

# Mastering Exploratory Analysis with pandas

Build an end-to-end data analysis workflow with Python

**Packt>**

[www.packt.com](http://www.packt.com)

Harish Garg

# **Mastering Exploratory Analysis with pandas**

Build an end-to-end data analysis workflow with Python

Harish Garg



**BIRMINGHAM - MUMBAI**



# Mastering Exploratory Analysis with pandas

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Commissioning Editor:** Pavan Ramchandani

**Acquisition Editor:** Nelson Morris

**Content Development Editor:** Karan Thakkar

**Technical Editor:** Suwarna Patil

**Copy Editor:** Safis Editing

**Project Coordinator:** Nidhi Joshi

**Proofreader:** Safis Editing

**Indexer:** Pratik Shirodkar

**Graphics:** Jisha Chirayil

**Production Coordinator:** Arvindkumar Gupta

First published: September 2018

Production reference: 1290918

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78961-963-8

[www.packtpub.com](http://www.packtpub.com)



[mapt.io](https://mapt.io)

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

# Why subscribe?

- Spend less time learning and more time coding with practical eBooks and videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

# Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.packt.com](http://www.packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub.com](mailto:customercare@packtpub.com) for more details.

At [www.packt.com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.



# Contributors

# About the author

**Harish Garg** is a data analyst, author, and software developer who is really passionate about data science and Python. He is a graduate of Udacity's Data Analyst Nanodegree program. He has 17 years of industry experience in data analysis using Python, developing and testing enterprise and consumer software, managing projects and software teams, and creating training material and tutorials. He also worked for 11 years for Intel Security (previously McAfee, Inc.). He regularly contributes articles and tutorials on data analysis and Python. He is also active in the open data community and is a contributing member of the Data4Democracy open data initiative. He has written data analysis pieces for the Takshashila think tank.

# Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

# Table of Contents

Title Page	
Copyright and Credits	
Mastering Exploratory Analysis with pandas	
Packt Upsell	
Why subscribe?	
Packt.com	
Contributors	
About the author	
Packt is searching for authors like you	
Preface	
Who this book is for	
What this book covers	
To get the most out of this book	
Download the example code files	
Download the color images	
Conventions used	
Get in touch	
Reviews	
1. Working with Different Kinds of Datasets	
Using advanced options while reading data from CSV files	
Importing modules	
Advanced read options	
Manipulating columns, index locations, and names	
Specifying a different row as a header	
Specifying a column as an index	
Choosing a subset of columns to be read	
Handling missing and NA data	
Choosing whether to skip over blank rows	
Data parsing options	
Skipping rows from the footer or end of the file	
Reading the subset of a file or a certain number of rows	
Reading data from Excel files	
Basic Excel read	

- Specifying which sheet should be read
- Reading data from multiple sheets
  - Finding out sheet names
  - Choosing header or column labels
  - No header
  - Skipping rows at the beginning
  - Skipping rows at the end
  - Choosing columns
  - Column names
  - Setting an index while reading data

- Handling missing data while reading

- Reading data from other popular formats

- Reading a JSON file
  - Reading JSON data into pandas
  - Reading HTML data
  - Reading a PICKLE file
  - Reading SQL data
  - Reading data from the clipboard

- Summary

## 2. Data Selection

- Introduction to datasets

- Selecting data from the dataset

- Multi-column selection
  - Dot notation
  - Selecting multiple rows and columns from a pandas DataFrame
  - Selecting a single row and multiple columns
  - Selecting values from a range of rows and all columns

- Sorting a pandas DataFrame

- Filtering rows of a pandas DataFrame

- Applying multiple filter criteria to a pandas DataFrame

- Filtering based on multiple conditions &#x2013; AND
  - Filtering based on multiple conditions &#x2013; OR
  - Filtering using the isin method
    - Using the isin method with multiple conditions

- Using the axis parameter in pandas

- Usage of the axis parameter
  - Axis usage examples

- More examples of the axis keyword

- The axis keyword

- Using string methods in pandas

- Checking for a substring

- Changing the values of a series or column into uppercase

- Changing the values into lowercase

- Finding the length of every value of a column

- Removing white spaces

- Replacing parts of a column's values

- Changing the datatype of a pandas series

- Changing an int datatype column to a float

- Changing the datatype while reading data

- Converting string to datetime

- Summary

### 3. Manipulating, Transforming, and Reshaping Data

- Modifying a pandas DataFrame using the inplace parameter

- Using the groupby method

- Handling missing values in pandas

- Indexing in pandas DataFrames

- Renaming columns in a pandas DataFrame

- Removing columns from a pandas DataFrame

- Working with date and time series data

- Handling SettingWithCopyWarning

- Applying a function to a pandas series or DataFrame

- Merging and concatenating multiple DataFrames into one

- Summary

### 4. Visualizing Data Like a Pro

- Controlling plot aesthetics

- Our first plot with seaborn

- Changing the plot style with set\_style

- Setting the plot background to a white grid

- Setting the plot background to dark

- Setting the background to white

- Adding ticks

- Customizing styles

- Style parameters

- Plotting context presets

Choosing the colors for plots

Changing the color palette

Building custom color palettes

Plotting categorical data

Scatterplot

Swarm plot

Box plot

Violin plot

Bar plot

Wide-form plot

Plotting with Data-Aware Grids

Plotting with the FacetGrid() method

Plotting with the PairGrid() method

Plotting with the PairPlot() method

Summary

Other Books You May Enjoy

Leave a review - let other readers know what you think

# Preface

In this book, you will be learning in depth about pandas, which is a Python library for manipulating, transforming, and analyzing data. It is a popular framework for exploratory data visualization, which is a method for analyzing datasets and data pipelines based on their properties.

This book will be your practical guide to exploring datasets using pandas. You will start by setting up Python, pandas, and Jupyter Notebooks. You will learn how to use Jupyter Notebooks to run Python code. We will then show you how to get data into pandas and perform some exploratory analysis. You will learn how to manipulate and reshape data using pandas methods. You will also learn how to deal with missing data from your datasets, how to draw charts and plots using pandas and Matplotlib, and how to create some effective visualizations for your audience. Finally, we will wrap up your newly gained pandas knowledge by teaching you how to get data out of pandas and into a number of popular file formats.



# Who this book is for

This book is for the budding data scientist looking to learn about the popular pandas library, or the Python developer looking to step into the world of data analysis—if you fall into either of those categories, then this book is the ideal resource for you to get started.

# What this book covers

[Chapter 1](#), *Working with Different Kinds of Datasets*, teaches you about using advanced options when reading data from CSV files and Excel files.

[Chapter 2](#), *Data Selection*, looks at how to use the pandas series data structure to select data. You will also learn how to sort and filter data from pandas DataFrames and how to change datatypes in pandas series.

[Chapter 3](#), *Manipulating, Transforming, and Reshaping Data*, explores how to modify pandas DataFrames. You will also learn how to use the `groupBy` method, how to handle missing values, and how to index methods in pandas DataFrames. This chapter will also teach you how to work with dates and time data and how to apply functions to pandas series or DataFrames.

[Chapter 4](#), *Visualizing Data Like a Pro*, will show you how to control plot aesthetics, including how to choose colors for plots. You will also learn how to plot categorical data and get to grips with plotting with data-aware grids.

# **To get the most out of this book**

Some programming experience in Python would help you get the most out of this course.

# Download the example code files

You can download the example code files for this book from your account at [www.packt.com](http://www.packt.com). If you purchased this book elsewhere, you can visit [www.packt.com/support](http://www.packt.com/support) and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at [www.packt.com](http://www.packt.com).
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Mastering-Exploratory-Analysis-with-pandas>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

# Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: [https://www.packtpub.com/sites/default/files/downloads/9781789619638\\_ColorImages.pdf](https://www.packtpub.com/sites/default/files/downloads/9781789619638_ColorImages.pdf).

# Conventions used

There are a number of text conventions used throughout this book.

**CodeInText:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "How to use advanced options of the `read_excel` method."

A block of code is set as follows:

```
|df = pd.read_csv('IMDB.csv', encoding = "ISO-8859-1")  
|df.head()
```

Any command-line input or output is written as follows:

```
|conda install sqlite
```

**Bold:** Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select System info from the Administration panel."



*Warnings or important notes appear like this.*



*Tips and tricks appear like this.*

# Get in touch

Feedback from our readers is always welcome.

**General feedback:** If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at [customercare@packtpub.com](mailto:customercare@packtpub.com).

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packt.com/submit-errata](http://www.packt.com/submit-errata), selecting your book, clicking on the Errata Submission Form link, and entering the details.

**Piracy:** If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packt.com](mailto:copyright@packt.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

# Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit [packt.com](https://packt.com).



# Working with Different Kinds of Datasets

In this chapter, we will learn how to work with different kinds of dataset formats in pandas. We'll learn how to use the advanced options provided by pandas' imported CSV files. We will also look at how to work with Excel files in pandas, and how to use advanced options of the `read_excel` method. We'll explore some of the other pandas methods for working with popular data formats, such as HTML, JSON, PICKLE files, SQL, and so on.

# Using advanced options while reading data from CSV files

In this section, we will be working with CSV datasets and learn how to import CSV datasets as well as advanced options for pandas: the `read_csv` method.

# Importing modules

First, we will start by importing the `pandas` module with the following command:

```
|import pandas as pd
```

To read the CSV files, we use the `read_csv` method, as follows:

```
|df = pd.read_csv('IMDB.csv', encoding = "ISO-8859-1")  
|df.head()
```

In order to perform a basic import, pass the filename of the dataset to `read_csv`, and assign the resulting DataFrame to a variable. In the following screenshot, we can see that pandas has turned the dataset into a tabular format:

	X	Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	...	Votes45A	Votes45AM	Votes45AF	VotesIMDB
0	1	12 Years a Slave (2013)	8.1	496,092	Biography	Drama	History	96.0	20,000,000	134 min	...	7.8	7.8	8.1	8.0
1	2	127 Hours (2010)	7.6	297,075	Adventure	Biography	Drama	82.0	18,000,000	94 min	...	7.3	7.3	7.5	7.6
2	3	50/50 (2011)	7.7	283,935	Comedy	Drama	Romance	72.0	8,000,000	100 min	...	7.4	7.4	7.5	7.4
3	4	About Time (2013)	7.8	225,412	Comedy	Drama	Fantasy	NaN	12,000,000	123 min	...	7.6	7.5	7.8	7.7
4	5	Amour (2012)	7.9	76,121	Drama	Romance	NaN	94.0	8,900,000	127 min	...	7.9	7.8	8.1	6.6

5 rows × 16 columns

# Advanced read options

In Python, pandas has a lot of advanced options for the `read_csv` method, which is where you can control how the data is read from a CSV file.

# Manipulating columns, index locations, and names

By default, `read_csv` considers the entries in the first row of the CSV file as column names. We can turn this off by setting `header` to `None`, as shown in the following code block:

```
df = pd.read_csv('IMDB.csv', encoding = "ISO-8859-1", header=None)
df.head()
```

The output is as follows:

	0	1	2	3	4	5	6	7	8	9	...	48	49	50	51
0	X	Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	...	Votes45A	Votes45AM	Votes45AF	VotesIMDB
1	1	12 Years a Slave (2013)	8.1	496,092	Biography	Drama	History	96	20,000,000	134 min	...	7.8	7.8	8.1	8.0
2	2	127 Hours (2010)	7.6	297,075	Adventure	Biography	Drama	82	18,000,000	94 min	...	7.3	7.3	7.5	7.6
3	3	50/50 (2011)	7.7	283,935	Comedy	Drama	Romance	72	8,000,000	100 min	...	7.4	7.4	7.5	7.4
4	4	About Time (2013)	7.8	225,412	Comedy	Drama	Fantasy	NaN	12,000,000	123 min	...	7.6	7.5	7.8	7.7

5 rows x 58 columns

# Specifying a different row as a header

You can also set the column name from a different row—instead of the default first row—by passing the row number to the `header` option, as follows:

```
df = pd.read_csv('IMDB.csv', encoding = "ISO-8859-1", header=2)
df.head()
```

The output is as follows:

	2	127 Hours (2010)	7.6	297,075	Adventure	Biography	Drama	82	18,000,000	94 min	...	7.3	7.3.1	7.5.3	7.6.2	7.0	7.7.2	7.6.3	\$18,335,230
0	3	50/50 (2011)	7.7	283,935	Comedy	Drama	Romance	72.0	8,000,000	100 min	...	7.4	7.4	7.5	7.4	7.0	7.9	7.6	\$35,014,192
1	4	About Time (2013)	7.8	225,412	Comedy	Drama	Fantasy	NaN	12,000,000	123 min	...	7.6	7.5	7.8	7.7	6.9	7.8	7.7	\$15,322,921
2	5	Amour (2012)	7.9	76,121	Drama	Romance	NaN	94.0	8,900,000	127 min	...	7.9	7.8	8.1	6.6	7.2	7.9	7.8	\$6,739,492
3	6	Argo (2012)	7.7	486,840	Action	Biography	Drama	86.0	44,500,000	120 min	...	7.7	7.7	8.0	8.1	7.2	8.0	7.6	\$136,025,503
4	7	Arrival (2016)	8.0	370,842	Drama	Mystery	Sci-Fi	81.0	47,000,000	116 min	...	7.6	7.6	7.7	8.3	7.3	8.0	7.9	\$100,546,139

5 rows x 58 columns

# Specifying a column as an index

By default, `read_csv` assigns a default numeric index starting with zero while reading the data. However, you can change this behavior by passing the column name to the `index_col` option. `pandas` will then set the index to this column, as shown in the following code:

```
df = pd.read_csv('IMDB.csv', encoding = "ISO-8859-1", index_col='Title')
df.head()
```

Here, we passed a movie title as the index name. Now the index name is `Title`, instead of a default numeric index, as shown in the following screenshot:

	X	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	CVotes10	...	Votes45A	Votes45AM	Votes45AF
Title														
12 Years a Slave (2013)	1	8.1	496,092	Biography	Drama	History	96.0	20,000,000	134 min	75556	...	7.8	7.8	8.1
127 Hours (2010)	2	7.6	297,075	Adventure	Biography	Drama	82.0	18,000,000	94 min	28939	...	7.3	7.3	7.5
50/50 (2011)	3	7.7	283,935	Comedy	Drama	Romance	72.0	8,000,000	100 min	28304	...	7.4	7.4	7.5
About Time (2013)	4	7.8	225,412	Comedy	Drama	Fantasy	NaN	12,000,000	123 min	38556	...	7.6	7.5	7.8
Amour (2012)	5	7.9	76,121	Drama	Romance	NaN	94.0	8,900,000	127 min	11093	...	7.9	7.8	8.1

5 rows × 57 columns

# Choosing a subset of columns to be read

We can also choose to read a specific subset of columns in the CSV file. For this, we pass the column names as a list to use the columns option, as follows:

```
df = pd.read_csv('IMDB.csv', encoding = "ISO-8859-1", usecols=['Title', 'Genre1'])  
df.head()
```

Output of the preceding code snippet is as follows:

	Title	Genre1
0	12 Years a Slave (2013)	Biography
1	127 Hours (2010)	Adventure
2	50/50 (2011)	Comedy
3	About Time (2013)	Comedy
4	Amour (2012)	Drama



# Handling missing and NA data

Next, we will see how to handle missing data by reading a CSV file. By default, `read_csv` considers the following values missing and marks them as `NaN`:

```
NaN: ", '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', 'N/A', 'NA', 'NULL', 'NaN', 'nan'.
```

However, you can add to this list. To do so, simply pass the list of values you want to be considered as `NaN`, as shown in the following code:

```
|df = pd.read_csv('IMDB.csv', encoding = "ISO-8859-1", na_values=[' '])
```

# Choosing whether to skip over blank rows

Sometimes whole rows have no values; hence, we can choose what to do with these rows while reading data. By default, `read_csv` ignores blank rows, but we can turn this off by setting `skip_blank_lines` to `False`, as follows:

```
|df = pd.read_csv('IMDB.csv', encoding = "ISO-8859-1", skip_blank_lines=False)
```

# Data parsing options

We can choose which rows are skipped by reading a CSV file. We can pass the row numbers as a list to the `skiprows` option. The first row has the index zero, as follows:

```
df = pd.read_csv('IMDB.csv', encoding = "ISO-8859-1", skiprows = [1,3,7])
df.head()
```

The output is as follows:

	X	Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	...	Votes45A	Votes45AM	Votes45AF	VotesIMDB
0	2	127 Hours (2010)	7.6	297,075	Adventure	Biography	Drama	82.0	18,000,000	94 min	...	7.3	7.3	7.5	7.6
1	4	About Time (2013)	7.8	225,412	Comedy	Drama	Fantasy	NaN	12,000,000	123 min	...	7.6	7.5	7.8	7.7
2	5	Amour (2012)	7.9	76,121	Drama	Romance	NaN	94.0	8,900,000	127 min	...	7.9	7.8	8.1	6.6
3	6	Argo (2012)	7.7	486,840	Action	Biography	Drama	86.0	44,500,000	120 min	...	7.7	7.7	8.0	8.1
4	9	Before Midnight (2013)	7.9	106,553	Drama	Romance	NaN	94.0	3,000,000	109 min	...	7.3	7.4	7.2	8.5

5 rows x 58 columns

# Skipping rows from the footer or end of the file

To skip rows from the footer or the end of a file, use the `skipfooter` option and pass a number specifying the number of rows to skip. In the following code, we have passed 2. As we can see, there is a difference between the previous DataFrame we created and the DataFrame we created after skipping the last two rows:

```
df.tail(2)
df = pd.read_csv('IMDB.csv', encoding = "ISO-8859-1", skipfooter=2,
engine='python')
df.tail(2)
```

The following screenshot shows the output:

X		Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	...	Votes45A	Votes45AM	Votes45AF	VotesIM
112	117	X-Men: First Class (2011)	7.8	556,713	Action	Adventure	Sci-Fi	65.0	160,000,000	132 min	...	7.6	7.5	7.7	
113	118	Zootopia (2016)	8.1	309,474	Animation	Adventure	Comedy	78.0	150,000,000	108 min	...	7.8	7.8	8.1	

2 rows × 58 columns

X		Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	...	Votes45A	Votes45AM	Votes45AF	VotesIMDE
113	115	Wreck-It Ralph (2012)	7.7	295,125	Animation	Adventure	Comedy	72.0	165,000,000	NaN	...	7.4	7.4	7.5	7.4
114	116	X-Men: Days of Future Past (2014)	8.0	560,736	Action	Adventure	Sci-Fi	74.0	200,000,000	132 min	...	7.7	7.7	7.9	7.8

2 rows × 58 columns

# Reading the subset of a file or a certain number of rows

Sometimes a data file is too big, and we just want to have a peek at the first few rows. We can do that by passing the number of rows to import to the `nrows` option, as shown in the following code. Here, we passed `100` to `nrows`, which then read only the first hundred rows from the dataset:

```
| df = pd.read_csv('IMDB.csv', encoding = "ISO-8859-1", nrows=100)  
| df.shape
```

# Reading data from Excel files

In this section, we will learn how to work with Excel data using pandas and use pandas' `read_excel` method for reading data from Excel files. We will read and explore a real Excel dataset and explore some of the advanced options available for parsing Excel data.



*pandas internally uses the `Excel rd` Python library to extract data from Excel files. We can install it by executing `conda install xlrd`.*

Firstly, make sure the command-line program is running in admin mode ahead of installation, as shown in the following screenshot:

```
(C:\ProgramData\Anaconda3) C:\>conda install xlrd
Fetching package metadata .....
Solving package specifications: .

# All requested packages already installed.
# packages in environment at C:\ProgramData\Anaconda3:
#
xlrd                1.0.0                py36_0

(C:\ProgramData\Anaconda3) C:\>
```

The following screenshot shows the Excel dataset that we will be reading and exploring with pandas:

A	B	C	D	E	F	G	H	I	J
X	Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime
1	12 Years a Slave (2013)	8.1	496092	Biography	Drama	History	96	20000000	134 min
2	127 Hours (2010)	7.6	297075	Adventure	Biography	Drama	82	18000000	94 min
3	50/50 (2011)	7.7	283935	Comedy	Drama	Romance	72	8000000	100 min
4	About Time (2013)	7.8	225412	Comedy	Drama	Fantasy	NA	12000000	123 min
5	Amour (2012)	7.9	76121	Drama	Romance		94	8900000	127 min
6	Argo (2012)	7.7	486840	Action	Biography	Drama	86	44500000	120 min
7	Arrival (2016)	8	370842	Drama	Mystery	Sci-Fi	81	47000000	116 min
9	Before Midnight (2013)	7.9	106553	Drama	Romance		94	3000000	109 min
10	Big Hero 6 (2014)	7.8	315485	Animation	Action	Adventure	74	165000000	
11	Birdman or (The Unexpected Virtue of Ignorance) (2014)	7.8	448725	Comedy	Drama		88	18000000	119 min
12	Black Swan (2010)	8	587893	Drama	Thriller		79	13000000	108 min
13	Boyhood (2014)	7.9	290327	Drama			100	4000000	165 min
14	Bridge of Spies (2015)	7.6	223756	Drama	History	Thriller	81	40000000	142 min
15	Captain America: Civil War (2016)	7.9	431555	Action	Adventure	Sci-Fi	75	250000000	147 min
16	Captain America: The Winter Soldier (2014)	7.8	552706	Action	Adventure	Sci-Fi	70	170000000	136 min
17	Captain Fantastic (2016)	7.9	115194	Comedy	Drama		72	5000000	
18	Captain Phillips (2013)	7.8	350818	Biography	Drama	Thriller	83	55000000	134 min
19	Creed (2015)	7.6	179795	Drama	Sport		82	35000000	133 min
20	Dallas Buyers Club (2013)	8	357641	Biography	Drama		84	5000000	117 min
21	Dawn of the Planet of the Apes (2014)	7.6	349646	Action	Adventure	Drama	79	170000000	130 min
22	Deadpool (2016)	8	652127	Action	Adventure	Comedy	65	58000000	108 min
23	Despicable Me (2010)	7.7	417592	Animation	Adventure	Comedy	72	69000000	
24	Detachment (2011)	7.7	62352	Drama			NA		
25	Disconnect (2012)	7.6	65448	Drama	Thriller		64	10000000	115 min
26	Django Unchained (2012)	8.4	1056822	Drama	Western		81	100000000	165 min
27	Doctor Strange (2016)	7.6	328932	Action	Adventure	Fantasy	72	165000000	115 min
28	Drive (2011)	7.8	467642	Crime	Drama		78	15000000	100 min
29	Edge of Tomorrow (2014)	7.9	480513	Action	Adventure	Sci-Fi	71	178000000	113 min
30	End of Watch (2012)	7.7	194675	Crime	Drama	Thriller	68	7000000	109 min
31	Ex Machina (2014)	7.7	348550	Drama	Mystery	Sci-Fi	78	15000000	108 min



Previous screenshot is collection of Movie rating which can be found at: <https://github.com/saipranava/IMDB>.

# Basic Excel read

We are using pandas' `read_excel` method to read this data. In its simplest format, we are just passing the filename of the Excel dataset we want to the `read_excel` method. pandas converts the data from the Excel file into a pandas `DataFrame`. The pandas internally uses the Excel `rd` library for this. Here, pandas has read the data and created a tabular data object in the memory, which we can access, explore, and manipulate in our code, as shown in the following code:

```
df = pd.read_excel('IMDB.xlsx')
df.head()
```

The output for the previous code block is as follows:

	X	Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	...	Votes45A	Votes45AM	Votes45AF	VotesIMDB
0	1	12 Years a Slave (2013)	8.1	496092	Biography	Drama	History	96.0	20000000.0	134 min	...	7.8	7.8	8.1	8.0
1	2	127 Hours (2010)	7.6	297075	Adventure	Biography	Drama	82.0	18000000.0	94 min	...	7.3	7.3	7.5	7.6
2	3	50/50 (2011)	7.7	283935	Comedy	Drama	Romance	72.0	8000000.0	100 min	...	7.4	7.4	7.5	7.4
3	4	About Time (2013)	7.8	225412	Comedy	Drama	Fantasy	NaN	12000000.0	123 min	...	7.6	7.5	7.8	7.7
4	5	Amour (2012)	7.9	76121	Drama	Romance	NaN	94.0	8900000.0	127 min	...	7.9	7.8	8.1	6.6

5 rows x 58 columns

pandas has a lot of advanced options, which we can use to control how data should be read, as shown in the following screenshot:

```
pandas.read_excel(io, sheetname=0, header=0, skiprows=None, skip_footer=0, index_col=None, names=None, parse_cols=None, parse_dates=False, date_parser=None, na_values=None, thousands=None, convert_float=True, has_index_names=None, converters=None, dtype=None, true_values=None, false_values=None, engine=None, squeeze=False, **kwargs)
```

**\*\*from [Pandas Doc](#)**

# Specifying which sheet should be read

To specify which sheet should be read, pass the value to the `sheetname` option. As you can see in the following screenshot, we are simply passing `0`, which is the index value of the first sheet in the Excel sheet. This is quite handy, especially when we don't know the exact sheet name:

```
df = pd.read_excel('IMDB.xlsx', sheetname=0)
df.head()
```

The output is as follows:

	X	Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	...	Votes45A	Votes45AM	Votes45AF	VotesIMDB
0	1	12 Years a Slave (2013)	8.1	496092	Biography	Drama	History	96.0	20000000.0	134 min	...	7.8	7.8	8.1	8.0
1	2	127 Hours (2010)	7.6	297075	Adventure	Biography	Drama	82.0	18000000.0	94 min	...	7.3	7.3	7.5	7.6
2	3	50/50 (2011)	7.7	283935	Comedy	Drama	Romance	72.0	8000000.0	100 min	...	7.4	7.4	7.5	7.4
3	4	About Time (2013)	7.8	225412	Comedy	Drama	Fantasy	NaN	12000000.0	123 min	...	7.6	7.5	7.8	7.7
4	5	Amour (2012)	7.9	76121	Drama	Romance	NaN	94.0	8900000.0	127 min	...	7.9	7.8	8.1	6.6

5 rows x 58 columns

# Reading data from multiple sheets

Excel dataset files come with data and multiple sheets. In fact, this is one of the main reasons a lot of users prefer Excel over CSV. Luckily, pandas supports the reading of data from multiple sheets.

# Finding out sheet names

To find out the name of a sheet, pass the Excel file to the `ExcelFile` class and call the `sheet_names` property on the resulting object. The class prints the sheet names from the Excel file as a list. If we want to read data from a sheet named `data-movies`, it will look like the following code snippet:

```
|xls_file = pd.ExcelFile('IMDB.xlsx')  
|xls_file.sheet_names
```

Next, we call the `parse` method on the Excel file object we created earlier, and pass in the sheet names we want to read. We then assign the result to two separate `DataFrame` objects, as follows:

```
|df1 = xls_file.parse('movies')  
|df2 = xls_file.parse('by genre')  
|df1.head()
```

We now have our data from two sheets in two separate `DataFrames`, as shown in the following screenshot:

X	Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	...	Votes45A	Votes45AM	Votes45AF	VotesIMDB
0 1	12 Years a Slave (2013)	8.1	496092	Biography	Drama	History	96.0	20000000.0	134 min	...	7.8	7.8	8.1	8.0
1 2	127 Hours (2010)	7.6	297075	Adventure	Biography	Drama	82.0	18000000.0	94 min	...	7.3	7.3	7.5	7.6
2 3	50/50 (2011)	7.7	283935	Comedy	Drama	Romance	72.0	8000000.0	100 min	...	7.4	7.4	7.5	7.4

X	Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	...	Votes45A	Votes45AM	Votes45AF	VotesIMDB
0 1	12 Years a Slave (2013)	8.1	496092	Biography	Drama	History	96.0	20000000.0	134 min	...	7.8	7.8	8.1	8.0
1 2	127 Hours (2010)	7.6	297075	Adventure	Biography	Drama	82.0	18000000.0	94 min	...	7.3	7.3	7.5	7.6
2 3	50/50 (2011)	7.7	283935	Comedy	Drama	Romance	72.0	8000000.0	100 min	...	7.4	7.4	7.5	7.4

# Choosing header or column labels

pandas will, by default, set the column names or header to the values from the first non-blank row in the Excel file. However, we can change this behavior. In the following screenshot, we are passing the value 3 to the `header` option, which tells the `read_excel` method to set the header names from index row 3:

```
df = pd.read_excel('IMDB.xlsx', sheetname=1, header=3)
df.head()
```

The output of the preceding code is as follows:

	3	50/50 (2011)	7.7	283935	Comedy	Drama	Romance	72	8000000	100 min	...	7.4	7.4.1	7.5	7.4.2	7	7.9.3	7.6.3	\$35,014,192	4173591
0	4	About Time (2013)	7.8	225412	Comedy	Drama	Fantasy	NaN	12000000.0	123 min	...	7.6	7.5	7.8	7.7	6.9	7.8	7.7	\$15,322,921	71777528
1	5	Amour (2012)	7.9	76121	Drama	Romance	NaN	94.0	8900000.0	127 min	...	7.9	7.8	8.1	6.6	7.2	7.9	7.8	\$6,739,492	13100000
2	6	Argo (2012)	7.7	486840	Action	Biography	Drama	86.0	44500000.0	120 min	...	7.7	7.7	8.0	8.1	7.2	8.0	7.6	\$136,025,503	96300000
3	7	Arrival (2016)	8.0	370842	Drama	Mystery	Sci-Fi	81.0	47000000.0	116 min	...	7.6	7.6	7.7	8.3	7.3	8.0	7.9	\$100,546,139	102842047
4	9	Before Midnight (2013)	7.9	106553	Drama	Romance	NaN	94.0	3000000.0	109 min	...	7.3	7.4	7.2	8.5	7.0	8.0	7.9	\$8,114,627	3061842

5 rows x 58 columns



# No header

We can also tell `read_excel` to ignore the header and treat all rows as records. This is handy whenever Excel has no header row. To achieve this, we set `header` to `None`, as shown in the following code:

```
df = pd.read_excel('IMDB.xlsx', sheetname=1, header=None)
df.head()
```

The output is as follows:

	0	1	2	3	4	5	6	7	8	9	...	48	49	50	51
0	X	Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	...	Votes45A	Votes45AM	Votes45AF	VotesIMDB
1	1	12 Years a Slave (2013)	8.1	496092	Biography	Drama	History	96	20000000	134 min	...	7.8	7.8	8.1	8
2	2	127 Hours (2010)	7.6	297075	Adventure	Biography	Drama	82	18000000	94 min	...	7.3	7.3	7.5	7.6
3	3	50/50 (2011)	7.7	283935	Comedy	Drama	Romance	72	8000000	100 min	...	7.4	7.4	7.5	7.4
4	4	About Time (2013)	7.8	225412	Comedy	Drama	Fantasy	NaN	12000000	123 min	...	7.6	7.5	7.8	7.7

5 rows x 58 columns

# Skipping rows at the beginning

To skip rows at the beginning of a file, we simply set `skiprows` to the number of rows we want to skip, as shown in the following code:

```
| df = pd.read_excel('IMDB.xlsx', sheetname=1, skiprows=7)
```

# Skipping rows at the end

For this, we use the `skip_footer` option, as follows:

```
|df = pd.read_excel('IMDB.xlsx', sheetname=1, skip_footer=10)
```

# Choosing columns

We can also choose to read only a subset of columns. This is done by setting the `parse_cols` option to a numeric value, which will lead to columns being read from 0 to whichever index we set the parse columns value to. We set `parse_cols=2` in this instance, which reads the first three columns in the Excel file, as shown in the following code snippet:

```
df = pd.read_excel('IMDB.xlsx', sheetname= 0, parse_cols=2)
df.head()
```

The following is the output:

	X	Title	Rating
0	1	12 Years a Slave (2013)	8.1
1	2	127 Hours (2010)	7.6
2	3	50/50 (2011)	7.7
3	4	About Time (2013)	7.8
4	5	Amour (2012)	7.9

# Column names

We can choose to give different names to columns instead of the default names given in the header row. To do this, we pass the list of column names to the `names` parameter, as follows:

```
df = pd.read_excel('IMDB.xlsx', sheetname=0, parse_cols = 2, names=['X', 'Title',  
    'Rating'], )  
df.head()
```

In the following screenshot, we have set column names to the names we passed while reading:

	X	Title	Rating
0	1	12 Years a Slave (2013)	8.1
1	2	127 Hours (2010)	7.6
2	3	50/50 (2011)	7.7
3	4	About Time (2013)	7.8
4	5	Amour (2012)	7.9

# Setting an index while reading data

By default, `read_excel` labels zeros with the numeric index, starting with 0. We can set our index or row labels to a higher value or to our choosing. To do this, we pass a column name from our dataset to the `index_col` option. In the following code, we are setting our index to the `Title` column:

```
df = pd.read_excel('IMDB.xlsx', sheetname=0, index_col='Title')
df.head()
```

The output is as follows:

	X	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	CVotes10	...	Votes45A	Votes45AM	Votes45AF
Title														
12 Years a Slave (2013)	1	8.1	496092	Biography	Drama	History	96.0	20000000.0	134 min	75556	...	7.8	7.8	8.1
127 Hours (2010)	2	7.6	297075	Adventure	Biography	Drama	82.0	18000000.0	94 min	28939	...	7.3	7.3	7.5
50/50 (2011)	3	7.7	283935	Comedy	Drama	Romance	72.0	8000000.0	100 min	28304	...	7.4	7.4	7.5
About Time (2013)	4	7.8	225412	Comedy	Drama	Fantasy	NaN	12000000.0	123 min	38556	...	7.6	7.5	7.8
Amour (2012)	5	7.9	76121	Drama	Romance	NaN	94.0	8900000.0	127 min	11093	...	7.9	7.8	8.1

# Handling missing data while reading

The `read_excel` method has a list of values that it will consider as missing and will then set the values to `NaN`. We can add this when passing a list of values by using the `na_values` parameter, as shown in the following code:

```
|df = pd.read_excel('IMDB.xlsx', sheetname= 0, na_values=[' '])
```

# Reading data from other popular formats

In this section, we will explore pandas' features for reading and working with various popular data formats. We will also learn how to read data from the JSON format, HTML files, and the PICKLE dataset and how to read data from an SQL-based database.



# Reading a JSON file

JSON is a minimal readable format for structuring data. It is used primarily to transmit data between a server and web application as an alternative to XML, as shown in the following screenshot:

```
[
{
  "X": 1,
  "Title": "12 Years a Slave(2013) ",
  "Rating": 8.1,
  "TotalVotes": 496092,
  "Genre1": "Biography",
  "Genre2": "Drama",
  "Genre3": "History",
  "MetaCritic": 96,
  "Budget": 20000000,
  "Runtime": "134 min",
  "CVotes10": 75556,
  "CVotes09": 126223,
  "CVotes08": 161460,
  "CVotes07": 83070,
  "CVotes06": 27231,
  "CVotes05": 9603,
  "CVotes04": 4021,
  "CVotes03": 2420,
  "CVotes02": 1785,
  "CVotes01": 4739,
  "CVotesMale": 313823,
  "CVotesFemale": 82012,
  "CVotesU18": 1837,
  "CVotesU18M": 1363,
  "CVotesU18F": 457,
  "CVotes1829": 200910,
  "CVotes1829M": 153669,
  "CVotes1829F": 45301,
  "CVotes3044": 138762,
  "CVotes3044M": 112943,
  "CVotes3044F": 23895,
  "CVotes45A": 29252,
  "CVotes45AM": 23072,
  "CVotes45AF": 5726,
  "CVotes1000": 664,
  "CVotesUS": 53328,
  "CVotesnUS": 224519,
  "VotesM": 8.1,
  "VotesF": 8.1,
  "VotesU18": 8.4,
  "VotesU18M": 8.4,
  "VotesU18F": 8.5,
  "Votes1829": 8.2,
  "Votes1829M": 8.2.
}
```



# Reading JSON data into pandas

To read JSON data, pandas provides a method called `read_json`, where we pass the filename and location of the JSON data file we want to read. The file location can be local, or even on the internet with a valid URL scheme. We assign the resulting DataFrame to the variable `df`.

The `read_json` method reads the JSON data and converts it into a pandas DataFrame object, a tabular data format, as shown in the following code. The JSON data, which is now easily accessible in a DataFrame format, can be manipulated and explored with greater ease:

```
movies_json = pd.read_json('IMDB.json')
movies_json.head()
```

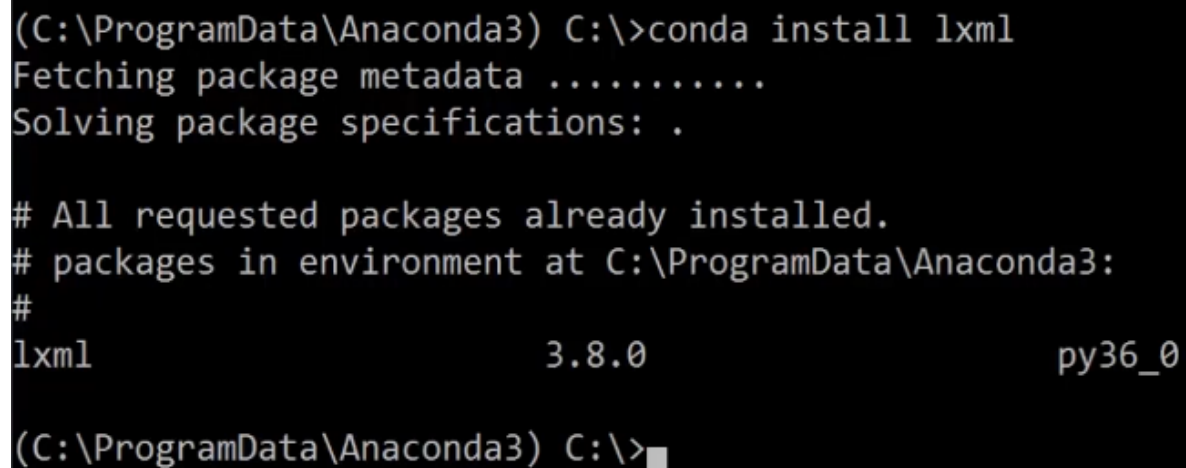
The previous code block will produce the following output:

	Budget	CVotes01	CVotes02	CVotes03	CVotes04	CVotes05	CVotes06	CVotes07	CVotes08	CVotes09	...	VotesF	VotesIMDB	VotesM	VotesU18
0	20000000	4739	1785	2420	4021	9603	27231	83070	161460	126223	...	8.1	8.0	8.1	8.4
1	18000000	2059	1161	1930	3796	9403	28394	78451	98845	44110	...	7.6	7.6	7.6	7.9
2	8000000	1202	634	1109	2381	7545	24252	71485	99524	47501	...	7.7	7.4	7.7	7.9
3	12000000	1182	664	1084	2210	5673	16542	45487	70850	43170	...	7.9	7.7	7.8	8.2
4	8900000	995	534	710	1188	2585	5945	14187	22942	15944	...	7.9	6.6	7.8	8.6

5 rows × 58 columns

# Reading HTML data

pandas uses the `lxml` Python module internally to read HTML data. You can install it from the command-line program by executing `conda install lxml`, as shown in the following screenshot:



```
(C:\ProgramData\Anaconda3) C:\>conda install lxml
Fetching package metadata .....
Solving package specifications: .

# All requested packages already installed.
# packages in environment at C:\ProgramData\Anaconda3:
#
lxml                3.8.0                py36_0

(C:\ProgramData\Anaconda3) C:\>■
```

We can also import HTML data from local files, or even directly from the internet, as well:

X	Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget
1	12 Years a Slave (2013)	8.1	496	092	Biography	Drama	History	96
2	127 Hours (2010)	7.6	297	075	Adventure	Biography	Drama	82
3	50/50 (2011)	7.7	283	935	Comedy	Drama	Romance	72
4	About Time (2013)	7.8	225	412	Comedy	Drama	Fantasy	NA
5	Amour (2012)	7.9	76	121	Drama	Romance		94
6	Argo (2012)	7.7	486	840	Action	Biography	Drama	86
7	Arrival (2016)	8	370	842	Drama	Mystery	Sci-Fi	81
9	Before Midnight (2013)	7.9	106	553	Drama	Romance		94
10	Big Hero 6 (2014)	7.8	315	485	Animation	Action	Adventure	74
11	Birdman or (The Unexpected Virtue of Ignorance) (2014)	7.8	448	725	Comedy	Drama		88
12	Black Swan (2010)	8	587	893	Drama	Thriller		79
13	Boyhood (2014)	7.9	290	327	Drama			100
14	Bridge of Spies (2015)	7.6	223	756	Drama	History	Thriller	81
15	Captain America: Civil War (2016)	7.9	431	555	Action	Adventure	Sci-Fi	75
16	Captain America: The Winter Soldier (2014)	7.8	552	706	Action	Adventure	Sci-Fi	70

Here, we pass in the location of the HTML file, or the URL, to the `read_html` method. `read_html` extracts the tabular data from HTML, and then converts it into a pandas DataFrame . In the following code, we have the data we extracted from the HTML file in a tabular format:

```
|pd.read_html('IMDB.html')
```

The output is as follows:

	0	1	2	\
		Title	Rating	
0	X			
1	1	12 Years a Slave(2013)	8.1	
2	2	127 Hours (2010)	7.6	
3	3	50/50 (2011)	7.7	
4	4	About Time (2013)	7.8	
5	5	Amour (2012)	7.9	
6	6	Argo (2012)	7.7	
7	7	Arrival (2016)	8	
8	9	Before Midnight (2013)	7.9	
9	10	Big Hero 6 (2014)	7.8	
10	11	Birdman or (The Unexpected Virtue of Ignorance...	7.8	
11	12	Black Swan (2010)	8	
12	13	Boyhood (2014)	7.9	
13	14	Bridge of Spies (2015)	7.6	
14	15	Captain America: Civil War (2016)	7.9	
15	16	Captain America: The Winter Soldier (2014)	7.8	
16	17	Captain Fantastic (2016)	7.9	
17	18	Captain Phillips (2013)	7.8	
18	19	Creed (2015)	7.6	

# Reading a PICKLE file

Pickling is a way to convert a Python object of any type, including a list, dictionary, and so on, into a character string. The idea is that this character string contains all the information necessary for reconstructing the object in another Python script.

We use `read_pickle` method to read our PICKLE file, as shown in the following code. As with other data formats, pandas creates a DataFrame from the read data:

```
df = pd.read_pickle('IMDB.p')
df.head()
```

The output is as follows:

X		Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	...	Votes45A	Votes45AM	Votes45AF	VotesIMDB
0	1	12 Years a Slave (2013)	8.1	496092	Biography	Drama	History	96.0	20000000.0	134 min	...	7.8	7.8	8.1	8.0
1	2	127 Hours (2010)	7.6	297075	Adventure	Biography	Drama	82.0	18000000.0	94 min	...	7.3	7.3	7.5	7.6
2	3	50/50 (2011)	7.7	283935	Comedy	Drama	Romance	72.0	8000000.0	100 min	...	7.4	7.4	7.5	7.4
3	4	About Time (2013)	7.8	225412	Comedy	Drama	Fantasy	NaN	12000000.0	123 min	...	7.6	7.5	7.8	7.7
4	5	Amour (2012)	7.9	76121	Drama	Romance	NaN	94.0	8900000.0	127 min	...	7.9	7.8	8.1	6.6

5 rows x 58 columns



# Reading SQL data

Here we will be reading SQL data from the popular database browser that is SQLite which can be installed by executing the following command:

```
|conda install sqlite
```

Then we will import the SQLite Python module as follows:

```
|import sqlite3
```

Then, create a connection to the SQLite DB you want to read data from, as follows:

```
|conn = sqlite3.connect("IMDB.sqlite")
|df = pd.read_sql_query("SELECT * FROM IMDB;", conn)
|df.head()
```

Next, pass the SQL query you want the data from to pandas with the `read_sql_query` method. The method reads the data and creates a `DataFrame` object, as shown in the following screenshot:

	X	Title	Rating	TotalVotes	Genre1	Genre2	Genre3	MetaCritic	Budget	Runtime	...	Votes45A	Votes45AM	Votes45AF	VotesIMDB
0	1	12 Years a Slave (2013)	8.1	496092	Biography	Drama	History	96	20000000	134 min	...	7.8	7.8	8.1	8.0
1	2	127 Hours (2010)	7.6	297075	Adventure	Biography	Drama	82	18000000	94 min	...	7.3	7.3	7.5	7.6
2	3	50/50 (2011)	7.7	283935	Comedy	Drama	Romance	72	8000000	100 min	...	7.4	7.4	7.5	7.4
3	4	About Time (2013)	7.8	225412	Comedy	Drama	Fantasy	NA	12000000	123 min	...	7.6	7.5	7.8	7.7
4	5	Amour (2012)	7.9	76121	Drama	Romance		94	8900000	127 min	...	7.9	7.8	8.1	6.6

5 rows x 58 columns

An SQLite database was used for this demo, but you can read data from other databases too. To do so, just call the appropriate DB Python module.

# Reading data from the clipboard

To read data from the clipboard, first copy some data. In the following example, we have copied a table from a the movies dataset:

M6

fx

22942

	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA
1	Budget	Runtime	CVotes10	CVotes09	CVotes08	CVotes07	CVotes06	CVotes05	CVotes04	CVotes03	CVotes02	CVotes01	CVotesMe	CVotesFe	CVotesU1	CVotesU1	CVotesU1	CVotes18	CVotes18
2	2000000	134 min	75556	126223	161460	83070	27231	9603	4021	2420	1785	4739	313823	82012	1837	1363	457	200910	153669
3	1800000	94 min	28939	44110	98845	78451	28394	9403	3796	1930	1161	2059	212866	44600	745	567	170	133336	106007
4	800000	100 min	28304	47501	99524	71485	24252	7545	2381	1109	634	1202	188925	58348	506	348	153	132350	96269
5	1200000	123 min	38556	43170	70850	45487	16542	5673	2210	1084	664	1182	126718	58098	654	325	321	92940	57778
6	8900000	127 min	11093	15944	22942	14187	5945	2585	1188	710	534	995	49808	16719	121	95	24	28593	20107
7	4450000	120 min	43875	89490	171495	115165	37332	12630	4992	2910	2020	6941	334838	67910	971	795	162	178794	146371
8	4700000	116 min	55533	87850	109536	65440	26913	10556	5057	3083	2194	4734	237437	46272	1943	1544	376	126301	101741
9	300000	109 min	16953	22109	31439	19251	8142	3412	1649	1033	826	1745	67076	23823	208	138	66	43312	30016
10	1.65E+08		50311	61304	103726	65681	22389	6830	2251	1036	539	1439	187383	58731	2446	1571	855	128237	91744
11	1800000	119 min	60209	94476	121637	80828	38373	19161	10116	6750	5378	11807	292808	63310	1891	1538	334	178850	142244
12	1300000	108 min	93798	136615	174500	97826	40319	16993	9084	6065	3981	8726	356707	143077	1112	583	516	244970	159567
13	400000	165 min	49673	62055	76838	52238	23789	10431	4906	3071	2248	5086	183807	51558	1393	995	381	123006	92639
14	4000000	142 min	15757	32840	83322	63800	19183	5178	1657	735	419	878	152707	23978	846	732	104	76784	64810
15	2.5E+08	147 min	81893	90156	117188	79377	32782	12322	5095	2994	1989	7786	264239	43818	3572	2865	683	148991	124124
16	1.7E+08	136 min	84943	103896	169440	120197	44124	14639	5571	2735	1932	5248	360615	66751	3765	2900	844	208526	170111
17	500000		16165	24762	39686	22429	7134	2255	982	542	419	832	71760	19138	447	329	112	40918	30740
18	5500000	134 min	37461	70216	133266	76657	21791	6099	2051	1062	707	1517	247889	41602	995	838	147	131052	110723
19	3500000	133 min	21364	28964	58237	45563	16432	5118	1655	848	464	1151	126214	13323	888	811	72	61826	55428
20	500000	117 min	37544	82276	145488	66156	16777	4582	1721	870	654	1588	231258	63266	864	650	205	145018	110493
21	1.7E+08	130 min	37605	58021	112652	84789	33747	11561	4734	2392	1470	2707	250421	32032	1720	1468	236	128574	111856
22	5800000	108 min	147467	147966	170810	105717	41811	15510	7046	4273	3037	8538	391955	79804	4598	3601	969	232840	186139

Next, use pandas' `read_clipboard` method to read the data and create a `DataFrame` , as follows:

```
df = pd.read_clipboard()  
df.head()
```

The data copied from the web page is now in memory as a `DataFrame` , as shown in the following screenshot. This method is quite handy when it comes to bringing data in quickly to pandas:

	CVotes10	CVotes09	CVotes08	CVotes07	CVotes06	CVotes05	CVotes04	CVotes03	CVotes02	CVotes01	CVotesMale	CVotesFemale	CVotesU18
0	75556	126223	161460	83070	27231	9603	4021	2420	1785	4739	313823	82012	1837
1	28939	44110	98845	78451	28394	9403	3796	1930	1161	2059	212866	44600	745
2	28304	47501	99524	71485	24252	7545	2381	1109	634	1202	188925	58348	506
3	38556	43170	70850	45487	16542	5673	2210	1084	664	1182	126718	58098	654
4	11093	15944	22942	14187	5945	2585	1188	710	534	995	49808	16719	121

# Summary

In this chapter, we learned how to work with different kinds of dataset formats in pandas. We learned how to use the advanced options provided by pandas while importing CSV files. We also saw how to work with Excel datasets, and we explored the methods available for working with various data formats such as HTML, JSON, PICKLE files, SQL, and so on.

In the next chapter, we will learn how to use pandas techniques in advanced data selection.

# Data Selection

In this chapter, we'll learn about advanced techniques of data selection with pandas, how to select a subset of data, how to select multiple rows and columns from a dataset, how to do sorting on a pandas DataFrame or a series, how to filter rows of a pandas DataFrame, and also learn how to apply multiple filters to a pandas DataFrame. We'll also look at how to use the `axis` parameter in pandas and the uses of string methods in pandas. Finally, we'll learn about how to change the datatype of a pandas series.

First, we'll learn about how to select a subset of data from a pandas DataFrame and create a series object. We'll start by importing a real dataset. We'll introduce some of the pandas data selection methods, and we'll apply these methods to our real dataset to demonstrate the selection of a subset of data.

In this chapter, we will be covering the following topics:

- Selecting data from the dataset
- Sorting the dataset
- Filtering rows using the pandas DataFrame
- Filtering data using multiple conditions, such as AND, OR, and ISIN
- Using the `axis` parameter in pandas
- Changing the datatype of pandas a series

# Introduction to datasets

We will be using a real dataset from [zillow.com](https://www.zillow.com), an online real estate marketplace that releases house price datasets as part of their research efforts. These datasets are available in the public domain and are free to use after proper attribution to [zillow.com](https://www.zillow.com). We will be using the latest data on mean house prices of US regions, available at <https://www.zillow.com/research/data/>. It is a CSV dataset, or a text file with CSVs. Let's start by importing the pandas modules into our Jupyter Notebook, as follows:

```
|import pandas as pd
```

We will then read in our dataset. Since it's a CSV file, we are using pandas' `read_csv` method for this. We pass the file name, with a comma as a separator, to the `read_csv` method, and we create a DataFrame out of this data, which we name `data`.

The dataset we received is in the form of a CSV file; hence we will use the common pandas `read_csv` method. We need to pass the file name and comma as a separator. The following code block will create a DataFrame with the name `data`:

```
|data = pd.read_csv('data-zillow.csv', sep=',')
```

The DataFrame is created and now we will read a few records from the dataset. This can be done by calling the `head` method (`data.head()`) on the DataFrame. This will give the output with columns, such as `Date`, and some location fields, such as `RegionName`, `State`, `Metro`, and `County`. The last column title, `Zhvi`, is a Zillow term and is the mean house price of that particular region, as you can see in the following screenshot:



	Date	RegionID	RegionName	State		Metro	County	SizeRank	Zhvi
0	2017-05-31	6181	New York	NY		New York	Queens	0	672400
1	2017-05-31	12447	Los Angeles	CA	Los Angeles-Long Beach-Anaheim	Los Angeles		1	629900
2	2017-05-31	17426	Chicago	IL		Chicago	Cook	2	222700
3	2017-05-31	13271	Philadelphia	PA		Philadelphia	Philadelphia	3	137300
4	2017-05-31	40326	Phoenix	AZ		Phoenix	Maricopa	4	211300

# Selecting data from the dataset

We will be selecting the columns from the DataFrame as pandas series, which can be done in two ways. The first method is to use bracket notation for this selection, as seen in the following code block:

```
| regions = data['RegionName']
```

By passing the `RegionName` column, we will get a `series` object. This `series` object will contain the values from this particular column only. How can we be sure this is a `series` object? We can check that by passing the `series` object created for the `type` function, as follows:

```
| type(regions)
```

Now, let's also look at the data in the `series` object we just created:

0	New York
1	Los Angeles
2	Chicago
3	Philadelphia
4	Phoenix
5	Las Vegas
6	San Diego
7	Dallas
8	San Jose
9	Jacksonville
10	San Francisco
11	Austin
12	Detroit
13	Columbus
14	Memphis
15	Charlotte
16	El Paso

Let's find out the difference between a pandas DataFrame and pandas series. pandas DataFrame is a multidimensional tabular data structure with labeled rows and columns. Series is a data structure containing single column values. A pandas' DataFrame can be considered a container for one or more series objects.

# Multi-column selection

To select multiple columns from a DataFrame, we need to pass the columns as a list to the DataFrame as follows:

```
| region_n_state = data[['RegionName', 'State']]  
| region_n_state.head()
```

Now, let's confirm whether the resulting object is a series or a DataFrame by using the `type` function, as follows:

```
| type(region_n_state)
```

Following is the output:



The screenshot shows a Jupyter Notebook interface. At the top right, there is a 'Slide Type' dropdown menu set to 'Fragment'. Below this, a code cell contains the command `type(region_n_state)`. The output of this command is displayed below the code: `pandas.core.frame.DataFrame`.

As can be seen in the preceding screenshot, selecting multiple columns creates another DataFrame, whereas selecting just a single column creates a `series` object.

# Dot notation

There is another way to create a new series out of a subset of data selected from a DataFrame. This method is called dot notation. In this method, the column name is passed to the DataFrame like when passing a property, rather than as a parameter:

```
|data.State
```

The following is the output:

0	NY
1	CA
2	IL
3	PA
4	AZ
5	NV
6	CA
7	TX
8	CA
9	FL
10	CA
11	TX
12	MI
13	OH
14	TN
15	NC
16	TX

We can select multiple series, and create new series out of those. We will create a new series using the three columns `County`, `Metro`, and `State`. We then concatenate these series and create a column in the DataFrame, called `Address`. Let's take a look at the newly created column or series we just created:

```
|data['Address'] = data.County + ', ' + data.Metro + ', ' + data.State
```

The following is the output:

```
0               Queens, New York, NY
1   Los Angeles, Los Angeles-Long Beach-Anaheim, CA
2               Cook, Chicago, IL
3   Philadelphia, Philadelphia, PA
4       Maricopa, Phoenix, AZ
5       Clark, Las Vegas, NV
6   San Diego, San Diego, CA
7   Dallas, Dallas-Fort Worth, TX
8   Santa Clara, San Jose, CA
9       Duval, Jacksonville, FL
10  San Francisco, San Francisco, CA
11       Travis, Austin, TX
12       Wayne, Detroit, MI
13   Franklin, Columbus, OH
14       Shelby, Memphis, TN
15  Mecklenburg, Charlotte, NC
16       El Paso, El Paso, TX
```

# Selecting multiple rows and columns from a pandas DataFrame

In this section, we will learn more about methods for selecting multiple rows and columns from a dataset read into pandas. We will also introduce some of the pandas data selection methods, and we will apply these methods to our real dataset to demonstrate the selection of a subset of data.

Let's get started by importing pandas and reading the data from zillow.com in the same manner as we did in the previous section. This is done as follows:

```
import pandas as pd
zillow = pd.read_table('data-zillow.csv', sep=',')
zillow.head()
```

The following is the output:

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
0	2017-05-31	6181	New York	NY	New York	Queens	0	672400
1	2017-05-31	12447	Los Angeles	CA	Los Angeles-Long Beach-Anaheim	Los Angeles	1	629900
2	2017-05-31	17426	Chicago	IL	Chicago	Cook	2	222700
3	2017-05-31	13271	Philadelphia	PA	Philadelphia	Philadelphia	3	137300
4	2017-05-31	40326	Phoenix	AZ	Phoenix	Maricopa	4	211300

Next, let's look at some techniques for selecting rows and columns using this dataset. pandas has a method for selecting rows and columns, called `loc`. We will be using the `loc` method to call the DataFrame from the dataset we

created earlier. `loc` requires two parameters separated by a comma, where the first parameter is the rows to be selected, and the second parameter is the columns to be selected, as shown in following code block:

```
|zillow.loc[7, 'Metro']
```

As seen in the preceding command, we are passing `7` as the index of the row we want to select, and we're passing the column with the name `Metro` as the column to be selected. This gives us the value of the row with index `7`, and column `Metro`.

We can also achieve this selection by referring to the columns by their index, rather than column names. For this, we will use the `iloc` method. In the `iloc` method, we need to pass both the rows and columns as index numbers. As seen in the following screenshot, we get the same result for both the methods:

```
|zillow.iloc[7,4]
```

The following will be the output for the previous code block:

Slide Type <span>Fragment ▾</span>	
<code>zillow.loc[7, 'Metro']</code>	
'Dallas-Fort Worth'	

Slide Type <span>Fragment ▾</span>	
<code>zillow.iloc[7,4]</code>	
'Dallas-Fort Worth'	



# Selecting a single row and multiple columns

In this section, we will view records for a single row and multiple columns where we pass multiple columns as a list:

```
|zillow.loc[7, ['Metro', 'County']]
```

We have our values from the row with index 7 and the `Metro` and `County` columns. If we select a single row, the values will be displayed vertically, rather than horizontally. We can convert this call to use the column index rather than the column name by using the `iloc` method instead of `loc`, as follows:

```
|zillow.iloc[7, [4,5]]
```

The following is the output:

```
Metro      Dallas-Fort Worth  
County      Dallas  
Name: 7, dtype: object
```

Now, we will learn how to select a single row, but with values from all columns. For the columns portion of the parameters, we use a colon (:). This tells the `loc` method to select all columns:

```
|zillow.loc[11, :]
```

The following is the output:

```
Date          2017-05-31
RegionID      10221
RegionName    Austin
State         TX
Metro         Austin
County        Travis
SizeRank      11
Zhvi          321600
Name: 11, dtype: object
```

Next, we will learn how to select values from multiple rows and a single column. Here, we need to pass rows as a range of rows. We pass rows with index numbers from 101 to 105 inclusively. We pass the column name as the second part of the parameter list, as follows:

```
|zillow.loc[101:105, 'Metro']
```

Here, we have the values from multiple rows and a single column. Next, we select data from multiple rows and multiple contiguous columns; just as for the row index range, we pass the column names as a range, as follows:

```
|zillow.loc[201:204, "State":"County"]
```

We can also use the `iloc` method to achieve this if we want to pass the column index instead of column names, as follows:

```
|zillow.iloc[201:205, 3:6]
```

And we get the same result as before. Now, we will look at selecting multiple non-contiguous values, where we just need to pass the column names as a list, as you can see in the following code:

```
|zillow.loc[201:205, ['RegionName', 'State']]
```

The following is the output:

	RegionName	State
201	Canton	OH
202	Metairie	LA
203	Santa Maria	CA
204	Inglewood	CA
205	Orange	CA

# Selecting values from a range of rows and all columns

Here, we will use `loc` method to view the values from a range of rows and columns. To achieve this, the first parameter for the `loc` method is the range index of rows to be selected. Since we want values from all columns, we pass a colon (:) as the second parameter, as follows:

```
|zillow.loc[201:205, :]
```

The output can be seen in the following screenshot:

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
201	2017-05-31	51260	Canton	OH	Canton	Stark	201	94400
202	2017-05-31	5914	Metairie	LA	New Orleans	Jefferson	202	232700
203	2017-05-31	47570	Santa Maria	CA	Santa Maria-Santa Barbara	Santa Barbara	203	354600
204	2017-05-31	45888	Inglewood	CA	Los Angeles-Long Beach-Anaheim	Los Angeles	204	470600
205	2017-05-31	33252	Orange	CA	Los Angeles-Long Beach-Anaheim	Orange	205	652000

Selecting non-contiguous rows also works in a similar way. In the non-contiguous rows method, we pass row indices as a list to the `loc` method, as seen in the following code:

```
|zillow.loc[[0,5,10], :]
```

And the output would be as follows:

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
0	2017-05-31	6181	New York	NY	New York	Queens	0	672400
5	2017-05-31	18959	Las Vegas	NV	Las Vegas	Clark	5	216500
10	2017-05-31	20330	San Francisco	CA	San Francisco	San Francisco	10	1194300

Let's suppose that we want to select rows and columns based on a certain column's value. The following line of code shows that we are selecting the rows where the value of the `county` column is `Queens`:

```
| zillow.loc[zillow.County=="Queens"]
```

Now, let's select all of the rows for a specific column based on the value of a different column. In the following code block, we are selecting the values from the `county` column, for the rows where `Metro` is `New York`:

```
| zillow.loc[zillow.Metro=="New York", "County"]
```

In the following screenshot, we can see all the counties for `New York` from our dataset:

0	Queens
63	Essex
72	Hudson
138	Westchester
176	Passaic
225	Union
291	Middlesex
338	Ocean
387	Ocean
402	Passaic
406	Dutchess
521	Westchester
541	Ocean
564	Passaic
582	Hudson
591	Essex
611	Westchester

# Sorting a pandas DataFrame

In this section, we will learn about the pandas `sort_values` method. We will also use various methods to sort a pandas DataFrame and learn how to sort a pandas series object.

We will start by importing the pandas module and reading the dataset of house prices from zillow.com into the Jupyter Notebook. First, let's start with the simple type of sorting. We will use pandas' `sort_values` method for this. For example, imagine that we want to sort the data by the `Metro` column. We need to pass `Metro` as a parameter to the `sort_values` method, and call the method on the DataFrame as follows:

```
|zillow.sort_values('Metro')
```

This shows that the data has been sorted by the `Metro` column, as shown in the following screenshot:

---

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
9851	2017-05-31	48458	Westport	WA	Aberdeen	Grays Harbor	9851	144600
4996	2017-05-31	36873	Elma	WA	Aberdeen	Grays Harbor	4996	175200
5090	2017-05-31	35514	Hoquiam	WA	Aberdeen	Grays Harbor	5090	95700
9401	2017-05-31	33215	Ocean Shores	WA	Aberdeen	Grays Harbor	9401	152400
9149	2017-05-31	18370	Grayland	WA	Aberdeen	Grays Harbor	9149	143900
2073	2017-05-31	30116	Aberdeen	WA	Aberdeen	Grays Harbor	2073	127800
9859	2017-05-31	31062	Cosmopolis	WA	Aberdeen	Grays Harbor	9859	147400
4568	2017-05-31	56078	Montesano	WA	Aberdeen	Grays Harbor	4568	182000
8420	2017-05-31	19269	McCleary	WA	Aberdeen	Grays Harbor	8420	170700
7108	2017-05-31	6275	Oakville	WA	Aberdeen	Grays Harbor	7108	186900

If you notice, by default, the `Date` column is sorted in ascending order. We can change the sorting order, giving the `ascending` parameter the value of `False`, as shown in the following code block:

```
|sorted = zillow.sort_values('Metro', ascending=False)
```

The `ascending` parameter is optional, and when not passed, it is set to `True` by default. Now, we will look into how to sort data by more than one column. To do this, we need to pass the list of columns, by which we want our data to be sorted, to the parameter `column` of the `sort_values` method, as follows:

```
|sorted = zillow.sort_values(by=['Metro', 'County'])
```

The data has now been sorted by `Metro` first, and then the `County` column; that is, in the same order that we passed them into the `sort_values` method. We can take the multiple column sort further, and introduce a mixed ascending order. For example, we can sort by three columns: `Metro`, `County`, and the `Price` column, as follows:



```
sorted = zillow.sort_values(by=['Metro', 'County', 'Zhvi'], ascending=[True, True, False])
sorted.head()
```

You must have noticed that we are passing a list of three Boolean values in ascending parameter. This sets the sort order to ascending for `Metro` and `County`, and descending for the last column, which is `Zhvi`:

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
7108	2017-05-31	6275	Oakville	WA	Aberdeen	Grays Harbor	7108	186900
4568	2017-05-31	56078	Montesano	WA	Aberdeen	Grays Harbor	4568	182000
4996	2017-05-31	36873	Elma	WA	Aberdeen	Grays Harbor	4996	175200
8420	2017-05-31	19269	McCleary	WA	Aberdeen	Grays Harbor	8420	170700
9401	2017-05-31	33215	Ocean Shores	WA	Aberdeen	Grays Harbor	9401	152400

Next, we see how to sort a `series` object. First, let's create a series. Let's select the `RegionID` column from our dataset, and create a series as follows:

```
regions = zillow.RegionID
type(regions)
```

Before we sort it, let's look at the original series by using `regions.head()`. The output would be as follows:

```
0      6181
1     12447
2     17426
3     13271
4     40326
Name: RegionID, dtype: int64
```

Now, let's sort it by calling the `sort_values` method on it. Since the dataset contains only one column, we don't need to pass any column name. Hence,

the code to sort the data would be `regions.sort_values().head()`, and the output would be as follows:

```
3043    3301
4159    3304
4986    3305
1762    3310
3116    3312
Name: RegionID, dtype: int64
```

# Filtering rows of a pandas DataFrame

In this section, we'll learn about methods for filtering rows and columns from a pandas DataFrame, and we will introduce a couple of ways to accomplish this. We will also learn about pandas' `filter` method, and how to use it on our real dataset, as well as ways to protect data based on a Boolean series that we will create from our data. We'll also learn how to pass the conditions directly to the DataFrame for filtering data.

We will start by importing the pandas module and reading the dataset of house prices from [zillow.com](https://www.zillow.com) into the Jupyter Notebook. First, we explore pandas' `filter` method for filtering data. We can filter columns using the `filter` method. To do this, we need to pass columns as a list to the `filter` method's `items` parameter, as follows:

```
filtered_data = data.filter(items=['State', 'Metro'])
filtered_data.head()
```

The following is the output:

	State	Metro
0	NY	New York
1	CA	Los Angeles-Long Beach-Anaheim
2	IL	Chicago
3	PA	Philadelphia
4	AZ	Phoenix

As you can see in the preceding screenshot, we filtered columns by `State` and `Metro` and created a new DataFrame with the values from the filter columns.

We then displayed the filter data using the `head` method. Next, we used the `filter` method to filter column names using regular expressions. This is done by passing our regular expression to the `regex` parameter, as follows:

```
filtered_data = data.filter(regex='Region', axis=1)
filtered_data.head()
```

When we print out filtered data, we can see that it has selected the two columns with `Region` in their name, as shown in the following screenshot:

	RegionID	RegionName
0	6181	New York
1	12447	Los Angeles
2	17426	Chicago
3	13271	Philadelphia
4	40326	Phoenix

The `filter` method is not the only way to filter data. To filter rows, we can use some interesting techniques—first, we create a series of Boolean values. The Boolean value series is based on the price value column from our dataset. We are choosing to select rows that have values of more than 500000, as follows:

```
price_filter_series = data['Zhvi'] > 500000
price_filter_series.head()
```

We confirmed our series creation by printing some values from the top, as shown in the following screenshot:

```
0      True
1      True
2     False
3     False
4     False
Name: Zhvi, dtype: bool
```

As shown in the preceding screenshot, the `True` values are those that match our condition, that is, they stand for rows where the price is higher than 500000. Next, we use this Boolean series to filter the rows from our complete dataset, and get only those values that have a price higher than 500000. To do this, we pass the Boolean series to the dataset DataFrame in square brackets, as follows:

```
|data[price_filter_series].head()
```

The other way to filter a dataset without explicitly creating a Boolean series is by passing the condition for the values we want to the DataFrame directly. For example, imagine that we want to filter and select only rows where the house price is higher than or equal to 1000000. We pass the condition to the DataFrame as follows:

```
|data[data.Zhvi >= 1000000].head()
```

The following screenshot show the records that have a value greater than 1000000:

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
10	2017-05-31	20330	San Francisco	CA	San Francisco	San Francisco	10	1194300
181	2017-05-31	54626	Sunnyvale	CA	San Jose	Santa Clara	181	1509300
234	2017-05-31	13713	Santa Clara	CA	San Jose	Santa Clara	234	1071500
238	2017-05-31	16992	Berkeley	CA	San Francisco	Alameda	238	1102000
308	2017-05-31	13699	San Mateo	CA	San Francisco	San Mateo	308	1198300

# Applying multiple filter criteria to a pandas DataFrame

In this section, we will learn about methods for applying multiple filter criteria to a pandas DataFrame. We will use logical AND/OR conditional operators to select records from our real dataset. We'll also see how to use the `isin()` method for filtering records. We will demonstrate the `isin` method on our real dataset for both single column and multiple column filtering.

We will start by importing the pandas module and reading the dataset of house prices from [zillow.com](https://www.zillow.com) into the Jupyter Notebook, as follows:

```
| data = pd.read_table('data-zillow.csv', sep=',')  
| data.head()
```

The following is the output:

	Date	RegionID	RegionName	State		Metro	County	SizeRank	Zhvi
0	2017-05-31	6181	New York	NY		New York	Queens	0	672400
1	2017-05-31	12447	Los Angeles	CA	Los Angeles-Long Beach-Anaheim	Los Angeles		1	629900
2	2017-05-31	17426	Chicago	IL		Chicago	Cook	2	222700
3	2017-05-31	13271	Philadelphia	PA		Philadelphia	Philadelphia	3	137300
4	2017-05-31	40326	Phoenix	AZ		Phoenix	Maricopa	4	211300

# Filtering based on multiple conditions – AND

Now, let's look at some techniques to filter the data using multiple criteria or conditions. First, we select those rows that have values for prices higher than 1,000,000, and for which the `state` parameter is New York (NY), as shown here:

```
|data[(data['Zhvi'] > 1000000) & (data['State'] == 'NY')].head()
```

Pass the preceding multi-condition to the database's DataFrame. In the following screenshot, this technique has selected only those rows where the price value is higher than 1,000,000, and `State` is New York, and has filtered out all the other records:

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
1132	2017-05-31	18375	Great Neck	NY	New York	Nassau	1132	1235800
2405	2017-05-31	54333	Scarsdale	NY	New York	Westchester	2405	1468100
2619	2017-05-31	47495	Rye	NY	New York	Westchester	2619	1736400
3032	2017-05-31	25725	Manhasset	NY	New York	Nassau	3032	1483400
3064	2017-05-31	18955	Larchmont	NY	New York	Westchester	3064	1052200



# Filtering based on multiple conditions – OR

We use the same technique to filter data when we pass these conditions with the logical operator OR. Here, we select those records which are from either New York or California. To do this, we join the conditions using the logical operator OR, and pass this combined condition to the dataset. The resulting sub-dataset is from only these two states, as shown here:

```
|data[((data['State'] == 'CA') | (data['State'] == 'NY'))].head()
```

The following is the output:

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
0	2017-05-31	6181	New York	NY	New York	Queens	0	672400
1	2017-05-31	12447	Los Angeles	CA	Los Angeles-Long Beach-Anaheim	Los Angeles	1	629900
6	2017-05-31	54296	San Diego	CA	San Diego	San Diego	6	572100
8	2017-05-31	33839	San Jose	CA	San Jose	Santa Clara	8	877400
10	2017-05-31	20330	San Francisco	CA	San Francisco	San Francisco	10	1194300

# Filtering using the isin method

Another way to filter data is by using the `isin` method. We can use the `isin` method to filter our dataset by a list of values for a particular column or columns. Here, we select only those records from the `Metro` column that have values of either `New York` OR `San Francisco`.

We call the `isin` method on the `Metro` column, and pass it a list containing the cities we want to select. This will create a Boolean series. We then pass the Boolean series to our dataset DataFrame to make the necessary filtering and selection, as follows:

```
filter = data['Metro'].isin(['New York', 'San Francisco'])
data[filter].head()
```

The following screenshot shows the filtered data with records from the two cities of `New York` and `San Francisco` only:

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
0	2017-05-31	6181	New York	NY	New York	Queens	0	672400
10	2017-05-31	20330	San Francisco	CA	San Francisco	San Francisco	10	1194300
38	2017-05-31	13072	Oakland	CA	San Francisco	Alameda	38	680100
63	2017-05-31	12970	Newark	NJ	New York	Essex	63	232800
72	2017-05-31	25320	Jersey City	NJ	New York	Hudson	72	380000

# Using the `isin` method with multiple conditions

We can also use the `isin` method to filter rows based on values from multiple columns. In order to perform this, we pass a dictionary object where keys are column names, and values are lists of values for those columns from which we want to select records.

Taking an example, let's select values where the `State` parameter is `California` and the `Metro` parameter is `San Francisco`. We create a dictionary object with these two columns containing the values we want to select, and then we pass this dictionary item to the `isin` method, and call the `isin` method on the dataset. We then pass the filter to the `DataFrame` and select our records, as follows:

```
filter = data.isin({'State': ['CA'], 'Metro': ['San Francisco']})  
data[filter].head()
```

This will show `NaN`, or not available for those records that do not fulfill the multiple criteria that we specified earlier:

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	CA	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

# Using the axis parameter in pandas

In this section, we will learn about when and where to use the `axis` parameter or keyword while doing data analysis in pandas. We'll introduce the `axis` parameter, and will walk through various values to which the `axis` keyword can be set to. We will demonstrate how setting `axis` to rows or columns changes a method's behavior. We'll also show a few code examples for using the `axis` keyword.

We will start by importing the pandas module and reading the dataset of house prices from [zillow.com](https://www.zillow.com) into the Jupyter Notebook:

```
data = pd.read_table('data-zillow.csv', sep=',')
data.head()
```

The following is the output:

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
0	2017-05-31	6181	New York	NY	New York	Queens	0	672400
1	2017-05-31	12447	Los Angeles	CA	Los Angeles-Long Beach-Anaheim	Los Angeles	1	629900
2	2017-05-31	17426	Chicago	IL	Chicago	Cook	2	222700
3	2017-05-31	13271	Philadelphia	PA	Philadelphia	Philadelphia	3	137300
4	2017-05-31	40326	Phoenix	AZ	Phoenix	Maricopa	4	211300

# Usage of the axis parameter

The `axis` parameter tells you a particular method, along with which axis of the DataFrame the method should be executed. The following code depicts the input datatype of the `axis` parameter:

```
|data.head()
```

The axis can be specified vertically or horizontally, or in other words, along rows or columns: use `axis=0` for rows and `axis=1` for columns, as shown in the following screenshot:

	Date	RegionID	RegionName	State		Metro	County	SizeRank	Zhvi
0	2017-05-31	6181	New York	NY		New York	Queens	0	672400
1	2017-05-31	12447	Los Angeles	CA	Los Angeles-Long Beach-Anaheim	Los Angeles		1	629900
2	2017-05-31	17426	Chicago	IL		Chicago	Cook	2	222700
3	2017-05-31	13271	Philadelphia	PA		Philadelphia	Philadelphia	3	137300
4	2017-05-31	40326	Phoenix	AZ		Phoenix	Maricopa	4	211300

The following is the command:

```
|data.axes
```

Following is the output of the preceding command:

```
[RangeIndex(start=0, stop=10830, step=1),  
Index([u'Date', u'RegionID', u'RegionName', u'State', u'Metro', u'County',  
       u'SizeRank', u'Zhvi'],  
      dtype='object')]
```

# Axis usage examples

In the `axis` usage examples, we calculate the mean for the values in the dataset. We have passed `axis` as `0`. This means that the mean will be calculated along the row `axis`, as follows:

```
|data.mean(axis=0)
```

The following is the output:

```
RegionID      84344.818837
SizeRank      5414.500000
Zhvi          250307.590028
dtype: float64
```

Next, we set `axis` to `1`. We are using the exact same method on the same dataset; however, we are changing `axis` from `0` to `1`. Since we set `axis` to `1`, the mean has been calculated along the columns:

```
|data.mean(axis=1).head()
```

The following is the output:

```
0      226193.666667
1      214116.000000
2       80042.666667
3       50191.333333
4       83876.666667
dtype: float64
```

Sometimes it's difficult to remember whether `0` or `1` is for rows or columns. So instead of using `axis=0`, you can set `axis` to `rows`:

```
|data.mean(axis='rows')
```

The following is the output:

```
RegionID      84344.818837
SizeRank      5414.500000
Zhvi          250307.590028
dtype: float64
```

For columns, you can set `axis` to `columns`. It has the same effect as using `0` or `1`:

```
|data.mean(axis='columns').head()
```

The following is the output:

```
0      226193.666667
1      214116.000000
2       80042.666667
3       50191.333333
4       83876.666667
dtype: float64
```



# More examples of the axis keyword

Here, we use the `drop` method to drop a row or record. We tell the `drop` method to drop the record at the index of `0`, by passing the keyword `axis` as `0`:

```
|data.drop(0, axis=0).head()
```

The following is the output:

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
1	2017-05-31	12447	Los Angeles	CA	Los Angeles-Long Beach-Anaheim	Los Angeles	1	629900
2	2017-05-31	17426	Chicago	IL	Chicago	Cook	2	222700
3	2017-05-31	13271	Philadelphia	PA	Philadelphia	Philadelphia	3	137300
4	2017-05-31	40326	Phoenix	AZ	Phoenix	Maricopa	4	211300
5	2017-05-31	18959	Las Vegas	NV	Las Vegas	Clark	5	216500

In the following example, we set `axis` to `1`, which tells the `drop` method to drop the column with the `Date` label:

```
|data.drop('Date', axis=1).head
```

The following is the output:

	RegionID	RegionName	State		Metro	County	SizeRank	Zhvi
0	6181	New York	NY		New York	Queens	0	672400
1	12447	Los Angeles	CA	Los Angeles-Long Beach-Anaheim	Los Angeles		1	629900
2	17426	Chicago	IL		Chicago	Cook	2	222700
3	13271	Philadelphia	PA		Philadelphia	Philadelphia	3	137300
4	40326	Phoenix	AZ		Phoenix	Maricopa	4	211300

# The axis keyword

We can also use the `axis` keyword in a filtering method. Here, we filter by `regex Region`, with `axis` set to `column`:

```
|data.filter(regex='Region', axis=1).head()
```

The following is the output:

	RegionID	RegionName
0	6181	New York
1	12447	Los Angeles
2	17426	Chicago
3	13271	Philadelphia
4	40326	Phoenix

# Using string methods in pandas

In this section, we will learn about using various string methods on a pandas series. We will read a real dataset into pandas. We'll explore some of the string methods, and we will use these string methods to select and change values from our dataset.

We will start by importing the pandas module and reading the dataset of house prices from [zillow.com](https://www.zillow.com) into the Jupyter Notebook:

```
| data = pd.read_table('data-zillow.csv', sep=',')  
| data.head()
```

The following is the output:

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
0	2017-05-31	6181	New York	NY	New York	Queens	0	672400
1	2017-05-31	12447	Los Angeles	CA	Los Angeles-Long Beach-Anaheim	Los Angeles	1	629900
2	2017-05-31	17426	Chicago	IL	Chicago	Cook	2	222700
3	2017-05-31	13271	Philadelphia	PA	Philadelphia	Philadelphia	3	137300
4	2017-05-31	40326	Phoenix	AZ	Phoenix	Maricopa	4	211300

# Checking for a substring

In order to learn how to use string method to check for a substring from pandas series, we use the `contains` method from the `str` package.

Here, we call the `str.contains` method on the `RegionName` series from our dataset. We are looking for records that have the `New` substring in them. It prints out a Boolean series, printing `True` where a substring is found, and `False` where a substring is not found:

```
|data.RegionName.str.contains('New').head()
```

The following is the output:

```
0      True
1     False
2     False
3     False
4     False
Name: RegionName, dtype: bool
```

# Changing the values of a series or column into uppercase

There is a very common string method for converting Python strings to uppercase. We can use this to convert all the values from a column into uppercase. We do this by calling `str.upper` on the series. Here, we call it on the `RegionName` column:

```
|data.RegionName.str.upper().head()
```

The following is the output:

```
0      NEW YORK
1    LOS ANGELES
2      CHICAGO
3  PHILADELPHIA
4      PHOENIX
Name: RegionName, dtype: object
```

# Changing the values into lowercase

We use the `lower` string method for this:

```
|data.County.str.lower().head()
```

The following is the output:

```
0      queens
1  los angeles
2      cook
3  philadelphia
4    maricopa
Name: County, dtype: object
```

# Finding the length of every value of a column

We do this by calling the `str.len` method on one of our columns:

```
|data.County.str.len().head()
```

The following is the output:

```
0      6
1     11
2      4
3     12
4      8
Name: County, dtype: int64
```



# Removing white spaces

We can also do some data cleaning using string methods. For example, here, we use the `lstrip` method to remove all leading white spaces from the values of a column:

```
|data.RegionName.str.lstrip().head()
```

The following is the output:

```
0      New York
1    Los Angeles
2      Chicago
3  Philadelphia
4      Phoenix
Name: RegionName, dtype: object
```

# Replacing parts of a column's values

We can also change our data using string methods. Here, we replace the spaces in the `RegionName` column from our dataset with no space, using the `replace` method:

```
|data.RegionName.str.replace(' ', '').head()
```

The following is the output:

```
0      NewYork
1  LosAngeles
2    Chicago
3 Philadelphia
4    Phoenix
Name: RegionName, dtype: object
```

# Changing the datatype of a pandas series

In this section, we'll learn about changing the datatype of a pandas series. We'll see how to change datatypes after reading the data within. We'll also learn how to change datatypes while reading data in pandas. We'll walk through an example of changing an int column to float. We'll also see how to convert a string values column to the `datetime` datatype.

We will start by importing the pandas module and reading the dataset of house prices from [zillow.com](https://www.zillow.com) into the Jupyter Notebook:

```
| data = pd.read_table('data-zillow.csv', sep=',')  
| data.head()
```

The following is the output:

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
0	2017-05-31	6181	New York	NY	New York	Queens	0	672400
1	2017-05-31	12447	Los Angeles	CA	Los Angeles-Long Beach-Anaheim	Los Angeles	1	629900
2	2017-05-31	17426	Chicago	IL	Chicago	Cook	2	222700
3	2017-05-31	13271	Philadelphia	PA	Philadelphia	Philadelphia	3	137300
4	2017-05-31	40326	Phoenix	AZ	Phoenix	Maricopa	4	211300

# Changing an int datatype column to a float

To do this, we first check the datatypes of columns from our real dataset:

```
|data.dtypes
```

The following is the output:

```
Date          object
RegionID      int64
RegionName    object
State         object
Metro         object
County        object
SizeRank      int64
Zhvi          int64
dtype: object
```

We then use the `astype` method to change the datatype. We pass `float` to the `astype` method and call this method on the column, the datatype of which we want to change.

We assign the change back to the original column as follows:

```
|data['Zhvi'] = data.Zhvi.astype(float)
|data.dtypes
```

In the following screenshot, we can see that the change has been made—the datatype of our column has been changed from `int64` to `float64`:

Date	object
RegionID	int64
RegionName	object
State	object
Metro	object
County	object
SizeRank	int64
Zhvi	float64
dtype:	object

# Changing the datatype while reading data

We just changed the datatype of the column after the data has been read into pandas. Alternatively, we can change the datatype while reading in the data. For this, we pass the column name and the datatype into the column we want to change to the `read` data method. The column we wanted in float has been imported as `float64`:

```
data2 = pd.read_csv('data-zillow.csv', sep=',', dtype={'Zhvi':float})
data2.dtypes
```

The following is the output:

```
Date          object
RegionID      int64
RegionName    object
State         object
Metro         object
County        object
SizeRank      int64
Zhvi          float64
dtype: object
```

# Converting string to datetime

The main thing here is that our dataset has a date column, but it shows up as an object or a string datatype. We will convert this into a proper `datetime` column.

We will use pandas' `to_datetime` method for this, which can parse a few different `datetime` formats:

```
|pd.to_datetime(data2.Date,infer_datetime_format=True).head()
```

We can see that our `Date` field has been changed from an object to `datetime64`, as shown in the following screenshot:

```
0    2017-05-31
1    2017-05-31
2    2017-05-31
3    2017-05-31
4    2017-05-31
Name: Date, dtype: datetime64[ns]
```

# Summary

In this chapter, we learned about methods for selecting a subset of data from a pandas DataFrame. We also learned about how to apply these methods to a real dataset. We also learned about methods for selecting multiple rows and columns from a dataset that has been read into pandas, and we applied these methods to our real dataset to demonstrate the selection of a subset of data. We learned about the pandas `sort_values` method. We saw various ways to use the `sort_values` method to sort data in a pandas DataFrame. We also learned how to sort a pandas series object. We learned about methods for filtering rows and columns from a pandas DataFrame. We introduced a couple of ways to accomplish this. We learned about pandas' `filter` method, and how to use it on our real dataset. We also learned ways to filter data based on a Boolean series that we created from our data, and we learned how to pass the conditions for filtering data directly to the DataFrame. We learned various techniques of data selection in pandas, and how to select a subset of data. We also learned how to select multiple rows and columns from a dataset. We learned how to implement sorting on a pandas DataFrame or a series. We walked through how to filter the rows of a pandas DataFrame, how to apply multiple filters to such a DataFrame, as well as how to use the `axis` parameter in pandas. We also examined the uses of string methods in pandas, and finally, we learned how to change the datatypes of a pandas series.

In the next chapter, we'll learn about techniques for manipulating, transforming, and reshaping data.



# Manipulating, Transforming, and Reshaping Data

In this chapter, we will learn about the following topics:

- Modifying a pandas DataFrame using the `inplace` parameter
- Scenarios where you can use the `groupby` method
- How to handle missing values in pandas
- Exploring indexing in pandas DataFrames
- Renaming and removing columns in a pandas DataFrame
- Working with and transforming date and time data
- Handling `SettingWithCopyWarning`
- Applying a function to a pandas series or DataFrame
- Merging and concatenating multiple DataFrames into one

# Modifying a pandas DataFrame using the inplace parameter

In this section, we'll learn about how to modify a DataFrame using the `inplace` parameter. We'll first read a real dataset into pandas. We'll then introduce pandas' `inplace` parameter, and see how it impacts a method's execution end result. We'll also execute methods with and without `inplace` parameters to demonstrate the effect of `inplace`.

We'll start by importing the `pandas` module in to our Jupyter notebook, as follows:

```
|import pandas as pd
```

We will then read in our dataset:

```
|top_movies = pd.read_table('data-movies-top-grossing.csv', sep=',')
```

Since it's a CSV file, we are using pandas' `read_csv` function for this. Now that we have read in our dataset into a DataFrame, let's take a look at a few of the records:

```
|top_movies
```

Rank		Title	Worldwide gross	Year
0	1	Avatar	\$2,787,965,087	2009
1	2	Titanic	\$2,186,772,302	1997
2	3	Star Wars: The Force Awakens	\$2,068,223,624	2015
3	4	Jurassic World	\$1,671,713,208	2015
4	5	The Avengers	\$1,518,812,988	2012
5	6	Furious 7	\$1,516,045,911	2015
6	7	Avengers: Age of Ultron	\$1,405,403,694	2015
7	8	Harry Potter and the Deathly Hallows – Part 2	\$1,341,511,219	2011
8	9	Frozen	\$1,287,000,000	2013

The data we are using is from Wikipedia; it's the cross annex data for top movies worldwide to date. Most pandas DataFrame methods return a new DataFrame. However, you might want to use a method to modify the original DataFrame itself. This is where the `inplace` parameter is useful. Let's call a method on a DataFrame without the `inplace` parameter to see how it works in the code:

```
| top_movies.set_index('Rank').head()
```

	Title	Worldwide gross	Year
Rank			
1	Avatar	\$2,787,965,087	2009
2	Titanic	\$2,186,772,302	1997
3	Star Wars: The Force Awakens	\$2,068,223,624	2015
4	Jurassic World	\$1,671,713,208	2015
5	The Avengers	\$1,518,812,988	2012

Here, we are setting one of the columns as the index for our DataFrame. We can see that the index has been set in the memory. Let's check now to see if it has modified the original DataFrame or not:

```
|top_movies.head()
```

	Rank	Title	Worldwide gross	Year
0	1	Avatar	\$2,787,965,087	2009
1	2	Titanic	\$2,186,772,302	1997
2	3	Star Wars: The Force Awakens	\$2,068,223,624	2015
3	4	Jurassic World	\$1,671,713,208	2015
4	5	The Avengers	\$1,518,812,988	2012

We can see that in the original DataFrame there has been no change. The `set_index` method only created the change in a completely new DataFrame in

memory, which we could have saved in a new DataFrame. Now let's see how it works if we pass the `inplace` parameter:

```
|top_movies.set_index('Rank', inplace=True)
```

We pass `inplace=True` to the method. Now let's check the original DataFrame:

```
|top_movies.head()
```

	Title	Worldwide gross	Year
Rank			
1	Avatar	\$2,787,965,087	2009
2	Titanic	\$2,186,772,302	1997
3	Star Wars: The Force Awakens	\$2,068,223,624	2015
4	Jurassic World	\$1,671,713,208	2015
5	The Avengers	\$1,518,812,988	2012

We can see that passing `inplace=True` did modify the original DataFrame. Not all methods require the use of the `inplace` parameter to modify the original DataFrame. For example, the `rename(columns)` method modifies the original DataFrame, without the need for the `inplace` parameter:

```
|top_movies.rename(columns = {'Year': 'Release Year'}).head()
```

	Title	Worldwide gross	Release Year
Rank			
1	Avatar	\$2,787,965,087	2009
2	Titanic	\$2,186,772,302	1997
3	Star Wars: The Force Awakens	\$2,068,223,624	2015
4	Jurassic World	\$1,671,713,208	2015
5	The Avengers	\$1,518,812,988	2012

It's a good idea to get familiar with which methods need `inplace` and which don't. In this section, we learned about how to modify a DataFrame using the `inplace` parameter. We introduced pandas `inplace` parameter, and how it impacts a method's execution end result. We explored the execution of methods with and without the `inplace` parameter to demonstrate the difference in the results. In the next section, we'll learn about using the `groupby` method.

# Using the groupby method

In this section, we will learn about using the `groupby` method to split and aggregate data into groups. We'll explore how the `groupby` method works by breaking it into parts. We'll demonstrate `groupby` with statistical and other methods. We will also learn how to do interesting things with the `groupby` method's ability to iterate over the group data.

We will start by importing the `pandas` module into our Jupyter notebook, as we did in the previous section:

```
|import pandas as pd
```

We will then read in our CSV dataset:

```
|data = pd.read_table('data-zillow.csv', sep=',')  
|data.head()
```

	Date	RegionID	RegionName	State	Metro	County	SizeRank	Price
0	2017-05-31	6181	New York	NY	New York	Queens	0	672400
1	2017-05-31	12447	Los Angeles	CA	Los Angeles-Long Beach-Anaheim	Los Angeles	1	629900
2	2017-05-31	17426	Chicago	IL	Chicago	Cook	2	222700
3	2017-05-31	13271	Philadelphia	PA	Philadelphia	Philadelphia	3	137300
4	2017-05-31	40326	Phoenix	AZ	Phoenix	Maricopa	4	211300

Let's start by asking a question, and see if `pandas`' `groupby` method can help us get the answer. We want to get the mean `Price` value of every `State`:

```
|grouped_data = data[['State', 'Price']].groupby('State').mean()  
|grouped_data.head()
```

Price	
State	
AK	237783
AL	137645
AR	136331
AZ	232353
CA	617425

Here, we used the `groupby` method for aggregating data by states, and got the mean `Price` per `State`. In the background, the `groupby` method split the data into groups, and we then applied the function on the split data, and the result was put together and displayed.

Let's break this code into its individual pieces to see how it happened. First, splitting into groups is done as follows:

```
|grouped_data = data[['State', 'Price']].groupby('State')
```

We selected a subset of data that has only `State` and `Price` columns. We then call the `groupby` method on this data, and pass it in the `State` column, as that is the column we want the data to be grouped by. Then, we store the data in an object. Let's print out this data using the `list` method:

```
|list(grouped_data)
```



	State	Price
57	AK	293900
842	AK	221000
1793	AK	247800
1830	AK	213100
1974	AK	323100
3756	AK	206500
3869	AK	270700
4450	AK	224700
5229	AK	207500
5996	AK	249700
9622	AK	219600
10162	AK	175800
71	AL	112100
121	AL	61900
154	AL	138600
736	AL	110500
913	AL	105700
1010	AL	141600

Now, we have the data groups based on date. Next, we apply a function on the displayed data, and display the combined result:

```
|grouped_data.mean().head()
```

Price	
State	
AK	237783
AL	137645
AR	136331
AZ	232353
CA	617425

We are using the `mean` method to get the mean of the prices. After the data is split into groups, we can use pandas methods to get some interesting

information on these groups. For example, here, we get descriptive statistical information on each state separately:

```
|grouped_data.describe()
```

State		Price
AK	count	1.200000e+01
	mean	2.377833e+05
	std	4.143371e+04
	min	1.758000e+05
	25%	2.117000e+05
	50%	2.228500e+05
	75%	2.549500e+05
	max	3.231000e+05
AL	count	1.490000e+02
	mean	1.376456e+05
	std	7.253854e+04
	min	4.470000e+04
	25%	1.039000e+05
	50%	1.264000e+05
	75%	1.558000e+05
	max	5.989000e+05
	count	8.200000e+01
	mean	1.363317e+05
	std	4.237054e+04

We can also use `groupby` on multiple columns. For example, here, we are grouping by the `State` and `RegionName` columns, as follows:

```
|grouped_data = data[['State',  
                      'RegionName',
```

```
| 'Price']].groupby(['State', 'RegionName']).mean()
```

		Price
State	RegionName	
AK	Anchor Point	175800
	Anchorage	293900
	Fairbanks	221000
	Juneau	323100
	Kenai	206500

We can also get the number of records per state through the `groupby` and `size` methods, as follows:

```
|grouped_data = data.groupby(['State']).size()
```

State	
AK	12
AL	149
AR	82
AZ	102
CA	701
CO	166
CT	183
DC	1
DE	26
FL	528
GA	266
HI	43
IA	63
ID	33
IL	496
IN	418
KS	52

In all the code we have demonstrated in this section so far, we grouped by rows. However, we can also group by columns. In the following example, we do this by passing the `axis` parameter set to 1:

```
| grouped_data = data.groupby(data.dtypes, axis=1)  
| list(grouped_data)
```

[(dtype('int64'),		RegionID	SizeRank	Price
0	6181	0	672400	
1	12447	1	629900	
2	17426	2	222700	
3	13271	3	137300	
4	40326	4	211300	
5	18959	5	216500	
6	54296	6	572100	
7	38128	7	164700	
8	33839	8	877400	
9	25290	9	152300	
10	20330	10	1194300	
11	10221	11	321600	
12	17762	12	41500	
13	10920	13	128300	
14	32811	14	81100	
15	24043	15	183800	
16	17933	16	113400	
17	44269	17	554600	

We can also iterate over the split groups, and do interesting things with them, as follows:

```
| for state, grouped_data in data.groupby('State'):  
|     print(state, '\n', grouped_data)
```

AK					
	Date	RegionID	RegionName	State	Metro \
57	2017-05-31	23482	Anchorage	AK	Anchorage
842	2017-05-31	38465	Fairbanks	AK	Fairbanks
1793	2017-05-31	36906	Palmer	AK	Anchorage
1830	2017-05-31	29910	North Pole	AK	Fairbanks
1974	2017-05-31	5365	Juneau	AK	Juneau
3756	2017-05-31	52742	Kenai	AK	NaN
3869	2017-05-31	39281	Kodiak	AK	NaN
4450	2017-05-31	102611	Tanaina	AK	Anchorage
5229	2017-05-31	32296	Ketchikan	AK	Ketchikan
5996	2017-05-31	395445	Lakes	AK	Anchorage
9622	2017-05-31	54367	Seward	AK	NaN
10162	2017-05-31	28124	Anchor Point	AK	NaN
		County	SizeRank	Price	
57		Anchorage	57	293900	
842	Fairbanks	North Star	842	221000	
1793	Matanuska	Susitna	1793	247800	

Here, we iterate over the group data by `state`, and publish the result with `state` as the heading, followed by a table of all the records from that `state`.

In this section, we learned about using the `groupby` method to split and aggregate data into groups. We explored how the `groupby` method works by breaking it into its pieces. We demonstrated `groupby` with the statistical and other methods, and we also learned how to do interesting things through `groupby` by iterating over the group data. In the next section, we'll learn about how to handle missing values in data using `pandas`.

# Handling missing values in pandas

In this section, we will explore how we can use various pandas techniques to handle the missing data in our datasets. We will learn how to find out how much data is missing, and from which columns. We'll see how to drop rows or columns where all or a lot of records are missing data. We'll also learn how, instead of dropping data, we can also fill in the missing records with zeros or the mean of the remaining values.

We will start by importing the `pandas` module into our Jupyter notebook:

```
|import pandas as pd
```

We will then read in our CSV dataset:

```
|data = pd.read_csv('data-titanic.csv')  
|data.head()
```

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

This dataset is the Titanic's passenger survival dataset, available for download from Kaggle at <https://www.kaggle.com/c/titanic/data>.

Let's look at how many records are missing first. To do this, we first need to find out the total number of records in the dataset. We do this by calling the

shape property on the DataFrame:

```
| data.shape
```

```
data.shape
```

```
(891, 12)
```

We can see that the total number of records is 891, and that the total number of columns is 12.

We then find out the number of records in each column. We can do this by calling the `count` method on the DataFrame:

```
| data.count()
```

```
PassengerId    891
Survived        891
Pclass          891
Name            891
Sex             891
Age            714
SibSp           891
Parch           891
Ticket          891
Fare            891
Cabin          204
Embarked        889
dtype: int64
```

The difference between the total records and the count per column represents the number of records missing from that column. Out of the 12 columns, we have 3 columns where values are missing. For example, `Age` has only 714 values out of a total of 891 rows; `Cabin` has values for only 204 records; and `Embarked` has values for 889 records. There are different ways we could handle

these missing values. One of the ways is to drop any row where a value is missing, even from a single column, as follows:

```
| data_missing_dropped = data.dropna()  
| data_missing_dropped.shape
```

When we run this drop rows method, we assign the results back into a new DataFrame. This leaves us with just 183 records, out of a total of 891. However, this may lead to losing a lot of the data, and may not be acceptable.

Another method is to drop only those rows where all values are missing. Here is an example:

```
| data_all_missing_dropped = data.dropna(how="all")  
| data_all_missing_dropped.shape
```

We do this by setting the `how` parameter for the `dropna` method to `all`.

Instead of dropping rows, another method is to fill in the missing values with some data. We can fill in the missing values with 0, for example, as in the following screenshot:

```
| data_filled_zeros = data.fillna(0)  
| data_filled_zeros.count()
```



```
PassengerId      891
Survived          891
Pclass           891
Name             891
Sex              891
Age              891
SibSp            891
Parch            891
Ticket           891
Fare             891
Cabin            891
Embarked         891
dtype: int64
```

Here, we use pandas' `fillna` method, and we pass the numeric value of 0 to the column where that data should be filled in. You can see that we have now filled all the missing values with 0, and hence the count for all the columns has gone up to the total number of count of records in the dataset.

Also, instead of filling in missing values with 0, we could fill them with the mean of the remaining existing values. To do this, we call the `fillna` method on the column where we are filling the values in, and we pass the mean of the column as the parameter:

```
data_filled_in_mean = data.copy()
data_filled_in_mean.Age.fillna(data.Age.mean(), inplace=True)
data_filled_in_mean.count()
```

```
PassengerId      891
Survived          891
Pclass           891
Name             891
Sex              891
Age              891
SibSp            891
Parch            891
Ticket           891
Fare             891
Cabin            204
Embarked         889
dtype: int64
```

For example, here, we filled in the missing value of `Age` with the mean of the existing values.

In this section, we explored how we can use various pandas techniques to handle the missing data from our datasets. We learned how to find out how much data is missing, and from which columns. We saw how to remove rows or columns where all or a lot of the records are missing data. We also saw how, instead of removing, we can also fill in the missing records with 0 or mean of the remaining values. In the next section, we will learn about how to do dataset indexing in pandas DataFrames.

# Indexing in pandas DataFrames

In this section, we will explore how to set an index and use it for data analysis in pandas. We will learn how to set an index on the `DataFrame` after reading in the data, as well as while reading in data. We'll also see how to use this index for data selection.

As always, we start by importing the `pandas` module into our Jupyter notebook:

```
|import pandas as pd
```

We then read in our dataset:

```
|data = pd.read_csv('data-titanic.csv')
```

The following is how our default index looks like right now, which is a numeric index starting from 0:

```
|data.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Let's set it to a column of our choice. Here, we use the `set_index` method to set the index to the name of the passenger from our data:

```
|data.set_index('Name')
```

	PassengerId	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
Name											
Braund, Mr. Owen Harris	1	0	3	male	22.0	1	0	A/5 21171	7.2500	NaN	S
Cumings, Mrs. John Bradley (Florence Briggs Thayer)	2	1	1	female	38.0	1	0	PC 17599	71.2833	C85	C
Heikkinen, Miss. Laina	3	1	3	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
Futrelle, Mrs. Jacques Heath (Lily May Peel)	4	1	1	female	35.0	1	0	113803	53.1000	C123	S
Allen, Mr. William Henry	5	0	3	male	35.0	0	0	373450	8.0500	NaN	S
Moran, Mr. James	6	0	3	male	NaN	0	0	330877	8.4583	NaN	Q
McCarthy, Mr. Timothy J	7	0	1	male	54.0	0	0	17463	51.8625	E46	S
Palsson, Master. Gosta Leonard	8	0	3	male	2.0	3	1	349909	21.0750	NaN	S

As you can see, the index has been changed from the simple numeric value of 0 to the names of the passengers from our dataset.

Next, we'll see how to set an index while reading in the data. We do this by passing in an extra parameter, `index_col`, to the `read` method:

```
|data = pd.read_table('data-titanic.csv', sep=',', index_col=3)
```

	PassengerId	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
Name											
Braund, Mr. Owen Harris	1	0	3	male	22.0	1	0	A/5 21171	7.2500	NaN	S
Cumings, Mrs. John Bradley (Florence Briggs Thayer)	2	1	1	female	38.0	1	0	PC 17599	71.2833	C85	C
Heikkinen, Miss. Laina	3	1	3	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
Futrelle, Mrs. Jacques Heath (Lily May Peel)	4	1	1	female	35.0	1	0	113803	53.1000	C123	S
Allen, Mr. William Henry	5	0	3	male	35.0	0	0	373450	8.0500	NaN	S

The `index_col` parameter takes a single numeric value or a sequence of values. Here, we are passing the index of the `Name` column.

Next, let's see how to do data selection using the index. In the following screenshot, we call the `loc` method on the `DataFrame` and pass in the index level of the record that we want to select:

```
|dta.loc['Braund, Mr.Owen Harris',:]
```

```
PassengerId      1
Survived         0
Pclass           3
Sex              male
Age             22
SibSp            1
Parch            0
Ticket      A/5 21171
Fare          7.25
Cabin         NaN
Embarked        S
Name: Braund, Mr. Owen Harris, dtype: object
```

In this case, it's the name of one of the passengers from the dataset. We can do this because we set up the name as the index for the dataset previously.

And finally, we can reset the index back to what it was before we changed it. We use the `reset_index` method for this:

```
|data.reset_index(inplace=True)
```

We are passing `inplace=True` as we want to reset it in the original DataFrame itself.

In this section, we explored how to set the index and use it for data analysis in pandas. We also learned how to set an index on the DataFrame after we have read in the data. We also saw how to set an index while reading in the data from the CSV file. Finally, we saw some methods that allow us to use an index for data selection. In the next section, we will learn about how to go about renaming columns in a pandas DataFrame.

# Renaming columns in a pandas DataFrame

In this section, we'll learn about various methods for renaming column labels in pandas. We will learn how to rename columns both after reading in data and while reading in data, and we'll see how to rename either all or specific columns. We will start by importing the `pandas` module into our Jupyter notebook:

```
|import pandas as pd
```

There are a few ways by which we can rename the columns in a pandas DataFrame. One of the ways is to rename the columns while reading in the data from the dataset. To do this, we need to pass the column names as a list to the `names` parameter of the `read_csv` method:

```
|list_columns= ['Date', 'Region ID', 'Region Name', 'State', 'City', 'County', 'Size Rank', 'Price']  
|data = pd.read_csv('data-zillow.csv', names = list_columns)  
|data.head()
```

	Date	Region ID	Region Name	State	City	County	Size Rank	Price
0	Date	RegionID	RegionName	State	Metro	County	SizeRank	Zhvi
1	2017-05-31	6181	New York	NY	New York	Queens	0	672400
2	2017-05-31	12447	Los Angeles	CA	Los Angeles-Long Beach-Anaheim	Los Angeles	1	629900
3	2017-05-31	17426	Chicago	IL	Chicago	Cook	2	222700
4	2017-05-31	13271	Philadelphia	PA	Philadelphia	Philadelphia	3	137300

In the preceding example, we first created a list of column names we wanted; this should be the same number as the number of columns in the

actual dataset. We then pass the list to the `names` parameter in the `read_csv` method. We then see that we have the column names we wanted, and so the `read_csv` method has changed the column names from what was there by default in the text file to the names we supplied.

We can also rename column names after the data has been read. Let's read the dataset in again from the CSV file, but this time without supplying any column names. We can rename columns by using the `rename` method. Let's first look at the columns from our dataset:

```
|data.columns  
Index(['Date', 'RegionID', 'RegionName', 'State', 'Metro', 'County',  
       'SizeRank', 'Price'],  
      dtype='object')
```

Now, we call the `rename` method on the DataFrame and pass the column names, old and new values, to the `columns` parameter:

```
|data.rename(columns={'RegionName':'Region', 'Metro':'City'}, inplace=True)
```

In the preceding code block, we change only a few column names rather than all of them. Let's call the `columns` property again to see if the column names have really been changed:

```
|data.columns  
Index([u'Date', u'RegionID', u'Region', u'State', u'City', u'County',  
       u'SizeRank', u'Zhvi'],  
      dtype='object')
```

Now, we have our new column names in the dataset.

We can also rename all columns after the data has been read, as follows:

```
|data.columns = ['Date', 'Region ID', 'Region Name', 'State', 'City', 'County',  
               'Size Rank', 'Price']
```



We have set the `columns` property to a list of names to which we want all the columns to be renamed.

In this section, we learned about various methods for renaming column levels in pandas. We learned how to rename columns after reading in the data, and we learned how to rename columns while we are reading data from the CSV files. We also saw how to rename either all or specific columns.

# Removing columns from a pandas DataFrame

In this section, we'll look at how to remove columns or rows from a dataset in pandas. We will come to understand the `drop()` method and the functionality of its parameters in detail.

To start with, we first import the `pandas` module into our Jupyter notebook:

```
| import pandas as pd
```

After this, we read our CSV dataset using the following code:

```
| data = pd.read_csv('data-titanic.csv', index_col=3)
| data.head()
```

The dataset should look something like the following:

	PassengerId	Survived	Pclass	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
Name											
Braund, Mr. Owen Harris	1	0	3	male	22.0	1	0	A/5 21171	7.2500	NaN	S
Cummings, Mrs. John Bradley (Florence Briggs Thayer)	2	1	1	female	38.0	1	0	PC 17599	71.2833	C85	C
Heikkinen, Miss. Laina	3	1	3	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
Futrelle, Mrs. Jacques Heath (Lily May Peel)	4	1	1	female	35.0	1	0	113803	53.1000	C123	S
Allen, Mr. William Henry	5	0	3	male	35.0	0	0	373450	8.0500	NaN	S

To remove a single column from our dataset, the pandas `drop()` method is used. The `drop()` method consists of two parameters. The first parameter is the name of the column that needs to be eliminated; the second parameter is the `axis`. This parameter tells the `drop` method whether it should drop a row or column, and sets `inplace` to `True`, which tells the method to drop it from the original DataFrame itself.

In this example, let's consider removing the `Ticket` column. The code for this is as follows:

```
|data.drop('Ticket', axis=1, inplace=True)
```

Once this is executed, our dataset should look something like the following:

```
|data.head()
```

	PassengerId	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Cabin	Embarked
Name										
Braund, Mr. Owen Harris	1	0	3	male	22.0	1	0	7.2500	NaN	S
Cumings, Mrs. John Bradley (Florence Briggs Thayer)	2	1	1	female	38.0	1	0	71.2833	C85	C
Heikkinen, Miss. Laina	3	1	3	female	26.0	0	0	7.9250	NaN	S
Futrelle, Mrs. Jacques Heath (Lily May Peel)	4	1	1	female	35.0	1	0	53.1000	C123	S
Allen, Mr. William Henry	5	0	3	male	35.0	0	0	8.0500	NaN	S

If we observe closely, it is clear that the `Ticket` column has been removed, or dropped, from our dataset.

To remove multiple columns, we pass the ones that need to be dropped as a list to the `drop()` method. All the other parameters of the `drop()` method will stay the same.

Let's look at an example of how to eliminate rows using the `drop()` method.

In this example, we shall be dropping multiple rows. Thus, instead of passing the column names, we will pass the row index labels in the form of a list. The following code will be used to do this:

```
data.drop(['Parch', 'Fare'], axis=1, inplace=True)  
data.head()
```

	PassengerId	Survived	Pclass	Sex	Age	SibSp	Cabin	Embarked
Name								
Braund, Mr. Owen Harris	1	0	3	male	22.0	1	NaN	S
Cumings, Mrs. John Bradley (Florence Briggs Thayer)	2	1	1	female	38.0	1	C85	C
Heikkinen, Miss. Laina	3	1	3	female	26.0	0	NaN	S
Futrelle, Mrs. Jacques Heath (Lily May Peel)	4	1	1	female	35.0	1	C123	S
Allen, Mr. William Henry	5	0	3	male	35.0	0	NaN	S

As a result, the two rows, which correspond to the names of the passengers, passed to the `drop()` method, will be taken off the dataset.

We will now proceed to take a closer look at how to work with date and time data.

# Working with date and time series data

In this section, we will take a closer look at how to work with date and time series data in pandas. We'll also see how to:

- Convert strings to `datetime` types for advanced `datetime` series operations
- Select and filter `datetime` series data
- Explore properties of series data

We will begin by importing the `pandas` module in to our Jupyter notebook:

```
|import pandas as pd
```

For this example, let's create our own `DataFrame` dataset. We can do this by using the following code:

```
|dataset = pd.DataFrame({'DOB': ['1976-06-01', '1980-09-23', '1984-03-30', '1991-12-31', '1994-10-2', '1973-11-11'],  
|                          'Sex': ['F', 'M', 'F', 'M', 'M', 'F'],  
|                          'State': ['CA', 'NY', 'OH', 'OR', 'TX', 'CA'],  
|                          'Name': ['Jane', 'John', 'Cathy', 'Jo', 'Sam', 'Tai']}))
```

This dataset contains four columns and five rows corresponding to five fictional people. One of the rows present in our dataset is `DOB`, which contains the date of birth of the five people.

It is essential to check if the data present in the `DOB` column is the right datatype. To do this, we use the following code:

```
|dataset.dtypes
```

```
DOB      object
Name     object
Sex      object
State    object
dtype: object
```

As observed in the output, it is possible that the `DOB` column was set to the `object` or `string` datatype during creation. To change this to the `datetime` datatype, we use the `to_datetime()` method and pass the `DOB` column to it as follows:

```
|dataset.DOB = pd.to_datetime(dataset.DOB)
```

Once again, we can verify that `DOB` has been set to the `datetime` datatype by using the following code:

```
|dataset.dtypes
```

```
DOB      object
Name     object
Sex      object
State    object
dtype: object
```

Before moving on to selecting and filtering the `datetime` series, we need to make sure that the index is set for the `DOB` column. To do this, we use the following code:

```
|dataset.set_index('DOB', inplace=True)
```

After this, our `DOB` column is ready to be explored. If we want to take a look at the dataset, we can do so by using the codeword `dataset`, as follows:

```
|dataset
```

	Name	Sex	State
DOB			
1976-06-01	Jane	F	CA
1980-09-23	John	M	NY
1984-03-30	Cathy	F	OH
1991-12-31	Jo	M	OR
1994-10-2	Sam	M	TX
1973-11-11	Tai	F	CA

Before we start with filtering, we need to understand that there are four possible ways to filter the data present in the `DOB` column. They are as follows:

- **Records for a single year:** To display the records for a single year, we use the following code:

```
| dataset['1980']
```

	Name	Sex	State
DOB			
1980-09-23	John	M	NY

This code means that all records that exist for the year `1980` will be displayed. pandas does not require us to mention the entire date, as even a part of the date will help us yield results.

- **Records for and after a particular year:** To display all of the records for and after a particular year, we use the following code:

```
| dataset['1980':]
```

	Name	Sex	State
DOB			
1980-09-23	John	M	NY

- **Records up until a particular year:** To display all records up to and including a particular year, we need to use the following code:

```
| dataset[':1980']
```

	Name	Sex	State
DOB			
1976-06-01	Jane	F	CA
1980-09-23	John	M	NY
1973-11-11	Tai	F	CA

- **Records that exist in a range of years:** To display the records for a given range of years, we can use the following code:

```
| dataset['1980':'1984']
```

	Name	Sex	State
DOB			
1980-09-23	John	M	NY
1984-03-30	Cathy	F	OH

We can also use the time series properties to make the most efficient use of the `datetime` series data. The drawback in using this functionality is that the `datetime` field needs to be a column, not a row.

This can be done by resetting the `DOB` to an index. This is done as follows:



```
|dataset.reset_index(inplace=True)
```

We would also need to get the corresponding day of the year for each value in the `datetime` column. This can be done by calling the `dayofyear` property, as follows:

```
|dataset.DOB.dt.dayofyear
```

```
0      153
1      267
2       90
3      365
4      275
5      315
Name: DOB, dtype: int64
```

We can also display the day of the week by calling the `weekday_name` property, as follows:

```
|dataset.DOB.dt.weekday_name
```

```
0      Tuesday
1      Tuesday
2       Friday
3      Tuesday
4       Sunday
5       Sunday
Name: DOB, dtype: object
```

These are a few examples of the methods and properties of the `datetime` series data. There are a lot more that can be found in the pandas' reference documentation, available at.

# Handling SettingWithCopyWarning

In this section, we'll learn about the `SettingWithCopyWarning` warning, and ways to get around it.

We will also take a look at some scenarios where we might possibly encounter `SettingWithCopyWarning`, so that we can understand how to get rid of it.

Avid users of pandas have definitely run into `SettingWithCopyWarning`. Various websites, such as Stack Overflow and other forums, are filled with queries about dealing with this warning. It looks something like the following:

```
C:\Users\harish\Anaconda2\envs\python3\lib\site-packages\pandas\core\generic.py:2773: 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
self[name] = value
```

To understand how to get rid of it, we need to understand what `SettingWithCopyWarning` actually represents.

We all know that different data operations in pandas return either a view of the data or a copy. There is a chance that this can cause problems when the data is modified. The aim of `SettingWithCopyWarning` is to warn us that we may be trying to modify the original data when we may want to modify the copy

instead, or vice versa. This situation generally occurs during chained assignments.

The solution for this is to combine the chain of patients into a single operation using the `block` method. This helps pandas know which DataFrame it has to modify.

To get a better understanding of this, let's take look at the following example.

As always, we begin by importing the `pandas` module into our Jupyter Notebook, as follows:

```
|import pandas as pd
```

We then read in our CSV dataset:

```
|data = pd.read_csv('data-titanic.csv')
|data.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

After this, we proceed to create a scenario where we might encounter `SettingWithCopyWarning`. For this example, we select the records where the `Age` column is null, and set them to equal the mean of the values from the `Age` column. The following is the error generated:

```
|data[data.Age.isnull()].Age = data.Age.mean()
```

```
C:\Users\harish\Anaconda2\envs\python3\lib\site-packages\pandas\core\generic.py:2773: 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
self[name] = value
```

To confirm that our code doesn't work, we need to check if there are records that still have `Age` as null. This is done by using the following code:

```
|data[data.Age.isnull()].Age.head()
5      NaN
17     NaN
19     NaN
26     NaN
28     NaN
Name: Age, dtype: float64
```

It is clear that such records exist. To handle such a scenario, we use the `loc` method, as depicted here:

```
|data.loc[data.Age.isnull(), 'Age'] = data.Age.mean
```

At this point, we need to go back to confirm if the method has resolved `SettingWithCopyWarning`, which we do by using the following code line:

```
|data[data.Age.isnull()]

PassengerId  Survived  Pclass  Name  Sex  Age  SibSp  Parch  Ticket  Fare  Cabin  Embarked
```

---

As we can observe, the issue has not been resolved, so another way to handle the warning is by turning it off. We need to remember that the only reason we can and should turn it off is because it is a warning, and not an error. To do this, we set the `mode.chained_assignment` option to `None`:

```
|pd.set_option('mode.chained_assignment', None)
```

This is not a recommended solution, as this may affect the results of our operations. Another way to resolve this warning is by using the `is_copy` method. Here, we create a new copy of the DataFrame and set `is_copy` to `None`, as follows:

```
|data1 = data.loc[data.Age.isnull()]  
|data1.is_copy = None
```

Let's now look at how to apply a function to a pandas series or DataFrame.

# Applying a function to a pandas series or DataFrame

In this section, we will learn how to apply Python's pre-built and self-built functions to pandas data objects. We'll also learn about applying functions to a pandas series and pandas DataFrame.

We will begin by importing the `pandas` module into our Jupyter Notebook:

```
import pandas as pd
import numpy as np
```

We will then read in our CSV dataset:

```
data = pd.read_csv('data-titanic.csv')
data.head()
```

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Let's proceed to applying functions using pandas' `apply` method. In this example, we will creating a function using `lambda`, as follows:

```
func_lower = lambda x: x.lower()
```

Here, we pass a value `x` and converting it into lowercase. We then apply this function to the `Name` field in the dataset using the `apply()` method, as shown

here:

```
|data.Name.apply(func_lower)
0          braund, mr. owen harris
1  cumings, mrs. john bradley (florence briggs th...
2          heikkinen, miss. laina
3  futrelle, mrs. jacques heath (lily may peel)
4          allen, mr. william henry
5          moran, mr. james
6          mccarthy, mr. timothy j
7          palsson, master. gosta leonard
8  johnson, mrs. oscar w (elisabeth vilhelmina berg)
9          nasser, mrs. nicholas (adele achem)
10         sandstrom, miss. marguerite rut
11         bonnell, miss. elizabeth
12         saundercock, mr. william henry
13         andersson, mr. anders johan
14         vestrom, miss. hulda amanda adolfina
15         hewlett, mrs. (mary d kingcome)
16         rice, master. eugene
```

If you look closely, the values in the `Name` field have been converted into lowercase. Next, we see how to apply functions to values in multiple columns, or a whole DataFrame. We can use the `applymap()` method for this. It works in a fashion similar to the `apply()` method, but on multiple columns, or on the whole DataFrame. The following code depicts how to apply the `applymap()` method to the `Age` and `Pclass` columns:

```
|data[['Age', 'Pclass']].applymap(np.square)
```

	Age	Pclass
0	484.0	9
1	1444.0	1
2	676.0	9
3	1225.0	1
4	1225.0	9
5	NaN	9
6	2916.0	1
7	4.0	9
8	729.0	9
9	196.0	4

We also blast the `secure` method of Numpy apply to these two columns.

The preceding steps are for a predefined function. Let's now proceed to create our own function and then apply it to the values, as follows:

```
def my_func(i):
    return i + 20
```

The function that was created is a simple function that takes a value, adds 20 to it, and then returns the result. We use `applymap()` to apply this function to every value in the `Age` and `Pclass` columns, as shown here:

```
data[['Age', 'Pclass']].applymap(my_func)
```



	Age	Pclass
0	42.0	23
1	58.0	21
2	46.0	23
3	55.0	21
4	55.0	23
5	NaN	23
6	74.0	21
7	22.0	23
8	47.0	23
9	34.0	22

Let's move on to learning about merging and concatenating multiple DataFrames into one.

# Merging and concatenating multiple DataFrames into one

This section focuses on how to combine two or more DataFrames using the pandas `merge()` and `concat()` methods. We'll also explore the usage of the `merge()` method to join DataFrames in various ways.

We will begin by importing the `pandas` module. Let's create two DataFrames, where both contain the same parameters with similar data, but which are for different records:

```
dataset1 = pd.DataFrame({'Age': ['32', '26', '29'],  
                          'Sex': ['F', 'M', 'F'],  
                          'State': ['CA', 'NY', 'OH']},  
                          index=['Jane', 'John', 'Cathy'])  
dataset2 = pd.DataFrame({'Age': ['34', '23', '24', '21'],  
                          'Sex': ['M', 'F', 'F', 'F'],  
                          'State': ['AZ', 'OR', 'CA', 'WA']},  
                          index=['Dave', 'Kris', 'Xi', 'Jo'])  
  
dataset1  
dataset2
```

	Age	Sex	State
Jane	32	F	CA
John	26	M	NY
Cathy	29	F	OH

	Age	Sex	State
Dave	34	M	AZ
Kris	23	F	OR
Xi	24	F	CA
Jo	21	F	WA

In this example, let's put these two DataFrames together vertically. The pandas `concat()` method is used to do this with the two DataFrames passed as its parameters:

```
|pd.concat([dataset1, dataset2])
```

We can observe that `dataset2` has been appended to `dataset1` vertically.

Another way to concatenate datasets is to use the `append()` method. The results obtained by using this will be the same as the previous method:

```
|dataset1.append(dataset2)
```

	Age	Sex	State
Jane	32	F	CA
John	26	M	NY
Cathy	29	F	OH
Dave	34	M	AZ
Kris	23	F	OR
Xi	24	F	CA
Jo	21	F	WA

So far, we have concatenated rows in datasets, but it is also possible to concatenate columns. For this example, let's create two new datasets with the same row levels but different columns, as follows:

```
|dataset1 = pd.DataFrame({'Age': ['32', '26', '29'],  
|                        'Sex': ['F', 'M', 'F'],  
|                        'State': ['CA', 'NY', 'OH']},  
|                        index=['Jane', 'John', 'Cathy'])  
|dataset2 = pd.DataFrame({'City': ['SF', 'NY', 'Columbus'],  
|                        'Work Status': ['No', 'Yes', 'Yes']},  
|                        index=['Jane', 'John', 'Cathy'])
```

In such a scenario, we would concatenate horizontally. To concatenate by columns, we need to pass the `axis` parameter as `1`:

```
|pd.concat([dataset1, dataset2], axis=1)
```



	Age	Name	Sex	State
0	32	Jane	F	CA
1	26	John	M	NY
2	29	Cathy	F	OH
3	23	Sarah	F	TX

	City	Name	Work Status
0	SF	Jane	No
1	NY	John	Yes
2	Columbus	Cathy	Yes
3	Austin	Rob	Yes

To perform an inner merge on these datasets, we pass the DataFrames to the `merge()` method. We also specify the column on which the merge has to be carried out, while making sure we specify that it's an inner merge. Your dataset should look similar to the following table:

```
|pd.merge(dataset1, dataset2, on='Name', how='inner')
```

	Age	Name	Sex	State	City	Work Status
0	32	Jane	F	CA	SF	No
1	26	John	M	NY	NY	Yes
2	29	Cathy	F	OH	Columbus	Yes

This now means that we have the data from both datasets together. This contains only those rows that have common labels in both DataFrames. Next, we carry out the outer merge. This is done by passing the `how` parameter as `left` to the `merge()` method:

```
|pd.merge(dataset1, dataset2, on='Name', how='left')
```

	Age	Name	Sex	State	City	Work Status
0	32	Jane	F	CA	SF	No
1	26	John	M	NY	NY	Yes
2	29	Cathy	F	OH	Columbus	Yes
3	23	Sarah	F	TX	NaN	NaN

The result of this operation is that the rows which are in both datasets, as well as rows only present in the first dataset, will be retained. Rows that exist in the second dataset only will be discarded. To do a right merge, we set the `how` parameter to `right`:

```
|pd.merge(dataset1, dataset2, on='name', how='right')
```

	Age	Name	Sex	State	City	Work Status
0	32	Jane	F	CA	SF	No
1	26	John	M	NY	NY	Yes
2	29	Cathy	F	OH	Columbus	Yes
3	NaN	Rob	NaN	NaN	Austin	Yes

To retain everything, we do a full outer merge. The full outer merge is done by passing the `how` parameter as `outer`:

This now contains all the rows, irrespective of whether they exist in one dataset or another, or both, even for the columns which have no values and are marked as NaN.

# Summary

In this chapter, we learned about various pandas techniques to manipulate and reshape data. We learned how to modify a pandas DataFrame using the `inplace` parameter. We also learned the scenarios for which you can use the `groupby` method. We saw how to handle missing values in pandas. We explored indexing in pandas DataFrames, along with renaming and removing columns in a pandas DataFrame. We learned how to work with and transform date and time data. We learned how to handle `SettingWithCopyWarning`, and also saw how to apply functions to a pandas series or DataFrame. Lastly, we learned how to merge and concatenate multiple DataFrames.

In the next chapter, we'll learn about techniques for visualizing data like a pro, using the `seaborn` Python library.



# Visualizing Data Like a Pro

In this chapter, we'll learn about the advanced techniques of data visualization using the seaborn data visualization library.

In particular, we will cover the following topics:

- How to get started with seaborn
- The features of seaborn
- Drawing different kinds of plots
- Plotting categorical plots with seaborn
- Plotting with Data-Aware Grids

# Controlling plot aesthetics

In this section, we will learn how to control plot aesthetics using the seaborn plotting library. We'll learn how to install and get started with seaborn, and about the models we need to import. We'll explore some of the seaborn plotting methods to draw a few different kinds of plots. We'll also see how to control and change plot aesthetics using various seaborn methods and properties.

Before we can start creating plots with seaborn, we need to install it. We have been using Anaconda throughout this book to install various Python libraries, so we'll continue with that. To install seaborn, execute the following command:

```
|conda install seaborn
```

Make sure to run your command-line program in administrator mode before executing the command. Now we need to import the Python modules we need for this section, as follows:

```
|import pandas as pd  
|from matplotlib import pyplot as plt  
|%matplotlib inline  
|import seaborn as sns
```

We need to import pandas' Matplotlib and seaborn modules. We are using Matplotlib's inline magic command to make sure our plots are displayed correctly in the Jupyter Notebook along with the code.

Next, we read in our dataset using pandas and the following command:

```
|df = pd.read_csv('data-alcohol.csv')  
|df.head()
```

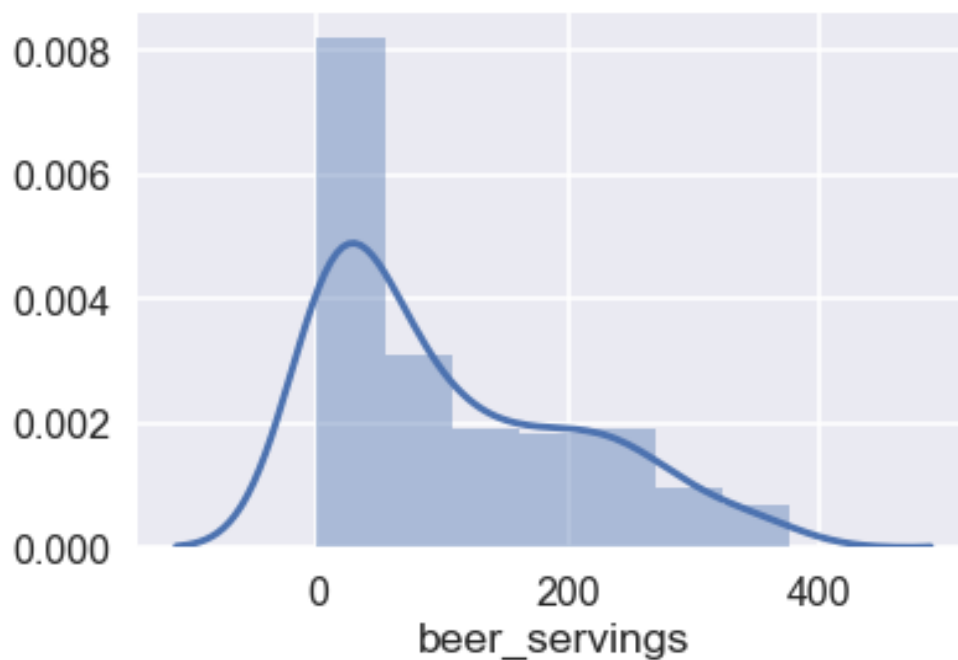
Our dataset is a CSV file. It consists of the alcohol consumption data for various countries. This data is available at the following link: <https://github.com/fivethirtyeight/data/tree/master/alcohol-consumption>.

# Our first plot with seaborn

In this section, we will create a distribution plot using just one variable, as follows:

```
|sns.distplot(df.beer_servings)
```

Here, `sns` refers to seaborn, which we imported earlier as `sns`. We now need to call the method of seaborn as `distplot` and pass in the column name from the data we read earlier. This should give us a nice distribution plot with a single line of code, as shown in the following screenshot:



This single line demonstrates the power and simplicity of the seaborn library.

# Changing the plot style with `set_style`

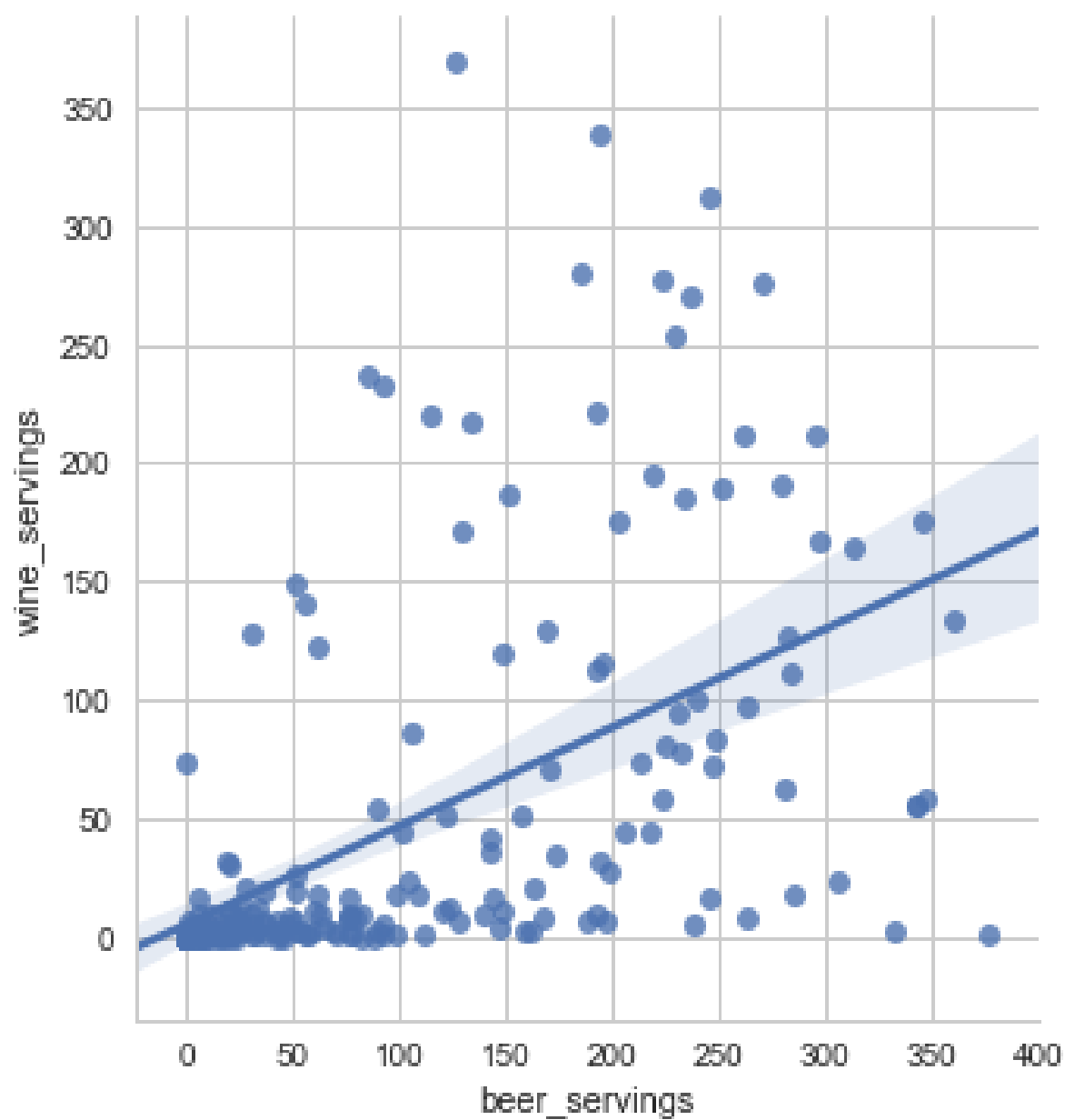
Now it's time to get into changing plot aesthetics with seaborn. While doing this, we'll also explore a number of different plot types, which we can draw with seaborn.

# Setting the plot background to a white grid

The default plot style is a blue grid. We can change this to `whitegrid` with the following command:

```
sns.set()  
sns.set_style("whitegrid")  
sns.lmplot(x='beer_servings', y='wine_servings', data=df);
```

Seaborn provides a method called `set_style`, which we call and then pass `whitegrid` as the parameter. We then call our plotting method to draw our scatterplot. We are using seaborn's `lmplot` method for this. We then pass two column names from our dataset as `x` and `y`, and set the `data` parameter to our pandas DataFrame. We should now have a scatterplot with a white grid background, as shown in the following screenshot:

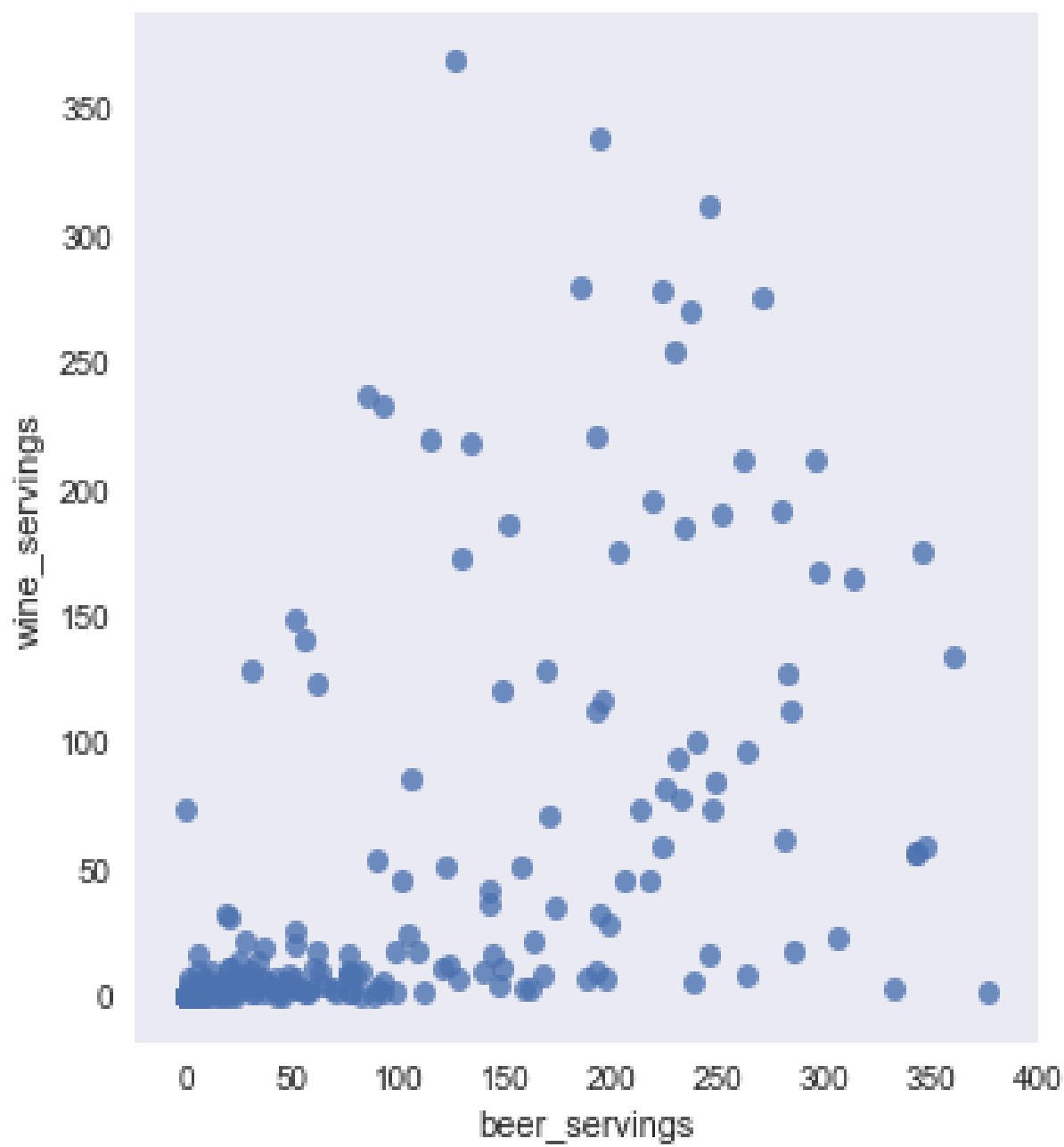


# Setting the plot background to dark

We will now look at how to set the plot background to `dark` and with no grid. To do this, we set the style to `dark` with the following command:

```
sns.set()  
sns.set_style("dark")  
sns.lmplot(x='beer_servings', y='wine_servings', data=df, fit_reg=False);
```

You may have noticed that we have another line of code at the start, which is `sns.set()`. By calling this, we are resetting the plot aesthetics to the default before making any changes. We do this to make sure the changes we made previously do not impact our grand plot, as follows:



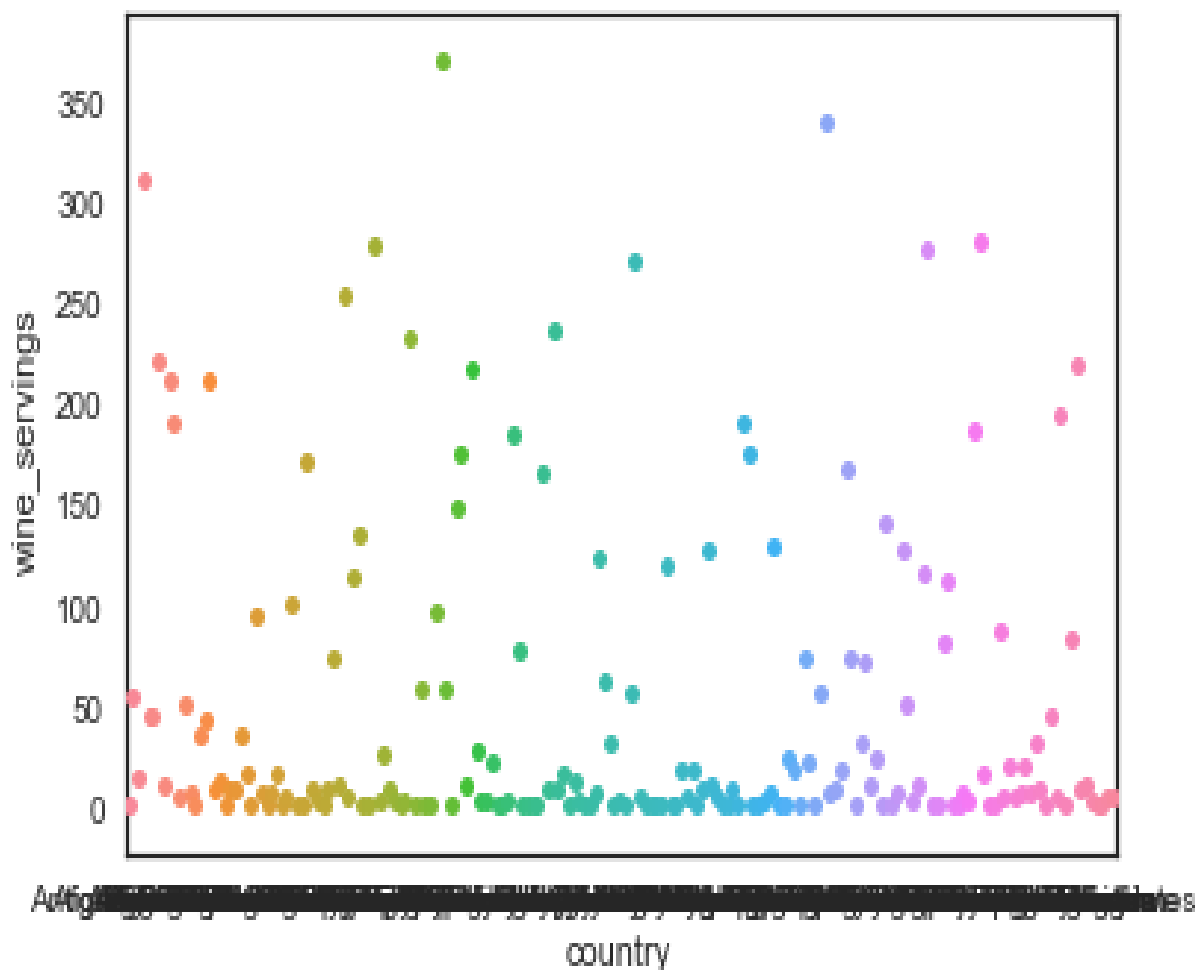


# Setting the background to white

We can also set the plot's background to solid `white`, and with no grid, with the following code:

```
sns.set()  
sns.set_style("white")  
sns.swarmplot(x='country', y='wine_servings', data=df);
```

The output is as follows:

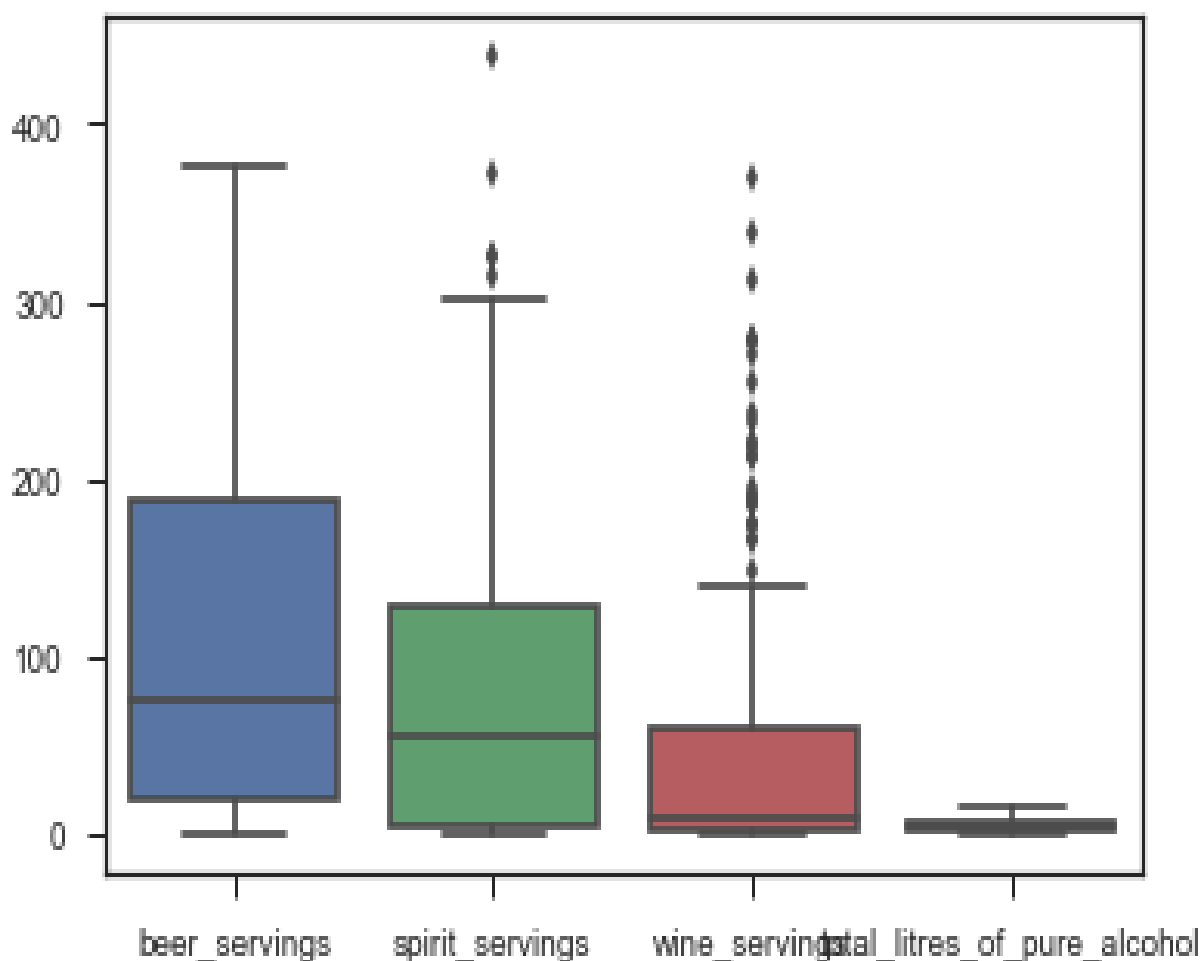


# Adding ticks

We can add ticks to the plot by setting `style` to `ticks`, as shown in the following code:

```
sns.set()  
sns.set_style("ticks")  
sns.boxplot(data=df);
```

The preceding code should give us the following output:



Here, we have also demonstrated how to create a box plot, which is done by calling the `boxplot` method from seaborn.

# Customizing styles

In seaborn, we can customize preset styles even further than previously discussed. Let's show you what we can do!

# Style parameters

Let's first take a look at all of the parameters these styles are made up of. We can get the parameters by calling the `axes_style` method on seaborn, as follows:

```
|sns.axes_style()
```

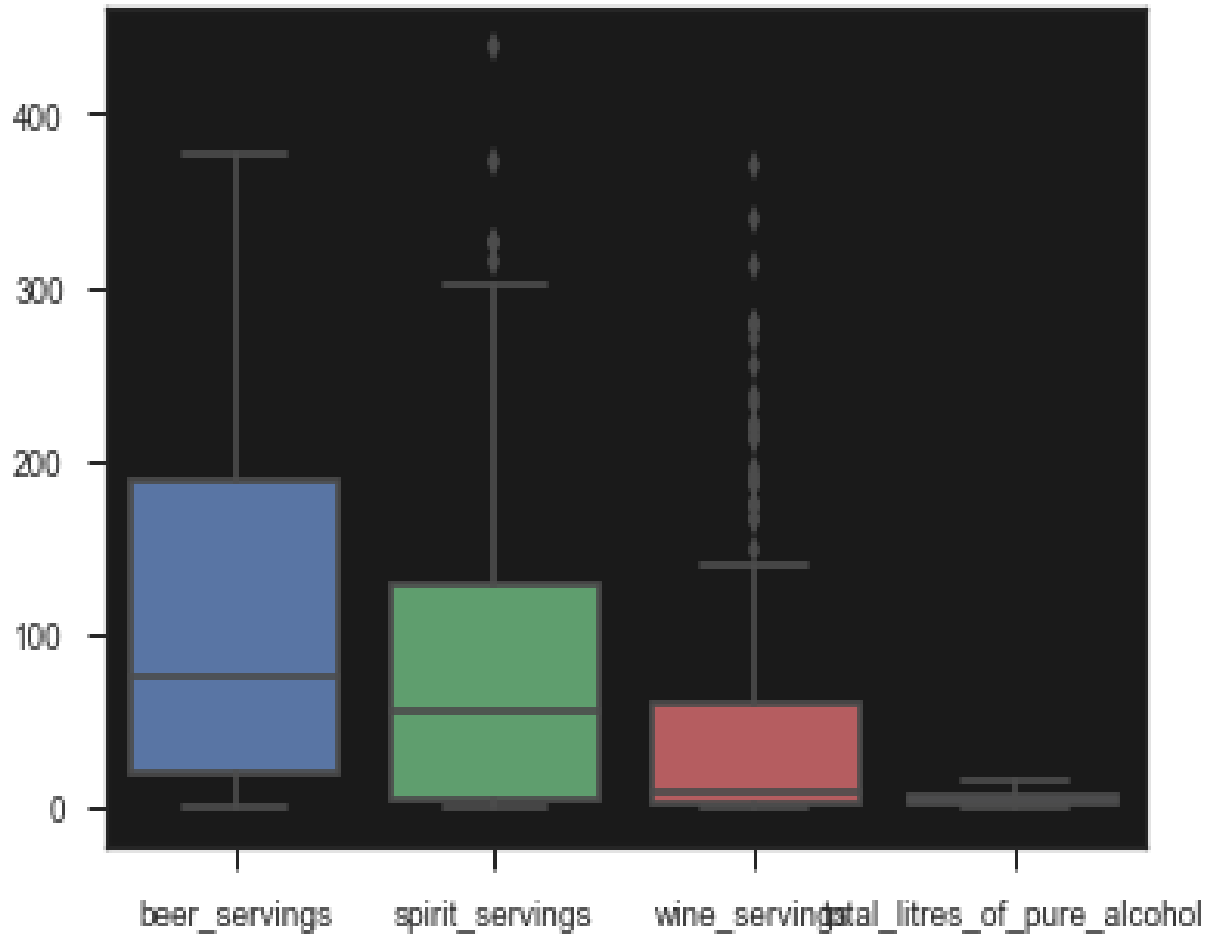
The output of the preceding code is as follows:

```
{'axes.axisbelow': True,
 'axes.edgecolor': '.15',
 'axes.facecolor': 'white',
 'axes.grid': False,
 'axes.labelcolor': '.15',
 'axes.linewidth': 1.25,
 'figure.facecolor': 'white',
 'font.family': ['sans-serif'],
 'font.sans-serif': ['Arial',
 'DejaVu Sans',
 'Liberation Sans',
 'Bitstream Vera Sans',
 'sans-serif'],
 'grid.color': '.8',
 'grid.linestyle': '-',
 'image.cmap': 'rocket',
 'legend.frameon': False,
 'legend.numpoints': 1,
 'legend.scatterpoints': 1,
 'lines.solid_capstyle': 'round',
 'text.color': '.15',
 'xtick.color': '.15',
 'xtick.direction': 'out',
 'xtick.major.size': 6.0,
 'xtick.minor.size': 3.0,
 'ytick.color': '.15',
```

Each one of the preceding parameters can be customized further. Let's try customizing one of them, as shown in the following snippet:

```
sns.set()
sns.set_style("ticks", {"axes.facecolor": ".1"})
sns.boxplot(data=df);
```

In the preceding code, we set the `style` to `ticks`, which has a solid white background, but we can customize this even further by setting `facecolor` separately. For the preceding code, we will be getting the following output:



Note that we can add more parameters to this dictionary and continue customizing the plot.

# Plotting context presets

Seaborn also provides some preset style context. For example, the default style context we have been using so far is called `notebook`. However, there are a few more, including one called `paper`. This context is set using a method called `set_context`, where we pass `paper` as the parameter, as follows:

```
sns.set()  
sns.set_context("paper")  
sns.lmplot(x='beer_servings', y='wine_servings', data=df);
```

The output for the previous code will be as follows:



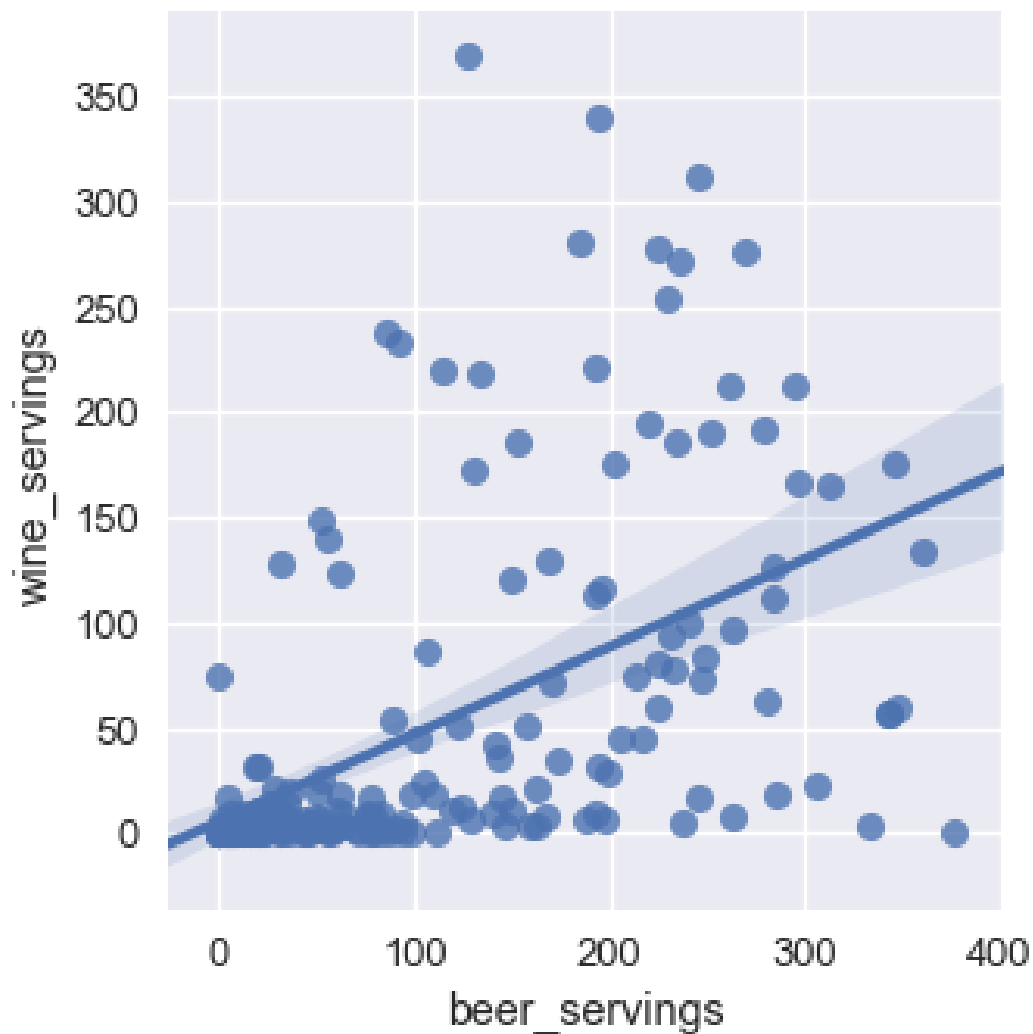
There are a few more contexts available; for example, one called `talk`. We can set that context with `talk`, as follows:

```
|sns.set()  
|sns.set_context("talk")
```



```
|plt.figure(figsize=(8, 6))  
|sns.lmplot(x='beer_servings', y='wine_servings', data=df);
```

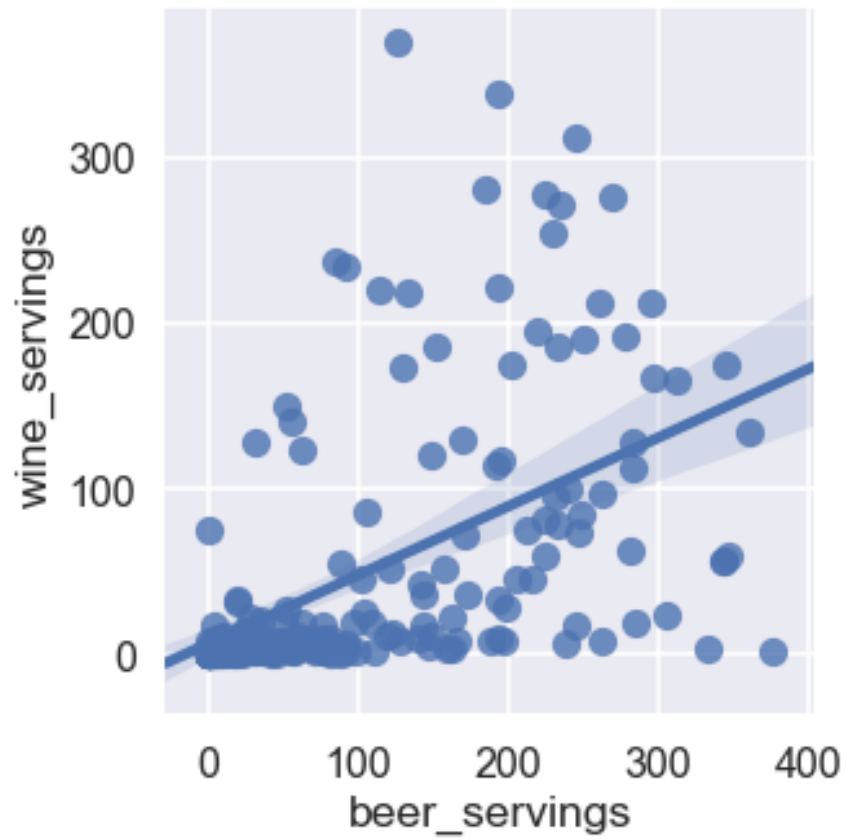
The output of the preceding code is as follows:



Another context we can use is called `poster`, which is set with the following code:

```
|sns.set()  
|sns.set_context("poster")  
|plt.figure(figsize=(8, 6))  
|sns.lmplot(x='beer_servings', y='wine_servings', data=df);
```

The output is as follows:



# Choosing the colors for plots

In this section, we will learn about using color palettes to customize plot colors in seaborn. We will explore some of the color palettes provided by seaborn and Matplotlib. We'll learn how to change a plot's colors by setting a different palette, and we'll also learn how to create our own palettes with custom colors.

Let's start by importing the modules we need in our Jupyter Notebook with the following code:

```
import pandas as pd
from matplotlib import pyplot as plt
%matplotlib inline
import seaborn as sns
```

We need to import pandas, Matplotlib, and seaborn. We then need to read in our CSV dataset; we do that by using the `read_csv` method, as follows:

```
df = pd.read_csv('data-alcohol.csv')
df.head()
```

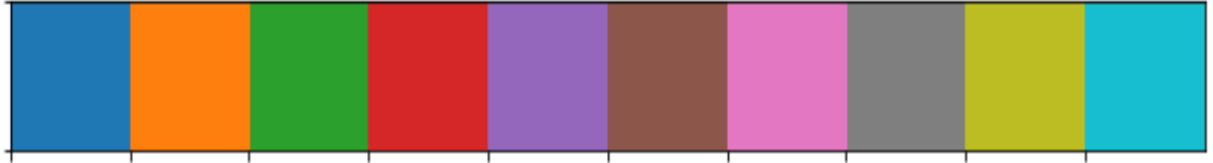
The output of the preceding code is as follows:

	country	beer_servings	spirit_servings	wine_servings	total_litres_of_pure_alcohol
0	Afghanistan	0	0	0	0.0
1	Albania	89	132	54	4.9
2	Algeria	25	0	14	0.7
3	Andorra	245	138	312	12.4
4	Angola	217	57	45	5.9

We now need to use seaborn to call the `color_palette` method to get the current palette, which is set by default. We then use the `palplot` method to display these colors, as follows:

```
| sns.palplot(sns.color_palette())
```

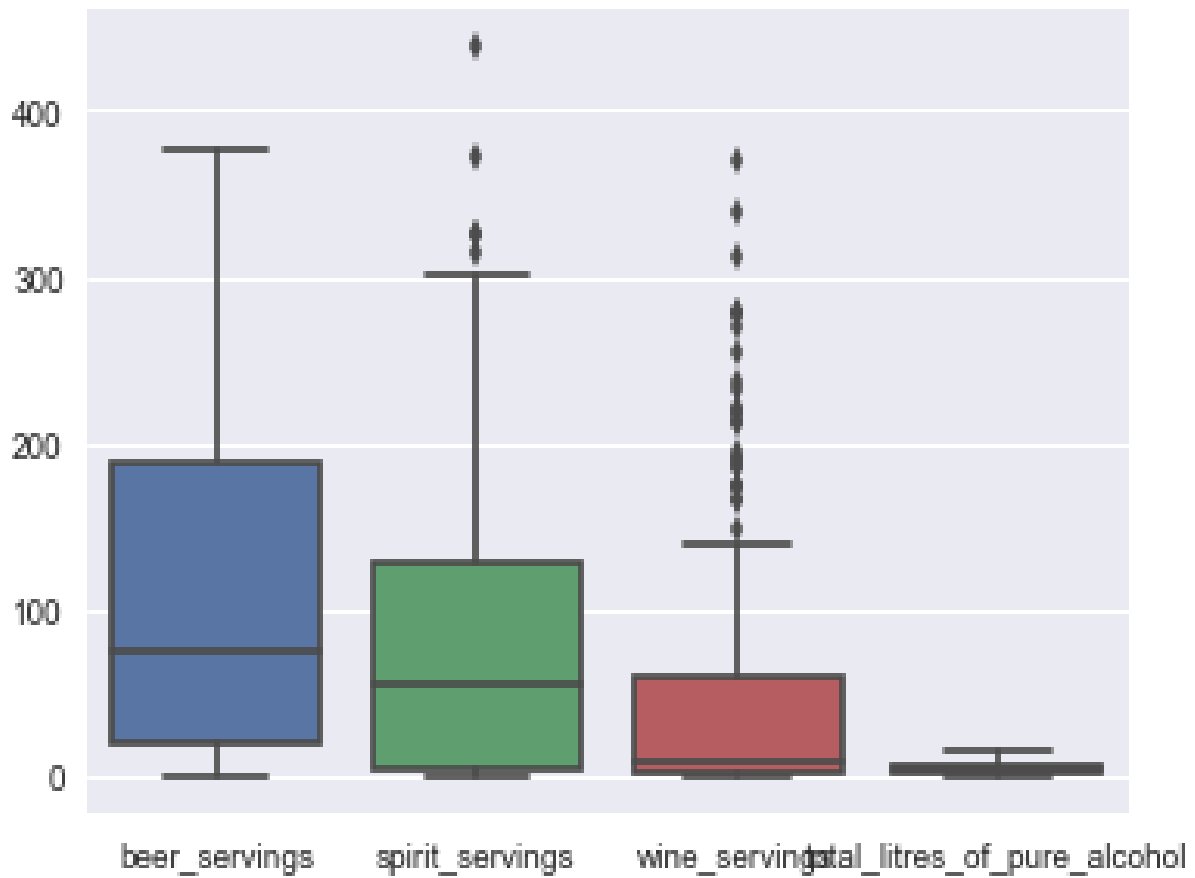
The output of the preceding code is as follows:



Now let's look at how this color palette looks in a plot:

```
| sns.set()  
| sns.boxplot(data=df);
```

The output from the preceding code should look like the following screenshot:



Here, we have drawn a box plot on our dataset. You may notice that the color scheme looks similar to what we saw earlier when we printed the default color palette.

# Changing the color palette

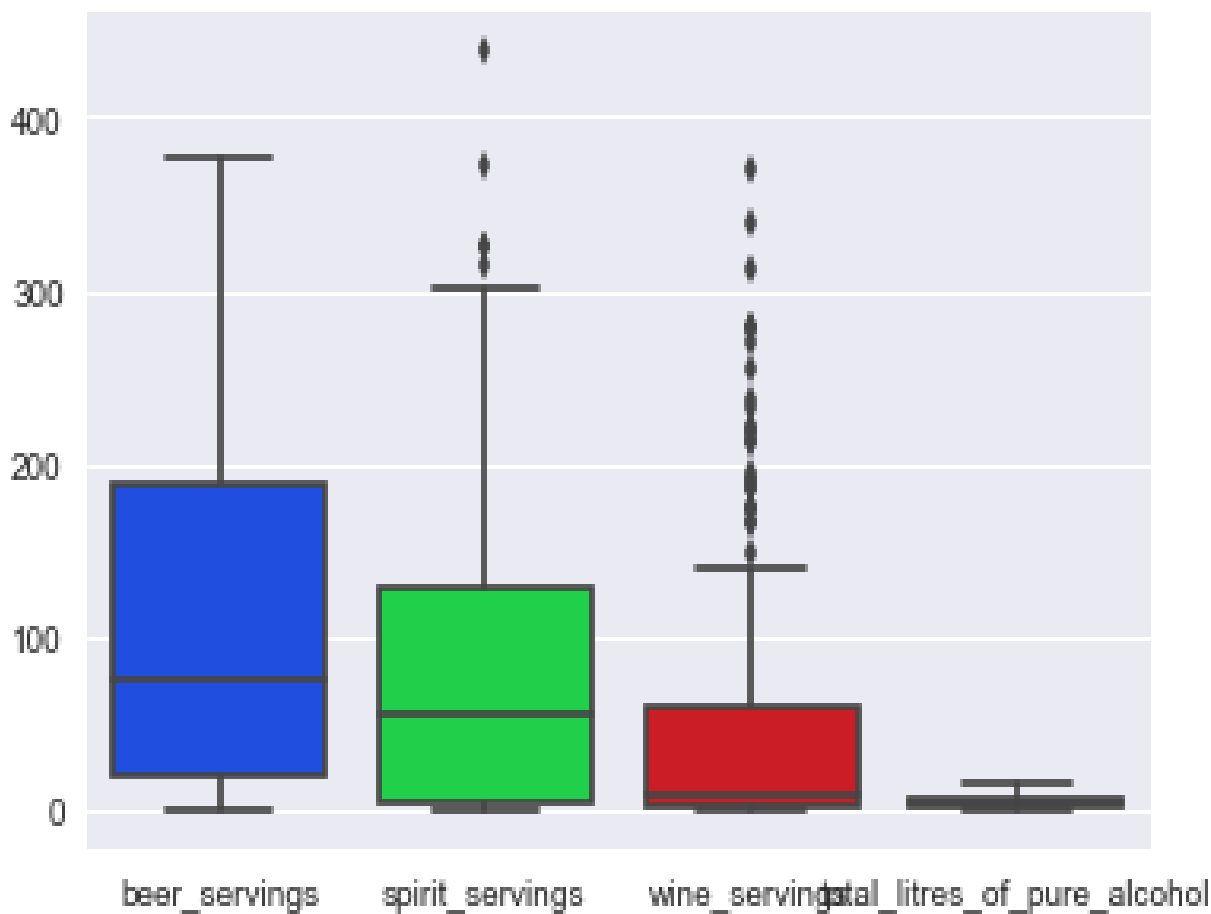
Let's move on and change the color palette to see how it impacts the plot's colors. The following code sets the color palette to `bright` (one of seaborn's predefined palettes):

```
|sns.set_palette("bright")
```

Let's see how it changes our plot's colors with the following command:

```
|sns.boxplot(data=df);
```

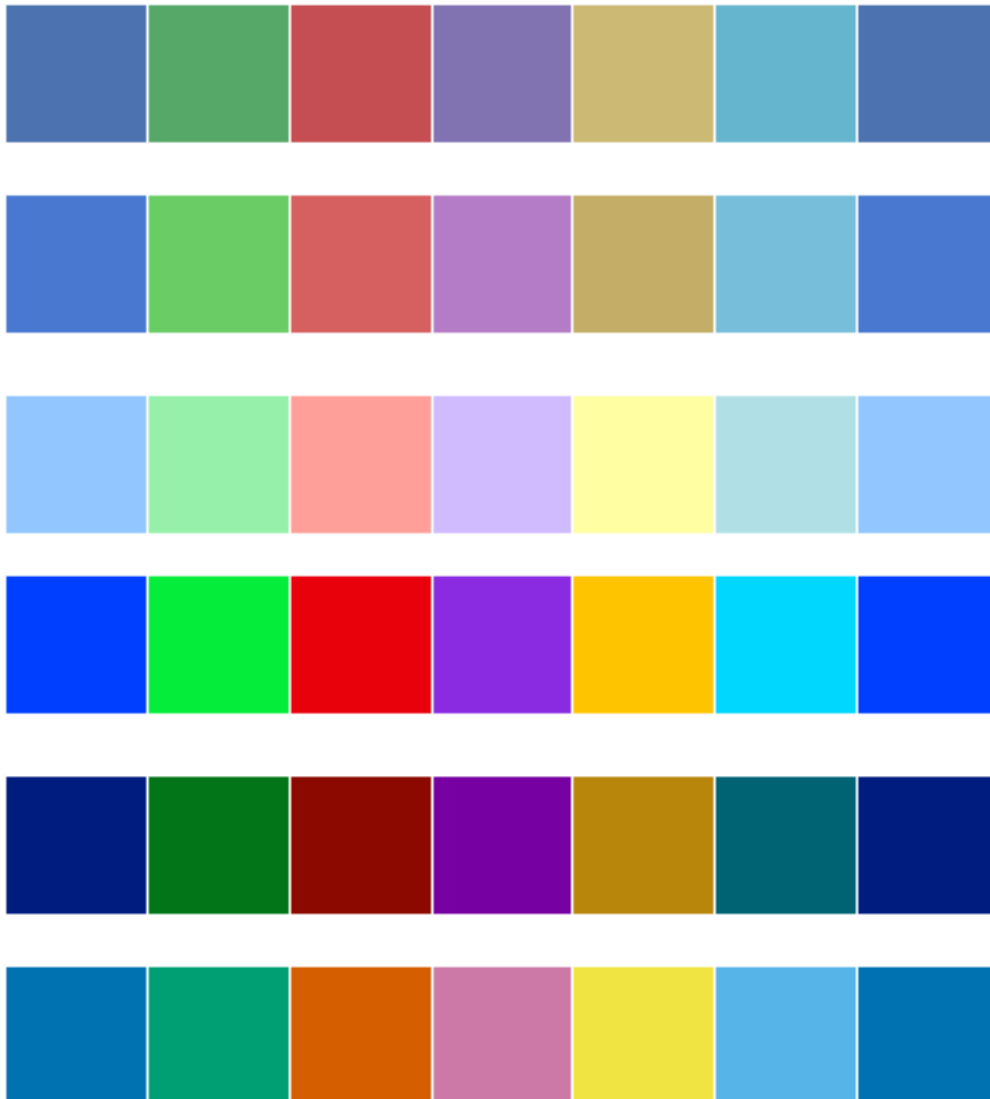
The output should now look like the following screenshot:



As you can see, our plot's color scheme has completely changed due to the new palette we set. `bright` is not the only predefined color palette in seaborn; there are a few others, including `deep`, `muted`, `pastel`, `bright`, `dark`, and `colorblind`, as follows:

```
sns.palplot(sns.color_palette("deep", 7))
sns.palplot(sns.color_palette("muted", 7))
sns.palplot(sns.color_palette("pastel", 7))
sns.palplot(sns.color_palette("bright", 7))
sns.palplot(sns.color_palette("dark", 7))
sns.palplot(sns.color_palette("colorblind", 7))
```

The output of each palette is as follows:



Seaborn also has the ability to set Matplotlib's colormaps as color palettes; for example:

```
|sns.palplot(sns.color_palette("RdBu", 7))  
|sns.palplot(sns.color_palette("Blues_d", 7))
```

The output from the preceding command is as follows:



Now let's use one of the Matplotlib colormaps as a color palette; we do this with the following command:

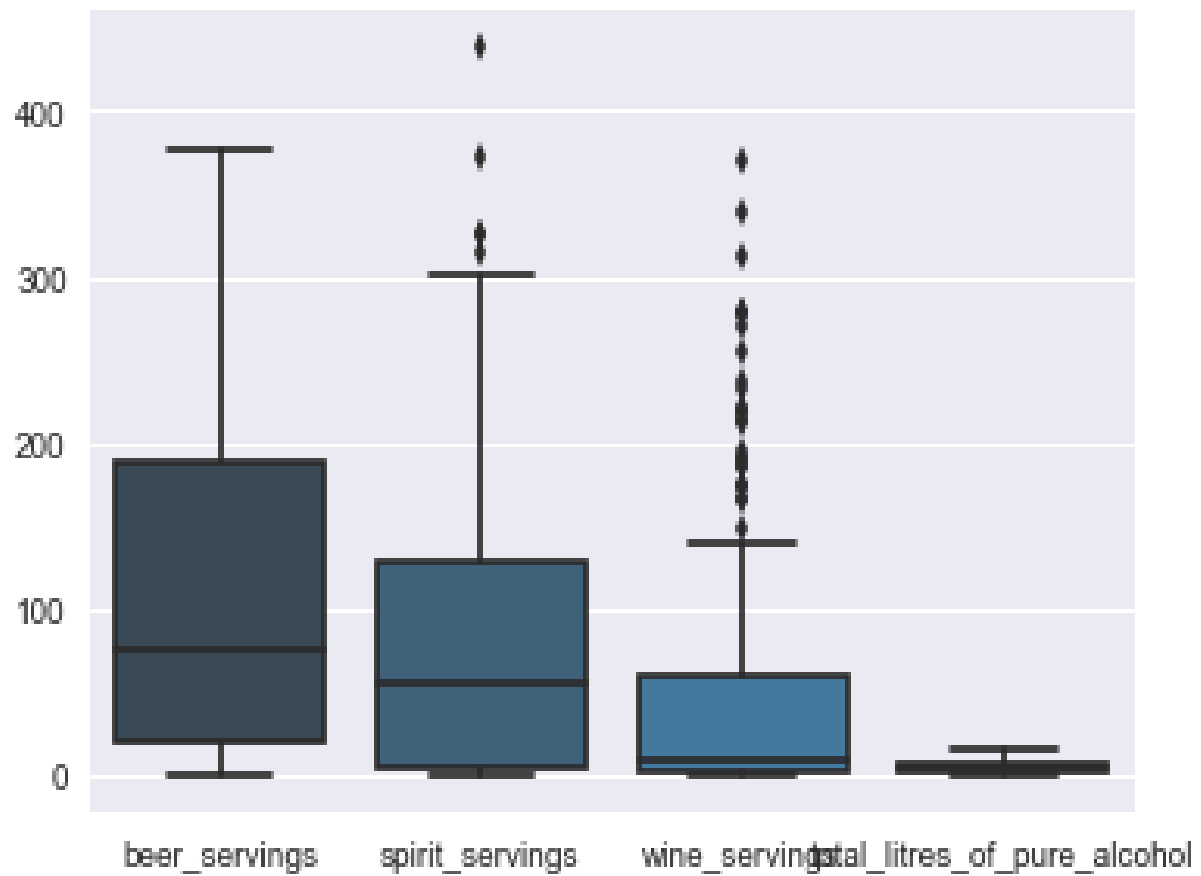
```
|sns.set_palette("Blues_d")
```

Here, we are setting the palette to `Blues_d`, which is a Matplotlib colormap. Let's now redraw our plot with the following code to see its impact:

```
|sns.boxplot(data=df);
```

The output from the preceding command should look like the following screenshot:





As you can see, our plot now has a color palette from the blue colormap.

# Building custom color palettes

To build custom palettes, we first need to create a list and assign the colors we want to it, as follows:

```
my_palette = ['#4B0082', '#0000FF', '#00FF00', '#FFFF00', '#FF7F00',  
              '#FF0000']  
sns.set_palette(my_palette)  
sns.palplot(sns.color_palette())
```

The output is as follows:

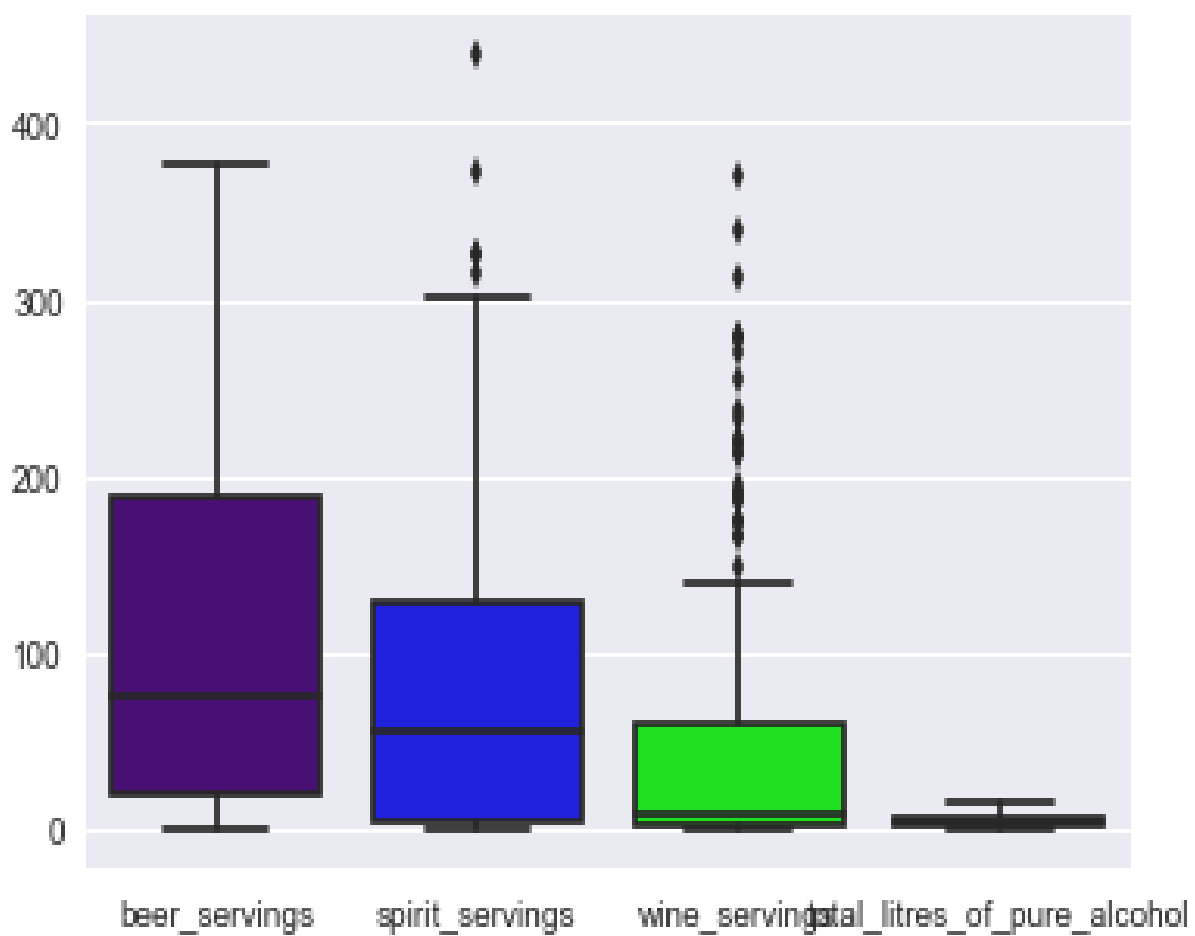


In the preceding screenshot, we created a new palette called `my_palette` with seven colors. We then set the palette to our newly created palette, which shows us what the colors look like.

Let's see how our plot looks with our new custom palette using the following command:

```
|sns.boxplot(data=df);
```

The output from the preceding command should look like the following screenshot:



# Plotting categorical data

In this section, we'll learn about the various categorical plots supported by seaborn and how to draw them. We will demonstrate how to draw plots including a scatterplot, swarmplots, box plots, bar plots, and more. We'll also learn how to draw wide-form categorical plots.

Let's start by importing our pandas module in our Jupyter Notebook with the following code:

```
import pandas as pd
from matplotlib import pyplot as plt
%matplotlib inline
import seaborn as sns
```

Along with pandas, we also need to import the Matplotlib and seaborn Python libraries. We then read in our CSV dataset, as follows:

```
df = pd.read_csv('data_simpsons_episodes.csv')
df.head()
```

Our dataset in this section is for the famous animated TV series, *The Simpsons*:

id	title	original_air_date	production_code	season	number_in_season	number_in_series	us_viewers_in_millions	views	imdb_rating	imdb_votes
0	10 Homer's Night Out	1990-03-25	7G10	1	10	10	30.3	50816.0	7.4	1511.0
1	12 Krusty Gets Busted	1990-04-29	7G12	1	12	12	30.4	62561.0	8.3	1716.0
2	14 Bart Gets an "F"	1990-10-11	7F03	2	1	14	33.6	59575.0	8.2	1638.0
3	17 Two Cars in Every Garage and Three Eyes on Eve...	1990-11-01	7F01	2	4	17	26.1	64959.0	8.1	1457.0
4	19 Dead Putting Society	1990-11-15	7F08	2	6	19	25.4	50691.0	8.0	1366.0

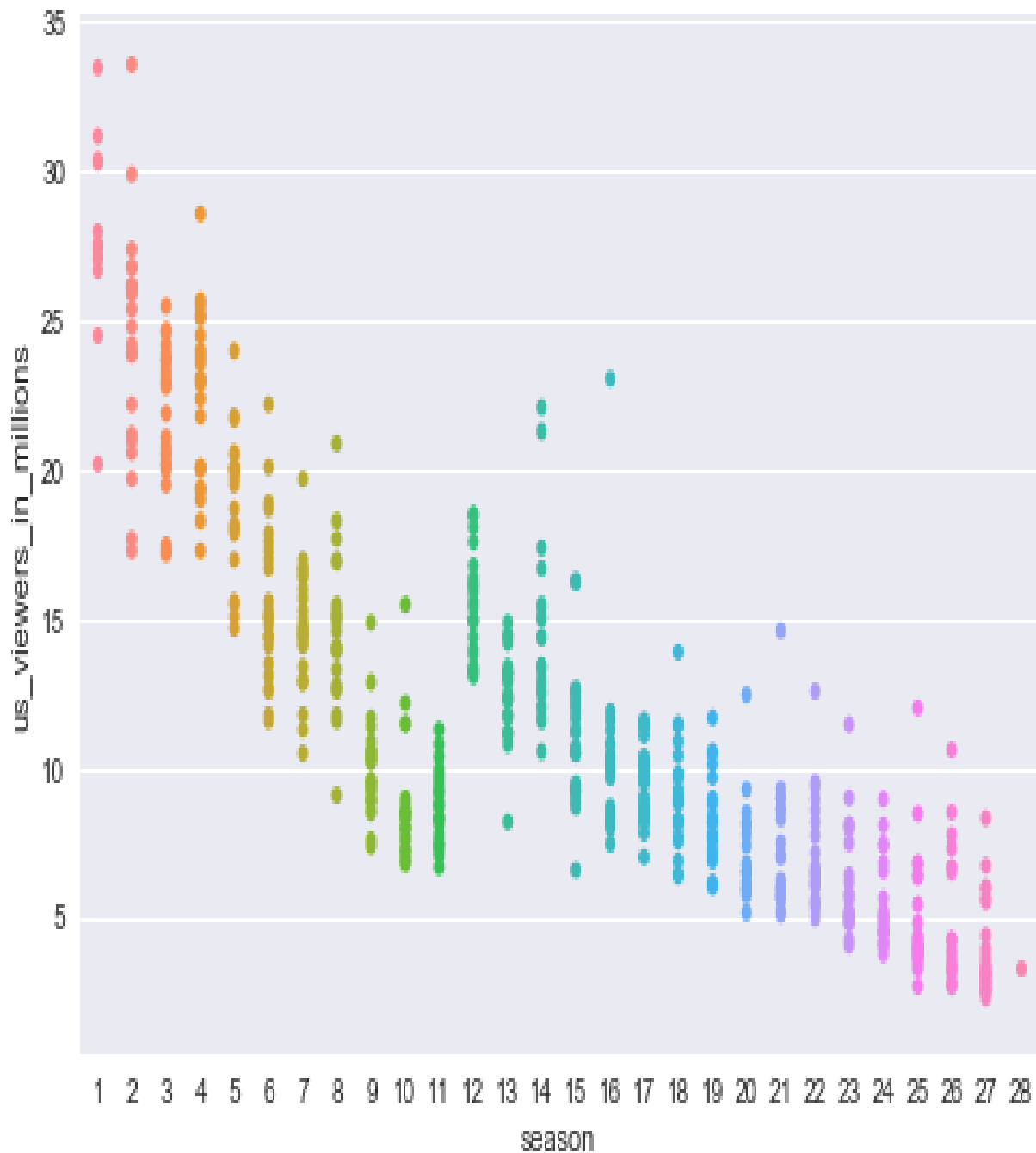
The preceding dataset includes release dates, viewership numbers, ratings, and a bunch of other observations for each *The Simpsons* episode.

# Scatterplot

Let's start with drawing a scatterplot; we do this with the following command:

```
|sns.stripplot(x="season", y="us_viewers_in_millions", data=df);
```

The output should be as follows:



Here, we have used seaborn's `stripplot` method. We plotted the season number on the x axis and US viewers in millions on the y axis. We also specified the name of the DataFrame used.

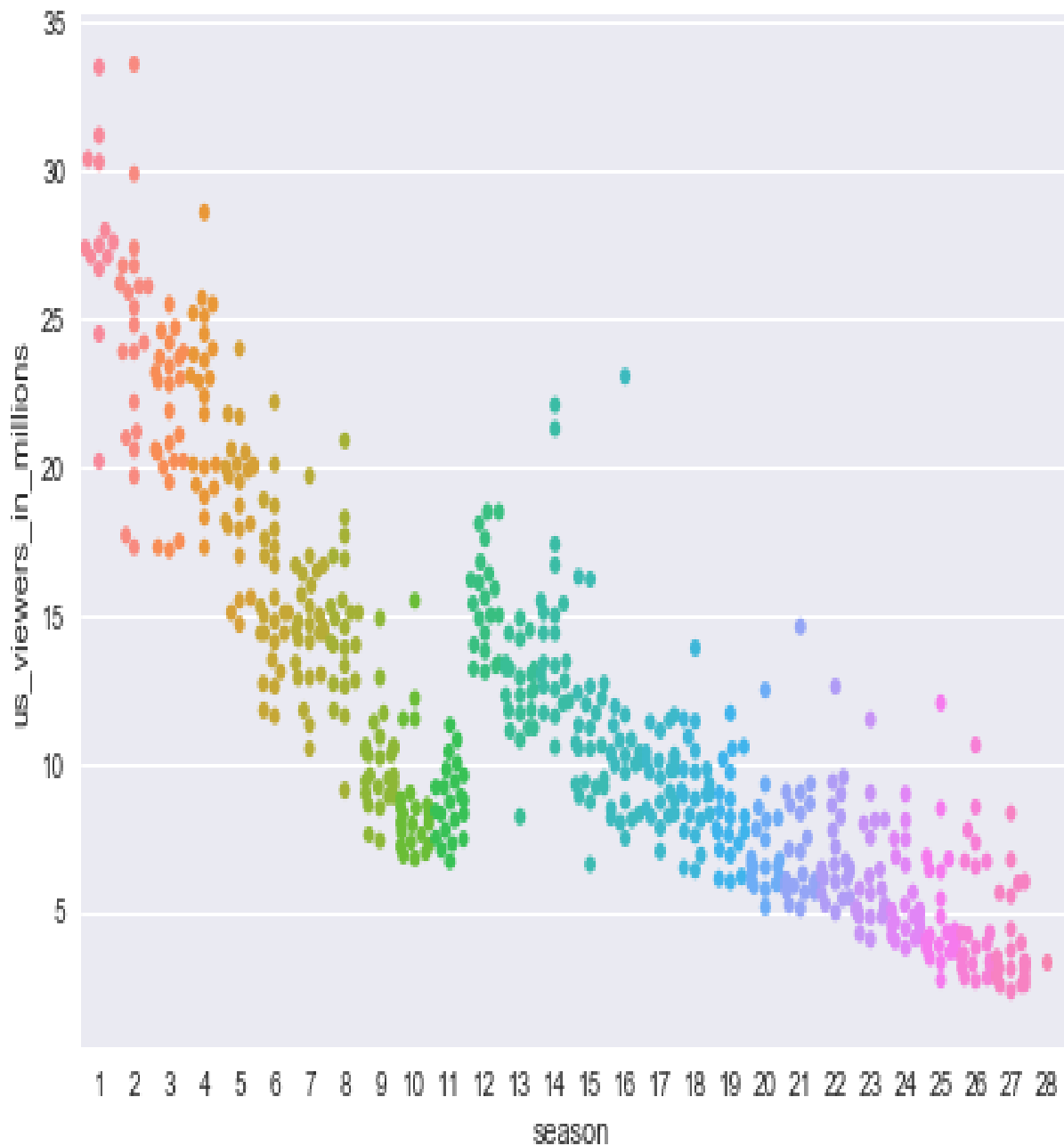
# Swarm plot

Let's now plot `swarmplot`. We use seaborn's `swarmplot` method for this, as follows:

```
|sns.swarmplot(x="season", y="us_viewers_in_millions", data=df);
```

The output is as follows:





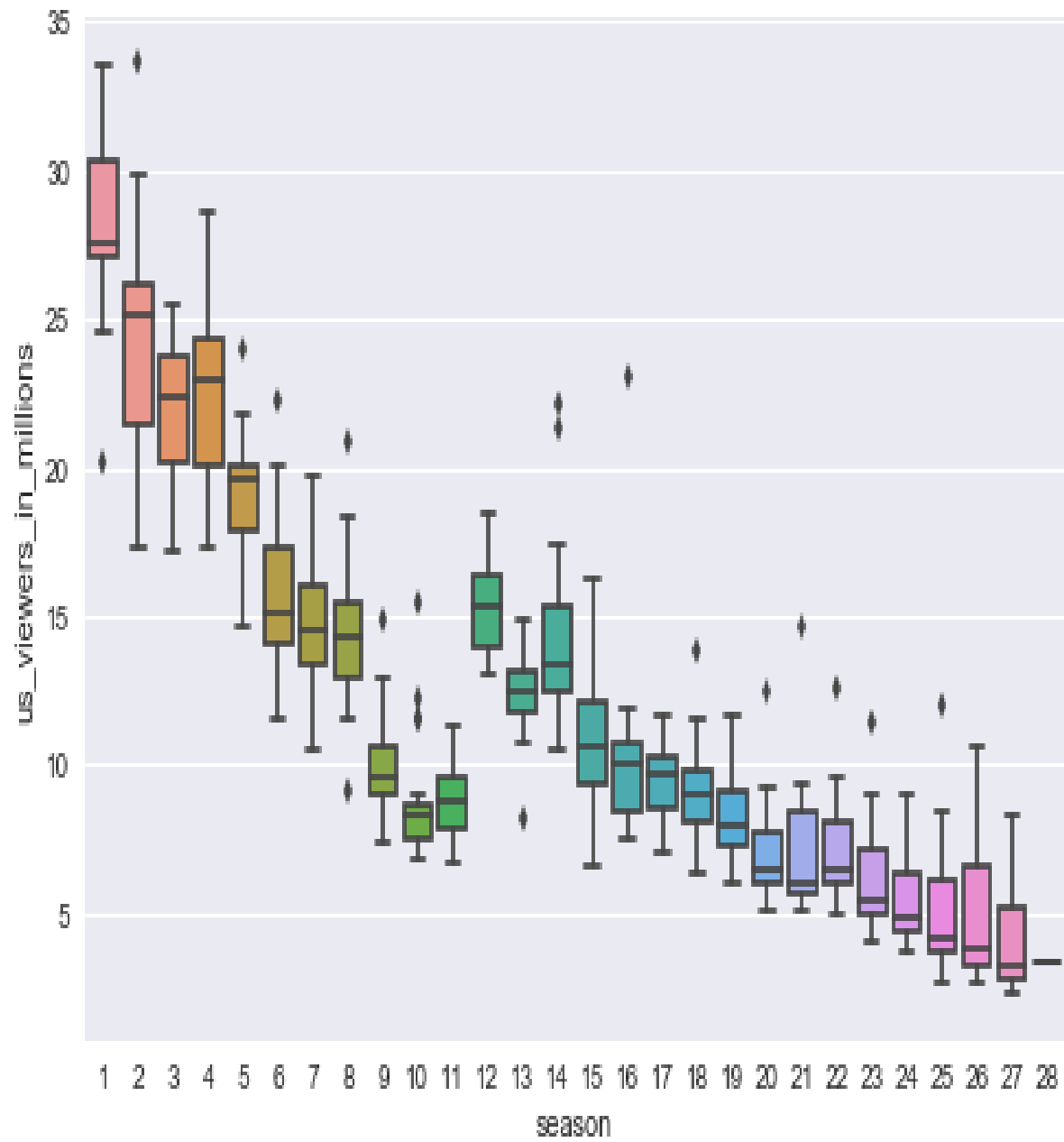
Here, we have also passed the season number on the x axis, and use viewers in millions on the y axis.

# Box plot

Now let's take the same data and create a box plot using the `boxplot` method, as follows:

```
|sns.boxplot(x="season", y="us_viewers_in_millions", data=df);
```

The output of the preceding command is as follows:

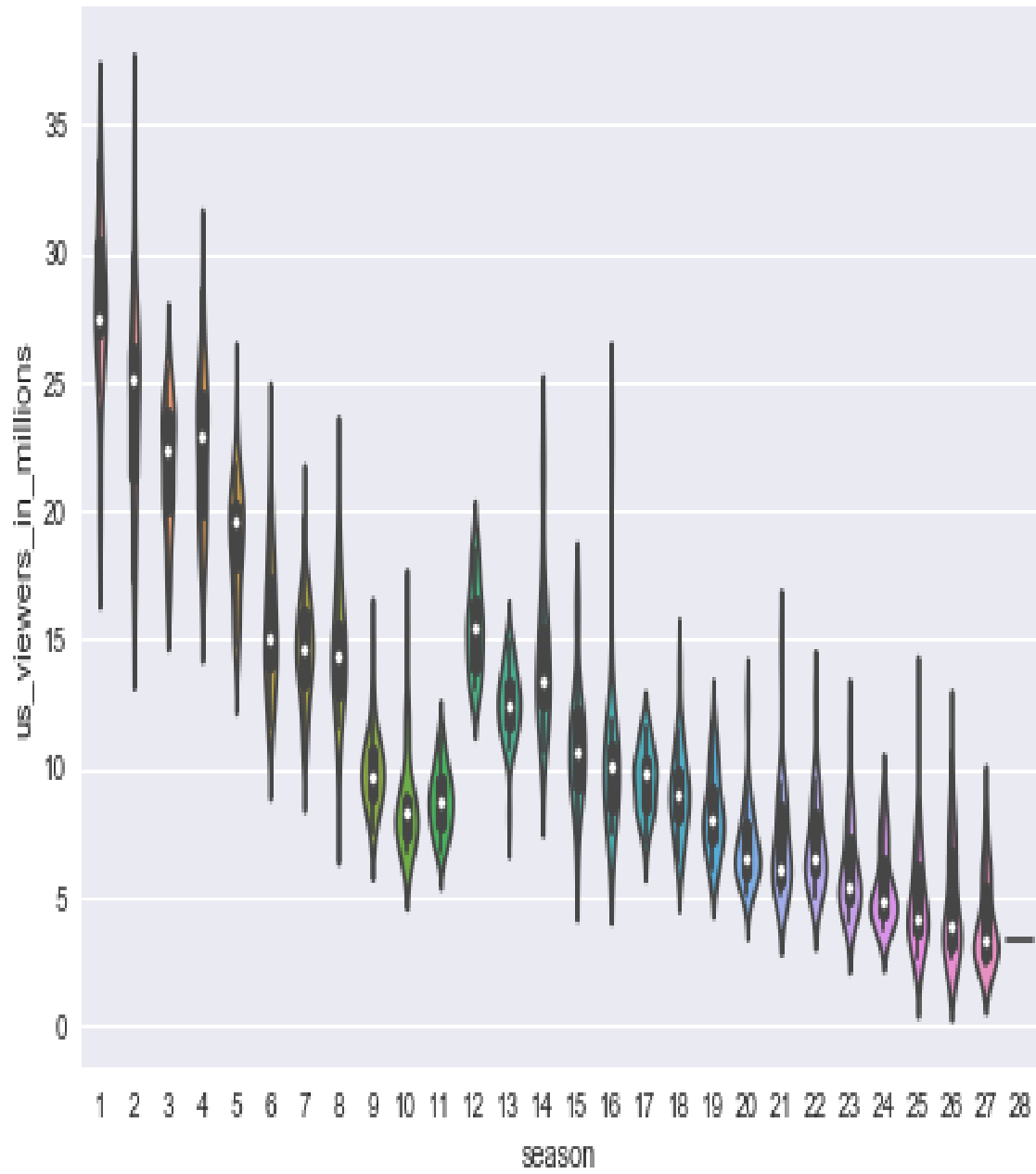


# Violin plot

A violin plot is created using the `violinplot()` method, as follows:

```
| sns.violinplot(x="season", y="us_viewers_in_millions", data=df);
```

The output of the preceding code is as follows:

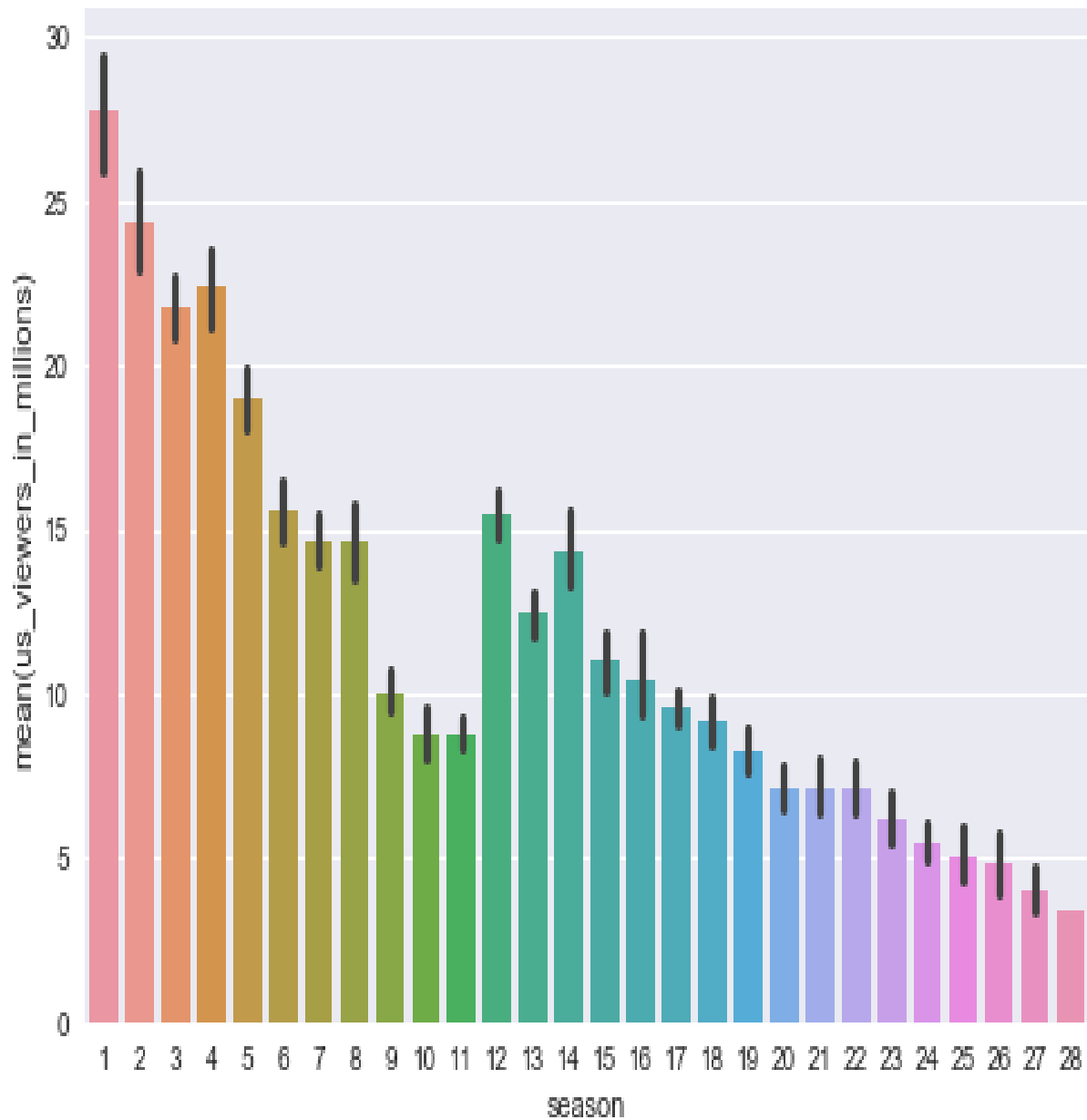


# Bar plot

To draw a bar plot, we use the following `barplot` method:

```
|sns.barplot(x="season", y="us_viewers_in_millions", data=df);
```

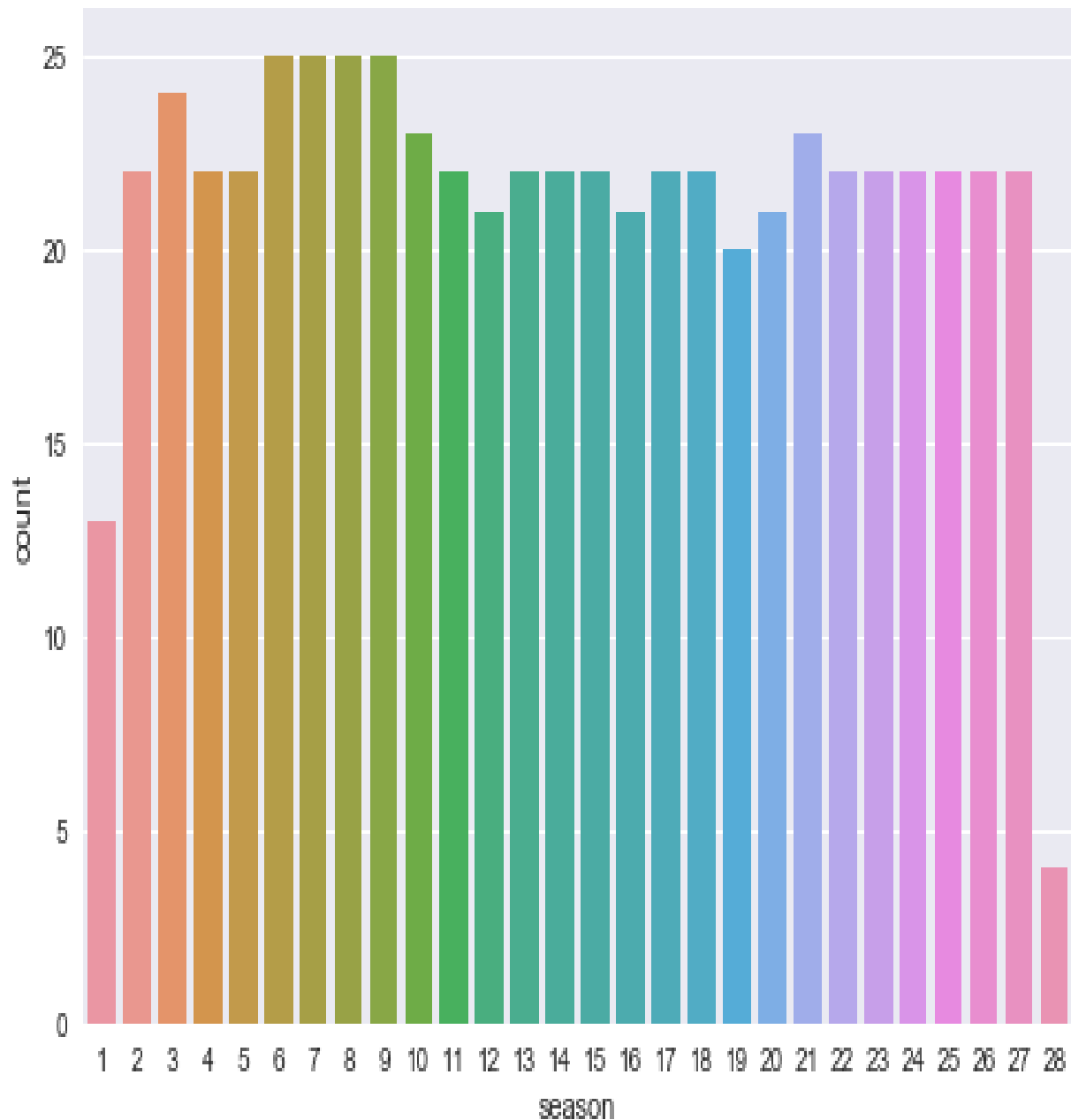
The output is as follows:



Note that there is another version of bar plots available, and these are drawn using the `countplot` method, as follows:

```
|sns.countplot(x="season", data=df);
```

The output of the preceding code is as follows:



This style of plot can be used when you want to show the number of observations in each category, rather than compute a status for the second variable.



# Wide-form plot

Seaborn also supports wide-form data plots. Let's read in the following dataset to demonstrate one:

```
| df = pd.read_csv('data-alcohol.csv')  
| df.head()
```

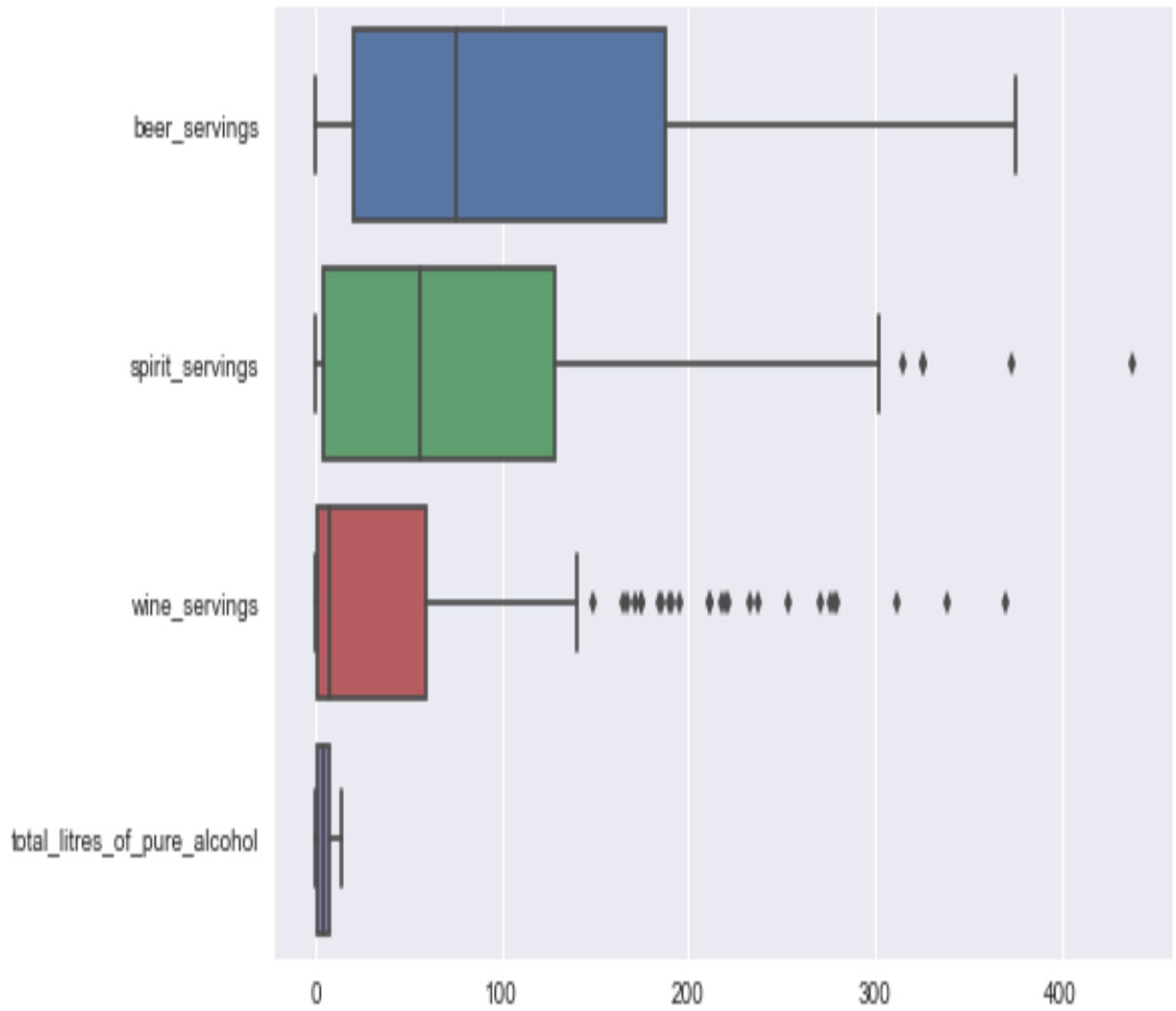
The output is as follows:

	country	beer_servings	spirit_servings	wine_servings	total_litres_of_pure_alcohol
0	Afghanistan	0	0	0	0.0
1	Albania	89	132	54	4.9
2	Algeria	25	0	14	0.7
3	Andorra	245	138	312	12.4
4	Angola	217	57	45	5.9

We can create a wide-form box plot with the following command:

```
| sns.boxplot(data=df, orient="h");
```

The output of the preceding code is as follows:



Here, we have created a wide-form box plot by passing in our dataset and passing the orient as `h`.

# Plotting with Data-Aware Grids

In this section, we'll learn about drawing multiple instances of the same plot on different subsets of our dataset. We'll learn about grid plotting with seaborn's `FacetGrid` method. We'll also explore seaborn's `PairGrid` and `PairPlot` methods for grid plotting.

Let's start by importing our Python modules in the Jupyter Notebook, in the following code:

```
import pandas as pd
from matplotlib import pyplot as plt
%matplotlib inline
import seaborn as sns
```

We now need to read in our first CSV dataset using the following code:

```
df = pd.read_csv('data-titanic.csv')
df.head()
```

The output from the preceding command is as follows:

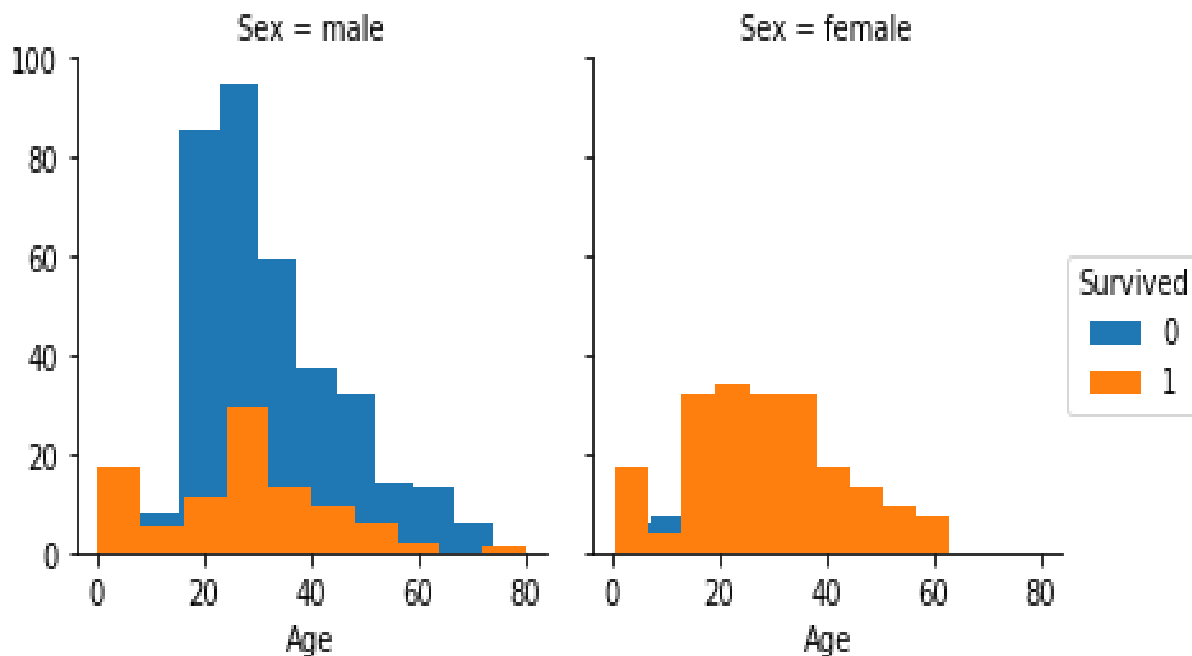
PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

# Plotting with the FacetGrid() method

Let's start by looking at how to plot multi-dimensional plots with the `FacetGrid` method, as shown in the following code:

```
g = sns.FacetGrid(df, col="Sex", hue='Survived')
g.map(plt.hist, "Age");
g.add_legend();
```

The output of the preceding code should look like the following screenshot:



Here, we have used the `FacetGrid` method to draw two side-by-side histograms for male and female passengers. This side-by-side display helps us to compare the survival rates of male and female passengers by their age. To draw this, we first created a grid using the `FacetGrid` method. We then passed in our dataset's DataFrame columns as `Sex` and `hue` as `Survived`. **Hue** stands for the depth of the plot. This then created the grid with two separate plots for male and female passengers. We then called the `map` method on the grid and

passed the `plt.hist` and `Age` parameters, which drew our two histograms. Finally, we added the legend with the `add_legend` method.

# Plotting with the PairGrid() method

Now let's look at how we can use the `PairGrid` method to draw grid-aware plots. We are using the MLB players dataset for this, as shown in the following code:

```
mlb = pd.read_csv('data-mlb-players.csv')
mlb.head()
```

The output is as follows:

	Position	Height	Weight	Age
0	Catcher	74	180.0	22.99
1	Catcher	74	215.0	34.69
2	Catcher	72	210.0	30.78
3	First_Baseman	72	210.0	35.43
4	First_Baseman	73	188.0	35.71

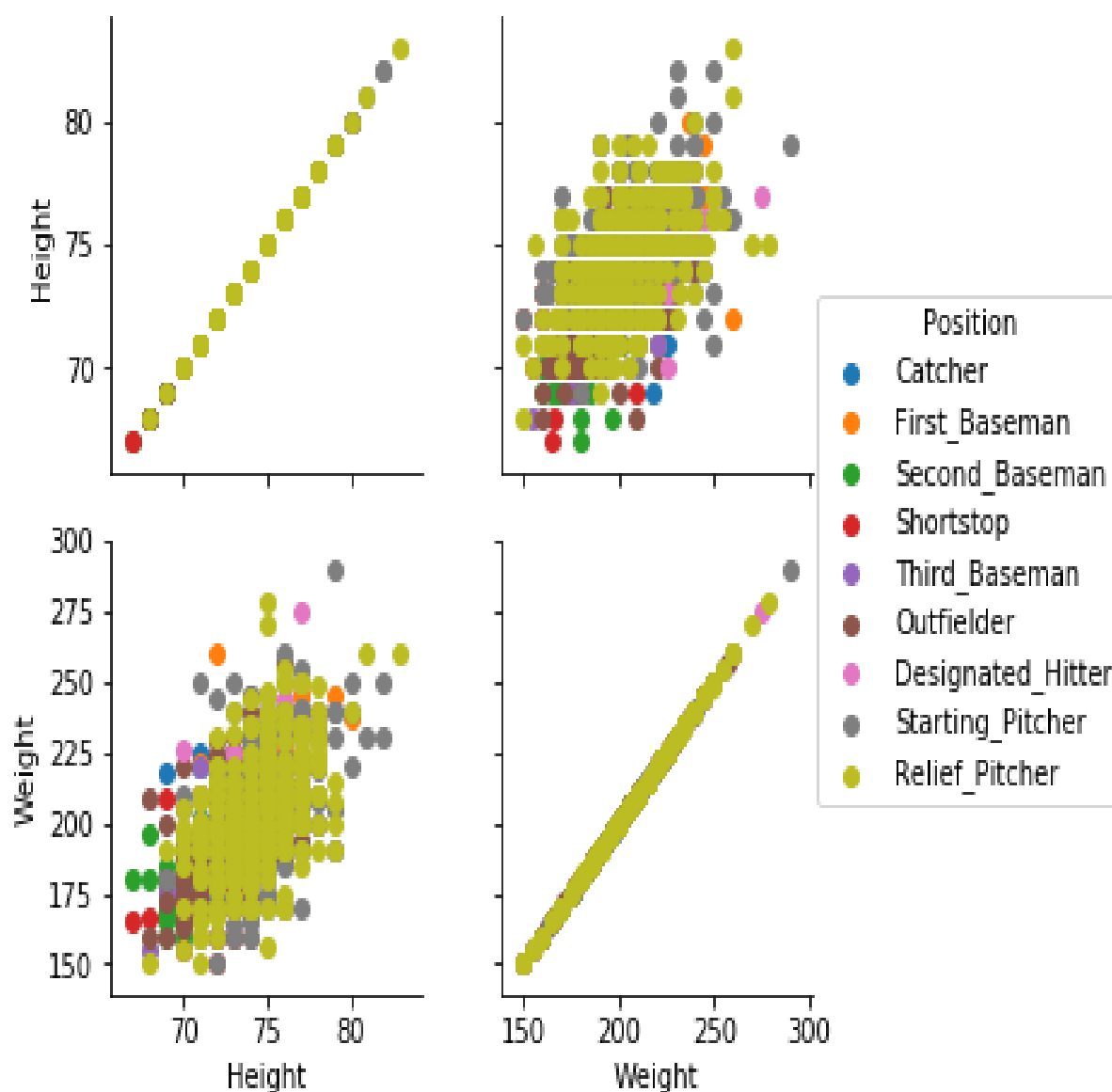
(Source: [http://wiki.stat.ucla.edu/socr/index.php/SOCR\\_Data\\_MLB\\_HeightsWeights](http://wiki.stat.ucla.edu/socr/index.php/SOCR_Data_MLB_HeightsWeights))

Let's create a plot with the following code:

```
g = sns.PairGrid(mlb, vars=["Height", "Weight"], hue="Position")
g.map(plt.scatter);
g.add_legend();
```

Here, we have passed our dataset of MLB players, and we set `vars` to a list containing players' `Height` and `Weight`. We then set `hue` as `Position`. We then called `map` with the `scatterplot` method on this grid. Finally, we added the

legend, which gives us a 2-by-2 grid with all the combinations of height and weight plots, as shown in the following screenshot:



The depth for these plots is provided by the player positions column.

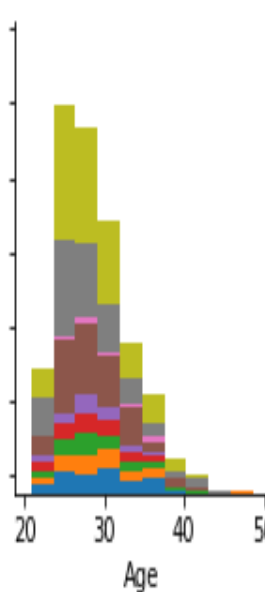
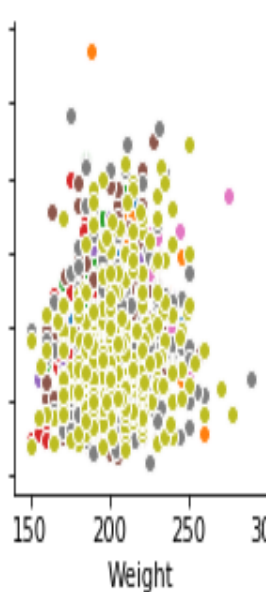
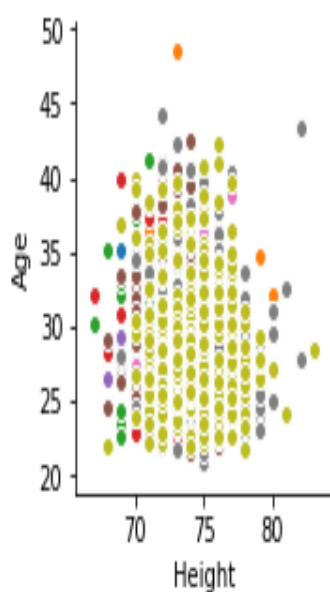
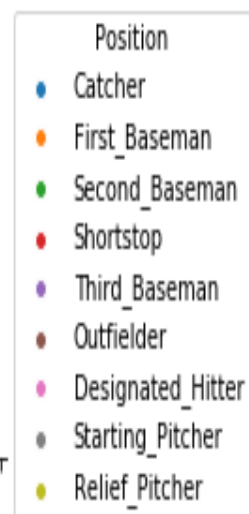
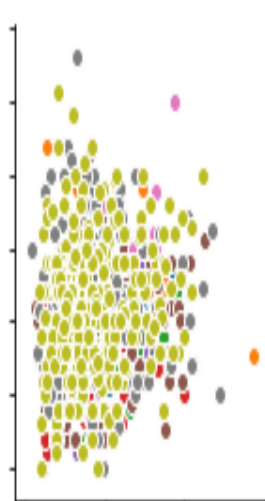
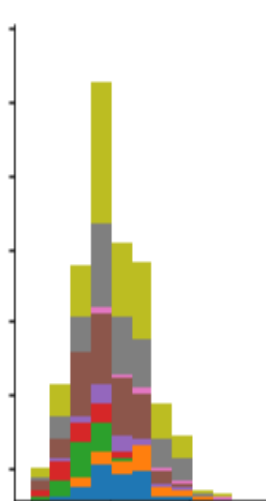
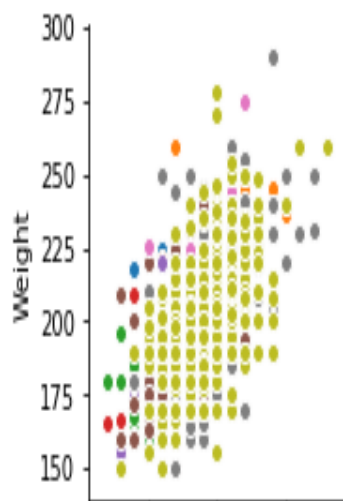
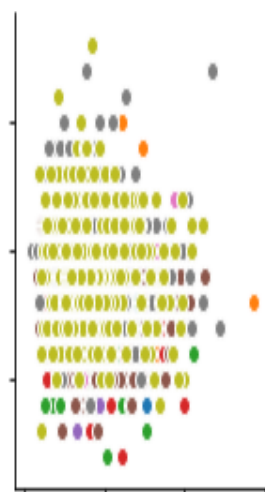
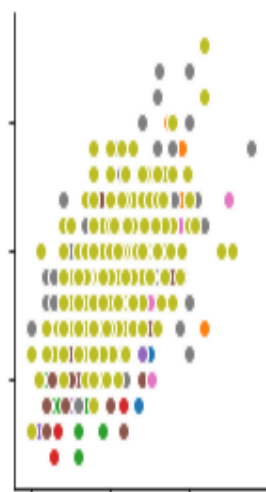
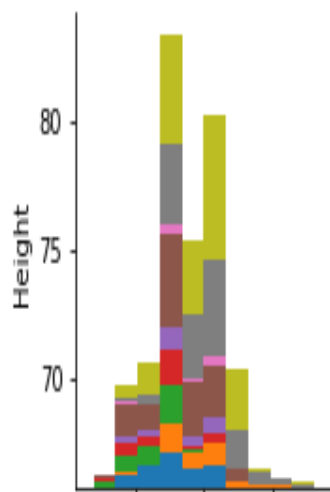
# Plotting with the PairPlot() method

`PairPlot` is directly called by passing the dataset, as follows. The depth is made up of the `hue` and `size` parameters:

```
|sns.pairplot(mlb, hue="Position", size=2.5);
```

The preceding command gives us a multi-plot in a 3-by-3 grid. This is because we have three observations or columns for each position, as shown in the following screenshot:





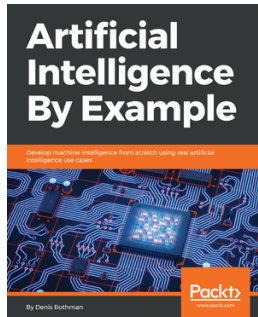
The observations present are Height, Weight, and Age.

# Summary

In this chapter, we learned about the advanced techniques of data visualization using seaborn's data visualization library. We learned how to get started with seaborn, and then explored some of its features, including how to control plot aesthetics, choosing colors for plots, and so on. We learned how to draw a few different kinds of plots, as well as how to plot categorical data with seaborn. Finally, we learned how to create plots with Data-Aware Grids.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

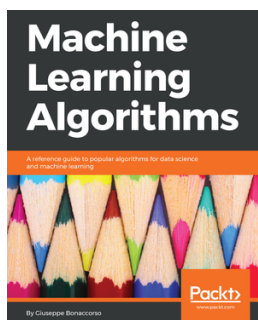


## **Artificial Intelligence By Example**

Denis Rothman

ISBN: 9781788990547

- Use adaptive thinking to solve real-life AI case studies
- Rise beyond being a modern-day factory code worker
- Acquire advanced AI, machine learning, and deep learning designing skills
- Learn about cognitive NLP chatbots, quantum computing, and IoT and blockchain technology
- Understand future AI solutions and adapt quickly to them
- Develop out-of-the-box thinking to face any challenge the market presents



# **Machine Learning Algorithms**

Giuseppe Bonaccorso

ISBN: 9781785889622

- Acquaint yourself with important elements of Machine Learning
- Understand the feature selection and feature engineering process
- Assess performance and error trade-offs for Linear Regression
- Build a data model and understand how it works by using different types of algorithm
- Learn to tune the parameters of Support Vector machines
- Implement clusters to a dataset
- Explore the concept of Natural Processing Language and Recommendation Systems
- Create a ML architecture from scratch.

# **Leave a review - let other readers know what you think**

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!