifeExp          43.487
pop            12311143
gdpPercap    469.709298
Name: 1703, dtype: object
```

```python
## get the first, 100th, and 1000th row
print(df.iloc[[0, 99, 999]])
```

```
         country continent  year  lifeExp       pop    gdpPercap
0    Afghanistan      Asia  1952   28.801   8425333   779.445314
99    Bangladesh      Asia  1967   43.453  62821884   721.186086
999     Mongolia      Asia  1967   51.253   1149500  1226.041130
```

```python
# subset columns with loc
# note the position of the colon
# it is used to select all rows
subset = df.loc[:, ['year', 'pop']]
print(subset)
```

```
      year       pop
0     1952   8425333
1     1957   9240934
2     1962  10267083
3     1967  11537966
4     1972  13079460
...    ...       ...
1699  1987   9216418
1700  1992  10704340
1701  1997  11404948
1702  2002  11926563
1703  2007  12311143

[1704 rows x 2 columns]
```

```python
# subset columns with iloc
# iloc will allow us to use integers
# -1 will select the last column
subset = df.iloc[:, [2, 4, -1]]
print(subset)
```

```
      year       pop   gdpPercap
0     1952   8425333  779.445314
1     1957   9240934  820.853030
2     1962  10267083  853.100710
3     1967  11537966  836.197138
4     1972  13079460  739.981106
...    ...       ...         ...
```

```
1699  1987   9216418  706.157306
1700  1992  10704340  693.420786
1701  1997  11404948  792.449960
1702  2002  11926563  672.038623
1703  2007  12311143  469.709298

[1704 rows x 3 columns]
```

```
# subset columns with loc
# but pass in integer values
# this will cause an error
subset = df.loc[:, [2, 4, -1]]
print(subset)
```

```
KeyError: "None of [Int64Index([2, 4, -1], dtype='int64')]
are in the [columns]"
```

```
# subset columns with iloc
# but pass in index names
# this will cause an error
subset = df.iloc[:, ['year', 'pop']]
print(subset)
```

```
IndexError: .iloc requires numeric indexers, got ['year' 'pop']
```

```
# create a range of integers from 0 - 4 inclusive
small_range = list(range(5))
print(small_range)
```

```
[0, 1, 2, 3, 4]
```

```
# subset the dataframe with the range
subset = df.iloc[:, small_range]
print(subset)
```

```
          country continent  year  lifeExp       pop
0     Afghanistan      Asia  1952   28.801   8425333
1     Afghanistan      Asia  1957   30.332   9240934
2     Afghanistan      Asia  1962   31.997  10267083
3     Afghanistan      Asia  1967   34.020  11537966
4     Afghanistan      Asia  1972   36.088  13079460
...           ...       ...   ...      ...       ...
1699     Zimbabwe    Africa  1987   62.351   9216418
1700     Zimbabwe    Africa  1992   60.377  10704340
1701     Zimbabwe    Africa  1997   46.809  11404948
1702     Zimbabwe    Africa  2002   39.989  11926563
1703     Zimbabwe    Africa  2007   43.487  12311143

[1704 rows x 5 columns]
```

```python
# create a range from 3 - 5 inclusive
small_range = list(range(3, 6))
print(small_range)
```

```
[3, 4, 5]
```

```python
subset = df.iloc[:, small_range]
print(subset)
```

```
      lifeExp       pop   gdpPercap
0      28.801   8425333  779.445314
1      30.332   9240934  820.853030
2      31.997  10267083  853.100710
3      34.020  11537966  836.197138
4      36.088  13079460  739.981106
...       ...       ...         ...
1699   62.351   9216418  706.157306
1700   60.377  10704340  693.420786
1701   46.809  11404948  792.449960
1702   39.989  11926563  672.038623
1703   43.487  12311143  469.709298

[1704 rows x 3 columns]
```

```python
# create a range from 0 - 5 inclusive, every other integer
small_range = list(range(0, 6, 2))
subset = df.iloc[:, small_range]
print(subset)
```

```
          country  year       pop
0     Afghanistan  1952   8425333
1     Afghanistan  1957   9240934
2     Afghanistan  1962  10267083
3     Afghanistan  1967  11537966
4     Afghanistan  1972  13079460
...           ...   ...       ...
1699     Zimbabwe  1987   9216418
1700     Zimbabwe  1992  10704340
1701     Zimbabwe  1997  11404948
1702     Zimbabwe  2002  11926563
1703     Zimbabwe  2007  12311143

[1704 rows x 3 columns]
```

```python
print(df.columns)
```

```
Index(['country', 'continent', 'year', 'lifeExp', 'pop',
       'gdpPercap'],
      dtype='object')
```

```python
small_range = list(range(3))
subset = df.iloc[:, small_range]
print(subset)
```

```
          country continent  year
0     Afghanistan      Asia  1952
1     Afghanistan      Asia  1957
2     Afghanistan      Asia  1962
3     Afghanistan      Asia  1967
4     Afghanistan      Asia  1972
```

```
...          ...       ...   ...
1699     Zimbabwe    Africa  1987
1700     Zimbabwe    Africa  1992
1701     Zimbabwe    Africa  1997
1702     Zimbabwe    Africa  2002
1703     Zimbabwe    Africa  2007

[1704 rows x 3 columns]
```

```
# slice the first 3 columns
subset = df.iloc[:, :3]
print(subset)
```

```
          country continent  year
0     Afghanistan      Asia  1952
1     Afghanistan      Asia  1957
2     Afghanistan      Asia  1962
3     Afghanistan      Asia  1967
4     Afghanistan      Asia  1972
...          ...       ...   ...
1699     Zimbabwe    Africa  1987
1700     Zimbabwe    Africa  1992
1701     Zimbabwe    Africa  1997
1702     Zimbabwe    Africa  2002
1703     Zimbabwe    Africa  2007

[1704 rows x 3 columns]
```

```
small_range = list(range(3, 6))
subset = df.iloc[:, small_range]
print(subset)
```

```
      lifeExp       pop   gdpPercap
0      28.801   8425333  779.445314
1      30.332   9240934  820.853030
2      31.997  10267083  853.100710
3      34.020  11537966  836.197138
```

```
4       36.088  13079460  739.981106
...        ...       ...         ...
1699    62.351   9216418  706.157306
1700    60.377  10704340  693.420786
1701    46.809  11404948  792.449960
1702    39.989  11926563  672.038623
1703    43.487  12311143  469.709298

[1704 rows x 3 columns]
```

```
# slice columns 3 to 5 inclusive
subset = df.iloc[:, 3:6]
print(subset)
```

```
      lifeExp       pop   gdpPercap
0      28.801   8425333  779.445314
1      30.332   9240934  820.853030
2      31.997  10267083  853.100710
3      34.020  11537966  836.197138
4      36.088  13079460  739.981106
...       ...       ...         ...
1699   62.351   9216418  706.157306
1700   60.377  10704340  693.420786
1701   46.809  11404948  792.449960
1702   39.989  11926563  672.038623
1703   43.487  12311143  469.709298

[1704 rows x 3 columns]
```

```
small_range = list(range(0, 6, 2))
subset = df.iloc[:, small_range]
print(subset)
```

```
          country  year       pop
0     Afghanistan  1952   8425333
```

```
1     Afghanistan  1957   9240934
2     Afghanistan  1962  10267083
3     Afghanistan  1967  11537966
4     Afghanistan  1972  13079460
...           ...   ...       ...
1699     Zimbabwe  1987   9216418
1700     Zimbabwe  1992  10704340
1701     Zimbabwe  1997  11404948
1702     Zimbabwe  2002  11926563
1703     Zimbabwe  2007  12311143

[1704 rows x 3 columns]
```

```
# slice every other columns
subset = df.iloc[:, 0:6:2]
print(subset)
```

```
          country  year       pop
0     Afghanistan  1952   8425333
1     Afghanistan  1957   9240934
2     Afghanistan  1962  10267083
3     Afghanistan  1967  11537966
4     Afghanistan  1972  13079460
...           ...   ...       ...
1699     Zimbabwe  1987   9216418
1700     Zimbabwe  1992  10704340
1701     Zimbabwe  1997  11404948
1702     Zimbabwe  2002  11926563
1703     Zimbabwe  2007  12311143

[1704 rows x 3 columns]
```

```
# get the 1st, 100th, and 1000th rows
# from the 1st, 4th, and 6th column
# note the columns we are hoping to get are:
# country, lifeExp, and gdpPercap
print(df.iloc[[0, 99, 999], [0, 3, 5]])
```

```
         country  lifeExp    gdpPercap
0    Afghanistan   28.801   779.445314
99    Bangladesh   43.453   721.186086
999     Mongolia   51.253  1226.041130
```

```
# if we use the column names directly,
# it makes the code a bit easier to read
# note now we have to use loc, instead of iloc
print(df.loc[[0, 99, 999], ['country', 'lifeExp', 'gdpPercap']])
```

```
         country  lifeExp    gdpPercap
0    Afghanistan   28.801   779.445314
99    Bangladesh   43.453   721.186086
999     Mongolia   51.253  1226.041130
```

```
print(df.loc[10:13, :])
```

```
        country continent  year  lifeExp       pop    gdpPercap
10  Afghanistan      Asia  2002   42.129  25268405   726.734055
11  Afghanistan      Asia  2007   43.828  31889923   974.580338
12      Albania    Europe  1952   55.230   1282697  1601.056136
13      Albania    Europe  1957   59.280   1476505  1942.284244
```

```
print(df.iloc[10:13, :])
```

```
        country continent  year  lifeExp       pop    gdpPercap
10  Afghanistan      Asia  2002   42.129  25268405   726.734055
11  Afghanistan      Asia  2007   43.828  31889923   974.580338
12      Albania    Europe  1952   55.230   1282697  1601.056136
```

```python
print(df)
```

```
          country continent  year  lifeExp       pop   gdpPercap
0     Afghanistan      Asia  1952   28.801   8425333  779.445314
1     Afghanistan      Asia  1957   30.332   9240934  820.853030
2     Afghanistan      Asia  1962   31.997  10267083  853.100710
3     Afghanistan      Asia  1967   34.020  11537966  836.197138
4     Afghanistan      Asia  1972   36.088  13079460  739.981106
...           ...       ...   ...      ...       ...         ...
1699     Zimbabwe    Africa  1987   62.351   9216418  706.157306
1700     Zimbabwe    Africa  1992   60.377  10704340  693.420786
1701     Zimbabwe    Africa  1997   46.809  11404948  792.449960
1702     Zimbabwe    Africa  2002   39.989  11926563  672.038623
1703     Zimbabwe    Africa  2007   43.487  12311143  469.709298

[1704 rows x 6 columns]
```

```python
# For each year in our data, what was the average life expectancy?
# To answer this question, we need to:
# 1. split our data into parts by year
# 2. get the 'lifeExp' column
# 3. calculate the mean
print(df.groupby('year')['lifeExp'].mean())
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
          ...
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, Length: 12, dtype: float64
```

```python
# create grouped object by year
grouped_year_df = df.groupby('year')
print(type(grouped_year_df))
```

```
<class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

```python
print(grouped_year_df)
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x15fdb7df0>
```

```python
grouped_year_df_lifeExp = grouped_year_df['lifeExp']
print(type(grouped_year_df_lifeExp))
```

```
<class 'pandas.core.groupby.generic.SeriesGroupBy'>
```

```python
print(grouped_year_df_lifeExp)
```

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x106c55ae0>
```

```python
mean_lifeExp_by_year = grouped_year_df_lifeExp.mean()
print(mean_lifeExp_by_year)
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
          ...
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, Length: 12, dtype: float64
```

```python
# the backslash allows us to break up 1 long line of python code
# into multiple lines
# df.groupby(['year', 'continent'])[['lifeExp', 'gdpPercap']].mean()
# is the same as
multi_group_var = df\
  .groupby(['year', 'continent'])\
  [['lifeExp', 'gdpPercap']]\
  .mean()
```

```python
# look at the first 10 rows
print(multi_group_var)
```

```
                  lifeExp     gdpPercap
year continent
1952 Africa     39.135500   1252.572466
     Americas   53.279840   4079.062552
     Asia       46.314394   5195.484004
     Europe     64.408500   5661.057435
     Oceania    69.255000  10298.085650
...                   ...           ...
2007 Africa     54.806038   3089.032605
     Americas   73.608120  11003.031625
     Asia       70.728485  12473.026870
     Europe     77.648600  25054.481636
     Oceania    80.719500  29810.188275

[60 rows x 2 columns]
```

```python
# we can also wrap the entire statement
# around round parentheses
# with each .method() on a new line
# this is the preferred style for writing "method chaining"
multi_group_var = (
  df
  .groupby(['year', 'continent'])
  [['lifeExp', 'gdpPercap']]
  .mean()
)
```

```python
flat = multi_group_var.reset_index()
print(flat)
```

```
    year continent    lifeExp     gdpPercap
0   1952    Africa  39.135500   1252.572466
1   1952  Americas  53.279840   4079.062552
2   1952      Asia  46.314394   5195.484004
3   1952    Europe  64.408500   5661.057435
4   1952   Oceania  69.255000  10298.085650
..   ...       ...        ...           ...
55  2007    Africa  54.806038   3089.032605
56  2007  Americas  73.608120  11003.031625
57  2007      Asia  70.728485  12473.026870
58  2007    Europe  77.648600  25054.481636
59  2007   Oceania  80.719500  29810.188275

[60 rows x 4 columns]
```

```python
# use the nunique (number unique)
# to calculate the number of unique values in a series
print(df.groupby('continent')['country'].nunique())
```

```
continent
Africa      52
Americas    25
Asia        33
Europe      30
Oceania      2
Name: country, dtype: int64
```

```python
global_yearly_life_expectancy = df.groupby('year')['lifeExp'].mean()
print(global_yearly_life_expectancy)
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
          ...
1987    63.212613
1992    64.160338

1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, Length: 12, dtype: float64
```

```
# matplotlib is the default plotting library
# we need to import first
import matplotlib.pyplot as plt

# use the .plot() DataFrame method
global_yearly_life_expectancy.plot()

# show the plot
plt.show()
```

```python
import pandas as pd

s = pd.Series(['banana', 42])
print(s)
```

```
0    banana
1        42
dtype: object
```

```python
# manually assign index values to a series
# by passing a Python list
s = pd.Series(
  data=["Wes McKinney", "Creator of Pandas"],
  index=["Person", "Who"],
)

print(s)
```

```
Person          Wes McKinney
Who        Creator of Pandas
dtype: object
```

```python
scientists = pd.DataFrame(
  {
    "Name": ["Rosaline Franklin", "William Gosset"],
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
    "Age": [37, 61],
  }
)

print(scientists)
```

```
                Name    Occupation        Born        Died  Age
0  Rosaline Franklin       Chemist  1920-07-25  1958-04-16   37
1     William Gosset  Statistician  1876-06-13  1937-10-16   61
```

```python
scientists = pd.DataFrame(
  data={
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
    "Age": [37, 61],
  },
  index=["Rosaline Franklin", "William Gosset"],
  columns=["Occupation", "Born", "Died", "Age"],
)
print(scientists)
```

```
                     Occupation        Born        Died  Age
Rosaline Franklin       Chemist  1920-07-25  1958-04-16   37
William Gosset     Statistician  1876-06-13  1937-10-16   61
```

```python
# create our example dataframe
# with a row index label
scientists = pd.DataFrame(
```

```python
    data={
      "Occupation": ["Chemist", "Statistician"],
      "Born": ["1920-07-25", "1876-06-13"],
      "Died": ["1958-04-16", "1937-10-16"],
      "Age": [37, 61],
    },
    index=["Rosaline Franklin", "William Gosset"],
    columns=["Occupation", "Born", "Died", "Age"],
)

print(scientists)
```

```
                    Occupation        Born        Died  Age
Rosaline Franklin      Chemist  1920-07-25  1958-04-16   37
William Gosset    Statistician  1876-06-13  1937-10-16   61
```

```python
# select by row index label
first_row = scientists.loc['William Gosset']
print(type(first_row))
```

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

```python
print(first_row)
```

```
Occupation    Statistician
Born            1876-06-13
Died            1937-10-16
Age                     61
Name: William Gosset, dtype: object
```

```
print(first_row.index)
```

```
Index(['Occupation', 'Born', 'Died', 'Age'], dtype='object')
```

```
print(first_row.values)
```

```
['Statistician' '1876-06-13' '1937-10-16' 61]
```

```
print(first_row.keys())
```

```
Index(['Occupation', 'Born', 'Died', 'Age'], dtype='object')
```

```
# get the first index using an attribute
print(first_row.index[0])
```

```
Occupation
```

```
# get the first index using a method
print(first_row.keys()[0])
```

```
Occupation
```

```python
# calculate the mean
print(ages.mean())
```

```
49.0
```

```python
# calculate the minimum
print(ages.min())
```

```
37
```

```python
# calculate the maximum
print(ages.max())
```

```
61
```

```python
# calculate the standard deviation
print(ages.std())
```

```
16.97056274847714
```

```python
scientists = pd.read_csv('data/scientists.csv')
```

```python
        ages = scientists['Age']
        print(ages)
```

```python
print(ages[ages > ages.mean()])
```

```
1    61
2    90
3    66
7    77
Name: Age, dtype: int64
```

```
print(ages > ages.mean())
```

```
0    False
1     True
2     True
3     True
4    False
5    False
6    False
7     True
Name: Age, dtype: bool
```

```
print(type(ages > ages.mean()))
```

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

```
# get index 0, 1, 4, 5, and 7
manual_bool_values = [
```

```
  True,  # 0
  True,  # 1
  False, # 2
  False, # 3
  True,  # 4
  True,  # 5
  False, # 6
  True,  # 7
]
print(ages[manual_bool_values])
```

```
0    37
1    61
4    56
5    45
7    77
Name: Age, dtype: int64
```

```python
print(ages + pd.Series([1, 100]))
```

```
0     38.0
1    161.0
2      NaN
3      NaN
4      NaN
5      NaN
6      NaN
7      NaN
dtype: float64
```

```python
import numpy as np
```

```python
# this will cause an error
print(ages + np.array([1, 100]))
```

```python
# ages as they appear in the data
print(ages)
```

```
0    37
1    61
2    90
3    66
4    56
5    45
6    41
7    77
Name: Age, dtype: int64
```

```python
rev_ages = ages.sort_index(ascending=False)
print(rev_ages)
```

```
7    77
6    41
5    45
4    56
3    66
2    90
1    61
0    37
Name: Age, dtype: int64
```

```python
# reference output to show index label alignment
print(ages * 2)
```

```
0     74
1    122
2    180
3    132
4    112
5     90
6     82
7    154
Name: Age, dtype: int64
```

```python
# note how we get the same values
# even though the vector is reversed
print(ages + rev_ages)
```

```
0     74
1    122
2    180
3    132
4    112
5     90
6     82
7    154
Name: Age, dtype: int64
```

```python
scientists.index
```

```
RangeIndex(start=0, stop=8, step=1)
```

```python
scientists.columns
```

```
Index(['Name', 'Born', 'Died', 'Age', 'Occupation'], dtype='object')
```

```python
scientists.values
```

```
array([['Rosaline Franklin', '1920-07-25', '1958-04-16', 37, 'Chemist'],
       ['William Gosset', '1876-06-13', '1937-10-16', 61, 'Statistician'],
       ['Florence Nightingale', '1820-05-12', '1910-08-13', 90, 'Nurse'],
       ['Marie Curie', '1867-11-07', '1934-07-04', 66, 'Chemist'],
       ['Rachel Carson', '1907-05-27', '1964-04-14', 56, 'Biologist'],
       ['John Snow', '1813-03-15', '1858-06-16', 45, 'Physician'],
       ['Alan Turing', '1912-06-23', '1954-06-07', 41,
        'Computer Scientist'],
       ['Johann Gauss', '1777-04-30', '1855-02-23', 77, 'Mathematician']],
      dtype=object)
```

```
# boolean vectors will subset rows
print(scientists.loc[scientists['Age'] > scientists['Age'].mean()])
```

```
                   Name        Born        Died  Age    Occupation
1        William Gosset  1876-06-13  1937-10-16   61  Statistician
2  Florence Nightingale  1820-05-12  1910-08-13   90         Nurse
3           Marie Curie  1867-11-07  1934-07-04   66       Chemist
7          Johann Gauss  1777-04-30  1855-02-23   77 Mathematician
```

```python
first_half = scientists[:4]
second_half = scientists[4:]
```

```python
print(first_half)
```

```
                   Name        Born        Died  Age    Occupation
0     Rosaline Franklin  1920-07-25  1958-04-16   37       Chemist
1        William Gosset  1876-06-13  1937-10-16   61  Statistician
2  Florence Nightingale  1820-05-12  1910-08-13   90         Nurse
3           Marie Curie  1867-11-07  1934-07-04   66       Chemist
```

```python
print(second_half)
```

```
            Name        Born        Died  Age          Occupation
4  Rachel Carson  1907-05-27  1964-04-14   56           Biologist
5      John Snow  1813-03-15  1858-06-16   45           Physician
6    Alan Turing  1912-06-23  1954-06-07   41  Computer Scientist
7   Johann Gauss  1777-04-30  1855-02-23   77       Mathematician
```

```python
# multiply by a scalar
print(scientists * 2)
```

```
                                       Name                  Born  \
0        Rosaline FranklinRosaline Franklin  1920-07-251920-07-25
1              William GossetWilliam Gosset  1876-06-131876-06-13
2  Florence NightingaleFlorence Nightingale  1820-05-121820-05-12
3                    Marie CurieMarie Curie  1867-11-071867-11-07
4                Rachel CarsonRachel Carson  1907-05-271907-05-27
5                        John SnowJohn Snow  1813-03-151813-03-15
6                    Alan TuringAlan Turing  1912-06-231912-06-23
7                  Johann GaussJohann Gauss  1777-04-301777-04-30
```

```
                 Died  Age                      Occupation
0  1958-04-161958-04-16   74                ChemistChemist
1  1937-10-161937-10-16  122      StatisticianStatistician
2  1910-08-131910-08-13  180                    NurseNurse
3  1934-07-041934-07-04  132                ChemistChemist
4  1964-04-141964-04-14  112            BiologistBiologist
5  1858-06-161858-06-16   90            PhysicianPhysician
6  1954-06-071954-06-07   82  Computer ScientistComputer Scientist
7  1855-02-231855-02-23  154      MathematicianMathematician
```

```python
# format the 'Born' column as a datetime
born_datetime = pd.to_datetime(scientists['Born'], format='%Y-%m-%d')
print(born_datetime)
```

```
0   1920-07-25
1   1876-06-13
2   1820-05-12
3   1867-11-07
4   1907-05-27
5   1813-03-15
6   1912-06-23
7   1777-04-30
Name: Born, dtype: datetime64[ns]
```

```python
# format the 'Died' column as a datetime
died_datetime = pd.to_datetime(scientists['Died'], format='%Y-%m-%d')
```

```python
scientists['born_dt'], scientists['died_dt'] = (
  born_datetime,
  died_datetime
)

print(scientists.head())
```

```
                 Name        Born        Died  Age    Occupation  \
0   Rosaline Franklin  1920-07-25  1958-04-16   37       Chemist
1      William Gosset  1876-06-13  1937-10-16   61  Statistician
2  Florence Nightingale  1820-05-12  1910-08-13   90         Nurse
3         Marie Curie  1867-11-07  1934-07-04   66       Chemist
4       Rachel Carson  1907-05-27  1964-04-14   56     Biologist

     born_dt    died_dt
0 1920-07-25 1958-04-16
1 1876-06-13 1937-10-16
2 1820-05-12 1910-08-13
3 1867-11-07 1934-07-04
4 1907-05-27 1964-04-14
```

```python
print(scientists.shape)
```

```
(8, 7)
```

```python
print(scientists.dtypes)
```

```
Name                  object
Born                  object
Died                  object
Age                    int64

Occupation                object
born_dt       datetime64[ns]
died_dt       datetime64[ns]
dtype: object
```

```python
# the frac=1 tells pandas to randomly select 100% of the values
# the random_state makes the randomization the same each time
scientists["Age"] = scientists["Age"].sample(frac=1, random_state=42)
```

```python
# the previous line of code is equivalent to
scientists['Age'] = (
  scientists['Age']
  .sample(frac=1, random_state=42)
)
```

```python
scientists['Age'] = (
  scientists['Age']
  .sample(frac=1, random_state=42)
  .values  # remove the index so it doesn't auto align the values
)

print(scientists['Age'])
```

```
0    61
1    45
2    37
3    77
4    90
5    56
6    66
7    41
Name: Age, dtype: int64
```

```
# subtracting dates will give us number of days
scientists['age_days'] = (
  scientists['died_dt'] - scientists['born_dt']
)

print(scientists)
```

```
                   Name        Born        Died  Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16   61
1        William Gosset  1876-06-13  1937-10-16   45
2  Florence Nightingale  1820-05-12  1910-08-13   37

3           Marie Curie  1867-11-07  1934-07-04   77
4         Rachel Carson  1907-05-27  1964-04-14   90
5             John Snow  1813-03-15  1858-06-16   56
6           Alan Turing  1912-06-23  1954-06-07   66
7          Johann Gauss  1777-04-30  1855-02-23   41

          Occupation    born_dt    died_dt   age_days
0            Chemist 1920-07-25 1958-04-16 13779 days
1       Statistician 1876-06-13 1937-10-16 22404 days
2              Nurse 1820-05-12 1910-08-13 32964 days
3            Chemist 1867-11-07 1934-07-04 24345 days
4          Biologist 1907-05-27 1964-04-14 20777 days
5          Physician 1813-03-15 1858-06-16 16529 days
6  Computer Scientist 1912-06-23 1954-06-07 15324 days
7      Mathematician 1777-04-30 1855-02-23 28422 days
```

```
# we can convert the value to just the year
# using the astype method
scientists['age_years'] = (
  scientists['age_days']
  .astype('timedelta64[Y]')
)
print(scientists)
```

```
                   Name        Born        Died  Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16   61
```

```
1       William Gosset  1876-06-13  1937-10-16   45
2  Florence Nightingale  1820-05-12  1910-08-13   37
3           Marie Curie  1867-11-07  1934-07-04   77
4         Rachel Carson  1907-05-27  1964-04-14   90
5             John Snow  1813-03-15  1858-06-16   56
6           Alan Turing  1912-06-23  1954-06-07   66
7          Johann Gauss  1777-04-30  1855-02-23   41

            Occupation    born_dt    died_dt   age_days  age_years
0              Chemist 1920-07-25 1958-04-16 13779 days       37.0
1         Statistician 1876-06-13 1937-10-16 22404 days       61.0
2                Nurse 1820-05-12 1910-08-13 32964 days       90.0
3              Chemist 1867-11-07 1934-07-04 24345 days       66.0
4            Biologist 1907-05-27 1964-04-14 20777 days       56.0
5            Physician 1813-03-15 1858-06-16 16529 days       45.0
6   Computer Scientist 1912-06-23 1954-06-07 15324 days       41.0
7        Mathematician 1777-04-30 1855-02-23 28422 days       77.0
```

```
scientists = scientists.assign(
  # new columns on the left of the equal sign
  # how to calculate values on the right of the equal sign
  # separate new columns with a comma
  age_days_assign=scientists['died_dt'] - scientists['born_dt'],
  age_year_assign=scientists['age_days'].astype('timedelta64[Y]')
)

print(scientists)
```

```
                   Name        Born        Died  Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16   61
1        William Gosset  1876-06-13  1937-10-16   45
2  Florence Nightingale  1820-05-12  1910-08-13   37
3           Marie Curie  1867-11-07  1934-07-04   77
4         Rachel Carson  1907-05-27  1964-04-14   90
5             John Snow  1813-03-15  1858-06-16   56
6           Alan Turing  1912-06-23  1954-06-07   66
7          Johann Gauss  1777-04-30  1855-02-23   41
```

```
           Occupation    born_dt    died_dt   age_days  age_years  \
0             Chemist 1920-07-25 1958-04-16 13779 days       37.0
1        Statistician 1876-06-13 1937-10-16 22404 days       61.0
2               Nurse 1820-05-12 1910-08-13 32964 days       90.0
3             Chemist 1867-11-07 1934-07-04 24345 days       66.0
4           Biologist 1907-05-27 1964-04-14 20777 days       56.0
5           Physician 1813-03-15 1858-06-16 16529 days       45.0
6  Computer Scientist 1912-06-23 1954-06-07 15324 days       41.0
7       Mathematician 1777-04-30 1855-02-23 28422 days       77.0

  age_days_assign  age_year_assign
0      13779 days             37.0
1      22404 days             61.0
2      32964 days             90.0
3      24345 days             66.0
4      20777 days             56.0
5      16529 days             45.0
6      15324 days             41.0
7      28422 days             77.0
```

```python
scientists = scientists.assign(
    age_days_assign=scientists["died_dt"] - scientists["born_dt"],
    age_year_assign=lambda df_: df_["age_days_assign"].astype(
        "timedelta64[Y]"
    ),
)
print(scientists)
```

```
                   Name        Born        Died  Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16   61
1        William Gosset  1876-06-13  1937-10-16   45
2  Florence Nightingale  1820-05-12  1910-08-13   37
3           Marie Curie  1867-11-07  1934-07-04   77
4         Rachel Carson  1907-05-27  1964-04-14   90
5             John Snow  1813-03-15  1858-06-16   56
6           Alan Turing  1912-06-23  1954-06-07   66
7          Johann Gauss  1777-04-30  1855-02-23   41

           Occupation    born_dt    died_dt   age_days  age_years  \
0             Chemist 1920-07-25 1958-04-16 13779 days       37.0
1        Statistician 1876-06-13 1937-10-16 22404 days       61.0
2               Nurse 1820-05-12 1910-08-13 32964 days       90.0
3             Chemist 1867-11-07 1934-07-04 24345 days       66.0
4           Biologist 1907-05-27 1964-04-14 20777 days       56.0
5           Physician 1813-03-15 1858-06-16 16529 days       45.0
6  Computer Scientist 1912-06-23 1954-06-07 15324 days       41.0
7       Mathematician 1777-04-30 1855-02-23 28422 days       77.0

  age_days_assign  age_year_assign
0      13779 days             37.0
1      22404 days             61.0
2      32964 days             90.0
3      24345 days             66.0
```

```
4      20777 days                56.0
5      16529 days                45.0
6      15324 days                41.0
7      28422 days                77.0
```

```
# all the current columns in our data
print(scientists.columns)
```

```
Index(['Name', 'Born', 'Died', 'Age', 'Occupation', 'born_dt',
       'died_dt', 'age_days', 'age_years', 'age_days_assign',
       'age_year_assign'],
      dtype='object')
```

```
# drop the shuffled age column
# you provide the axis=1 argument to drop column-wise
scientists_dropped = scientists.drop(['Age'], axis="columns")
```

```
# columns after dropping our column
print(scientists_dropped.columns)
```

```
Index(['Name', 'Born', 'Died', 'Occupation', 'born_dt', 'died_dt',
       'age_days', 'age_years', 'age_days_assign',
       'age_year_assign'],
      dtype='object')
```

```python
names = scientists['Name']
print(names)
```

```
0       Rosaline Franklin
1          William Gosset
2    Florence Nightingale
3             Marie Curie
4           Rachel Carson
5               John Snow
6             Alan Turing
7            Johann Gauss
Name: Name, dtype: object
```

```python
# pass in a string to the path you want to save
names.to_pickle('output/scientists_names_series.pickle')
```

```python
scientists.to_pickle('output/scientists_df.pickle')
```

```python
# for a Series
series_pickle = pd.read_pickle(
    "output/scientists_names_series.pickle"
)
print(series_pickle)
```

```
0       Rosaline Franklin
1          William Gosset
2    Florence Nightingale
3             Marie Curie
4           Rachel Carson
5               John Snow
6             Alan Turing
7            Johann Gauss
Name: Name, dtype: object
```

```python
# for a DataFrame
dataframe_pickle = pd.read_pickle('output/scientists_df.pickle')
print(dataframe_pickle)
```

```
print(dataframe_pickle)
```

```
                   Name        Born        Died  Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16   61
1        William Gosset  1876-06-13  1937-10-16   45
2  Florence Nightingale  1820-05-12  1910-08-13   37
3           Marie Curie  1867-11-07  1934-07-04   77
4         Rachel Carson  1907-05-27  1964-04-14   90
5             John Snow  1813-03-15  1858-06-16   56
6           Alan Turing  1912-06-23  1954-06-07   66
7          Johann Gauss  1777-04-30  1855-02-23   41

          Occupation    born_dt    died_dt   age_days  age_years  \
0            Chemist 1920-07-25 1958-04-16 13779 days       37.0
1       Statistician 1876-06-13 1937-10-16 22404 days       61.0
2              Nurse 1820-05-12 1910-08-13 32964 days       90.0
3            Chemist 1867-11-07 1934-07-04 24345 days       66.0
4          Biologist 1907-05-27 1964-04-14 20777 days       56.0
5          Physician 1813-03-15 1858-06-16 16529 days       45.0
6 Computer Scientist 1912-06-23 1954-06-07 15324 days       41.0
7      Mathematician 1777-04-30 1855-02-23 28422 days       77.0

  age_days_assign  age_year_assign
0      13779 days             37.0
1      22404 days             61.0
2      32964 days             90.0
3      24345 days             66.0
4      20777 days             56.0
5      16529 days             45.0
6      15324 days             41.0
7      28422 days             77.0
```

```
# do not write the row names in the CSV output
scientists.to_csv('output/scientists_df_no_index.csv', index=False)
```

```python
print(names)
```

```
0        Rosaline Franklin
1           William Gosset
2    Florence Nightingale
3              Marie Curie
4            Rachel Carson
5                John Snow
6              Alan Turing
7             Johann Gauss
Name: Name, dtype: object
```

```python
# convert the Series into a DataFrame
# before saving it to an Excel file
names_df = names.to_frame()

# save to an excel file
names_df.to_excel(
  'output/scientists_names_series_df.xls', engine='openpyxl'
)
```

```python
# saving a DataFrame into Excel format
scientists.to_excel(
  "output/scientists_df.xlsx",
  sheet_name="scientists",
  index=False
)
```

```python
# save to feather file
scientists.to_feather('output/scientists.feather')

# read feather file
sci_feather = pd.read_feather('output/scientists.feather')

print(sci_feather)
```

```
                   Name        Born        Died  Age  \
0     Rosaline Franklin  1920-07-25  1958-04-16   61
1        William Gosset  1876-06-13  1937-10-16   45
2  Florence Nightingale  1820-05-12  1910-08-13   37
3           Marie Curie  1867-11-07  1934-07-04   77
4         Rachel Carson  1907-05-27  1964-04-14   90
5             John Snow  1813-03-15  1858-06-16   56
6           Alan Turing  1912-06-23  1954-06-07   66
7          Johann Gauss  1777-04-30  1855-02-23   41

          Occupation    born_dt    died_dt   age_days  age_years  \
0            Chemist 1920-07-25 1958-04-16 13779 days       37.0
1       Statistician 1876-06-13 1937-10-16 22404 days       61.0
2              Nurse 1820-05-12 1910-08-13 32964 days       90.0
3            Chemist 1867-11-07 1934-07-04 24345 days       66.0
4          Biologist 1907-05-27 1964-04-14 20777 days       56.0
5          Physician 1813-03-15 1858-06-16 16529 days       45.0
6 Computer Scientist 1912-06-23 1954-06-07 15324 days       41.0
7      Mathematician 1777-04-30 1855-02-23 28422 days       77.0

  age_days_assign  age_year_assign
0      13779 days             37.0
1      22404 days             61.0
2      32964 days             90.0
3      24345 days             66.0
4      20777 days             56.0
5      16529 days             45.0
6      15324 days             41.0
7      28422 days             77.0
```

```python
# first 2 rows of data
sci_sub_dict = scientists.head(2)

# convert the dataframe into a dictionary
sci_dict = sci_sub_dict.to_dict()

# using the pretty print library to print the dictionary
import pprint
pprint.pprint(sci_dict)
```

```
{'Age': {0: 61, 1: 45},
 'Born': {0: '1920-07-25', 1: '1876-06-13'},
 'Died': {0: '1958-04-16', 1: '1937-10-16'},
 'Name': {0: 'Rosaline Franklin', 1: 'William Gosset'},
 'Occupation': {0: 'Chemist', 1: 'Statistician'},
 'age_days': {0: Timedelta('13779 days 00:00:00'),
              1: Timedelta('22404 days 00:00:00')},
 'age_days_assign': {0: Timedelta('13779 days 00:00:00'),
                     1: Timedelta('22404 days 00:00:00')},

'age_year_assign': {0: 37.0, 1: 61.0},
'age_years': {0: 37.0, 1: 61.0},
'born_dt': {0: Timestamp('1920-07-25 00:00:00'),
            1: Timestamp('1876-06-13 00:00:00')},
'died_dt': {0: Timestamp('1958-04-16 00:00:00'),
            1: Timestamp('1937-10-16 00:00:00')}}
```

```python
# read in the dictionary object back into a dataframe
sci_dict_df = pd.DataFrame.from_dict(sci_dict)
print(sci_dict_df)
```

```
                Name        Born        Died  Age    Occupation  \
0  Rosaline Franklin  1920-07-25  1958-04-16   61       Chemist
1     William Gosset  1876-06-13  1937-10-16   45  Statistician

     born_dt    died_dt   age_days  age_years age_days_assign  \
0 1920-07-25 1958-04-16 13779 days       37.0      13779 days
1 1876-06-13 1937-10-16 22404 days       61.0      22404 days

   age_year_assign
0             37.0
1             61.0
```

```python
# convert the dataframe into a dictionary
sci_json = sci_sub_dict.to_json(
  orient='records', indent=2, date_format="iso"
)
```

```python
pprint.pprint(sci_json)
```

```
('[\n'
 '  {\n'
 '    "Name":"Rosaline Franklin",\n'
 '    "Born":"1920-07-25",\n'
 '    "Died":"1958-04-16",\n'
 '    "Age":61,\n'
 '    "Occupation":"Chemist",\n'
 '    "born_dt":"1920-07-25T00:00:00.000Z",\n'
 '    "died_dt":"1958-04-16T00:00:00.000Z",\n'
 '    "age_days":"P13779DT0H0M0S",\n'
 '    "age_years":37.0,\n'
 '    "age_days_assign":"P13779DT0H0M0S",\n'
 '    "age_year_assign":37.0\n'
 '  },\n'
 '  {\n'
 '    "Name":"William Gosset",\n'
 '    "Born":"1876-06-13",\n'
 '    "Died":"1937-10-16",\n'
 '    "Age":45,\n'
 '    "Occupation":"Statistician",\n'
 '    "born_dt":"1876-06-13T00:00:00.000Z",\n'
 '    "died_dt":"1937-10-16T00:00:00.000Z",\n'
 '    "age_days":"P22404DT0H0M0S",\n'
 '    "age_years":61.0,\n'
 '    "age_days_assign":"P22404DT0H0M0S",\n'
 '    "age_year_assign":61.0\n'
 '  }\n'
 ']')
```

```
# copy the string to re-create the dataframe
sci_json_df = pd.read_json(
  ('[\n'
 '  {\n'
 '    "Name":"Rosaline Franklin",\n'
 '    "Born":"1920-07-25",\n'
 '    "Died":"1958-04-16",\n'
 '    "Age":61,\n'
 '    "Occupation":"Chemist",\n'
 '    "born_dt":"1920-07-25T00:00:00.000Z",\n'
 '    "died_dt":"1958-04-16T00:00:00.000Z",\n'
 '    "age_days":"P13779DT0H0M0S",\n'
 '    "age_years":37.0,\n'
 '    "age_days_assign":"P13779DT0H0M0S",\n'
 '    "age_year_assign":37.0\n'
 '  },\n'
 '  {\n'
```

```
 '    "Name":"William Gosset",\n'
 '    "Born":"1876-06-13",\n'
 '    "Died":"1937-10-16",\n'
 '    "Age":45,\n'
 '    "Occupation":"Statistician",\n'
 '    "born_dt":"1876-06-13T00:00:00.000Z",\n'
 '    "died_dt":"1937-10-16T00:00:00.000Z",\n'
 '    "age_days":"P22404DT0H0M0S",\n'
 '    "age_years":61.0,\n'
 '    "age_days_assign":"P22404DT0H0M0S",\n'
 '    "age_year_assign":61.0\n'
 '  }\n'
 ']'),
  orient="records"
)
print(sci_json_df)
```

```
                Name        Born        Died  Age    Occupation  \
0  Rosaline Franklin  1920-07-25  1958-04-16   61       Chemist
1     William Gosset  1876-06-13  1937-10-16   45  Statistician

                    born_dt                   died_dt  \
0  1920-07-25T00:00:00.000Z  1958-04-16T00:00:00.000Z
1  1876-06-13T00:00:00.000Z  1937-10-16T00:00:00.000Z

         age_days  age_years age_days_assign  age_year_assign
0  P13779DT0H0M0S         37  P13779DT0H0M0S               37
1  P22404DT0H0M0S         61  P22404DT0H0M0S               61
```

```python
sci_json_df["died_dt_json"] = pd.to_datetime(sci_json_df["died_dt"])

print(sci_json_df)
```

```
               Name        Born        Died  Age    Occupation  \
0  Rosaline Franklin  1920-07-25  1958-04-16   61       Chemist
1     William Gosset  1876-06-13  1937-10-16   45  Statistician

                    born_dt                   died_dt  \
0  1920-07-25T00:00:00.000Z  1958-04-16T00:00:00.000Z
1  1876-06-13T00:00:00.000Z  1937-10-16T00:00:00.000Z

         age_days  age_years age_days_assign  age_year_assign  \
0  P13779DT0H0M0S         37  P13779DT0H0M0S               37
1  P22404DT0H0M0S         61  P22404DT0H0M0S               61

               died_dt_json
0 1958-04-16 00:00:00+00:00
1 1937-10-16 00:00:00+00:00
```

```python
print(sci_json_df.dtypes)
```

```
Name                            object
Born                            object
Died                            object
Age                              int64
Occupation                      object
born_dt                         object
died_dt                         object
age_days                        object
age_years                        int64
age_days_assign                 object
age_year_assign                  int64
died_dt_json       datetime64[ns, UTC]
dtype: object
```

```python
# the anscombe data set can be found in the seaborn library
import seaborn as sns
anscombe = sns.load_data set("anscombe")
print(anscombe)
```

```
  data set     x      y
0        I  10.0   8.04
1        I   8.0   6.95
2        I  13.0   7.58
3        I   9.0   8.81
4        I  11.0   8.33
..     ...   ...    ...
39      IV   8.0   5.25
40      IV  19.0  12.50
41      IV   8.0   5.56
42      IV   8.0   7.91
43      IV   8.0   6.89

[44 rows x 3 columns]
```

```python
import matplotlib.pyplot as plt
```

```python
# create a subset of the data
# contains only data set 1 from anscombe
data set_1 = anscombe[anscombe['data set'] == 'I']

plt.plot(data set_1['x'], data set_1['y'])
plt.show()  # will need this to show explicitly show the plot
```

```python
plt.plot(data set_1['x'], data set_1['y'], 'o')
plt.show()
```

```python
# create subsets of the anscombe data
data set_2 = anscombe[anscombe['data set'] == 'II']
data set_3 = anscombe[anscombe['data set'] == 'III']
data set_4 = anscombe[anscombe['data set'] == 'IV']
```

```python
# create the entire figure where our subplots will go
fig = plt.figure()

# tell the figure how the subplots should be laid out
# in the example, we will have
# 2 row of plots, and each row will have 2 plots

# subplot has 2 rows and 2 columns, plot location 1
axes1 = fig.add_subplot(2, 2, 1)

# subplot has 2 rows and 2 columns, plot location 2
axes2 = fig.add_subplot(2, 2, 2)

# subplot has 2 rows and 2 columns, plot location 3
axes3 = fig.add_subplot(2, 2, 3)
```

```python
# subplot has 2 rows and 2 columns, plot location 4
axes4 = fig.add_subplot(2, 2, 4)

plt.show()
```

```
# you need to run all the plotting code together, same as above
fig = plt.figure()
axes1 = fig.add_subplot(2, 2, 1)
axes2 = fig.add_subplot(2, 2, 2)
axes3 = fig.add_subplot(2, 2, 3)
axes4 = fig.add_subplot(2, 2, 4)

# add a plot to each of the axes created above
axes1.plot(data set_1['x'], data set_1['y'], 'o')
axes2.plot(data set_2['x'], data set_2['y'], 'o')
axes3.plot(data set_3['x'], data set_3['y'], 'o')
axes4.plot(data set_4['x'], data set_4['y'], 'o')

plt.show()
```

```
# you need to run all the plotting code together, same as above
fig = plt.figure()
axes1 = fig.add_subplot(2, 2, 1)
axes2 = fig.add_subplot(2, 2, 2)
axes3 = fig.add_subplot(2, 2, 3)
axes4 = fig.add_subplot(2, 2, 4)
axes1.plot(data set_1['x'], data set_1['y'], 'o')
axes2.plot(data set_2['x'], data set_2['y'], 'o')
axes3.plot(data set_3['x'], data set_3['y'], 'o')
axes4.plot(data set_4['x'], data set_4['y'], 'o')

# add a small title to each subplot
axes1.set_title("data set_1")
axes2.set_title("data set_2")
axes3.set_title("data set_3")
axes4.set_title("data set_4")

# add a title for the entire figure (title above the title)
fig.suptitle("Anscombe Data") # note spelling of "suptitle"

# use a tight layout so the plots and titles don't overlap
fig.set_tight_layout(True)

# show the figure
plt.show()
```

```python
tips = sns.load_data set("tips")
print(tips)
```

```
     total_bill   tip     sex smoker   day    time  size
0         16.99  1.01  Female     No   Sun  Dinner     2
1         10.34  1.66    Male     No   Sun  Dinner     3
2         21.01  3.50    Male     No   Sun  Dinner     3
3         23.68  3.31    Male     No   Sun  Dinner     2
4         24.59  3.61  Female     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       29.03  5.92    Male     No   Sat  Dinner     3
240       27.18  2.00  Female    Yes   Sat  Dinner     2
241       22.67  2.00    Male    Yes   Sat  Dinner     2
242       17.82  1.75    Male     No   Sat  Dinner     2
243       18.78  3.00  Female     No  Thur  Dinner     2

[244 rows x 7 columns]
```

```python
# create the figure object
fig = plt.figure()

# subplot has 1 row, 1 column, plot location 1
axes1 = fig.add_subplot(1, 1, 1)

# make the actual histogram
axes1.hist(data=tips, x='total_bill', bins=10)

# add labels
axes1.set_title('Histogram of Total Bill')
axes1.set_xlabel('Frequency')
axes1.set_ylabel('Total Bill')

plt.show()
```

```python
# create the figure object
scatter_plot = plt.figure()
axes1 = scatter_plot.add_subplot(1, 1, 1)

# make the actual scatter plot
axes1.scatter(data=tips, x='total_bill', y='tip')

# add labels
axes1.set_title('Scatterplot of Total Bill vs Tip')
axes1.set_xlabel('Total Bill')
axes1.set_ylabel('Tip')

plt.show()
```

```python
# create the figure object
boxplot = plt.figure()
axes1 = boxplot.add_subplot(1, 1, 1)
```

```
# make the actual box plot
axes1.boxplot(
  # first argument of box plot is the data
  # since we are plotting multiple pieces of data
  # we have to put each piece of data into a list
  x=[
      tips.loc[tips["sex"] == "Female", "tip"],
      tips.loc[tips["sex"] == "Male", "tip"],
  ],
  # we can then pass in an optional labels parameter
  # to label the data we passed
  labels=["Female", "Male"],
)


# add labels
axes1.set_xlabel('Sex')
axes1.set_ylabel('Tip')
axes1.set_title('Boxplot of Tips by Gender')

plt.show()
```

```
# assign color values
colors = {
    "Female": "#f1a340",  # orange
    "Male": "#998ec3",    # purple
}

scatter_plot = plt.figure()
axes1 = scatter_plot.add_subplot(1, 1, 1)

axes1.scatter(
  data=tips,
  x='total_bill',
  y='tip',

  # set the size of the dots based on party size
  # we multiply the values by 10 to make the points bigger
  # and also to emphasize the difference
  s=tips["size"] ** 2 * 10,

  # set the color for the sex using our color values above
  c=tips['sex'].map(colors),

  # set the alpha so points are more transparent
  # this helps with overlapping points
  alpha=0.5
)

# label the axes
axes1.set_title('Colored by Sex and Sized by Size')
axes1.set_xlabel('Total Bill')
axes1.set_ylabel('Tip')

# figure title on top
scatter_plot.suptitle("Total Bill vs Tip")

plt.show()
```

```
# load seaborn if you have not done so already
import seaborn as sns

tips = sns.load_data set("tips")
```

```
# set the default seaborn context optimized for paper print
# the default is "notebook"
sns.set_context("paper")
```

```
# the subplots function is a shortcut for
# creating separate figure objects and
# adding individual subplots (axes) to the figure
hist, ax = plt.subplots()

# use seaborn to draw a histogram into the axes
sns.histplot(data=tips, x="total_bill", ax=ax)

# use matplotlib notation to set a title
ax.set_title('Total Bill Histogram')

# use matplotlib to show the figure
plt.show()
```

```
den, ax = plt.subplots()

sns.kdeplot(data=tips, x="total_bill", ax=ax)

ax.set_title('Total Bill Density')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Unit Probability')

plt.show()
```

```
rug, ax = plt.subplots()

# plot 2 things into the axes we created
sns.rugplot(data=tips, x="total_bill", ax=ax)
sns.histplot(data=tips, x="total_bill", ax=ax)
```

```
ax.set_title("Rug Plot and Histogram of Total Bill")
ax.set_title("Total Bill")

plt.show()
```

```
# the FacetGrid object creates the figure and axes for us
fig = sns.displot(data=tips, x="total_bill", kde=True, rug=True)

fig.set_axis_labels(x_var="Total Bill", y_var="Count")
fig.figure.suptitle('Distribution of Total Bill')

plt.show()
```

```
count, ax = plt.subplots()

# we can use the viridis palette to help distinguish the colors
sns.countplot(data=tips, x='day', palette="viridis", ax=ax)

ax.set_title('Count of days')
ax.set_xlabel('Day of the Week')
ax.set_ylabel('Frequency')

plt.show()
```

```python
scatter, ax = plt.subplots()

# use fit_reg=False if you do not want the regression line
sns.scatterplot(data=tips, x='total_bill', y='tip', ax=ax)

ax.set_title('Scatter Plot of Total Bill and Tip')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Tip')

plt.show()
```

```python
reg, ax = plt.subplots()

# use fit_reg=False if you do not want the regression line
sns.regplot(data=tips, x='total_bill', y='tip', ax=ax)

ax.set_title('Regression Plot of Total Bill and Tip')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Tip')

plt.show()
```

```python
# use if you do not want the regression line
fig = sns.lmplot(data=tips, x='total_bill', y='tip')

plt.show()
```

```python
# jointplot creates the figure and axes for us
joint = sns.jointplot(data=tips, x='total_bill', y='tip')

joint.set_axis_labels(xlabel='Total Bill', ylabel='Tip')

# add a title and move the text up so it doesn't clash with histogram
joint.figure.suptitle('Joint Plot of Total Bill and Tip', y=1.03)

plt.show()
```

```python
# we can use jointplot with kind="hex" for a hexbin plot
hexbin = sns.jointplot(
  data=tips, x="total_bill", y="tip", kind="hex"
)

hexbin.set_axis_labels(xlabel='Total Bill', ylabel='Tip')
hexbin.figure.suptitle('Hexbin Plot of Total Bill and Tip', y=1.03)

plt.show()
```

```python
kde, ax = plt.subplots()

# shade will fill in the contours
sns.kdeplot(data=tips, x="total_bill", y="tip", shade=True, ax=ax)

ax.set_title('Kernel Density Plot of Total Bill and Tip')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Tip')

plt.show()
```

```python
kde2d = sns.jointplot(data=tips, x="total_bill", y="tip", kind="kde")

kde2d.set_axis_labels(xlabel='Total Bill', ylabel='Tip')
kde2d.fig.suptitle('2D KDE Plot of Total Bill and Tip', y=1.03)

plt.show()
```

```python
import numpy as np

bar, ax = plt.subplots()

# plot the average total bill for each value of time
# mean is calculated using numpy
sns.barplot(
  data=tips, x="time", y="total_bill", estimator=np.mean, ax=ax
)

ax.set_title('Bar Plot of Average Total Bill for Time of Day')
ax.set_xlabel('Time of Day')
ax.set_ylabel('Average Total Bill')

plt.show()
```

```python
box, ax = plt.subplots()

# the y is optional, but x would have to be a numeric variable
sns.boxplot(data=tips, x='time', y='total_bill', ax=ax)

ax.set_title('Box Plot of Total Bill by Time of Day')
ax.set_xlabel('Time of Day')
ax.set_ylabel('Total Bill')

plt.show()
```

```python
violin, ax = plt.subplots()

sns.violinplot(data=tips, x='time', y='total_bill', ax=ax)

ax.set_title('Violin plot of total bill by time of day')
ax.set_xlabel('Time of day')
ax.set_ylabel('Total Bill')

plt.show()
```

```
# create the figure with 2 subplots
box_violin, (ax1, ax2) = plt.subplots(nrows=1, ncols=2)

sns.boxplot(data=tips, x='time', y='total_bill', ax=ax1)
sns.violinplot(data=tips, x='time', y='total_bill', ax=ax2)

# set the titles
ax1.set_title('Box Plot')
ax1.set_xlabel('Time of day')
ax1.set_ylabel('Total Bill')

ax2.set_title('Violin Plot')
ax2.set_xlabel('Time of day')
ax2.set_ylabel('Total Bill')

box_violin.suptitle("Comparison of Box Plot with Violin Plot")

# space out the figure so labels do not overlap
box_violin.set_tight_layout(True)

plt.show()
```

```
fig = sns.pairplot(data=tips)

fig.figure.suptitle(
 'Pairwise Relationships of the Tips Data', y=1.03
)

plt.show()
```

```
# create a PairGrid, make the diagonal plots on a different scale
pair_grid = sns.PairGrid(tips, diag_sharey=False)

# set a separate function to plot the upper, bottom, and diagonal
# functions need to return an axes, not a figure
```

```
# we can use plt.scatter instead of sns.regplot
pair_grid = pair_grid.map_upper(sns.regplot)
pair_grid = pair_grid.map_lower(sns.kdeplot)
pair_grid = pair_grid.map_diag(sns.histplot)

plt.show()
```

```
violin, ax = plt.subplots()

sns.violinplot(
  data=tips,
  x="time",
  y="total_bill",
  hue="smoker", # set color based on smoker variable
  split=True,
  palette="viridis", # palette specifies the colors for hue
```

```
  ax=ax,
)

plt.show()
```

```
# note the use of lmplot instead of regplot to return a figure
scatter = sns.lmplot(
  data=tips,
  x="total_bill",
  y="tip",
  hue="smoker",
  fit_reg=False,
  palette="viridis",
)

plt.show()
```

```
fig = sns.pairplot(
 tips,
 hue="time",
 palette="viridis",
```

```
 height=2,  # facet height to make the entire figure smaller
)

plt.show()
```

```
fig, ax = plt.subplots()

sns.scatterplot(
  data=tips,
  x="total_bill",
  y="tip",
  hue="time",
  size="size",
  palette="viridis",
  ax=ax,
)

plt.show()
```

```
anscombe_plot = sns.relplot(
  data=anscombe,
  x="x",
  y="y",
  kind="scatter",
  col="data set",
```

```
  col_wrap=2,
  height=2,
  aspect=1.6,  # aspect ratio of each facet
)

anscombe_plot.figure.set_tight_layout(True)

plt.show()
```

```
'''python
colors = {
  "Yes": "#f1a340", # orange
  "No" : "#998ec3", # purple
}
# make the faceted scatter plot
# this is the only part that is needed to draw the figure
facet2 = sns.relplot(
  data=tips,
  x="total_bill",
  y="tip",
  hue="smoker",
  style="sex",
```

```
  kind="scatter",
  col="day",
  row="time",
  palette=colors,
  height=1.7,  # adjusted to fit figure on page
)

# below is to make the plot pretty
# adjust facet titles
facet2.set_titles(
  row_template="{row_name}",
  col_template="{col_name}"
)

# adjust the legend to not have it overlap the figure
sns.move_legend(
  facet2,
  loc="lower center",
  bbox_to_anchor=(0.5, 1),
  ncol=2,  #number legend columns
  title=None,  #legend title
  frameon=False,  #remove frame (i.e., border box) around legend
)

facet2.figure.set_tight_layout(True)

plt.show()'''
```

```
# create the FacetGrid
facet = sns.FacetGrid(tips, col='time')

# for each value in time, plot a histogram of total bill
# you pass in parameters as if you were passing them directly
# into sns.histplot()
facet.map(sns.histplot, 'total_bill')
plt.show()
```

```python
facet = sns.FacetGrid(
  tips, col='day', hue='sex', palette="viridis"
)
facet.map(plt.scatter, 'total_bill', 'tip')
facet.add_legend()
plt.show()
```

```python
facet = sns.FacetGrid(
  tips, col='time', row='smoker', hue='sex', palette="viridis"
)
facet.map(plt.scatter, 'total_bill', 'tip')
plt.show()
```

```python
# initial plot for comparison
fig, ax = plt.subplots()
sns.violinplot(
  data=tips, x="time", y="total_bill", hue="sex", split=True, ax=ax
)

plt.show()
```

```python
# Use this to set a global default style
# sns.set_style("whitegrid")
```

```python
# temporarily set style and plot
# remove the with line + indentation if using sns.set_style()
with sns.axes_style("darkgrid"):
```

```python
  fig, ax = plt.subplots()
  sns.violinplot(
    data=tips, x="time", y="total_bill", hue="sex", split=True, ax=ax
  )

plt.show()
```

```python
seaborn_styles = ["darkgrid", "whitegrid", "dark", "white", "ticks"]

fig = plt.figure()
for idx, style in enumerate(seaborn_styles):
  plot_position = idx + 1
  with sns.axes_style(style):
    ax = fig.add_subplot(2, 3, plot_position)
    violin = sns.violinplot(
      data=tips, x="time", y="total_bill", ax=ax
    )
    violin.set_title(style)
fig.set_tight_layout(True)
plt.show()
```

```python
contexts = pd.DataFrame(
  {
    "paper": sns.plotting_context("paper"),
    "notebook": sns.plotting_context("notebook"),
    "talk": sns.plotting_context("talk"),
    "poster": sns.plotting_context("poster"),

  }
)
print(contexts)
```

```
                   paper  notebook   talk  poster
axes.linewidth       1.0      1.25  1.875     2.5
grid.linewidth       0.8      1.00  1.500     2.0
lines.linewidth      1.2      1.50  2.250     3.0
lines.markersize     4.8      6.00  9.000    12.0
patch.linewidth      0.8      1.00  1.500     2.0
xtick.major.width    1.0      1.25  1.875     2.5
ytick.major.width    1.0      1.25  1.875     2.5
xtick.minor.width    0.8      1.00  1.500     2.0
ytick.minor.width    0.8      1.00  1.500     2.0
```

xtick.major.size	4.8	6.00	9.000	12.0
ytick.major.size	4.8	6.00	9.000	12.0
xtick.minor.size	3.2	4.00	6.000	8.0
ytick.minor.size	3.2	4.00	6.000	8.0
font.size	9.6	12.00	18.000	24.0
axes.labelsize	9.6	12.00	18.000	24.0
axes.titlesize	9.6	12.00	18.000	24.0
xtick.labelsize	8.8	11.00	16.500	22.0
ytick.labelsize	8.8	11.00	16.500	22.0
legend.fontsize	8.8	11.00	16.500	22.0
legend.title_fontsize	9.6	12.00	18.000	24.0

```
context_styles = contexts.columns

fig = plt.figure()
for idx, context in enumerate(context_styles):
  plot_position = idx + 1
  with sns.plotting_context(context):
    ax = fig.add_subplot(2, 2, plot_position)
    violin = sns.violinplot(
      data=tips, x="time", y="total_bill", ax=ax
    )
    violin.set_title(context)
fig.set_tight_layout(True)
plt.show()
```

```
box_violin, (ax1, ax2) = plt.subplots(nrows=1, ncols=2)

sns.boxplot(data=tips, x='time', y='total_bill', ax=ax1)
sns.violinplot(data=tips, x='time', y='total_bill', ax=ax2)

ax1.set_title('Box Plot')
ax1.set_xlabel('Time of day')
ax1.set_ylabel('Total Bill')

ax2.set_title('Violin Plot')
ax2.set_xlabel('Time of day')
ax2.set_ylabel('Total Bill')

box_violin.suptitle("Comparison of Box Plot with Violin Plot")

box_violin.set_tight_layout(True)
plt.show()
```

Returns the Axes object with the plot drawn onto it.

```
print(type(ax2))
```

```
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

```
print(type(box_violin))
```

```
<class 'matplotlib.figure.Figure'>
```

```
fig = sns.pairplot(data=tips)
fig.figure.suptitle(
 'Pairwise Relationships of the Tips Data', y=1.03
)
plt.show()
```

```
print(type(fig))
```

```
<class 'seaborn.axisgrid.PairGrid'>
```

```
# on a series
fig, ax = plt.subplots()
tips['total_bill'].plot.hist(ax=ax)
plt.show()
```

```
# on a dataframe
# set alpha channel transparency to see through the overlapping bars
fig, ax = plt.subplots()
tips[['total_bill', 'tip']].plot.hist(alpha=0.5, bins=20, ax=ax)
plt.show()
```

```
fig, ax = plt.subplots()
tips.plot.scatter(x='total_bill', y='tip', ax=ax)
plt.show()
```

```
fig, ax = plt.subplots()
tips.plot.hexbin(x='total_bill', y='tip', ax=ax)
plt.show()
```

```
fig, ax = plt.subplots()
tips.plot.hexbin(x='total_bill', y='tip', gridsize=10, ax=ax)
plt.show()
```

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

```python
# show only the first few columns
print(pew.iloc[:, 0:5])
```

```
                   religion  <$10k  $10-20k  $20-30k  $30-40k
0                  Agnostic     27       34       60       81
1                   Atheist     12       27       37       52
2                  Buddhist     27       21       30       34
3                  Catholic    418      617      732      670
4        Don't know/refused     15       14       15       11
..                      ...    ...      ...      ...      ...
13                 Orthodox     13       17       23       32
14          Other Christian      9        7       11       13
15             Other Faiths     20       33       40       46
16    Other World Religions      5        2        3        4
17             Unaffiliated    217      299      374      365

[18 rows x 5 columns]
```

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

```python
print(pew_long)
```

```
                religion            variable  value
0               Agnostic               <$10k     27
1                Atheist               <$10k     12
2               Buddhist               <$10k     27
3               Catholic               <$10k    418
4     Don't know/refused               <$10k     15
..                   ...                 ...    ...
175             Orthodox  Don't know/refused     73
176      Other Christian  Don't know/refused     18
177         Other Faiths  Don't know/refused     71
178  Other World Religions  Don't know/refused      8
179          Unaffiliated  Don't know/refused    597

[180 rows x 3 columns]
```

```python
# melt method
pew_long = pew.melt(id_vars='religion')

# melt function
pew_long = pd.melt(pew, id_vars='religion')
```

```python
pew_long = pew.melt(
  id_vars="religion", var_name="income", value_name="count"
)
```

```python
print(pew_long)
```

```
                    religion             income  count
0                   Agnostic              <$10k     27
1                    Atheist              <$10k     12
2                   Buddhist              <$10k     27
3                   Catholic              <$10k    418
4         Don't know/refused              <$10k     15
..                       ...                ...    ...
175                 Orthodox  Don't know/refused     73
176          Other Christian  Don't know/refused     18
177             Other Faiths  Don't know/refused     71
178    Other World Religions  Don't know/refused      8
179              Unaffiliated  Don't know/refused    597

[180 rows x 3 columns]
```

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

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

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

```
3  2000  3 Doors Down                     Loser  4:24   2000-10-21
4  2000      504 Boyz           Wobble Wobble  3:35   2000-04-15

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

```python
# use a list to reference more than 1 variable
billboard_long = billboard.melt(
  id_vars=["year", "artist", "track", "time", "date.entered"],
  var_name="week",
  value_name="rating",
)

print(billboard_long)
```

```
       year            artist                    track  time  \
0      2000             2 Pac  Baby Don't Cry (Keep...  4:22
1      2000           2Ge+her  The Hardest Part Of ...  3:15
2      2000      3 Doors Down               Kryptonite  3:53
3      2000      3 Doors Down                    Loser  4:24
4      2000          504 Boyz            Wobble Wobble  3:35
...     ...               ...                      ...   ...
24087  2000       Yankee Grey     Another Nine Minutes  3:10
24088  2000  Yearwood, Trisha          Real Live Woman  3:55
24089  2000   Ying Yang Twins  Whistle While You Tw...  4:19
24090  2000     Zombie Nation            Kernkraft 400  3:30
24091  2000   matchbox twenty                     Bent  4:12

      date.entered  week  rating
0       2000-02-26   wk1    87.0
1       2000-09-02   wk1    91.0
2       2000-04-08   wk1    81.0
3       2000-10-21   wk1    76.0
4       2000-04-15   wk1    57.0
...            ...   ...     ...
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

[24092 rows x 7 columns]
```

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

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

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

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

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

```
print(ebola_long)
```

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

```
1947    3/27/2014    5    Deaths_Mali      NaN
1948    3/26/2014    4    Deaths_Mali      NaN
1949    3/25/2014    3    Deaths_Mali      NaN
1950    3/24/2014    2    Deaths_Mali      NaN
1951    3/22/2014    0    Deaths_Mali      NaN

[1952 rows x 4 columns]
```

```
# 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
```

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

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

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

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

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

```python
print(status_values)
```

```
0        Cases
1        Cases
2        Cases
3        Cases
4        Cases
         ...
1947    Deaths
1948    Deaths
1949    Deaths
1950    Deaths
1951    Deaths
Name: variable, Length: 1952, dtype: object
```

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

```python
print(ebola_long)
```

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

```
...          ...  ...          ...    ...     ...     ...
1947   3/27/2014    5  Deaths_Mali    NaN  Deaths    Mali
1948   3/26/2014    4  Deaths_Mali    NaN  Deaths    Mali
1949   3/25/2014    3  Deaths_Mali    NaN  Deaths    Mali
1950   3/24/2014    2  Deaths_Mali    NaN  Deaths    Mali
1951   3/22/2014    0  Deaths_Mali    NaN  Deaths    Mali

[1952 rows x 6 columns]
```

```
# reset our ebola_long data
ebola_long = ebola.melt(id_vars=['Date', 'Day'])

# split the column by _ into a dataframe using expand
variable_split = ebola_long.variable.str.split('_', expand=True)

print(variable_split)
```

```
           0       1
0      Cases  Guinea
1      Cases  Guinea
2      Cases  Guinea
3      Cases  Guinea
4      Cases  Guinea
...      ...     ...
1947  Deaths    Mali
1948  Deaths    Mali
1949  Deaths    Mali
1950  Deaths    Mali
1951  Deaths    Mali

[1952 rows x 2 columns]
```

```python
ebola_long[['status', 'country']] = variable_split
```

```python
print(ebola_long)
```

```
            Date  Day      variable   value  status country
0       1/5/2015  289  Cases_Guinea  2776.0   Cases  Guinea
1       1/4/2015  288  Cases_Guinea  2775.0   Cases  Guinea
2       1/3/2015  287  Cases_Guinea  2769.0   Cases  Guinea
3       1/2/2015  286  Cases_Guinea     NaN   Cases  Guinea
4     12/31/2014  284  Cases_Guinea  2730.0   Cases  Guinea
...          ...  ...           ...     ...     ...     ...
1947   3/27/2014    5   Deaths_Mali     NaN  Deaths    Mali
1948   3/26/2014    4   Deaths_Mali     NaN  Deaths    Mali
1949   3/25/2014    3   Deaths_Mali     NaN  Deaths    Mali
1950   3/24/2014    2   Deaths_Mali     NaN  Deaths    Mali
1951   3/22/2014    0   Deaths_Mali     NaN  Deaths    Mali

[1952 rows x 6 columns]
```

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

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

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

```python
print(weather_melt)
```

```
          id  year  month element  day  temp
0    MX17004  2010      1    tmax   d1   NaN
1    MX17004  2010      1    tmin   d1   NaN
2    MX17004  2010      2    tmax   d1   NaN
3    MX17004  2010      2    tmin   d1   NaN
4    MX17004  2010      3    tmax   d1   NaN
..       ...   ...    ...     ...  ...   ...
677  MX17004  2010     10    tmin  d31   NaN
678  MX17004  2010     11    tmax  d31   NaN
679  MX17004  2010     11    tmin  d31   NaN
680  MX17004  2010     12    tmax  d31   NaN
681  MX17004  2010     12    tmin  d31   NaN

[682 rows x 6 columns]
```

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

```python
print(weather_tidy)
```

```
element                    tmax  tmin
id      year month day
MX17004 2010 1     d30     27.8  14.5
             2     d11     29.7  13.4
                   d2      27.3  14.4
                   d23     29.9  10.7
                   d3      24.1  14.4
...                         ...   ...
             11    d27     27.7  14.2
                   d26     28.1  12.1
                   d4      27.2  12.0
             12    d1      29.9  13.8
                   d6      27.8  10.5

[33 rows x 2 columns]
```

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

```
element       id  year  month  day  tmax  tmin
0        MX17004  2010      1  d30  27.8  14.5
1        MX17004  2010      2  d11  29.7  13.4
2        MX17004  2010      2   d2  27.3  14.4
3        MX17004  2010      2  d23  29.9  10.7
4        MX17004  2010      2   d3  24.1  14.4
..           ...   ...    ...  ...   ...   ...
28       MX17004  2010     11  d27  27.7  14.2
29       MX17004  2010     11  d26  28.1  12.1
30       MX17004  2010     11   d4  27.2  12.0
31       MX17004  2010     12   d1  29.9  13.8
32       MX17004  2010     12   d6  27.8  10.5

[33 rows x 6 columns]
```

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

print(weather_tidy)
```

```
element       id  year  month  day  tmax  tmin
0        MX17004  2010      1  d30  27.8  14.5
1        MX17004  2010      2  d11  29.7  13.4
2        MX17004  2010      2   d2  27.3  14.4
3        MX17004  2010      2  d23  29.9  10.7
4        MX17004  2010      2   d3  24.1  14.4
```

```
..          ...   ...    ...  ...   ...   ...
28      MX17004  2010     11  d27  27.7  14.2
29      MX17004  2010     11  d26  28.1  12.1
30      MX17004  2010     11   d4  27.2  12.0
31      MX17004  2010     12   d1  29.9  13.8
32      MX17004  2010     12   d6  27.8  10.5

[33 rows x 6 columns]
```

```python
def my_function(): # define a new function called my_function
  # indentation for
  # function code
  pass # this statement is here to make a valid empty function
```

```python
def my_sq(x):
  """Squares a given value
  """
  return x ** 2


def avg_2(x, y):
  """Calculates the average of 2 numbers
  """
  return (x + y) / 2
```

```python
import pandas as pd

df = pd.DataFrame({"a": [10, 20, 30], "b": [20, 30, 40]})
print(df)
```

	a	b
0	10	20
1	20	30
2	30	40

```python
# get the first column
print(type(df['a']))
```

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

```python
# get the first row
print(type(df.iloc[0]))
```

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

```python
# apply our square function on the 'a' column
sq = df['a'].apply(my_sq)
print(sq)
```

```
0    100
1    400
2    900
Name: a, dtype: int64
```

```python
# pass in the exponent, 3
cubed = my_exp(2, 3)
print(cubed)
```

```
8
```

```python
# if we don't pass in all the parameters
my_exp(2)
```

```
TypeError: my_exp() missing 1 required positional argument: 'e'
```

```python
# the exponent, e, to 2
ex = df['a'].apply(my_exp, e=2)
print(ex)
```

```
0    100
1    400
2    900
Name: a, dtype: int64
```

```python
# exponent, e, to 3
ex = df['a'].apply(my_exp, e=3)
print(ex)
```

```
0     1000
1     8000
2    27000
Name: a, dtype: int64
```

```python
df = pd.DataFrame({"a": [10, 20, 30], "b": [20, 30, 40]})
print(df)
```

```
    a   b
0  10  20
1  20  30
2  30  40
```

```python
# will cause an error
print(df.apply(avg_3))
```

```
TypeError: avg_3() missing 2 required positional arguments: 'y' and 'z'
```

```python
def avg_3_apply(col):
  """The avg_3 function but apply compatible
  by taking in all the values as the first argument
  and parsing out the values within the function
  """
  x = col[0]
  y = col[1]
  z = col[2]
  return (x + y + z) / 3
```

```python
print(df.apply(avg_3_apply))
```

```
a    20.0
b    30.0
dtype: float64
```

```python
# will cause an error
print(df.apply(avg_3_apply, axis=1))
```

```
IndexError: index 2 is out of bounds for axis 0 with size 2
```

```python
def avg_2_apply(row):
  """Taking the average of row value.
  Assuming that there are only 2 values in a row.
  """
  x = row[0]
  y = row[1]
  return (x + y) / 2

print(df.apply(avg_2_apply, axis=0))
```

```
a    15.0
b    25.0
dtype: float64
```

```python
df = pd.DataFrame({"a": [10, 20, 30], "b": [20, 30, 40]})
print(df)
```

```
    a   b
0  10  20
1  20  30
2  30  40
```

```python
print(avg_2(df['a'], df['b']))
```

```
0    15.0
1    25.0
2    35.0
dtype: float64
```

```python
import numpy as np

def avg_2_mod(x, y):
  """Calculate the average, unless x is 20
  If the value is 20, return a missing value
  """
  if (x == 20):
    return(np.NaN)
  else:
    return (x + y) / 2
```

```python
# will cause an error
print(avg_2_mod(df['a'], df['b']))
```

```
ValueError: The truth value of a Series is ambiguous. Use a.empty,
a.bool(), a.item(), a.any() or a.all().
```

```python
import numpy as np

# np.vectorize actually creates a new function
avg_2_mod_vec = np.vectorize(avg_2_mod)

# use the newly vectorized function
print(avg_2_mod_vec(df['a'], df['b']))
```

```
[15. nan 35.]
```

```python
# to use the vectorize decorator
# we use the @ symbol before our function definition
@np.vectorize
def v_avg_2_mod(x, y):
  """Calculate the average, unless x is 20
  Same as before, but we are using the vectorize decorator
  """
  if (x == 20):
    return(np.NaN)
  else:
    return (x + y) / 2

# we can then directly use the vectorized function
# without having to create a new function
print(v_avg_2_mod(df['a'], df['b']))
```

```
[15. nan 35.]
```

```python
import numba

@numba.vectorize
def v_avg_2_numba(x, y):
  """Calculate the average, unless x is 20
  Using the numba decorator.
  """
  # we now have to add type information to our function
  if (int(x) == 20):
    return(np.NaN)
  else:
    return (x + y) / 2
```

```python
print(v_avg_2_numba(df['a'], df['b']))
```

```
ValueError: Cannot determine Numba type of
<class 'pandas.core.series.Series'>
```

```python
# passing in the numpy array
print(v_avg_2_numba(df['a'].values, df['b'].values))
```

```
[15. nan 35.]
```

```python
df = pd.DataFrame({'a': [10, 20, 30],
                   'b': [20, 30, 40]})
print(df)
```

```
    a   b
0  10  20
1  20  30
2  30  40
```

```python
def my_sq(x):
  return x ** 2

df['a_sq'] = df['a'].apply(my_sq)
print(df)
```

```
    a   b  a_sq
0  10  20   100
1  20  30   400
2  30  40   900
```

```python
df['a_sq_lamb'] = df['a'].apply(lambda x: x ** 2)
print(df)
```

```
    a   b  a_sq  a_sq_lamb
0  10  20   100        100
1  20  30   400        400
2  30  40   900        900
```

```python
import pandas as pd

df1 = pd.read_csv('data/concat_1.csv')
df2 = pd.read_csv('data/concat_2.csv')
df3 = pd.read_csv('data/concat_3.csv')
```

```python
print(df1)
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

```python
print(df2)
```

```
    A   B   C   D
0  a4  b4  c4  d4
1  a5  b5  c5  d5
2  a6  b6  c6  d6
3  a7  b7  c7  d7
```

```python
print(df3)
```

```
     A    B    C    D
0   a8   b8   c8   d8
1   a9   b9   c9   d9
2  a10  b10  c10  d10
3  a11  b11  c11  d11
```

```python
print(df1.index)
```

```
RangeIndex(start=0, stop=4, step=1)
```

```python
print(df1.columns)
```

```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

```python
row_concat = pd.concat([df1, df2, df3])
print(row_concat)
```

```
      A    B    C    D
0    a0   b0   c0   d0
1    a1   b1   c1   d1
2    a2   b2   c2   d2
3    a3   b3   c3   d3
0    a4   b4   c4   d4
..  ...  ...  ...  ...
3    a7   b7   c7   d7
0    a8   b8   c8   d8
1    a9   b9   c9   d9
2   a10  b10  c10  d10
3   a11  b11  c11  d11

[12 rows x 4 columns]
```

```python
# subset the fourth row of the concatenated dataframe
print(row_concat.iloc[3, :])
```

```
A    a3
B    b3
C    c3
D    d3
Name: 3, dtype: object
```

```
# create a new row of data
new_row_series = pd.Series(['n1', 'n2', 'n3', 'n4'])
print(new_row_series)
```

```
0    n1
1    n2
2    n3
3    n4
dtype: object
```

```
# attempt to add the new row to a dataframe
print(pd.concat([df1, new_row_series]))
```

```
     A    B    C    D    0
0   a0   b0   c0   d0  NaN
1   a1   b1   c1   d1  NaN
2   a2   b2   c2   d2  NaN
3   a3   b3   c3   d3  NaN
0  NaN  NaN  NaN  NaN   n1
1  NaN  NaN  NaN  NaN   n2
2  NaN  NaN  NaN  NaN   n3
3  NaN  NaN  NaN  NaN   n4
```

```
new_row_df = pd.DataFrame(
  # note the double brackets to create a "row" of data
  data=[["n1", "n2", "n3", "n4"]],
  columns=["A", "B", "C", "D"],
)

print(new_row_df)
```

```
    A   B   C   D
0  n1  n2  n3  n4
```

```
# concatenate the row of data
print(pd.concat([df1, new_row_df]))
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
0  n1  n2  n3  n4
```

```
row_concat_i = pd.concat([df1, df2, df3], ignore_index=True)
print(row_concat_i)
```

```
     A    B    C    D
0    a0   b0   c0   d0
1    a1   b1   c1   d1
2    a2   b2   c2   d2
3    a3   b3   c3   d3
4    a4   b4   c4   d4
..  ...  ...  ...  ...
7    a7   b7   c7   d7
8    a8   b8   c8   d8
```

```
9    a9   b9   c9   d9
10  a10  b10  c10  d10
11  a11  b11  c11  d11

[12 rows x 4 columns]
```

```
col_concat = pd.concat([df1, df2, df3], axis="columns")
print(col_concat)
```

```
    A   B   C   D   A   B   C   D    A    B    C    D
0  a0  b0  c0  d0  a4  b4  c4  d4   a8   b8   c8   d8
1  a1  b1  c1  d1  a5  b5  c5  d5   a9   b9   c9   d9
2  a2  b2  c2  d2  a6  b6  c6  d6  a10  b10  c10  d10
3  a3  b3  c3  d3  a7  b7  c7  d7  a11  b11  c11  d11
```

```
col_concat['new_col_list'] = ['n1', 'n2', 'n3', 'n4']
print(col_concat)
```

```
    A   B   C   D   A   B   C   D    A    B    C    D new_col_list
0  a0  b0  c0  d0  a4  b4  c4  d4   a8   b8   c8   d8           n1
1  a1  b1  c1  d1  a5  b5  c5  d5   a9   b9   c9   d9           n2
2  a2  b2  c2  d2  a6  b6  c6  d6  a10  b10  c10  d10           n3
3  a3  b3  c3  d3  a7  b7  c7  d7  a11  b11  c11  d11           n4
```

```
col_concat['new_col_series'] = pd.Series(['n1', 'n2', 'n3', 'n4'])
print(col_concat)
```

```
    A   B   C   D   A   B   C   D    A    B    C    D new_col_list  \
0  a0  b0  c0  d0  a4  b4  c4  d4   a8   b8   c8   d8           n1   
1  a1  b1  c1  d1  a5  b5  c5  d5   a9   b9   c9   d9           n2   
2  a2  b2  c2  d2  a6  b6  c6  d6  a10  b10  c10  d10           n3   
3  a3  b3  c3  d3  a7  b7  c7  d7  a11  b11  c11  d11           n4   

  new_col_series  
0             n1  
1             n2  
2             n3  
3             n4  
```

```
print(pd.concat([df1, df2, df3], axis="columns", ignore_index=True))
```

```
   0   1   2   3   4   5   6   7    8    9    10   11
0  a0  b0  c0  d0  a4  b4  c4  d4   a8   b8   c8   d8
1  a1  b1  c1  d1  a5  b5  c5  d5   a9   b9   c9   d9
2  a2  b2  c2  d2  a6  b6  c6  d6  a10  b10  c10  d10
3  a3  b3  c3  d3  a7  b7  c7  d7  a11  b11  c11  d11
```

```python
# rename the columns of our dataframes
df1.columns = ['A', 'B', 'C', 'D']
df2.columns = ['E', 'F', 'G', 'H']
df3.columns = ['A', 'C', 'F', 'H']

print(df1)
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

```python
print(df2)
```

```
    E   F   G   H
0  a4  b4  c4  d4
1  a5  b5  c5  d5
2  a6  b6  c6  d6
3  a7  b7  c7  d7
```

```python
print(df3)
```

```
     A    C    F    H
0   a8   b8   c8   d8
1   a9   b9   c9   d9
2  a10  b10  c10  d10
3  a11  b11  c11  d11
```

```python
row_concat = pd.concat([df1, df2, df3])
print(row_concat)
```

```
      A    B    C    D    E    F    G    H
0    a0   b0   c0   d0  NaN  NaN  NaN  NaN
1    a1   b1   c1   d1  NaN  NaN  NaN  NaN
2    a2   b2   c2   d2  NaN  NaN  NaN  NaN
3    a3   b3   c3   d3  NaN  NaN  NaN  NaN
0   NaN  NaN  NaN  NaN   a4   b4   c4   d4
..  ...  ...  ...  ...  ...  ...  ...  ...
3   NaN  NaN  NaN  NaN   a7   b7   c7   d7
0    a8  NaN   b8  NaN  NaN   c8  NaN   d8
1    a9  NaN   b9  NaN  NaN   c9  NaN   d9
2   a10  NaN  b10  NaN  NaN  c10  NaN  d10
3   a11  NaN  b11  NaN  NaN  c11  NaN  d11

[12 rows x 8 columns]
```

```python
print(pd.concat([df1, df2, df3], join='inner'))
```

```
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]

[12 rows x 0 columns]
```

```
print(pd.concat([df1,df3], ignore_index=False, join='inner'))
```

```
     A    C
0   a0   c0
1   a1   c1
2   a2   c2
3   a3   c3
0   a8   b8
1   a9   b9
2  a10  b10
3  a11  b11
```

```
col_concat = pd.concat([df1, df2, df3], axis="columns")
print(col_concat)
```

```
     A    B    C    D    E    F    G    H    A    C    F    H
0   a0   b0   c0   d0  NaN  NaN  NaN  NaN   a8   b8   c8   d8
1   a1   b1   c1   d1  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
2   a2   b2   c2   d2  NaN  NaN  NaN  NaN   a9   b9   c9   d9
3   a3   b3   c3   d3  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
4  NaN  NaN  NaN  NaN   a4   b4   c4   d4  NaN  NaN  NaN  NaN
5  NaN  NaN  NaN  NaN   a5   b5   c5   d5  a10  b10  c10  d10
6  NaN  NaN  NaN  NaN   a6   b6   c6   d6  NaN  NaN  NaN  NaN
7  NaN  NaN  NaN  NaN   a7   b7   c7   d7  a11  b11  c11  d11
```

```
print(pd.concat([df1, df3], axis="columns", join='inner'))
```

```
    A   B   C   D   A   C   F   H
0  a0  b0  c0  d0  a8  b8  c8  d8
2  a2  b2  c2  d2  a9  b9  c9  d9
```

```
data/billboard-by_week/billboard-XX.csv
```

```python
from pathlib import Path

# from my current directory fine (glob) the this pattern
```

```python
billboard_data_files = (
    Path(".")
    .glob("data/billboard-by_week/billboard-*.csv")
)

# this line is optional if you want to see the full list of files
billboard_data_files = sorted(list(billboard_data_files))

print(billboard_data_files)
```

```
[PosixPath('data/billboard-by_week/billboard-01.csv'),
PosixPath('data/billboard-by_week/billboard-02.csv'),
PosixPath('data/billboard-by_week/billboard-03.csv'),
PosixPath('data/billboard-by_week/billboard-04.csv'),
PosixPath('data/billboard-by_week/billboard-05.csv'),
..    ...                  ...                 ...    ...
PosixPath('data/billboard-by_week/billboard-72.csv'),
PosixPath('data/billboard-by_week/billboard-73.csv'),
PosixPath('data/billboard-by_week/billboard-74.csv'),
PosixPath('data/billboard-by_week/billboard-75.csv'),
PosixPath('data/billboard-by_week/billboard-76.csv')]
```

```python
billboard_data_files = list(billboard_data_files)
```

```python
billboard01 = pd.read_csv(billboard_data_files[0])
billboard02 = pd.read_csv(billboard_data_files[1])
billboard03 = pd.read_csv(billboard_data_files[2])
```

```python
# just look at one of the data sets we loaded
print(billboard01)
```

```
     year             artist                    track  time  \
0    2000              2 Pac  Baby Don't Cry (Keep...  4:22
1    2000            2Ge+her  The Hardest Part Of ...  3:15
2    2000       3 Doors Down               Kryptonite  3:53
3    2000       3 Doors Down                    Loser  4:24
4    2000           504 Boyz            Wobble Wobble  3:35
..    ...                ...                      ...   ...
312  2000        Yankee Grey     Another Nine Minutes  3:10
313  2000  Yearwood, Trisha          Real Live Woman  3:55
314  2000    Ying Yang Twins  Whistle While You Tw...  4:19
315  2000      Zombie Nation            Kernkraft 400  3:30
316  2000    matchbox twenty                     Bent  4:12

    date.entered week  rating
0     2000-02-26  wk1    87.0
1     2000-09-02  wk1    91.0
2     2000-04-08  wk1    81.0
3     2000-10-21  wk1    76.0
4     2000-04-15  wk1    57.0
..           ...  ...     ...
312   2000-04-29  wk1    86.0
313   2000-04-01  wk1    85.0
314   2000-03-18  wk1    95.0
315   2000-09-02  wk1    99.0
316   2000-04-29  wk1    60.0

[317 rows x 7 columns]
```

```
# shape of each dataframe
print(billboard01.shape)
print(billboard02.shape)
print(billboard03.shape)
```

```
(317, 7)
(317, 7)
(317, 7)
```

```
# concatenate the dataframes together
billboard = pd.concat([billboard01,  billboard02, billboard03])

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

```
(951, 7)
```

```python
# this part was the same as earlier
from pathlib import Path
billboard_data_files = (
    Path(".")
    .glob("data/billboard-by_week/billboard-*.csv")
)

# create an empty list to append to
list_billboard_df = []

# loop though each CSV filename
for csv_filename in billboard_data_files:
    # 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_billboard_df.append(df)

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

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

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

```python
# look at the first dataframe
print(list_billboard_df[0])
```

```
     year            artist                    track  time  \
0    2000             2 Pac  Baby Don't Cry (Keep...  4:22
1    2000           2Ge+her  The Hardest Part Of ...  3:15
2    2000      3 Doors Down               Kryptonite  3:53
3    2000      3 Doors Down                    Loser  4:24
4    2000          504 Boyz            Wobble Wobble  3:35
..    ...               ...                      ...   ...
312  2000       Yankee Grey     Another Nine Minutes  3:10
313  2000  Yearwood, Trisha          Real Live Woman  3:55
314  2000   Ying Yang Twins  Whistle While You Tw...  4:19
315  2000     Zombie Nation            Kernkraft 400  3:30
316  2000   matchbox twenty                     Bent  4:12

    date.entered  week  rating
0     2000-02-26  wk15     NaN
1     2000-09-02  wk15     NaN
2     2000-04-08  wk15    38.0
3     2000-10-21  wk15    72.0
4     2000-04-15  wk15    78.0
..           ...   ...     ...
312   2000-04-29  wk15     NaN
313   2000-04-01  wk15     NaN
314   2000-03-18  wk15     NaN
315   2000-09-02  wk15     NaN
316   2000-04-29  wk15     3.0

[317 rows x 7 columns]
```

```python
billboard_loop_concat = pd.concat(list_billboard_df)
print(billboard_loop_concat.shape)
```

```
(24092, 7)
```

```python
# we have to re-create the generator because we
# "used it up" in the previous example
billboard_data_files = (
```

```python
    Path(".")
    .glob("data/billboard-by_week/billboard-*.csv")
)
# the loop code without comments
list_billboard_df = []
for csv_filename in billboard_data_files:
    df = pd.read_csv(csv_filename)
    list_billboard_df.append(df)

billboard_data_files = (
    Path(".")
    .glob("data/billboard-by_week/billboard-*.csv")
)

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

```python
billboard_concat_comp = pd.concat(billboard_dfs)
```

```python
print(billboard_concat_comp)
```

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

```
..   ...                ...                         ...   ...
312  2000       Yankee Grey     Another Nine Minutes  3:10
313  2000   Yearwood, Trisha         Real Live Woman  3:55
314  2000    Ying Yang Twins  Whistle While You Tw...  4:19
315  2000      Zombie Nation           Kernkraft 400  3:30
316  2000    matchbox twenty                    Bent  4:12

    date.entered  week  rating
0     2000-02-26  wk15     NaN
1     2000-09-02  wk15     NaN
2     2000-04-08  wk15    38.0
3     2000-10-21  wk15    72.0
4     2000-04-15  wk15    78.0
..           ...   ...     ...
312   2000-04-29  wk18     NaN
313   2000-04-01  wk18     NaN
314   2000-03-18  wk18     NaN
315   2000-09-02  wk18     NaN
316   2000-04-29  wk18     3.0

[24092 rows x 7 columns]
```

```
person = pd.read_csv('data/survey_person.csv')
site = pd.read_csv('data/survey_site.csv')
survey = pd.read_csv('data/survey_survey.csv')
visited = pd.read_csv('data/survey_visited.csv')
```

```
print(person)
```

```
      ident   personal    family
0      dyer    William      Dyer
1        pb      Frank   Pabodie
2      lake   Anderson      Lake
3       roe  Valentina   Roerich
4  danforth      Frank  Danforth
```

```
print(site)
```

```
    name    lat    long
0   DR-1 -49.85 -128.57
1   DR-3 -47.15 -126.72
2  MSK-4 -48.87 -123.40
```

```
print(visited)
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3         NaN
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

```
print(survey)
```

```
    taken person quant  reading
0     619   dyer   rad     9.82
1     619   dyer   sal     0.13
2     622   dyer   rad     7.80
3     622   dyer   sal     0.09
4     734     pb   rad     8.41
..    ...    ...   ...      ...
16    752    roe   sal    41.60
17    837   lake   rad     1.46
18    837   lake   sal     0.21
19    837    roe   sal    22.50
20    844    roe   rad    11.25

[21 rows x 4 columns]
```

```python
visited_subset = visited.loc[[0, 2, 6], :]
print(visited_subset)
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
2    734   DR-3  1939-01-07
6    837  MSK-4  1932-01-14
```

```python
# get a count of the values in the site column
print(
  visited_subset["site"].value_counts()
)
```

```
DR-1     1
DR-3     1
MSK-4    1
Name: site, dtype: int64
```

```python
# the default value for 'how' is 'inner'
# so it doesn't need to be specified
o2o_merge = site.merge(
    visited_subset, left_on="name", right_on="site"
)
print(o2o_merge)
```

```
    name    lat    long  ident   site       dated
0   DR-1 -49.85 -128.57    619   DR-1  1927-02-08
1   DR-3 -47.15 -126.72    734   DR-3  1939-01-07
2  MSK-4 -48.87 -123.40    837  MSK-4  1932-01-14
```

```python
# get a count of the values in the site column
print(
  visited["site"].value_counts()
)
```

```
DR-3     4
DR-1     3
MSK-4    1
Name: site, dtype: int64
```

```python
m2o_merge = site.merge(visited, left_on='name', right_on='site')
print(m2o_merge)
```

```
    name    lat    long  ident   site       dated
0   DR-1 -49.85 -128.57    619   DR-1  1927-02-08
1   DR-1 -49.85 -128.57    622   DR-1  1927-02-10
2   DR-1 -49.85 -128.57    844   DR-1  1932-03-22
3   DR-3 -47.15 -126.72    734   DR-3  1939-01-07
4   DR-3 -47.15 -126.72    735   DR-3  1930-01-12
5   DR-3 -47.15 -126.72    751   DR-3  1930-02-26
6   DR-3 -47.15 -126.72    752   DR-3         NaN
7  MSK-4 -48.87 -123.40    837  MSK-4  1932-01-14
```

```python
ps = person.merge(survey, left_on='ident', right_on='person')
vs = visited.merge(survey, left_on='ident', right_on='taken')
```

```python
print(ps)
```

```
   ident   personal   family  taken person quant  reading
0   dyer    William     Dyer    619   dyer   rad     9.82
1   dyer    William     Dyer    619   dyer   sal     0.13
2   dyer    William     Dyer    622   dyer   rad     7.80
3   dyer    William     Dyer    622   dyer   sal     0.09
4     pb      Frank  Pabodie    734     pb   rad     8.41
..   ...        ...      ...    ...    ...   ...      ...
14  lake   Anderson     Lake    837   lake   rad     1.46
15  lake   Anderson     Lake    837   lake   sal     0.21
16   roe  Valentina  Roerich    752    roe   sal    41.60
17   roe  Valentina  Roerich    837    roe   sal    22.50
18   roe  Valentina  Roerich    844    roe   rad    11.25

[19 rows x 7 columns]
```

```python
print(vs)
```

```
    ident   site       dated  taken person quant  reading
0     619   DR-1  1927-02-08    619   dyer   rad     9.82
1     619   DR-1  1927-02-08    619   dyer   sal     0.13
2     622   DR-1  1927-02-10    622   dyer   rad     7.80
3     622   DR-1  1927-02-10    622   dyer   sal     0.09
4     734   DR-3  1939-01-07    734     pb   rad     8.41
..    ...    ...         ...    ...    ...   ...      ...
16    752   DR-3         NaN    752    roe   sal    41.60
17    837  MSK-4  1932-01-14    837   lake   rad     1.46
18    837  MSK-4  1932-01-14    837   lake   sal     0.21
19    837  MSK-4  1932-01-14    837    roe   sal    22.50
20    844   DR-1  1932-03-22    844    roe   rad    11.25

[21 rows x 7 columns]
```

```python
print(
  ps["quant"].value_counts()
)
```

```
rad     8
sal     8
temp    3
Name: quant, dtype: int64
```

```python
print(
  vs["quant"].value_counts()
)
```

```
sal     9
rad     8
temp    4
Name: quant, dtype: int64
```

```python
print(ps_vs.loc[0, :])
```

```
ident_x            dyer
personal        William
family             Dyer
taken_x             619
person_x           dyer
                   ...
site               DR-1
dated        1927-02-08
taken_y             619
person_y           dyer
reading_y          9.82
Name: 0, Length: 13, dtype: object
```

```
print(ps.shape) # left dataframe
```

```
(19, 7)
```

```
print(vs.shape) # right dataframe
```

```
(21, 7)
```

```
print(ps_vs.shape) # after merge
```

```
(148, 13)
```

```
assert ps_vs.shape[0] <= vs.shape[0]
```

```python
import pandas as pd

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

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

print(billboard_long)
```

```
       year           artist                    track  time  \
0      2000            2 Pac  Baby Don't Cry (Keep...  4:22
1      2000          2Ge+her  The Hardest Part Of ...  3:15
2      2000     3 Doors Down                Kryptonite  3:53
3      2000     3 Doors Down                     Loser  4:24
4      2000         504 Boyz             Wobble Wobble  3:35
...     ...              ...                       ...   ...
24087  2000      Yankee Grey     Another Nine Minutes  3:10
24088  2000  Yearwood, Trisha          Real Live Woman  3:55
24089  2000   Ying Yang Twins  Whistle While You Tw...  4:19
24090  2000     Zombie Nation             Kernkraft 400  3:30
24091  2000   matchbox twenty                      Bent  4:12

      date.entered  week  rating
0       2000-02-26   wk1    87.0
1       2000-09-02   wk1    91.0
2       2000-04-08   wk1    81.0
3       2000-10-21   wk1    76.0
4       2000-04-15   wk1    57.0
...            ...   ...     ...
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

[24092 rows x 7 columns]
```

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

```
       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
...     ...           ...    ...   ...          ...   ...     ...
22510  2000  3 Doors Down  Loser  4:24   2000-10-21  wk72     NaN
22827  2000  3 Doors Down  Loser  4:24   2000-10-21  wk73     NaN
23144  2000  3 Doors Down  Loser  4:24   2000-10-21  wk74     NaN
23461  2000  3 Doors Down  Loser  4:24   2000-10-21  wk75     NaN
23778  2000  3 Doors Down  Loser  4:24   2000-10-21  wk76     NaN

[76 rows x 7 columns]
```

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

```
(24092, 4)
```

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

```
(317, 4)
```

```python
billboard_songs['id'] = billboard_songs.index + 1
print(billboard_songs)
```

```
     year             artist                    track  time   id
0    2000              2 Pac  Baby Don't Cry (Keep...  4:22    1
1    2000            2Ge+her  The Hardest Part Of ...  3:15    2
2    2000       3 Doors Down                Kryptonite  3:53    3
3    2000       3 Doors Down                     Loser  4:24    4
4    2000           504 Boyz             Wobble Wobble  3:35    5
..    ...                ...                       ...   ...  ...
312  2000         Yankee Grey     Another Nine Minutes  3:10  313
313  2000   Yearwood, Trisha          Real Live Woman  3:55  314
314  2000    Ying Yang Twins  Whistle While You Tw...  4:19  315
315  2000      Zombie Nation            Kernkraft 400  3:30  316
316  2000    matchbox twenty                     Bent  4:12  317

[317 rows x 5 columns]
```

```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)
```

```
(24092, 8)
```

```python
print(billboard_ratings)
```

```
       year           artist                    track  time  \
0      2000            2 Pac  Baby Don't Cry (Keep...  4:22
1      2000            2 Pac  Baby Don't Cry (Keep...  4:22
2      2000            2 Pac  Baby Don't Cry (Keep...  4:22
3      2000            2 Pac  Baby Don't Cry (Keep...  4:22
4      2000            2 Pac  Baby Don't Cry (Keep...  4:22
...     ...              ...                      ...   ...
24087  2000  matchbox twenty                     Bent  4:12
```

```
24088  2000  matchbox twenty                          Bent  4:12
24089  2000  matchbox twenty                          Bent  4:12
24090  2000  matchbox twenty                          Bent  4:12
24091  2000  matchbox twenty                          Bent  4:12

      date.entered  week  rating   id
0       2000-02-26   wk1    87.0    1
1       2000-02-26   wk2    82.0    1
2       2000-02-26   wk3    72.0    1
3       2000-02-26   wk4    77.0    1
4       2000-02-26   wk5    87.0    1
...            ...   ...     ...  ...
24087   2000-04-29  wk72     NaN  317
24088   2000-04-29  wk73     NaN  317
24089   2000-04-29  wk74     NaN  317
24090   2000-04-29  wk75     NaN  317
24091   2000-04-29  wk76     NaN  317

[24092 rows x 8 columns]
```

```
billboard_ratings = billboard_ratings[
    ["id", "date.entered", "week", "rating"]
]
print(billboard_ratings)
```

```
        id date.entered  week  rating
0        1   2000-02-26   wk1    87.0
1        1   2000-02-26   wk2    82.0
```

```
2        1   2000-02-26   wk3    72.0
3        1   2000-02-26   wk4    77.0
4        1   2000-02-26   wk5    87.0
...    ...          ...   ...     ...
24087  317   2000-04-29  wk72     NaN
24088  317   2000-04-29  wk73     NaN
24089  317   2000-04-29  wk74     NaN
24090  317   2000-04-29  wk75     NaN
24091  317   2000-04-29  wk76     NaN

[24092 rows x 4 columns]
```

```python
import pandas as pd
df = pd.read_csv('data/gapminder.tsv', sep='\t')

# calculate the average life expectancy for each year
avg_life_exp_by_year = df.groupby('year')["lifeExp"].mean()

print(avg_life_exp_by_year)
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
          ...
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, Length: 12, dtype: float64
```

```python
# get a list of unique years in the data
years = df.year.unique()
print(years)
```

```
[1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007]
```

```python
# subset the data for the year 1952
y1952 = df.loc[df.year == 1952, :]
print(y1952)
```

```
                 country continent  year  lifeExp       pop  \
0            Afghanistan      Asia  1952   28.801   8425333
12               Albania    Europe  1952   55.230   1282697
24               Algeria    Africa  1952   43.077   9279525
36                Angola    Africa  1952   30.015   4232095
48             Argentina  Americas  1952   62.485  17876956
...                  ...       ...   ...      ...       ...
1644             Vietnam      Asia  1952   40.412  26246839
1656  West Bank and Gaza      Asia  1952   43.160   1030585
1668         Yemen, Rep.      Asia  1952   32.548   4963829
1680              Zambia    Africa  1952   42.038   2672000
1692            Zimbabwe    Africa  1952   48.451   3080907

        gdpPercap
0      779.445314
12    1601.056136
24    2449.008185
36    3520.610273
48    5911.315053
...           ...
1644   605.066492
1656  1515.592329
1668   781.717576
1680  1147.388831
1692   406.884115

[142 rows x 6 columns]
```

```python
y1952_mean = y1952["lifeExp"].mean()
print(y1952_mean)
```

```
49.057619718309866
```

```python
# group by continent and describe each group
continent_describe = df.groupby('continent')["lifeExp"].describe()
print(continent_describe)
```

```
           count       mean        std     min       25%      50%  \
continent
Africa     624.0  48.865330   9.150210  23.599  42.37250  47.7920
Americas   300.0  64.658737   9.345088  37.579  58.41000  67.0480
Asia       396.0  60.064903  11.864532  28.801  51.42625  61.7915
Europe     360.0  71.903686   5.433178  43.585  69.57000  72.2410
Oceania     24.0  74.326208   3.795611  69.120  71.20500  73.6650

                75%     max
continent
Africa     54.41150  76.442
Americas   71.69950  80.653
Asia       69.50525  82.603
Europe     75.45050  81.757
Oceania    77.55250  81.235
```

```python
import numpy as np

# calculate the average life expectancy by continent
# but use the np.mean function
cont_le_agg = df.groupby('continent')["lifeExp"].agg(np.mean)

print(cont_le_agg)
```

```
continent
Africa      48.865330
Americas    64.658737
Asia        60.064903
Europe      71.903686
Oceania     74.326208
Name: lifeExp, dtype: float64
```

```python
def my_mean(values):
  """My version of calculating a mean"""
  # get the total number of numbers for the denominator
  n = len(values)

  # start the sum at 0
  sum = 0
  for value in values:
      # add each value to the running sum
      sum += value

  # return the summed values divided by the number of values
  return sum / n
```

```python
# use our custom function into agg
agg_my_mean = df.groupby('year')["lifeExp"].agg(my_mean)

print(agg_my_mean)
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
          ...
1987    63.212613
1992    64.160338
1997    65.014676

2002    65.694923
2007    67.007423
Name: lifeExp, Length: 12, dtype: float64
```

```python
def my_mean_diff(values, diff_value):
    """Difference between the mean and diff_value
    """
    n = len(values)
```

```
    n = len(values)
    sum = 0
    for value in values:
        sum += value
    mean = sum / n
    return(mean - diff_value)

# calculate the global average life expectancy mean
global_mean = df["lifeExp"].mean()
print(global_mean)
```

```
59.474439366197174
```

```
# custom aggregation function with multiple parameters
agg_mean_diff = (
  df
  .groupby("year")
  ["lifeExp"]
  .agg(my_mean_diff, diff_value=global_mean)
)

print(agg_mean_diff)
```

```
year
1952   -10.416820
1957    -7.967038
1962    -5.865190
1967    -3.796150
1972    -1.827053
          ...
1987     3.738173
1992     4.685899
1997     5.540237
2002     6.220483
2007     7.532983
Name: lifeExp, Length: 12, dtype: float64
```

```python
# calculate the count, mean, std of the lifeExp by continent
gdf = (
  df
  .groupby("year")
  ["lifeExp"]
  .agg([np.count_nonzero, np.mean, np.std])
)

print(gdf)
```

```
      count_nonzero       mean        std
year
1952            142  49.057620  12.225956
1957            142  51.507401  12.231286
1962            142  53.609249  12.097245
1967            142  55.678290  11.718858
1972            142  57.647386  11.381953
...             ...        ...        ...
1987            142  63.212613  10.556285
1992            142  64.160338  11.227380
1997            142  65.014676  11.559439
2002            142  65.694923  12.279823
2007            142  67.007423  12.073021

[12 rows x 3 columns]
```

```python
# use a dictionary on a dataframe to agg different columns
# for each year, calculate the
# average lifeExp, median pop, and median gdpPercap
gdf_dict = df.groupby("year").agg(
```

```
  {
    "lifeExp": "mean",
    "pop": "median",
    "gdpPercap": "median"
  }
)

print(gdf_dict)
```

```
        lifeExp         pop    gdpPercap
year
1952  49.057620   3943953.0  1968.528344
1957  51.507401   4282942.0  2173.220291
1962  53.609249   4686039.5  2335.439533
1967  55.678290   5170175.5  2678.334740
1972  57.647386   5877996.5  3339.129407
...         ...         ...          ...
1987  63.212613   7774861.5  4280.300366
1992  64.160338   8688686.5  4386.085502
1997  65.014676   9735063.5  4781.825478
2002  65.694923  10372918.5  5319.804524
2007  67.007423  10517531.0  6124.371108

[12 rows x 3 columns]
```

```
gdf = (
  df
  .groupby("year")
  ["lifeExp"]
  .agg(
    [
      np.count_nonzero,
      np.mean,
      np.std,
    ]
  )
  .rename(
    columns={
      "count_nonzero": "count",
      "mean": "avg",
      "std": "std_dev",
```

```python
    }
  )
  .reset_index() # return a flat dataframe
)

print(gdf)
```

```
    year  count        avg    std_dev
0   1952    142  49.057620  12.225956
1   1957    142  51.507401  12.231286
2   1962    142  53.609249  12.097245
3   1967    142  55.678290  11.718858
4   1972    142  57.647386  11.381953
..   ...    ...        ...        ...
7   1987    142  63.212613  10.556285
8   1992    142  64.160338  11.227380
9   1997    142  65.014676  11.559439
10  2002    142  65.694923  12.279823
11  2007    142  67.007423  12.073021

[12 rows x 4 columns]
```

```python
def my_zscore(x):
  '''Calculates the z-score of provided data
  'x' is a vector or series of values
  '''
  return((x - x.mean()) / x.std())
```

```python
transform_z = df.groupby('year')["lifeExp"].transform(my_zscore)

print(transform_z)
```

```
0       -1.656854
1       -1.731249
2       -1.786543
3       -1.848157
4       -1.894173
           ...
1699    -0.081621
1700    -0.336974
1701    -1.574962
1702    -2.093346
1703    -1.948180
Name: lifeExp, Length: 1704, dtype: float64
```

```python
# note the number of rows in our data
print(df.shape)
```

```
(1704, 6)
```

```python
# note the number of values in our transformation
print(transform_z.shape)
```

```
(1704,)
```

```python
from scipy.stats import zscore

# calculate a grouped zscore
```

```python
sp_z_grouped = df.groupby('year')["lifeExp"].transform(zscore)

# calculate a nongrouped zscore
sp_z_nogroup = zscore(df["lifeExp"])
```

```python
import seaborn as sns
import numpy as np

# set the seed so results are deterministic
np.random.seed(42)

# sample 10 rows from tips
tips_10 = sns.load_dataset("tips").sample(10)

# randomly pick 4 'total_bill' values and turn them into missing
tips_10.loc[
    np.random.permutation(tips_10.index)[:4],
    "total_bill"
] = np.NaN

print(tips_10)
```

```
     total_bill   tip     sex smoker   day    time  size
24        19.82  3.18    Male     No   Sat  Dinner     2
6          8.77  2.00    Male     No   Sun  Dinner     2
153         NaN  2.00    Male     No   Sun  Dinner     4
211         NaN  5.16    Male    Yes   Sat  Dinner     4
198         NaN  2.00  Female    Yes  Thur   Lunch     2
176         NaN  2.00    Male    Yes   Sun  Dinner     2
192       28.44  2.56    Male    Yes  Thur   Lunch     2
124       12.48  2.52  Female     No  Thur   Lunch     2
9         14.78  3.23    Male     No   Sun  Dinner     2
101       15.38  3.00  Female    Yes   Fri  Dinner     2
```

```python
count_sex = tips_10.groupby('sex').count()
print(count_sex)
```

```
        total_bill  tip  smoker  day  time  size
sex
Male             4    7       7    7     7     7
Female           2    3       3    3     3     3
```

```python
def fill_na_mean(x):
  """Returns the average of a given vector"""
  avg = x.mean()
  return x.fillna(avg)


# calculate a mean 'total_bill' by 'sex'
total_bill_group_mean = (
  tips_10
  .groupby("sex")
  .total_bill
  .transform(fill_na_mean)
)

# assign to a new column in the original data
# you can also replace the original column by using 'total_bill'
tips_10["fill_total_bill"] = total_bill_group_mean
```

```python
print(tips_10[['sex', 'total_bill', 'fill_total_bill']])
```

```
        sex  total_bill  fill_total_bill
24     Male       19.82          19.8200
6      Male        8.77           8.7700
153    Male         NaN          17.9525
211    Male         NaN          17.9525
198  Female         NaN          13.9300
176    Male         NaN          17.9525
192    Male       28.44          28.4400
124  Female       12.48          12.4800
9      Male       14.78          14.7800
101  Female       15.38          15.3800
```

```python
# load the tips data set
tips = sns.load_dataset('tips')

# note the number of rows in the original data
print(tips.shape)
```

```
(244, 7)
```

```python
# look at the frequency counts for the table size
print(tips['size'].value_counts())
```

```
2    156
3     38
4     37
5      5
1      4
6      4
Name: size, dtype: int64
```

```python
# filter the data such that each group has more than 30 observations
tips_filtered = (
  tips
  .groupby("size")
  .filter(lambda x: x["size"].count() >= 30)
)
```

```
print(tips_filtered.shape)
```

```
(231, 7)
```

```
print(tips_filtered['size'].value_counts())
```

```
2    156
3     38
4     37
Name: size, dtype: int64
```

```
tips_10 = sns.load_dataset('tips').sample(10, random_state=42)
print(tips_10)
```

```
     total_bill   tip     sex smoker   day    time  size
24        19.82  3.18    Male     No   Sat  Dinner     2
6          8.77  2.00    Male     No   Sun  Dinner     2
153       24.55  2.00    Male     No   Sun  Dinner     4
211       25.89  5.16    Male    Yes   Sat  Dinner     4
198       13.00  2.00  Female    Yes  Thur   Lunch     2
176       17.89  2.00    Male    Yes   Sun  Dinner     2
192       28.44  2.56    Male    Yes  Thur   Lunch     2
124       12.48  2.52  Female     No  Thur   Lunch     2
9         14.78  3.23    Male     No   Sun  Dinner     2
101       15.38  3.00  Female    Yes   Fri  Dinner     2
```

```
# save just the grouped object
grouped = tips_10.groupby('sex')

# note that we just get back the object and its memory location
print(grouped)
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x15ed37880>
```

```python
# see the actual groups of the groupby
# it returns only the index
print(grouped.groups)
```

```
{'Male': [24, 6, 153, 211, 176, 192, 9], 'Female': [198, 124, 101]}
```

```python
# calculate the mean on relevant columns
avgs = grouped.mean()
print(avgs)
```

```
        total_bill       tip      size
sex
Male         20.02  2.875714  2.571429
Female       13.62  2.506667  2.000000
```

```python
# list all the columns
print(tips_10.columns)
```

```
Index(['total_bill', 'tip', 'sex', 'smoker', 'day', 'time', 'size'],
 dtype='object')
```

```python
# get the 'Female' group
female = grouped.get_group('Female')
print(female)
```

```
     total_bill   tip     sex smoker   day    time  size
198       13.00  2.00  Female    Yes  Thur   Lunch     2
124       12.48  2.52  Female     No  Thur   Lunch     2
101       15.38  3.00  Female    Yes   Fri  Dinner     2
```

```python
for sex_group in grouped:
    print(sex_group)
```

```
('Male',      total_bill   tip   sex   smoker  day        time  size
24         19.82  3.18  Male     No   Sat     Dinner     2
6           8.77  2.00  Male     No   Sun     Dinner     2
153        24.55  2.00  Male     No   Sun     Dinner     4
211        25.89  5.16  Male     Yes  Sat     Dinner     4
176        17.89  2.00  Male     Yes  Sun     Dinner     2
192        28.44  2.56  Male     Yes  Thur    Lunch      2
9          14.78  3.23  Male     No   Sun     Dinner     2)
('Female',      total_bill   tip      sex smoker   day    time  size
198        13.00  2.00  Female   Yes  Thur    Lunch      2
124        12.48  2.52  Female   No   Thur    Lunch      2
101        15.38  3.00  Female   Yes  Fri     Dinner     2)
```

```python
# you can't really get the 0 element from the grouped object
print(grouped[0])
```

```
KeyError: 'Column not found: 0'
```

```python
for sex_group in grouped:
    # get the type of the object (tuple)
    print(f'the type is: {type(sex_group)}\n')

    # get the length of the object (2 elements)
    print(f'the length is: {len(sex_group)}\n')

    # get the first element
    first_element = sex_group[0]
    print(f'the first element is: {first_element}\n')

    # the type of the first element (string)
    print(f'it has a type of: {type(sex_group[0])}\n')

    # get the second element
```

```python
    second_element = sex_group[1]
    print(f'the second element is:\n{second_element}\n')

    # get the type of the second element (dataframe)
    print(f'it has a type of: {type(second_element)}\n')

    # print what we have
    print(f'what we have:')
    print(sex_group)

    # stop after first iteration
    break
```

```
the type is: <class 'tuple'>

the length is: 2

the first element is: Male

it has a type of: <class 'str'>

the second element is:
     total_bill   tip   sex smoker   day    time  size
24        19.82  3.18  Male     No   Sat  Dinner     2
6          8.77  2.00  Male     No   Sun  Dinner     2
153       24.55  2.00  Male     No   Sun  Dinner     4
211       25.89  5.16  Male    Yes   Sat  Dinner     4
176       17.89  2.00  Male    Yes   Sun  Dinner     2
192       28.44  2.56  Male    Yes  Thur   Lunch     2
9         14.78  3.23  Male     No   Sun  Dinner     2

it has a type of: <class 'pandas.core.frame.DataFrame'>
```

```
what we have:
('Male',      total_bill   tip  sex  smoker   day      time  size
24         19.82  3.18  Male     No   Sat      Dinner   2
6           8.77  2.00  Male     No   Sun      Dinner   2
153        24.55  2.00  Male     No   Sun      Dinner   4
211        25.89  5.16  Male     Yes  Sat      Dinner   4
176        17.89  2.00  Male     Yes  Sun      Dinner   2
192        28.44  2.56  Male     Yes  Thur     Lunch    2
9          14.78  3.23  Male     No   Sun      Dinner   2)
```

```python
# mean by sex and time
bill_sex_time = tips_10.groupby(['sex', 'time'])

group_avg = bill_sex_time.mean()
```

```python
# type of the group_avg
print(type(group_avg))
```

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

```python
print(group_avg.columns)
```

```
Index(['total_bill', 'tip', 'size'], dtype='object')
```

```python
print(group_avg.index)
```

```
MultiIndex([(  'Male',  'Lunch'),
            (  'Male', 'Dinner'),
            ('Female',  'Lunch'),
            ('Female', 'Dinner')],
           names=['sex', 'time'])
```

```python
group_method = tips_10.groupby(['sex', 'time']).mean().reset_index()
print(group_method)
```

```
      sex    time  total_bill       tip      size
0    Male   Lunch   28.440000  2.560000  2.000000
1    Male  Dinner   18.616667  2.928333  2.666667
2  Female   Lunch   12.740000  2.260000  2.000000
3  Female  Dinner   15.380000  3.000000  2.000000
```

```python
group_param = tips_10.groupby(['sex', 'time'], as_index=False).mean()
print(group_param)
```

```
      sex    time  total_bill       tip      size
0    Male   Lunch   28.440000  2.560000  2.000000
1    Male  Dinner   18.616667  2.928333  2.666667
2  Female   Lunch   12.740000  2.260000  2.000000
3  Female  Dinner   15.380000  3.000000  2.000000
```

```python
# notice that we can even read a compressed zip file of a csv
intv_df = pd.read_csv('data/epi_sim.zip')

print(intv_df)
```

```
         ig_type  intervened        pid  rep  sid        tr
0              3          40  294524448    1  201  0.000135
1              3          40  294571037    1  201  0.000135
2              3          40  290699504    1  201  0.000135
3              3          40  288354895    1  201  0.000135
4              3          40  292271290    1  201  0.000135
...          ...         ...        ...  ...  ...       ...
9434648        2          87  345636694    2  201  0.000166
9434649        3          87  295125214    2  201  0.000166
9434650        2          89  292571119    2  201  0.000166
9434651        3          89  292528142    2  201  0.000166
9434652        2          95  291956763    2  201  0.000166

[9434653 rows x 6 columns]
```

```python
count_only = (
  intv_df
  .groupby(["rep", "intervened", "tr"])
  ["ig_type"]
  .count()
)

print(count_only)
```

```
rep  intervened  tr      
0    8           0.000166    1
     9           0.000152    3
                 0.000166    1
     10          0.000152    1
                 0.000166    1
                            ..
2    193         0.000135    1
                 0.000152    1
     195         0.000135    1
     198         0.000166    1
     199         0.000135    1
Name: ig_type, Length: 1196, dtype: int64
```

```python
print(type(count_only))
```

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

```python
count_mean = count_only.groupby(level=[0, 1, 2]).mean()
print(count_mean.head())
```

```
rep  intervened  tr
0    8           0.000166    1.0
     9           0.000152    3.0
                 0.000166    1.0
     10          0.000152    1.0
                 0.000166    1.0
Name: ig_type, dtype: float64
```

```
count_mean = (
    intv_df
    .groupby(["rep", "intervened", "tr"])["ig_type"]
    .count()
    .groupby(level=[0, 1, 2])
    .mean()
)
```

```
import seaborn as sns
import matplotlib.pyplot as plt


fig = sns.lmplot(
  data=count_mean.reset_index(),
  x="intervened",
  y="ig_type",
  hue="rep",
  col="tr",
  fit_reg=False,
  palette="viridis"
)

plt.show()
```

```
cumulative_count = (
  intv_df
  .groupby(["rep", "intervened", "tr"])
  ["ig_type"]
  .count()
  .groupby(level=["rep"])
  .cumsum()
  .reset_index()
)

fig = sns.lmplot(
 data=cumulative_count,
 x="intervened",
 y="ig_type",
 hue="rep",
 col="tr",
 fit_reg=False,
 palette="viridis"
)
plt.show()
```

```python
# Just import the numpy missing values
from numpy import NaN, NAN, nan
```

```python
# set the location for data
visited_file = 'data/survey_visited.csv'
```

```python
print(pd.read_csv(visited_file))
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3         NaN
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

```python
print(pd.read_csv(visited_file, keep_default_na=False))
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3            
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

```
print(
  pd.read_csv(visited_file, na_values=[""], keep_default_na=False)
)
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3         NaN
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

```python
visited = pd.read_csv('data/survey_visited.csv')
survey = pd.read_csv('data/survey_survey.csv')
```

```python
print(visited)
```

```
   ident   site       dated
0    619   DR-1  1927-02-08
1    622   DR-1  1927-02-10
2    734   DR-3  1939-01-07
3    735   DR-3  1930-01-12
4    751   DR-3  1930-02-26
5    752   DR-3         NaN
6    837  MSK-4  1932-01-14
7    844   DR-1  1932-03-22
```

```python
print(survey)
```

```
    taken person quant  reading
0     619   dyer   rad     9.82
1     619   dyer   sal     0.13
```

```
2     622   dyer   rad     7.80
3     622   dyer   sal     0.09
4     734     pb   rad     8.41
..    ...    ...   ...      ...
16    752    roe   sal    41.60
17    837   lake   rad     1.46
18    837   lake   sal     0.21
19    837    roe   sal    22.50
20    844    roe   rad    11.25

[21 rows x 4 columns]
```

```
vs = visited.merge(survey, left_on='ident', right_on='taken')
print(vs)
```

```
    ident   site       dated  taken person quant  reading
0     619   DR-1  1927-02-08    619   dyer   rad     9.82
1     619   DR-1  1927-02-08    619   dyer   sal     0.13
2     622   DR-1  1927-02-10    622   dyer   rad     7.80
3     622   DR-1  1927-02-10    622   dyer   sal     0.09
4     734   DR-3  1939-01-07    734     pb   rad     8.41
..    ...    ...         ...    ...    ...   ...      ...
16    752   DR-3         NaN    752    roe   sal    41.60
17    837  MSK-4  1932-01-14    837   lake   rad     1.46
18    837  MSK-4  1932-01-14    837   lake   sal     0.21
19    837  MSK-4  1932-01-14    837    roe   sal    22.50
20    844   DR-1  1932-03-22    844    roe   rad    11.25

[21 rows x 7 columns]
```

```python
# missing value in a series
num_legs = pd.Series({'goat': 4, 'amoeba': nan})
print(num_legs)
```

```
goat      4.0
amoeba    NaN
dtype: float64
```

```python
# missing value in a dataframe
scientists = pd.DataFrame(
  {
```

```python
    "Name": ["Rosaline Franklin", "William Gosset"],
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
    "missing": [NaN, nan],
  }
)
print(scientists)
```

```
                Name    Occupation        Born        Died  missing
0  Rosaline Franklin       Chemist  1920-07-25  1958-04-16      NaN
1     William Gosset  Statistician  1876-06-13  1937-10-16      NaN
```

```python
# create a new dataframe
scientists = pd.DataFrame(
  {
    "Name": ["Rosaline Franklin", "William Gosset"],
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
  }
)

# assign a column of missing values
scientists["missing"] = nan

print(scientists)
```

```
                Name    Occupation        Born        Died  missing
0  Rosaline Franklin       Chemist  1920-07-25  1958-04-16      NaN
1     William Gosset  Statistician  1876-06-13  1937-10-16      NaN
```

```python
gapminder = pd.read_csv('data/gapminder.tsv', sep='\t')

life_exp = gapminder.groupby(['year'])['lifeExp'].mean()
print(life_exp)
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
          ...
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, Length: 12, dtype: float64
```

```python
# subset
y2000 = life_exp[life_exp.index > 2000]
print(y2000)
```

```
year
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

```python
# reindex
print(y2000.reindex(range(2000, 2010)))
```

```
year
2000          NaN
2001          NaN
2002    65.694923
2003          NaN
2004          NaN
2005          NaN
2006          NaN
2007    67.007423
2008          NaN
2009          NaN
Name: lifeExp, dtype: float64
```

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

```python
# count the number of non-missing values
print(ebola.count())
```

```
Date                    122
Day                     122
Cases_Guinea             93
Cases_Liberia            83
Cases_SierraLeone        87
                       ...
Deaths_Nigeria           38
Deaths_Senegal           22
Deaths_UnitedStates      18
Deaths_Spain             16
Deaths_Mali              12
Length: 18, dtype: int64
```

```python
num_rows = ebola.shape[0]
num_missing = num_rows - ebola.count()
print(num_missing)
```

```
Date                      0
Day                       0
Cases_Guinea             29
Cases_Liberia            39
Cases_SierraLeone        35
                       ...
Deaths_Nigeria           84
Deaths_Senegal          100
Deaths_UnitedStates     104
Deaths_Spain            106
Deaths_Mali             110
Length: 18, dtype: int64
```

```python
import numpy as np

print(np.count_nonzero(ebola.isnull()))
```

```
1214
```

```python
print(np.count_nonzero(ebola['Cases_Guinea'].isnull()))
```

```
29
```

```python
# value counts from the Cases_Guinea column
cnts = ebola.Cases_Guinea.value_counts(dropna=False)
print(cnts)
```

```
NaN       29
86.0       3
495.0      2
112.0      2
390.0      2
          ..
1199.0     1
1298.0     1
1350.0     1
1472.0     1
49.0       1
Name: Cases_Guinea, Length: 89, dtype: int64
```

```python
# select the values in the Series where the index is a NaN value
print(cnts.loc[pd.isnull(cnts.index)])
```

```
NaN    29
Name: Cases_Guinea, dtype: int64
```

```
# check if the value is missing, and sum up the results
print(ebola.Cases_Guinea.isnull().sum())
```

```
29
```

```
# fill the missing values to 0 and only look at the first 5 columns
print(ebola.fillna(0).iloc[:, 0:5])
```

```
           Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
0      1/5/2015  289        2776.0            0.0            10030.0
1      1/4/2015  288        2775.0            0.0             9780.0
2      1/3/2015  287        2769.0         8166.0             9722.0
3      1/2/2015  286           0.0         8157.0                0.0
4    12/31/2014  284        2730.0         8115.0             9633.0
..          ...  ...           ...            ...                ...
117   3/27/2014    5         103.0            8.0                6.0
118   3/26/2014    4          86.0            0.0                0.0
119   3/25/2014    3          86.0            0.0                0.0
120   3/24/2014    2          86.0            0.0                0.0
121   3/22/2014    0          49.0            0.0                0.0

[122 rows x 5 columns]
```

```
print(ebola.fillna(method='ffill').iloc[:, 0:5])
```

```
           Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
0      1/5/2015  289        2776.0            NaN            10030.0
1      1/4/2015  288        2775.0            NaN             9780.0
```

```
2      1/3/2015  287         2769.0          8166.0             9722.0
3      1/2/2015  286         2769.0          8157.0             9722.0
4    12/31/2014  284         2730.0          8115.0             9633.0
..          ...  ...            ...             ...                ...
117   3/27/2014    5          103.0             8.0                6.0
118   3/26/2014    4           86.0             8.0                6.0
119   3/25/2014    3           86.0             8.0                6.0
120   3/24/2014    2           86.0             8.0                6.0
121   3/22/2014    0           49.0             8.0                6.0

[122 rows x 5 columns]
```

```
print(ebola.fillna(method='bfill').iloc[:, 0:5])
```

```
           Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
0      1/5/2015  289        2776.0         8166.0            10030.0
1      1/4/2015  288        2775.0         8166.0             9780.0
2      1/3/2015  287        2769.0         8166.0             9722.0
3      1/2/2015  286        2730.0         8157.0             9633.0
4    12/31/2014  284        2730.0         8115.0             9633.0
..          ...  ...           ...            ...                ...
117   3/27/2014    5         103.0            8.0                6.0
118   3/26/2014    4          86.0            NaN                NaN
119   3/25/2014    3          86.0            NaN                NaN
120   3/24/2014    2          86.0            NaN                NaN
121   3/22/2014    0          49.0            NaN                NaN

[122 rows x 5 columns]
```

```
print(ebola.interpolate().iloc[:, 0:5])
```

```
           Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
0      1/5/2015  289        2776.0            NaN            10030.0
1      1/4/2015  288        2775.0            NaN             9780.0
```

```
2       1/3/2015  287          2769.0          8166.0            9722.0
3       1/2/2015  286          2749.5          8157.0            9677.5
4     12/31/2014  284          2730.0          8115.0            9633.0
..           ...  ...             ...             ...               ...
117    3/27/2014    5           103.0             8.0               6.0
118    3/26/2014    4            86.0             8.0               6.0
119    3/25/2014    3            86.0             8.0               6.0
120    3/24/2014    2            86.0             8.0               6.0
121    3/22/2014    0            49.0             8.0               6.0

[122 rows x 5 columns]
```

```
ebola_dropna = ebola.dropna()
print(ebola_dropna.shape)
```

```
(1, 18)
```

```
print(ebola_dropna)
```

```
          Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone  \
19  11/18/2014  241        2047.0         7082.0             6190.0

    Cases_Nigeria  Cases_Senegal  Cases_UnitedStates  Cases_Spain  \
19           20.0            1.0                 4.0          1.0

    Cases_Mali  Deaths_Guinea  Deaths_Liberia  Deaths_SierraLeone  \
19         6.0         1214.0          2963.0              1267.0

    Deaths_Nigeria  Deaths_Senegal  Deaths_UnitedStates  \
19             8.0             0.0                  1.0

    Deaths_Spain  Deaths_Mali
19           0.0          6.0
```

```python
ebola_subset = ebola.loc[
    :,
    [
        "Cases_Guinea",
        "Cases_Liberia",
        "Cases_SierraLeone",
        "Cases_multiple",
    ],
]
print(ebola_subset.head(n=10))
```

```
   Cases_Guinea  Cases_Liberia  Cases_SierraLeone  Cases_multiple
0        2776.0            NaN            10030.0             NaN
1        2775.0            NaN             9780.0             NaN
2        2769.0         8166.0             9722.0         20657.0
3           NaN         8157.0                NaN             NaN
4        2730.0         8115.0             9633.0         20478.0
5        2706.0         8018.0             9446.0         20170.0
6        2695.0            NaN             9409.0             NaN
7        2630.0         7977.0             9203.0         19810.0
8        2597.0            NaN             9004.0             NaN
9        2571.0         7862.0             8939.0         19372.0
```

```python
# skipping missing values is True by default
print(ebola.Cases_Guinea.sum(skipna = True))
```

```
84729.0
```

```python
print(ebola.Cases_Guinea.sum(skipna = False))
```

```
nan
```

```python
scientists = pd.DataFrame(
  {
    "Name": ["Rosaline Franklin", "William Gosset"],
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
    "Age": [37, 61]
```

```
    "Age": [37, 61]
  }
)

print(scientists)
```

```
                Name    Occupation        Born        Died  Age
0  Rosaline Franklin       Chemist  1920-07-25  1958-04-16   37
1     William Gosset  Statistician  1876-06-13  1937-10-16   61
```

```
print(scientists.dtypes)
```

```
Name          object
Occupation    object
Born          object
Died          object
Age            int64
dtype: object
```

```
scientists.loc[1, "Name"] = pd.NA
scientists.loc[1, "Age"] = pd.NA

print(scientists)
```

```
                Name    Occupation        Born        Died   Age
0  Rosaline Franklin       Chemist  1920-07-25  1958-04-16    37
1               <NA>  Statistician  1876-06-13  1937-10-16  <NA>
```

```
print(scientists.dtypes)
```

```
Name          object
Occupation    object
Born          object
```

```
Died          object
Age           object
dtype: object
```

```
scientists = pd.DataFrame(
  {
    "Name": ["Rosaline Franklin", "William Gosset"],
    "Occupation": ["Chemist", "Statistician"],
    "Born": ["1920-07-25", "1876-06-13"],
    "Died": ["1958-04-16", "1937-10-16"],
    "Age": [37, 61]
  }
)

scientists.loc[1, "Name"] = np.NaN
scientists.loc[1, "Age"] = np.NaN

print(scientists.dtypes)
```

```
Name           object
Occupation     object
Born           object
Died           object
Age           float64
dtype: object
```

```python
import pandas as pd
import seaborn as sns

tips = sns.load_data set("tips")
```

```python
# convert the category sex column into a string dtype
tips['sex_str'] = tips['sex'].astype(str)
```

```python
# convert total_bill into a string
tips['total_bill'] = tips['total_bill'].astype(str)
print(tips.dtypes)
```

```
total_bill      object
tip            float64
sex           category
smoker        category
day           category
time          category
size             int64
sex_str         object
dtype: object
```

```python
# convert it back to a float
tips['total_bill'] = tips['total_bill'].astype(float)
print(tips.dtypes)
```

```
total_bill     float64
tip            float64
sex           category
smoker        category
day           category
time          category
size             int64
sex_str         object
dtype: object
```

```python
# subset the tips data
tips_sub_miss = tips.head(10).copy()
```

```python
# assign some 'missing' values
tips_sub_miss.loc[[1, 3, 5, 7], 'total_bill'] = 'missing'
```

```python
print(tips_sub_miss)
```

```
  total_bill   tip     sex smoker  day    time  size sex_str
0      16.99  1.01  Female     No  Sun  Dinner     2  Female
1    missing  1.66    Male     No  Sun  Dinner     3    Male
2      21.01  3.50    Male     No  Sun  Dinner     3    Male
3    missing  3.31    Male     No  Sun  Dinner     2    Male
4      24.59  3.61  Female     No  Sun  Dinner     4  Female
5    missing  4.71    Male     No  Sun  Dinner     4    Male
6       8.77  2.00    Male     No  Sun  Dinner     2    Male
7    missing  3.12    Male     No  Sun  Dinner     4    Male
8      15.04  1.96    Male     No  Sun  Dinner     2    Male
9      14.78  3.23    Male     No  Sun  Dinner     2    Male
```

```python
# this will cause an error
tips_sub_miss['total_bill'].astype(float)
```

```
ValueError: could not convert string to float: 'missing'
```

```python
# this will cause an error
pd.to_numeric(tips_sub_miss['total_bill'])
```

```
ValueError: Unable to parse string "missing" at position 1
```

```python
tips_sub_miss["total_bill"] = pd.to_numeric(
    tips_sub_miss["total_bill"], errors="ignore"
)

print(tips_sub_miss)
```

```
  total_bill   tip     sex smoker  day    time  size sex_str
0      16.99  1.01  Female     No  Sun  Dinner     2  Female
1    missing  1.66    Male     No  Sun  Dinner     3    Male
2      21.01  3.50    Male     No  Sun  Dinner     3    Male
3    missing  3.31    Male     No  Sun  Dinner     2    Male
4      24.59  3.61  Female     No  Sun  Dinner     4  Female
5    missing  4.71    Male     No  Sun  Dinner     4    Male
6       8.77  2.00    Male     No  Sun  Dinner     2    Male
7    missing  3.12    Male     No  Sun  Dinner     4    Male
8      15.04  1.96    Male     No  Sun  Dinner     2    Male
9      14.78  3.23    Male     No  Sun  Dinner     2    Male
```

```
print(tips_sub_miss.dtypes)
```

```
total_bill      object
tip            float64
sex           category
smoker        category
day           category
time          category
size             int64
sex_str         object
dtype: object
```

```python
tips_sub_miss["total_bill"] = pd.to_numeric(
    tips_sub_miss["total_bill"], errors="coerce"
)
```

```python
print(tips_sub_miss)
```

```
   total_bill   tip     sex smoker  day    time  size sex_str
0       16.99  1.01  Female     No  Sun  Dinner     2  Female
1         NaN  1.66    Male     No  Sun  Dinner     3    Male
2       21.01  3.50    Male     No  Sun  Dinner     3    Male
3         NaN  3.31    Male     No  Sun  Dinner     2    Male
4       24.59  3.61  Female     No  Sun  Dinner     4  Female
5         NaN  4.71    Male     No  Sun  Dinner     4    Male
6        8.77  2.00    Male     No  Sun  Dinner     2    Male
7         NaN  3.12    Male     No  Sun  Dinner     4    Male
8       15.04  1.96    Male     No  Sun  Dinner     2    Male
9       14.78  3.23    Male     No  Sun  Dinner     2    Male
```

```python
print(tips_sub_miss.dtypes)
```

```
total_bill     float64
tip            float64
sex           category

smoker        category
day           category
time          category
size             int64
sex_str         object
dtype: object
```

```python
# convert the sex column into a string object first
tips['sex'] = tips['sex'].astype('str')
print(tips.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   total_bill  244 non-null    float64
 1   tip         244 non-null    float64
 2   sex         244 non-null    object
 3   smoker      244 non-null    category
 4   day         244 non-null    category
 5   time        244 non-null    category
 6   size        244 non-null    int64
 7   sex_str     244 non-null    object
dtypes: category(3), float64(2), int64(1), object(2)
memory usage: 10.8+ KB
None
```

```python
# convert the sex column back into categorical data
tips['sex'] = tips['sex'].astype('category')
print(tips.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   total_bill  244 non-null    float64
 1   tip         244 non-null    float64
 2   sex         244 non-null    category
 3   smoker      244 non-null    category
 4   day         244 non-null    category
 5   time        244 non-null    category
 6   size        244 non-null    int64
 7   sex_str     244 non-null    object
dtypes: category(4), float64(2), int64(1), object(1)
memory usage: 9.3+ KB
None
```

```
# get the first 3 characters
# note index 3 is really the 4th character
print(word[0:3])
```

```
gra
```

```
# get the last letter from "a scratch"
print(sent[-1])
```

```
h
```

```
# note that the last index is one position is smaller than
# the number returned for len
s_len = len(sent)
print(s_len)
```

```
9
```

```
print(sent[2:s_len])
```

```
scratch
```

```
# step by 2, to get every other character
print(sent[::2])
```

```
asrth
```

```
coords = ' '.join([d1, m1, s1, u1, d2, m2, s2, u2])
print(coords)
```

```
40° 46' 52.837" N 73° 58' 26.302" W
```

```python
coords.split(" ")
```

```
['40°', "46'", '52.837"', 'N', '73°', "58'", '26.302"', 'W']
```

```python
multi_str = """Guard: What? Ridden on a horse?
King Arthur: Yes!
Guard: You're using coconuts!
King Arthur: What?
Guard: You've got ... coconut[s] and you're bangin' 'em together.
"""

print(multi_str)
```

```
Guard: What? Ridden on a horse?
King Arthur: Yes!
Guard: You're using coconuts!
King Arthur: What?
Guard: You've got ... coconut[s] and you're bangin' 'em together.
```

```python
multi_str_split = multi_str.splitlines()
```

```python
print(multi_str_split)
```

```
[
  "Guard: What? Ridden on a horse?",
  "King Arthur: Yes!",
  "Guard: You're using coconuts!",
  "King Arthur: What?",
  "Guard: You've got ... coconut[s] and you're bangin' 'em together."
]
```

```python
guard = multi_str_split[::2]
```

```python
print(guard)
```

```
[
  "Guard: What? Ridden on a horse?",
  "Guard: You're using coconuts!",
  "Guard: You've got ... coconut[s] and you're bangin' 'em together."
]
```

```python
guard = multi_str.replace("Guard: ", "").splitlines()[::2]
```

```python
print(guard)
```

```
[
  "What? Ridden on a horse?",
  "You're using coconuts!",
  "You've got ... coconut[s] and you're bangin' 'em together."
]
```

```python
num = 7
s = f"I only know {num} digits of pi."
print(s)
```

```
I only know 7 digits of pi.
```

```python
const = "e"
value = 2.718
s = f"Some digits of {const}: {value}"
print(s)
```

```
Some digits of e: 2.718
```

```python
lat = "40.7815° N"
lon = "73.9733° W"
s = f"Hayden Planetarium Coordinates: {lat}, {lon}"
print(s)
```

```
Hayden Planetarium Coordinates: 40.7815° N, 73.9733° W
```

```python
word = "scratch"

s = f"""Black Knight: 'Tis but a {word}.
King Arthur: A {word}? Your arm's off!
"""
print(s)
```

```
Black Knight: 'Tis but a scratch.
King Arthur: A scratch? Your arm's off!
```

```python
p = 3.14159265359
print(f"Some digits of pi: {p}")
```

```
Some digits of pi: 3.14159265359
```

```python
digits = 67890
s = f"In 2005, Lu Chao of China recited {67890:,} digits of pi."
print(s)
```

```
In 2005, Lu Chao of China recited 67,890 digits of pi.
```

```python
prop = 7 / 67890
s = f"I remember {prop:.4} or {prop:.4%} of what Lu Chao recited."
print(s)
```

```
I remember 0.0001031 or 0.0103% of what Lu Chao recited.
```

```python
id = 42
print(f"My ID number is {id:05d}")
```

```
My ID number is 00042
```

```python
id_zfill = "42".zfill(5)
print(f"My ID number is {id_zfill}")
```

```
My ID number is 00042
```

```python
print(f"My ID number is {'42'.zfill(5)}")
```

```
My ID number is 00042
```

```python
m = re.match(pattern='\d\d\d\d\d\d\d\d\d\d', string=tele_num)
print(type(m))
```

```
<class 're.Match'>
```

```python
print(m)
```

```
<re.Match object; span=(0, 10), match='1234567890'>
```

```python
# get the first index of the string match
print(m.start())
```

```
0
```

```python
# get the last index of the string match
print(m.end())
```

```
10
```

```python
# get the first and last index of the string match
print(m.span())
```

```
(0, 10)
```

```python
# the string that matched the pattern
print(m.group())
```

```
1234567890
```

```python
tele_num_spaces = '123 456 7890'
```

```python
# we can simplify the previous pattern
m = re.match(pattern='\d{10}', string=tele_num_spaces)
print(m)
```

```
None
```

```python
# you may see the RegEx pattern as a separate variable
# because it can get long and
# make the actual match function call hard to read
p = '\d{3}\s?\d{3}\s?\d{4}'
m = re.match(pattern=p, string=tele_num_spaces)
print(m)
```

```
<re.Match object; span=(0, 12), match='123 456 7890'>
```

```python
tele_num_space_paren_dash = '(123) 456-7890'
p = '\(?\d{3}\)?\s?\d{3}\s?-?\d{4}'
m = re.match(pattern=p, string=tele_num_space_paren_dash)
print(m)
```

```
<re.Match object; span=(0, 14), match='(123) 456-7890'>
```

```python
cnty_tele_num_space_paren_dash = '+1 (123) 456-7890'
p = '\+?1\s?\(?\d{3}\)?\s?\d{3}\s?-?\d{4}'
m = re.match(pattern=p, string=cnty_tele_num_space_paren_dash)
print(m)
```

```
<re.Match object; span=(0, 17), match='+1 (123) 456-7890'>
```

```python
"multiple" "strings" "next" "to" "each" "other"
```

```
'multiplestringsnexttoeachother'
```

```python
p = (
  '\+?'
  '1'
  '\s?'
  '\(?'
  '\d{3}'
  '\)?'
  '\s?'
  '\d{3}'
  '\s?'
  '-?'
  '\d{4}'
)
print(p)
```

```
\+?1\s?\(?\d{3}\)?\s?\d{3}\s?-?\d{4}
```

```python
p = (
  '\+?'    # maybe starts with a +
  '1'      # the number 1
  '\s?'    # maybe there's a whitespace
  '\(?'    # maybe there's an open round parenthesis (
  '\d{3}'  # 3 numbers
  '\)?'    # maybe there's a closing round parenthesis )
  '\s?'    # maybe there's a whitespace
  '\d{3}'  # 3 numbers
  '\s?'    # maybe there's a whitespace
  '-?'     # maybe there's a dash character
  '\d{4}'  # 4 numbers
)
print(p)
```

```
\+?1\s?\(?\d{3}\)?\s?\d{3}\s?-?\d{4}
```

```python
cnty_tele_num_space_paren_dash = '+1 (123) 456-7890'
m = re.match(pattern=p, string=cnty_tele_num_space_paren_dash)
print(m)
```

```
<re.Match object; span=(0, 17), match='+1 (123) 456-7890'>
```

```python
# python will concatenate 2 strings next to each other
s = (
  "14 Ncuti Gatwa, "
  "13 Jodie Whittaker, war John Hurt, 12 Peter Capaldi, "
  "11 Matt Smith, 10 David Tennant, 9 Christopher Eccleston"
)

print(s)
```

```
14 Ncuti Gatwa, 13 Jodie Whittaker, war John Hurt, 12 Peter Capaldi,
11 Matt Smith, 10 David Tennant, 9 Christopher Eccleston
```

```python
# pattern to match 1 or more digits
p = "\d+"

m = re.findall(pattern=p, string=s)
print(m)
```

```
['14', '13', '12', '11', '10', '9']
```

```python
multi_str = """Guard: What? Ridden on a horse?
King Arthur: Yes!
Guard: You're using coconuts!
King Arthur: What?
Guard: You've got ... coconut[s] and you're bangin' 'em together.
"""

p = '\w+\s?\w+:\s?'

s = re.sub(pattern=p, string=multi_str, repl='')
print(s)
```

```
What? Ridden on a horse?
Yes!
You're using coconuts!
What?
You've got ... coconut[s] and you're bangin' 'em together.
```

```python
guard = s.splitlines()[ ::2]
kinga = s.splitlines()[1::2] # skip the first element
```

```python
print(guard)
```

```
[
  "What? Ridden on a horse?",
  "You're using coconuts!",
  "You've got ... coconut[s] and you're bangin' 'em together."
]
```

```python
print(kinga)
```

```
[
  "Yes!",
  "What?"
]
```

```python
# pattern to match 10 digits
p = re.compile('\d{10}')
s = '1234567890'

# note: calling match on the compiled pattern
# not using the re.match function
m = p.match(s)
print(m)
```

```
<re.Match object; span=(0, 10), match='1234567890'>
```

```python
p = re.compile('\d+')
s = (
  "14 Ncuti Gatwa, "
  "13 Jodie Whittaker, war John Hurt, 12 Peter Capaldi, "
  "11 Matt Smith, 10 David Tennant, 9 Christopher Eccleston"
)

m = p.findall(s)
print(m)
```

```
['14', '13', '12', '11', '10', '9']
```

```python
p = re.compile('\w+\s?\w+:\s?')
s = "Guard: You're using coconuts!"

m = p.sub(string=s, repl='')
print(m)
```

```
You're using coconuts!
```

```python
import regex

# a re example using the regex library
p = regex.compile('\d+')
s = (
  "14 Ncuti Gatwa, "
  "13 Jodie Whittaker, war John Hurt, 12 Peter Capaldi, "
  "11 Matt Smith, 10 David Tennant, 9 Christopher Eccleston"
)

m = p.findall(s)
print(m)
```

```
['14', '13', '12', '11', '10', '9']
```

```python
from datetime import datetime
```

```python
now = datetime.now()
print(f"Last time this chapter was rendered for print: {now}")
```

```
Last time this chapter was rendered for print: 2022-09-01 01:55:41.496795
```

```python
import pandas as pd
ebola = pd.read_csv('data/country_timeseries.csv')

# top left corner of the data
print(ebola.iloc[:5, :5])
```

```
         Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
0    1/5/2015  289        2776.0            NaN            10030.0
1    1/4/2015  288        2775.0            NaN             9780.0
2    1/3/2015  287        2769.0         8166.0             9722.0
3    1/2/2015  286           NaN         8157.0                NaN
4  12/31/2014  284        2730.0         8115.0             9633.0
```

```python
print(ebola.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 18 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Date                 122 non-null    object
 1   Day                  122 non-null    int64
 2   Cases_Guinea         93 non-null     float64
 3   Cases_Liberia        83 non-null     float64
 4   Cases_SierraLeone    87 non-null     float64
```

```
 5   Cases_Nigeria        38 non-null     float64
 6   Cases_Senegal        25 non-null     float64
 7   Cases_UnitedStates   18 non-null     float64
 8   Cases_Spain          16 non-null     float64
 9   Cases_Mali           12 non-null     float64
 10  Deaths_Guinea        92 non-null     float64
 11  Deaths_Liberia       81 non-null     float64
 12  Deaths_SierraLeone   87 non-null     float64
 13  Deaths_Nigeria       38 non-null     float64
 14  Deaths_Senegal       22 non-null     float64
 15  Deaths_UnitedStates  18 non-null     float64
 16  Deaths_Spain         16 non-null     float64
 17  Deaths_Mali          12 non-null     float64
dtypes: float64(16), int64(1), object(1)
memory usage: 17.3+ KB
None
```

```
ebola['date_dt'] = pd.to_datetime(ebola['Date'])
```

```
ebola['date_dt'] = pd.to_datetime(ebola['Date'], format='%m/%d/%Y')
```

```python
print(ebola.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 21 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Date                 122 non-null    object
 1   Day                  122 non-null    int64
 2   Cases_Guinea         93 non-null     float64
 3   Cases_Liberia        83 non-null     float64
 4   Cases_SierraLeone    87 non-null     float64
 5   Cases_Nigeria        38 non-null     float64
 6   Cases_Senegal        25 non-null     float64
 7   Cases_UnitedStates   18 non-null     float64
 8   Cases_Spain          16 non-null     float64
 9   Cases_Mali           12 non-null     float64
 10  Deaths_Guinea        92 non-null     float64

 11  Deaths_Liberia       81 non-null     float64
 12  Deaths_SierraLeone   87 non-null     float64
 13  Deaths_Nigeria       38 non-null     float64
 14  Deaths_Senegal       22 non-null     float64
 15  Deaths_UnitedStates  18 non-null     float64
 16  Deaths_Spain         16 non-null     float64
 17  Deaths_Mali          12 non-null     float64
 18  date_dt              122 non-null    datetime64[ns]
 19  date_dt_a            122 non-null    datetime64[ns]
 20  date_dt_al           122 non-null    datetime64[ns]
dtypes: datetime64[ns](3), float64(16), int64(1), object(1)
memory usage: 20.1+ KB
None
```

```python
ebola = pd.read_csv('data/country_timeseries.csv', parse_dates=["Date"])
print(ebola.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 18 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Date                 122 non-null    datetime64[ns]
 1   Day                  122 non-null    int64
 2   Cases_Guinea         93 non-null     float64
 3   Cases_Liberia        83 non-null     float64
 4   Cases_SierraLeone    87 non-null     float64
 5   Cases_Nigeria        38 non-null     float64
 6   Cases_Senegal        25 non-null     float64
 7   Cases_UnitedStates   18 non-null     float64
 8   Cases_Spain          16 non-null     float64

 9   Cases_Mali           12 non-null     float64
 10  Deaths_Guinea        92 non-null     float64
 11  Deaths_Liberia       81 non-null     float64
 12  Deaths_SierraLeone   87 non-null     float64
 13  Deaths_Nigeria       38 non-null     float64
 14  Deaths_Senegal       22 non-null     float64
 15  Deaths_UnitedStates  18 non-null     float64
 16  Deaths_Spain         16 non-null     float64
 17  Deaths_Mali          12 non-null     float64
dtypes: datetime64[ns](1), float64(16), int64(1)
memory usage: 17.3 KB
None
```

```python
d = pd.to_datetime('2021-12-14')
print(d)
```

```
2021-12-14 00:00:00
```

```python
print(type(d))
```

```
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

```python
ebola['date_dt'] = pd.to_datetime(ebola['Date'])
```

```python
print(ebola[['Date', 'date_dt']])
```

```
           Date    date_dt
0    2015-01-05 2015-01-05
1    2015-01-04 2015-01-04
2    2015-01-03 2015-01-03
3    2015-01-02 2015-01-02
4    2014-12-31 2014-12-31
..          ...        ...
117  2014-03-27 2014-03-27
118  2014-03-26 2014-03-26
119  2014-03-25 2014-03-25
120  2014-03-24 2014-03-24
121  2014-03-22 2014-03-22

[122 rows x 2 columns]
```

```python
ebola['year'] = ebola['date_dt'].dt.year
print(ebola[['Date', 'date_dt', 'year']])
```

```
           Date    date_dt  year
0    2015-01-05 2015-01-05  2015
1    2015-01-04 2015-01-04  2015
2    2015-01-03 2015-01-03  2015
3    2015-01-02 2015-01-02  2015
4    2014-12-31 2014-12-31  2014
..          ...        ...   ...
117  2014-03-27 2014-03-27  2014
118  2014-03-26 2014-03-26  2014
119  2014-03-25 2014-03-25  2014
120  2014-03-24 2014-03-24  2014
121  2014-03-22 2014-03-22  2014

[122 rows x 3 columns]
```

```python
ebola = ebola.assign(
    month=ebola["date_dt"].dt.month,
    day=ebola["date_dt"].dt.day
)
```

```python
print(ebola[['Date', 'date_dt', 'year', 'month', 'day']])
```

```
           Date    date_dt  year  month  day
0    2015-01-05 2015-01-05  2015      1    5
1    2015-01-04 2015-01-04  2015      1    4
2    2015-01-03 2015-01-03  2015      1    3
3    2015-01-02 2015-01-02  2015      1    2
4    2014-12-31 2014-12-31  2014     12   31
..          ...        ...   ...    ...  ...
117  2014-03-27 2014-03-27  2014      3   27
118  2014-03-26 2014-03-26  2014      3   26
119  2014-03-25 2014-03-25  2014      3   25
120  2014-03-24 2014-03-24  2014      3   24
121  2014-03-22 2014-03-22  2014      3   22

[122 rows x 5 columns]
```

```python
print(ebola.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 22 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Date                 122 non-null    datetime64[ns]
 1   Day                  122 non-null    int64
 2   Cases_Guinea         93 non-null     float64
 3   Cases_Liberia        83 non-null     float64
 4   Cases_SierraLeone    87 non-null     float64
 5   Cases_Nigeria        38 non-null     float64
 6   Cases_Senegal        25 non-null     float64
 7   Cases_UnitedStates   18 non-null     float64
 8   Cases_Spain          16 non-null     float64
 9   Cases_Mali           12 non-null     float64
 10  Deaths_Guinea        92 non-null     float64
 11  Deaths_Liberia       81 non-null     float64
 12  Deaths_SierraLeone   87 non-null     float64
 13  Deaths_Nigeria       38 non-null     float64
 14  Deaths_Senegal       22 non-null     float64
```

```python
print(ebola.iloc[-5:, :5])
```

```
          Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
117 2014-03-27    5         103.0            8.0                6.0
118 2014-03-26    4          86.0            NaN                NaN
119 2014-03-25    3          86.0            NaN                NaN
120 2014-03-24    2          86.0            NaN                NaN
121 2014-03-22    0          49.0            NaN                NaN
```

```python
ebola['outbreak_d'] = ebola['date_dt'] - ebola['date_dt'].min()
```

```python
print(ebola[['Date', 'Day', 'outbreak_d']])
```

```
           Date  Day outbreak_d
0    2015-01-05  289   289 days
1    2015-01-04  288   288 days
2    2015-01-03  287   287 days
3    2015-01-02  286   286 days
4    2014-12-31  284   284 days
..          ...  ...        ...
117  2014-03-27    5     5 days
118  2014-03-26    4     4 days
119  2014-03-25    3     3 days
120  2014-03-24    2     2 days
121  2014-03-22    0     0 days

[122 rows x 3 columns]
```

```python
print(ebola.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 23 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   Date                 122 non-null    datetime64[ns]
 1   Day                  122 non-null    int64
 2   Cases_Guinea         93 non-null     float64
 3   Cases_Liberia        83 non-null     float64
 4   Cases_SierraLeone    87 non-null     float64
 5   Cases_Nigeria        38 non-null     float64
 6   Cases_Senegal        25 non-null     float64
 7   Cases_UnitedStates   18 non-null     float64
 8   Cases_Spain          16 non-null     float64
 9   Cases_Mali           12 non-null     float64
 10  Deaths_Guinea        92 non-null     float64
 11  Deaths_Liberia       81 non-null     float64
 12  Deaths_SierraLeone   87 non-null     float64
 13  Deaths_Nigeria       38 non-null     float64
 14  Deaths_Senegal       22 non-null     float64
 15  Deaths_UnitedStates  18 non-null     float64
 16  Deaths_Spain         16 non-null     float64
 17  Deaths_Mali          12 non-null     float64
 18  date_dt              122 non-null    datetime64[ns]
 19  year                 122 non-null    int64
 20  month                122 non-null    int64
 21  day                  122 non-null    int64
 22  outbreak_d           122 non-null    timedelta64[ns]
dtypes: datetime64[ns](2), float64(16), int64(4), timedelta64[ns](1)
memory usage: 22.0 KB
None
```

```
banks = pd.read_csv('data/banklist.csv')
print(banks.head())
```

```
                                           Bank Name  \
0                                Fayette County Bank   
1  Guaranty Bank, (d/b/a BestBank in Georgia & Mi...   
2                                     First NBC Bank   
3                                      Proficio Bank   
4                      Seaway Bank and Trust Company   

                 City  ST   CERT  \
0          Saint Elmo  IL   1802   
1           Milwaukee  WI  30003   
2         New Orleans  LA  58302   
3  Cottonwood Heights  UT  35495   
4             Chicago  IL  19328   

                Acquiring Institution Closing Date Updated Date  
0          United Fidelity Bank, fsb    26-May-17    26-Jul-17  
1  First-Citizens Bank & Trust Company     5-May-17    26-Jul-17  
2                         Whitney Bank    28-Apr-17    26-Jul-17  
3                    Cache Valley Bank     3-Mar-17    18-May-17  
4                  State Bank of Texas    27-Jan-17    18-May-17  
```

```python
banks = pd.read_csv(
    "data/banklist.csv", parse_dates=["Closing Date", "Updated Date"]
)

print(banks.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 553 entries, 0 to 552
Data columns (total 7 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Bank Name              553 non-null    object
 1   City                   553 non-null    object
 2   ST                     553 non-null    object
 3   CERT                   553 non-null    int64
 4   Acquiring Institution  553 non-null    object
 5   Closing Date           553 non-null    datetime64[ns]
 6   Updated Date           553 non-null    datetime64[ns]
dtypes: datetime64[ns](2), int64(1), object(4)
memory usage: 30.4+ KB
None
```

```python
banks = banks.assign(
  closing_quarter=banks['Closing Date'].dt.quarter,
  closing_year=banks['Closing Date'].dt.year
)
```

```python
closing_year = banks.groupby(['closing_year']).size()
```

```python
closing_year_q = (
  banks
  .groupby(['closing_year', 'closing_quarter'])
  .size()
)
```

```python
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax = closing_year.plot()
plt.show()
```

```python
fig, ax = plt.subplots()
ax = closing_year_q.plot()
plt.show()
```

```python
# we can install and use the pandas_datafreader
# to get data from the Internet
import pandas_datareader.data as web

# in this example we are getting stock information about Tesla
tesla = web.DataReader('TSLA', 'yahoo')

print(tesla)
```

```
      Date        High         Low        Open       Close  \
2017-09-05   23.699333   23.059334   23.586666   23.306000
2017-09-06   23.398666   22.770666   23.299999   22.968666
2017-09-07   23.498667   22.896667   23.065332   23.374001
2017-09-08   23.318666   22.820000   23.266001   22.893333
2017-09-11   24.247334   23.333332   23.423332   24.246000

...          ...         ...         ...         ...
2022-08-25   302.959991  291.600006  302.359985  296.070007
2022-08-26   302.000000  287.470001  297.429993  288.089996
2022-08-29   287.739990  280.700012  282.829987  284.820007
2022-08-30   288.480011  272.649994  287.869995  277.700012
2022-08-31   281.250000  271.809998  280.619995  275.609985

      Date      Volume   Adj Close
2017-09-05  57526500.0   23.306000
2017-09-06  61371000.0   22.968666
2017-09-07  63588000.0   23.374001
2017-09-08  48952500.0   22.893333
```

```
2017-09-11  115006500.0   24.246000
       ...          ...         ...
2022-08-25   53230000.0  296.070007
2022-08-26   56905800.0  288.089996
2022-08-29   41864700.0  284.820007
2022-08-30   50541800.0  277.700012
2022-08-31   51788900.0  275.609985

[1257 rows x 6 columns]
```

```python
# the stock data was saved
# so we do not need to rely on the Internet again
# instead we can load the same data set as a file
tesla = pd.read_csv(
  'data/tesla_stock_yahoo.csv', parse_dates=["Date"]
)

print(tesla)
```

```
          Date        Open        High         Low       Close  \
0   2010-06-29   19.000000   25.000000   17.540001   23.889999
1   2010-06-30   25.790001   30.420000   23.299999   23.830000
2   2010-07-01   25.000000   25.920000   20.270000   21.959999
3   2010-07-02   23.000000   23.100000   18.709999   19.200001
4   2010-07-06   20.000000   20.000000   15.830000   16.110001
...        ...         ...         ...         ...         ...
1786 2017-08-02  318.940002  327.119995  311.220001  325.890015
1787 2017-08-03  345.329987  350.000000  343.149994  347.089996
1788 2017-08-04  347.000000  357.269989  343.299988  356.910004
1789 2017-08-07  357.350006  359.480011  352.750000  355.170013
1790 2017-08-08  357.529999  368.579987  357.399994  365.220001

       Adj Close    Volume
0      23.889999  18766300
1      23.830000  17187100
2      21.959999   8218800
3      19.200001   5139800
4      16.110001   6866900
```

```
...          ...        ...
1786  325.890015  13091500
1787  347.089996  13535000
1788  356.910004   9198400
1789  355.170013   6276900
1790  365.220001   7449837

[1791 rows x 7 columns]
```

```
print(
  tesla.loc[
    (tesla.Date.dt.year == 2010) & (tesla.Date.dt.month == 6)
  ]
)
```

```
        Date       Open   High        Low      Close  Adj Close  \
0 2010-06-29  19.000000  25.00  17.540001  23.889999  23.889999
1 2010-06-30  25.790001  30.42  23.299999  23.830000  23.830000

     Volume
0  18766300
1  17187100
```

```
tesla.index = tesla['Date']
print(tesla.index)
```

```
DatetimeIndex(['2010-06-29', '2010-06-30', '2010-07-01',
               '2010-07-02', '2010-07-06', '2010-07-07',
               '2010-07-08', '2010-07-09', '2010-07-12',
               '2010-07-13',

 ...
 '2017-07-26', '2017-07-27', '2017-07-28',
 '2017-07-31', '2017-08-01', '2017-08-02',
 '2017-08-03', '2017-08-04', '2017-08-07',
 '2017-08-08'],
dtype='datetime64[ns]', name='Date', length=1791, freq=None)
```

```python
print(tesla['2015'])
```

```
                 Date        Open        High         Low  \
Date
2015-01-02 2015-01-02  222.869995  223.250000  213.259995
2015-01-05 2015-01-05  214.550003  216.500000  207.160004
2015-01-06 2015-01-06  210.059998  214.199997  204.210007
2015-01-07 2015-01-07  213.350006  214.779999  209.779999
2015-01-08 2015-01-08  212.809998  213.800003  210.009995
...               ...         ...         ...         ...
2015-12-24 2015-12-24  230.559998  231.880005  228.279999
2015-12-28 2015-12-28  231.490005  231.979996  225.539993
2015-12-29 2015-12-29  230.059998  237.720001  229.550003
2015-12-30 2015-12-30  236.600006  243.630005  235.669998
2015-12-31 2015-12-31  238.509995  243.449997  238.369995

                 Close   Adj Close   Volume
Date
2015-01-02  219.309998  219.309998  4764400
2015-01-05  210.089996  210.089996  5368500
2015-01-06  211.279999  211.279999  6261900
2015-01-07  210.949997  210.949997  2968400
2015-01-08  210.619995  210.619995  3442500
...                ...         ...      ...
2015-12-24  230.570007  230.570007   708000
2015-12-28  228.949997  228.949997  1901300
2015-12-29  237.190002  237.190002  2406300
2015-12-30  238.089996  238.089996  3697900
2015-12-31  240.009995  240.009995  2683200

[252 rows x 7 columns]
```

```python
print(tesla.loc['2015'])
```

```python
print(tesla['2010-06'])
```

```
                  Date       Open   High        Low      Close  \
Date
2010-06-29 2010-06-29  19.000000  25.00  17.540001  23.889999
2010-06-30 2010-06-30  25.790001  30.42  23.299999  23.830000

            Adj Close    Volume
Date
2010-06-29  23.889999  18766300
2010-06-30  23.830000  17187100
```

```python
print(tesla.loc['2010-06'])
```

```python
tesla['ref_date'] = tesla['Date'] - tesla['Date'].min()
```

```python
tesla.index = tesla['ref_date']
```

```python
print(tesla)
```

```
                Date        Open        High         Low  \
ref_date
0 days    2010-06-29   19.000000   25.000000   17.540001
1 days    2010-06-30   25.790001   30.420000   23.299999
2 days    2010-07-01   25.000000   25.920000   20.270000
3 days    2010-07-02   23.000000   23.100000   18.709999
7 days    2010-07-06   20.000000   20.000000   15.830000
...              ...         ...         ...         ...
2591 days 2017-08-02  318.940002  327.119995  311.220001
2592 days 2017-08-03  345.329987  350.000000  343.149994
2593 days 2017-08-04  347.000000  357.269989  343.299988
2596 days 2017-08-07  357.350006  359.480011  352.750000
2597 days 2017-08-08  357.529999  368.579987  357.399994
                Close   Adj Close    Volume  ref_date
ref_date
0 days      23.889999   23.889999  18766300    0 days
1 days      23.830000   23.830000  17187100    1 days
2 days      21.959999   21.959999   8218800    2 days

3 days      19.200001   19.200001   5139800    3 days
7 days      16.110001   16.110001   6866900    7 days
...               ...         ...       ...       ...
2591 days  325.890015  325.890015  13091500 2591 days
2592 days  347.089996  347.089996  13535000 2592 days
2593 days  356.910004  356.910004   9198400 2593 days
2596 days  355.170013  355.170013   6276900 2596 days
2597 days  365.220001  365.220001   7449837 2597 days

[1791 rows x 8 columns]
```

```python
print(tesla['0 day': '10 day'])
```

```
                  Date       Open       High        Low      Close  \
ref_date
0 days   2010-06-29  19.000000  25.000000  17.540001  23.889999
1 days   2010-06-30  25.790001  30.420000  23.299999  23.830000
2 days   2010-07-01  25.000000  25.920000  20.270000  21.959999
3 days   2010-07-02  23.000000  23.100000  18.709999  19.200001
7 days   2010-07-06  20.000000  20.000000  15.830000  16.110001
8 days   2010-07-07  16.400000  16.629999  14.980000  15.800000
9 days   2010-07-08  16.139999  17.520000  15.570000  17.459999
10 days  2010-07-09  17.580000  17.900000  16.549999  17.400000

          Adj Close    Volume ref_date
ref_date
0 days    23.889999  18766300   0 days
1 days    23.830000  17187100   1 days
2 days    21.959999   8218800   2 days
3 days    19.200001   5139800   3 days
7 days    16.110001   6866900   7 days
8 days    15.800000   6921700   8 days
9 days    17.459999   7711400   9 days
10 days   17.400000   4050600  10 days
```

```python
ebola = pd.read_csv(
  'data/country_timeseries.csv', parse_dates=["Date"]
)
```

```python
print(ebola.iloc[:, :5])
```

```
           Date  Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
0    2015-01-05  289        2776.0            NaN            10030.0
1    2015-01-04  288        2775.0            NaN             9780.0
2    2015-01-03  287        2769.0         8166.0             9722.0
3    2015-01-02  286           NaN         8157.0                NaN
4    2014-12-31  284        2730.0         8115.0             9633.0
..          ...  ...           ...            ...                ...
117  2014-03-27    5         103.0            8.0                6.0
118  2014-03-26    4          86.0            NaN                NaN
119  2014-03-25    3          86.0            NaN                NaN
120  2014-03-24    2          86.0            NaN                NaN
121  2014-03-22    0          49.0            NaN                NaN

[122 rows x 5 columns]
```

```
head_range = pd.date_range(start='2014-12-31', end='2015-01-05')
print(head_range)
```

```
DatetimeIndex(['2014-12-31', '2015-01-01', '2015-01-02',
               '2015-01-03', '2015-01-04', '2015-01-05'],
              dtype='datetime64[ns]', freq='D')
```

```
ebola_5.index = ebola_5['Date']
```

```
ebola_5 = ebola_5.reindex(head_range)

print(ebola_5.iloc[:, :5])
```

```
                 Date    Day  Cases_Guinea  Cases_Liberia  \
2014-12-31 2014-12-31  284.0        2730.0         8115.0
2015-01-01        NaT    NaN           NaN            NaN
2015-01-02 2015-01-02  286.0           NaN         8157.0
2015-01-03 2015-01-03  287.0        2769.0         8166.0
2015-01-04 2015-01-04  288.0        2775.0            NaN
2015-01-05 2015-01-05  289.0        2776.0            NaN
```

```
            Cases_SierraLeone
2014-12-31             9633.0
2015-01-01                NaN
2015-01-02                NaN
2015-01-03             9722.0
2015-01-04             9780.0
2015-01-05            10030.0
```

```
# business days during the week of Jan 1, 2022
print(pd.date_range('2022-01-01', '2022-01-07', freq='B'))
```

```
DatetimeIndex(['2022-01-03', '2022-01-04', '2022-01-05',
               '2022-01-06', '2022-01-07'],
              dtype='datetime64[ns]', freq='B')
```

```
# every other business day during the week of Jan 1, 2022
print(pd.date_range('2022-01-01', '2017-01-07', freq='2B'))
```

```
DatetimeIndex([], dtype='datetime64[ns]', freq='2B')
```

```
print(pd.date_range('2022-01-01', '2022-12-31', freq='WOM-1THU'))
```

```
DatetimeIndex(['2022-01-06', '2022-02-03', '2022-03-03',
               '2022-04-07', '2022-05-05', '2022-06-02',
               '2022-07-07', '2022-08-04', '2022-09-01',
               '2022-10-06', '2022-11-03', '2022-12-01'],
              dtype='datetime64[ns]', freq='WOM-1THU')
```

```
print(pd.date_range('2022-01-01', '2022-12-31', freq='WOM-3FRI'))
```

```
DatetimeIndex(['2022-01-21', '2022-02-18', '2022-03-18',
               '2022-04-15', '2022-05-20', '2022-06-17',
               '2022-07-15', '2022-08-19', '2022-09-16',
               '2022-10-21', '2022-11-18', '2022-12-16'],
              dtype='datetime64[ns]', freq='WOM-3FRI')
```

```python
import matplotlib.pyplot as plt

ebola.index = ebola['Date']
```

```python
fig, ax = plt.subplots()
ax = ebola.plot(ax=ax)
ax.legend(fontsize=7, loc=2, borderaxespad=0.0)
plt.show()
```

```python
ebola_sub = ebola[['Day', 'Cases_Guinea', 'Cases_Liberia']]
print(ebola_sub.tail(10))
```

```
            Day  Cases_Guinea  Cases_Liberia
Date
2014-04-04   13         143.0           18.0
2014-04-01   10         127.0            8.0
2014-03-31    9         122.0            8.0
2014-03-29    7         112.0            7.0
2014-03-28    6         112.0            3.0
2014-03-27    5         103.0            8.0
2014-03-26    4          86.0            NaN
2014-03-25    3          86.0            NaN
2014-03-24    2          86.0            NaN
2014-03-22    0          49.0            NaN
```

```python
ebola = pd.read_csv(
  "data/country_timeseries.csv",
  index_col="Date",
  parse_dates=["Date"],
)

print(ebola.iloc[:, :4])
```

```
            Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
Date
2015-01-05  289        2776.0            NaN            10030.0
2015-01-04  288        2775.0            NaN             9780.0
2015-01-03  287        2769.0         8166.0             9722.0
2015-01-02  286           NaN         8157.0                NaN
2014-12-31  284        2730.0         8115.0             9633.0
...         ...           ...            ...                ...
2014-03-27    5         103.0            8.0                6.0
2014-03-26    4          86.0            NaN                NaN
2014-03-25    3          86.0            NaN                NaN
2014-03-24    2          86.0            NaN                NaN
2014-03-22    0          49.0            NaN                NaN

[122 rows x 4 columns]
```

```python
new_idx = pd.date_range(ebola.index.min(), ebola.index.max())

print(new_idx)
```

```
DatetimeIndex(['2014-03-22', '2014-03-23', '2014-03-24',
               '2014-03-25', '2014-03-26', '2014-03-27',
               '2014-03-28', '2014-03-29', '2014-03-30',
               '2014-03-31',
               ...
               '2014-12-27', '2014-12-28', '2014-12-29',
               '2014-12-30', '2014-12-31', '2015-01-01',
               '2015-01-02', '2015-01-03', '2015-01-04',
               '2015-01-05'],
              dtype='datetime64[ns]', length=290, freq='D')
```

```python
new_idx = reversed(new_idx)
print(new_idx)
```

```
<reversed object at 0x105aedfc0>
```

```python
ebola = ebola.reindex(new_idx)
```

```python
print(ebola.iloc[:, :4])
```

```
              Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
Date
2015-01-05  289.0        2776.0            NaN            10030.0
2015-01-04  288.0        2775.0            NaN             9780.0
2015-01-03  287.0        2769.0         8166.0             9722.0
2015-01-02  286.0           NaN         8157.0                NaN
2015-01-01    NaN           NaN            NaN                NaN
...           ...           ...            ...                ...
2014-03-26    4.0          86.0            NaN                NaN
2014-03-25    3.0          86.0            NaN                NaN
2014-03-24    2.0          86.0            NaN                NaN
2014-03-23    NaN           NaN            NaN                NaN
2014-03-22    0.0          49.0            NaN                NaN

[290 rows x 4 columns]
```

```
last_valid = ebola.apply(pd.Series.last_valid_index)
print(last_valid)
```

```
Day                    2014-03-22
Cases_Guinea           2014-03-22
Cases_Liberia          2014-03-27
Cases_SierraLeone      2014-03-27
Cases_Nigeria          2014-07-23
                          ...
Deaths_Nigeria         2014-07-23
Deaths_Senegal         2014-09-07
Deaths_UnitedStates    2014-10-01
Deaths_Spain           2014-10-08
Deaths_Mali            2014-10-22
Length: 17, dtype: datetime64[ns]
```

```
earliest_date = ebola.index.min()
print(earliest_date)
```

```
2014-03-22 00:00:00
```

```
shift_values = last_valid - earliest_date
print(shift_values)
```

```
Day                       0 days
Cases_Guinea              0 days
Cases_Liberia             5 days
Cases_SierraLeone         5 days
Cases_Nigeria           123 days
                          ...
Deaths_Nigeria          123 days
Deaths_Senegal          169 days
Deaths_UnitedStates     193 days
Deaths_Spain            200 days
Deaths_Mali             214 days
Length: 17, dtype: timedelta64[ns]
```

```
ebola_dict = {}

for idx, col in enumerate(ebola):
    d = shift_values[idx].days
    shifted = ebola[col].shift(d)
    ebola_dict[col] = shifted

#print(ebola_dict)
```

```
ebola_shift = pd.DataFrame(ebola_dict)
```

```
print(ebola_shift.tail())
```

```
            Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone  \
Date
2014-03-26  4.0          86.0            8.0                2.0
2014-03-25  3.0          86.0            NaN                NaN
2014-03-24  2.0          86.0            7.0                NaN
2014-03-23  NaN           NaN            3.0                2.0
2014-03-22  0.0          49.0            8.0                6.0
```

```
            Cases_Nigeria  Cases_Senegal  Cases_UnitedStates  \
Date
2014-03-26            1.0            NaN                 1.0
2014-03-25            NaN            NaN                 NaN
2014-03-24            NaN            NaN                 NaN
2014-03-23            NaN            NaN                 NaN
2014-03-22            0.0            1.0                 1.0

            Cases_Spain  Cases_Mali  Deaths_Guinea  Deaths_Liberia  \
Date
2014-03-26          1.0         NaN           62.0             4.0
2014-03-25          NaN         NaN           60.0             NaN
2014-03-24          NaN         NaN           59.0             2.0
2014-03-23          NaN         NaN            NaN             3.0
2014-03-22          1.0         1.0           29.0             6.0

            Deaths_SierraLeone  Deaths_Nigeria  Deaths_Senegal  \
Date
2014-03-26                 2.0             1.0             NaN
2014-03-25                 NaN             NaN             NaN
2014-03-24                 NaN             NaN             NaN
2014-03-23                 2.0             NaN             NaN
2014-03-22                 5.0             0.0             0.0

            Deaths_UnitedStates  Deaths_Spain  Deaths_Mali
Date
2014-03-26                  0.0           1.0          NaN
2014-03-25                  NaN           NaN          NaN
2014-03-24                  NaN           NaN          NaN
2014-03-23                  NaN           NaN          NaN
2014-03-22                  0.0           1.0          1.0
```

```python
ebola_shift.index = ebola_shift['Day']
ebola_shift = ebola_shift.drop(['Day'], axis="columns")
```

```python
print(ebola_shift.tail())
```

```
     Cases_Guinea  Cases_Liberia  Cases_SierraLeone  Cases_Nigeria  \
Day
4.0          86.0            8.0                2.0            1.0
3.0          86.0            NaN                NaN            NaN
2.0          86.0            7.0                NaN            NaN
NaN           NaN            3.0                2.0            NaN
0.0          49.0            8.0                6.0            0.0
```

```
     Cases_Senegal  Cases_UnitedStates  Cases_Spain  Cases_Mali  \
Day
4.0            NaN                 1.0          1.0         NaN
3.0            NaN                 NaN          NaN         NaN
2.0            NaN                 NaN          NaN         NaN
NaN            NaN                 NaN          NaN         NaN
0.0            1.0                 1.0          1.0         1.0

     Deaths_Guinea  Deaths_Liberia  Deaths_SierraLeone  \
Day
4.0           62.0             4.0                 2.0
3.0           60.0             NaN                 NaN
2.0           59.0             2.0                 NaN
NaN            NaN             3.0                 2.0
0.0           29.0             6.0                 5.0

     Deaths_Nigeria  Deaths_Senegal  Deaths_UnitedStates  \
Day
4.0             1.0             NaN                  0.0
3.0             NaN             NaN                  NaN
2.0             NaN             NaN                  NaN
NaN             NaN             NaN                  NaN
0.0             0.0             0.0                  0.0

     Deaths_Spain  Deaths_Mali
Day
4.0           1.0          NaN
3.0           NaN          NaN
2.0           NaN          NaN
NaN           NaN          NaN
0.0           1.0          1.0
```

```python
# downsample daily values to monthly values
# since we have multiple values, we need to aggregate the results
# here we will use the mean
```

```python
down = ebola.resample('M').mean()
```

```python
print(down.iloc[:, :5])
```

```
                  Day  Cases_Guinea  Cases_Liberia  \
Date
2014-03-31    4.500000     94.500000       6.500000
2014-04-30   24.333333    177.818182      24.555556
2014-05-31   51.888889    248.777778      12.555556
2014-06-30   84.636364    373.428571      35.500000
2014-07-31  115.700000    423.000000     212.300000
...                ...           ...            ...
2014-09-30  177.500000    967.888889    2815.625000
2014-10-31  207.470588   1500.444444    4758.750000
2014-11-30  237.214286   1950.500000    7039.000000
2014-12-31  271.181818   2579.625000    7902.571429
2015-01-31  287.500000   2773.333333    8161.500000

            Cases_SierraLeone  Cases_Nigeria
Date
2014-03-31           3.333333            NaN
2014-04-30           2.200000            NaN
2014-05-31           7.333333            NaN
2014-06-30         125.571429            NaN
2014-07-31         420.500000       1.333333
...                       ...            ...
2014-09-30        1726.000000      20.714286
2014-10-31        3668.111111      20.000000
2014-11-30        5843.625000      20.000000
2014-12-31        8985.875000      20.000000
2015-01-31        9844.000000            NaN

[11 rows x 5 columns]
```

```python
# here we will upsample our downsampled value
# notice how missing dates are populated,
# but they are filled in with missing values
up = down.resample('D').mean()
print(up.iloc[:, :5])
```

```
              Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone  \
Date
2014-03-31    4.5     94.500000            6.5           3.333333
2014-04-01    NaN           NaN            NaN                NaN
2014-04-02    NaN           NaN            NaN                NaN
2014-04-03    NaN           NaN            NaN                NaN
2014-04-04    NaN           NaN            NaN                NaN
...           ...           ...            ...                ...
2015-01-27    NaN           NaN            NaN                NaN
2015-01-28    NaN           NaN            NaN                NaN
2015-01-29    NaN           NaN            NaN                NaN
2015-01-30    NaN           NaN            NaN                NaN
2015-01-31  287.5   2773.333333         8161.5        9844.000000

            Cases_Nigeria
Date
2014-03-31            NaN
2014-04-01            NaN
2014-04-02            NaN
2014-04-03            NaN
2014-04-04            NaN
...                   ...
2015-01-27            NaN
2015-01-28            NaN
2015-01-29            NaN
2015-01-30            NaN
2015-01-31            NaN

[307 rows x 5 columns]
```

```
print(len(pytz.all_timezones))
```

```
594
```

```python
import re
regex = re.compile(r'^US')
selected_files = filter(regex.search, pytz.common_timezones)
print(list(selected_files))
```

```
['US/Alaska', 'US/Arizona', 'US/Central', 'US/Eastern', 'US/Hawaii',
'US/Mountain', 'US/Pacific']
```

```python
# 7AM Eastern
depart = pd.Timestamp('2017-08-29 07:00', tz='US/Eastern')
print(depart)
```

```
2017-08-29 07:00:00-04:00
```

```python
arrive = pd.Timestamp('2017-08-29 09:57')
print(arrive)
```

```
2017-08-29 09:57:00
```

```python
arrive = arrive.tz_localize('US/Pacific')
print(arrive)
```

```
2017-08-29 09:57:00-07:00
```

```python
print(arrive.tz_convert('US/Eastern'))
```

```
2017-08-29 12:57:00-04:00
```

```python
import pandas as pd
import seaborn as sns

tips = sns.load_dataset('tips')
print(tips)
```

```
     total_bill   tip     sex smoker   day    time  size
0         16.99  1.01  Female     No   Sun  Dinner     2
1         10.34  1.66    Male     No   Sun  Dinner     3
2         21.01  3.50    Male     No   Sun  Dinner     3
3         23.68  3.31    Male     No   Sun  Dinner     2
4         24.59  3.61  Female     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       29.03  5.92    Male     No   Sat  Dinner     3
240       27.18  2.00  Female    Yes   Sat  Dinner     2
241       22.67  2.00    Male    Yes   Sat  Dinner     2
242       17.82  1.75    Male     No   Sat  Dinner     2
243       18.78  3.00  Female     No  Thur  Dinner     2

[244 rows x 7 columns]
```

```python
import statsmodels.formula.api as smf
```

```python
model = smf.ols(formula='tip ~ total_bill', data=tips)
```

```python
results = model.fit()
```

```python
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    tip   R-squared:                       0.457
Model:                            OLS   Adj. R-squared:                  0.454
Method:                 Least Squares   F-statistic:                     203.4
Date:                Thu, 01 Sep 2022   Prob (F-statistic):           6.69e-34
Time:                        01:55:45   Log-Likelihood:                -350.54
No. Observations:                 244   AIC:                             705.1
Df Residuals:                     242   BIC:                             712.1
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.9203      0.160      5.761      0.000       0.606       1.235
total_bill     0.1050      0.007     14.260      0.000       0.091       0.120
==============================================================================
Omnibus:                       20.185   Durbin-Watson:                   2.151
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               37.750
Skew:                           0.443   Prob(JB):                     6.35e-09
Kurtosis:                       4.711   Cond. No.                         53.0
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
```

```python
print(results.params)
```

```
Intercept     0.920270
total_bill    0.105025
dtype: float64
```

```python
print(results.conf_int())
```

```
                    0         1
Intercept    0.605622  1.234918
total_bill   0.090517  0.119532
```

```python
from sklearn import linear_model
```

```python
# create our LinearRegression object
lr = linear_model.LinearRegression()
```

```python
# note it is an uppercase X
# and a lowercase y
# this will fail because our X has only 1 variable
predicted = lr.fit(X=tips['total_bill'], y=tips['tip'])
```

```
ValueError: Expected 2D array, got 1D array instead:
array=[16.99 10.34 21.01 23.68 24.59 25.29  8.77 26.88 15.04 14.78 10.27 35.26
 15.42 18.43 14.83 21.58 10.33 16.29 16.97 20.65 17.92 20.29 15.77 39.42
 19.82 17.81 13.37 12.69 21.7  19.65  9.55 18.35 15.06 20.69 17.78 24.06
 16.31 16.93 18.69 31.27 16.04 17.46 13.94  9.68 30.4  18.29 22.23 32.4
 28.55 18.04 12.54 10.29 34.81  9.94 25.56 19.49 38.01 26.41 11.24 48.27
 20.29 13.81 11.02 18.29 17.59 20.08 16.45  3.07 20.23 15.01 12.02 17.07
 26.86 25.28 14.73 10.51 17.92 27.2  22.76 17.29 19.44 16.66 10.07 32.68
 15.98 34.83 13.03 18.28 24.71 21.16 28.97 22.49  5.75 16.32 22.75 40.17
 27.28 12.03 21.01 12.46 11.35 15.38 44.3  22.42 20.92 15.36 20.49 25.21
 18.24 14.31 14.    7.25 38.07 23.95 25.71 17.31 29.93 10.65 12.43 24.08
 11.69 13.42 14.26 15.95 12.48 29.8   8.52 14.52 11.38 22.82 19.08 20.27
 11.17 12.26 18.26  8.51 10.33 14.15 16.   13.16 17.47 34.3  41.19 27.05
 16.43  8.35 18.64 11.87  9.78  7.51 14.07 13.13 17.26 24.55 19.77 29.85
 48.17 25.   13.39 16.49 21.5  12.66 16.21 13.81 17.51 24.52 20.76 31.71
 10.59 10.63 50.81 15.81  7.25 31.85 16.82 32.9  17.89 14.48  9.6  34.63
 34.65 23.33 45.35 23.17 40.55 20.69 20.9  30.46 18.15 23.1  15.69 19.81
 28.44 15.48 16.58  7.56 10.34 43.11 13.   13.51 18.71 12.74 13.   16.4
 20.53 16.47 26.59 38.73 24.27 12.76 30.06 25.89 48.33 13.27 28.17 12.9
 28.15 11.59  7.74 30.14 12.16 13.42  8.58 15.98 13.42 16.27 10.09 20.45
 13.28 22.12 24.01 15.69 11.61 10.77 15.53 10.07 12.6  32.83 35.83 29.03
 27.18 22.67 17.82 18.78].
Reshape your data either using array.reshape(-1, 1) if your data has a
single feature or array.reshape(1, -1) if it contains a single sample.
```

```python
# this will fail
predicted = lr.fit(
    X=tips["total_bill"].reshape(-1, 1), y=tips["tip"]
)
```

```
AttributeError: 'Series' object has no attribute 'reshape'
```

```python
# we fix the data by putting it in the correct shape for sklearn
predicted = lr.fit(
    X=tips["total_bill"].values.reshape(-1, 1), y=tips["tip"]
)
```

```python
# note the .fit() method chain at the end
model = smf.ols(formula="tip ~ total_bill + size", data=tips).fit()
```

```python
print(model.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    tip   R-squared:                       0.468
Model:                            OLS   Adj. R-squared:                  0.463
Method:                 Least Squares   F-statistic:                     105.9
Date:                Thu, 01 Sep 2022   Prob (F-statistic):           9.67e-34
Time:                        01:55:46   Log-Likelihood:                -347.99
No. Observations:                 244   AIC:                             702.0
Df Residuals:                     241   BIC:                             712.5
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.6689      0.194      3.455      0.001       0.288       1.050
total_bill     0.0927      0.009     10.172      0.000       0.075       0.111
size           0.1926      0.085      2.258      0.025       0.025       0.361
==============================================================================
Omnibus:                       24.753   Durbin-Watson:                   2.100
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               46.169
Skew:                           0.545   Prob(JB):                     9.43e-11
Kurtosis:                       4.831   Cond. No.                         67.6
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
```

```python
lr = linear_model.LinearRegression()

# since we are performing multiple regression
# we no longer need to reshape our X values
predicted = lr.fit(X=tips[["total_bill", "size"]], y=tips["tip"])

print(predicted.coef_)
```

```
[0.09271334 0.19259779]
```

```python
print(predicted.intercept_)
```

```
0.6689447408125035
```

```python
print(tips.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   total_bill  244 non-null    float64
 1   tip         244 non-null    float64
 2   sex         244 non-null    category
 3   smoker      244 non-null    category
 4   day         244 non-null    category
 5   time        244 non-null    category
 6   size        244 non-null    int64
dtypes: category(4), float64(2), int64(1)
memory usage: 7.4 KB
None
```

```python
print(tips.sex.unique())
```

```
['Female', 'Male']
Categories (2, object): ['Male', 'Female']
```

```python
model = smf.ols(
    formula="tip ~ total_bill + size + sex + smoker + day + time",
    data=tips,
).fit()
```

```
print(model.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    tip   R-squared:                       0.470
Model:                            OLS   Adj. R-squared:                  0.452
Method:                 Least Squares   F-statistic:                     26.06
Date:                Thu, 01 Sep 2022   Prob (F-statistic):           1.20e-28
Time:                        01:55:46   Log-Likelihood:                -347.48
No. Observations:                 244   AIC:                             713.0
Df Residuals:                     235   BIC:                             744.4
Df Model:                           8
Covariance Type:            nonrobust
==============================================================================
                  coef    std err          t      P>|t|      [0.025    0.975]
------------------------------------------------------------------------------
Intercept       0.5908      0.256      2.310      0.022       0.087     1.095
sex[T.Female]   0.0324      0.142      0.229      0.819      -0.247     0.311
smoker[T.No]    0.0864      0.147      0.589      0.556      -0.202     0.375
day[T.Fri]      0.1623      0.393      0.412      0.680      -0.613     0.937
day[T.Sat]      0.0408      0.471      0.087      0.931      -0.886     0.968
day[T.Sun]      0.1368      0.472      0.290      0.772      -0.793     1.066
time[T.Dinner] -0.0681      0.445     -0.153      0.878      -0.944     0.808
total_bill      0.0945      0.010      9.841      0.000       0.076     0.113
size            0.1760      0.090      1.966      0.051      -0.000     0.352
==============================================================================
Omnibus:                       27.860   Durbin-Watson:                   2.096
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               52.555
Skew:                           0.607   Prob(JB):                     3.87e-12
Kurtosis:                       4.923   Cond. No.                         281.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors
is correctly specified.
```

```python
print(tips.day.unique())
```

```
['Sun', 'Sat', 'Thur', 'Fri']
Categories (4, object): ['Thur', 'Fri', 'Sat', 'Sun']
```

```python
tips_dummy = pd.get_dummies(
    tips[["total_bill", "size", "sex", "smoker", "day", "time"]]
)

print(tips_dummy)
```

```
     total_bill  size  sex_Male  sex_Female  smoker_Yes  smoker_No  \
0         16.99     2         0           1           0          1
1         10.34     3         1           0           0          1
2         21.01     3         1           0           0          1
3         23.68     2         1           0           0          1
4         24.59     4         0           1           0          1
..          ...   ...       ...         ...         ...        ...
239       29.03     3         1           0           0          1
240       27.18     2         0           1           1          0
241       22.67     2         1           0           1          0
242       17.82     2         1           0           0          1
243       18.78     2         0           1           0          1

     day_Thur  day_Fri  day_Sat  day_Sun  time_Lunch  time_Dinner
0           0        0        0        1           0            1
1           0        0        0        1           0            1
2           0        0        0        1           0            1
3           0        0        0        1           0            1
4           0        0        0        1           0            1
```

..	...	...	...	...	...	...
239	0	0	1	0	0	1
240	0	0	1	0	0	1
241	0	0	1	0	0	1
242	0	0	1	0	0	1
243	1	0	0	0	0	1

[244 rows x 12 columns]

```python
x_tips_dummy_ref = pd.get_dummies(
    tips[["total_bill", "size", "sex", "smoker", "day", "time"]],
    drop_first=True,
)

print(x_tips_dummy_ref)
```

```
     total_bill  size  sex_Female  smoker_No  day_Fri  day_Sat  \
0         16.99     2           1          1        0        0
1         10.34     3           0          1        0        0
2         21.01     3           0          1        0        0
3         23.68     2           0          1        0        0
4         24.59     4           1          1        0        0
..          ...   ...         ...        ...      ...      ...
239       29.03     3           0          1        0        1
240       27.18     2           1          0        0        1
241       22.67     2           0          0        0        1
242       17.82     2           0          1        0        1
243       18.78     2           1          1        0        0

     day_Sun  time_Dinner
0          1            1
1          1            1
2          1            1
3          1            1
4          1            1
..       ...          ...
239        0            1
240        0            1
241        0            1
242        0            1
243        0            1

[244 rows x 8 columns]
```

```python
lr = linear_model.LinearRegression()
predicted = lr.fit(X=x_tips_dummy_ref, y=tips["tip"])
```

```python
print(predicted.intercept_)
```

```
0.5908374259513787
```

```python
print(predicted.coef_)
```

```
[ 0.09448701  0.175992    0.03244094  0.08640832  0.1622592   0.04080082
  0.13677854 -0.0681286 ]
```

```python
import numpy as np

# create and fit the model
lr = linear_model.LinearRegression()
predicted = lr.fit(X=x_tips_dummy_ref, y=tips["tip"])

# get the intercept along with other coefficients
values = np.append(predicted.intercept_, predicted.coef_)

# get the names of the values
names = np.append("intercept", x_tips_dummy_ref.columns)

# put everything in a labeled dataframe
results = pd.DataFrame({"variable": names, "coef": values})

print(results)
```

```
      variable      coef
0    intercept  0.590837
1   total_bill  0.094487
2         size  0.175992
3   sex_Female  0.032441
4    smoker_No  0.086408
5      day_Fri  0.162259
6      day_Sat  0.040801
7      day_Sun  0.136779
8  time_Dinner -0.068129
```

```python
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
```

```python
categorical_features = ["sex", "smoker", "day", "time"]
categorical_transformer = OneHotEncoder(drop="first")
```

```python
preprocessor = ColumnTransformer(
  transformers=[
    ("cat", categorical_transformer, categorical_features),
  ],
  remainder="passthrough",  # keep the numeric columns
)
```

```python
pipe = Pipeline(
  steps=[
    ("preprocessor", preprocessor),
    ("lr", linear_model.LinearRegression()),
  ]
)
```

```python
pipe.fit(
  X=tips[["total_bill", "size", "sex", "smoker", "day", "time"]],
  y=tips["tip"],
)
```

```
Pipeline(steps=[('preprocessor',
                 ColumnTransformer(remainder='passthrough',
                                   transformers=[('cat',
                                                  OneHotEncoder(drop='first'),
                                                  ['sex', 'smoker', 'day',
                                                   'time'])])),
                ('lr', LinearRegression())])
```

```python
print(type(pipe))
```

```
<class 'sklearn.pipeline.Pipeline'>
```

```python
# combine the intercept and coefficients into single vector
coefficients = np.append(
  pipe.named_steps["lr"].intercept_, pipe.named_steps["lr"].coef_
)

# combine the intercept text with the other feature names
labels = np.append(
  ["intercept"], pipe[:-1].get_feature_names_out()
)

# create a dataframe of all the results
coefs = pd.DataFrame({"variable": labels, "coef": coefficients})

print(coefs)
```

```
                variable      coef
0              intercept  0.803817
1          cat__sex_Male -0.032441
2        cat__smoker_Yes -0.086408
3           cat__day_Sat -0.121458
4           cat__day_Sun -0.025481
5          cat__day_Thur -0.162259
6        cat__time_Lunch  0.068129
7  remainder__total_bill  0.094487
8        remainder__size  0.175992
```

```python
import seaborn as sns

titanic = sns.load_data set("titanic")
titanic.to_csv("data/titanic.csv", index=False)
```

```python
titanic_sub = (
    titanic[["survived", "sex", "age", "embarked"]].copy().dropna()
)

print(titanic_sub)
```

```
     survived     sex   age embarked
0           0    male  22.0        S
1           1  female  38.0        C
2           1  female  26.0        S
3           1  female  35.0        S
4           0    male  35.0        S
..        ...     ...   ...      ...
885         0  female  39.0        Q
886         0    male  27.0        S
887         1  female  19.0        S
889         1    male  26.0        C
890         0    male  32.0        Q

[712 rows x 4 columns]
```

```python
# count of values in the survived column
print(titanic_sub["survived"].value_counts())
```

```
0    424
1    288
Name: survived, dtype: int64
```

```python
# count of values in the embarked column
print(titanic_sub["embarked"].value_counts())
```

```
S    554
C    130
Q     28
Name: embarked, dtype: int64
```

```python
import statsmodels.formula.api as smf

# formula for the model
form = 'survived ~ sex + age + embarked'

# fitting the logistic regression model, note the .fit() at the end
py_logistic_smf = smf.logit(formula=form, data=titanic_sub).fit()

print(py_logistic_smf.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.509889
         Iterations 6
                           Logit Regression Results
==============================================================================
Dep. Variable:               survived   No. Observations:                  712
Model:                          Logit   Df Residuals:                      707
Method:                           MLE   Df Model:                            4
Date:                Thu, 01 Sep 2022   Pseudo R-squ.:                  0.2444
Time:                        01:55:49   Log-Likelihood:                -363.04

converged:                       True   LL-Null:                       -480.45
Covariance Type:            nonrobust   LLR p-value:                 1.209e-49
================================================================================
                   coef    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------------
Intercept        2.2046      0.322      6.851      0.000       1.574       2.835
sex[T.male]     -2.4760      0.191    -12.976      0.000      -2.850      -2.102
embarked[T.Q]   -1.8156      0.535     -3.393      0.001      -2.864      -0.767
embarked[T.S]   -1.0069      0.237     -4.251      0.000      -1.471      -0.543
age             -0.0081      0.007     -1.233      0.217      -0.021       0.005
================================================================================
```

```python
import numpy as np

# get the coefficients into a dataframe
res_sm = pd.DataFrame(py_logistic_smf.params, columns=["coefs_sm"])

# calculate the odds
res_sm["odds_sm"] = np.exp(res_sm["coefs_sm"])

# round the decimals
print(res_sm.round(3))
```

```
                coefs_sm  odds_sm
Intercept          2.205    9.066
sex[T.male]       -2.476    0.084
embarked[T.Q]     -1.816    0.163
embarked[T.S]     -1.007    0.365
age               -0.008    0.992
```

```python
titanic_dummy = pd.get_dummies(
    titanic_sub[["survived", "sex", "age", "embarked"]],
    drop_first=True
)
```

```python
# note our outcome variable is the first column (index 0)
print(titanic_dummy)
```

```
     survived   age  sex_male  embarked_Q  embarked_S
0           0  22.0         1           0           1
1           1  38.0         0           0           0
2           1  26.0         0           0           1
3           1  35.0         0           0           1
4           0  35.0         1           0           1
..        ...   ...       ...         ...         ...
885         0  39.0         0           1           0
886         0  27.0         1           0           1
887         1  19.0         0           0           1
889         1  26.0         1           0           0
890         0  32.0         1           1           0

[712 rows x 5 columns]
```

```python
from sklearn import linear_model

# this is the only part that fits the model
py_logistic_sklearn1 = (
  linear_model.LogisticRegression().fit(
    X=titanic_dummy.iloc[:, 1:],  # all the columns except first
    y=titanic_dummy.iloc[:, 0]    # just the first column
  )
)
```

```python
# get the names of the dummy variable columns
dummy_names = titanic_dummy.columns.to_list()

# get the intercept and coefficients into a dataframe
sk1_res1 = pd.DataFrame(
    py_logistic_sklearn1.intercept_,
    index=["Intercept"],
    columns=["coef_sk1"],
)
sk1_res2 = pd.DataFrame(
    py_logistic_sklearn1.coef_.T,
    index=dummy_names[1:],
    columns=["coef_sk1"],
)

# put the results into a single dataframe to show the results
res_sklearn_pd_1 = pd.concat([sk1_res1, sk1_res2])

# calculate the odds
res_sklearn_pd_1["odds_sk1"] = np.exp(res_sklearn_pd_1["coef_sk1"])

print(res_sklearn_pd_1.round(3))
```

```
            coef_sk1  odds_sk1
Intercept      2.024     7.571
age           -0.008     0.992
sex_male      -2.372     0.093
embarked_Q    -1.369     0.254
embarked_S    -0.887     0.412
```

```python
# fit another logistic regression with no penalty
py_logistic_sklearn2 = linear_model.LogisticRegression(
    penalty="none"  # this parameter is important!
).fit(
    X=titanic_dummy.iloc[:, 1:],  # all the columns except first
    y=titanic_dummy.iloc[:, 0]    # just the first column
)
# rest of the code is the same as before, except variable names
sk2_res1 = pd.DataFrame(
    py_logistic_sklearn2.intercept_,
    index=["Intercept"],
    columns=["coef_sk2"],
)
sk2_res2 = pd.DataFrame(
    py_logistic_sklearn2.coef_.T,
    index=dummy_names[1:],
    columns=["coef_sk2"],
)

res_sklearn_pd_2 = pd.concat([sk2_res1, sk2_res2])
res_sklearn_pd_2["odds_sk2"] = np.exp(res_sklearn_pd_2["coef_sk2"])
```

```python
sm_results = res_sm.round(3)

# sort values to make things easier to compare
sm_results = sm_results.sort_index()

print(sm_results)
```

```
               coefs_sm  odds_sm
Intercept         2.205    9.066
age              -0.008    0.992
embarked[T.Q]    -1.816    0.163
embarked[T.S]    -1.007    0.365
sex[T.male]      -2.476    0.084
```

```python
# concatenate the 2 model results
sk_results = pd.concat(
  [res_sklearn_pd_1.round(3), res_sklearn_pd_2.round(3)],
  axis="columns",
)

# sort cols and rows to make things easy to compare
sk_results = sk_results[sk_results.columns.sort_values()]
sk_results = sk_results.sort_index()

print(sk_results)
```

```
           coef_sk1  coef_sk2  odds_sk1  odds_sk2
Intercept     2.024     2.205     7.571     9.066
age          -0.008    -0.008     0.992     0.992
embarked_Q   -1.369    -1.816     0.254     0.163
embarked_S   -0.887    -1.007     0.412     0.365
sex_male     -2.372    -2.476     0.093     0.084
```

```python
acs = pd.read_csv('data/acs_ny.csv')
print(acs.columns)
```

```
Index(['Acres', 'FamilyIncome', 'FamilyType', 'NumBedrooms',
       'NumChildren', 'NumPeople', 'NumRooms', 'NumUnits',
       'NumVehicles', 'NumWorkers', 'OwnRent', 'YearBuilt',
       'HouseCosts', 'ElectricBill', 'FoodStamp', 'HeatingFuel',
       'Insurance', 'Language'],
      dtype='object')
```

```python
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
sns.countplot(data = acs, x = "NumBedrooms", ax=ax)

ax.set_title('Number of Bedrooms')
ax.set_xlabel('Number of Bedrooms in a House')
ax.set_ylabel('Count')

plt.show()
```

```python
model = smf.poisson(
  "NumBedrooms ~ HouseCosts + OwnRent", data=acs
)
results = model.fit()

print(results.summary())
```

```
Optimization terminated successfully.
         Current function value: 1.680998
         Iterations 10
```

```
                          Poisson Regression Results
==============================================================================
Dep. Variable:            NumBedrooms   No. Observations:                22745
Model:                        Poisson   Df Residuals:                    22741
Method:                           MLE   Df Model:                            3
Date:                Thu, 01 Sep 2022   Pseudo R-squ.:                0.008309
Time:                        01:55:49   Log-Likelihood:                -38234.
converged:                       True   LL-Null:                       -38555.
Covariance Type:            nonrobust   LLR p-value:                 1.512e-138
=======================================================================================
                          coef    std err          z      P>|z|      [0.025      0.975]
---------------------------------------------------------------------------------------
Intercept               1.1387      0.006    184.928      0.000       1.127       1.151
OwnRent[T.Outright]    -0.2659      0.051     -5.182      0.000      -0.367      -0.165
OwnRent[T.Rented]      -0.1237      0.012     -9.996      0.000      -0.148      -0.099
HouseCosts           6.217e-05   2.96e-06     21.017      0.000    5.64e-05     6.8e-05
=======================================================================================
```

```python
import statsmodels.api as sm
import statsmodels.formula.api as smf

model = smf.glm(
    "NumBedrooms ~ HouseCosts + OwnRent",
    data=acs,
    family=sm.families.Poisson(sm.genmod.families.links.log()),
).fit()
```

```python
print(results.summary())
```

```
                 Poisson Regression Results
==============================================================================
Dep. Variable:           NumBedrooms   No. Observations:                22745
Model:                       Poisson   Df Residuals:                    22741
Method:                          MLE   Df Model:                            3
Date:               Thu, 01 Sep 2022   Pseudo R-squ.:                0.008309
Time:                       01:55:49   Log-Likelihood:                -38234.
converged:                      True   LL-Null:                       -38555.
Covariance Type:           nonrobust   LLR p-value:                1.512e-138
=======================================================================================
                          coef    std err          z      P>|z|      [0.025      0.975]
---------------------------------------------------------------------------------------
Intercept               1.1387      0.006    184.928      0.000       1.127       1.151
OwnRent[T.Outright]    -0.2659      0.051     -5.182      0.000      -0.367      -0.165
OwnRent[T.Rented]      -0.1237      0.012     -9.996      0.000      -0.148      -0.099
HouseCosts           6.217e-05   2.96e-06     21.017      0.000    5.64e-05     6.8e-05
=======================================================================================
```

```python
fig, ax = plt.subplots()

sns.countplot(data = acs, x = "NumPeople", ax=ax)
ax.set_title('Number of People')
ax.set_xlabel('Number of People in a Household')
ax.set_ylabel('Count')

plt.show()
```

```python
model = smf.glm(
  "NumPeople ~ Acres + NumVehicles",
  data=acs,
  family=sm.families.NegativeBinomial(
    sm.genmod.families.links.log()
  ),
)

results = model.fit()
```

```python
print(results.summary())
```

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:              NumPeople   No. Observations:                22745
Model:                            GLM   Df Residuals:                    22741
Model Family:        NegativeBinomial   Df Model:                            3
Link Function:                    log   Scale:                          1.0000
Method:                          IRLS   Log-Likelihood:                -53542.
Date:                Thu, 01 Sep 2022   Deviance:                       2605.6
Time:                        01:55:50   Pearson chi2:                 2.99e+03
No. Iterations:                     6   Pseudo R-squ. (CS):           0.003504
Covariance Type:            nonrobust
==================================================================================
                     coef    std err          z      P>|z|      [0.025      0.975]
----------------------------------------------------------------------------------
Intercept          1.0418      0.025     41.580      0.000       0.993       1.091
Acres[T.10+]      -0.0225      0.040     -0.564      0.573      -0.101       0.056
Acres[T.Sub 1]     0.0509      0.019      2.671      0.008       0.014       0.088
NumVehicles        0.0661      0.008      8.423      0.000       0.051       0.081
==================================================================================
```

```python
print(acs["Acres"].value_counts())
```

```
Sub 1     17114
1-10       4627
10+        1004
Name: Acres, dtype: int64
```

```python
sm.families.family.Binomial.links
```

```
[statsmodels.genmod.families.links.Logit,
 statsmodels.genmod.families.links.probit,
 statsmodels.genmod.families.links.cauchy,
 statsmodels.genmod.families.links.Log,
 statsmodels.genmod.families.links.CLogLog,
 statsmodels.genmod.families.links.LogLog,
 statsmodels.genmod.families.links.identity]
```

```python
bladder = pd.read_csv('data/bladder.csv')

print(bladder)
```

```
     id  rx  number  size  stop  event  enum
0     1   1       1     3     1      0     1
1     1   1       1     3     1      0     2
2     1   1       1     3     1      0     3
3     1   1       1     3     1      0     4
4     2   1       2     1     4      0     1
..   ..  ..     ...   ...   ...    ...   ...
335  84   2       2     1    54      0     4
336  85   2       1     3    59      0     1
337  85   2       1     3    59      0     2
338  85   2       1     3    59      0     3
339  85   2       1     3    59      0     4

[340 rows x 7 columns]
```

```python
print(bladder['rx'].value_counts())
```

```
1    188
2    152
Name: rx, dtype: int64
```

```python
from lifelines import KaplanMeierFitter
```

```python
kmf = KaplanMeierFitter()
kmf.fit(bladder['stop'], event_observed=bladder['event'])
```

```
<lifelines.KaplanMeierFitter:"KM_estimate", fitted with 340 total
observations,  228 right-censored observations>
```

```python
import matplotlib.pyplot as plt
```

```python
fig, ax = plt.subplots()
kmf.survival_function_.plot(ax=ax)
ax.set_title('Survival function of cancer recurrence')
plt.show()
```

```python
fig, ax = plt.subplots()
kmf.plot(ax=ax)
ax.set_title('Survival with confidence intervals')
plt.show()
```

```python
from lifelines import CoxPHFitter

cph = CoxPHFitter()
```

```python
cph_bladder_df = bladder[
    ["rx", "number", "size", "enum", "stop", "event"]
]
cph.fit(cph_bladder_df, duration_col="stop", event_col="event")
```

```
<lifelines.CoxPHFitter: fitted with 340 total observations, 228
right-censored observations>
```

```python
cph.print_summary()
```

```
rx1 = bladder.loc[bladder['rx'] == 1]
rx2 = bladder.loc[bladder['rx'] == 2]

kmf1 = KaplanMeierFitter()
kmf1.fit(rx1['stop'], event_observed=rx1['event'])

kmf2 = KaplanMeierFitter()
kmf2.fit(rx2['stop'], event_observed=rx2['event'])

fig, axes = plt.subplots()

# put both plots on the same axes
kmf1.plot_loglogs(ax=axes)
kmf2.plot_loglogs(ax=axes)
```

```
axes.legend(['rx1', 'rx2'])

plt.show()
```

```
cph_strat = CoxPHFitter()
cph_strat.fit(
    cph_bladder_df,
    duration_col="stop",
    event_col="event",
    strata=["rx"],
)

cph_strat.print_summary()
```

```python
import pandas as pd
housing = pd.read_csv('data/housing_renamed.csv')

print(housing.head())
```

```
  neighborhood            type  units  year_built   sq_ft    income  \
0    FINANCIAL  R9-CONDOMINIUM     42      1920.0   36500   1332615
1    FINANCIAL  R4-CONDOMINIUM     78      1985.0  126420   6633257
2    FINANCIAL  RR-CONDOMINIUM    500         NaN  554174  17310000
3    FINANCIAL  R4-CONDOMINIUM    282      1930.0  249076  11776313
4      TRIBECA  R4-CONDOMINIUM    239      1985.0  219495  10004582

   income_per_sq_ft  expense  expense_per_sq_ft  net_income  \
0             36.51   342005               9.37      990610
1             52.47  1762295              13.94     4870962
2             31.24  3543000               6.39    13767000
3             47.28  2784670              11.18     8991643
4             45.58  2783197              12.68     7221385

      value  value_per_sq_ft       boro
0   7300000           200.00  Manhattan
1  30690000           242.76  Manhattan
2  90970000           164.15  Manhattan
3  67556006           271.23  Manhattan
4  54320996           247.48  Manhattan
```

```python
import statsmodels
import statsmodels.api as sm
import statsmodels.formula.api as smf

house1 = smf.glm(
    "value_per_sq_ft ~ units + sq_ft + boro", data=housing
).fit()

print(house1.summary())
```

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:        value_per_sq_ft   No. Observations:                 2626
Model:                            GLM   Df Residuals:                     2619
Model Family:                Gaussian   Df Model:                            6
Link Function:               identity   Scale:                          1879.5
Method:                          IRLS   Log-Likelihood:                -13621.
Date:                Thu, 01 Sep 2022   Deviance:                   4.9224e+06
Time:                        01:55:55   Pearson chi2:                 4.92e+06
No. Iterations:                     3   Pseudo R-squ. (CS):             0.7772
Covariance Type:            nonrobust
=========================================================================================
                            coef    std err          z      P>|z|      [0.025      0.975]
-----------------------------------------------------------------------------------------
Intercept                43.2909      5.330      8.122      0.000      32.845      53.737
boro[T.Brooklyn]         34.5621      5.535      6.244      0.000      23.714      45.411
boro[T.Manhattan]       130.9924      5.385     24.327      0.000     120.439     141.546
boro[T.Queens]           32.9937      5.663      5.827      0.000      21.895      44.092
boro[T.Staten Island]    -3.6303      9.993     -0.363      0.716     -23.216      15.956
units                    -0.1881      0.022     -8.511      0.000      -0.231      -0.145
sq_ft                     0.0002   2.09e-05     10.079      0.000       0.000       0.000
=========================================================================================
```

```python
import seaborn as sns
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
sns.scatterplot(
  x=house1.fittedvalues, y=house1.resid_deviance, ax=ax
)

plt.show()
```

```python
# get the data used for the residual plot and boro color
res_df = pd.DataFrame(
  {
    "fittedvalues": house1.fittedvalues, # get a model attribute
    "resid_deviance": house1.resid_deviance,
    "boro": housing["boro"], # get a value from data column
  }
)
```

```python
# greyscale friendly color palette
color_dict = dict(
  {
    "Manhattan": "#d7191c",
    "Brooklyn": "#fdae61",
    "Queens": "#ffffbf",
    "Bronx": "#abdda4",
    "Staten Island": "#2b83ba",
  }
)

fig, ax = plt.subplots()
fig = sns.scatterplot(
  x="fittedvalues",
  y="resid_deviance",
  data=res_df,
  hue="boro",
  ax=ax,
  palette=color_dict,
  edgecolor='black',
)

plt.show()
```

```python
from scipy import stats

# make a copy of the variable so we don't need to keep typing it
resid = house1.resid_deviance.copy()
```

```python
fig = statsmodels.graphics.gofplots.qqplot(resid, line='r')
plt.show()
```

```python
resid_std = stats.zscore(resid)

fig, ax = plt.subplots()
sns.histplot(resid_std, ax=ax)
plt.show()
```

```python
f1 = 'value_per_sq_ft ~ units + sq_ft + boro'
f2 = 'value_per_sq_ft ~ units *  sq_ft + boro'
f3 = 'value_per_sq_ft ~ units + sq_ft *  boro + type'
f4 = 'value_per_sq_ft ~ units + sq_ft *  boro + sq_ft *  type'
f5 = 'value_per_sq_ft ~ boro + type'

house1 = smf.ols(f1, data=housing).fit()
house2 = smf.ols(f2, data=housing).fit()
house3 = smf.ols(f3, data=housing).fit()
house4 = smf.ols(f4, data=housing).fit()
house5 = smf.ols(f5, data=housing).fit()
```

```python
mod_results = (
  pd.concat(
    [
      house1.params,
      house2.params,
      house3.params,
      house4.params,
      house5.params,
    ],
    axis=1,
  )
  .rename(columns=lambda x: "house" + str(x + 1))
  .reset_index()
  .rename(columns={"index": "param"})
  .melt(id_vars="param", var_name="model", value_name="estimate")
)
```

```python
print(mod_results)
```

```
                                param   model    estimate
0                           Intercept  house1   43.290863
1                  boro[T.Brooklyn]  house1   34.562150
2                 boro[T.Manhattan]  house1  130.992363
3                    boro[T.Queens]  house1   32.993674
4             boro[T.Staten Island]  house1   -3.630251
..                                ...     ...         ...
85            sq_ft:boro[T.Queens]  house5         NaN
86     sq_ft:boro[T.Staten Island]  house5         NaN
87  sq_ft:type[T.R4-CONDOMINIUM]  house5         NaN
88  sq_ft:type[T.R9-CONDOMINIUM]  house5         NaN
89  sq_ft:type[T.RR-CONDOMINIUM]  house5         NaN

[90 rows x 3 columns]
```

```python
color_dict = dict(
  {
    "house1": "#d7191c",
    "house2": "#fdae61",
    "house3": "#ffffbf",
    "house4": "#abdda4",
    "house5": "#2b83ba",
  }
)
```

```python
fig, ax = plt.subplots()
ax = sns.pointplot(
  x="estimate",
  y="param",
  hue="model",
  data=mod_results,
  dodge=True,  # jitter the points
  join=False,  # don't connect the points
  palette=color_dict

)

plt.tight_layout()
plt.show()
```

```python
model_names = ["house1", "house2", "house3", "house4", "house5"]
house_anova = statsmodels.stats.anova.anova_lm(
  house1, house2, house3, house4, house5
)

house_anova.index = model_names

print(house_anova)
```

```
        df_resid           ssr  df_diff        ss_diff          F  \
house1    2619.0  4.922389e+06      0.0            NaN        NaN
house2    2618.0  4.884872e+06      1.0   37517.437605  20.039049
house3    2612.0  4.619926e+06      6.0  264945.539994  23.585728
house4    2609.0  4.576671e+06      3.0   43255.441192   7.701289
house5    2618.0  4.901463e+06     -9.0 -324791.847907  19.275539

              Pr(>F)
house1           NaN
house2  7.912333e-06
house3  2.754431e-27
house4  4.025581e-05
house5           NaN
```

```
house_models = [house1, house2, house3, house4, house5]
```

```
abic = pd.DataFrame(
  {
    "model": model_names,
    "aic": [mod.aic for mod in house_models],
    "bic": [mod.bic for mod in house_models],
  }
)

print(abic.sort_values(by=["aic", "bic"]))
```

```
    model           aic           bic
3  house4  27084.800043  27184.644733
2  house3  27103.502577  27185.727615
1  house2  27237.939618  27284.925354
4  house5  27246.843392  27293.829128
0  house1  27256.031113  27297.143632
```

```python
def deviance_table(*models):
  """Create a table of model diagnostics from model objects"""
  return pd.DataFrame(
    {
      "df_residuals": [mod.df_resid for mod in models],
```

```python
      "resid_stddev": [mod.deviance for mod in models],
      "df": [mod.df_model for mod in models],
      "deviance": [mod.deviance for mod in models],
    }
  )
```

```python
f1 = 'value_per_sq_ft ~ units + sq_ft + boro'
f2 = 'value_per_sq_ft ~ units * sq_ft + boro'
f3 = 'value_per_sq_ft ~ units + sq_ft * boro + type'
f4 = 'value_per_sq_ft ~ units + sq_ft * boro + sq_ft * type'
f5 = 'value_per_sq_ft ~ boro + type'

glm1 = smf.glm(f1, data=housing).fit()
glm2 = smf.glm(f2, data=housing).fit()
glm3 = smf.glm(f3, data=housing).fit()
glm4 = smf.glm(f4, data=housing).fit()
glm5 = smf.glm(f5, data=housing).fit()

glm_anova = deviance_table(glm1, glm2, glm3, glm4, glm5)
print(glm_anova)
```

```
   df_residuals  resid_stddev  df      deviance
0          2619  4.922389e+06   6  4.922389e+06
1          2618  4.884872e+06   7  4.884872e+06
2          2612  4.619926e+06  13  4.619926e+06
3          2609  4.576671e+06  16  4.576671e+06
4          2618  4.901463e+06   7  4.901463e+06
```

```python
# create a binary variable
housing["high"] = (housing["value_per_sq_ft"] >= 150).astype(int)

print(housing["high"].value_counts())
```

```
0    1619
1    1007
Name: high, dtype: int64
```

```python
# create and fit our logistic regression using GLM

f1 = "high ~ units + sq_ft + boro"
f2 = "high ~ units * sq_ft + boro"
f3 = "high ~ units + sq_ft * boro + type"
f4 = "high ~ units + sq_ft * boro + sq_ft * type"
f5 = "high ~ boro + type"
```

```python
logistic = statsmodels.genmod.families.family.Binomial(
    link=statsmodels.genmod.families.links.Logit()
)

glm1 = smf.glm(f1, data=housing, family=logistic).fit()
glm2 = smf.glm(f2, data=housing, family=logistic).fit()
glm3 = smf.glm(f3, data=housing, family=logistic).fit()
glm4 = smf.glm(f4, data=housing, family=logistic).fit()
glm5 = smf.glm(f5, data=housing, family=logistic).fit()
```

```python
# show the deviances from our GLM models
print(deviance_table(glm1, glm2, glm3, glm4, glm5))
```

```
   df_residuals  resid_stddev  df     deviance
0          2619   1695.631547   6  1695.631547
1          2618   1686.126740   7  1686.126740
2          2612   1636.492830  13  1636.492830
3          2609   1619.431515  16  1619.431515
4          2618   1666.615696   7  1666.615696
```

```python
mods = [glm1, glm2, glm3, glm4, glm5]

abic_glm = pd.DataFrame(
  {
    "model": model_names,
    "aic": [mod.aic for mod in house_models],
    "bic": [mod.bic for mod in house_models],
  }
)

print(abic_glm.sort_values(by=["aic", "bic"]))
```

```
    model           aic           bic
3  house4  27084.800043  27184.644733
2  house3  27103.502577  27185.727615
1  house2  27237.939618  27284.925354
4  house5  27246.843392  27293.829128
0  house1  27256.031113  27297.143632
```

```python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

print(housing.columns)
```

```
Index(['neighborhood', 'type', 'units', 'year_built', 'sq_ft',
       'income', 'income_per_sq_ft', 'expense', 'expense_per_sq_ft',
       'net_income', 'value', 'value_per_sq_ft', 'boro', 'high'],
      dtype='object')
```

```python
# get training and test data
X_train, X_test, y_train, y_test = train_test_split(
    pd.get_dummies(
        housing[["units", "sq_ft", "boro"]], drop_first=True
    ),

    housing["value_per_sq_ft"],
    test_size=0.20,
    random_state=42,
)
```

```python
lr = LinearRegression().fit(X_train, y_train)
print(lr.score(X_test, y_test))
```

```
0.6137125285030868
```

```python
from patsy import dmatrices

y, X = dmatrices(
  "value_per_sq_ft ~ units + sq_ft + boro",
  housing,
  return_type="dataframe",
)
X_train, X_test, y_train, y_test = train_test_split(
  X, y, test_size=0.20, random_state=42
)

lr = LinearRegression().fit(X_train, y_train)
print(lr.score(X_test, y_test))
```

```
0.6137125285030818
```

```python
from sklearn.model_selection import KFold, cross_val_score

# get a fresh new housing data set
housing = pd.read_csv('data/housing_renamed.csv')
```

```python
kf = KFold(n_splits=5)

y, X = dmatrices('value_per_sq_ft ~ units + sq_ft + boro', housing)
```

```python
coefs = []
scores = []
for train, test in kf.split(X):
  X_train, X_test = X[train], X[test]
  y_train, y_test = y[train], y[test]
  lr = LinearRegression().fit(X_train, y_train)
  coefs.append(pd.DataFrame(lr.coef_))
  scores.append(lr.score(X_test, y_test))
```

```python
coefs_df = pd.concat(coefs)
coefs_df.columns = X.design_info.column_names
print(coefs_df)
```

```
   Intercept  boro[T.Brooklyn]  boro[T.Manhattan]  boro[T.Queens]  \
0        0.0         33.369037         129.904011       32.103100
0        0.0         32.889925         116.957385       31.295956
0        0.0         30.975560         141.859327       32.043449
0        0.0         41.449196         130.779013       33.050968
0        0.0        -38.511915          56.069855      -17.557939

   boro[T.Staten Island]     units     sq_ft
0          -4.381085e+00 -0.205890  0.000220
0          -4.919232e+00 -0.146180  0.000155
0          -4.379916e+00 -0.179671  0.000194
0          -3.430209e+00 -0.207904  0.000232
0           3.552714e-15 -0.145829  0.000202
```

```
import numpy as np
print(coefs_df.apply(np.mean))
```

```
Intercept                  0.000000
boro[T.Brooklyn]          20.034361
boro[T.Manhattan]        115.113918
boro[T.Queens]            22.187107
boro[T.Staten Island]     -3.422088
units                     -0.177095
sq_ft                      0.000201
dtype: float64
```

```
print(scores)
```

```
[0.02731416291043942, -0.5538362212110504, -0.1563637168806138,
-0.3234202061929452, -1.6929655586752923]
```

```python
# use cross_val_scores to calculate CV scores
model = LinearRegression()
scores = cross_val_score(model, X, y, cv=5)
print(scores)
```

```
[ 0.02731416 -0.55383622 -0.15636372 -0.32342021 -1.69296556]
```

```python
print(scores.mean())
```

```
-0.5398543080098925
```

```python
# create the predictor and response matrices
y1, X1 = dmatrices(
  "value_per_sq_ft ~ units + sq_ft + boro", housing)

y2, X2 = dmatrices("value_per_sq_ft ~ units*sq_ft + boro", housing)

y3, X3 = dmatrices(
  "value_per_sq_ft ~ units + sq_ft*boro + type", housing
)

y4, X4 = dmatrices(
    "value_per_sq_ft ~ units + sq_ft*boro + sq_ft*type", housing
)

y5, X5 = dmatrices("value_per_sq_ft ~ boro + type", housing)
```

```python
# fit our models
model = LinearRegression()

scores1 = cross_val_score(model, X1, y1, cv=5)
scores2 = cross_val_score(model, X2, y2, cv=5)
scores3 = cross_val_score(model, X3, y3, cv=5)
scores4 = cross_val_score(model, X4, y4, cv=5)
scores5 = cross_val_score(model, X5, y5, cv=5)
```

```python
scores_df = pd.DataFrame(
    [scores1, scores2, scores3, scores4, scores5]
)

print(scores_df.apply(np.mean, axis=1))
```

```
0   -5.398543e-01
1   -1.088184e+00
2   -8.668885e+25
3   -7.634198e+25
4   -3.172546e+25
dtype: float64
```

```python
import pandas as pd
acs = pd.read_csv('data/acs_ny.csv')
print(acs.columns)
```

```
Index(['Acres', 'FamilyIncome', 'FamilyType', 'NumBedrooms',
       'NumChildren', 'NumPeople', 'NumRooms', 'NumUnits',
       'NumVehicles', 'NumWorkers', 'OwnRent', 'YearBuilt',
       'HouseCosts', 'ElectricBill', 'FoodStamp', 'HeatingFuel',
       'Insurance', 'Language'],
      dtype='object')
```

```python
from patsy import dmatrices

# sequential strings get concatenated together in Python
response, predictors = dmatrices(
  "FamilyIncome ~ NumBedrooms + NumChildren + NumPeople + "
  "NumRooms + NumUnits + NumVehicles + NumWorkers + OwnRent + "
  "YearBuilt + ElectricBill + FoodStamp + HeatingFuel + "
  "Insurance + Language",
  data=acs,
)
```

```python
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
  predictors, response, random_state=0
)
```

```python
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

lr = make_pipeline(
  StandardScaler(with_mean=False), LinearRegression()
)
```

```python
lr = lr.fit(X_train, y_train)
print(lr)
```

```
Pipeline(steps=[('standardscaler', StandardScaler(with_mean=False)),
                ('linearregression', LinearRegression())])
```

```python
model_coefs = pd.DataFrame(
  data=list(
    zip(
      predictors.design_info.column_names,
      lr.named_steps["linearregression"].coef_[0],
    )
  ),
  columns=["variable", "coef_lr"],
)

print(model_coefs)
```

```
                       variable       coef_lr
0                     Intercept  2.697159e-13
1   NumUnits[T.Single attached]  9.661755e+03
2   NumUnits[T.Single detached]  8.345408e+03
3            OwnRent[T.Outright]  2.382740e+03
4              OwnRent[T.Rented]  2.260806e+03
..                          ...           ...
34                     NumRooms  1.340575e+04
35                  NumVehicles  7.228920e+03
36                   NumWorkers  1.877535e+04
37                 ElectricBill  1.000008e+04
38                    Insurance  3.072892e+04

[39 rows x 2 columns]
```

```python
# score on the _training_ data
print(lr.score(X_train, y_train))
```

```
0.2726140465638568
```

```python
# score on the _testing_ data
print(lr.score(X_test, y_test))
```

```
0.26976979568488013
```

```python
from sklearn.linear_model import Lasso

lasso = make_pipeline(
  StandardScaler(with_mean=False),
  Lasso(max_iter=10000, random_state=42),
)

lasso = lasso.fit(X_test, y_test)
print(lasso)
```

```
Pipeline(steps=[('standardscaler', StandardScaler(with_mean=False)),
                ('lasso', Lasso(max_iter=10000, random_state=42))])
```

```python
coefs_lasso = pd.DataFrame(
  data=list(
    zip(
      predictors.design_info.column_names,
      lasso.named_steps["lasso"].coef_.tolist(),
    )
  ),
  columns=["variable", "coef_lasso"],
)
```

```
model_coefs = pd.merge(model_coefs, coefs_lasso, on='variable')
print(model_coefs)
```

```
                        variable       coef_lr    coef_lasso
0                      Intercept  2.697159e-13      0.000000
1   NumUnits[T.Single attached]  9.661755e+03   7765.482025
2   NumUnits[T.Single detached]  8.345408e+03   7512.067593
3            OwnRent[T.Outright]  2.382740e+03   2431.710977
4              OwnRent[T.Rented]  2.260806e+03    604.186925
..                           ...           ...           ...
34                      NumRooms  1.340575e+04  10940.150208
35                   NumVehicles  7.228920e+03   7724.681161
36                    NumWorkers  1.877535e+04  16911.035390
37                  ElectricBill  1.000008e+04   9516.123582
38                     Insurance  3.072892e+04  32155.544169

[39 rows x 3 columns]
```

```
print(lasso.score(X_train, y_train))
```

```
0.2669751487716776
```

```
print(lasso.score(X_test, y_test))
```

```
0.2752627973740016
```

```
from sklearn.linear_model import Ridge

ridge = make_pipeline(
    StandardScaler(with_mean=False), Ridge(random_state=42)
)

ridge = ridge.fit(X_train, y_train)
print(ridge)
```

```
Pipeline(steps=[('standardscaler', StandardScaler(with_mean=False)),
                ('ridge', Ridge(random_state=42))])
```

```
coefs_ridge = pd.DataFrame(
  data=list(
    zip(
      predictors.design_info.column_names,
      ridge.named_steps["ridge"].coef_.tolist()[0],
    )
  ),
  columns=["variable", "coef_ridge"],
)

model_coefs = pd.merge(model_coefs, coefs_ridge, on="variable")
print(model_coefs)
```

```
                      variable       coef_lr    coef_lasso  \
0                    Intercept  2.697159e-13      0.000000
1   NumUnits[T.Single attached]  9.661755e+03   7765.482025
2   NumUnits[T.Single detached]  8.345408e+03   7512.067593
3           OwnRent[T.Outright]  2.382740e+03   2431.710977
4             OwnRent[T.Rented]  2.260806e+03    604.186925
..                         ...           ...           ...
34                    NumRooms  1.340575e+04  10940.150208
35                 NumVehicles  7.228920e+03   7724.681161
36                  NumWorkers  1.877535e+04  16911.035390
37                ElectricBill  1.000008e+04   9516.123582
38                   Insurance  3.072892e+04  32155.544169

     coef_ridge
0      0.000000
1   9659.413514
2   8342.247690
3   2381.429615
4   2259.526329
..          ...
```

```
34  13405.409584
35   7228.542922
36  18773.079462
37  10000.853603
38  30727.230542

[39 rows x 4 columns]
```

```
from sklearn.linear_model import ElasticNet

en = ElasticNet(random_state=42).fit(X_train, y_train)

coefs_en = pd.DataFrame(
    list(zip(predictors.design_info.column_names, en.coef_)),
    columns=["variable", "coef_en"],
)

model_coefs = pd.merge(model_coefs, coefs_en, on="variable")
print(model_coefs)
```

```
                         variable       coef_lr    coef_lasso  \
0                       Intercept  2.697159e-13      0.000000
1   NumUnits[T.Single attached]  9.661755e+03   7765.482025
2   NumUnits[T.Single detached]  8.345408e+03   7512.067593
3            OwnRent[T.Outright]  2.382740e+03   2431.710977
4              OwnRent[T.Rented]  2.260806e+03    604.186925
..                            ...           ...           ...
34                       NumRooms  1.340575e+04  10940.150208
35                    NumVehicles  7.228920e+03   7724.681161
36                     NumWorkers  1.877535e+04  16911.035390
37                   ElectricBill  1.000008e+04   9516.123582
38                      Insurance  3.072892e+04  32155.544169

     coef_ridge       coef_en
0      0.000000      0.000000
1   9659.413514   1342.291706
2   8342.247690    168.728479
```

```
3     2381.429615    445.533238
4     2259.526329   -600.673747
..            ...            ...
34   13405.409584   5685.101939
35    7228.542922   6059.776166
36   18773.079462  12247.547800
37   10000.853603     97.566664
38   30727.230542     32.484207

[39 rows x 5 columns]
```

```
print(model_coefs.loc[model_coefs["coef_lasso"] == 0])
```

```
                variable       coef_lr  coef_lasso  coef_ridge  \
0              Intercept  2.697159e-13         0.0    0.000000
25  HeatingFuel[T.Solar]  1.442204e+02         0.0  142.354045

     coef_en
0   0.000000
25  0.994142
```

```
from sklearn.linear_model import ElasticNetCV

en_cv = ElasticNetCV(cv=5, random_state=42).fit(
    X_train, y_train.ravel() # ravel is to remove the 1d warning
)

coefs_en_cv = pd.DataFrame(
    list(zip(predictors.design_info.column_names, en_cv.coef_)),
    columns=["variable", "coef_en_cv"],
)

model_coefs = pd.merge(model_coefs, coefs_en_cv, on="variable")
print(model_coefs)
```

```
                       variable       coef_lr   coef_lasso  \
0                     Intercept  2.697159e-13     0.000000   
1   NumUnits[T.Single attached]  9.661755e+03  7765.482025   
2   NumUnits[T.Single detached]  8.345408e+03  7512.067593   
3           OwnRent[T.Outright]  2.382740e+03  2431.710977   
4             OwnRent[T.Rented]  2.260806e+03   604.186925   
..                          ...           ...          ...   
34                     NumRooms  1.340575e+04 10940.150208   
35                  NumVehicles  7.228920e+03  7724.681161   
36                   NumWorkers  1.877535e+04 16911.035390   
37                 ElectricBill  1.000008e+04  9516.123582   
38                    Insurance  3.072892e+04 32155.544169   

      coef_ridge       coef_en  coef_en_cv  
0       0.000000      0.000000    0.000000  
1    9659.413514   1342.291706   -0.000000  
2    8342.247690    168.728479    0.000000  
3    2381.429615    445.533238    0.000000  
4    2259.526329   -600.673747   -0.000000  
..           ...           ...         ...  
34  13405.409584   5685.101939    0.028443  
35   7228.542922   6059.776166    0.000000  
36  18773.079462  12247.547800    0.000000  
37  10000.853603     97.566664   26.166320  
38  30727.230542     32.484207   38.561748  

[39 rows x 6 columns]
```

```
print(model_coefs.loc[model_coefs["coef_en_cv"] == 0])
```

```
                      variable       coef_lr    coef_lasso  \
0                    Intercept  2.697159e-13      0.000000
1   NumUnits[T.Single attached]  9.661755e+03   7765.482025
2   NumUnits[T.Single detached]  8.345408e+03   7512.067593
3           OwnRent[T.Outright]  2.382740e+03   2431.710977
4             OwnRent[T.Rented]  2.260806e+03    604.186925
..                         ...           ...           ...
31                 NumBedrooms  3.755708e+03   4447.892458
32                 NumChildren  9.524915e+03   6905.672216
33                   NumPeople -1.153672e+04  -8777.265840
35                 NumVehicles  7.228920e+03   7724.681161
36                  NumWorkers  1.877535e+04  16911.035390

      coef_ridge       coef_en  coef_en_cv
0       0.000000      0.000000         0.0
1    9659.413514   1342.291706        -0.0
2    8342.247690    168.728479         0.0
3    2381.429615    445.533238         0.0
4    2259.526329   -600.673747        -0.0
..           ...           ...         ...
31   3755.521256   2073.910045         0.0
32   9521.180875   2498.719581         0.0
33 -11533.098634  -2562.412933         0.0
35   7228.542922   6059.776166         0.0
36  18773.079462  12247.547800         0.0

[36 rows x 6 columns]
```

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

```python
wine = wine.drop('Cultivar', axis=1)

# note that the data values are all numeric
print(wine.columns)
```

```
Index(['Alcohol', 'Malic acid', 'Ash', 'Alcalinity of ash  ',
       'Magnesium', 'Total phenols', 'Flavanoids',
       'Nonflavanoid phenols', 'Proanthocyanins', 'Color intensity',
       'Hue', 'OD280/OD315 of diluted wines', 'Proline            '],
      dtype='object')
```

```
print(wine.head())
```

```
   Alcohol  Malic acid   Ash  Alcalinity of ash    Magnesium  \
0    14.23        1.71  2.43                 15.6        127
1    13.20        1.78  2.14                 11.2        100
2    13.16        2.36  2.67                 18.6        101
3    14.37        1.95  2.50                 16.8        113
4    13.24        2.59  2.87                 21.0        118

   Total phenols  Flavanoids  Nonflavanoid phenols  Proanthocyanins  \
0          2.80        3.06                  0.28             2.29
1          2.65        2.76                  0.26             1.28
2          2.80        3.24                  0.30             2.81
3          3.85        3.49                  0.24             2.18
4          2.80        2.69                  0.39             1.82

   Color intensity   Hue  OD280/OD315 of diluted wines  \
0             5.64  1.04                          3.92
1             4.38  1.05                          3.40
2             5.68  1.03                          3.17
3             7.80  0.86                          3.45
4             4.32  1.04                          2.93

   Proline
0              1065
1              1050
2              1185
3              1480
4               735
```

```python
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3, random_state=42).fit(wine.values)
```

```python
print(kmeans)
```

```
KMeans(n_clusters=3, random_state=42)
```

```python
import numpy as np
print(np.unique(kmeans.labels_, return_counts=True))
```

```
(array([0, 1, 2], dtype=int32), array([69, 47, 62]))
```

```python
kmeans_3 = pd.DataFrame(kmeans.labels_, columns=['cluster'])
print(kmeans_3)
```

```
     cluster
0          1
1          1
2          1
3          1
4          2
..       ...
173        2
174        2
175        2
176        2
177        0

[178 rows x 1 columns]
```

```python
from sklearn.decomposition import PCA
```

```python
# project our data into 2 components
pca = PCA(n_components=2).fit(wine)
```

```python
# transform our data into the new space
pca_trans = pca.transform(wine)

# give our projections a name
pca_trans_df = pd.DataFrame(pca_trans, columns=['pca1', 'pca2'])

# concatenate our data
kmeans_3 = pd.concat([kmeans_3, pca_trans_df], axis=1)

print(kmeans_3)
```

```
     cluster        pca1       pca2
0          1  318.562979  21.492131
1          1  303.097420  -5.364718
2          1  438.061133  -6.537309
3          1  733.240139   0.192729
4          2  -11.571428  18.489995
..       ...         ...        ...
173        2   -6.980211  -4.541137
174        2    3.131605   2.335191
175        2   88.458074  18.776285
176        2   93.456242  18.670819
177        0 -186.943190  -0.213331

[178 rows x 3 columns]
```

```python
import seaborn as sns
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

sns.scatterplot(
  x="pca1",
  y="pca2",
  data=kmeans_3,
  hue="cluster",
  ax=ax
)

plt.show()
```

```
wine_all = pd.read_csv('data/wine.csv')
print(wine_all.head())
```

```
   Cultivar  Alcohol  Malic acid   Ash  Alcalinity of ash    \
0         1    14.23        1.71  2.43               15.6
1         1    13.20        1.78  2.14               11.2
2         1    13.16        2.36  2.67               18.6
3         1    14.37        1.95  2.50               16.8
4         1    13.24        2.59  2.87               21.0

   Magnesium  Total phenols  Flavanoids  Nonflavanoid phenols  \
0        127           2.80        3.06                  0.28
1        100           2.65        2.76                  0.26
2        101           2.80        3.24                  0.30
3        113           3.85        3.49                  0.24
4        118           2.80        2.69                  0.39

   Proanthocyanins  Color intensity   Hue  \
0             2.29             5.64  1.04
1             1.28             4.38  1.05
2             2.81             5.68  1.03
3             2.18             7.80  0.86
4             1.82             4.32  1.04

   OD280/OD315 of diluted wines  Proline
0                          3.92                1065
1                          3.40                1050
2                          3.17                1185
3                          3.45                1480
4                          2.93                 735
```

```python
pca_all = PCA(n_components=2).fit(wine_all)
pca_all_trans = pca_all.transform(wine_all)
pca_all_trans_df = pd.DataFrame(
  pca_all_trans, columns=["pca_all_1", "pca_all_2"]
)

kmeans_3 = pd.concat(
  [kmeans_3, pca_all_trans_df, wine_all["Cultivar"]], axis=1
)
```

```python
with sns.plotting_context(context="talk"):
  fig = sns.relplot(
   x="pca_all_1",
   y="pca_all_2",
   data=kmeans_3,
   row="cluster",
   col="Cultivar",
  )

fig.figure.set_tight_layout(True)
plt.show()
```

```python
print(
  pd.crosstab(
    kmeans_3["cluster"], kmeans_3["Cultivar"], margins=True
  )
)
```

```
Cultivar   1   2   3  All
cluster
0          0  50  19   69
1         46   1   0   47
2         13  20  29   62
All       59  71  48  178
```

```python
from scipy.cluster import hierarchy
```

```python
wine = pd.read_csv('data/wine.csv')
wine = wine.drop('Cultivar', axis=1)
```

```python
import matplotlib.pyplot as plt
```

```python
wine_complete = hierarchy.complete(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_complete)
plt.show()
```

```python
wine_single = hierarchy.single(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_single)
plt.show()
```

```python
wine_average = hierarchy.average(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_average)
plt.show()
```

```python
wine_centroid = hierarchy.centroid(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_centroid)
plt.show()
```

```python
wine_ward = hierarchy.ward(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_ward)
plt.show()
```

```python
wine_complete = hierarchy.complete(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(
```

```
    wine_complete,
    # default MATLAB threshold
    color_threshold=0.7 * max(wine_complete[:,2]),
    above_threshold_color='y')
plt.show()
```

https://carpentries.github.io/workshop-template/#python

```bash
cd ~/Downloads
bash Anaconda3-*.sh # your version number will differ
```

```
% python
Python 3.9.12 (main, Jun  1 2022, 06:34:44)
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

```bash
# type this in the (bash) terminal, not in python
conda create -n py38 python=3.8
```

```
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /Users/danielchen/anaconda3/envs/py38

  added / updated specs:
    - python=3.8


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    ca-certificates-2022.07.19 |       hca03da5_0         124 KB
    certifi-2022.6.15          |   py38hca03da5_0         153 KB
    libffi-3.4.2               |       hc377ac9_4         106 KB
    ncurses-6.3                |       h1a28f6b_3         866 KB
    openssl-1.1.1q             |       h1a28f6b_0         2.2 MB
    pip-22.1.2                 |   py38hca03da5_0         2.5 MB
    python-3.8.13              |       hbdb9e5c_0        10.6 MB
    setuptools-63.4.1          |   py38hca03da5_0         1.1 MB
    sqlite-3.39.2              |       h1058600_0         1.1 MB
    ------------------------------------------------------------
                                           Total:        18.6 MB

The following NEW packages will be INSTALLED:
```

```
  ca-certificates pkgs/main/osx-arm64::ca-certificates-2022.07.19-hca03da5_0
  certifi         pkgs/main/osx-arm64::certifi-2022.6.15-py38hca03da5_0
  libcxx          pkgs/main/osx-arm64::libcxx-12.0.0-hf6beb65_1
  libffi          pkgs/main/osx-arm64::libffi-3.4.2-hc377ac9_4
  ncurses         pkgs/main/osx-arm64::ncurses-6.3-h1a28f6b_3
  openssl         pkgs/main/osx-arm64::openssl-1.1.1q-h1a28f6b_0
  pip             pkgs/main/osx-arm64::pip-22.1.2-py38hca03da5_0
  python          pkgs/main/osx-arm64::python-3.8.13-hbdb9e5c_0
  readline        pkgs/main/osx-arm64::readline-8.1.2-h1a28f6b_1
  setuptools      pkgs/main/osx-arm64::setuptools-63.4.1-py38hca03da5_0
  sqlite          pkgs/main/osx-arm64::sqlite-3.39.2-h1058600_0
  tk              pkgs/main/osx-arm64::tk-8.6.12-hb8d0fd4_0
  wheel           pkgs/main/noarch::wheel-0.37.1-pyhd3eb1b0_0
  xz              pkgs/main/osx-arm64::xz-5.2.5-h1a28f6b_1
  zlib            pkgs/main/osx-arm64::zlib-1.2.12-h5a0b063_2


Proceed ([y]/n)? y

Downloading and Extracting Packages
certifi-2022.6.15     | 153 KB    | ########################## | 100%
python-3.8.13         | 10.6 MB   | ########################## | 100%
openssl-1.1.1q        | 2.2 MB    | ########################## | 100%
```

```
setuptools-63.4.1    | 1.1 MB    | ########################## | 100%
ca-certificates-2022 | 124 KB    | ########################## | 100%
pip-22.1.2           | 2.5 MB    | ########################## | 100%
sqlite-3.39.2        | 1.1 MB    | ########################## | 100%
ncurses-6.3          | 866 KB    | ########################## | 100%
libffi-3.4.2         | 106 KB    | ########################## | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate py38
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

```
% python
```

```
Python 3.8.13 (default, Mar 28 2022, 06:13:39)
[Clang 12.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
```

```
conda create --name p4e python=3
```

```
# typed into your terminal, not in Python
conda install pandas
```

```
conda install -c conda-forge pandas
```

```
conda install -c conda-forge pandas matplotlib pyarrow openpyxl \
  seaborn numba regex pandas-datareader statsmodels scikit-learn \
  arrow lifelines
```

```python
print(pandas.read_csv('data/concat_1.csv'))
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

```python
print(pd.read_csv('data/concat_1.csv'))
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

```python
print(read_csv('data/concat_1.csv'))
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

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

```python
# this code is wide and will run off the page
weather_melt = weather.melt(id_vars=["id", "year", "month", "element"],
var_name="day", value_name="temp")
```

```python
# previous line of code can be rewritten as
weather_melt = weather.melt(
  id_vars=["id", "year", "month", "element"],
  var_name="day",
  value_name="temp",
)
```

```python
# this will error, putting line break before the .melt
# previous line of code can be rewritten as
weather_melt = weather
  .melt(
    id_vars=["id", "year", "month", "element"],
    var_name="day",
    value_name="temp")
```

```python
# use a \ at the end of the line
weather_melt = weather \
  .melt(
    id_vars=["id", "year", "month", "element"],
    var_name="day",
    value_name="temp")
```

```python
# wrap the entire statement around ( )
weather_melt = (weather
  .melt(
    id_vars=["id", "year", "month", "element"],
    var_name="day",
    value_name="temp")
)
```

```python
my_list = ['a', 1, True, 3.14]
print(my_list)
```

```
['a', 1, True, 3.14]
```

```python
# get the first item - index 0
print(my_list[0])
```

```
a
```

```python
my_list.append('appended a new value!')
print(my_list)
```

```
['zzzzz', 1, True, 3.14, 'appended a new value!']
```

```python
my_tuple =('a', 1, True, 3.14)
print(my_tuple)
```

```
('a', 1, True, 3.14)
```

```python
# this will cause an error
my_tuple[0] = 'zzzzz'
```

```
TypeError: 'tuple' object does not support item assignment
```

```python
my_dict = {'fname': 'Daniel', 'lname': 'Chen'}
print(my_dict)
```

```
{'fname': 'Daniel', 'lname': 'Chen'}
```

```python
# get all the keys in the dictionary
print(my_dict.keys())
```

```
dict_keys(['fname', 'lname'])
```

```python
# get all the values in the dictionary
print(my_dict.values())
```

```
dict_values(['Daniel', 'Chen'])
```

```python
print(my_dict.items())
```

```
dict_items([('fname', 'Daniel'), ('lname', 'Chen')])
```

```python
# get every other value starting from the first value
print(l[::2])
```

```
['one', 'three']
```

```python
# an example list of values to iterate over
l = [1, 2, 3]

# write a for loop that prints the value and its squared value
for i in l:
  # print the current value
  print(f"the current value is: {i}")

  # print the square of the value
  print(f"its squared value is: {i*i}")

  # end of the loop, the \n at the end creates a new line
  print("end of loop, going back to the top\n")
```

```
the current value is: 1
its squared value is: 1
end of loop, going back to the top

the current value is: 2
its squared value is: 4
end of loop, going back to the top

the current value is: 3
its squared value is: 9
end of loop, going back to the top
```

```python
# create a list
l = [1, 2, 3, 4, 5]

# list of newly calculated results
r = []

# iterate over the list
for i in l:
  # square each number and add the new value to a new list
  r.append(i ** 2)

print(r)
```

```
[1, 4, 9, 16, 25]
```

```python
# note the square brackets around on the right-hand side
# this saves the final results as a list
rc = [i ** 2 for i in l]
print(rc)
```

```
[1, 4, 9, 16, 25]
```

```python
print(type(rc))
```

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

```python
def empty_function():
  """This is an empty function with a docstring.
  These docstrings are used to help document the function.
  They can be created by using 3 single quotes or 3 double quotes.
  The PEP-8 style guide says to use double quotes.
  """
  pass  # this function still does nothing
```

```python
def print_value():
  """Just prints the value 3
  """
  print(3)
```

```python
# call our print_value function
print_value()
```

```
3
```

```python
def print_value(value):
  """Prints the value passed into the parameter 'value'
  """
  print(value)
```

```python
print_value(3)
```

```
3
```

```python
print_value("Hello!")
```

```
Hello!
```

```python
def person(fname, lname):
  """A function that takes 3 values, and prints them
  """
  print(fname)
  print(lname)
```

```python
person('Daniel', 'Chen')
```

```
Daniel
Chen
```

```python
def my_mean_2(x, y):
  """A function that returns the mean of 2 values
  """
  mean_value = (x + y) / 2
  return mean_value

m = my_mean_2(0, 10)
print(m)
```

```
5.0
```

```python
def my_mean_3(x, y, z=20):
  """A function with a parameter z that has a default value
  """
  # you can also directly return values without having to create
  # an intermediate variable
  return (x + y + z) / 3
```

```python
def my_mean(*args):
    """Calculate the mean for an arbitrary number of values
    """
    # add up all the values
    sum = 0
    for i in args:
        sum += i
    return sum / len(args)
```

```python
print(my_mean(0, 10))
```

```
5.0
```

```python
print(my_mean(0, 50, 100))
```

```
50.0
```

```python
print(my_mean(3, 10, 25, 2))
```

```
10.0
```

```python
def greetings(welcome_word, **kwargs):
    """Prints out a greeting to a person,
    where the person's fname and lname are provided by the kwargs
    """
    print(welcome_word)
    print(kwargs.get('fname'))
    print(kwargs.get('lname'))
```

```python
greetings('Hello!', fname='Daniel', lname='Chen')
```

```
Hello!
Daniel
Chen
```

```python
import itertools
prod = itertools.product([1, 2, 3], ['a', 'b', 'c'])

for i in prod:
  print(i)
```

```
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
```

```python
# this will not work because we already used this generator
for i in prod:
  print(i)
```

```python
# create a new generator
prod = itertools.product([1, 2, 3], ['a', 'b', 'c'])
for i in prod:
  print(i)
```

```
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
```

```
import matplotlib.pyplot as plt

f, ax = plt.subplots()
```

```python
import pandas as pd

df = pd.read_csv('data/concat_1.csv')
print(df)
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

```python
a = df['A']
print(a)
```

```
0    a0
1    a1
2    a2
3    a3
Name: A, dtype: object
```

```python
print(type(a))
```

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

```python
print(a.values)
```

```
['a0' 'a1' 'a2' 'a3']
```

```python
print(type(a.values))
```

```
<class 'numpy.ndarray'>
```

```python
class Person(object):
  def __init__(self, fname, lname, age):
    self.fname = fname
    self.lname = lname
    self.age = age

  def celebrate_birthday(self):
    self.age += 1
    return(self)
```

```python
ka = Person(fname='King', lname='Arthur', age=39)
```

```python
import pandas as pd

dat = pd.read_csv("data/concat_1.csv")
print(dat)
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

```python
subset = dat[["A", "C"]]
print(subset)
```

```
    A   C
0  a0  c0
1  a1  c1
2  a2  c2
3  a3  c3
```

```python
# this will trigger the warning
subset["new"] = ["bunch", "of", "new", "values"]
print(subset)
```

```
    A   C     new
0  a0  c0   bunch
1  a1  c1      of
2  a2  c2     new
3  a3  c3  values
```

```
/var/folders/2b/qckmp39n7qn1dh0tpcm8g89w0000gn/T/ipykernel_29772/
4023129152.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/
pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  subset["new"] = ["bunch", "of", "new", "values"]
```

```python
subset = dat[["A", "C"]].copy() # explicity copy
print(subset)
```

```
    A   C
0  a0  c0
1  a1  c1
2  a2  c2
3  a3  c3
```

```python
# no more warning!
subset["new"] = ["bunch", "of", "new", "values"]
print(subset)
```

```
    A   C     new
0  a0  c0   bunch
1  a1  c1      of
2  a2  c2     new
3  a3  c3  values
```

```python
# reset our data
dat = pd.read_csv("data/concat_1.csv")
print(dat)
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

```python
# want to replace the c2 value
# filter the rows and separately select the column
dat.loc[dat["C"] == "c2"]["C"] = "new value"

print(dat)
```

```
    A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3

/var/folders/2b/qckmp39n7qn1dh0tpcm8g89w0000gn/T/ipykernel_29772/
3306879196.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/
pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  dat.loc[dat["C"] == "c2"]["C"] = "new value"
```

```python
dat = pd.read_csv("data/concat_1.csv")
dat.loc[dat["C"] == "c2", ["C"] ] = "new value"
print(dat)
```

```
    A   B          C   D
0  a0  b0         c0  d0
1  a1  b1         c1  d1
2  a2  b2  new value  d2
3  a3  b3         c3  d3
```

```python
class Person(object):
    def __init__(self, fname, lname, age):
        self.fname = fname
        self.lname = lname
        self.age = age

    def celebrate_birthday(self):
        self.age += 1
        return(self)
```

```python
ka = Person(fname='King', lname='Arthur', age=39)
print(ka.age)
```

```
39
```

```python
# King Arthur has 2 birthdays in a row!
ka.celebrate_birthday().celebrate_birthday()
```

```
<__main__.Person at 0x1039903a0>
```

```python
print(ka.age)
```

```
41
```

```python
import pandas as pd

weather = pd.read_csv('data/weather.csv')
print(weather.head())
```

```
        id  year  month element  d1    d2    d3  d4    d5  d6  ...  \
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  ...
3  MX17004  2010      2    tmin NaN  14.4  14.4 NaN   NaN NaN  ...
4  MX17004  2010      3    tmax NaN   NaN   NaN NaN  32.1 NaN  ...

   d22   d23  d24  d25  d26  d27  d28  d29   d30  d31
0  NaN   NaN  NaN  NaN  NaN  NaN  NaN  NaN  27.8  NaN
1  NaN   NaN  NaN  NaN  NaN  NaN  NaN  NaN  14.5  NaN
2  NaN  29.9  NaN  NaN  NaN  NaN  NaN  NaN   NaN  NaN
3  NaN  10.7  NaN  NaN  NaN  NaN  NaN  NaN   NaN  NaN
4  NaN   NaN  NaN  NaN  NaN  NaN  NaN  NaN   NaN  NaN

[5 rows x 35 columns]
```

```python
weather_tidy = (
    weather
    .melt(
        id_vars=["id", "year", "month", "element"],
        var_name="day",
        value_name="temp",
    )
    .pivot_table(
        index=["id", "year", "month", "day"],
        columns="element",
        values="temp",
    )
    .reset_index()
)

print(weather_tidy)
```

```
element        id  year  month  day  tmax  tmin
0         MX17004  2010      1  d30  27.8  14.5
1         MX17004  2010      2  d11  29.7  13.4
2         MX17004  2010      2   d2  27.3  14.4
3         MX17004  2010      2  d23  29.9  10.7
4         MX17004  2010      2   d3  24.1  14.4
..            ...   ...    ...  ...   ...   ...
28        MX17004  2010     11  d27  27.7  14.2
29        MX17004  2010     11  d26  28.1  12.1
30        MX17004  2010     11   d4  27.2  12.0
31        MX17004  2010     12   d1  29.9  13.8
32        MX17004  2010     12   d6  27.8  10.5

[33 rows x 6 columns]
```

```python
import pandas as pd
import numpy as np
import numba


def avg_2(x, y):
  return (x + y) / 2


@np.vectorize
def v_avg_2_mod(x, y):
  """Calculate the average, unless x is 20
  Same as before, but we are using the vectorize decorator
  """
  if (x == 20):
    return(np.NaN)
  else:
    return (x + y) / 2


@numba.vectorize
def v_avg_2_numba(x, y):
  """Calculate the average, unless x is 20
  Using the numba decorator.
  """
  # we now have to add type information to our function
  if (int(x) == 20):
    return(np.NaN)
```

```python
  else:
    return (x + y) / 2


df = pd.DataFrame({"a": [10, 20, 30], "b": [20, 30, 40]})
print(df)
```

	a	b
0	10	20
1	20	30
2	30	40

```python
%%timeit
avg_2(df['a'], df['b'])
```

```
67.1 µs ± 12.7 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```python
%%timeit
v_avg_2_mod(df['a'], df['b'])
```

```
16.6 µs ± 1.05 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```python
%%timeit
v_avg_2_numba(df['a'].values, df['b'].values)
```

```
3.92 µs ± 632 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```python
s = 'I only know %d digits of pi' % 7
print(s)
```

```
I only know 7 digits of pi
```

```python
print(
    "Some digits of %(cont)s: %(value).2f"
    % {"cont": "e", "value": 2.718}
)
```

```
Some digits of e: 2.72
```

```python
# using variables multiple times by index
s = """Black Knight: 'Tis but a {0}.
King Arthur: A {0}? Your arm's off!
"""
print(s.format('scratch'))
```

```
Black Knight: 'Tis but a scratch.
King Arthur: A scratch? Your arm's off!
```

```python
s = 'Hayden Planetarium Coordinates: {lat}, {lon}'
print(s.format(lat='40.7815° N', lon='73.9733° W'))
```

```
Hayden Planetarium Coordinates: 40.7815° N, 73.9733° W
```

```python
print('Some digits of pi: {}'.format(3.14159265359))
```

```
Some digits of pi: 3.14159265359
```

```python
print(
  "In 2005, Lu Chao of China recited {:,} digits of pi".format(67890)
)
```

```
In 2005, Lu Chao of China recited 67,890 digits of pi
```

```python
# the 0 in {0:.4} and {0:.4%} refer to the 0 index in this format
# the .4 refers to how many decimal values, 4
# if we provide a %, it will format the decimal as a percentage
print(
  "I remember {0:.4} or {0:.4%} of what Lu Chao recited".format(
    7 / 67890
  )
)
```

```
I remember 0.0001031 or 0.0103% of what Lu Chao recited
```

```python
# the first 0 refers to the index in this format
# the second 0 refers to the character to fill
# the 5 in this case refers to how many characters in total
# the d signals a digit will be used
# Pad the number with 0s so the entire string has 5 characters
print("My ID number is {0:05d}".format(42))
```

```
My ID number is 00042
```

```python
my_num = 7

if my_num % 2 == 0:
  print("fizz")
elif my_num % 4 == 0:
  print("buzz")
else:
  print("Not multiple of 2 or 4.")
```

```
Not multiple of 2 or 4.
```

```python
import pandas as pd

acs = pd.read_csv('data/acs_ny.csv')
print(acs.columns)
```

```
Index(['Acres', 'FamilyIncome', 'FamilyType', 'NumBedrooms', 'NumChildren',
       'NumPeople', 'NumRooms', 'NumUnits', 'NumVehicles', 'NumWorkers',
       'OwnRent', 'YearBuilt', 'HouseCosts', 'ElectricBill', 'FoodStamp',
       'HeatingFuel', 'Insurance', 'Language'],
      dtype='object')
```

```python
print(acs.head())
```

```
  Acres  FamilyIncome   FamilyType  NumBedrooms  NumChildren  NumPeople  \
0  1-10           150      Married            4            1          3
1  1-10           180  Female Head            3            2          4
2  1-10           280  Female Head            4            0          2
3  1-10           330  Female Head            2            1          2
4  1-10           330    Male Head            3            1          2

     Num          Num          Num          Num        Own        Year  \
    Rooms         Units         Vehicles     Workers     Rent       Built
0       9  Single detached            1           0  Mortgage    1950-1959
1       6  Single detached            2           0    Rented  Before 1939
2       8  Single detached            3           1  Mortgage    2000-2004
3       4  Single detached            1           0    Rented    1950-1959
4       5  Single attached            1           0  Mortgage  Before 1939

     House      Electric    Food       Heating   Insurance        Language
     Costs      Bill        Stamp      Fuel
0     1800            90      No           Gas       2500          English
1      850            90      No           Oil          0          English
```

```
2      2600           260         No           Oil          6600  Other European
3      1800           140         No           Oil             0         English
4       860           150         No           Gas           660         Spanish
```

```
acs["ge150k"] = pd.cut(
    acs["FamilyIncome"],
    [0, 150000, acs["FamilyIncome"].max()],
    labels=[0, 1],
)

acs["ge150k_i"] = acs["ge150k"].astype(int)
print(acs["ge150k_i"].value_counts())
```

```
0    18294
1     4451
Name: ge150k_i, dtype: int64
```

```python
acs.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 22745 entries, 0 to 22744
Data columns (total 20 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Acres         22745 non-null  object
 1   FamilyIncome  22745 non-null  int64
 2   FamilyType    22745 non-null  object
 3   NumBedrooms   22745 non-null  int64
 4   NumChildren   22745 non-null  int64
 5   NumPeople     22745 non-null  int64
 6   NumRooms      22745 non-null  int64
 7   NumUnits      22745 non-null  object
 8   NumVehicles   22745 non-null  int64
 9   NumWorkers    22745 non-null  int64
 10  OwnRent       22745 non-null  object
 11  YearBuilt     22745 non-null  object
 12  HouseCosts    22745 non-null  int64
 13  ElectricBill  22745 non-null  int64

 14  FoodStamp     22745 non-null  object
 15  HeatingFuel   22745 non-null  object
 16  Insurance     22745 non-null  int64
 17  Language      22745 non-null  object
 18  ge150k        22745 non-null  category
 19  ge150k_i      22745 non-null  int64
dtypes: category(1), int64(11), object(8)
memory usage: 3.3+ MB
```

```python
acs_sub = acs[
  [
    "ge150k_i",
    "HouseCosts",
    "NumWorkers",
    "OwnRent",
    "NumBedrooms"
```

```python
        "NumBedrooms",
        "FamilyType",
    ]
].copy()

print(acs_sub)
```

```
       ge150k_i  HouseCosts  NumWorkers   OwnRent  NumBedrooms   FamilyType
0             0        1800           0  Mortgage            4      Married
1             0         850           0    Rented            3  Female Head
2             0        2600           1  Mortgage            4  Female Head
3             0        1800           0    Rented            2  Female Head
4             0         860           0  Mortgage            3    Male Head
...         ...         ...         ...       ...          ...          ...
22740         1        1700           2  Mortgage            5      Married
22741         1        1300           2  Mortgage            4      Married
22742         1         410           3  Mortgage            4      Married
22743         1        1600           3  Mortgage            3      Married
22744         1        6500           2  Mortgage            4      Married

[22745 rows x 6 columns]
```

```python
import statsmodels.formula.api as smf

# we break up the formula string to fit on the page
model = smf.logit(
    "ge150k_i ~ HouseCosts + NumWorkers + OwnRent + NumBedrooms
      + FamilyType",
    data=acs_sub,
)

results = model.fit()
```

```
Optimization terminated successfully.
         Current function value: 0.391651
         Iterations 7
```

```python
print(results.summary())
```

```
                           Logit Regression Results
==============================================================================
Dep. Variable:                ge150k_i   No. Observations:                22745
Model:                           Logit   Df Residuals:                    22737
Method:                            MLE   Df Model:                            7
Date:                 Thu, 01 Sep 2022   Pseudo R-squ.:                  0.2078
Time:                         01:57:02   Log-Likelihood:                -8908.1
converged:                        True   LL-Null:                       -11244.
Covariance Type:             nonrobust   LLR p-value:                     0.000
===========================================================================================
                              coef    std err          z      P>|z|      [0.025      0.975]
-------------------------------------------------------------------------------------------
Intercept                  -5.8081      0.120    -48.456      0.000      -6.043      -5.573
OwnRent[T.Outright]         1.8276      0.208      8.782      0.000       1.420       2.236
OwnRent[T.Rented]          -0.8763      0.101     -8.647      0.000      -1.075      -0.678
FamilyType[T.Male Head]     0.2874      0.150      1.913      0.056      -0.007       0.582
FamilyType[T.Married]       1.3877      0.088     15.781      0.000       1.215       1.560
HouseCosts                  0.0007   1.72e-05     42.453      0.000       0.001       0.001
NumWorkers                  0.5873      0.026     22.393      0.000       0.536       0.639
NumBedrooms                 0.2365      0.017     13.985      0.000       0.203       0.270
===========================================================================================
```

```python
import statsmodels.formula.api as smf

# we break up the formula string to fit on the page
model = smf.logit(
    "ge150k_i ~ HouseCosts + NumWorkers + OwnRent + NumBedrooms + FamilyType",
    data=acs_sub,
)

results = model.fit()
```

```
Optimization terminated successfully.
         Current function value: 0.391651
         Iterations 7
```

```python
print(results.summary())
```

```
                           Logit Regression Results
==============================================================================
Dep. Variable:                ge150k_i   No. Observations:                22745
Model:                           Logit   Df Residuals:                    22737
```

```
Model:                          Logit   Df Residuals:                    22737
Method:                           MLE   Df Model:                            7
Date:                Thu, 01 Sep 2022   Pseudo R-squ.:                  0.2078
Time:                        01:57:02   Log-Likelihood:                -8908.1
converged:                       True   LL-Null:                       -11244.
Covariance Type:            nonrobust   LLR p-value:                     0.000

===========================================================================================
                              coef    std err          z      P>|z|      [0.025      0.975]
-------------------------------------------------------------------------------------------
Intercept                  -5.8081      0.120    -48.456      0.000      -6.043      -5.573
OwnRent[T.Outright]         1.8276      0.208      8.782      0.000       1.420       2.236
OwnRent[T.Rented]          -0.8763      0.101     -8.647      0.000      -1.075      -0.678
FamilyType[T.Male Head]     0.2874      0.150      1.913      0.056      -0.007       0.582
FamilyType[T.Married]       1.3877      0.088     15.781      0.000       1.215       1.560
HouseCosts                  0.0007   1.72e-05     42.453      0.000       0.001       0.001
NumWorkers                  0.5873      0.026     22.393      0.000       0.536       0.639
NumBedrooms                 0.2365      0.017     13.985      0.000       0.203       0.270
===========================================================================================
```

```python
import numpy as np

# exponentiate our results
odds_ratios = np.exp(results.params)
print(odds_ratios)
```

```
Intercept                  0.003003
OwnRent[T.Outright]        6.219147
OwnRent[T.Rented]          0.416310
FamilyType[T.Male Head]    1.332901
FamilyType[T.Married]      4.005636
HouseCosts                 1.000731
NumWorkers                 1.799117
NumBedrooms                1.266852
dtype: float64
```

```python
print(acs.OwnRent.unique())
```

```
['Mortgage' 'Rented' 'Outright']
```

```python
predictors = pd.get_dummies(acs_sub.iloc[:, 1:], drop_first=True)
print(predictors)
```

```
       HouseCosts  NumWorkers  NumBedrooms OwnRent_Outright OwnRent_Rented \
0            1800           0            4                0              0
1             850           0            3                0              1
2            2600           1            4                0              0
3            1800           0            2                0              1
4             860           0            3                0              0
...           ...         ...          ...              ...            ...
22740        1700           2            5                0              0
22741        1300           2            4                0              0
22742         410           3            4                0              0
22743        1600           3            3                0              0
22744        6500           2            4                0              0
```

```
       FamilyType_Male Head  FamilyType_Married
0                         0                   1
1                         0                   0
2                         0                   0
3                         0                   0
4                         1                   0
...                     ...                 ...
22740                     0                   1
22741                     0                   1
22742                     0                   1
22743                     0                   1
22744                     0                   1

[22745 rows x 7 columns]
```

```
from sklearn import linear_model
lr = linear_model.LogisticRegression()
```

```
results = lr.fit(X = predictors, y = acs['ge150k_i'])
```

```
/Users/danielchen/.pyenv/versions/3.10.4/envs/pfe_book/lib/python3.10/
site-packages/sklearn/linear_model/_logistic.py:444: ConvergenceWarning:
lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
    regression
  n_iter_i = _check_optimize_result(
```

```python
print(results.coef_)
```

```
[[ 5.83764740e-04  7.29381775e-01  2.82543789e-01  7.03519146e-02
  -2.11748592e+00 -1.02984936e+00  2.50310160e-01]]
```

```python
values = np.append(results.intercept_, results.coef_)

# get the names of the values
names = np.append("intercept", predictors.columns)

# put everything in a labeled dataframe
results = pd.DataFrame(
    values,
    index=names,
    columns=["coef"],  # you need the square brackets here
)

print(results)
```

```
                          coef
intercept            -4.820884
HouseCosts            0.000584
NumWorkers            0.729382
NumBedrooms           0.282544
OwnRent_Outright      0.070352
OwnRent_Rented       -2.117486
FamilyType_Male Head -1.029849
FamilyType_Married    0.250310
```

```
results['or'] = np.exp(results['coef'])
print(results)
```

```
                          coef        or
intercept            -4.820884  0.008060
HouseCosts            0.000584  1.000584
NumWorkers            0.729382  2.073798
NumBedrooms           0.282544  1.326500
OwnRent_Outright      0.070352  1.072886
OwnRent_Rented       -2.117486  0.120334
FamilyType_Male Head -1.029849  0.357061
FamilyType_Married    0.250310  1.284424
```

```r
library(MASS)

library(tidyverse)
library(tidymodels)

library(pscl)

# load the tips data
tips <- readr::read_csv("data/tips.csv")

# load the titanic data
titanic <- readr::read_csv("data/titanic.csv")

# subset the columns and drop missing values
titanic_sub <- titanic %>%
  dplyr::select(survived, sex, age, embarked) %>%
  tidyr::drop_na()


# load the ACS data and fix the data types
acs <- readr::read_csv("data/acs_ny.csv") %>%
  dplyr::mutate( # data gets loaded differently from pandas
    NumChildren = as.integer(NumChildren),
    FamilyIncome = as.numeric(FamilyIncome),
    NumBedrooms = as.numeric(NumBedrooms),
    HouseCosts = as.numeric(HouseCosts),
    ElectricBill = as.numeric(ElectricBill),
    NumVehicles = as.numeric(NumVehicles)
  )
```

```r
r_lm <- lm(tip ~ total_bill, data = tips)
print(summary(r_lm))
```

```
Call:
lm(formula = tip ~ total_bill, data = tips)
```

```
Residuals:
    Min      1Q  Median      3Q     Max 
-3.1982 -0.5652 -0.0974  0.4863  3.7434 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 0.920270   0.159735   5.761 2.53e-08 ***
total_bill  0.105025   0.007365  14.260  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.022 on 242 degrees of freedom
Multiple R-squared:  0.4566,    Adjusted R-squared:  0.4544 
F-statistic: 203.4 on 1 and 242 DF,  p-value: < 2.2e-16
```

```r
r_lm %>%
  broom::tidy()
```

```
# A tibble: 2 x 5
  term        estimate std.error statistic  p.value
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept)    0.920   0.160        5.76 2.53e- 8
2 total_bill     0.105   0.00736     14.3  6.69e-34
```

```r
r_lm2 <- lm(tip ~ total_bill + size, data = tips)
print(summary(r_lm2))
```

```
Call:
lm(formula = tip ~ total_bill + size, data = tips)

Residuals:
    Min      1Q  Median      3Q     Max 
-2.9279 -0.5547 -0.0852  0.5095  4.0425 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 0.668945   0.193609   3.455  0.00065 ***
```

```
total_bill  0.092713   0.009115  10.172  < 2e-16 ***
size        0.192598   0.085315   2.258  0.02487 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.014 on 241 degrees of freedom
Multiple R-squared:  0.4679,	Adjusted R-squared:  0.4635
F-statistic: 105.9 on 2 and 241 DF,  p-value: < 2.2e-16
```

```r
r_lm2 %>%
  broom::tidy()
```

```
# A tibble: 3 x 5
  term        estimate std.error statistic  p.value
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept)   0.669     0.194       3.46 6.50e- 4
2 total_bill    0.0927    0.00911    10.2  1.88e-20
3 size          0.193     0.0853      2.26 2.49e- 2
```

```r
r_lm3 <- lm(
  tip ~ total_bill + size + sex + smoker + day + time, data = tips
)
print(summary(r_lm3))
```

```
Call:
lm(formula = tip ~ total_bill + size + sex + smoker + day + time,
    data = tips)

Residuals:
    Min      1Q  Median      3Q     Max
-2.8475 -0.5729 -0.1026  0.4756  4.1076

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.803817   0.352702   2.279   0.0236 *
total_bill   0.094487   0.009601   9.841   <2e-16 ***
size         0.175992   0.089528   1.966   0.0505 .
sexMale     -0.032441   0.141612  -0.229   0.8190
```

```
smokerYes   -0.086408   0.146587  -0.589   0.5561
daySat      -0.121458   0.309742  -0.392   0.6953
daySun      -0.025481   0.321298  -0.079   0.9369
dayThur     -0.162259   0.393405  -0.412   0.6804
timeLunch    0.068129   0.444617   0.153   0.8783
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.024 on 235 degrees of freedom
Multiple R-squared:  0.4701,    Adjusted R-squared:  0.452
F-statistic: 26.06 on 8 and 235 DF,  p-value: < 2.2e-16
```

```r
r_lm3 %>%
  broom::tidy()
```

```
# A tibble: 9 x 5
  term        estimate std.error statistic  p.value
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept)   0.804    0.353       2.28   2.36e- 2
2 total_bill    0.0945   0.00960     9.84   2.34e-19
3 size          0.176    0.0895      1.97   5.05e- 2
4 sexMale      -0.0324   0.142      -0.229  8.19e- 1
5 smokerYes    -0.0864   0.147      -0.589  5.56e- 1
6 daySat       -0.121    0.310      -0.392  6.95e- 1
7 daySun       -0.0255   0.321      -0.0793 9.37e- 1
8 dayThur      -0.162    0.393      -0.412  6.80e- 1
9 timeLunch     0.0681   0.445       0.153  8.78e- 1
```

```r
# fit a logistic regression model
r_logistic_glm <- glm(
  survived ~ sex + age + embarked,
  family = binomial (link = "logit"),
  data = titanic_sub
)

summary(r_logistic_glm)
```

```
Call:
glm(formula = survived ~ sex + age + embarked, family =
binomial(link = "logit"),      data = titanic_sub)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.1185  -0.6498  -0.5972   0.7937   2.1977

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  2.204585   0.321796   6.851 7.34e-12 ***
sexmale     -2.475962   0.190807 -12.976  < 2e-16 ***
age         -0.008079   0.006550  -1.233  0.21746
embarkedQ   -1.815592   0.535031  -3.393  0.00069 ***
embarkedS   -1.006949   0.236857  -4.251 2.13e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 960.90  on 711  degrees of freedom
Residual deviance: 726.08  on 707  degrees of freedom
AIC: 736.08

Number of Fisher Scoring iterations: 4
```

```r
# get the coefficient table and calculate the odds
res_r_glm <- r_logistic_glm %>%
  broom::tidy() %>%
  dplyr::mutate(odds = exp(estimate) %>% round(6))

res_r_glm
```

```
# A tibble: 5 x 6
  term        estimate std.error statistic  p.value   odds
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>  <dbl>
1 (Intercept)  2.20      0.322        6.85 7.34e-12 9.07
2 sexmale     -2.48      0.191      -13.0  1.67e-38 0.0841
3 age         -0.00808   0.00655     -1.23 2.17e- 1 0.992
4 embarkedQ   -1.82      0.535       -3.39 6.90e- 4 0.163
5 embarkedS   -1.01      0.237       -4.25 2.13e- 5 0.365
```

```r
poi <- glm(
  NumBedrooms ~ HouseCosts + OwnRent,
  family=poisson(link = "log"),
  data=acs
)
```

```r
summary(poi)
```

```
Call:
glm(formula = NumBedrooms ~ HouseCosts + OwnRent,
    family = poisson(link = "log"),  data = acs)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.8300  -0.2815  -0.1293   0.2890   2.8142

Coefficients:
                  Estimate Std. Error z value Pr(>|z|)
(Intercept)      1.139e+00  6.158e-03 184.928  < 2e-16 ***
HouseCosts       6.217e-05  2.958e-06  21.017  < 2e-16 ***
OwnRentOutright -2.659e-01  5.131e-02  -5.182 2.19e-07 ***
OwnRentRented   -1.237e-01  1.237e-02  -9.996  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)
```

```
    Null deviance: 7479.9  on 22744  degrees of freedom
Residual deviance: 6839.2  on 22741  degrees of freedom
AIC: 76477

Number of Fisher Scoring iterations: 4
```

```r
poi %>%
  broom::tidy()
```

```
# A tibble: 4 x 5
  term               estimate  std.error statistic  p.value
  <chr>                 <dbl>      <dbl>     <dbl>    <dbl>
1 (Intercept)      1.14      0.00616       185.   0
2 HouseCosts       0.0000622 0.00000296     21.0  4.60e-98
3 OwnRentOutright -0.266     0.0513         -5.18 2.19e- 7
4 OwnRentRented   -0.124     0.0124        -10.0  1.58e-23
```

```r
od <- MASS::glm.nb(
  NumPeople ~ Acres + NumVehicles,
  data=acs,
  link=log
)
```

```
Warning in theta.ml(Y, mu, sum(w), w, limit = control$maxit, trace
= control$trace > : iteration limit reached

Warning in theta.ml(Y, mu, sum(w), w, limit = control$maxit, trace
= control$trace > : iteration limit reached
```

```r
summary(od)
```

```
Call:
MASS::glm.nb(formula = NumPeople ~ Acres + NumVehicles, data = acs,
    link = log, init.theta = 99662.32096)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.3263  -0.7064  -0.1315   0.3153   5.3101

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  1.033460   0.012036  85.865  < 2e-16 ***
Acres10+    -0.025287   0.019301  -1.310     0.19

AcresSub 1   0.050768   0.009143   5.553 2.81e-08 ***
NumVehicles  0.070067   0.003683  19.023  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for Negative Binomial(99662.32) family taken to be 1)

    Null deviance: 12127  on 22744  degrees of freedom
Residual deviance: 11754  on 22741  degrees of freedom
AIC: 80879
```

```
Number of Fisher Scoring iterations: 1

              Theta:  99662
          Std. Err.:  93669
Warning while fitting theta: iteration limit reached

 2 x log-likelihood:  -80869.33
```

```
od %>%
  broom::tidy()
```

```
# A tibble: 4 x 5
  term        estimate std.error statistic  p.value
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept)   1.03      0.0120      85.9  0
2 Acres10+     -0.0253    0.0193      -1.31 1.90e- 1
3 AcresSub 1    0.0508    0.00914      5.55 2.81e- 8
4 NumVehicles   0.0701    0.00368     19.0  1.10e-80
```

```
pm <- glm(
  NumChildren ~ FamilyIncome + FamilyType + OwnRent,
  family = poisson(link="log"),
  data = acs
)

pchisq(
  2 * (logLik(od) - logLik(pm)),
  df = 1,
  lower.tail = FALSE
)
```

```
'log Lik.' 1 (df=5)
```