Data Visualization with Python

Create an impact with meaningful data insights using interactive and engaging visuals

Mario Döbler and Tim Großmann

Packt
www.packt.com

Data Visualization with Python

Author: Mario Döbler and Tim Großmann

Reviewer: José España

Managing Editor: Aditya Shah and Bhavesh Bangera

Acquisitions Editor: Kunal Sawant

Production Editor: Shantanu Zagade

Editorial Board: David Barnes, Ewan Buckingham, Shivangi Chatterji, Simon Cox, Manasa Kumar, Alex Mazonowicz, Douglas Paterson, Dominic Pereira, Shiny Poojary, Saman Siddiqui, Erol Staveley, Ankita Thakur and Mohita Vyas

First Published: February 2019

Production Reference: 2100419

ISBN: 978-1-78995-646-7

Table of Contents

Chapter 2: All You Need to Know About Plots

Chapter 3: A Deep Dive into Matplotlib

Chapter 5: Plotting Geospatial Data

Preface

About

This section briefly introduces the author, the coverage of this book, the technical skills you'll need to get started, and the hardware and software requirements required to complete all of the included activities and exercises.

About the Book

You'll begin Data Visualization with Python with an introduction to data visualization and its importance. Then, you'll learn about statistics by computing mean, median, and variance for some numbers, and observing the difference in their values. You'll also learn about key NumPy and Pandas techniques, such as indexing, slicing, iterating, filtering, and grouping. Next, you'll study different types of visualizations, compare them, and find out how to select a particular type of visualization using this comparison. You'll explore different plots, including custom creations.

After you get a hang of the various visualization libraries, you'll learn to work with Matplotlib and Seaborn to simplify the process of creating visualizations. You'll also be introduced to advanced visualization techniques, such as geoplots and interactive plots. You'll learn how to make sense of geospatial data, create interactive visualizations that can be integrated into any webpage, and take any dataset to build beautiful and insightful visualizations. You'll study how to plot geospatial data on a map using Choropleth plot, and study the basics of Bokeh, extending plots by adding widgets and animating the display of information.

This book ends with an interesting activity in which you will be given a new dataset and you must apply all that you've learned to create an insightful capstone visualization.

About the Author

Mario Döbler is a Ph.D. student with focus in deep learning at the University of Stuttgart. He previously interned at the Bosch Center for Artificial Intelligence in Silicon Valley in the field of deep learning, using state-of-the-art algorithms to develop cutting-edge products. In his master thesis, he dedicated himself to apply deep learning to medical data to drive medical applications.

Tim Großmann is a CS student with an interest in diverse topics, ranging from AI to IoT. He previously worked at the Bosch Center for Artificial Intelligence in Silicon Valley, in the field of big data engineering. He's highly involved in different open source projects and actively speaks at meetups and conferences about his projects and experiences.

Objectives

- Get an overview of various plots and their best use cases
- Work with different plotting libraries and get to know their strengths and weaknesses
- Learn how to create insightful visualizations
- Understand what makes a good visualization
- Improve your Python data wrangling skills
- Work with real-world data
- Learn industry standard tools
- Develop your general understanding of data formats and representations

Audience

This book is aimed at developers or scientists who want to get into data science or want to use data visualizations to enrich their personal and professional projects. Prior experience in data analytics and visualization is not needed; however, some knowledge of Python and high-school level math is recommended. Even though this is a beginner-level book on data visualization, more experienced students will benefit from improving their Python skills by working with real-world data.

Approach

This book thoroughly explains the technology in easy-to-understand language while perfectly balancing theory and exercises. Each chapter is designed to build on the learning from the previous chapter. The book also contains multiple activities that use real-life business scenarios for you to practice and apply your new skills in a highly relevant context.

Minimum Hardware Requirements

For the optimal student experience, we recommend the following hardware configuration:

- OS: Windows 7 SP1 32/64-bit, Windows 8.1 32/64-bit or Windows 10 32/64-bit, Ubuntu 14.04 or later, or macOS Sierra or later
- Processor: Dual Core or better
- Memory: 4GB RAM
- Storage: 10 GB available space

Software Requirements

You'll also need the following software installed in advance:

- Browser: Google Chrome or Mozilla Firefox
- Conda
- JupyterLab and Jupyter Notebook
- Sublime Text (latest version), Atom IDE (latest version), or other similar text editor applications
- Python 3
- The following Python libraries installed: NumPy, pandas, Matplotlib, seaborn, geoplotlib, Bokeh, and squarify

Conventions

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are

shown as follows: "axis=0 is horizontal and **axis=1** is vertical, so if we want to have the result for each row, we need to choose **axis=1**".

A block of code is set as follows:

```
# indexing the first value of the second row (1st row, 1st value)

first_val_first_row = dataset[0][0]

np.mean(first_val_first_row)
```

New terms and important words are shown in bold:

"To draw conclusions from visualized data, we need to handle our data and transform it into the best possible representation. This is where **data wrangling** is used."

Installation and Setup

Before you start this book, we'll install Python 3.6, pip, and the other libraries used throughout this book. You will find the steps to install them here.

Installing Python

Install Python 3.6 following the instructions in this link: https://realpython.com/installing-python/.

Installing pip

1. To install pip, go to the following link and download the **get-pip.py** file: https://pip.pypa.io/en/stable/installing/.
2. Then, use the following command to install it:

   ```
   python get-pip.py
   ```

   You might need to use the **python3 get-pip.py** command, due to previous versions of Python on your computer that already use the

python command.

Installing libraries

Using the pip command, install the following libraries:

python -m pip install --user numpy matplotlib jupyterlab pandas squarify bokeh geoplotlib seaborn

Working with JupyterLab and Jupyter Notebook

You'll be working on different exercises and activities in JupyterLab. These exercises and activities can be downloaded from the associated GitHub repository.

Download the repository from here: https://github.com/TrainingByPackt/Data-Visualization-with-Python.

You can either download it using GitHub or as a zipped folder by clicking on the green **Clone or download** button on the upper-right side.

In order to open Jupyter Notebooks, you have to traverse into the directory with your terminal. To do that, type:

cd Data-Visualization-with-Python/<your current chapter>.

For example:

cd Data-Visualization-with-Python/chapter01/

To complete the process, perform the following steps:

1. To reach each activity and exercise, you have to use **cd** once more to go into each folder, like so:

   cd Activity01

2. Once you are in the folder of your choice, simply call **jupyter-lab** to start up JupyterLab. Similarly, for Jupyter Notebook, call **jupyter**

notebook.

Importing Python Libraries

Every exercise and activity in this book will make use of various libraries. Importing libraries into Python is very simple and here's how we do it:

1. To import libraries, such as NumPy and pandas, we have to run the following code. This will import the whole **numpy** library into our current file:

   ```
   import numpy # import numpy
   ```

2. In the first cells of the exercises and activities of this bookware, you will see the following code. We can use **np** instead of **numpy** in our code to call methods from **numpy**:

   ```
   import numpy as np # import numpy and assign alias np
   ```

3. In later chapters, partial imports will be present, as shown in the following code. This only loads the **mean** method from the library:

   ```
   from numpy import mean # only import the mean method of numpy
   ```

Installing the Code Bundle

Copy the code bundle for the class to the **C:/Code** folder.

Additional Resources

The code bundle for this book is also hosted on GitHub at: https://github.com/TrainingByPackt/Data-Visualization-with-Python.

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

Chapter 1

The Importance of Data Visualization and Data Exploration

Learning Objectives

By the end of this chapter, you will be able to:

- Explain why data visualization is important
- Calculate basic statistical values such as median, mean, and variance
- Use NumPy for data wrangling
- Use pandas for data wrangling

In this chapter, you will also learn about the basic operations of NumPy and pandas.

Introduction

Unlike machines, people are not usually equipped for interpreting a lot of information from a random set of numbers and messages in a given piece of data. While they may know what the data is basically comprised of, they might need help to understand it completely. Out of all of our logical capabilities, we understand things best through the processing of visual information. When data is represented visually, the probability of understanding complex builds and numbers increases.

Python has recently emerged as a programming language that performs well for data analysis. Python has applications across data science pipelines that convert data into a usable format, analyze it, and extract useful conclusions from the data to represent it well. It provides data visualization libraries that can help you assemble graphical representations quickly.

In this book, you will learn how to use Python in combination with various libraries, such as **NumPy**, **pandas**, **Matplotlib**, **seaborn**, and **geoplotlib**, to create impactful data visualizations using real-world data. Besides that, you will also learn about the features of different types of charts and compare their advantages and disadvantages. This will help you choose the chart type that's suited to visualizing your data.

Once we understand the basics, we can cover more advanced concepts, such as interactive visualizations and how **Bokeh** can be used to create animated visualizations that tell a story. Upon completion of this book, you will be able to perform **data wrangling**, extract important information, and visualize your insights descriptively.

Introduction to Data Visualization

Computers and smartphones store data such as names and numbers in a digital format. **Data representation** refers to the form in which you can store, process, and transmit data.

Representations can narrate a story and convey key discoveries to your audience. Without appropriately modeling your information to use it to make meaningful findings, its value is reduced. Creating representations helps to achieve a clearer, more concise, and direct perspective of information, making it easier for anyone to understand the data.

Information isn't really equivalent to data. Representations are a useful apparatus to find insights hidden in data. Thus, representations transform information into useful data.

The Importance of Data Visualization

Visual data is very easy to understand compared to data in any other form. Instead of just looking at data in the columns of an Excel spreadsheet, we get a better idea of what our data contains by using a visualization. For instance, it's easy to see a pattern emerge from the numerical data that's given in the following graph:


Figure 1.1: A simple example of data visualization

Figure 1.1: A simple example of data visualization

Visualizing data has many advantages, such as the following:

- Complex data can be easily understood
- A simple visual representation of outliers, target audiences, and future markets can be created
- Storytelling can be done using dashboards and animations
- Data can be explored through interactive visualizations

Data Wrangling

To draw conclusions from visualized data, we need to handle our data and transform it into the best possible representation. This is where **data wrangling** is used. It is the discipline of augmenting, transforming, and enriching data in a way that allows it to be displayed and understood by machine learning algorithms.

Look at the following data wrangling process flow diagram to understand how accurate and actionable data can be obtained for business analysts to work on. As you can see, the Employee Engagement data is in its raw form initially. It gets imported as a DataFrame and is later cleaned. The cleaned data is then transformed into corresponding graphs, from which insights can be modeled. Based on these insights, we can communicate the final results. For example, employee engagement can be measured based on raw data gathered from feedback surveys, employee tenure, exit interviews, one-on-one meetings, and so on. This data is cleaned and made into graphs based on parameters such as referrals, faith in leadership, and scope of promotions. The percentages, that is, insights generated from the graphs, help us reach our result, which is to determine the measure of employee engagement:

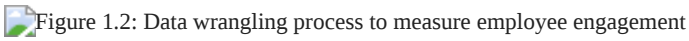
Figure 1.2: Data wrangling process to measure employee engagement

Figure 1.2: Data wrangling process to measure employee engagement

Tools and Libraries for Visualization

There are several approaches to creating data visualizations. Depending on your background, you might want to use a non-coding tool such as **Tableau**, which gives you the ability to get a good feel for your data. Besides Python, which will be used in this book, **MATLAB** and **R** are heavily used in data analytics.

However, Python is the most popular language in the industry. Its ease of use and the speed at which you can manipulate and visualize data, combined with the availability of a number of libraries, makes Python the best choice.

Note

MATLAB (https://www.mathworks.com/products/matlab.html), R (https://www.r-project.org), and Tableau (https://www.tableau.com) are not part of this book, so we will only cover the highlighted tools and libraries for Python.

Overview of Statistics

Statistics is a combination of the analysis, collection, interpretation, and representation of numerical data. **Probability** is a measure of the likelihood that an event will occur and is quantified as a number between 0 and 1.

A **probability distribution** is a function that provides the probabilities for every possible event. Probability distribution is frequently used for statistical analysis. The higher the probability, the more likely the event. There are two types of probability distribution, namely discrete probability distribution and continuous probability distribution.

A **discrete probability distribution** shows all the values that a random variable can take, together with their probability. The following diagram illustrates an example of a discrete probability distribution. If we have a 6-sided die, we can roll each number between 1 and 6. We have six events that can occur based on the number rolled. There is an equal probability of rolling any of the numbers, and the individual probability of any of the six events occurring is 1/6:

 Figure 1.3: Discrete probability distribution for die rolls

Figure 1.3: Discrete probability distribution for die rolls

A **continuous probability distribution** defines the probabilities of each possible value of a continuous random variable. The following diagram illustrates an example of a continuous probability distribution. This example illustrates the distribution of the time needed to drive home. In most cases, around 60 minutes is needed, but sometimes less time is needed because there is no traffic, and sometimes much more time is needed if there are traffic jams:

 Figure 1.4: Continuous probability distribution for the time taken to reach home

Figure 1.4: Continuous probability distribution for the time taken to reach home

Measures of Central Tendency

Measures of central tendency are often called **averages** and describe central or typical values for a probability distribution. There are three kinds of averages that we are going to discuss in this chapter:

- **Mean**: The arithmetic average that is computed by summing up all measurements and dividing the sum by the number of observations. The mean is calculated as follows:

- **Median**: This is the middle value of the ordered dataset. If there is an even number of observations, the median will be the average of the two middle values. The median is less prone to outliers compared to the mean, where outliers are distinct values in data.

- **Mode**: Our last measure of central tendency, the mode is defined as the most frequent value. There may be more than one mode in cases where multiple values are equally frequent.

Example:

A die was rolled ten times and we got the following numbers: 4, 5, 4, 3, 4, 2, 1, 1, 2, and 1.

The mean is calculated by summing up all events and dividing them by the number of observations: (4+5+4+3+4+2+1+1+2+1)/10=2.7.

To calculate the median, the die rolls have to be ordered according to their values. The ordered values are as follows: 1, 1, 1, 2, 2, 3, 4, 4, 4, 5. Since we have an even number of die rolls, we need to take the average of the two middle values. The average of the two middle values is (2+3)/2=2.5.

The modes are 1 and 4 since they are the two most frequent events.

Measures of Dispersion

Dispersion, also called **variability**, is the extent to which a probability distribution is stretched or squeezed.

The different measures of dispersion are as follows:

- **Variance**: The variance is the expected value of the squared deviation from the mean. It describes how far a set of numbers is spread out from their mean. Variance is calculated as follows:

- **Standard deviation**: It's the square root of the variance.
- **Range**: This is the difference between the largest and smallest values in a dataset.
- **Interquartile range**: Also called the **midspread** or **middle 50%**, it is the difference between the 75th and 25th percentiles, or between the upper and lower quartiles.

Correlation

The measures we have discussed so far only considered single variables. In contrast, **correlation** describes the statistical relationship between two variables:

In positive correlation, both variables move in the same direction

In negative correlation, the variables move in opposite directions

In zero correlation, the variables are not related

Note

One thing you should be aware of is that correlation does not imply causation. Correlation describes the relationship between two or more variables, while causation describes how one event is caused by another. For example: sleeping with your shoes on is correlated with waking up with a headache. This does not mean that sleeping with your shoes on causes a headache in the morning. There might be a third, hidden variable, for example, someone was up working late the previous night, which caused both them falling asleep with their shoes on and waking up with a headache.

Example:

You want to find a decent apartment to rent that is not too expensive compared to other apartments you've found. The other apartments you found on a website are priced as follows: $700, $850, $1,500, and $750 per month:

- The mean is
- The median is
- The standard deviation is

- The range is
- The median is a better statistical measure in this case, since it is less prone to outliers (the rent amount of $1,500).

Types of Data

It is important to understand what kind of data you are dealing with so that you can select both the right statistical measure and the right visualization. We categorize data as categorical/qualitative and numerical/quantitative. Categorical data describes characteristics, for example, the color of an object or a person's gender. We can further divide categorical data into nominal and ordinal data. In contrast to nominal data, ordinal data has an order.

Numerical data can be divided into discrete and continuous data. We speak of discrete data if the data can only have certain values, whereas continuous data can take any value (sometimes limited to a range).

Another aspect to consider is whether the data has a temporal domain – in other words, is it bound to time or does it change over time? If the data is bound to a location, it might be interesting to show the spatial relationship, so you should keep that in mind as well:

 Figure 1.5: Classification of types of data

Figure 1.5: Classification of types of data

Summary Statistics

In real-world applications, we often encounter enormous datasets, therefore, **summary statistics** are used to summarize important aspects of data. They are necessary to communicate large amounts of information in a compact and simple way.

We have already covered measures of central tendency and dispersion, which are both summary statistics. It is important to know that measures of central tendency show a center point in a set of data values, whereas measures of dispersion show how spread out the data values are.

The following table gives an overview of which measure of central tendency is best suited to a particular type of data:

 Figure 1.6: Best suited measures of central tendency for different data types

Figure 1.6: Best-suited measures of central tendency for different types of data

NumPy

When handling data, we often need a way to work with multidimensional arrays. As we discussed previously, we also have to apply some basic mathematical and statistical operations on that data. This is exactly where **NumPy** positions itself. It provides support for large n-dimensional arrays and is the built-in support for many high-level mathematical and statistical operations.

Note

*Before NumPy, there was a library called **Numeric**. However, it's no longer used, as NumPy's signature **ndarray** allows for the performant handling of large and high-dimensional matrices.*

Those ndarrays are the essence of NumPy. They are what makes it faster than using Python's built-in lists. Other than the built-in list datatype, ndarrays provide a stridden view of memory (for example, `int[]` in Java). Since they are homogeneously typed,

meaning all the elements must be of the same type, the stride is consistent, which results in less memory wastage and better access times.

A **stride** is the number of locations between the beginnings of two adjacent elements in an array. They are normally measured in bytes or in units of the size of the array elements. A stride can be larger or equal to the size of the element, but not smaller, otherwise it would intersect the memory location of the next element.

Note

Remember that NumPy arrays have a "defined" datatype. This means you are not able to insert strings into an integer type array. NumPy is mostly used with double-precision datatypes.

Exercise 1: Loading a Sample Dataset and Calculating the Mean

Note

All exercises and activities will be developed in the Jupyter Notebook. Please download the GitHub repository with all the prepared templates from https://github.com/TrainingByPackt/Data-Visualization-with-Python

In this exercise, we will be loading the **`normal_distribution.csv`** dataset and calculating the mean of each row and each column in it:

1. Open the **`exercise01.ipynb`** Jupyter Notebook from the **`Lesson01`** folder to implement this exercise.

   To do that, you need to navigate to the path of this file. In the command-line terminal, type **`jupyter-lab.`**

2. You will now see a browser window open, showing the directory content you called the previous command in. Click on **`exercise01.ipynb`**. This will open the notebook.

3. The notebook for Chapter01 should now be open and ready for you to modify.After a short introduction, you should see a cell that imports the necessary dependencies. In this case, we will import **`numpy`** with an alias:

   ```
   # importing the necessary dependencies
   import numpy as np
   ```

4. Look for the cell that has a comment saying, "loading the dataset." This is the place you want to insert the **`genfromtxt`** method call. This method helps in loading the data from a given text or **`.csv`** file.

   The complete line should look like this:

   ```
   # loading the dataset
   dataset = np.genfromtxt('./data/normal_distribution.csv', delimiter=',')
   ```

5. If everything works as expected, the generation should run through without any error or output. Have a look at the data you just imported by simply writing the name of the ndarray in the next cell. This is implemented in the following code. Simply executing a cell that returns a value such as an ndarray GI will use Jupyter formatting, which looks nicer and, in most cases, displays more information than using **`print`**:

   ```
   # looking at the dataset
   dataset
   ```

The output of the preceding code is as follows:

 Figure 1.7: The first rows of the normal_distribution.csv file

Figure 1.7: The first rows of the normal_distribution.csv file

6. To get a quick overview of our dataset, we want to print out the "shape" of it.

   This will give us an output of the form (rows, columns). Print out the shape using the **`dataset.shape`** command. We can also call the rows as instances and the columns as features. This means that our dataset has 24 instances and 8 features:

   ```
   # printing the shape of our dataset
   dataset.shape
   ```

   The output of the preceding code is as follows:

 Figure 1.8: Shape of the dataset

Figure 1.8: Shape of the dataset

7. Calculating the mean is the next step once we've loaded and checked our dataset.

   The first row in a **`numpy array`** can be accessed by simply indexing it with zero, like this: **`dataset[0]`**. As we mentioned before, NumPy has some built-in functions for calculations such as mean. Therefore, we can simply call **`np.mean()`** and pass in the dataset row to get the result. The printed output should look as follows:

   ```
   # calculating the mean for the first row
   np.mean(dataset[0])
   ```

   The output of the preceding code is: as follows:

 Figure 1.9: Mean of elements in the first row

Figure 1.9: Mean of elements in the first row

8. We can also do the same for the first column by using **`np.mean()`** in combination with the column indexing **`dataset[:, 0]`**:

   ```
   # calculating the mean for the first column
   np.mean(dataset[:, 0])
   ```

   The output of the preceding code is as follows:

 Figure 1.10: Mean of elements in the first column

Figure 1.10: Mean of elements in the first column

9. If we want to get the mean for every single row, aggregated in a list, we can make use of the **`axis`** tools of NumPy. By simply passing the **`axis`** parameter in the **`np.mean()`** call, we can define the dimension our data will be aggregated on.

   `axis=0` is horizontal and **`axis=1`** is vertical, so if we want to have the result for each row, we need to choose **`axis=1`**:

   ```
   # mean for each row
   np.mean(dataset, axis=1)
   ```

The output of the preceding code is as follows:

 Figure 1.11: Mean of elements for each row

Figure 1.11: Mean of elements for each row

If we want to have the result of each column, we need to choose **`axis=0`**.

```
# mean for each column
np.mean(dataset, axis=0)
```

The output of the preceding code is as follows:

 Figure 1.12: Mean of elements for each column

Figure 1.12: Mean of elements for each column

10. As the last task of this exercise, we also want to have the mean of the whole matrix. We could sum up all the values we retrieved in the previous steps, but NumPy allows us to simply pass in the whole dataset to do this calculation:

    ```
    # calculating the mean for the whole matrix
    np.mean(dataset)
    ```

    The output of the preceding code is as follows:

     Figure 1.13: Mean of elements for the complete dataset

    Figure 1.13: Mean of elements for the complete dataset

Congratulations! You are already one step closer to using NumPy in combination with plotting libraries and creating impactful visualizations. Since we've now covered the very basics and went through calculating the mean in the previous exercise, it's now up to you to solve the upcoming activity.

Activity 1: Using NumPy to Compute the Mean, Median, Variance, and Standard Deviation for the Given Numbers

In this activity, we will use the skills we've learned to import datasets and perform some basic calculations (mean, median, variance, and standard deviation) to compute our tasks.

We want to consolidate our new skills and get acquainted with NumPy:

1. Open the **`activity01.ipynb`** Jupyter Notebook from the **`Lesson01`** folder to implement this activity.
2. Now, **`import numpy`** into your Jupyter Notebook and give it the alias **`np`**.
3. Load the **`normal_distribution.csv`** dataset by using the **`genfromtxt`** method of **`numpy`**.
4. Make sure everything works by having a look at the ndarray.
5. Given the dataset, go in and use the built-in **`numpy`** methods.
6. Using these methods, first calculate the mean of the third row, the last column, and the intersection of the first 3 rows and first 3 columns.

7. Then, calculate the median of the last row, the last 3 columns, and each row.
8. Calculate the variance of each column, the intersection of the last 2 rows, and the first 2 columns.
9. Calculate the standard deviation for the dataset.

Note:

The solution for this activity can be found on page 244.

Congratulations! You've completed your first activity using NumPy. In the following activities, this knowledge will be further consolidated.

Basic NumPy Operations

In this section, we will learn basic NumPy operations such as indexing, slicing, splitting, and iterating and implement them in an activity.

Indexing

Indexing elements in a NumPy **`array`**, on a high level, works the same as with built-in Python Lists. Therefore, we are able to index elements in multi-dimensional matrices:

```
dataset[0] # index single element in outermost dimension
dataset[-1] # index in reversed order in outermost dimension
dataset[1, 1] # index single element in two-dimensional data
dataset[-1, -1] # index in reversed order in two-dimensional data
```

Slicing

Slicing has also been adapted from Python's Lists. Being able to easily slice parts of lists into new ndarrays is very helpful when handling large amounts of data:

```
dataset[1:3] # rows 1 and 2
dataset[:2, :2] # 2x2 subset of the data
dataset[-1, ::-1] # last row with elements reversed
dataset[-5:-1, :6:2] # last 4 rows, every other element up to index 6
```

Splitting

Splitting data can be helpful in many situations, from plotting only half of your time-series data to separating test and training data for machine learning algorithms.

There are two ways of splitting your data, horizontally and vertically. Horizontal splitting can be done with the **`hsplit`** method. Vertical splitting can be done with the **`vsplit`** method:

```
np.hsplit(dataset, (3)) # split horizontally in 3 equal lists
np.vsplit(dataset, (2)) # split vertically in 2 equal lists
```

Iterating

Iterating the NumPy data structures, ndarrays, is also possible. It steps over the whole list of data one after another, visiting every single element in the ndarray once. Considering that they can have several dimensions, indexing gets very complex.

`nditer` is a multi-dimensional iterator object that iterates over a given number of arrays:

```
# iterating over whole dataset (each value in each row)
for x in np.nditer(dataset):
print(x)
```

`ndenumerate` will give us exactly this index, thus returning (**`0`**, **`1`**) for the second value in the first row:

```
# iterating over whole dataset with indices matching the position in the dataset
for index, value in np.ndenumerate(dataset):
print(index, value)
```

Activity 2: Indexing, Slicing, Splitting, and Iterating

In this activity, we will use the features of NumPy to index, slice, split, and iterate ndarrays to consolidate what we've learned. Our client wants us to prove that our dataset is nicely distributed around the mean value of 100:

1. Open the **`activity02.ipynb`** Jupyter Notebook from the **`Lesson01`** folder to implement this activity.
2. Now, **`import numpy`** into your Jupyter Notebook and give it the alias **`np`**.
3. Load the **`normal_distribution.csv`** dataset using NumPy. Make sure that everything works by having a look at the ndarray, as in the previous activity. Follow the task description in the notebook.
4. With the dataset loaded, use the previously discussed indexing feature to index the second row of the dataset (second row), index the last element of the dataset (last row), index the first value of the second row (second row, first value), and index the last value of the second-to-last row (using combined access).
5. Creating sub-lists of the input requires slicing. Slice an intersection of four elements (2x2) of the first two rows and first two columns, select every second element of the fifth row, and reverse the entry order, selecting the first two rows in reverse order.
6. If we need a smaller dataset, we can also use splitting to effectively divide our dataset. Use this concept to split our dataset horizontally into three equal parts and split our dataset vertically on index 2.
7. The last task in this activity will be iterating over the complete dataset. Use the previously discussed methods to iterate over the whole dataset, with indices matching the position in the dataset.

Note:

The solution for this activity can be found on page 248.

Congratulations! We've already covered most of the basic data wrangling methods for NumPy. In the next activity, we'll take a look at more advanced features that will give you the tools to get better insights into your data.

Advanced NumPy Operations

In this section, we will learn GI advanced NumPy operations such as filtering, sorting, combining, and reshaping and implement them in an Activity.

Filtering

Filtering is a very powerful tool that can be used to clean up your data if you want to avoid outlier values. It's also helpful to get some better insights into your data.

In addition to the **`dataset[dataset > 10]`** shorthand notation, we can use the built-in NumPy **`extract`** method, which does the same thing using a different notation, but gives us greater control with more complex examples.

If we only want to extract the indices of the values that match our given condition, we can use the built-in **`where`** method. For example, **`np.where(dataset > 5)`** will return a list of indices of the values from the initial dataset that are bigger than 5:

```
dataset[dataset > 10] # values bigger than 10
np.extract((dataset < 3), dataset) # alternative – values smaller than 3
dataset[(dataset > 5) & (dataset < 10)] # values bigger 5 and smaller 10
np.where(dataset > 5) # indices of values bigger than 5 (rows and cols)
```

Sorting

Sorting each row of a dataset can be really useful. Using NumPy, we are also able to sort on other dimensions, such as columns.

In addition, **`argsort`** gives us the possibility to get a list of indices, which would result in a sorted list:

```
np.sort(dataset) # values sorted on last axis
np.sort(dataset, axis=0) # values sorted on axis 0
np.argsort(dataset) # indices of values in sorted list
```

Combining

Stacking rows and columns onto an existing dataset can be helpful when you have two datasets of the same dimension saved to different files.

Given two datasets, we use **vstack** to "stack" **`dataset_1`** on top of **`dataset_2`**, which will give us a combined dataset with all the rows from **`dataset_1`**, followed by all the rows from **`dataset_2`**.

If we use **hstack**, we stack our datasets "next to each other," meaning that the elements from the first row of **`dataset_1`** will be followed by the elements of the first row of **`dataset_2`**. This will be applied to each row:

```
np.vstack([dataset_1, dataset_2]) # combine datasets vertically
np.hstack([dataset_1, dataset_2]) # combine datasets horizontally
np.stack([dataset_1, dataset_2], axis=0) # combine datasets on axis 0
```

Note

Combining with stacking can take some getting used to. Please look at the examples in the NumPy documentation for further information:https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.hstack.html.

Reshaping

Reshaping can be crucial for some algorithms. Depending on the nature of your data, it might help you to reduce dimensionality to make visualization easier:

```
dataset.reshape(-1, 2) # reshape dataset to two columns x rows
np.reshape(dataset, (1, -1)) # reshape dataset to one row x columns
```

Here, **`-1`** is an unknown dimension that NumPy identifies automatically. NumPy will first figure out the length of any given array and the remaining dimensions and will thus make sure that it satisfies the given mentioned standard.

Activity 3: Filtering, Sorting, Combining, and Reshaping

This last activity for NumPy provides some more complex tasks to consolidate our learning. It will also combine most of the previously learned methods as a recap. Perform the following steps:

1. Open the **`activity03.ipynb`** Jupyter Notebook from the **`Lesson01`** folder to implement this activity.
2. NumPy will be the only necessary dependency for this activity, so make sure to import it.
3. Again, load the **`normal_distribution.csv`** dataset using NumPy. Make sure that everything works by having a look at the ndarray.
4. After loading the dataset, use the previously discussed filtering features to filter for values that are greater than 105, filter for values that are between 90 to 95, and get indices of values that have a delta (difference) of less than 1 to 100.
5. Sorting data is an important feature when trying to display data on an ordinal datatype column. Use **sorting** to sort values for each row, sort values for each column, get indices of positions for each row, and get the 3 smallest values for each row (the remaining values are not sorted).
6. Using the split data from *Activity02* of this chapter, we can also use the previously discussed combining features to add the second half of the first column back together, add the second column to our combined dataset, and add the third column to our combined dataset.
7. Use the reshaping features to reshape the dataset in a one-dimensional list with all the values and reshape the dataset into a matrix with only two columns.

Note:

The solution for this activity can be found on page 252.

Next, we will be learning about pandas, which will bring several advantages when working with data that is more complex than simple multi-dimensional numerical data. pandas also supports different datatypes in datasets, meaning that we can have columns that hold strings and others that have numbers.

NumPy itself, as you've seen, has some really powerful tools. Some of them are even more powerful when combined with pandas.

pandas

The **pandas** Python library offers data structures and methods to manipulate different types of data, such as numerical and temporal. These operations are easy to use and highly optimized for performance.

Data formats such as **CSV**, **JSON**, and databases can be used for **DataFrame** creation. DataFrames are the internal representation of data and are very similar to tables, but are more powerful. Importing and reading both files and in-memory data is abstracted into a user-friendly interface. When it comes to handling missing data, pandas provides built-in solutions to clean up and augment your data, meaning it fills in missing values with reasonable values.

Integrated indexing and label-based slicing in combination with fancy indexing (what we already saw with NumPy) makes handling data simple. More complex techniques such as **reshaping**, **pivoting,** and **melting** data, together with the possibility of easily **joining** and **merging** data, provide powerful tooling to handle your data correctly.

If you're working with **time-series data**, operations such as **date range generation**, **frequency conversion**, and **moving window statistics** can provide an advanced interface for your wrangling.

Note

Installation instructions for pandas can be found here: https://pandas.pydata.org/.

Advantages of pandas over NumPy

The following are some of the advantages of pandas:

- **High level of abstraction**: pandas has a higher abstraction level than NumPy, which gives it a simpler interface for users to interact with. It abstracts away some of the more complex concepts and makes it easier to use and understand.
- **Less intuition**: Many methods, such as joining, selecting, and loading files, are usable without much intuition and without taking away much of the powerful nature of pandas.
- **Faster processing**: The internal representation of DataFrames allows faster processing for some operations. Of course, this always depends on the data and its structure.
- **Easy DataFrames design**: DataFrames are designed for operations with and on large datasets.

Disadvantages of pandas

The following are some of the disadvantages of pandas:

- **Less applicable**: Due to its higher abstraction, it's generally less applicable than NumPy. Especially when used outside of its scope, operations quickly get very complex and hard to do.
- **More disk space**: Due to the mentioned internal representation of DataFrames and the way pandas trades disk space for a more performant execution, the memory usage of complex operations can spike.
- **Performance problems**: Especially when doing heavy joins, which is not recommended, memory usage can get critical and might lead to performance problems.
- **Hidden complexity**: The comparatively simple interface has its downsides too. Less experienced users often tend to overuse methods and execute them several times instead of reusing what they've already calculated. This hidden complexity makes users think that the operations themselves are simple, which is not the case.

  ### *Note*

  Always try to think about how to design your workflows instead of excessively using operations.

Exercise 2: Loading a Sample Dataset and Calculating the Mean

In this exercise, we will be loading the **`world_population.csv`** dataset and calculating the mean of some rows and columns. Our dataset holds the yearly population density for every country. Let's use pandas to get some really quick and easy insights:

1. Open the **`exercise02.ipynb`** Jupyter Notebook from the **`Lesson01`** folder to implement this exercise.

2. Import the pandas libraries:

   ```
   # importing the necessary dependencies
   import pandas as pd
   ```

3. After importing pandas, we can use the **`read_csv`** method to load the mentioned dataset. We want to use the first column, containing the country names, as our index. We will use the **`index_col`** parameter for that. The complete line should look like this:

   ```
   # loading the dataset
   dataset = pd.read_csv('./data/world_population.csv', index_col=0)
   ```

4. As before, have a look at the data you just imported by simply writing the name of the dataset in the next cell. pandas uses a data structure called DataFrames. We want to only print some of the rows to avoid filling the screen. pandas DataFrames come with two methods, **`head()`** and **`tail()`**, to do this for you. Both take number, **`n`**, as a parameter, which describes how many rows should be returned:

   ### *Note*

   Simply executing a cell that returns a value such as a DataFrame will use Jupyter formatting, which looks nicer and, in most cases, displays more information than using ***`print`***.

   ```
   # looking at the dataset
   dataset.head()
   ```

   The output of the preceding code is as follows:

Country Name	Country Code	Indicator Name	Indicator Code	1960	1961	1962	1963	1964	1965
Aruba	ABW	Population density (people per sq. km of land ...	EN.POP.DNST	NaN	307.972222	312.366667	314.983333	316.827778	318.666667
Andorra	AND	Population density (people per sq. km of land ...	EN.POP.DNST	NaN	30.587234	32.714894	34.914894	37.170213	39.470213
Afghanistan	AFG	Population density (people per sq. km of land ...	EN.POP.DNST	NaN	14.038148	14.312061	14.599692	14.901579	15.218206
Angola	AGO	Population density (people per sq. km of land ...	EN.POP.DNST	NaN	4.305195	4.384299	4.464433	4.544558	4.624228
Albania	ALB	Population density (people per sq. km of land ...	EN.POP.DNST	NaN	60.576642	62.456898	64.329234	66.209307	68.058066

5 rows × 60 columns

Figure 1.14: The first five rows of our dataset

5. To get a quick overview of our dataset, we want to print out its shape.

   This will give us the output in the form **`(rows, columns)`**. Print out the shape using the **`dataset.shape`** command. This works exactly the same as with NumPy ndarrays:

   ```
   # printing the shape of our dataset
   dataset.shape
   ```

   The output of the preceding code is as follows:

    Figure 1.15: The shape of the dataset

Figure 1.15: The shape of the dataset

6. Calculating the mean is the next step, once we've loaded and checked our dataset. Indexing rows work a little bit differently and we'll look at this in detail in the activities that follow. For now, we only want to index the column with the year 1961.

   pandas DataFrames have built-in functions for calculations such as **`mean`**. This means we can simply call **`dataset.mean()`** to get the result.

   The printed output should look as follows:

   ```
   # calculating the mean for 1961 column
   dataset["1961"].mean()
   ```

The output of the preceding code is as follows:

 Figure 1.16: Mean of elements in the 1961 column

Figure 1.16: Mean of elements in the 1961 column

7. Just to see the difference in population density over the years, we'll do the same with the column for the year 2015 (the population more than doubled in the given time range):

```
# calculating the mean for 2015 column
dataset["2015"].mean()
```

The output of the preceding code is as follows:

 Figure 1.17: Mean of elements in the 2015 column

Figure 1.17: Mean of elements in the 2015 column

8. If we want to get the mean for every single country (row), we can make use of pandas **`axis`** tools. By simply passing the **`axis`** parameter in the **`dataset.mean()`** call, we define the dimension our data will be aggregated on.

`axis=0` is horizontal (per column) and **`axis=1`** is vertical (per row), so if we want to have the result for each row, we need to choose **`axis=1`**. Since we have 264 rows in our dataset, we want to restrict the amount of returned countries to only 10. As we discussed before, we can use the **`head(10)`** and **`tail(10)`** methods with a parameter of 10:

```
# mean for each country (row)
dataset.mean(axis=1).head(10)
```

The output of the preceding code is as follows:

 Figure 1.18: Mean of elements in the first 10 countries (rows)

Figure 1.18: Mean of elements in the first 10 countries (rows)

Following is the code for tail method:

```
# mean for each feature (col)
dataset.mean(axis=0).tail(10)
```

The output of the preceding code is as follows:

 Figure 1.19: Mean of elements for the last 10 years (columns)

Figure 1.19: Mean of elements for the last 10 years (columns)

9. The last task of this exercise is to calculate the mean of the whole DataFrame. Since pandas DataFrames can have different datatypes in each column, aggregating this value on the whole dataset out of the box makes no sense. By default, **`axis=0`** will be used, which means that this will give us the same result as the cell before:

```
# calculating the mean for the whole matrix
dataset.mean()
```

The output of the preceding code is as follows:

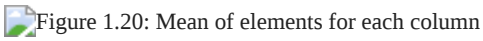
Figure 1.20: Mean of elements for each column

Figure 1.20: Mean of elements for each column

Congratulations! We've now seen that the interface of pandas has some similar methods to NumPy, which makes it really easy to understand. We have now covered the very basics, which will help you solve the first activity using pandas. In the following activity, you will consolidate your basic knowledge of pandas and use the methods you just learned to solve several computational tasks.

Activity 4: Using pandas to Compute the Mean, Median, and Variance for the Given Numbers

In this activity, we will take the previously learned skills of importing datasets and doing some basic calculations and apply them to solve the tasks of our first activity using pandas.

We want to consolidate our new skills and get acquainted with pandas:

1. Open the Jupyter Notebook **`activity04.ipynb`** from the **`Lesson01`** folder to implement this activity.
2. Since we are now working with pandas, we have to import it at the start of our activity to be able to use it in the notebook.
3. Load the **`world_population.csv`** dataset using the **`read_csv`** method of pandas.
4. Given the dataset, go in and use pandas' built-in methods to calculate the mean of the third row, the last row, and the country Germany.
5. Then, calculate the median of the last row, the last 3 rows, and the first 10 countries.
6. Last, calculate the variance of the last 5 columns.

> ***Note:***
>
> *The solution for this activity can be found on page 256.*

Congratulations! You've completed your first activity with pandas, which showed you some of the similarities but also differences when working with NumPy and pandas. In the following activities, this knowledge will be consolidated. You'll also be introduced to more complex features and methods of pandas.

Basic Operations of pandas

In this section, we will learn the basic pandas operations such as Indexing, Slicing, and Iterating and implement it with an Activity.

Indexing

Indexing with pandas is a bit more complex than with NumPy. We can only access columns with the single bracket. To use the indices of the rows to access them, we need the **`iloc`** method. If we want to access them by **`index_col`** (which was set in the **`read_csv`** call), we need to use the **`loc`** method:

```
dataset["2000"] # index the 2000 col
```

```
dataset.iloc[-1] # index the last row
dataset.loc["Germany"] # index the row with index Germany
dataset[["2015"]].loc[["Germany"]] # index row Germany and column 2015
```

Slicing

Slicing with pandas is even more powerful. We can use the default slicing syntax we've already seen with NumPy or use multi-selection. If we want to slice different rows or columns by name, we can simply pass a list into the bracket:

```
dataset.iloc[0:10] # slice of the first 10 rows
dataset.loc[["Germany", "India"]] # slice of rows Germany and India
# subset of Germany and India with years 1970/90
dataset.loc[["Germany", "India"]][["1970", "1990"]]
```

Iterating

Iterating DataFrames is also possible. Considering that they can have several dimensions and dtypes, the indexing is very high level and iterating over each row has to be done separately:

```
# iterating the whole dataset
for index, row in dataset.iterrows():
print(index, row)
```

Series

A pandas Series is a one-dimensional labelled array that is capable of holding any type of data. We can create a Series by loading datasets from a .csv file, Excel spreadsheet, or SQL database. There are many different ways to create them. For example:

- NumPy arrays:

  ```
  # import pandas
  import pandas as pd
  # import numpy
  import numpy as np
  # creating a numpy array
  numarr = np.array(['p','y','t','h','o','n'])
  ser = pd.Series(numarr)
  print(ser)
  ```

- pandas lists:

  ```
  # import pandas
  import pandas as pd
  # creating a pandas list
  plist = ['p','y','t','h','o','n']
  ```

```
ser = pd.Series(plist)
print(ser)
```

Activity 5: Indexing, Slicing, and Iterating using pandas

In this activity, we will use previously discussed pandas features to index, slice, and iterate DataFrames using pandas Series. To get some understandable insights into our dataset, we need to be able to explicitly index, slice, and iterate our data. For example, we can compare several countries in terms of population density growth.

After looking at the distinct operations, we want to display the population density of Germany, Singapore, the United States, and India for the years 1970, 1990, and 2010:

1. Open the **`activity05.ipynb`** Jupyter Notebook from the **`Lesson01`** folder to implement this activity.
2. Before loading the dataset, we need to import pandas. We'll again use the **`pd`** alias to refer to pandas.
3. Load the **`world_population.csv`** dataset using pandas. Make sure everything works by having a look at the DataFrames.
4. With the dataset loaded, use the previously discussed indexing feature to index the row for the USA, the second to last row, the column of the year 2000 as Series, and the population density for India in 2000.
5. Creating sub-lists of the dataset requires slicing. Use this concept to slice the countries in rows 2 to 5, slice the countries Germany, Singapore, the United States, and India, and slice Germany, Singapore, the United States, and India with their population density for the years 1970, 1990, and 2010.
6. The last task in this activity will be iterating over the first three countries in our dataset. Use the previously discussed iteration method to iterate over the whole dataset and print the name, country code, and population for the years 1970, 1990, and 2010.

Note:

The solution for this activity can be found on page 261.

Congratulations! We've already covered most of the basic data wrangling methods using pandas. In the next activity, we'll take a look at more advanced features such as filtering, sorting, and reshaping to prepare you for the next chapter.

Advanced pandas Operations

In this section, we will learn advanced pandas operations such as filtering, sorting, and reshaping and implement them in an activity.

Filtering

Filtering in pandas has a higher-level interface than NumPy. You can still use the "simple" brackets-based **conditional filtering**. However, you're also able to use more complex queries, for example, filter rows based on a regular expression:

```
dataset.filter(items=["1990"]) # only column 1994
dataset[(dataset["1990"] < 10)] # countries'population density < 10 in 1999
dataset.filter(like="8", axis=1) # years containing an 8
```

```
dataset.filter(regex="a$", axis=0) # countries ending with a
```

Sorting

Sorting each row or column based on a given row or column will help you get better insights into your data and find the ranking of a given dataset. With pandas, we are able to do this pretty easily. Sorting in ascending and descending order can be done using the parameter known as **ascending**. Of course, you can do more complex sorting by providing more than one value in the **`by = [ ]`** list. Those will then be used to sort values for which the first value is the same:

```
dataset.sort_values(by=["1999"]) # values sorted by 1999
# values sorted by 1999 descending
dataset.sort_values(by=["1994"], ascending=False)
```

Reshaping

Reshaping can be crucial for easier visualization and algorithms. However, depending on your data, this can get really complex:

```
dataset.pivot(index=["1999"] * len(dataset), columns="Country Code", values="1999")
```

Note

Reshaping is a very complex topic. If you want to dive deeper into it, this is a good resource to get started: https://bit.ly/2SjWzaB.

Activity 6: Filtering, Sorting, and Reshaping

This last activity for pandas provides some more complex tasks and also combines most of the methods learned previously as a recap. After this activity, students should be able to read the most basic pandas code and understand its logic:

1. Open the **`activity06.ipynb`** Jupyter Notebook from the **`Lesson01`** folder to implement this activity. Refer back to the introduction of this chapter for instructions on how to open a Jupyter Notebook in JupyterLab.

2. Filtering, sorting, and reshaping are all pandas methods. Before we can use them, we have to import pandas.

3. Again, load the **`world_population.csv`** dataset using pandas and make sure everything works by having a look at the DataFrames:
   Figure 1.21: Looking at the first two columns of our dataset

   Figure 1.21: Looking at the first two columns of our dataset

4. After loading the dataset, use the previously discussed filtering features to get the years 1961, 2000, and 2015 as columns, and all the countries that had a greater population density than 500 in the year 2000. Also, use filtering to get a new dataset that only contains the years 2000 and later, countries that start with A, and countries that contain the word "land."

5. Sorting data is an important feature when trying to display data on an ordinal datatype column. Use the sorting operation to sort values in ascending order by 1961, in ascending order by 2015, and in descending order by 2015.

6. If our input dataset does not suit our needs, we have to reshape it. Use the reshaping features to reshape the dataset to 2015 as row and country codes as columns.

Note:

The solution for this activity can be found on page 267.

You've now completed the topic about pandas, which concludes this chapter. We've learned about the basic tools that help you wrangle and work with data. pandas itself is an incredibly powerful and heavily used tool for data wrangling.

Summary

NumPy and pandas are essential tools for data wrangling. Their user-friendly interfaces and performant implementation make data handling easy. Even though they only provide a little insight into our datasets, they are absolutely valuable for wrangling, augmenting, and cleaning our datasets. Mastering these skills will improve the quality of your visualizations.

In this chapter, we learned the basics of NumPy and pandas, and statistics concepts. Even though the statistical concepts covered are very basic, they are necessary to enrich our visualizations with information that, in most cases, is not directly provided in our datasets. This hands-on experience will help you implement exercises and activities in the following chapters.

In the next chapter, we will focus on the different types of visualizations and how to decide which visualization would be best in your case. This will give you theoretical knowledge so that you know when to use a specific chart type and why. It will also lay down the fundamentals of the chapters, which will heavily focus on teaching you how to use Matplotlib and seaborn to create the plots discussed. After we have covered basic visualization techniques with Matplotlib and seaborn, we will dive deeper and explore the possibilities of interactive and animated charts, which will introduce the element of storytelling into our visualizations.

Chapter 2

All You Need to Know About Plots

Learning Objectives

By the end of this chapter, you will be able to:

- Identify the best plot type for a given dataset and scenario
- Explain the design practices of certain plots
- Design outstanding, tangible visualizations

In this chapter, we will learn the basics of different types of plots.

Introduction

In this chapter, we will focus on various visualizations and identify which visualization is best to show certain information for a given dataset. We will describe every visualization in detail and give practical examples, such as comparing different stocks over time or comparing the ratings for different movies. Starting with comparison plots, which are great for comparing multiple variables over time, we will look at their types, such as line charts, bar charts, and radar charts. Relation plots are handy to show relationships among variables. We will cover scatter plots for showing the relationship between two variables, bubble plots for three variables, correlograms for variable pairs, and, finally, heatmaps.

Composition plots, which are used to visualize variables that are part of a whole, as well as pie charts, stacked bar charts, stacked area charts, and Venn diagrams are going to be explained. To get a deeper insight into the distribution of variables, distribution plots are used. As a part of distribution plots, histograms, density plots, box plots, and violin plots will be covered. Finally, we will talk about dot maps, connection maps, and choropleth maps, which can be categorized into geo plots. Geo plots are useful for visualizing geospatial data.

Comparison Plots

Comparison plots include charts that are well-suited for comparing multiple variables or variables over time. For a comparison among items, bar charts (also called column charts) are the best way to go. Line charts are great for visualizing variables over time. For a certain time period (say, less than ten time points), vertical bar charts can be used as well. Radar charts or spider plots are great for visualizing multiple variables for multiple groups.

Line Chart

Line charts are used to display quantitative values over a continuous time period and show information as a series. A line chart is ideal for a time series, which is connected by straight-line segments.

The value is placed on the y-axis, while the x-axis is the timescale.

Uses:

- Line charts are great for comparing multiple variables and visualizing trends for both single as well as multiple variables, especially if your dataset has many time periods (roughly more than ten).
- For smaller time periods, vertical bar charts might be the better choice.

The following diagram shows a trend of real-estate prices (in million US dollars) for two decades. Line charts are well-suited for showing data trends:

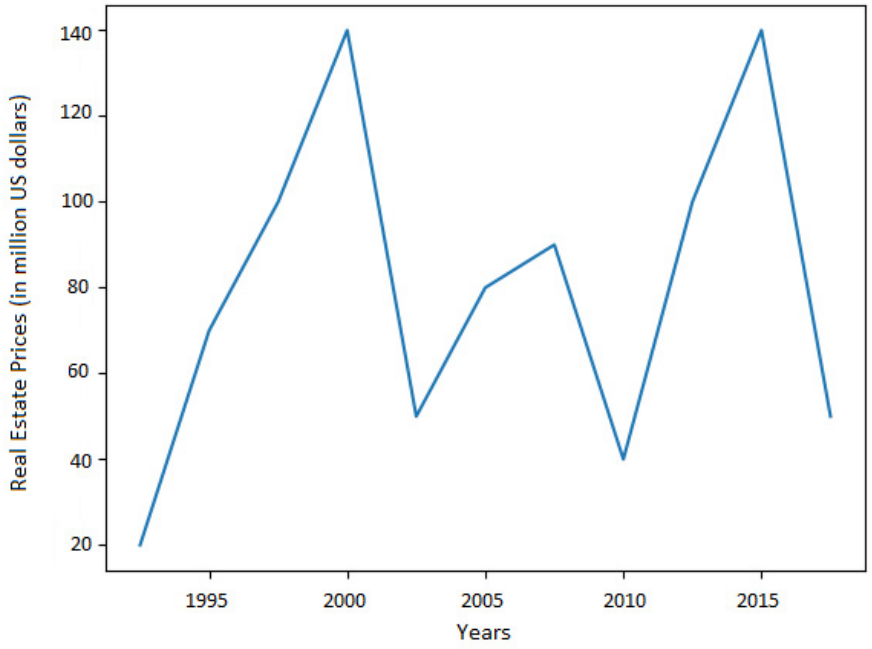

Figure 2.1: Line chart for a single variable

Example:

The following diagram is a multiple variable line chart that compares the stock-closing prices for Google, Facebook, Apple, Amazon, and Microsoft. A line chart is great for comparing values and visualizing the trend of the stock. As we can see, Amazon shows the highest growth:

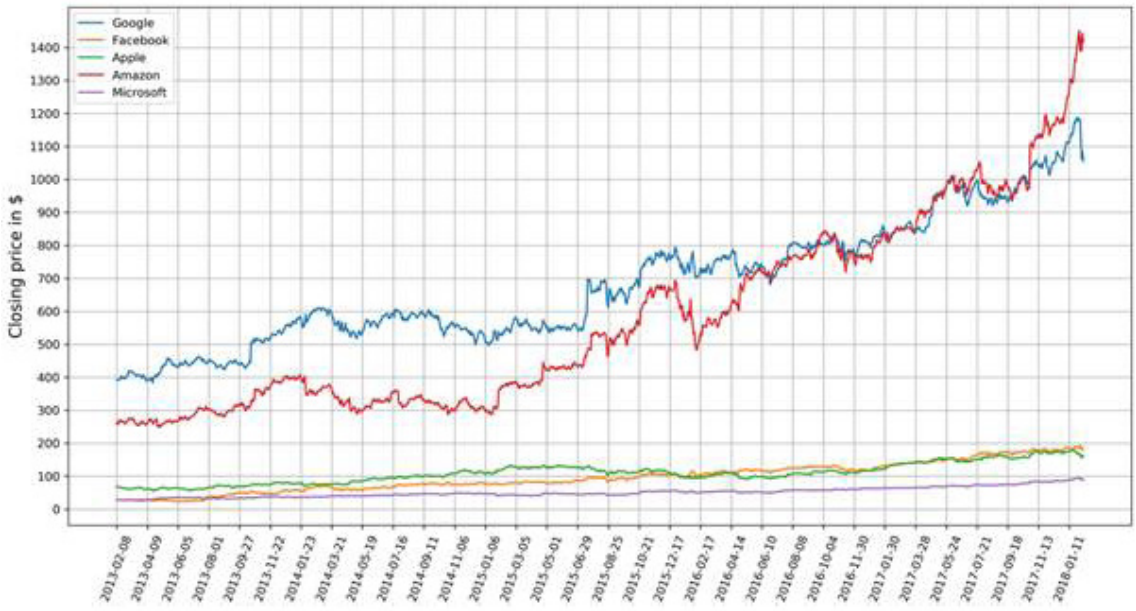

Figure 2.2: Line chart showing stock trends for the five companies

Design practices:

- Avoid too many lines per chart
- Adjust your scale so that the trend is clearly visible

Note

Design practices for plots with multiple variables. A legend should be available to describe each variable.

Bar Chart

The bar length encodes the value. There are two variants of bar charts: vertical bar charts and horizontal bar charts.

Uses:

- While they are both used to compare numerical values across categories, vertical bar charts are sometimes used to show a single variable over time.

The do's and the don'ts of bar charts:

- Don't confuse vertical bar charts with histograms. Bar charts compare different variables or categories, while histograms show the distribution for a single variable. Histograms will be discussed later in this chapter.
- Another common mistake is to use bar charts to show central tendencies among groups or categories. Use box plots or violin plots to show statistical measures or distributions in these cases.

Examples:

The following diagram shows a vertical bar chart. Each bar shows the marks out of 100 that five students obtained in a test:

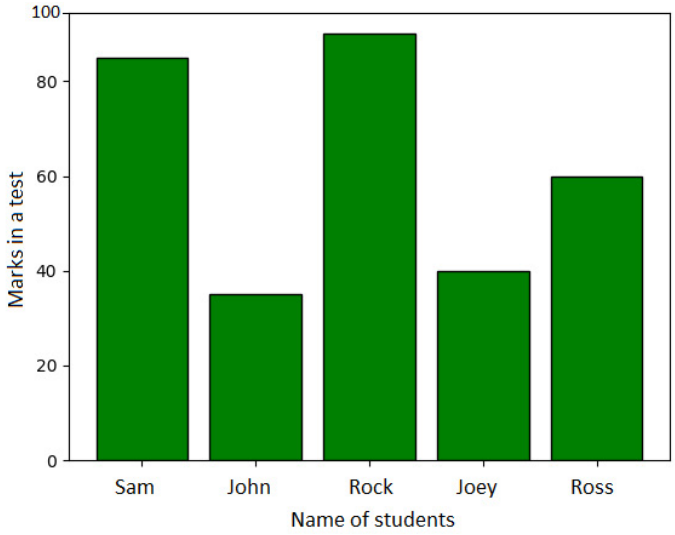

Figure 2.3: Vertical bar chart using student test data

The following diagram shows a horizontal bar chart. Each bar shows the marks out of 100 that five students obtained in a test:

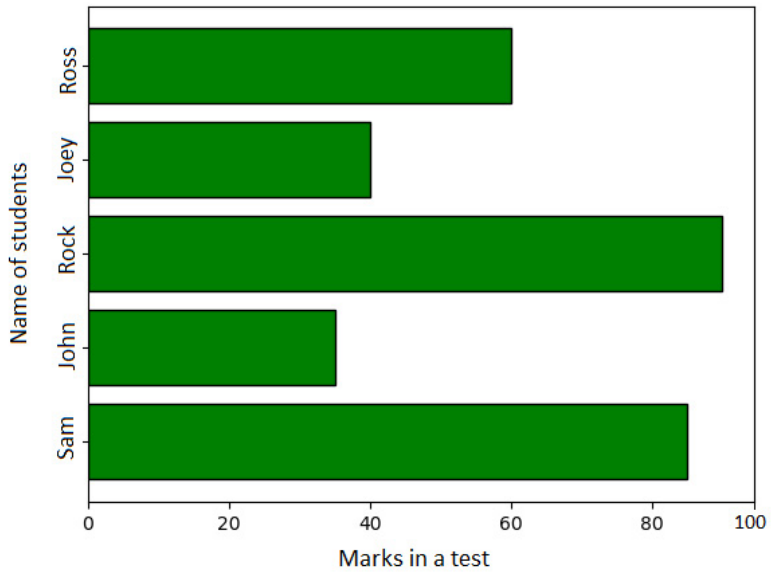

Figure 2.4: Horizontal bar chart using student test data

The following diagram compares movie ratings, giving two different scores. The Tomatometer is the percentage of approved critics who have given a positive review for the movie. The Audience Score is the percentage of users who have given a score of 3.5 or higher out of 5. As we can see, **The Martian** is the only movie with both a high Tomatometer score and Audience Score. **The Hobbit: An Unexpected Journey** has a relatively high Audience Score compared to the Tomatometer score, which might be due to a huge fan base:

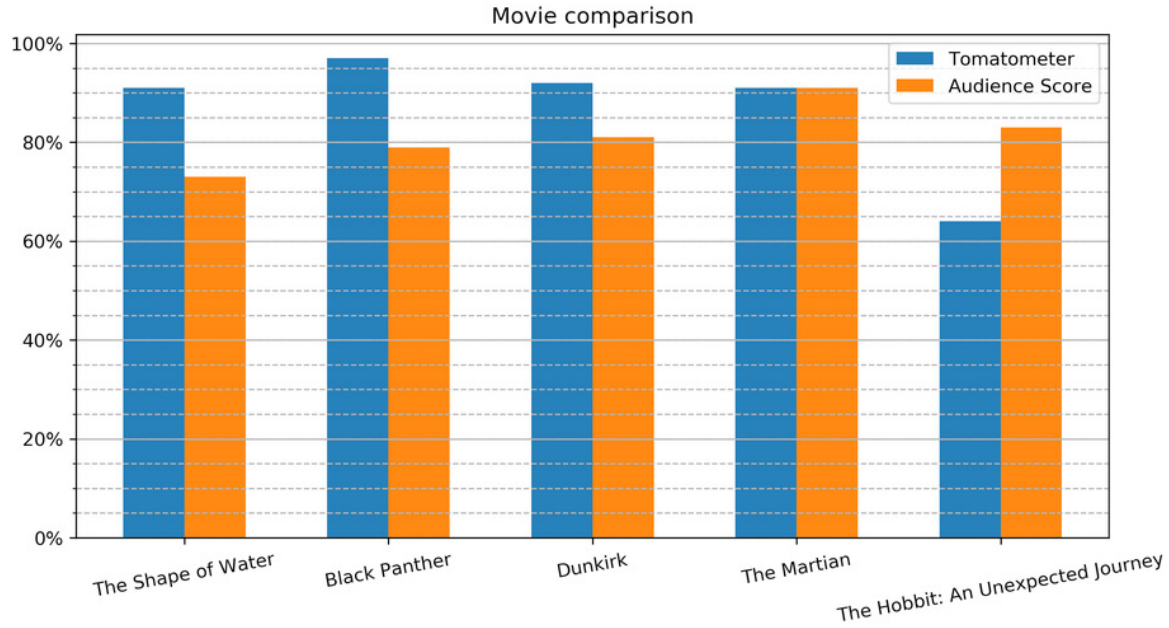

Figure 2.5: Comparative bar chart

Design practices:

- The axis corresponding to the numerical variable should start at zero. Starting with another value might be misleading, as it makes a small value difference look like a big one.
- Use horizontal labels, that is, as long as the number of bars is small and the chart doesn't look too cluttered.

Radar Chart

Radar charts, also known as **spider** or **web charts**, visualize multiple variables with each variable plotted on its own axis, resulting in a polygon. All axes are arranged radially, starting at the center with equal distances between one another and have the same scale.

Uses:

- Radar charts are great for comparing multiple quantitative variables for a single group or multiple groups.
- They are also useful to show which variables score high or low within a dataset, making them ideal to visualize performance

Examples:

The following diagram shows a radar chart for a single variable. This chart displays data about a student scoring marks in different subjects:

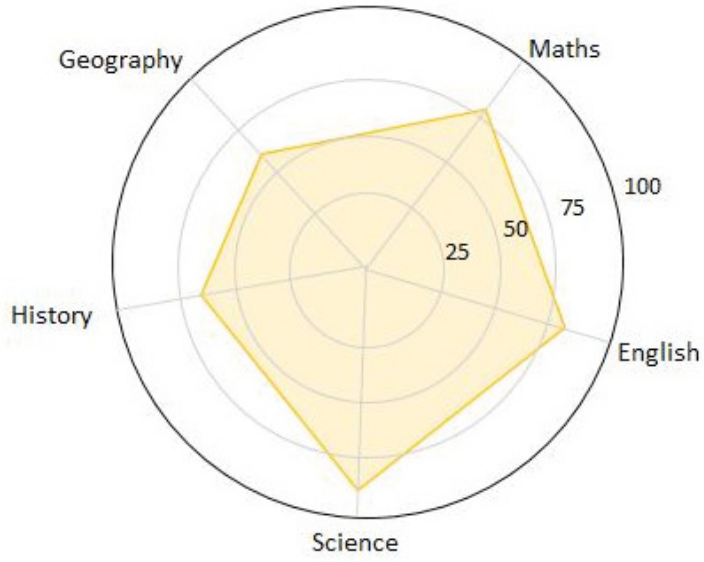

Figure 2.6: Radar chart for one variable (student)

The following diagram shows a radar chart for two variables/groups. Here, the chart explains the marks that were scored by two students in different subjects:

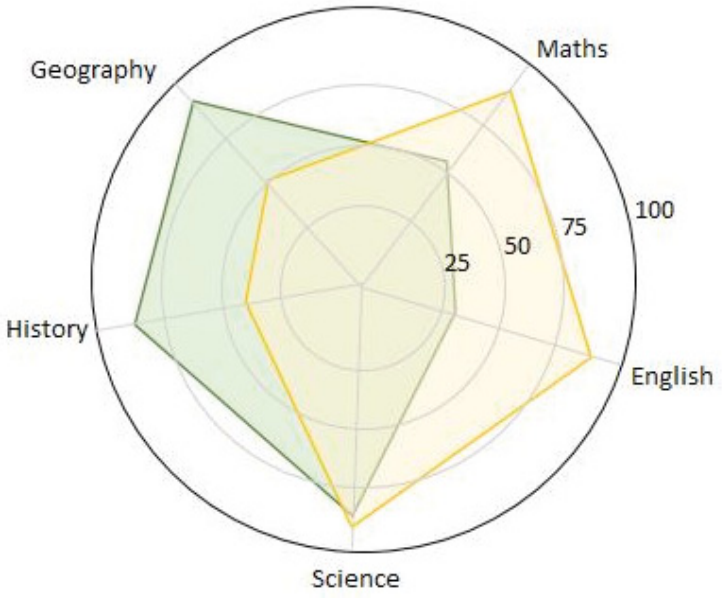

Figure 2.7: Radar chart for two variables (two students)

The following diagram shows a radar chart for multiple variables/groups. Each chart displays data about a student's performance in different subjects:

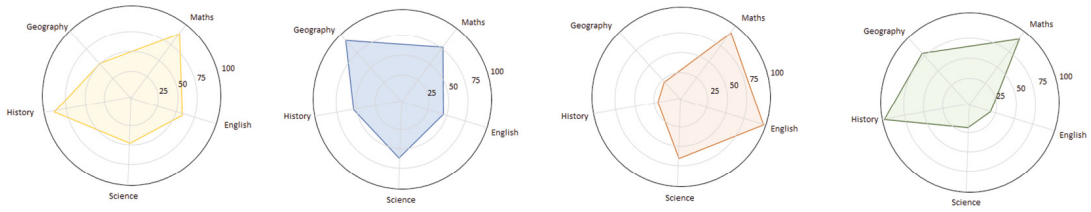

Figure 2.8: Radar chart with faceting for multiple variables (multiple subjects)

Design practices:

- Try to display ten factors or fewer on one radar chart to make it easier to read.
- Use **faceting** for multiple variables/groups, as shown in the preceding diagram, to maintain clarity.

Activity 7: Employee Skill Comparison

You are given scores of four employees (A, B, C, and D) for five attributes: Efficiency, Quality, Commitment, Responsible Conduct, and Cooperation. Your task is to compare the employees and their skills:

1. Which charts are suitable for this task?
2. You are given the following bar and radar charts. List the advantages and disadvantages for both charts. Which is the better chart for this task in your opinion and why?

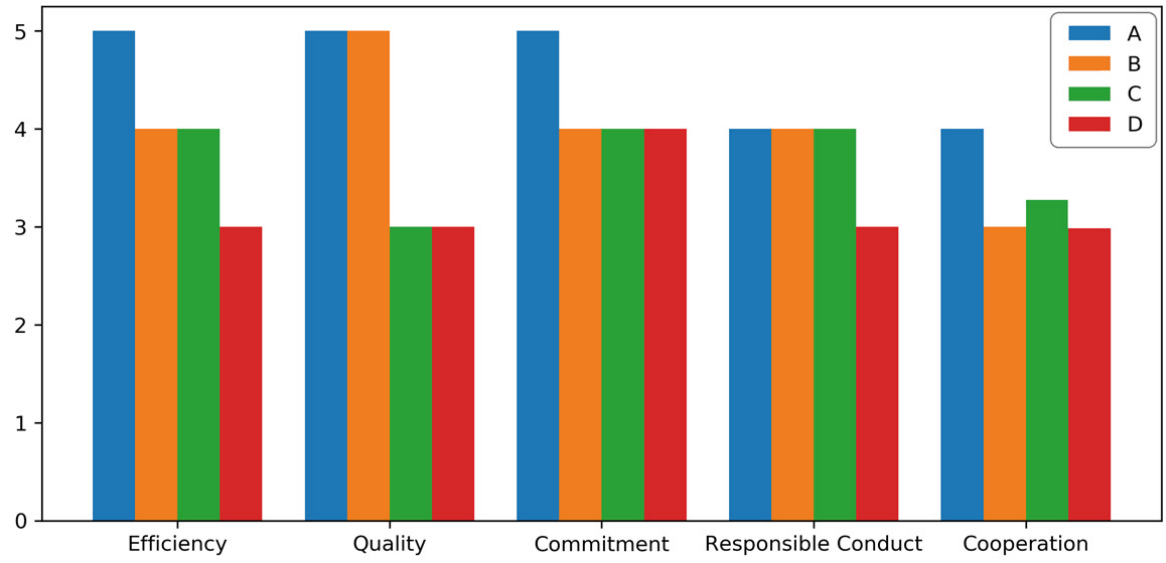

Figure 2.9: Employee skills comparison with a bar chart

The following figure shows a radar chart for employee skills:

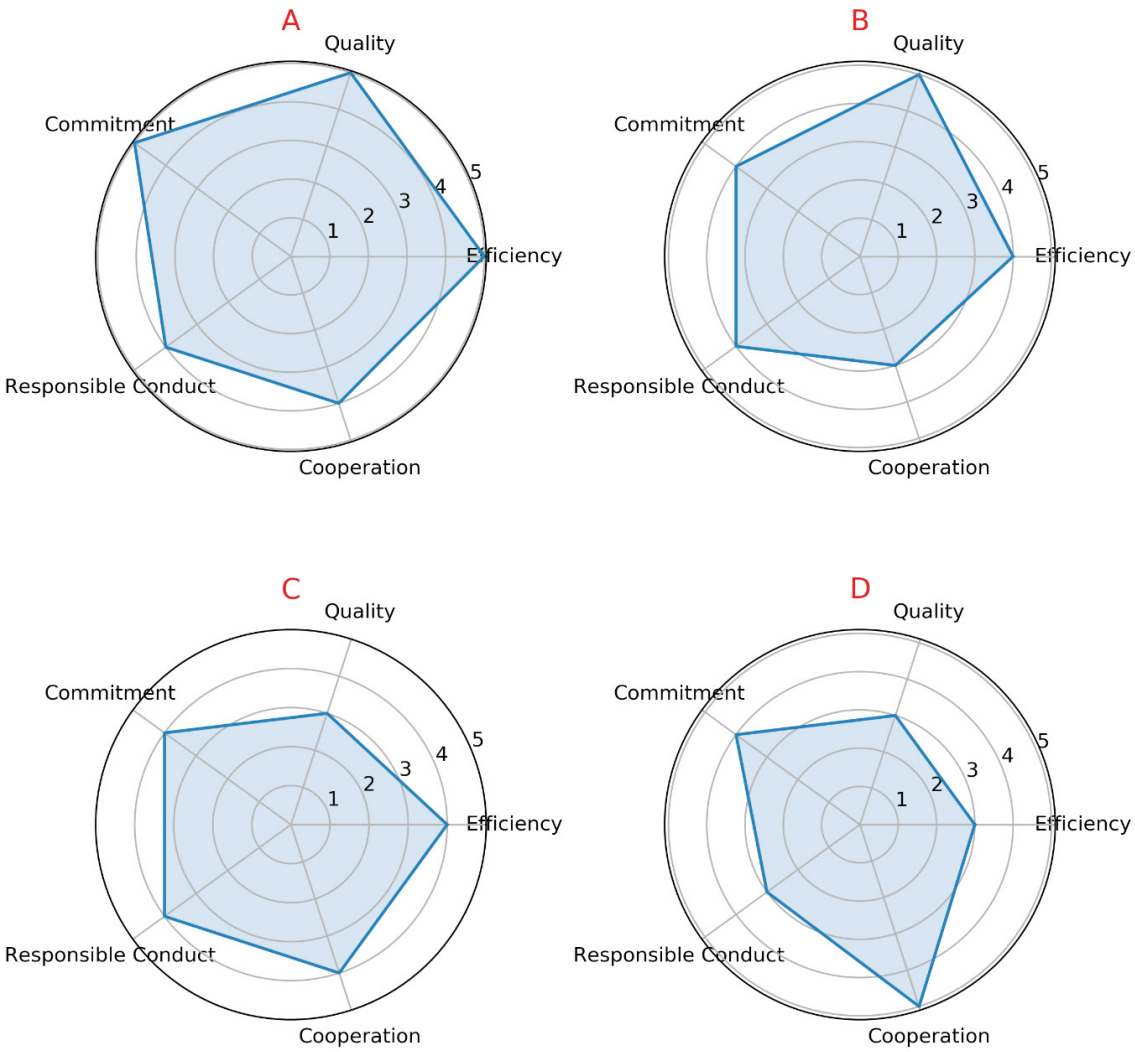

Figure 2.10: Employee skills comparison with a radar chart

3. What could be improved in the respective visualizations?

Note:

The solution for this activity can be found on page 275.

Relation Plots

Relation plots are perfectly suited to show relationships among variables. A scatter plot visualizes the correlation between two variables for one or multiple groups. Bubble plots can be used to show relationships between three variables. The additional third variable is represented by the dot size. Heatmaps are great for revealing patterns or correlating between two qualitative variables. A correlogram is a perfect visualization to show the correlation among multiple variables.

Scatter Plot

Scatter plots show data points for two numerical variables, displaying a variable on both axes.

Uses:

- You can detect whether a correlation (relationship) exists between two variables.
- They allow you to plot the relationship for multiple groups or categories using different colors.
- A bubble plot, which is a variation of the scatter plot, is an excellent tool for visualizing the correlation of a third variable.

Examples:

The following diagram shows a scatter plot of **height** and **weight** of persons belonging to a single group:

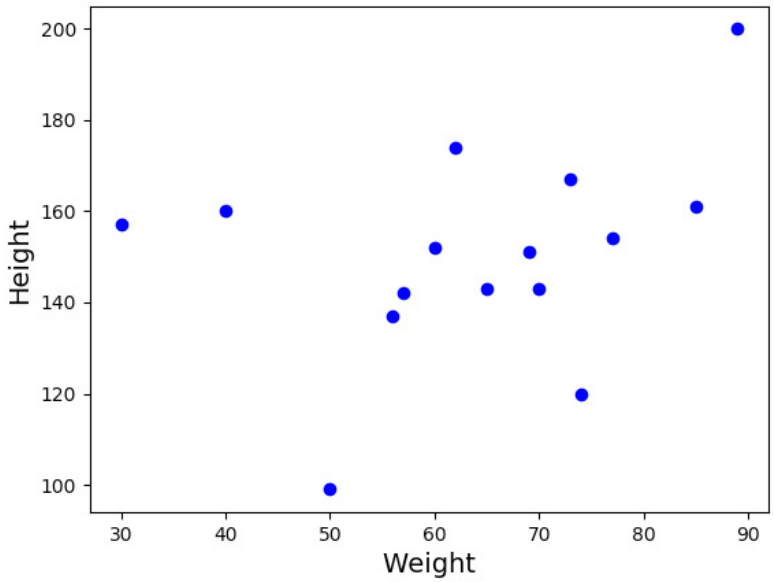

Figure 2.11: Scatter plot with a single variable (one group)

The following diagram shows the same data as in the previous plot but differentiates between groups. In this case, we have different groups: **A**, **B**, and **C**:

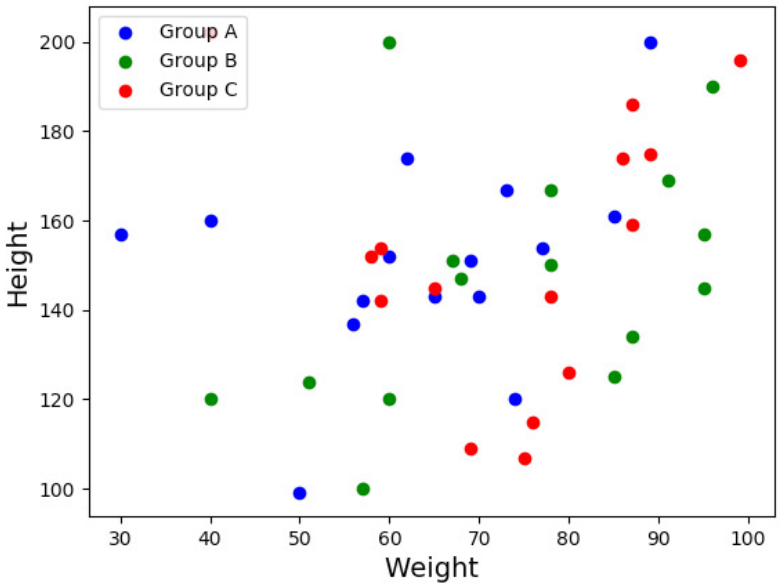

Figure 2.12: Scatter plot with multiple variables (three groups)

The following diagram shows the correlation between the body mass and the maximum longevity for various animals grouped by their classes. There is a positive correlation between the body mass and the maximum longevity:

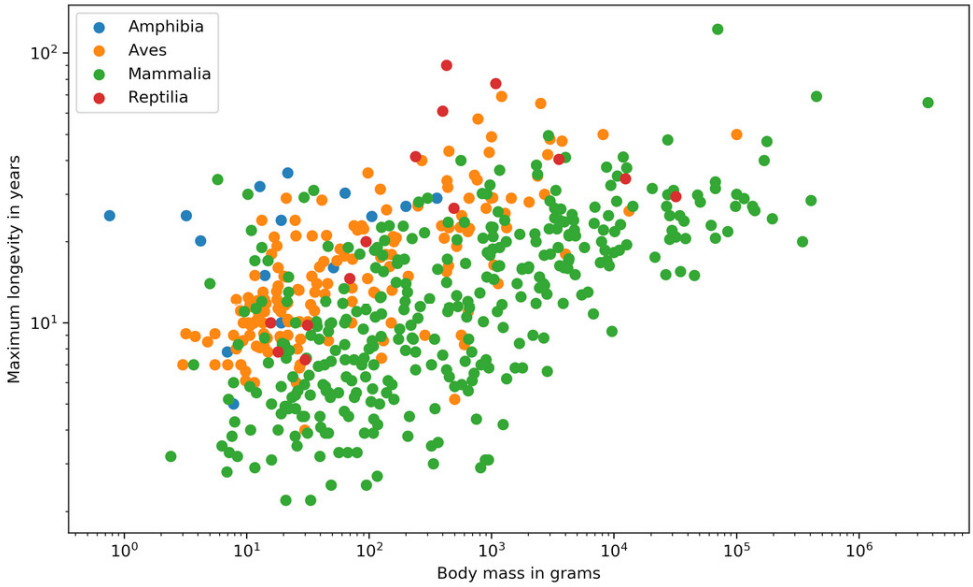

Figure 2.13: Correlation between body mass and maximum longevity for animals

Design practices:

- Start both axes at zero to represent data accurately.
- Use contrasting colors for data points and avoid using symbols for scatter plots with multiple groups or categories.

Variants: **scatter plots with marginal histograms**

In addition to the scatter plot, which visualizes the correlation between two numerical variables, you can plot the marginal distribution for each variable in the form of histograms to give better insight into how each variable is distributed.

Examples:

The following diagram shows the correlation between the body mass and the maximum longevity for animals in the Aves class. The marginal histograms are also shown, which helps to get a better insight into both variables:

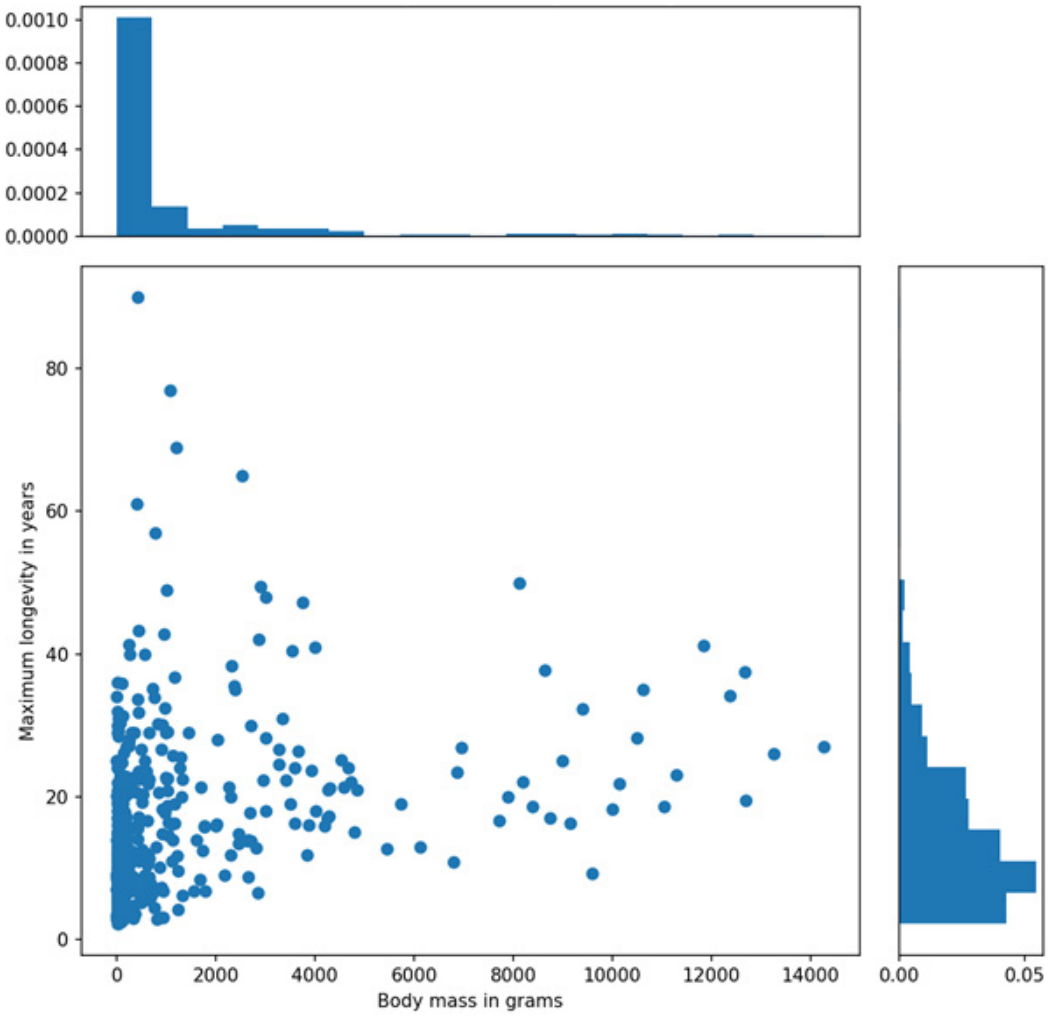

Figure 2.14: Correlation between body mass and maximum longevity of the Aves class with marginal histograms

Bubble Plot

A **bubble plot** extends a scatter plot by introducing a third numerical variable. The value of the variable is represented by the size of the dots. The area of the dots is proportional to the value. A legend is used to link the size of the dot to an actual numerical value.

Uses:

- To show a correlation between three variables.

Example:

The following diagram shows a bubble plot that highlights the relationship between heights and age of humans:

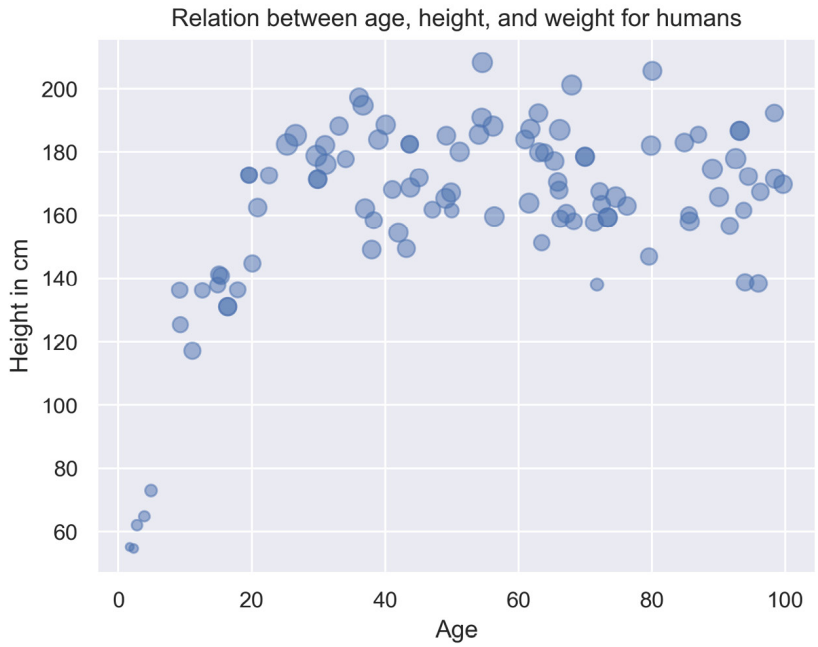

Figure 2.15: Bubble plot showing relation between height and age of humans

Design practices:

- Design practices for the scatter plot are also applicable to the bubble plot.
- Don't use it for very large amounts of data, since too many bubbles make the chart hard to read.

Correlogram

A **correlogram** is a combination of scatter plots and histograms. Histograms will be discussed in detail later in this chapter. A correlogram or correlation matrix visualizes the relationship between each pair of numerical variables using a scatter plot.

The diagonals of the correlation matrix represent the distribution of each variable in the form of a histogram. You can also plot the relationship for multiple groups or categories using different colors. A correlogram is a great chart for exploratory data analysis to get a feeling for your data, especially the correlation between variable pairs.

Examples:

The following diagram shows a correlogram for height, weight, and age of humans. The diagonal plots show a histogram for each variable. The off-diagonal elements show scatter plots between variable pairs:

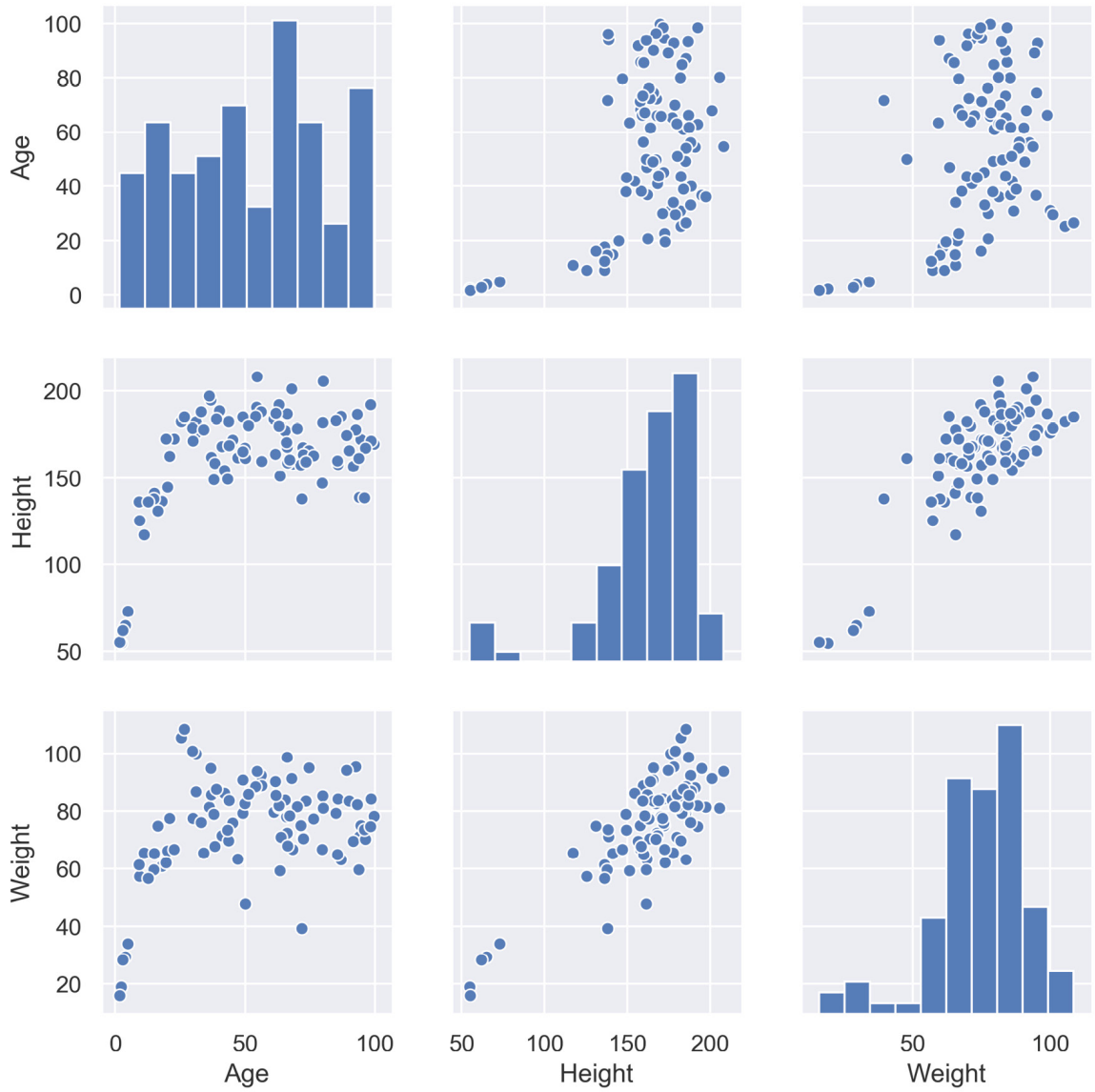

Figure 2.16: Correlogram with single category

The following diagram shows the correlogram with data samples separated by color into different groups:

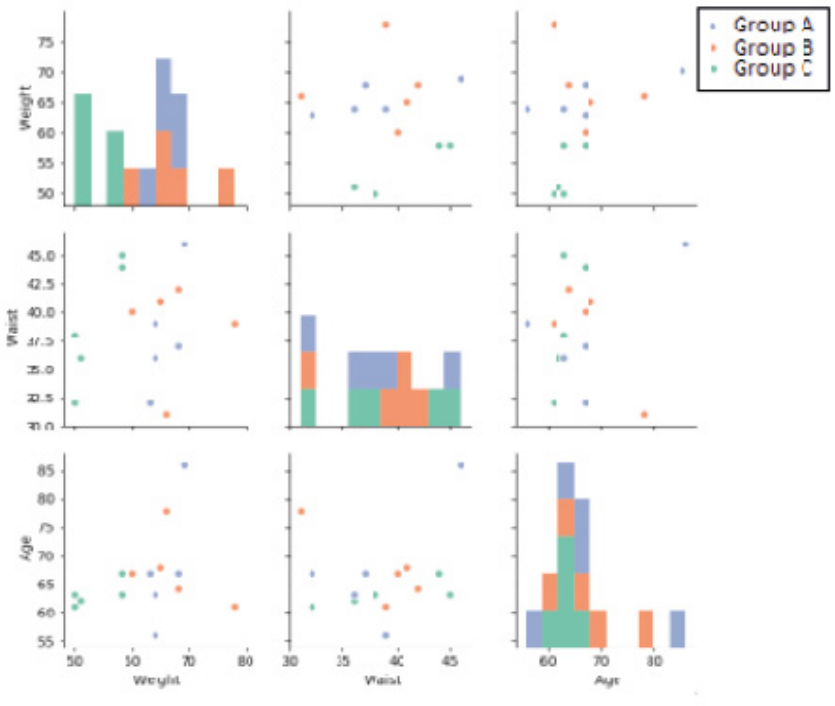

Figure 2.17: Correlogram with multiple categories

Design practices:

- Start both axes at zero to represent data accurately.
- Use contrasting colors for data points and avoid using symbols for scatter plots with multiple groups or categories.

Heatmap

A **heatmap** is a visualization where values contained in a matrix are represented as colors or color saturation. Heatmaps are great for visualizing multivariate data, where categorical variables are placed in the rows and columns and a numerical or categorical variable is represented as colors or color saturation.

Uses:

- Visualization of multivariate data. Great for finding patterns in your data.

Examples:

The following diagram shows a heatmap for the most popular products on the Electronics category page across various e-commerce websites:

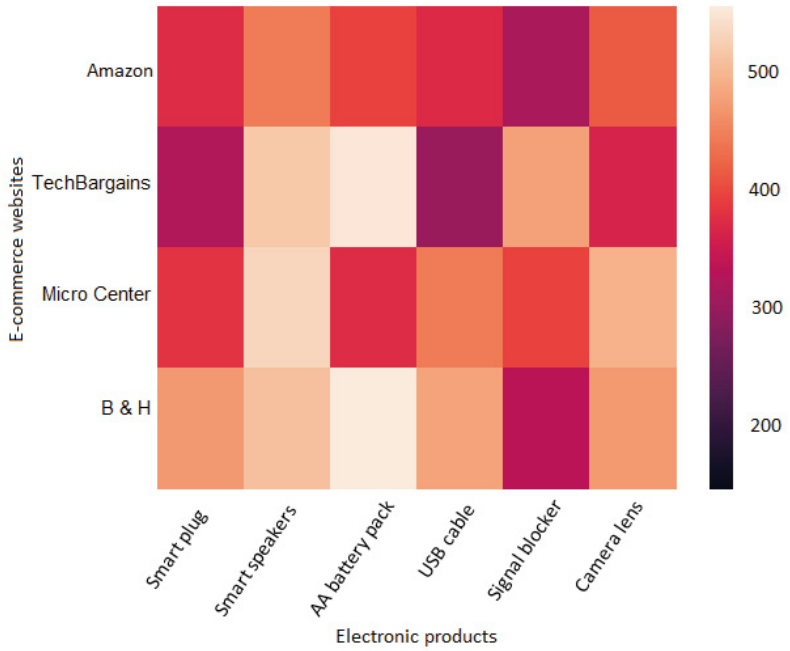

Figure 2.18: Heatmap for popular products in the Electronics category

Variants: **annotated heatmaps**

Let's see the same example the we saw previously in an annotated heatmap:

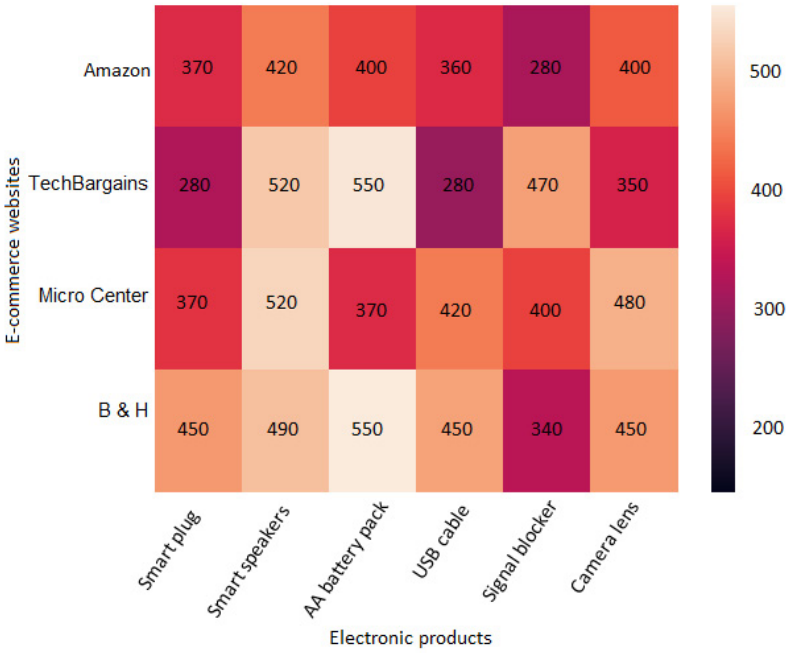

Figure 2.19: Annotated heatmap for popular products in the Electronics category

Activity 8: Road Accidents Occurring over Two Decades

You are given a diagram that gives information about the road accidents that have occurred over the past two decades during the months of January, April, July, and October:

1. Identify the year during which the number of road accidents occurred were the least.
2. For the past two decades, identify the month for which accidents show a marked decrease:

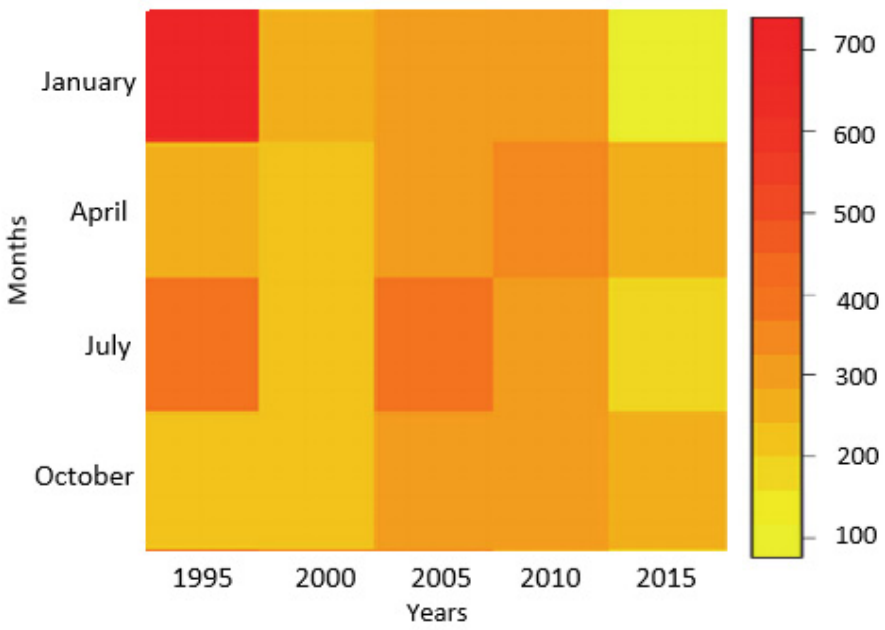

Figure 2.20: Total accidents over 20 years

Note:

The solution for this activity can be found on page 275.

Composition Plots

Composition plots are ideal if you think about something as a part of a whole. For static data, you can use pie charts, stacked bar charts, or Venn diagrams. **Pie charts** or **donut charts** help show proportions and percentages for groups. If you need an additional dimension, stacked bar charts are great. Venn diagrams are the best way to visualize overlapping groups, where each group is represented by a circle. For data that changes over time, you can use either stacked bar charts or stacked area charts.

Pie Chart

Pie charts illustrate numerical proportion by dividing a circle into slices. Each arc length represents a proportion of a category. The full circle equals to 100%. For humans, it is easier to compare bars than arc lengths; therefore, it is recommended to use bar charts or stacked bar charts most of the time.

Uses:

- Compare items that are part of a whole.

Examples:

The following diagram shows a pie chart that shows different fielding positions of the cricket ground, such as long on, long off, third man, and fine leg:

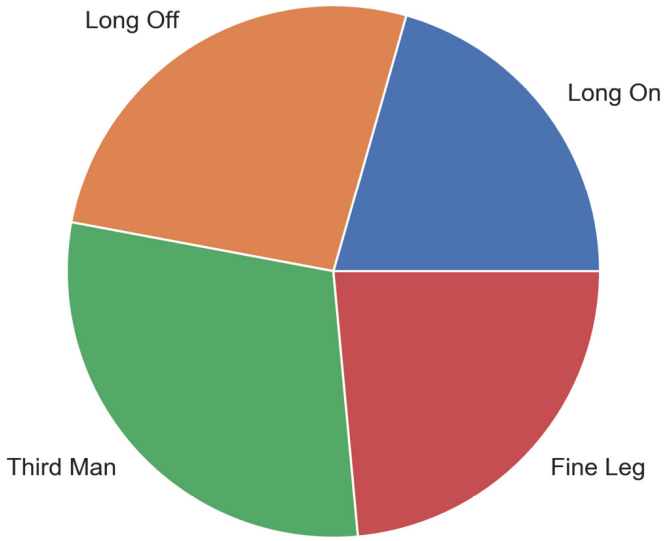

Figure 2.21: Pie chart showing fielding positions in a cricket ground

The following diagram shows water usage around the world:

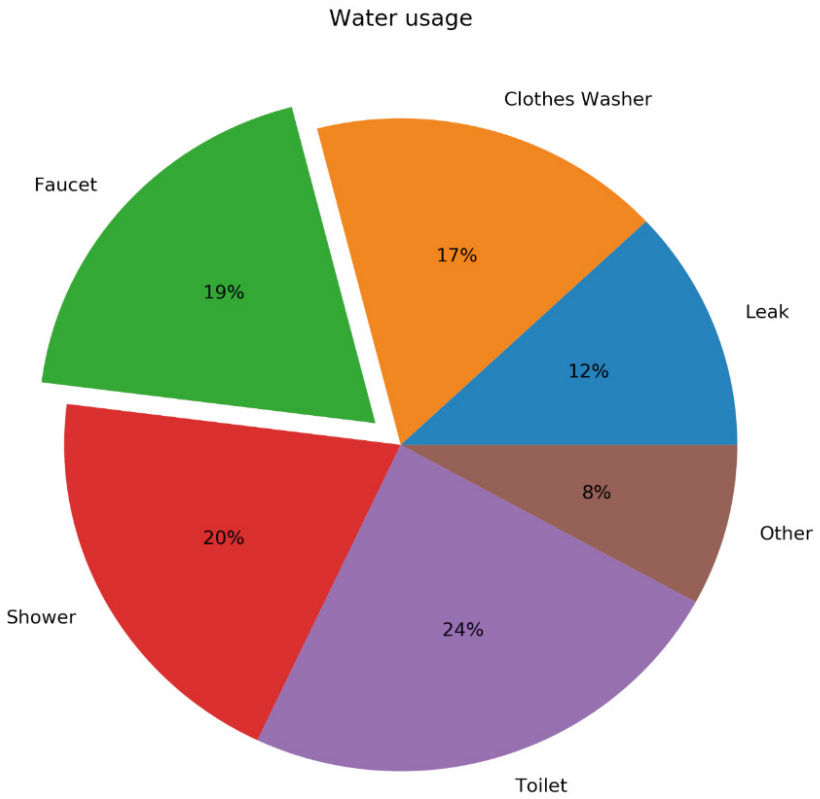

Figure 2.22: Pie chart for global water usage

Design practices:

- Arrange the slices according to their size in increasing/decreasing order, either in a clockwise or anticlockwise manner.
- Make sure that every slice has a different color.

Variants: **donut chart**

An alternative to a pie chart is a **donut chart**. In contrast to pie charts, it is easier to compare the size of slices, since the reader focuses more on reading the length of the arcs instead of the area. Donut charts are also more space-efficient because the center is cut out, so it can be used to display information or further divide groups into sub-groups.

The following figure shows a basic donut chart:

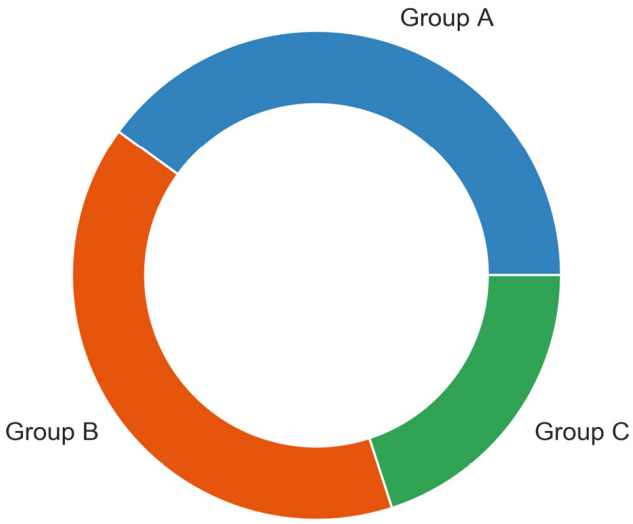

Figure 2.23: Donut chart

The following figure shows a donut chart with subgroups:

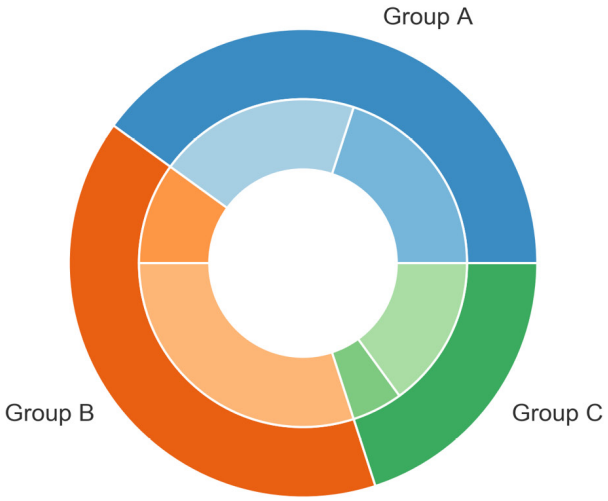

Figure 2.24: Donut chart with subgroups

Design practices:

- Use the same color (that's used for the category) for the subcategories. Use varying brightness levels for the different subcategories.

Stacked Bar Chart

Stacked bar charts are used to show how a category is divided into sub-categories and the proportion of the sub-category, in comparison to the overall category. You can either compare total amounts across each bar or show a percentage of each group. The latter is also referenced as a **100% stacked bar chart** and makes it easier to see relative differences between quantities in each group.

Uses:

- Compare variables that can be divided into sub-variables.

Examples:

The following diagram shows a generic stacked bar chart with five groups:

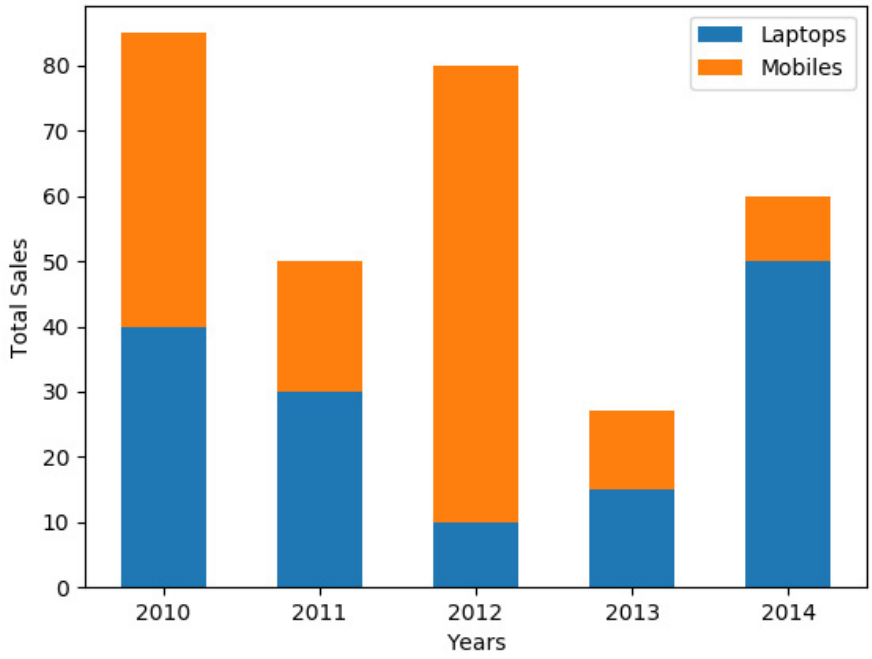

Figure 2.25: Stacked bar chart to show sales of laptops and mobiles

The following diagram shows a 100% stacked bar chart with the same data that was used in the preceding diagram:

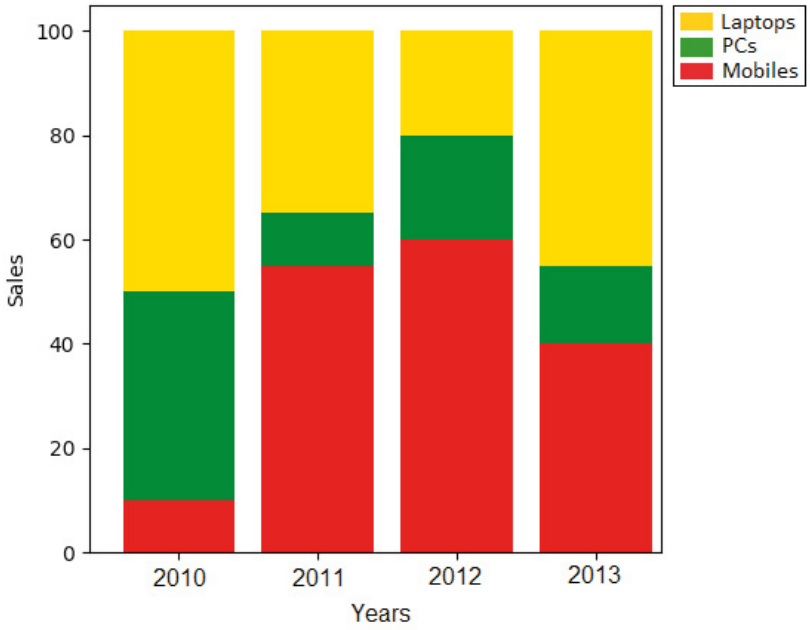

Figure 2.26: 100% stacked bar chart to show sales of laptops, PCs, and mobiles

The following diagram illustrates the daily total sales of a restaurant over several days. The daily total sales of non-smokers are stacked on top of the daily total sales of smokers:

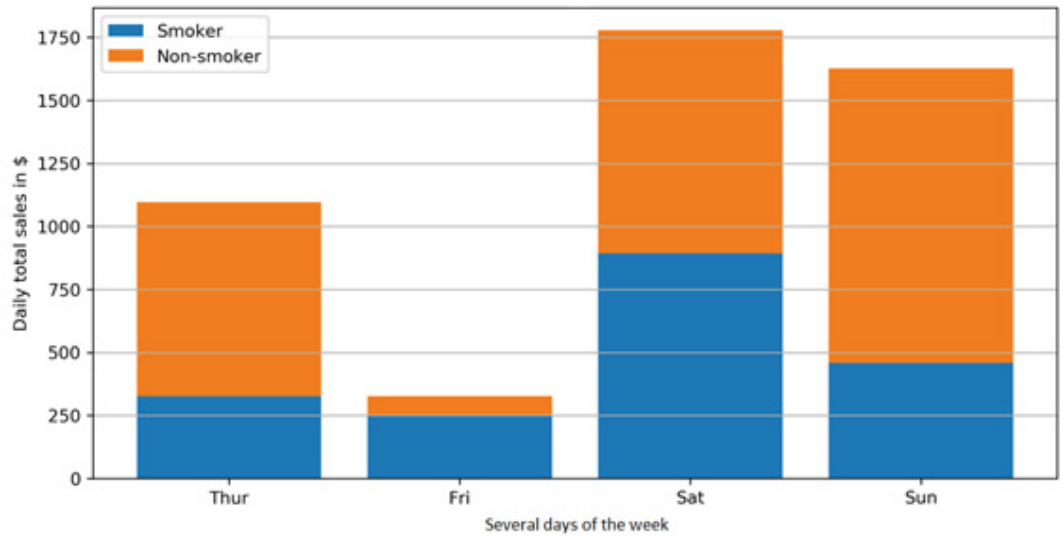

Figure 2.27: Daily total sales of restaurant categorized by smokers and non-smokers

Design practices:

- Use contrasting colors for stacked bars.
- Ensure that the bars are adequately spaced to eliminate visual clutter. The ideal space guideline between each bar is half the width of a bar.

- Categorize data alphabetically, sequentially, or by value to order it uniformly and make things easier for your audience.

Stacked Area Chart

Stacked area charts show trends for part-of-a-whole relations. The values of several groups are illustrated on top of one another. It helps to analyze both individual and overall trend information.

Uses:

- Show trends for time series that are part of a whole.

Examples:

The following diagram shows a stacked area chart with the net profits of companies like Google, Facebook, Twitter, and Snapchat over a decade:

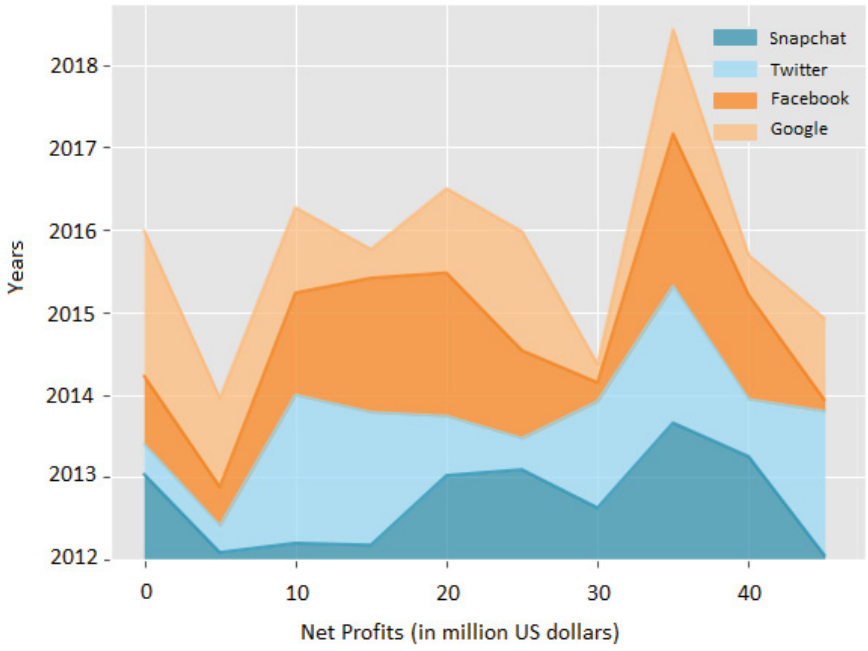

Figure 2.28: Stacked area chart to show net profits of four companies

Design practices:

- Using transparent colors might improve information visibility.

Activity 9: Smartphone Sales Units

You want to compare smartphone sales units for the five biggest smartphone manufacturers over time and see whether there is any trend:

1. Looking at the following line chart, analyze the sales of each manufacturer and identify the one whose performance is exceptional in the fourth quarter when compared to the third quarter.
2. Analyze the performance of all manufacturers, and make a prediction about two companies whose sales unit will show a downward trend and an upward trend:

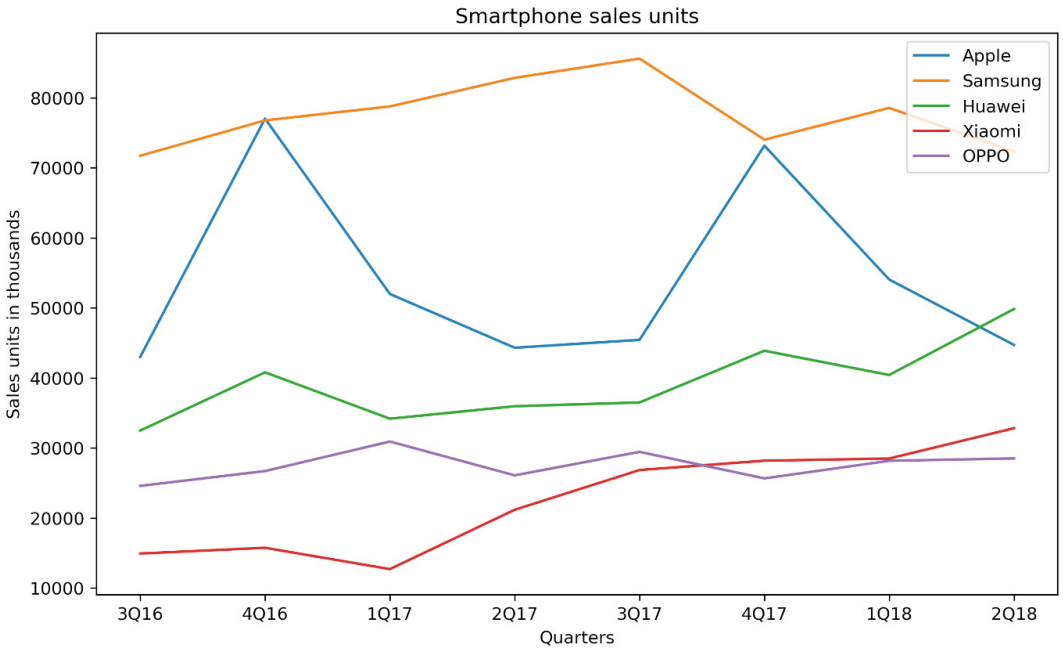

Figure 2.29: Line chart of smartphone sales units

Note:

The solution for this activity can be found on page 275.

Venn Diagram

Venn diagrams, also known as **set diagrams**, show all possible logical relations between a finite collections of different sets. Each set is represented by a circle. The circle size illustrates the importance of a group. The size of an overlap represents the intersection between multiple groups.

Uses:

- To show overlaps for different sets

Example:

- Visualizing the intersection of the following diagram shows a Venn diagram for students in two groups taking the same class in a semester:

Figure 2.30: Venn diagram to show students taking the same class

Figure 2.30: Venn diagram to show students taking the same class

Design practices:

- It is not recommended to use Venn diagrams if you have more than three groups. It would become difficult to understand.

Distribution Plots

Distribution plots give a deep insight into how your data is distributed. For a single variable, a histogram is well-suited. For multiple variables, you can either use a box plot or a violin plot. The violin plot visualizes the densities of your variables, whereas the box plot just

visualizes the median, the interquartile range, and the range for each variable.

Histogram

A **histogram** visualizes the distribution of a single numerical variable. Each bar represents the frequency for a certain interval. Histograms help get an estimate of statistical measures. You see where values are concentrated and you can easily detect outliers. You can either plot a histogram with absolute frequency values or alternatively normalize your histogram. If you want to compare distributions of multiple variables, you can use different colors for the bars.

Uses:

- Get insights into the underlying distribution for a dataset

Example:

The following diagram shows the distribution of the Intelligence Quotient (IQ) for a test group. The solid line indicates the mean and the dashed lines indicate the standard deviation:


Figure 2.31: Distribution of Intelligence Quotient (IQ) for a test group of a hundred adults

Figure 2.31: Distribution of Intelligence Quotient (IQ) for a test group of a hundred adults

Design practices:

- Try different numbers of bins, since the shape of the histogram can vary significantly.

Density Plot

A **density plot** shows the distribution of a numerical variable. It is a variation of a histogram that uses **kernel smoothing**, allowing for smoother distributions. An advantage they have over histograms is that density plots are better at determining the distribution shape, since the distribution shape for histograms heavily depends on the number of bins (data intervals).

Uses:

- You can compare the distribution of several variables by plotting the density on the same axis and using different colors.

Example:

The following diagram shows a basic density plot:


Figure 2.32: Density plot

Figure 2.32: Density plot

The following diagram shows a basic multi-density plot:

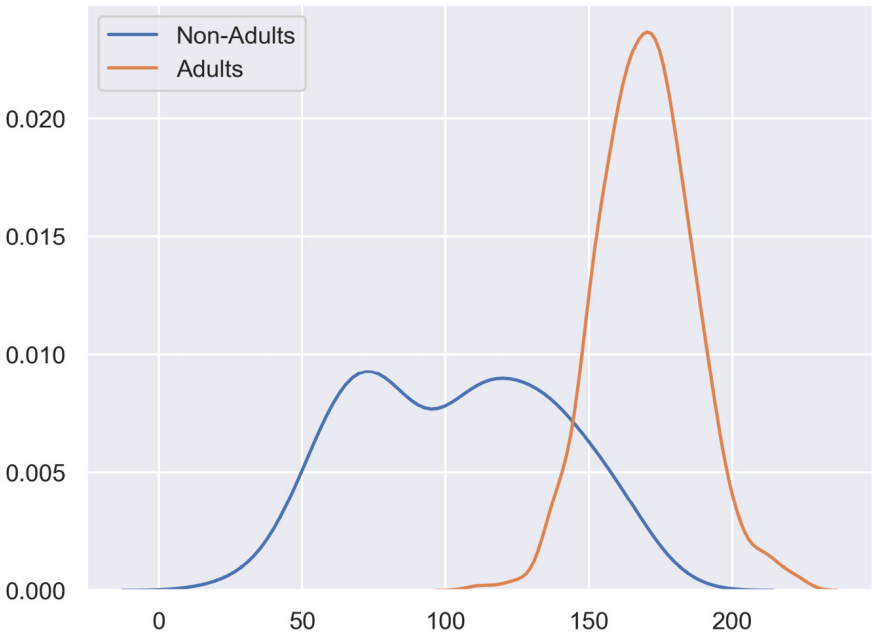

Figure 2.33: Multi-density plot

Design practices:

- Use contrasting colors for plotting the density of multiple variables.

Box Plot

The **box plot** shows multiple statistical measurements. The box extends from the lower to the upper quartile values of the data, thus allowing us to visualize the interquartile range. The horizontal line within the box denotes the median. The **whiskers** extending from the box show the range of the data. It is also an option to show data **outliers**, usually as circles or diamonds, past the end of the whiskers.

Uses:

- If you want to compare statistical measures for multiple variables or groups, you can simply plot multiple boxes next to one another.

Examples:

The following diagram shows a basic box plot:

Figure 2.34: Box plot showing single variable

Figure 2.34: Box plot showing single variable

The following diagram shows a basic box plot for multiple variables:

Figure 2.35: Box plot for multiple variables

Figure 2.35: Box plot for multiple variables

Violin Plot

Violin plots are a combination of box plots and density plots. Both the statistical measures and the distribution are visualized. The thick black bar in the center represents the interquartile range, the thin black line shows the 95% confidence interval, and the white dot shows

the median. On both sides of the center-line, the density is visualized.

Uses:

- If you want to compare statistical measures for multiple variables or groups, you can simply plot multiple violins next to one another.

Examples:

The following diagram shows a violin plot for a single variable and shows how students have performed in **Math**:

 Figure 2.37: Violin plot for a single variable (Math)

Figure 2.36: Violin plot for a single variable (math)

The following diagram shows a violin plot for two variables and shows the performance of students in **English** and **Math**:

 Figure 2.37: Violin plot for multiple variables (English and math)

Figure 2.37: Violin plot for multiple variables (English and math)

The following diagram shows a violin plot for a single variable divided into three groups and shows the performance of three divisions of students in **English**:

 Figure 2.38: Violin plot with multiple categories (three groups of students)

Figure 2.38: Violin plot with multiple categories (three groups of students)

Design practices:

- Scale the axes accordingly so that the distribution is clearly visible and not flat.

Activity 10: Frequency of Trains during Different Time Intervals

You are provided with a histogram that states the total number of trains arriving at different time intervals:

1. By looking at the following graph, can you identify the interval during which the most number of trains arrive?
2. How would the histogram change if the number of trains arriving between 4 and 6 pm were to be increased by 50?
 Figure 2.39: Frequency of trains during different time intervals

Figure 2.39: Frequency of trains during different time intervals

Note:

The solution for this activity can be found on page 276.

Geo Plots

Geological plots are a great way to visualize geospatial data. Choropleth maps can be used to compare quantitative values for different countries, states, and so on. If you want to show connections between different locations, connection maps are the way to go.

Dot Map

In a **dot map**, each dot represents a certain number of observations. Each dot has the same size and value (the number of observations each dot represents). The dots are not meant to be counted—they are only intended to give an impression of magnitude. The size and value

are important factors for the effectiveness and impression of the visualization. You can use different colors or symbols for the dots to show multiple categories or groups.

Uses:

- For the visualization of geospatial data

Example:

The following diagram shows a dot map where each dot represents a certain amount of bus stops throughout the world:

 Figure 2.40: Dot map showing bus stops worldwide

Figure 2.40: Dot map showing bus stops worldwide

Design practices:

- Do not show too many locations. You should still be able to see the map to get a feel for the actual location.
- Choose a dot size and value so that in dense areas, the dots start to blend. The dot map should give a good impression of the underlying spatial distribution.

Choropleth Map

In a **choropleth map**, each tile is colored to encode a variable. A tile represents a geographic region for, for example, counties and countries. Choropleth maps provide a good way to show how a variable varies across a geographic area. One thing to keep in mind for choropleth maps is that the human eye naturally gives more prominence to larger areas, so you might want to normalize your data by dividing the map area-wise.

Uses:

- For the visualization of geospatial data grouped into geological regions, for example, states, or countries

Example:

The following diagram shows a choropleth map of a weather forecast in the USA:

 Figure 2.41: Choropleth map showing weather forecast of USA

Figure 2.41: Choropleth map showing weather forecast of USA

Design practices:

- Use darker colors for higher values, as they are perceived as being higher in magnitude.
- Limit the color gradation, since the human eye is limited to how many colors it can easily distinguish between. Seven color gradations should be enough.

Connection Map

In a **connection map**, each line represents a certain number of connections between two locations. The link between the locations can be drawn with a straight or rounded line representing the shortest distance between them.

Each line has the same thickness and value (number of connections each line represents). The lines are not meant to be counted; they are only intended to give an impression of magnitude. The size and value of a connection line are important factors for the effectiveness and impression of the visualization.

You can use different colors for the lines to show multiple categories or groups, or you can use a colormap to encode the length of the connection.

Uses:

- For the visualization of connections

Examples:

The following diagram shows a connection map of flight connections around the world:


Figure 2.42: Connection map showing Flight connections around the world

Figure 2.42: Connection map showing Flight connections around the world

Design practices:

- Do not show too many connections. You should still see the map to get a feel of the actual locations of the start and end points.
- Choose a line thickness and value so that the lines start to blend in dense areas. The connection map should give a good impression of the underlying spatial distribution.

What Makes a Good Visualization?

There are multiple aspects to what makes a good visualization:

- Most importantly, a visualization should be self-explanatory and visually appealing. To make it self-explanatory, use a legend, descriptive labels for your x-axis and y-axis, and titles.
- A visualization should tell a story and be designed for your audience. Before creating your visualization, think about your target audience—create simple visualizations for a non-specialist audience and more technical detailed visualizations for a specialist audience. Think about a story to tell with your visualization so that your visualization leaves an impression on the audience.

Common design practices:

- Colors are more perceptible than symbols.
- To show additional variables on a 2D plot, use color, shape, and size.
- Keep it simple and don't overload the visualization with too much information.

Activity 11: Identifying the Ideal Visualization

The following visualizations are not ideal as they do not represent data well. Answer the following questions for each visualization:

1. What are the bad aspects of these visualizations?
2. How could we improve the visualizations? Sketch the right visualization for both scenarios.

The first visualization is supposed to illustrate the top 30 YouTubers according to the number of subscribers:


Figure 2.43: Pie chart showing Top 30 YouTubers

Figure 2.43: Pie chart showing Top 30 YouTubers

The second visualization is supposed to illustrate the number of people playing a certain game in a casino over two days:

Figure 2.44: Line chart displaying casino data for two days

Note:

The solution for this activity can be found on page 277.

Summary

In this chapter, the most important visualizations were discussed. The visualizations were categorized into comparison, relation, composition, distribution, and geological plots. For each plot, a description, practical examples, and design practices were given. Comparison plots, such as line charts, bar charts, and radar charts, are well-suited for comparing multiple variables or variables over time. Relation plots are perfectly suited to show relationships between variables. Scatter plots, bubble plots, which are an extension of scatter plots, correlograms, and heatmaps were considered. Composition plots are ideal if you think about something as a part of a whole. We first covered pie charts and continued with stacked bar charts, stacked area charts, and Venn diagrams. For distribution plots that give a deep insight into how your data is distributed, histograms, density plots, box plots, and violin plots were considered. Regarding geospatial data, we discussed dot maps, connection maps, and choropleth maps. Finally, some remarks were given on what makes a good visualization. In the next chapter, we will dive into Matplotlib and create our own visualizations. We will cover all the plots that we have discussed in this chapter.

Chapter 3

A Deep Dive into Matplotlib

Learning Objectives

By the end of this chapter, you will be able to:

- Describe the fundamentals of Matplotlib
- Create visualizations using the built-in plots that are provided by Matplotlib
- Customize your visualization plots
- Write mathematical expressions using TeX

In this chapter, we will learn how to customize you visualization using Matplotlib.

Introduction

Matplotlib is probably the most popular plotting library for Python. It is used for data science and machine learning visualizations all around the world. John Hunter began developing Matplotlib in 2003. It aimed to emulate the commands of the **MATLAB** software, which was the scientific standard back then. Several features such as the global style of MATLAB were introduced into Matplotlib to make the transition to Matplotlib easier for MATLAB users.

Before we start working with Matplotlib to create our first visualizations, we will understand and try to grasp the concepts behind the plots.

Overview of Plots in Matplotlib

Plots in Matplotlib have a hierarchical structure that nests Python objects to create a tree-like structure. Each plot is encapsulated in a `Figure` object. This `Figure` is the top-level container of the visualization. It can have multiple axes, which are basically individual plots inside this top-level container.

Going a level deeper, we again find Python objects that control axes, tick marks, legend, title, textboxes, the grid, and many other objects. All of these objects can be customized.

The two main components of a plot are as follows:

- **Figure**

  The Figure is an outermost container and is used as a canvas to draw on. It allows you to draw multiple plots within it. It not only holds the Axes object but also has the capability to configure the **Title**.

- **Axes**

  The Axes is an actual plot, or subplot, depending on whether you want to plot single or multiple visualizations. Its sub-objects include the x and y axis, spines, and legends.

Observing this design on a higher level, we can see that this hierarchical structure allows us to create a complex and customizable visualization.

When looking at the "anatomy" of a Figure, which is shown in the following diagram, we get an idea about the complexity of an insightful visualization. Matplotlib gives us the ability not only to simply display data, but also design the whole **Figure** around it, by adjusting the

Grid, **x** and **y ticks**, **tick labels**, and the **Legend**. This implies that we can modify every single bit of a plot, starting from the Title and Legend, right down to even the major and minor ticks on the spines to make it more expressive:

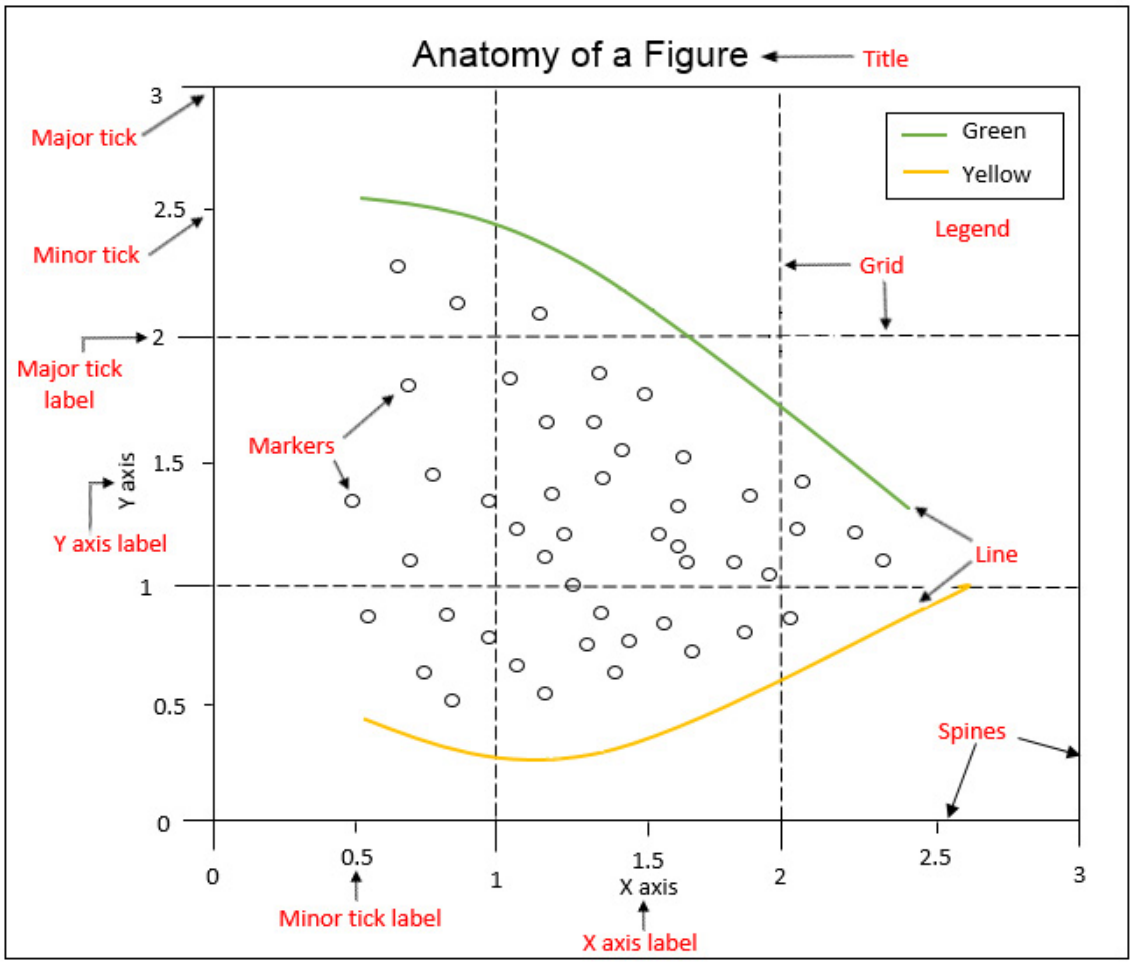

Figure 3.1: Anatomy of a Matplotlib Figure

Taking a deeper look into the anatomy of a Figure object, we can observe the following components:

- **Spines**: Lines connecting the axis tick marks
- **Title**: Text label of the whole Figure object
- **Legend**: They describe the content of the plot
- **Grid**: Vertical and horizontal lines used as an extension of the tick marks
- **X/Y axis label**: Text label for the X/Y axis below the spines
- **Minor tick**: Small value indicators between the major tick marks
- **Minor tick label**: Text label that will be displayed at the minor ticks
- **Major tick**: Major value indicators on the spines
- **Major tick label**: Text label that will be displayed at the major ticks

- **Line**: Plotting type that connects data points with a line
- **Markers**: Plotting type that plots every data point with a defined marker

In this book, we will focus on Matplotlib's submodule, **pyplot**, which provides MATLAB-like plotting.

Pyplot Basics

pyplot contains a simpler interface for creating visualizations, which allows the users to plot the data without explicitly configuring the **Figure** and **Axes** themselves. They are implicitly and automatically configured to achieve the desired output. It is handy to use the alias **`plt`** to reference the imported submodule, as follows:

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

The following sections describe some of the common operations that are performed when using pyplot.

Creating Figures

We use **`plt.figure()`** to create a new **Figure**. This function returns a Figure instance, but it is also passed to the backend. Every Figure-related command that follows is applied to the current Figure and does not need to know the Figure instance.

By default, the Figure has a width of 6.4 inches and a height of 4.8 inches with a dpi of 100. To change the default values of the Figure, we can use the parameters **`figsize`** and **`dpi`**.

The following code snippet shows how we can manipulate a Figure:

```
plt.figure(figsize=(10, 5)) #To change the width and the height
plt.figure(dpi=300) #To change the dpi
```

Closing Figures

Figures that are not used anymore should be closed by explicitly calling **`plt.close()`**, which also cleans up memory efficiently.

If nothing is specified, the current Figure will be closed. To close a specific Figure, you can either provide a reference to a Figure instance or provide the Figure number. To find the **number** of a figure object, we can make use of the **`number`** attribute, like so:

```
plt.gcf().number
```

By using **`plt.close('all')`**, all Figures will be closed. The following example shows how a Figure can be created and closed:

```
plt.figure(num=10) #Create Figure with Figure number 10
plt.close(10) #Close Figure with Figure number 10
```

Format Strings

Before we actually plot something, let's quickly discuss **format strings**. They are a neat way to specify **colors**, **marker types**, and **line styles**. A format string is specified as "**`[color][marker][line]`**", where each item is optional. If the **`color`** is the only argument of the format string, you can use any **`matplotlib.colors`**. Matplotlib recognizes the following formats, among others:

- RGB or RGBA float tuples (for example, (0.2, 0.4, 0.3) or (0.2, 0.4, 0.3, 0.5))
- RGB or RGBA hex strings (for example, '#0F0F0F' or '#0F0F0F0F')

The following diagram is an example of how a color can be represented in one particular format:

Format	Colors
'b'	blue
'r'	red
'g'	green
'm'	magenta
'c'	cyan
'b'	black
'w'	white
'y'	yellow

Figure 3.2 Color specified in string format

All the available marker options are illustrated in the following diagram:

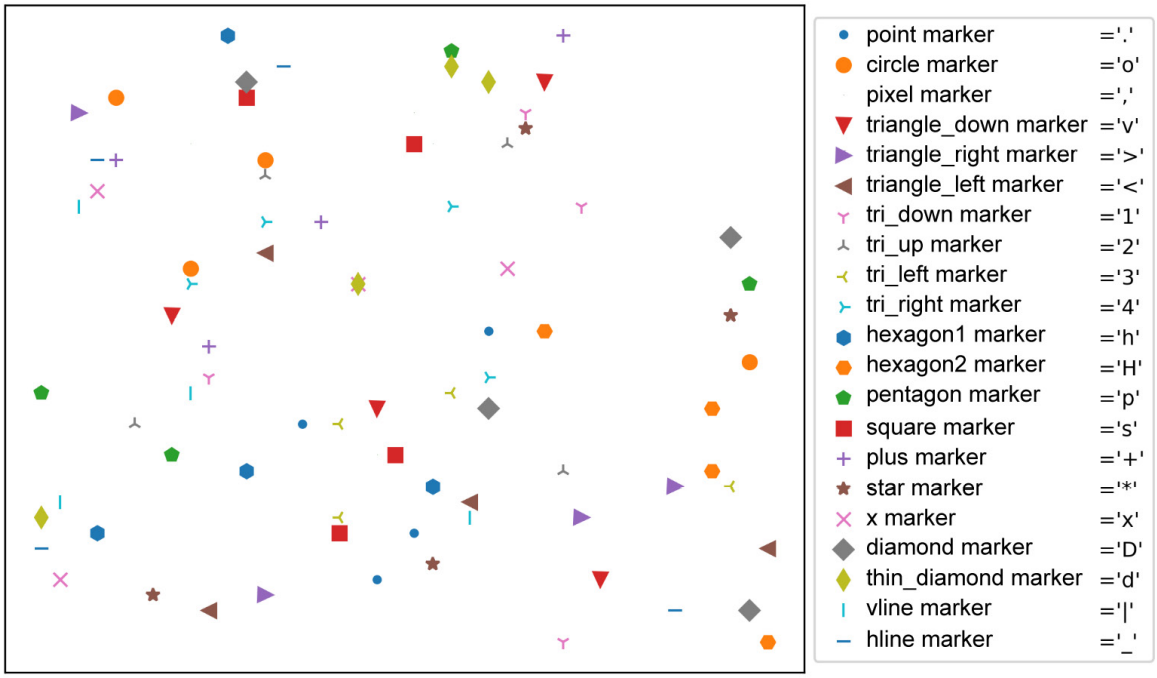

Figure 3.3: Markers in format strings

All the available line styles are illustrated in the following diagram:

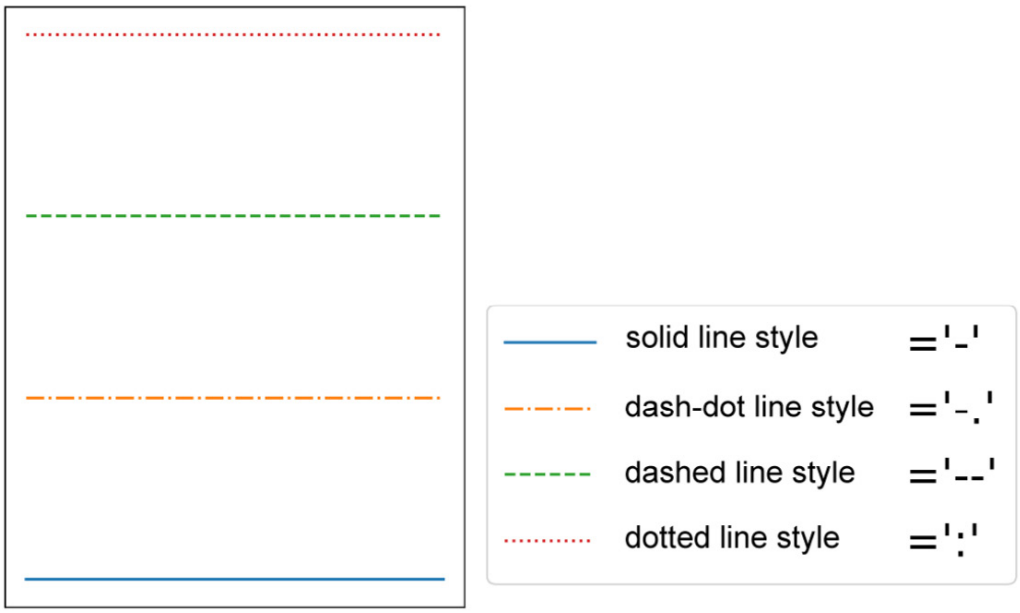

Figure 3.4: Line styles

Plotting

With **`plt.plot([x], y, [fmt])`**, you can plot data points as lines and/or markers. The function returns a list of Line2D objects representing the plotted data. By default, if you do not provide a format string, the data points will be connected with straight, solid lines. **`plt.plot([0, 1, 2, 3], [2, 4, 6, 8])`** produces a plot, as shown in the following diagram. Since **`x`** is optional and default values are **`[0, …, N-1]`**, **`plt.plot([2, 4, 6, 8])`** results in the same plot:

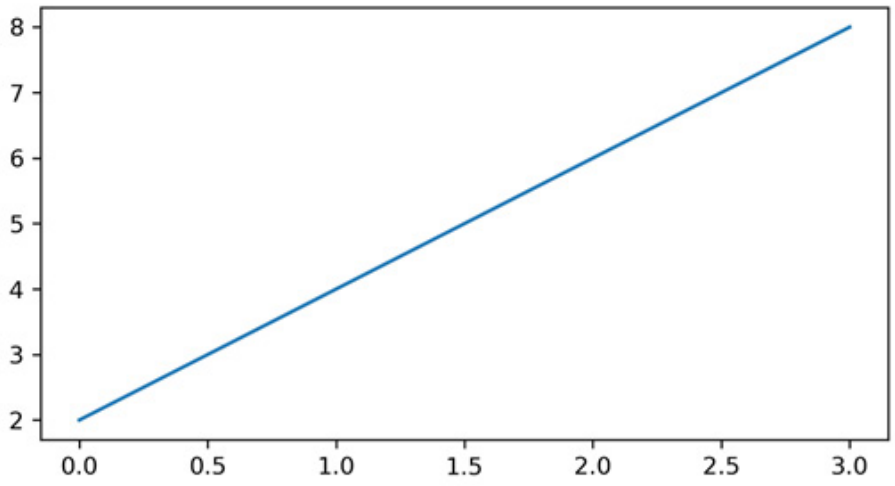

Figure 3.5: Plotting data points as a line

If you want to plot markers instead of lines, you can just specify a format string with any marker type. For example, **`plt.plot([0, 1, 2, 3], [2, 4, 6, 8], 'o')`** displays data points as circles, as shown in the following diagram:

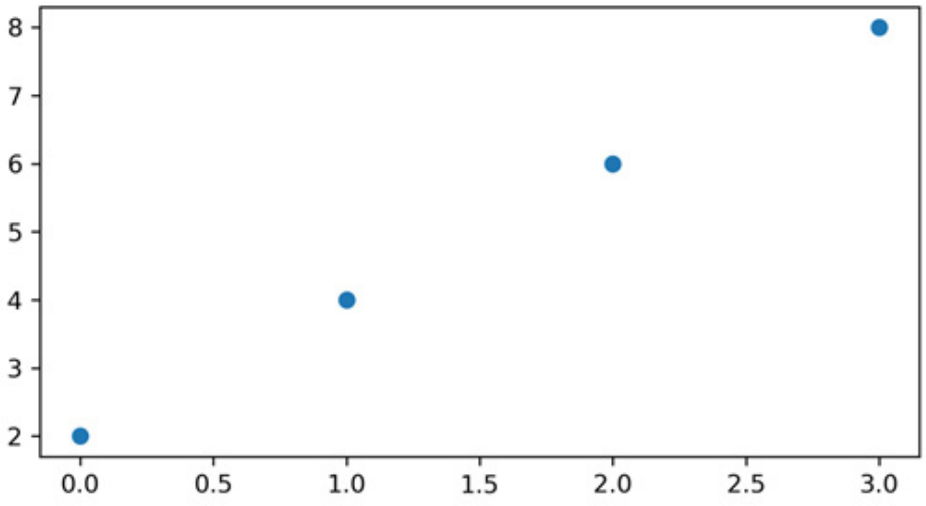

Figure 3.6: Plotting data points with markers (circles)

To plot multiple data pairs, the syntax **`plt.plot([x], y, [fmt], [x], y2, [fmt2], …)`** can be used. **`plt.plot([2, 4, 6, 8], 'o', [1, 5, 9, 13], 's')`** results in the following diagram. Similarly, you can use **`plt.plot`** multiple times, since we are working on the same Figure and Axes:

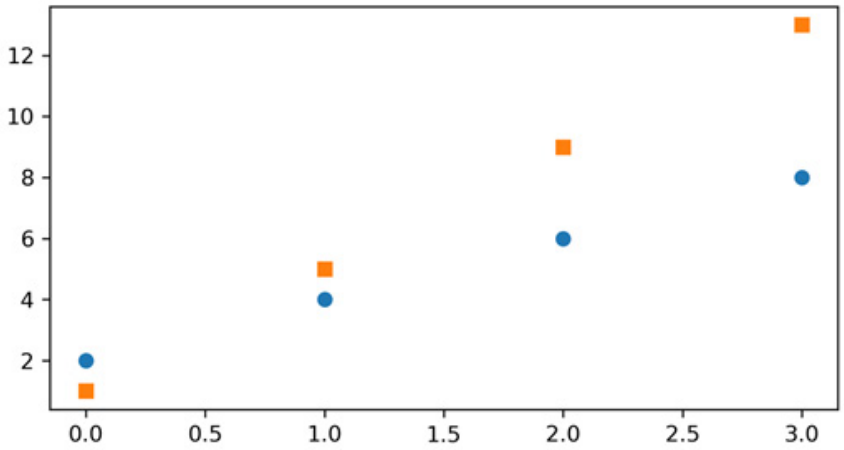

Figure 3.7: Plotting data points with multiple markers

Any **Line2D** properties can be used instead of format strings to further customize the plot. For example, the following code snippet shows how we can additionally specify the **`linewidth`** and the **`markersize`**:

```
plt.plot([2, 4, 6, 8], color='blue', marker='o', linestyle='dashed', linewidth=2,
markersize=12)
```

Plotting Using pandas DataFrames

It is pretty straightforward to use **`pandas.DataFrame`** as a data source. Instead of providing x and y values, you can provide the **`pandas.DataFrame`** in the data parameter and give keys for **`x`** and **`y`**, as follows:

```
plt.plot('x_key', 'y_key', data=df)
```

Displaying Figures

`plt.show()` is used to display a Figure or multiple Figures. To display Figures within a Jupyter Notebook, simply set the **`%matplotlib inline`** command in the beginning of the code.

Saving Figures

`plt.savefig(fname)` saves the current Figure. There are some useful optional parameters you can specify, such as **`dpi`**, **`format`**, or **`transparent`**. The following code snippet gives an example of how you can save a Figure:

```
plt.figure()
plt.plot([1, 2, 4, 5], [1, 3, 4, 3], '-o')
plt.savefig('lineplot.png', dpi=300, bbox_inches='tight')
#bbox_inches='tight' removes the outer white margins
```

Note

All exercises and activities will be developed in the Jupyter Notebook. Please download the GitHub repository with all the prepared templates from https://github.com/TrainingByPackt/Data-Visualization-with-Python.

Exercise 3: Creating a Simple Visualization

In this exercise, we will create our first simple plot using Matplotlib:

1. Open the Jupyter Notebook **`exercise03.ipynb`** from the **`Lesson03`** folder to implement this exercise.

   Navigate to the path of this file and type in the following at the command line: **`jupyter-lab.`**

2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   ```
   import numpy as np
   import matplotlib.pyplot as plt
   %matplotlib inline
   ```

3. Explicitly create a Figure and set the dpi to 200:

   ```
   plt.figure(dpi=200)
   ```

4. Plot the following data pairs **`(x, y)`** as circles, which are connected via line segments: **`(1, 1)`**, **`(2, 3)`**, **`(4, 4)`**, **`(5, 3)`** . Then, visualize the plot:

   ```
   plt.plot([1, 2, 4, 5], [1, 3, 4, 3], '-o')
   plt.show()
   ```

   Your output should look similar to this:

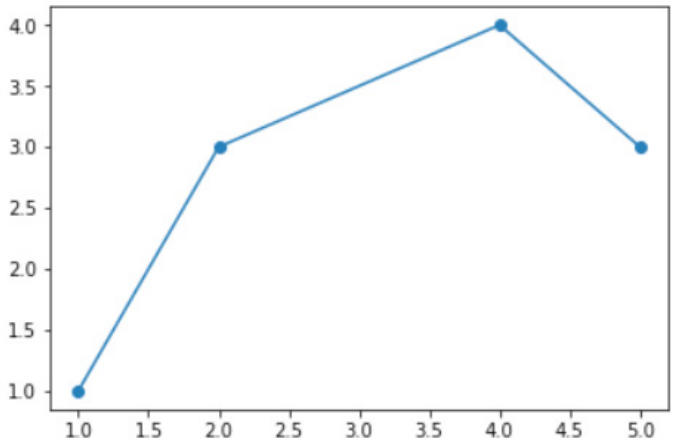

Figure 3.8: A simple visualization that was created with the help of given data pairs and is connected via line segments

5. Save the plot using the **`plt.savefig()`** method. Here, we can either provide a filename within the method or specify the full path:

```
plt.savefig(exercise03.png);
```

Basic Text and Legend Functions

All of the functions we have discussed in this topic, except for the legend, create and return a **`matplotlib.text.Text()`** instance. We are mentioning it here so that you know that all of the discussed properties can be used for the other functions as well. All text functions are illustrated in Figure 3.9.

Labels

Matplotlib provides a few **label** functions that we can use for setting labels to the x and y axes. The **`plt.xlabel()`** and **`plt.ylabel()`** functions are used to set the label for the current axes. The **`set_xlabel()`** and **`set_ylabel()`** functions are used to set the label for specified axes.

Example:

```
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
```

Titles

A **title** describes a particular chart/graph. The titles are placed above the axes in the center, left edge, or right edge. There are two options for titles – you can either set the **Figure title** or the title of an **Axes**. The **`suptitle()`** function sets the title for current and specified Figure. The **`title()`** function helps in setting the title for the current and specified axes.

Example:

```
fig = plt.figure()
fig.suptitle('Suptitle', fontsize=10, fontweight='bold')
```

This creates a bold figure title with a text suptitle and a font size of 10.

Text

There are two options for **text** – you can either add text to a Figure or text to an Axes. The **`figtext(x, y, text)`** and **`text(x, y, text)`** functions add a text at location **`x`**, or **`y`** for a figure.

Example:

```
ax.text(4, 6, 'Text in Data Coords', bbox={'facecolor': 'yellow', 'alpha':0.5, 'pad':10})
```

This creates a yellow textbox with the text "Text in Data Coords".

Annotations

Compared to text that is placed at an arbitrary position on the Axes, **annotations** are used to annotate some features of the plot. In annotation, there are two locations to consider: the annotated location **`xy`** and the location of the annotation, text **`xytext`**. It is useful to specify the parameter **`arrowprops`**, which results in an arrow pointing to the annotated location.

Example:

```
ax.annotate('Example of Annotate', xy=(4,2), xytext=(8,4),
arrowprops=dict(facecolor='green', shrink=0.05))
```

This creates a green arrow pointing to the data coordinates (4, 2) with the text "Example of Annotate" at data coordinates (8, 4):

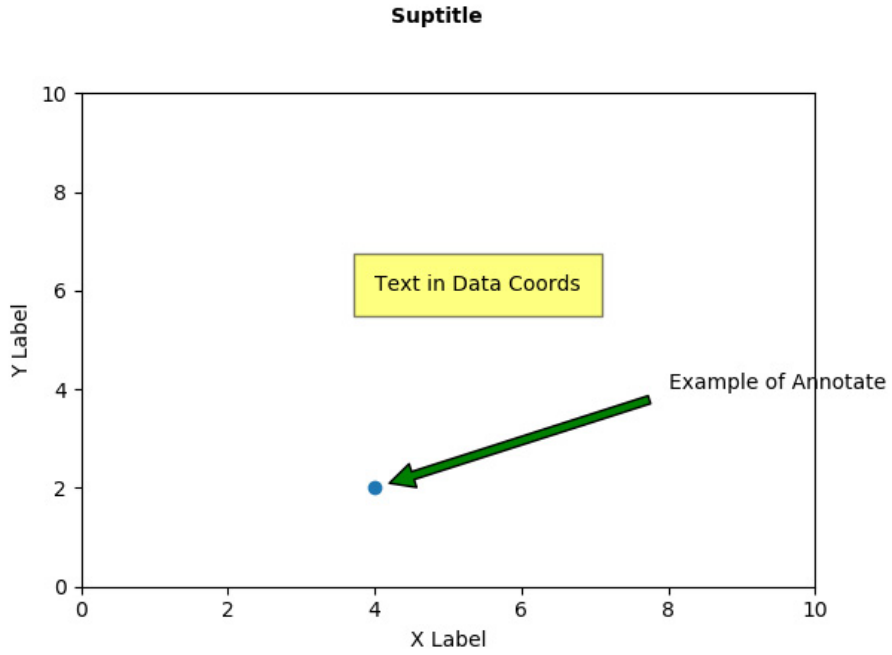

Figure 3.9: Implementation of Text commands

Legends

For adding a **legend** to your Axes, we have to specify the **`label`** parameter at the time of artist creation. Calling **`plt.legend()`** for the current Axes or **`Axes.legend()`** for a specific Axes will add the legend. The **`loc`** parameter specifies the location of the legend.

Example:

```
…
plt.plot([1, 2, 3], label='Label 1')
plt.plot([2, 4, 3], label='Label 2')
plt.legend()
…
```

This example is illustrated in the following diagram:

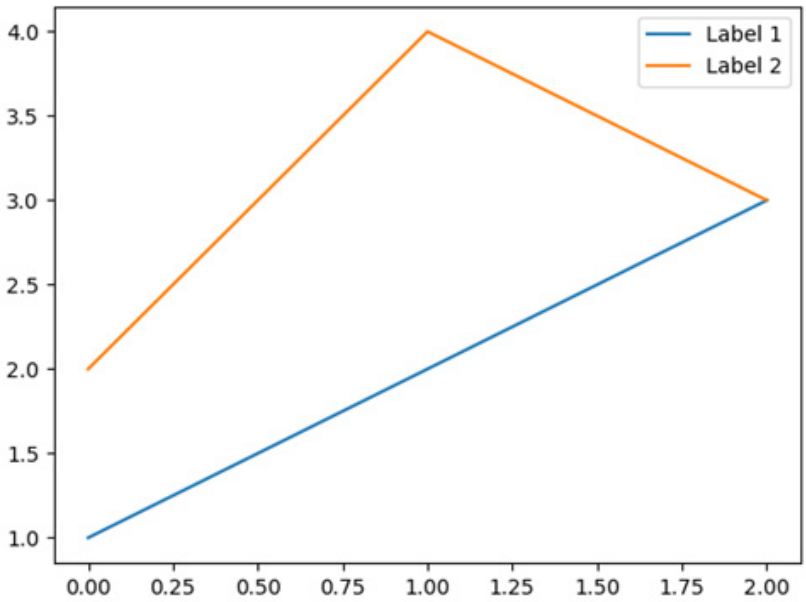

Figure 3.10: Legend example

Activity 12: Visualizing Stock Trends by Using a Line Plot

In this activity, we will create a line plot to show stock trends. Let's look at the following scenario: you are interested in investing in stocks. You downloaded the stock prices for the "big five": Amazon, Google, Apple, Facebook, and Microsoft:

1. Use pandas to read the data located in the subfolder **`data`**.
2. Use Matplotlib to create a line chart visualizing the closing prices for the past five years (whole data sequence) for all five companies. Add labels, titles, and a legend to make the visualization self-explanatory. Use **`plt.grid()`** to add a grid to your plot.
3. After executing the preceding steps, the expected output should be as follows:

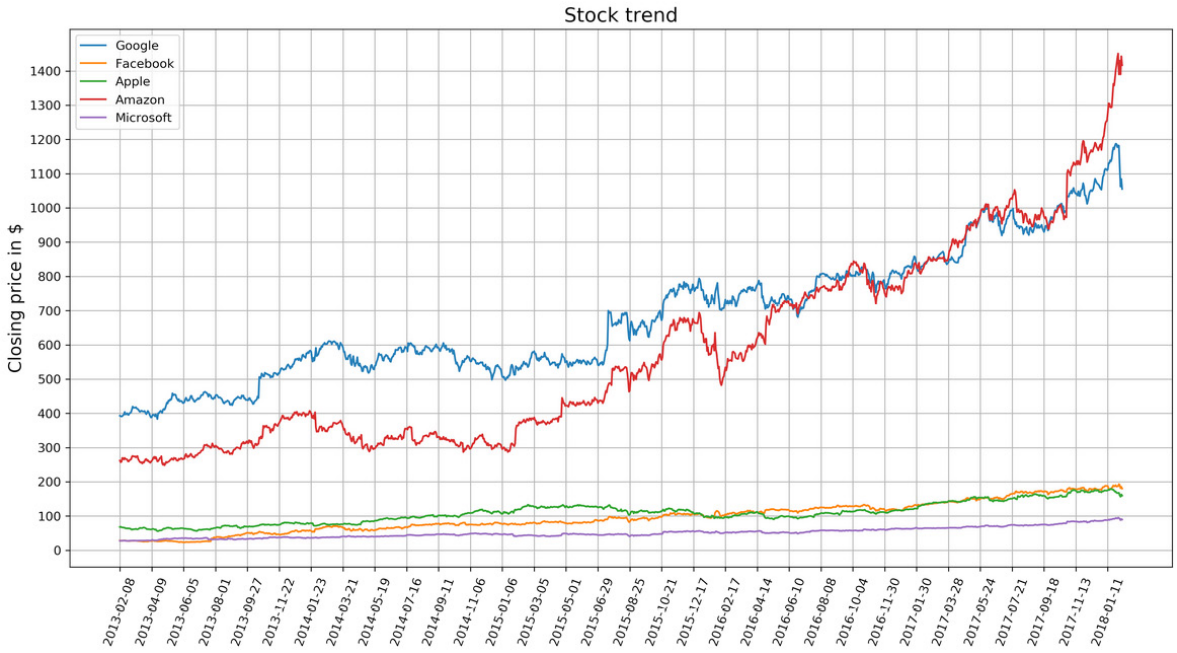

Figure 3.11: Visualization of stock trends of five companies

Note:

The solution for this activity can be found on page 279.

Basic Plots

In this section, we are going to go through the different types of basic plots.

Bar Chart

`plt.bar(x, height, [width])` creates a vertical bar plot. For horizontal bars, use the **`plt.barh()`** function.

Important parameters:

- **`x`**: Specifies the x coordinates of the bars
- **`height`**: Specifies the height of the bars
- **`width`** (optional): Specifies the width of all bars; the default is 0.8

Example:

```
plt.bar(['A', 'B', 'C', 'D'], [20, 25, 40, 10])
```

The preceding code creates a bar plot, as shown in the following diagram:

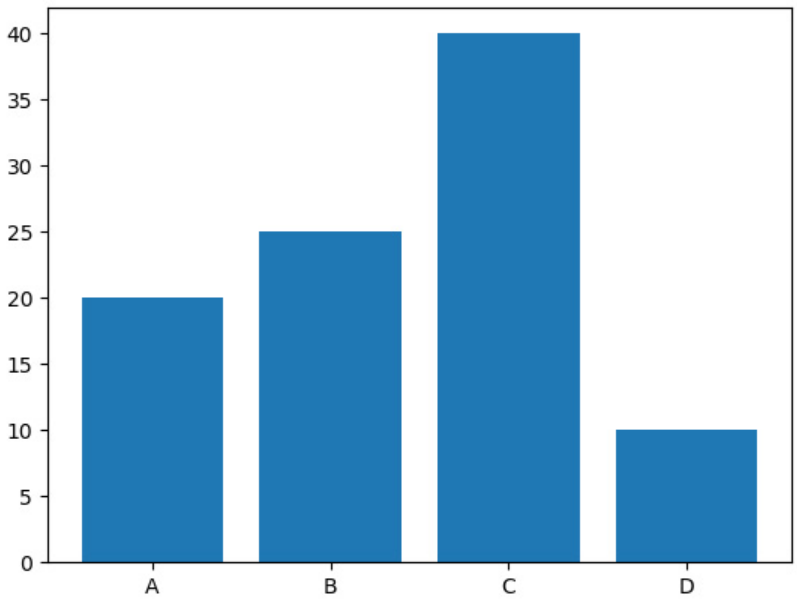

Figure 3.12: A simple bar chart

If you want to have subcategories, you have to use the **`plt.bar()`** function multiple times with shifted x-coordinates. This is done in the following example and illustrated in the diagram that follows. The **`arange()`** function is a method in the **NumPy** package that returns evenly spaced values within a given interval. The **`gca()`** function helps in getting the instance of current axes on any current figure. The **`set_xticklabels()`** function is used to set the x-tick labels with the list of given string labels.

Example:

```
…
labels = ['A', 'B', 'C', 'D']
x = np.arange(len(labels))
width = 0.4
plt.bar(x – width / 2, [20, 25, 40, 10], width=width)
plt.bar(x – width / 2, [30, 15, 30, 20], width=width)
# Ticks and tick labels must be set manually
plt.ticks(x)
ax = plt.gca()
ax.set_xticklabels(labels)
…
```

The preceding code creates a bar chart with subcategories, as shown in the following diagram:

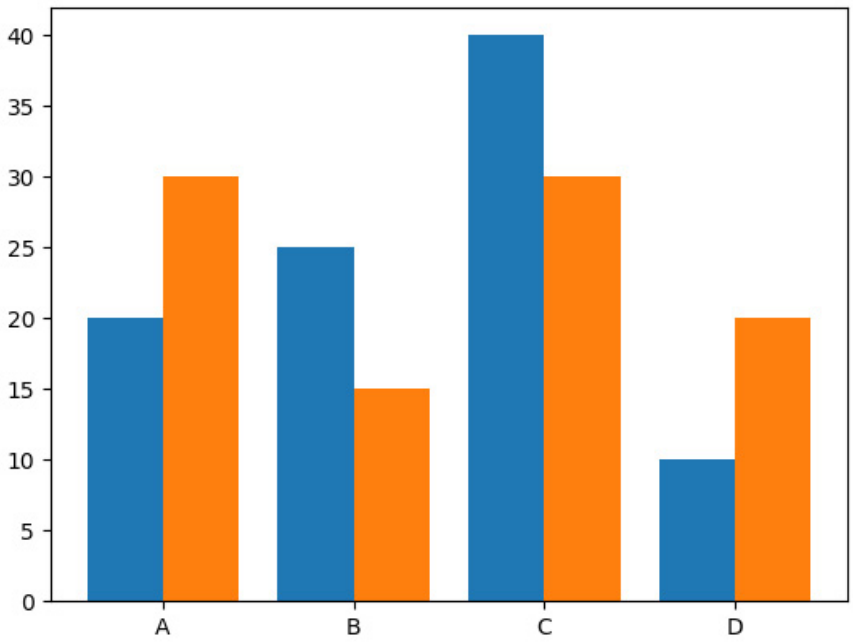

Figure 3.13: Bar chart with subcategories

Activity 13: Creating a Bar Plot for Movie Comparison

In this activity, we will use a bar plot to compare movie scores. You are given five movies with scores from Rotten Tomatoes. The Tomatometer is the percentage of approved Tomatometer critics who have given a positive review for the movie. The Audience Score is the percentage of users who have given a score of 3.5 or higher out of 5. Compare these two scores among the five movies:

1. Use pandas to read the data located in the subfolder data.
2. Use Matplotlib to create a visually appealing bar plot comparing the two scores for all five movies.
3. Use the movie titles as labels for the x-axis. Use percentages in an interval of 20 for the y-axis and minor ticks in an interval of 5. Add a legend and a suitable title to the plot.
4. After executing the preceding steps, the expected output should be as follows:

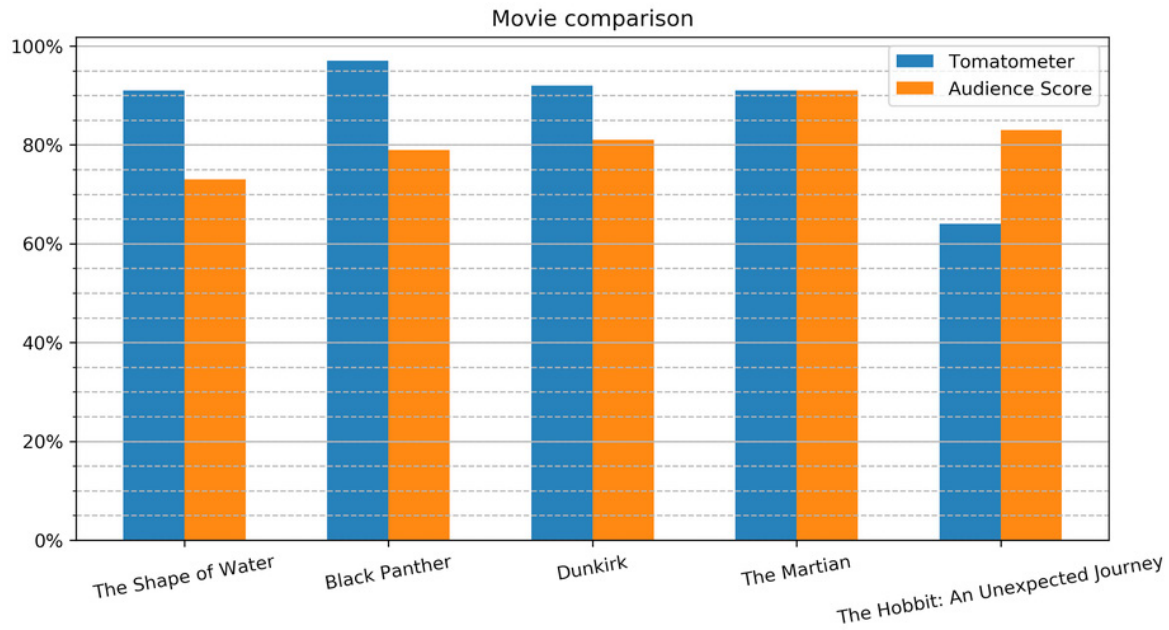

Figure 3.14: Bar plot comparing scores of five movies

Note:

The solution for this activity can be found on page 280.

Pie Chart

The **`plt.pie(x, [explode], [labels], [autopct])`** function creates a pie chart.

Important parameters:

- **`x`**: Specifies the slice sizes.
- **`explode`** (optional): Specifies the fraction of the radius offset for each slice. The explode-array must have the same length as the x-array.
- **`labels`** (optional): Specifies the labels for each slice.
- **`autopct`** (optional): Shows percentages inside the slices according to the specified format string. Example: '%1.1f%%'.

Example:

```
…
plt.pie([0.4, 0.3, 0.2, 0.1], explode=(0.1, 0, 0, 0), labels=['A', 'B', 'C', 'D'])
…
```

The result of the preceding code is visualized in following diagram:

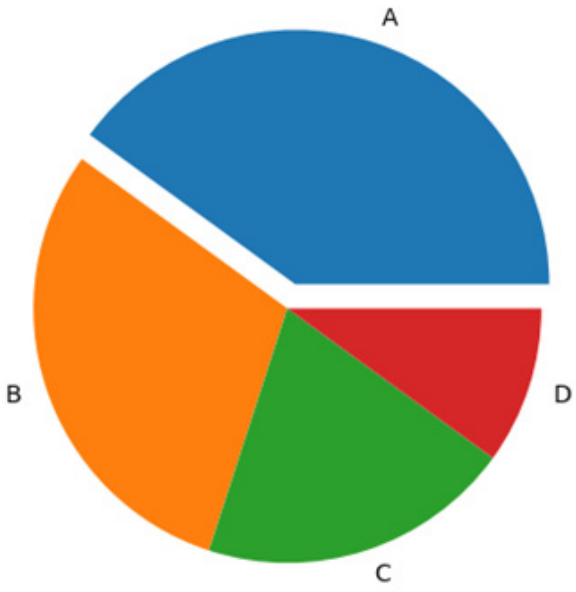

Figure 3.15: Basic pie chart

Exercise 4: Creating a Pie Chart for Water Usage

In this exercise, we will use a pie chart to visualize water usage:

1. Open the Jupyter Notebook **`exercise04.ipynb`** from the **`Lesson03`** folder to implement this exercise.

   Navigate to the path of this file and type in the following at the command line: **`jupyter-lab.`**

2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   ```
   # Import statements
   import pandas as pd
   import matplotlib.pyplot as plt
   %matplotlib inline
   ```

3. Use pandas to read the data located in the subfolder data:

   ```
   # Load dataset
   data = pd.read_csv('./data/water_usage.csv')
   ```

4. Use a pie chart to visualize the water usage. Highlight one usage of your choice using the explode parameter. Show the percentages for each slice and add a title:

   ```
   # Create figure
   plt.figure(figsize=(8, 8), dpi=300)
   # Create pie plot
   plt.pie('Percentage', explode=(0, 0, 0.1, 0, 0, 0), labels='Usage', data=data,
   autopct='%.0f%%')
   # Add title
   ```

```
plt.title('Water usage')
# Show plot
plt.show()
```

The following Figure shows output of the preceding code:

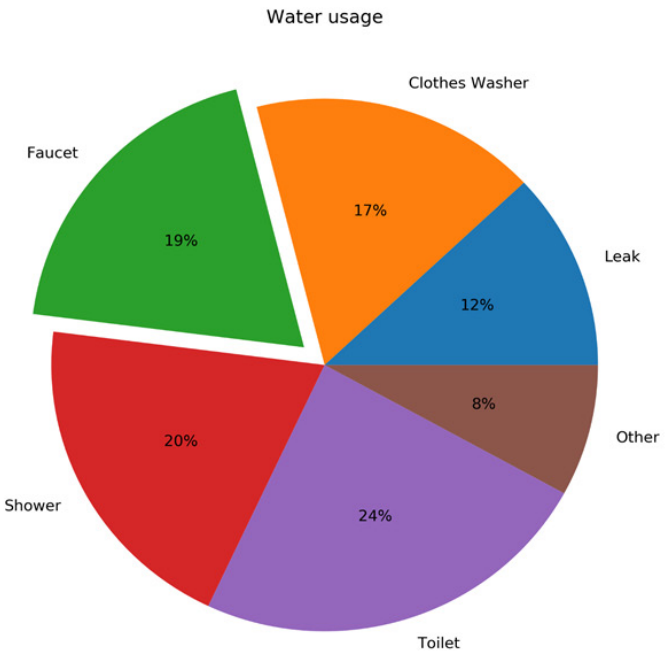

Figure 3.16: Pie chart for water usage

Stacked Bar Chart

A **stacked bar chart** uses the same **`plt.bar`** function as bar charts. For each stacked bar, the **`plt.bar`** function must be called and the **`bottom`** parameter must be specified starting with the second stacked bar. This will become clear with the following example:

```
…
plt.bar(x, bars1)
plt.bar(x, bars2, bottom=bars1)
plt.bar(x, bars3, bottom=np.add(bars1, bars2))
…
```

The result of the preceding code is visualized in the following diagram:

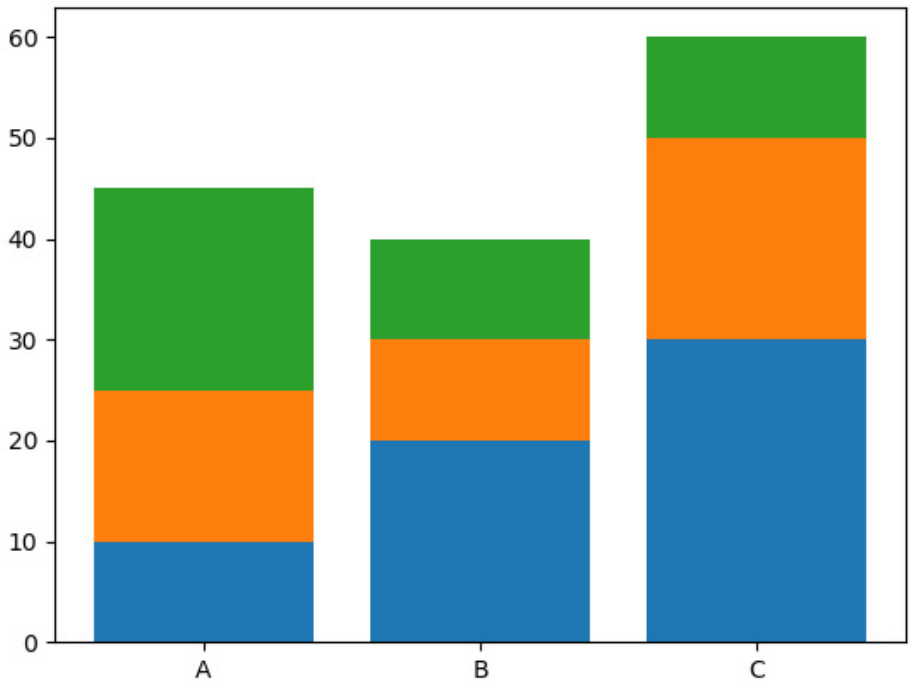

Figure 3.17: Stacked bar chart

Activity 14: Creating a Stacked Bar Plot to Visualize Restaurant Performance

In this activity, we will use a stacked bar plot to visualize the performance of a restaurant. Let's look at the following scenario: you are the owner of a restaurant and, due to a new law, you have to introduce a No Smoking Day. To make as few losses as possible, you want to visualize how many sales are made every day, categorized by smokers and non-smokers:

1. Use the given dataset and create a matrix where the elements contain the sum of the total bills for each day and smokers/non-smokers.
2. Create a stacked bar plot, stacking the summed total bills separated by smoker and non-smoker for each day. Add a legend, labels, and a title.
3. After executing the preceding steps, the expected output should be as follows:

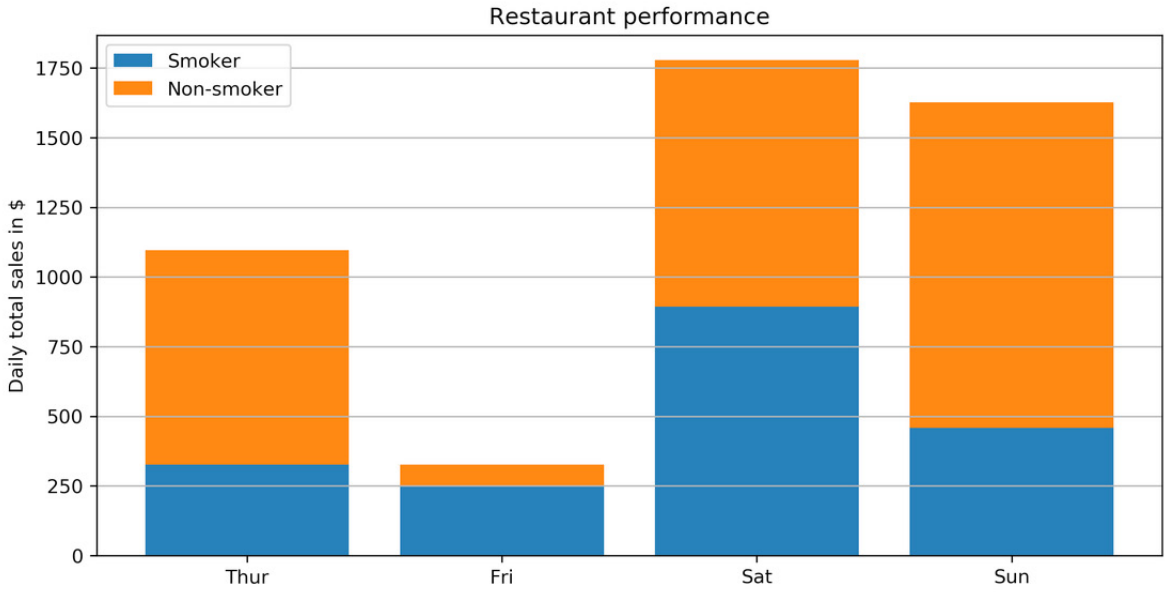

Figure 3.18: Stacked bar chart showing performance of restaurant on different days

Note:

The solution for this activity can be found on page 282.

Stacked Area Chart

`plt.stackplot(x, y)` creates a stacked area plot.

Important parameters:

- **`x`**: Specifies the x-values of the data series.
- **`y`**: Specifies the y-values of the data series. For multiple series, either as a 2d array, or any number of 1D arrays, call the following function: **`plt.stackplot(x, y1, y2, y3, …).`**
- **`labels`** (Optional): Specifies the labels as a list or tuple for each data series.

Example:

```
…
plt.stackplot([1, 2, 3, 4], [2, 4, 5, 8], [1, 5, 4, 2])
…
```

The result of the preceding code is shown in the following diagram:

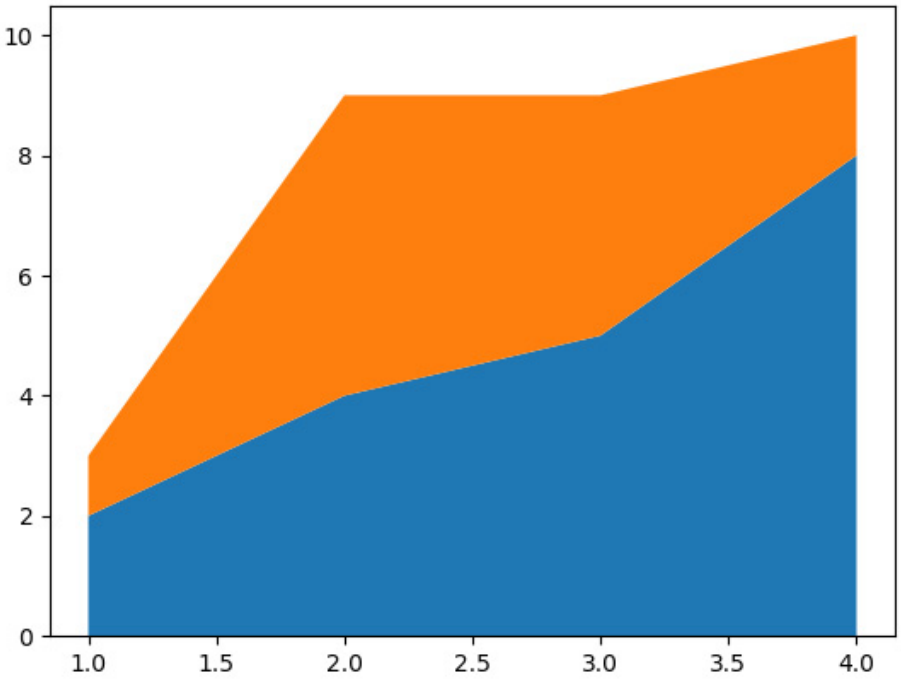

Figure 3.19: Stacked area chart

Activity 15: Comparing Smartphone Sales Units Using a Stacked Area Chart

In this activity, we will compare smartphone sales units using a stacked area chart. Let's look at the following scenario: you want to invest in one of the biggest five smartphone manufacturers. Looking at the quarterly sales units as part of a whole may be a good indicator for which company to invest in:

1. Use pandas to read the data located in the subfolder data.
2. Create a visually appealing stacked area chart. Add a legend, labels, and a title.
3. After executing the preceding steps, the expected output should be as follows:

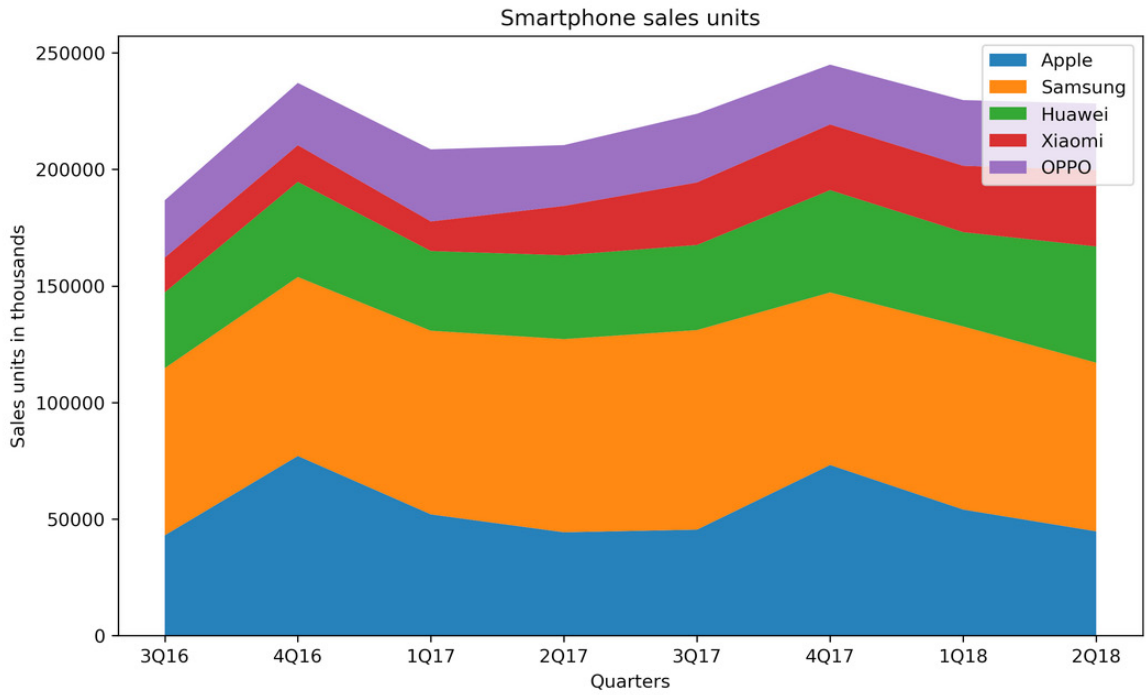

Figure 3.20: Stacked area chart comparing sales units of different smartphone manufacturers

Note:

The solution for this activity can be found on page 283.

Histogram

`plt.hist(x)` creates a histogram.

Important parameters:

- **`x`**: Specifies the input values
- **`bins`**: (optional): Either specifies the number of bins as an integer or specifies the bin edges as a list
- **`range`**: (optional): Specifies the lower and upper range of the bins as a tuple
- **`density`**: (optional): If true, the histogram represents a probability density

Example:

```
...
plt.hist(x, bins=30, density=True)
...
```

The result of the preceding code is shown in the following diagram:

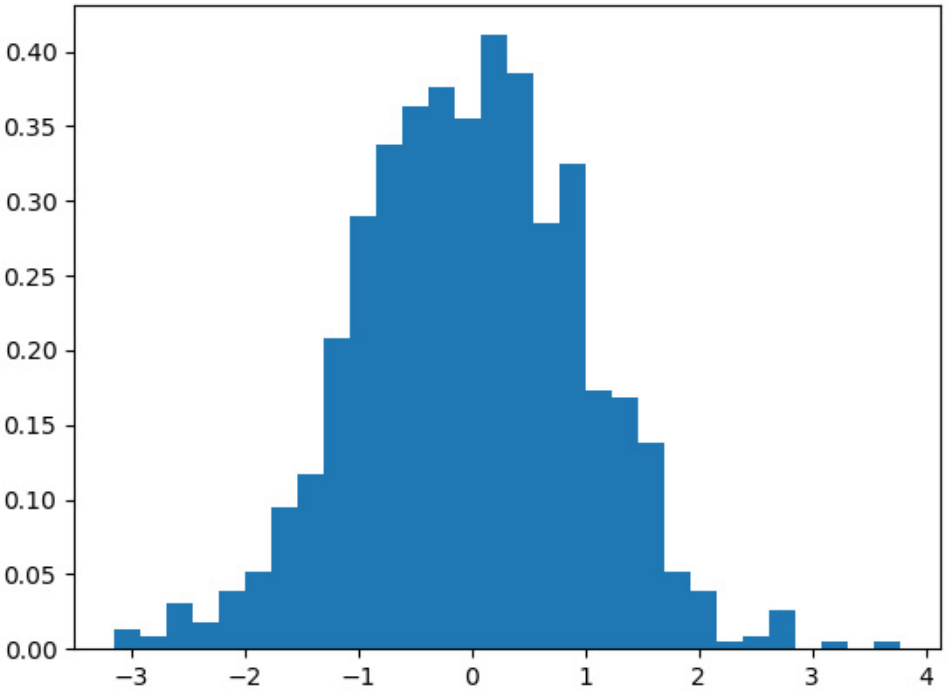

Figure 3.21: Histogram

`plt.hist2d(x, y)` creates a 2D histogram. An example of a 2D historgram is shown in the following diagram:

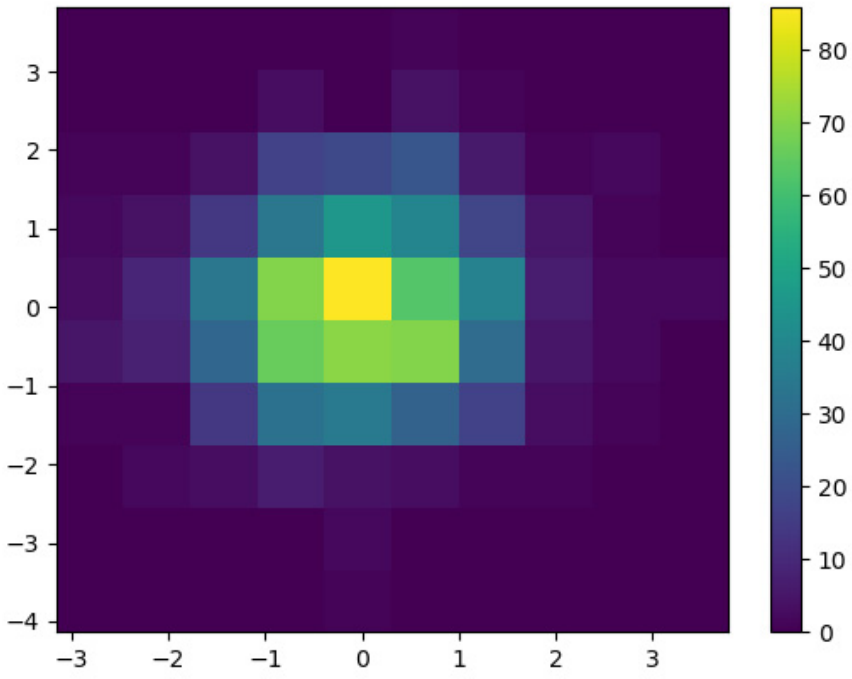

Figure 3.22: 2D histogram with color bar

Box Plot

`plt.boxplot(x)` creates a box plot.

Important parameters:

- **`x`**: Specifies the input data. It specifies either a 1D array for a single box or a sequence of arrays for multiple boxes.
- **`notch`**: Optional: If true, notches will be added to the plot to indicate the confidence interval around the median.
- **`labels`**: Optional: Specifies the labels as a sequence.
- **`showfliers`**: Optional: By default, it is true, and outliers are plotted beyond the caps.
- **`showmeans`**: Optional: If true, arithmetic means are shown.

Example:

```
…
plt.boxplot([x1, x2], labels=['A', 'B'])
…
```

The result of the preceding code is shown in the following diagram:

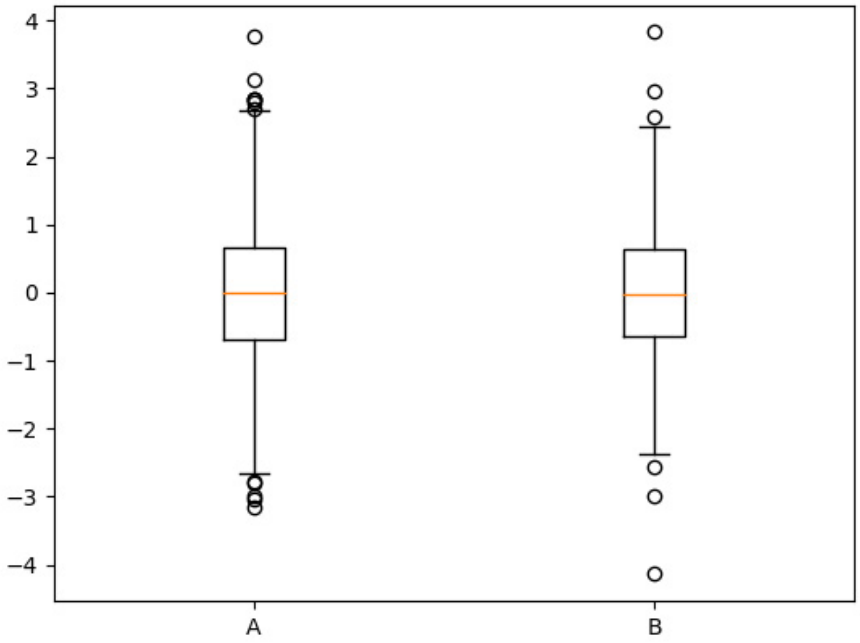

Figure 3.23: Box plot

Activity 16: Using a Histogram and a Box Plot to Visualize the Intelligence Quotient

In this activity, we will visualize the intelligence quotient (IQ) using a histogram and box plots.

Note

`plt.axvline(x, [color=…], [linestyle=…])` *draws a vertical line at position* ***`x`****.*

1. Plot a histogram with 10 bins for the given IQ scores. IQ scores are normally distributed with a mean of 100 and a standard deviation of 15. Visualize the mean as a vertical solid red line, and the standard deviation using dashed vertical lines. Add labels and a title.

2. Create a box plot to visualize the same IQ scores. Add labels and a title.

3. Create a box plot for each of the IQ scores of the different test groups. Add labels and a title.

4. The expected output for step 1 is as follows:

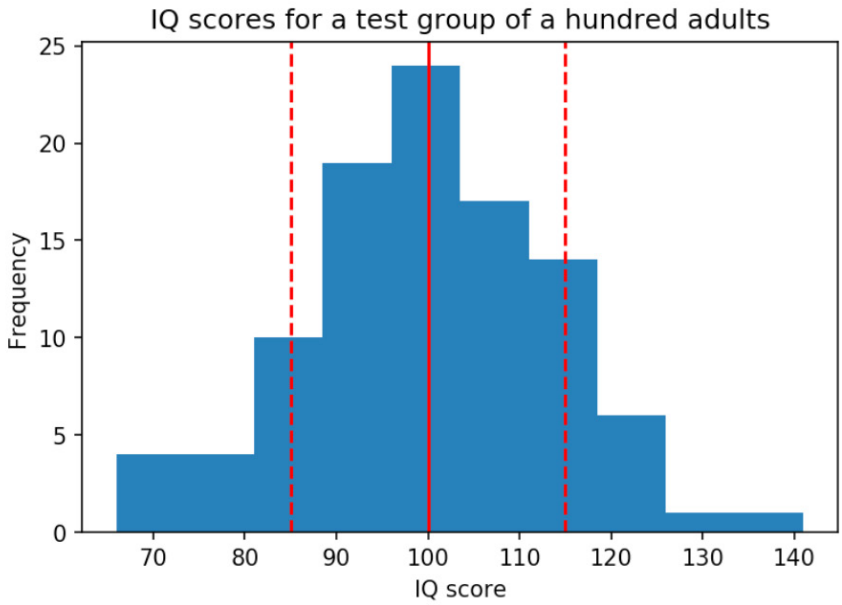

Figure 3.24: Histogram for IQ test

5. The expected output for step 2 is as follows:

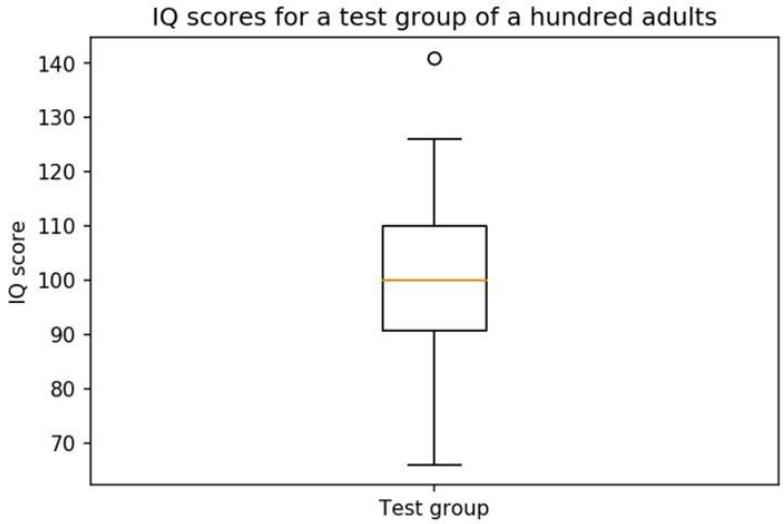

Figure 3.25: Box plot for IQ scores

6. The expected output for step 3 is as follows:

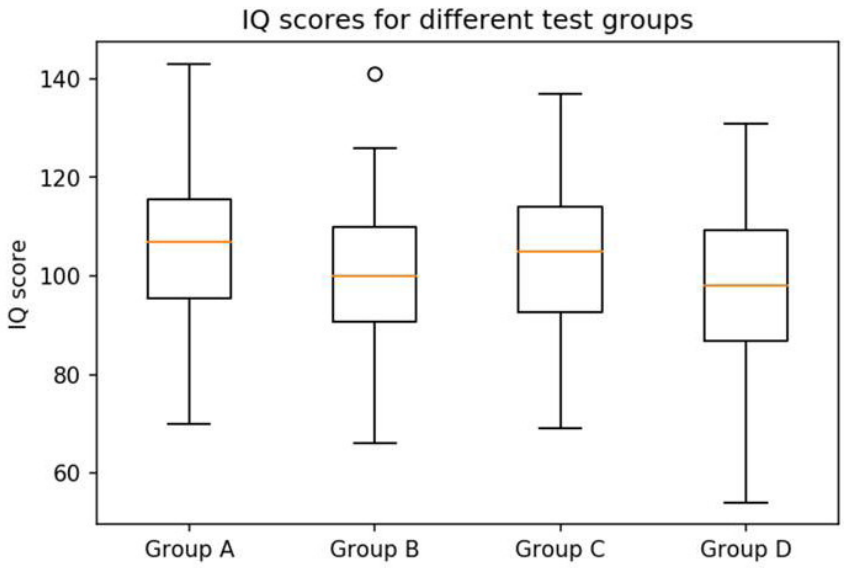

Figure 3.26: Box plot for IQ scores of different test groups

Note:

The solution for this activity can be found on page 284.

Scatter Plot

`plt.scatter(x, y)` creates a scatter plot of y versus x with optionally varying marker size and/or color.

Important parameters:

- **`x`**, **`y`**: Specifies the data positions.
- **`s`**: Optional: Specifies the marker size in points squared.
- **`c`**: Optional: Specifies the marker color. If a sequence of numbers is specified, the numbers will be mapped to colors of the color map.

Example:

```
…
plt.scatter(x, y)
…
```

The result of the preceding code is shown in the following diagram:

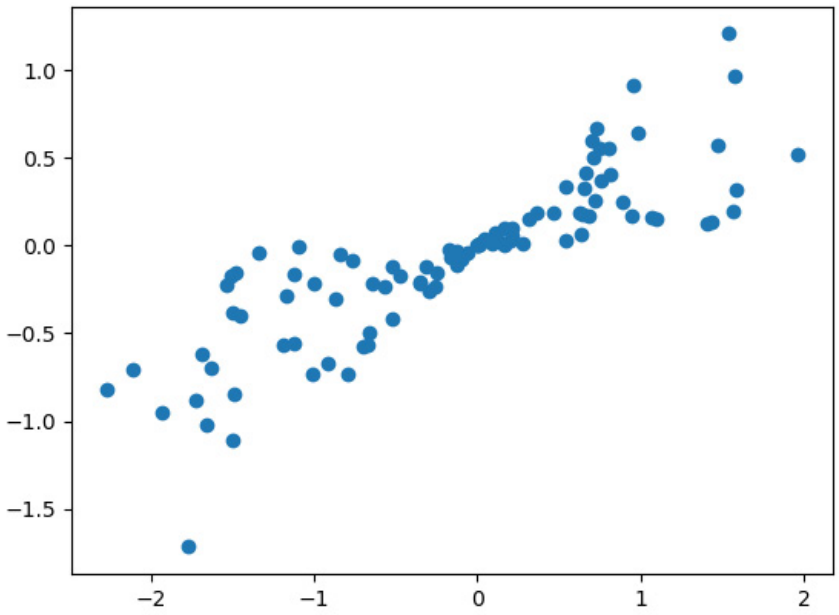

Figure 3.27: Scatter plot

Activity 17: Using a Scatter Plot to Visualize Correlation Between Various Animals

In this activity, we will use a scatter plot to show correlation within a dataset. Let's look at the following scenario: you are given a dataset containing information about various animals. Visualize the correlation between the various animal attributes:

Note

`Axes.set_xscale('log')` *and* ***`Axes.set_yscale('log')`*** *change the scale of the x-axis and y-axis to a logarithmic scale, respectively.*

1. The given dataset is not complete. Filter the data so that you end up with samples containing a body mass and a maximum longevity. Sort the data according to the animal class.
2. Create a scatter plot visualizing the correlation between the body mass and the maximum longevity. Use different colors for grouping data samples according to their class. Add a legend, labels, and a title. Use a log scale for both the x-axis and y-axis.
3. After executing the preceding steps, the expected output should be as follows:

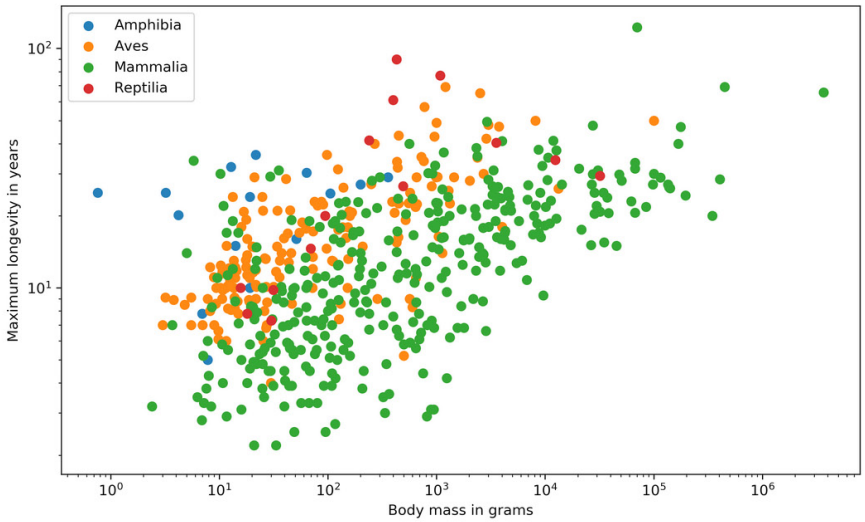

Figure 3.28: Scatter plot on animal statistics

Note:

The solution for this activity can be found on page 287.

Bubble Plot

The **`plt.scatter`** function is used to create a bubble plot. To visualize a third or a fourth variable, the parameters **`s`** (scale) and **`c`** (color) can be used.

Example:

```
...
plt.scatter(x, y, s=z*500, c=c, alpha=0.5)
plt.colorbar()
...
```

The result of the preceding code is shown in the following diagram:

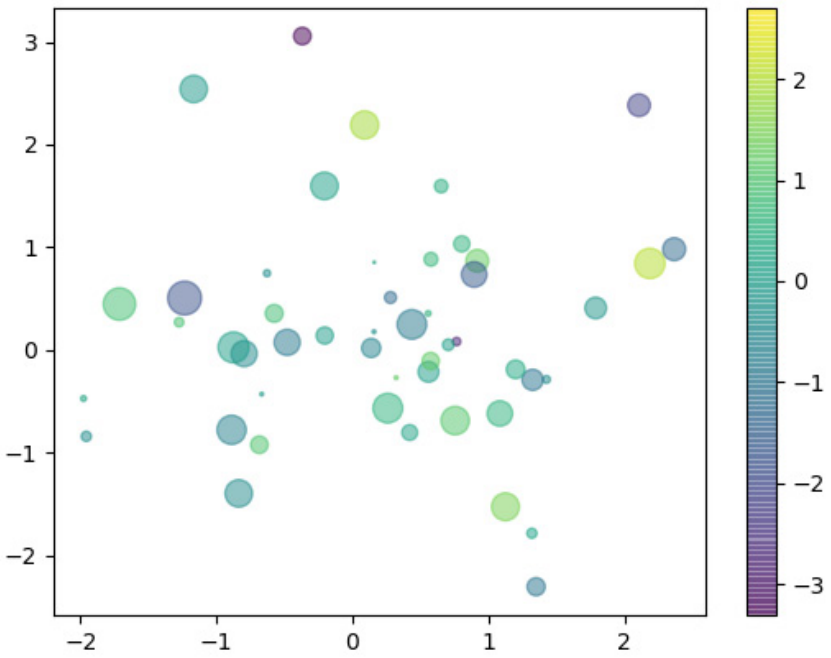

Figure 3.29: Bubble plot with color bar

Layouts

There are multiple ways to define a visualization layout in Matplotlib. We will start with **subplots** and how to use the **tight layout** to create visually appealing plots and then cover **GridSpec**, which offers a more flexible way to create multi-plots.

Subplots

It is often useful to display several plots next to each other. Matplotlib offers the concept of subplots, which are multiple Axes within a Figure. These plots can be grids of plots, nested plots, and so forth.

Explore the following options to create subplots:

- **`plt.subplots(nrows, ncols)`** creates a Figure and a set of subplots.
- **`plt.subplot(nrows, ncols, index)`** or equivalently **`plt.subplot(pos)`** adds a subplot to the current Figure. The index starts at 1. **`plt.subplot(2, 2, 1)`** is equivalent to **`plt.subplot(221)`**.
- **`Figure.subplots(nrows, ncols)`** adds a set of subplots to the specified Figure.
- **`Figure.add_subplot(nrows, ncols, index)`** or equivalently **`Figure.add_subplot(pos)`** adds a subplot to the specified Figure.

For sharing the x or y axis, the parameters **`sharex`** and **`sharey`** must be set, respectively. The axis will have the same limits, ticks, and scale.

`plt.subplot` and **`Figure.add_subplot`** has the option to set a projection. For a polar projection, either set the **`projection='polar'`** parameter or set the **`parameter polar=True`** parameter.

Example 1:

```
…
fig, axes = plt.subplots(2, 2)
axes = axes.ravel()
for i, ax in enumerate(axes):
ax.plot(series[i])
…
…
for i in range(4):
plt.subplot(2, 2, i+1)
plt.plot(series[i])
…
```

Both examples yield the same result, as shown in the following diagram:

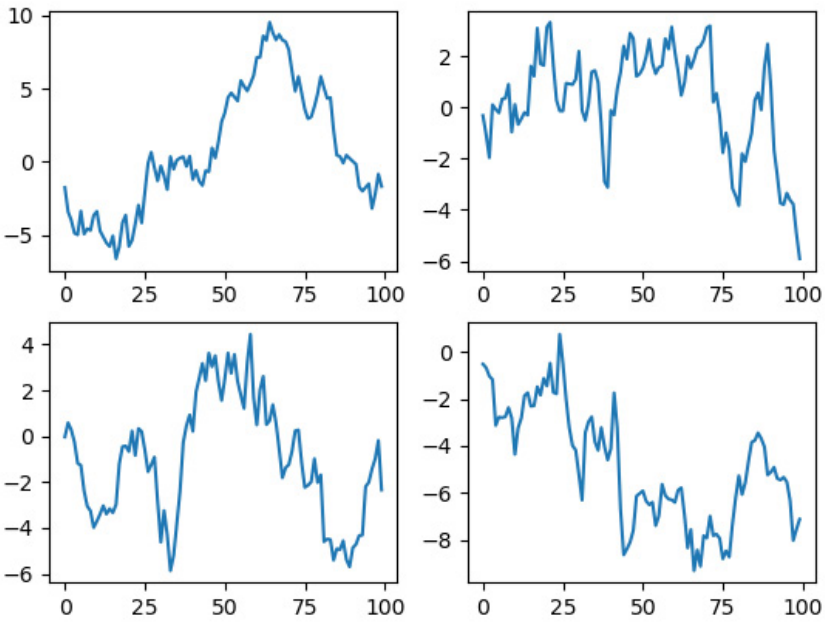

Figure 3.30: Subplots

Example 2:

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
axes = axes.ravel()
for i, ax in enumerate(axes):
ax.plot(series[i])
```

Setting **`sharex`** and **`sharey`** to true results in the following diagram. This allows for a better comparison:

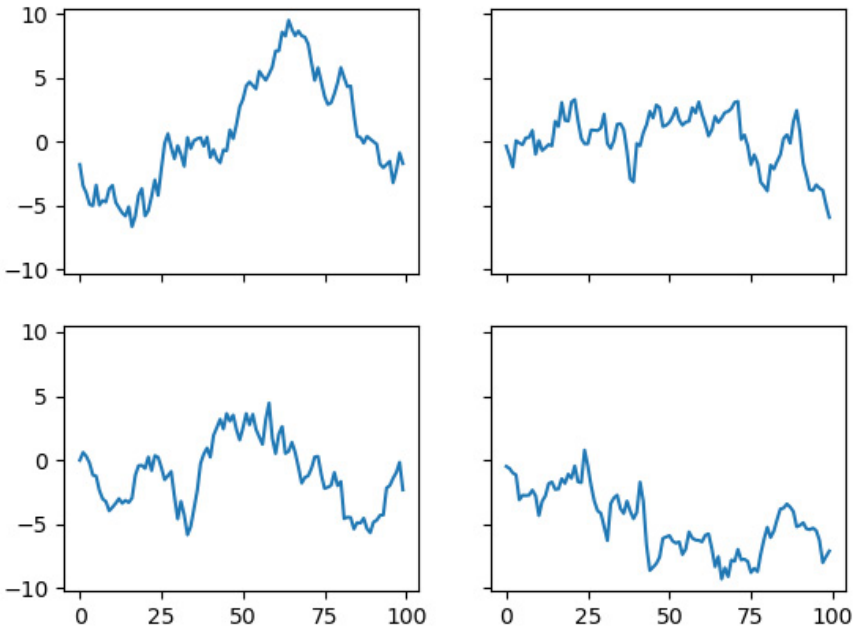

Figure 3.31: Subplots with a shared x and y axis

Tight Layout

`plt.tight_layout()` adjusts subplot parameters so that the subplots fit well in the Figure.

Examples:

If you do not use **`plt.tight_layout()`**, subplots might overlap:

```
…
fig, axes = plt.subplots(2, 2)
axes = axes.ravel()
for i, ax in enumerate(axes):
ax.plot(series[i])
ax.set_title('Subplot ' + str(i))
…
```

The result of the preceding code is shown in the following diagram:

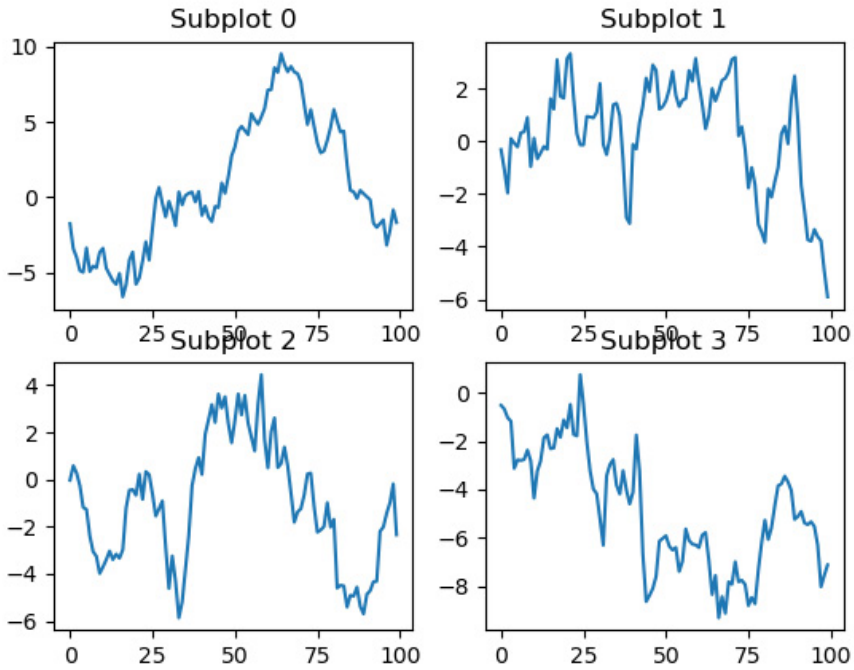

Figure 3.32: Subplots with no layout option

Using **`plt.tight_layout()`** results in no overlapping of the subplots:

```
…
fig, axes = plt.subplots(2, 2)
axes = axes.ravel()
for i, ax in enumerate(axes):
ax.plot(series[i])
ax.set_title('Subplot ' + str(i))
plt.tight_layout()
…
```

The result of the preceding code is shown in the following diagram:

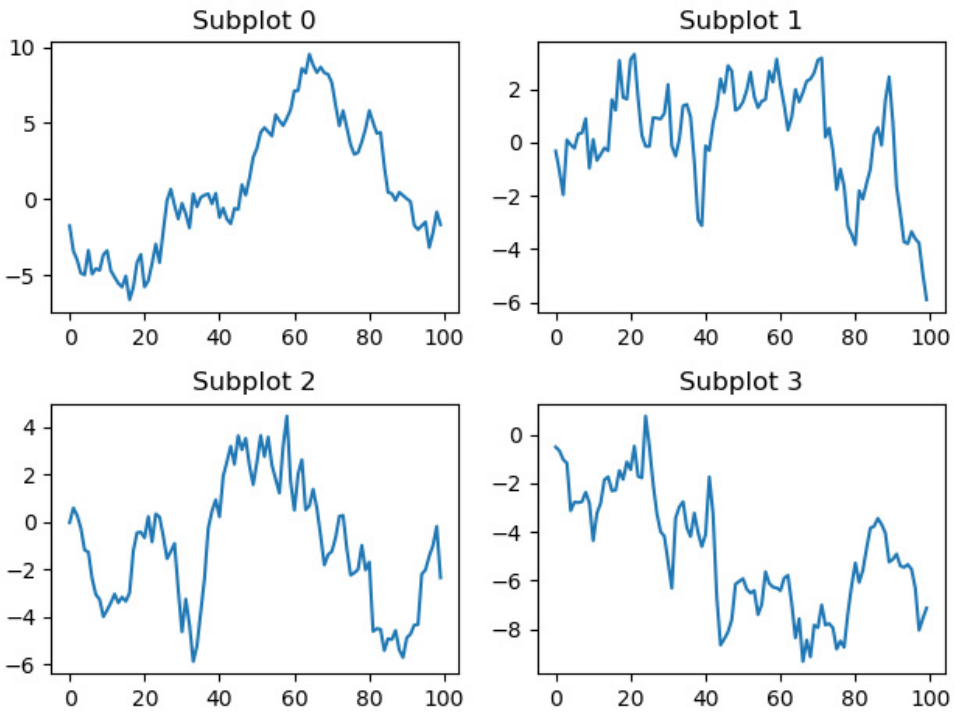

Figure 3.33: Subplots with a tight layout

Radar Charts

Radar charts, also known as **spider** or **web charts**, visualize multiple variables, with each variable plotted on its own axis, resulting in a polygon. All axes are arranged radially, starting at the center with equal distance between each other and have the same scale.

Exercise 5: Working on Radar Charts

In this exercise, it is shown step-by-step how to create a radar chart:

1. Open the Jupyter Notebook **`exercise05.ipynb`** from the **`Lesson03`** folder to implement this exercise.

   Navigate to the path of this file and type in the following at the command line: **`jupyter-lab.`**

2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   ```
   # Import settings
   import numpy as np
   import pandas as pd
   import matplotlib.pyplot as plt
   %matplotlib inline
   ```

3. The following dataset contains ratings of five different attributes for four employees:

   ```
   # Sample data
   # Attributes: Efficiency, Quality, Commitment, Responsible Conduct, Cooperation
   data = pd.DataFrame({
   ```

```
'Employee': ['A', 'B', 'C', 'D'],
'Efficiency': [5, 4, 4, 3,],
'Quality': [5, 5, 3, 3],
'Commitment': [5, 4, 4, 4],
'Responsible Conduct': [4, 4, 4, 3],
'Cooperation': [4, 3, 4, 5]
})
```

4. Create angle values and close the plot:

```
attributes = list(data.columns[1:])
values = list(data.values[:, 1:])
employees = list(data.values[:, 0])
angles = [n / float(len(attributes)) * 2 * np.pi for n in range(len(attributes))]
# Close the plot
angles += angles[:1]
values = np.asarray(values)
values = np.concatenate([values, values[:, 0:1]], axis=1)
```

5. Create subplots with the polar projection. Set a tight layout so that nothing overlaps:

```
# Create figure
plt.figure(figsize=(8, 8), dpi=150)
# Create subplots
for i in range(4):
ax = plt.subplot(2, 2, i + 1, polar=True)
ax.plot(angles, values[i])
ax.set_yticks([1, 2, 3, 4, 5])
ax.set_xticks(angles)
ax.set_xticklabels(attributes)
ax.set_title(employees[i], fontsize=14, color='r')
# Set tight layout
plt.tight_layout()
# Show plot
plt.show()
```

The following figure shows the output of the preceding code:

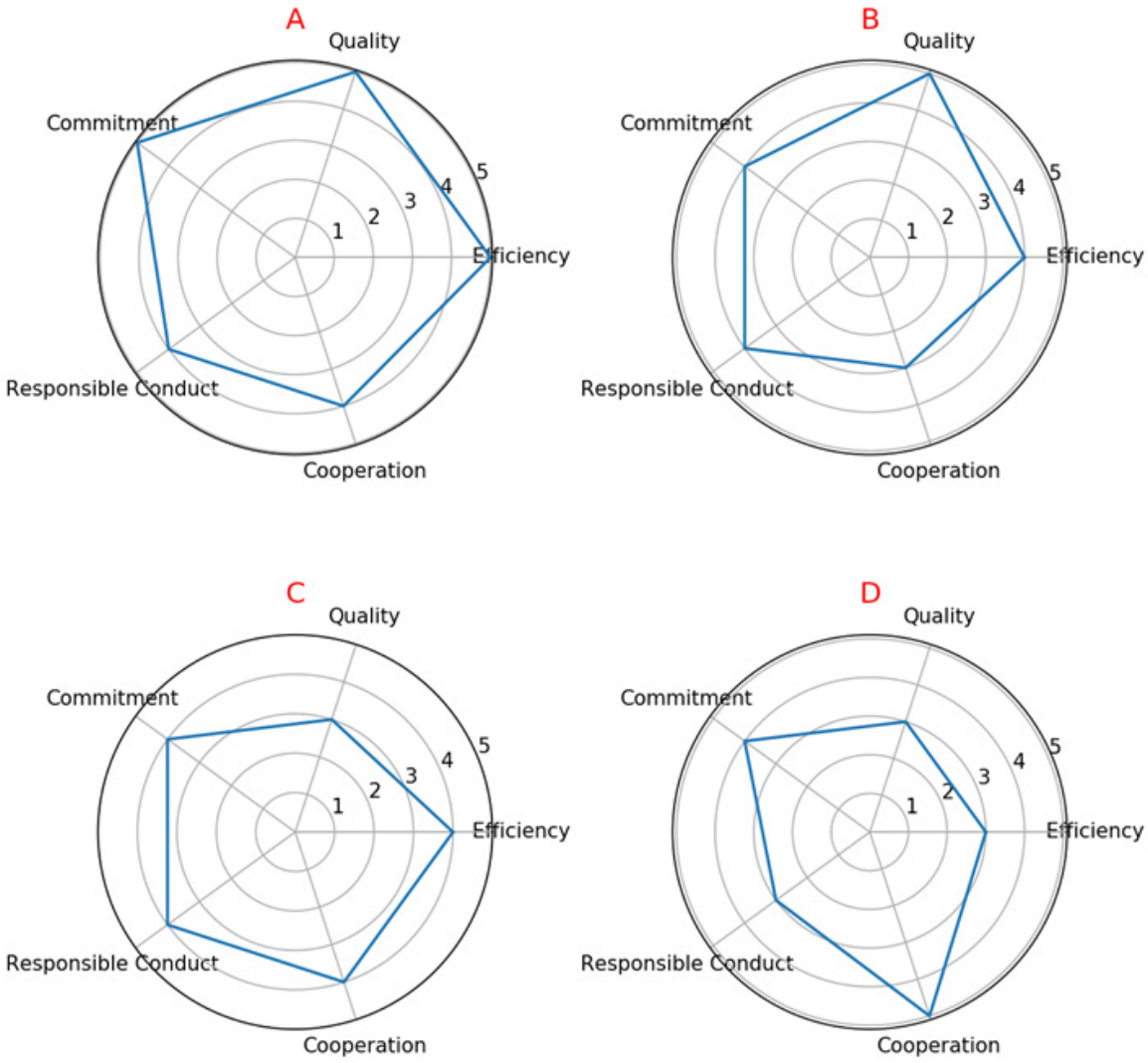

Figure 3.34: Radar charts

GridSpec

`matplotlib.gridspec.GridSpec(nrows, ncols)` specifies the geometry of the grid in which a subplot will be placed.

Example:

```
…
gs = matplotlib.gridspec.GridSpec(3, 4)
ax1 = plt.subplot(gs[:3, :3])
ax2 = plt.subplot(gs[0, 3])
ax3 = plt.subplot(gs[1, 3])
ax4 = plt.subplot(gs[2, 3])
ax1.plot(series[0])
```

```
ax2.plot(series[1])
ax3.plot(series[2])
ax4.plot(series[3])
plt.tight_layout()
…
```

The result of the preceding code is shown in the following diagram:

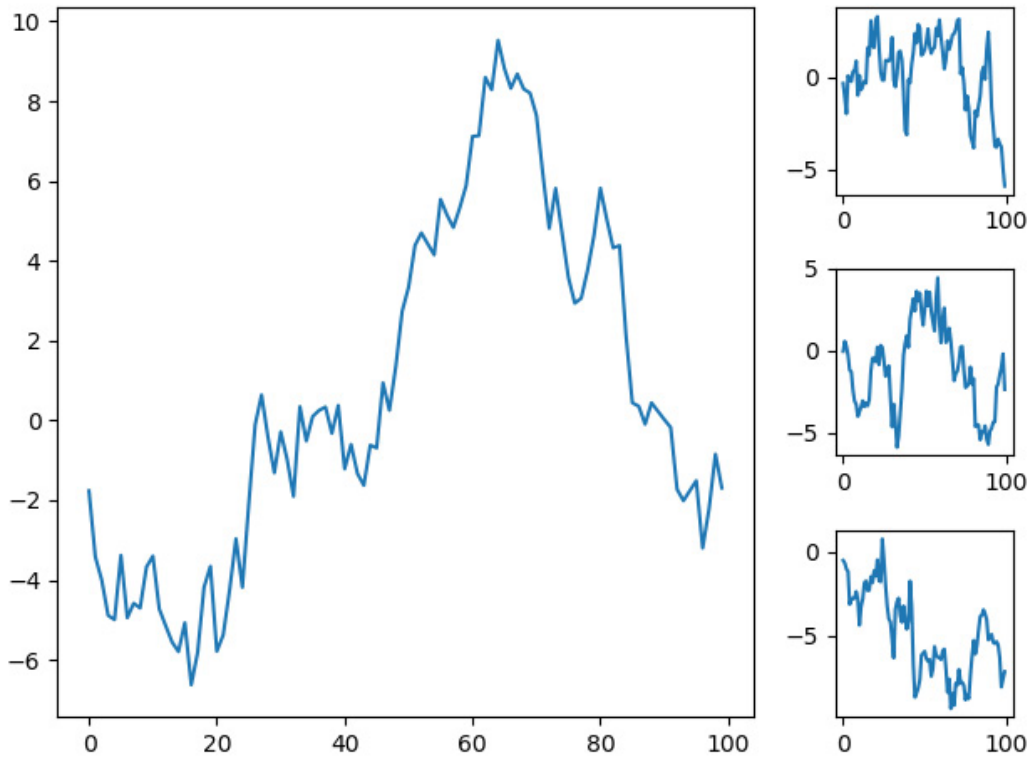

Figure 3.35: GridSpec

Activity 18: Creating Scatter Plot with Marginal Histograms

In this activity, we will make use of **GridSpec** to visualize a **scatter plot** with **marginal histograms**:

1. The given dataset, **AnAge**, which was already used in the previous activity, is not complete. Filter the data so that you end up with samples containing a body mass and a maximum longevity. Select all of the samples of the Aves class with body mass less than 20,000.

2. Create a Figure with a constrained layout. Create a gridspec of size 4x4. Create a scatter plot of size 3x3 and marginal histograms of size 1x3 and 3x1. Add labels and a Figure title.

3. After executing the preceding steps, the expected output should be as follows:

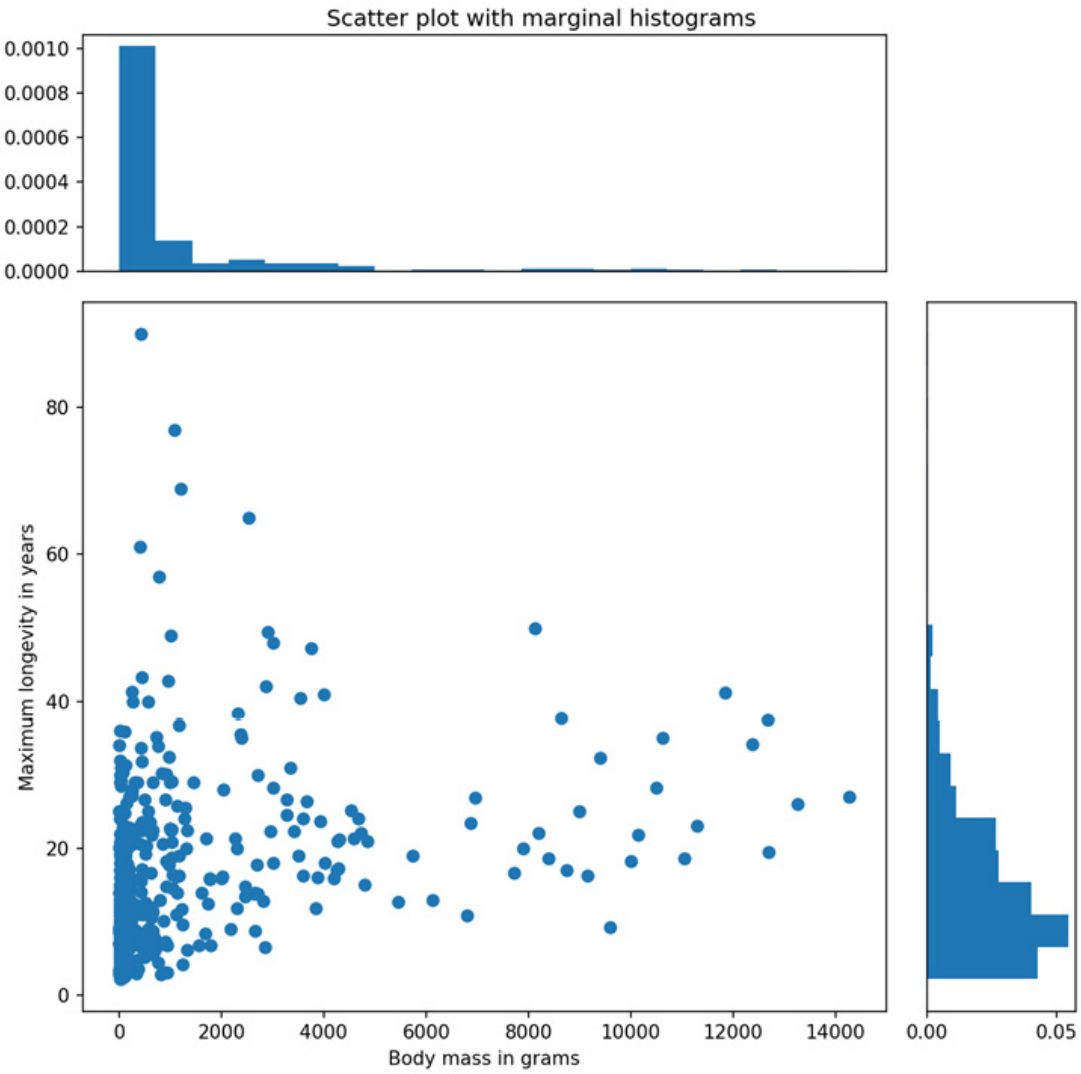

Figure 3.36: Scattered plots with marginal histograms

Note:

The solution for this activity can be found on page 289.

Images

In case you want to include images in your visualizations or in case you are working with image data, Matplotlib offers several functions to deal with images. In this section, we will show you how to **load**, **save**, and **plot** images with Matplotlib.

Note

The images that are used in this topic are from https://unsplash.com/.

Basic Image Operations

Following are the basic operations that are used for designing an image:

Loading Images

In case you encounter image formats that are not supported by Matplotlib, we recommend using the **Pillow** library for loading the image. In Matplotlib, loading images is part of the image submodule. We use the alias **`mpimg`** for the submodule, as follows:

```
import matplotlib.image as mpimg
```

`mpimg.imread(fname)` reads an image and returns it as a **`numpy.array`**. For grayscale images, the returned array has a shape (height, width), for RGB images (height, width, 3), and for RGBA images (height, width, 4). The array values range from 0 to 255.

Saving images

`mpimg.imsave(fname, array)` saves a **`numpy.array`** as an image file. If the **`format`** parameter is not given, the format is deduced from the filename extension. With the optional parameters **`vmin`** and **`vmax`**, the color limits can be set manually. For a grayscale image, the default for the optional parameter **`cmap`** is **`'viridis'`**; you might want to change it to **`'gray'`**.

Plotting a Single Image

`plt.imshow(img)` displays an image and returns an **`AxesImage`**. For grayscale images with shape (height, width), the image array is visualized using a colormap. The default colormap is **`'viridis'`**, as illustrated in Figure 3.38. For actually visualizing a grayscale image, the colormap has to be set to **`'gray'`**, that is, **`plt.imshow(img, cmap='gray')`**, which is illustrated in the following diagram. Values for grayscale, RGB, and RGBA images can be either **`float`** or **`uint8`**, and range from **`[0…1]`** or **`[0…255]`**, respectively. To manually define the value range, the parameters **`vmin`** and **`vmax`** must be specified. A visualization of an RGB image is shown in the following diagrams:

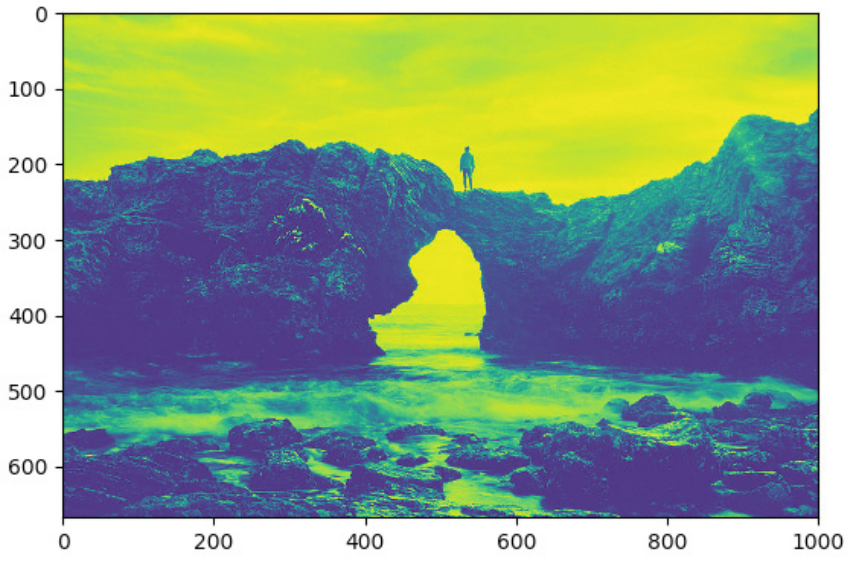

Figure 3.37: Grayscale image with default viridis colormap

The following Figure shows a Grayscale image with gray colormap:

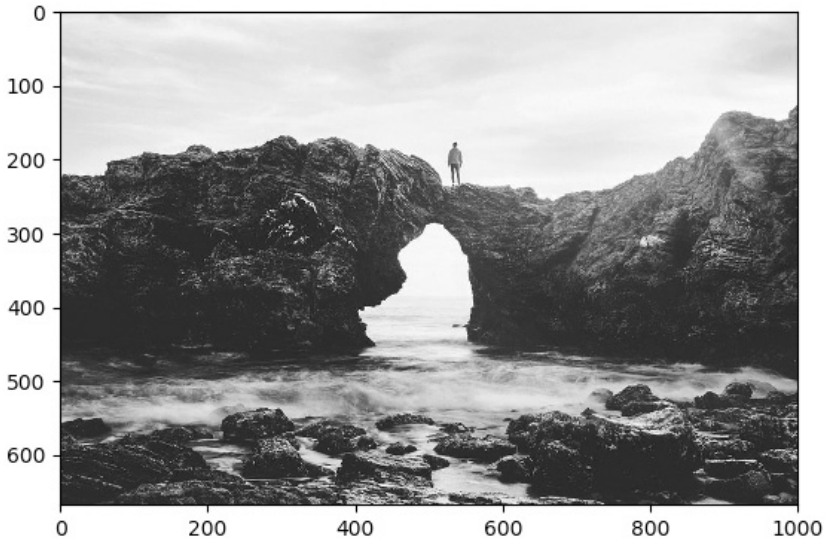

Figure 3.38: Grayscale image with gray colormap

The following Figure shows a RGB image:

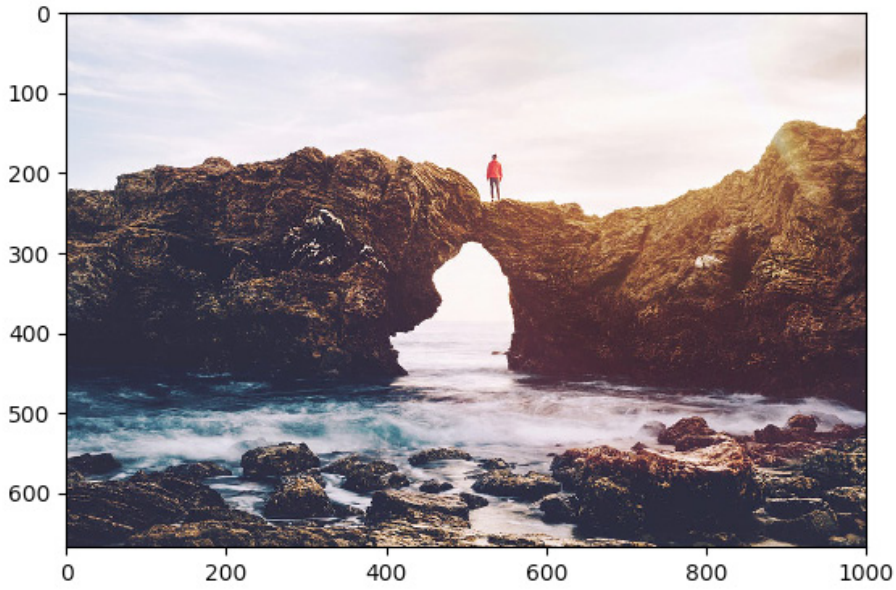

Figure 3.39: RGB image

Sometimes, it might be helpful to get an insight into the color values. We can simply add a color bar to the image plot. It is recommended to use a colormap with a high contrast, for example, '**`jet`**':

```
…
plt.imshow(img, cmap='jet')
plt.colorbar()
…
```

The preceding example is illustrated in the following diagram:

 Figure 3.40: Image with jet colormap and color bar

Figure 3.40: Image with jet colormap and color bar

Another way to get insight into the image values is to plot a histogram, as shown in the following diagram. To plot the histogram for an image array, the array has to be flattened using **`numpy.ravel`**:

```
…
plt.hist(img.ravel(), bins=256, range=(0, 1))
…
```

The following Figure shows an output of the preceding code:

 Figure 3.41: Histogram of image values

Figure 3.41: Histogram of image values

Plotting Multiple Images in a Grid

For plotting multiple images in a grid, we can simply use **`plt.subplots`** and plot an image per Axes:

```
…
fig, axes = plt.subplots(1, 2)
for i in range(2):
axes[i].imshow(imgs[i])
…
```

The result of the preceding code is shown in the following diagram:

Figure 3.42: Multiple images within a grid

In some situations, it would be neat to remove the ticks and add labels. **`axes.set_xticks([])`** and **`axes.set_yticks([])`** remove x-ticks and y-ticks, respectively. **`axes.set_xlabel('label')`** adds a label:

```
…
fig, axes = plt.subplots(1, 2)
labels = ['coast', 'beach']
for i in range(2):
axes[i].imshow(imgs[i])
axes[i].set_xticks([])
axes[i].set_yticks([])
axes[i].set_xlabel(labels[i])
…
```

The result of the preceding code is shown in the following diagram:

 Figure 3.43: Multiple images with labels

Figure 3.43: Multiple images with labels

Activity 19: Plotting Multiple Images in a Grid

In this activity, we will plot images in a grid:

1. Load all four images from the subfolder data.
2. Visualize the images in a 2x2 grid. Remove the axes and give each image a label.
3. After executing the preceding steps, the expected output should be as follows:

Figure 3.44: Visualizing images in the 2x2 grid

Note:

The solution for this activity can be found on page 290.

Writing Mathematical Expressions

In case you need to write mathematical expressions within the code, Matplotlib supports **TeX**. You can use it in any text by placing your mathematical expression in a pair of dollar signs. There is no need to have TeX installed since Matplotlib comes with its own parser.

An example of this is given in the following code:

```
…
plt.xlabel('$x$')
plt.ylabel('$\cos(x)$')
…
```

The following Figure shows an output of the preceding code:

Figure 3.45: Figure demonstrating mathematical expressions

TeX examples:

- **`'$\alpha_i>\beta_i$'`** produces
- **`'$\sum_{i=0}^\infty x_i$'`** produces
- **`'$\sqrt[3]{8}$'`** produces
- **`'$\frac{3 - \frac{x}{2}}{5}$'`** produces

Summary

In this chapter, we provided a detailed introduction to Matplotlib, one of the most popular visualization libraries for Python. We started off with the basics of pyplot and its operations, and then followed up with a deep insight into the numerous possibilities that helps to enrich visualizations with text. Using practical examples, this chapter covered the most popular plotting functions that Matplotlib offers out of

the box, including comparison charts, composition, and distribution plots. This chapter is rounded off with how to visualize images and write mathematical expressions.

In the next chapter, we will learn about the Seaborn library. Seaborn is built on top of Matplotlib and provides a higher-level abstraction to make visually appealing visualizations. We will also address advanced visualization types.

Chapter 4

Simplifying Visualizations Using Seaborn

Learning Objectives

By the end of this chapter, you will be able to:

- Explain why Seaborn is better than Matplotlib
- Design visually appealing plots efficiently
- Create insightful Figures

In this chapter, we will see how Seaborn is different than Matplolib and also construct effective plots using Figure.

Introduction

Unlike **Matplotlib**, **Seaborn** is not a standalone Python library. It is built on top of Matplotlib and provides a higher-level abstraction to make visually appealing statistical visualizations. A neat feature of Seaborn is the ability to integrate with DataFrames from the pandas library.

With Seaborn, we attempt to make visualization a central part of data exploration and understanding. Internally, Seaborn operates on DataFrames and arrays that contain the complete dataset. This enables it to perform semantic mappings and statistical aggregations that are essential for displaying informative visualizations. Seaborn can also be solely used to change the style and appearance of Matplotlib visualizations.

The most prominent features of Seaborn are as follows:

- Beautiful out of the box plots with different themes
- Built-in color palettes that can be used to reveal patterns in the dataset
- Dataset-oriented interface
- A high-level abstraction that still allows for complex visualizations

Advantages of Seaborn

Seaborn is built on top of Matplotlib, and also addresses some of the main pain points of working with Matplotlib.

Working with DataFrames using Matplotlib adds some inconvenient overhead. For example: simply exploring your dataset can take up a lot of time, since you need some additional data wrangling to be able to plot the data from the DataFrames using Matplotlib.

Seaborn, however, is built to operate on DataFrames and full dataset arrays, which makes this process simpler. It internally performs the necessary semantic mappings and statistical aggregation to produce informative plots. The following is an example of plotting using the Seaborn library:

```
import seaborn as sns
import pandas as pd
```

```
sns.set(style="ticks")
data = pd.read_csv("data/salary.csv")
sns.relplot(x="Salary", y="Age", hue="Education", style="Education",
col="Gender", data=data)
```

This creates the following plot:

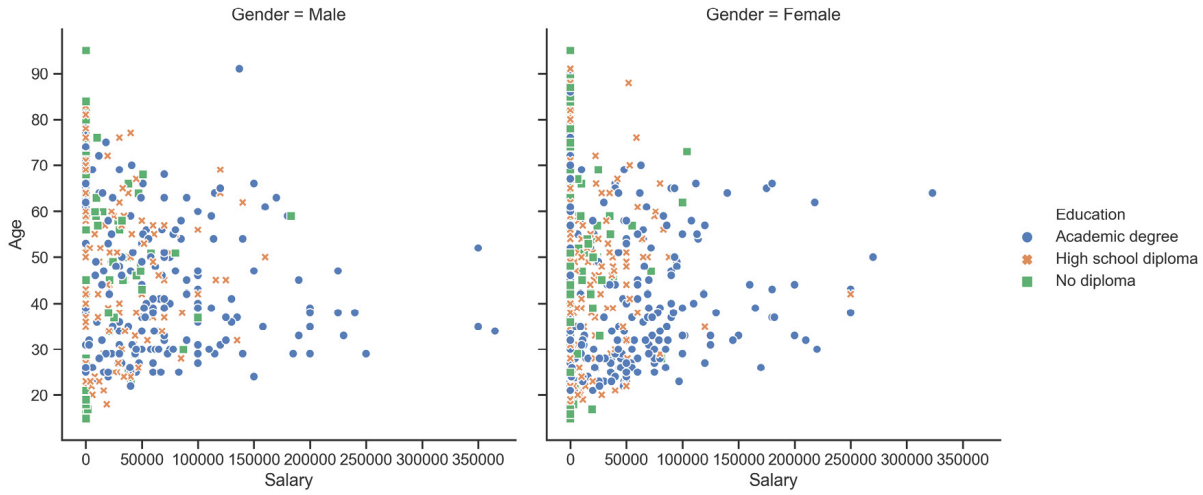

Figure 4.1: Seaborn Relation plot

Behind the scenes, Seaborn uses Matplotlib to draw plots. Even though many tasks can be accomplished with just Seaborn, a further customization might require the usage of Matplotlib. We only provided the names of the variables in the dataset and the roles they play in the plot. Unlike in Matplotlib, it is not necessary to translate the variables into parameters of the visualization.

Other pain points are the default Matplotlib parameters and configurations. The default parameters in Seaborn provide better visualizations without additional customization. We will look at these default parameters in detail in the upcoming topics.

For users who are already familiar with Matplotlib, the extension with Seaborn is trivial, since the core concepts are mostly similar.

Controlling Figure Aesthetics

As we mentioned previously, Matplotlib is highly customizable. But this also has the effect that it is difficult to know what settings to tweak to achieve a visually appealing plot. In contrast, Seaborn provides several customized themes and a high-level interface for controlling the appearance of Matplotlib figures.

The following code snippet creates a simple line plot in Matplotlib:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
```

```
plt.plot(x1, label='Group A')
plt.plot(x2, label='Group B')
plt.legend()
plt.show()
```

This is what the plot looks like with Matplotlib's default parameters:

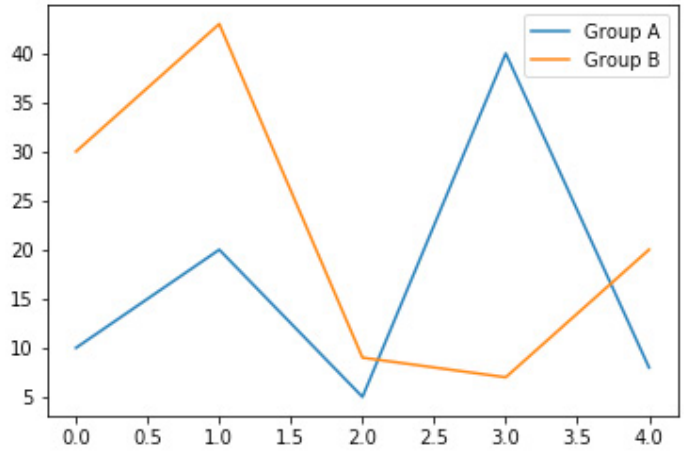

Figure 4.2: Matplotlib line plot

To switch to the Seaborn defaults, simply call the **`set()`** function:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
plt.plot(x1, label='Group A')
plt.plot(x2, label='Group B')
plt.legend()
plt.show()
```

The plot is as follows:

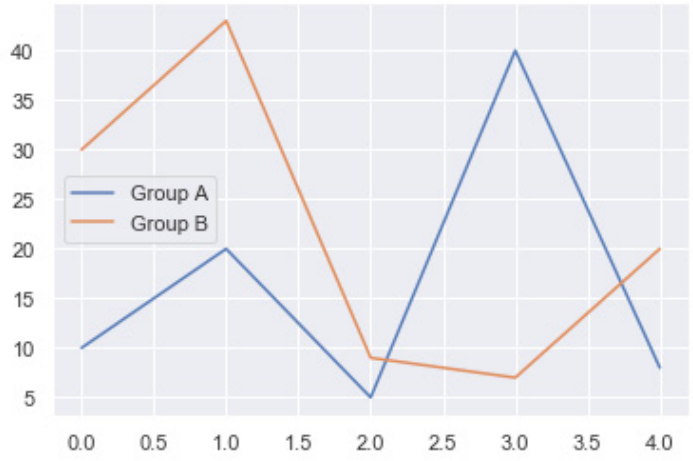

Figure 4.3: Seaborn line plot

Seaborn categorizes Matplotlib's parameters into two groups. The first group contains parameters for the aesthetics of the plot, while the second group scales various elements of the figure so that it can be easily used in different contexts, such as visualizations that are used for presentations, posters, and so on.

Seaborn Figure Styles

To control the style, Seaborn provides two methods: **`set_style(style, [rc])`** and **`axes_style(style, [rc])`**.

`seaborn.set_style(style, [rc])` sets the aesthetic style of the plots.

Parameters:

- **`style`**: A dictionary of parameters or the name of one of the following preconfigured sets: **`darkgrid`**, **`whitegrid`**, **`dark`**, **`white`**, or **`ticks`**
- **`rc`** (optional): Parameter mappings to override the values in the preset Seaborn style dictionaries

Here is an example:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("whitegrid")
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
plt.plot(x1, label='Group A')
plt.plot(x2, label='Group B')
plt.legend()
plt.show()
```

This results in the following plot:

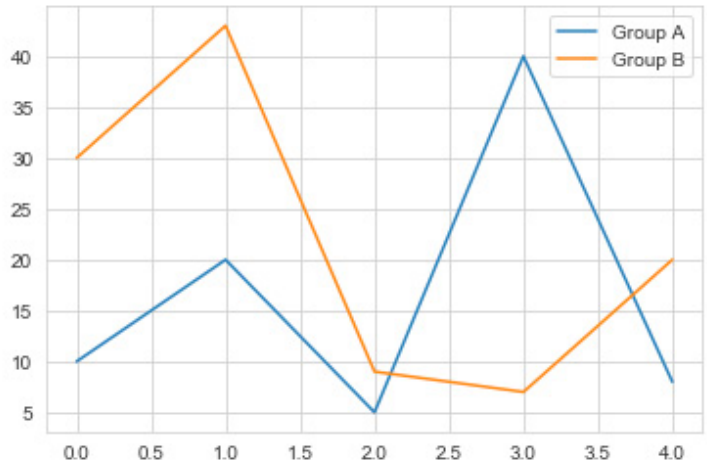

Figure 4.4: Seaborn line plot with whitegrid style

`seaborn.axes_style(style, [rc])` returns a parameter dictionary for the aesthetic style of the plots. The function can be used in a with statement to temporarily change the style parameters.

Here are the parameters:

- **`style`**: A dictionary of parameters or the name of one of the following preconfigured sets: **`darkgrid`**, **`whitegrid`**, **`dark`**, **`white`**, or **`ticks`**
- **`rc`** (optional): Parameter mappings to override the values in the preset Seaborn style dictionaries.

Here is an example:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
with sns.axes_style('dark'):
plt.plot(x1, label='Group A')
plt.plot(x2, label='Group B')
plt.legend()
plt.show()
```

The aesthetics are only changed temporarily. The result is shown in the following diagram:

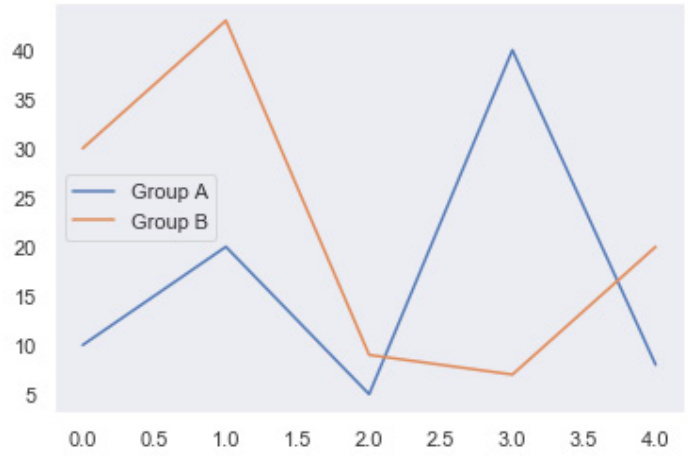

Figure 4.5: Seaborn Line plot with dark axes style

For further customization, you can pass a dictionary of parameters to the **`rc`** argument. You can only override parameters that are part of the style definition.

Removing Axes Spines

Sometimes, it might be desirable to remove the top and right axes spines.

`seaborn.despine(fig=None, ax=None, top=True, right=True, left=False, bottom=False, offset=None, trim=False)` removes the top and right spines from the plot.

The following code helps to remove the axes spines:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("white")
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
plt.plot(x1, label='Group A')
plt.plot(x2, label='Group B')
sns.despine()
plt.legend()
plt.show()
```

This results in the following plot:

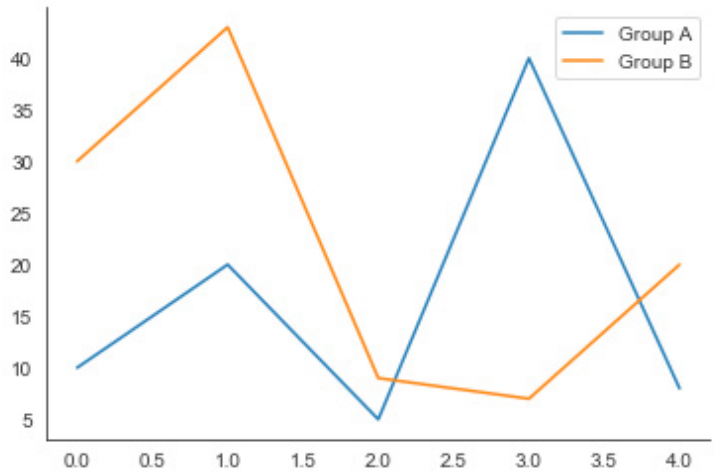

Figure 4.6: Despined Seaborn line plot

Contexts

A separate set of parameters controls the scale of plot elements. This is a handy way to use the same code to create plots that are suited for use in contexts where larger or smaller plots are necessary. To control the context, two functions can be used.

`seaborn.set_context(context, [font_scale], [rc])` sets the plotting context parameters. This does not change the overall style of the plot, but affects things such as the size of the labels, lines, and so on. The base context is **`notebook`**, and the other contexts are **`paper`**, **`talk`**, and **`poster`**, which are versions of the **`notebook`** parameters scaled by 0.8, 1.3, and 1.6, respectively.

Here are the parameters:

- **`context`**: A dictionary of parameters or the name of one of the following preconfigured sets: paper, notebook, talk, or poster
- **`font_scale`** (optional): A scaling factor to independently scale the size of font elements
- **`rc`** (optional): Parameter mappings to override the values in the preset Seaborn context dictionaries

The following code helps set the context:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_context("poster")
plt.figure()
x1 = [10, 20, 5, 40, 8]
x2 = [30, 43, 9, 7, 20]
plt.plot(x1, label='Group A')
plt.plot(x2, label='Group B')
```

```
plt.legend()
plt.show()
```

The preceding code generates the following output:

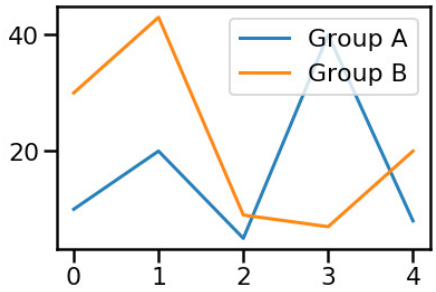

Figure 4.7: Seaborn line plot with poster context

`seaborn.plotting_context(context, [font_scale], [rc])` returns a parameter dictionary to scale elements of the Figure. This function can be used with a statement to temporarily change the context parameters.

Here are the parameters:

- **`context`**: A dictionary of parameters or the name of one of the following preconfigured sets: paper, notebook, talk, or poster
- **`font_scale`** (optional): A scaling factor to independently scale the size of font elements
- **`rc`** (optional): Parameter mappings to override the values in the preset Seaborn context dictionaries

Activity 20: Comparing IQ Scores for Different Test Groups by Using a Box Plot

In this activity, we will compare IQ scores among different test groups using a box plot of the Seaborn library:

1. Use pandas to read the data located in the subfolder data.
2. Access the data of each group in the column, convert them into a list, and assign this list to the variables of each respective group.
3. Create a pandas DataFrame using the preceding data by using the data of each respective group.
4. Create a box plot for each of the IQ scores of different test groups using Seaborn's **`boxplot`** function.
5. Use the **`whitegrid`** style, set the context to talk, and remove all axes spines, except the one on the bottom. Add a title to the plot.
6. After executing the preceding steps, the final output should be as follows:

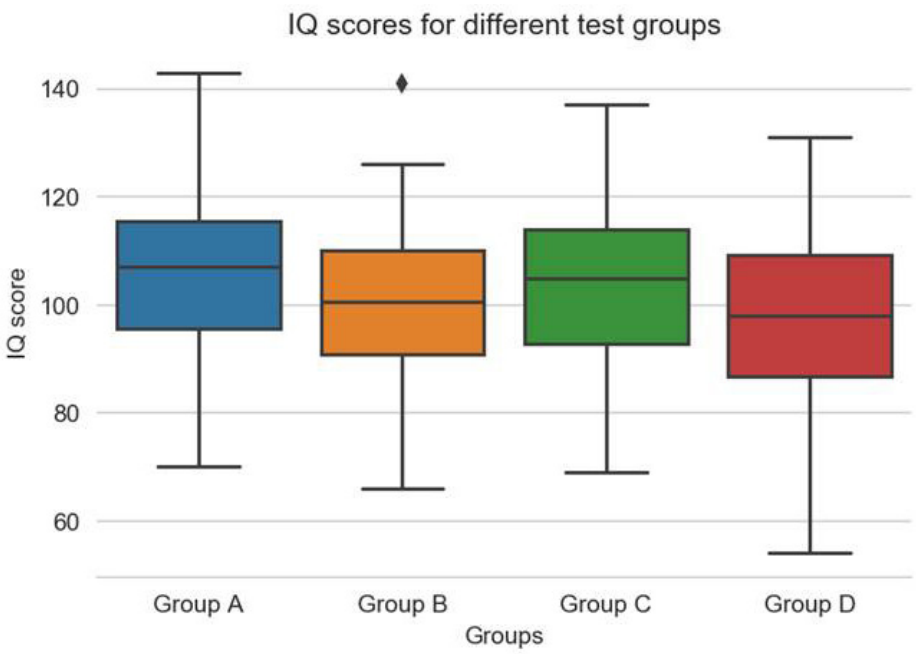

Figure 4.8: IQ scores of groups

Note:

The solution for this activity can be found on page 292.

Color Palettes

Color is a very important factor for your visualization. Color can reveal patterns in the data if used effectively or hide patterns if used poorly. Seaborn makes it easy to select and use **color palettes** that are suited to your task. The **`color_palette()`** function provides an interface for many of the possible ways to generate colors.

`seaborn.color_palette([palette], [n_colors], [desat])` returns a list of colors, thus defining a color palette.

Here are the parameters:

- **`palette`** (optional): Name of palette or None to return the current palette.
- **`n_colors`** (optional): Number of colors in the palette. If the specified number of colors is larger than the number of colors in the palette, the colors will be cycled.
- **`desat`** (optional): Proportion to desaturate each color by.

You can set the palette for all plots with **`set_palette()`**. This function accepts the same arguments as **`color_palette()`**. In the following sections, we will explain how color palettes are divided into different groups.

Categorical Color Palettes

Categorical palettes are best for distinguishing discrete data that does not have an inherent ordering. There are six default themes in Seaborn: **`deep`**, **`muted`**, **`bright`**, **`pastel`**, **`dark`**, and **`colorblind`**. The code and output for each theme is provided in the

following code:

```
import seaborn as sns
palette1 = sns.color_palette("deep")
sns.palplot(palette1)
```

The following figure shows the output of the preceding code:

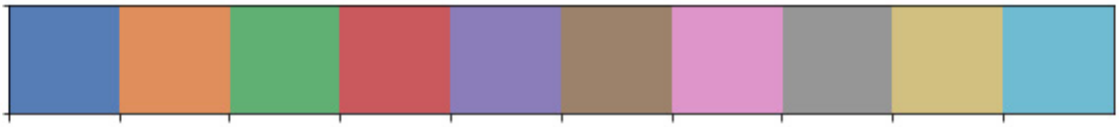

Figure 4.9: Deep color palette

```
palette2 = sns.color_palette("muted")
sns.palplot(palette2)
```

The following figure shows the output of the preceding code:

Figure 4.10: Muted color palette

The following figure shows a bright color palette:

```
palette3 = sns.color_palette("bright")
sns.palplot(palette3)
```

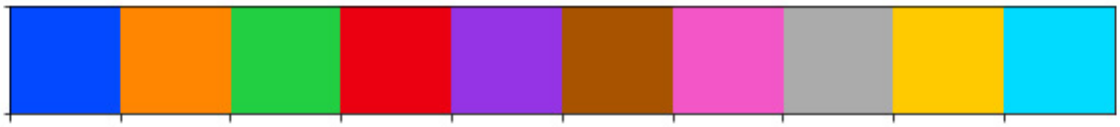

Figure 4.11: Bright color palette

The following figure shows a pastel color palette:

```
palette4 = sns.color_palette("pastel")
sns.palplot(palette4)
```

Figure 4.12: Pastel color palette

The following figure shows a dark color palette:

```
palette5 = sns.color_palette("dark")
sns.palplot(palette5)
```

Figure 4.13: Dark color palette

```
palette6 = sns.color_palette("colorblind")
sns.palplot(palette6)
```

The following figure shows the output of the preceding code:

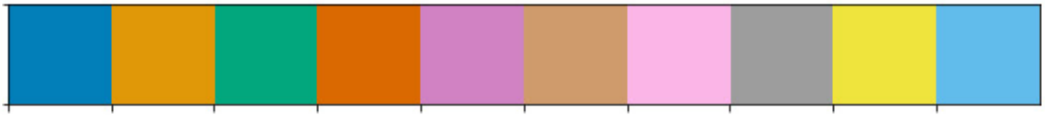

Figure 4.14: Colorblind color palette

Sequential Color Palettes

Sequential color palettes are appropriate when the data ranges from relatively low or uninteresting values to relatively high or interesting values. The following code snippets, along with their respective outputs, give us a better insight into **sequential color palettes**:

```
custom_palette2 = sns.light_palette("brown")
sns.palplot(custom_palette2)
```

The following figure shows the output of the preceding code:

Figure 4.15: Custom brown color palette

The preceding palette can also be reversed by setting the `reverse` parameter to `True` in the following code:

```
custom_palette3 = sns.light_palette("brown", reverse=True)
sns.palplot(custom_palette3)
```

The following figure shows the output of the preceding code:

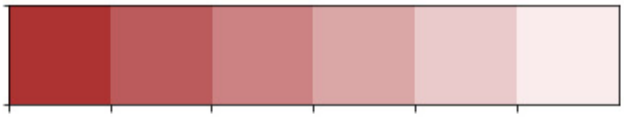

Figure 4.16: Custom reversed brown color palette

Diverging Color Palettes

`Diverging color palettes` are used for data that consists of a well-defined midpoint. Emphasis is being laid on both high and low values. For example: if you are plotting any population changes for a particular region from some baseline population, it is best to use diverging colormaps to show the relative increase and decrease in the population. The following code snippet and output provides a better understanding about diverging plot, wherein we use the **`coolwarm`** template, which is built into Matplotlib:

```
custom_palette4 = sns.color_palette("coolwarm", 7)
sns.palplot(custom_palette4)
```

The following figure shows the output of the preceding code:

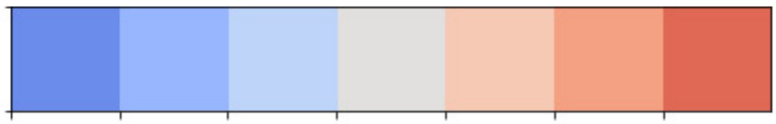

Figure 4.17: Coolwarm color palette

You can use the **`diverging_palette()`** function to create custom diverging palettes. We can pass two **hues** in degrees as parameters, along with the total number of palettes. The following code snippet and output provides better insight:

```
custom_palette5 = sns.diverging_palette(440, 40, n=7)
sns.palplot(custom_palette5)
```

The following figure shows the output of the preceding code:

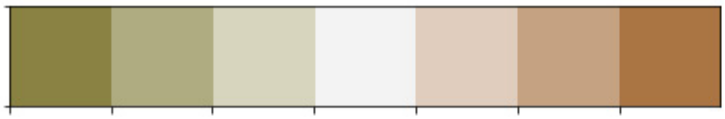

Figure 4.18: Custom diverging color palette

Activity 21: Using Heatmaps to Find Patterns in Flight Passengers' Data

In this activity, we will use a heatmap to find the patterns in the flight passengers' data:

1. Use pandas to read the data located in the subfolder data. The given dataset contains the monthly figures for flight passengers from the years 2001 to 2012.
2. Use a heatmap to visualize the given data.
3. Use your own color map. Make sure that the lowest value is the darkest color and that the highest value is the brightest color.
4. After executing the preceding steps, the expected output should be as follows:

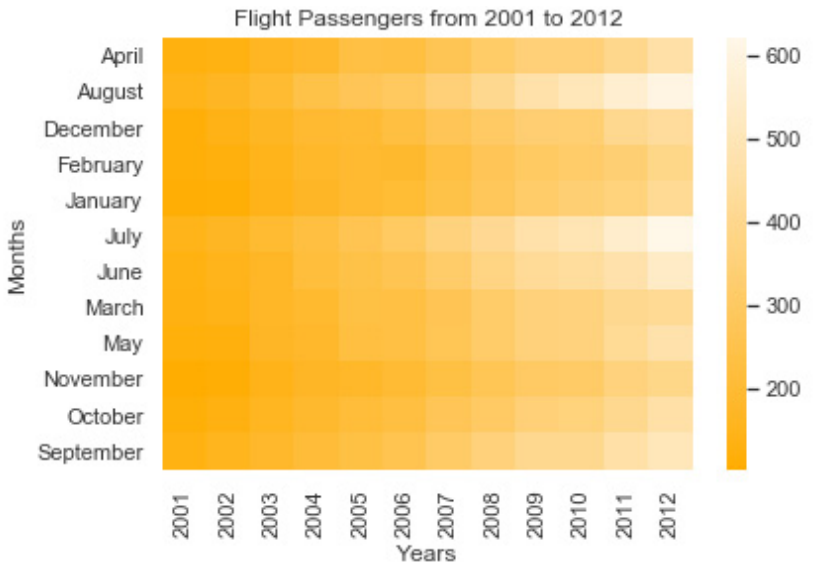

Figure 4.19: Heatmap of Flight Passengers data

Note:

The solution for this activity can be found on page 294.

Interesting Plots in Seaborn

In the previous chapter, we discussed various plots in Matplotlib, but there are still a few visualizations left that we want to discuss.

Bar Plots

In the last chapter, we already explained how to create bar plots with Matplotlib. Creating bar plots with subgroups was quite tedious, but Seaborn offers a very convenient way to create various bar plots. They can be also used in Seaborn to represent estimates of central tendency with the height of each rectangle and indicates the uncertainty around that estimate using error bars.

The following example gives you a good idea of how this works:

```
import pandas as pd
import seaborn as sns
data = pd.read_csv("data/salary.csv")
sns.set(style="whitegrid")
sns.barplot(x="Education", y="Salary", hue="District", data=data)
```

The result is shown in the following diagram:

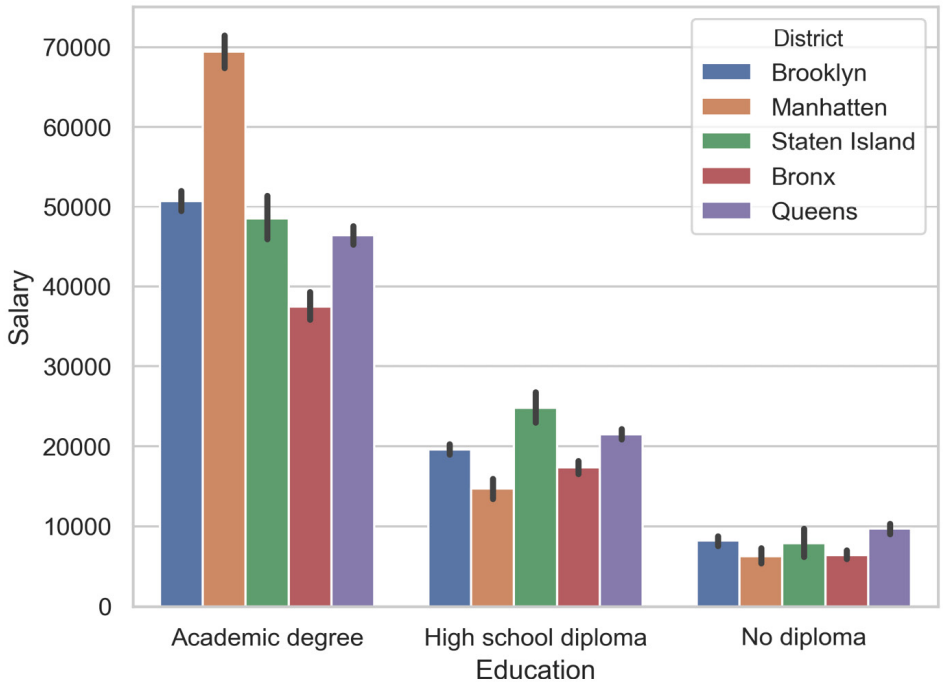

Figure 4.20: Seaborn bar plot

Activity 22: Movie Comparison Revisited

In this activity, we will use a bar plot to compare movie scores. You will be given five movies with scores from Rotten Tomatoes. The Tomatometer is the percentage of approved Tomatometer critics who have given a positive review for the movie. The Audience Score is the percentage of users who have given a score of 3.5 or higher out of 5. Compare these two scores among the five movies:

1. Use pandas to read the data located in the subfolder data.
2. Transform the data into a useable format for Seaborn's bar plot function.
3. Use Seaborn to create a visually appealing bar plot that compares the two scores for all five movies.
4. After executing the preceding steps, the expected output should look as follows:

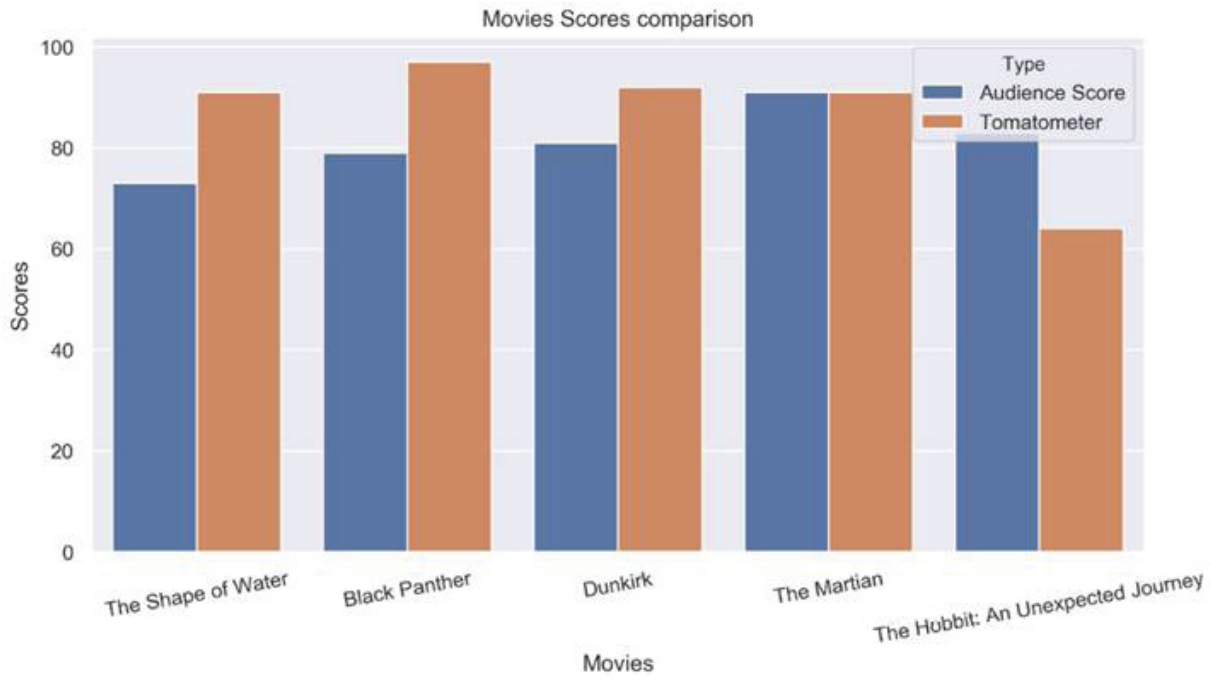

Figure 4.21: Movie Scores comparison

Note:

The solution for this activity can be found on page 295.

Kernel Density Estimation

It is often useful to visualize how variables of a dataset are distributed. Seaborn offers handy functions to examine univariate and bivariate distributions. One possible way to look at a univariate distribution in Seaborn is by using the **`distplot()`** function. This will draw a histogram and fit a **kernel density estimate** (**KDE**), as illustrated in the following example:

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
x = np.random.normal(size=50)
sns.distplot(x)
```

The result is shown in the following diagram:

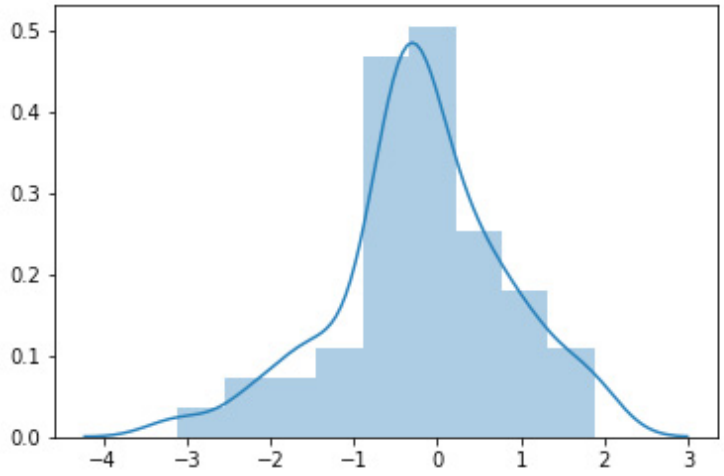

Figure 4.22: KDE with histogram for univariate distribution

To just visualize the KDE, Seaborn provides the **`kdeplot()`** function:

```
sns.kdeplot(x, shade=True)
```

The KDE plot is shown in the following diagram, along with a shaded area under the curve:

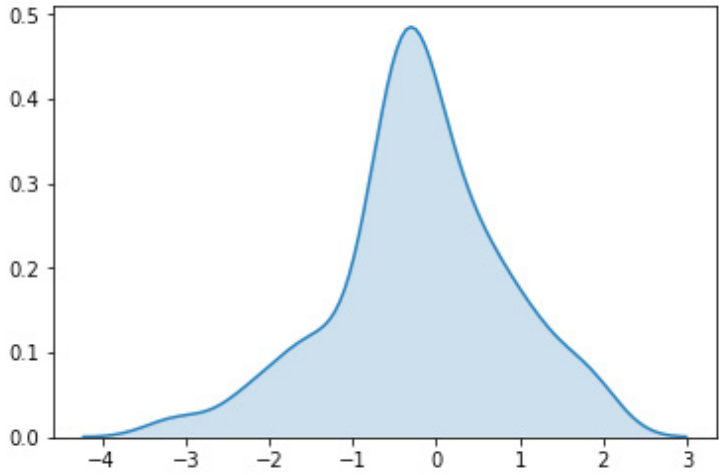

Figure 4.23: KDE for univariate distribution

Plotting Bivariate Distributions

For visualizing **bivariate distributions**, we will introduce three different plots. The first two plots use the **`jointplot()`** function, which creates a multi-panel figure that shows both the joint relationship between both variables and the corresponding marginal distributions.

A scatter plot shows each observation as points on the **`x`** and **`y`** axis. Additionally, a histogram for each variable is shown:

```
import pandas as pd
import seaborn as sns
```

```
data = pd.read_csv("data/salary.csv")
sns.set(style="white")
sns.jointplot(x="Salary", y="Age", data=data)
```

The scatter plot with marginal histograms is shown in the following diagram:


Figure 4.28: Scatter plot with marginal histograms

Figure 4.24: Scatter plot with marginal histograms

It is also possible to use the KDE procedure to visualize bivariate distributions. The joint distribution is shown as a contour plot, as demonstrated in the following code:

```
sns.jointplot('Salary', 'Age', data=subdata, kind='kde', xlim=(0, 500000), ylim=(0,
100))
```

The result is shown in the following diagram:


Figure 4.25: Contour plot

Figure 4.25: Contour plot

Visualizing Pairwise Relationships

For visualizing multiple pairwise bivariate distributions in a dataset, Seaborn offers the **`pairplot()`** function. This function creates a matrix where off-diagonal elements visualize the relationship between each pair of variables and the diagonal elements show the marginal distributions.

The following example gives us a better understanding of this:

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
mydata = pd.read_csv("data/basic_details.csv")
sns.set(style="ticks", color_codes=True)
g = sns.pairplot(mydata, hue="Groups")
```

The pair plot, also called a **correlogram**, is shown in the following diagram. Scatter-plots are shown for all variable pairs on the off-diagonal, while KDEs are shown on the diagonal. Groups are highlighted by different colors:


Figure 4.26: Seaborn pair plot

Figure 4.26: Seaborn pair plot

Violin Plots

A different approach to visualizing statistical measures is by using **violin plots**. They combine box plots with the kernel density estimation procedure that we described previously. It provides a richer description of the variable's distribution. Additionally, the quartile and whisker values from the box plot are shown inside the violin.

The following example demonstrates the usage of violin plots:

```
import pandas as pd
import seaborn as sns
data = pd.read_csv("data/salary.csv")
sns.set(style="whitegrid")
sns.violinplot('Education', 'Salary', hue='Gender', data=data, split=True, cut=0)
```

The result looks as follows:

 Figure 4.31: Seaborn violin plot

Figure 4.27: Seaborn violin plot

Activity 23: Comparing IQ Scores for Different Test Groups by Using a Violin Plot

In this activity, we will compare IQ scores among different test groups by using the violin plot that's provided by Seaborn's library:

1. Use pandas to read the data located in the subfolder data.
2. Access the data of each group in the column, convert them into a list, and assign this list to the variables of each respective group.
3. Create a pandas DataFrame using the preceding data by using the data of each respective group.
4. Create a box plot for each of the IQ scores of different test groups using Seaborn's **`violinplot`** function.
5. Use the **`whitegrid`** style, set the context to talk, and remove all axes spines, except the one on the bottom. Add a title to the plot.
6. After executing the preceding steps, the final output should look as follows:

 Figure 4.28: Violin plot showing IQ scores of different groups

Figure 4.28: Violin plot showing IQ scores of different groups

Note:

The solution for this activity can be found on page 297.

Multi-Plots in Seaborn

In the previous topic, we introduced a multi-plot, namely the pair plot. In this topic, we want to talk about a different way to create flexible multi-plots.

FacetGrid

The **FacetGrid** is useful for visualizing a certain plot for multiple variables separately. A FacetGrid can be drawn with up to three dimensions: `row`, `col`, and `hue`. The first two have the obvious correspondence to the rows and columns of an array. The `hue` is the third dimension and shown with different colors. The `FacetGrid` class has to be initialized with a DataFrame and the names of the variables that will form the row, column, or hue dimensions of the grid. These variables should be **categorical** or **discrete**.

`seaborn.FacetGrid(data, row, col, hue, …)` initializes a multi-plot grid for plotting conditional relationships.

Here are some interesting parameters:

- `data`: A tidy ("long-form") DataFrame where each column corresponds to a variable and each row corresponds to an observation
- `row, col, hue`: Variables that define subsets of the given data, which will be drawn on separate facets in the grid
- `sharex, sharey` (optional): Share `x`/`y` axes across rows/columns
- `height` (optional): Height (in inches) of each facet

Initializing the grid does not draw anything on them yet. For visualizing data on this grid, the `FacetGrid.map()` method has to be used. You can provide any plotting function and the name(s) of the variable(s) in the data frame to plot.

`FacetGrid.map(func, *args, **kwargs)` applies a plotting function to each facet of the grid.

Here are the parameters:

- `func`: A plotting function that takes data and keyword arguments.
- `*args`: The column names in data that identify variables to plot. The data for each variable is passed to `func` in the order the variables are specified in.
- `**kwargs`: Keyword arguments that are passed to the plotting function.

The following example visualizes FacetGrid with scatter plot:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
data = pd.read_csv("data/salary.csv")
g = sns.FacetGrid(subdata, col='District')
g.map(plt.scatter, 'Salary', 'Age')
```


Figure 4.29: FacetGrid with scatter plots

Figure 4.29: FacetGrid with scatter plots

Activity 24: Top 30 YouTube Channels

In this activity, we will visualize the total number of subscribers and the total number of views for the top 30 YouTube channels by using the `FacetGrid()` function that's provided by the Seaborn library. Visualize the given data using a FacetGrid with two

columns. The first column should show the number of subscribers for each YouTube channel, whereas the second column should show the number of views. Following are the steps to implement this activity:

1. Use pandas to read the data located in the subfolder data.
2. Access the data of each group in the column, convert them into a list, and assign this list to variables of each respective group.
3. Create a pandas DataFrame using the preceding data by using the data of each respective group.
4. Create a FacetGrid with two columns to visualize the data.
5. After executing the preceding steps, the final output should look as follows:

 Figure 4.30: Subscribers and Views of top 30 YouTube channels

Figure 4.30: Subscribers and Views of top 30 YouTube channels

Note:

The solution for this activity can be found on page 299.

Regression Plots

Many datasets contain multiple quantitative variables, and the goal is to find a relationship among those variables. We previously mentioned a few functions that show the joint distribution of two variables. It can be helpful to estimate relationships between two variables. We will only cover linear regression in this topic; however, Seaborn provides a wider range of regression functionality if needed.

To visualize linear relationships, determined through linear regression, the **`regplot()`** function is offered by Seaborn. The following code snippet gives a simple example:

```
import numpy as np
import seaborn as sns
x = np.arange(100)
y = x + np.random.normal(0, 5, size=100)
sns.regplot(x, y)
```

The **`regplot()`** function draws a scatter plot, a regression line, and a 95% confidence interval for that regression, as shown in the following diagram:

 Figure 4.31: Seaborn regression plot

Figure 4.31: Seaborn regression plot

Activity 25: Linear Regression

In this activity, we will use a regression plot to visualize the linear relationships:

1. Use pandas to read the data located in the subfolder data.

2. Filter the data so that you end up with samples containing a body mass and a maximum longevity. Only consider samples for the Mammalia class and a body mass below 200,000.
3. Create a regression plot to visualize the linear relationship of the variables.
4. After executing the preceding steps, the output should look as follows:


Figure 4.32: Linear regression for animal attribute relations

Figure 4.32: Linear regression for animal attribute relations

Note:

The solution for this activity can be found on page 300.

Squarify

At this point, we will briefly talk about **tree maps**. Tree maps display hierarchical data as a set of nested rectangles. Each group is represented by a rectangle, of which its area is proportional to its value. Using color schemes, it is possible to represent hierarchies: groups, subgroups, and so on. Compared to pie charts, tree maps efficiently use space. Matplotlib and Seaborn do not offer tree maps, and so the **Squarify** library that is built on top of Matplotlib is used. Seaborn is a great addition for creating color palettes.

The following code snippet is a basic tree map example. It requires the Squarify library:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import squarify
colors = sns.light_palette("brown", 4)
squarify.plot(sizes=[50, 25, 10, 15], label=["Group A", "Group B", "Group C", "Group D"], color=colors)
plt.axis("off")
plt.show()
```

The result is shown in the following diagram:

Figure 4.44: Tree map

Figure 4.33: Tree map

Activity 26: Water Usage Revisited

In this activity, we will use a tree map to visualize the percentage of water that's used for different purposes:

1. Use pandas to read the data located in the subfolder data.
2. Use a tree map to visualize the water usage.
3. Show the percentages for each tile and add a title.

4. After executing the preceding steps, the expected output should be as follows:

 Figure 4.34: Tree map of water usage

Figure 4.34: Tree map of water usage

Note:

The solution for this activity can be found on page 301.

Summary

In this chapter, we demonstrated how Seaborn helps create visually appealing figures. We discussed various options for controlling figure aesthetics, such as figure style, controlling spines, and setting the context of visualizations. We talked about color palettes in detail. Further visualizations were introduced for visualizing univariate and bivariate distributions. Moreover, we discussed FacetGrids, which can be used for creating multi-plots, and regression plots as a way to analyze the relationships between two variables. Finally, we discussed the Squarify library, which is used to create tree maps. In the next chapter, we will show you how to visualize geospatial data in various ways by using the Geoplotlib library.

Chapter 5

Plotting Geospatial Data

Learning Objectives

By the end of this chapter, you will be able to:

- Utilize Geoplotlib to create stunning geographical visualizations
- Identify the different types of geospatial charts
- Demonstrate datasets containing geospatial data for plotting
- Explain the importance of plotting geospatial information

In this chapter, we will use the geoplotib library to visualize different geospatial data.

Introduction

Geoplotlib is an open source Python library for geospatial data visualizations. It has a wide range of geographical visualizations and supports hardware acceleration. It also provides performance rendering for large datasets with millions of data points. As discussed in the earlier chapters, Matplotlib provides the ways to visualize geographical data. However, Matplotlib is not designed for this task, as its interfaces are complex and inconvenient to use. Matplotlib also restricts the ways in which geographical data can be displayed. The **Basemap** and **Cartopy** libraries enable it so that you can plot on a world map. However, these packages do not support drawing on **map tiles**.

Geoplotlib, on the other hand, was designed exactly for this purpose and therefore it not only provides **map tiles** but also allows for interactivity and simple animations. It provides a simple interface that allows access to powerful geospatial visualizations.

Note

For a better understanding of the available features of Geoplotlib, you can visit the following link: https://github.com/andrea-cuttone/geoplotlib/wiki/User-Guide.

To understand the concepts, design, and implementation of Geoplotlib, let's take a brief look at its conceptual architecture. The two inputs that are fed to Geoplotlib are your **data sources** and the **map tiles**. The map tiles, as we'll see later, can be replaced by different providers. The outputs describe the possibilities to not only render images inside the Jupyter Notebooks but also to work in an interactive window that allows for **zooming** and **panning** the maps. The schema of the components of **Geoplotlib** looks like this:

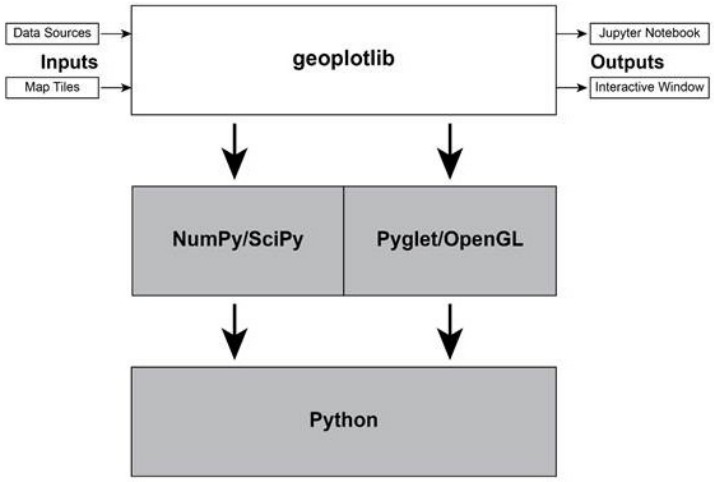

Figure 5.1: Conceptual architecture of geoplotlib

Geoplotlib uses the concept of layers that can be placed on top one another, providing a powerful interface for even complex visualizations. It comes with several common visualization layers that are easy to set up and use.

From the preceding diagram, we can see that **geoplotlib** is built on top of **NumPy**/**SciPy**, and **Pyglet**/**OpenGL**. These libraries take care of numerical operations and rendering. Both components are based on Python, thus enabling the usage of the full Python ecosystem.

The Design Principles of Geoplotlib

Taking a closer look at the internal design of Geoplotlib, we can see that it is built around three design principles:

- **Simplicity**: Looking at the example provided here, we can quickly see that Geoplotlib abstracts away the complexity of plotting map tiles and the already provided layers such as **dot-density** and **histogram**. It has a simple API that provides common visualizations. These visualizations can be created using custom data with only a few lines of code. If our dataset comes with **lat** and **lon** columns, we can display those **datapoints** as dots on a map with five lines of code, like this:

  ```
  import geoplotlib

  from geoplotlib.utils import read_csv dataset =
  read_csv('./data/poaching_points_cleaned.csv')

  geoplotlib.dot(dataset)

  geoplotlib.show()
  ```

  In addition to this, everyone who's used Matplotlib before will have no problems understanding the syntax of Geoplotlib, because its syntax is inspired by the syntax of Matplotlib.

- **Integration**: Geoplotlib visualizations are purely Python-based. This means that the generic Python code can be executed and other libraries such as **pandas** can be used for **data wrangling** purposes. We can manipulate and enrich our datasets using pandas **DataFrames** and later simply convert them into a Geoplotlib **`DataAccessObject`**, which we need for optimum compatibility, like this:

  ```
  import pandas as pd

  from geoplotlib.utils import DataAccessObject

  pd_dataset = pd.read_csv('./data/poaching_points_cleaned.csv')

  # data wrangling with pandas DataFrames here
  ```

```
dataset = DataAccessObject(pd_dataset)
```

Geoplotlib fully integrates into the Python ecosystem. This enables us to even plot geographical data inline inside our Jupyter Notebooks. This possibility allows us to design our visualizations quickly and iteratively.

- **Performance**: As we mentioned before, Geoplotlib is able to handle large amounts of data due to the usage of NumPy for accelerated numerical operations and **OpenGL** for accelerated graphical rendering.

Geospatial Visualizations

Choropleth plot, **Voronoi tessellation**, and **Delaunay triangulation** are a few of the geospatial visualizations that will be used in this chapter. The explanation for each of them is provided here:

Choropleth Plot

This kind of geographical plot displays areas such as states of a country in a shaded or colored manner. The shade or color is determined by a single or set of data points. It gives an abstract view of a geographical area to visualize the relations and differences between the different areas. In Figure 5.21, we can see that the shade of each state of the US is determined by the percentage of obesity. The darker the shade, the higher the percentage.

Voronoi tessellation

In a **Voronoi tessellation**, each pair of data points is basically separated by a line that has the same distance from both data points. The separation creates cells that, for every given point, marks which data point is closer. The closer the data points, the smaller the cells.

Delaunay triangulation

A **Delaunay triangulation** is related to the Voronoi tessellation. When connecting each data point to every other data point that shares an edge, we end up with a plot that is triangulated. The closer the data points, the smaller the triangles will be. This gives us a visual clue about the density of points in specific areas. When combined with color gradients, we get insights about points of interests, which can be compared with a heatmap.

Exercise 6: Visualizing Simple Geospatial Data

In this exercise, we'll be looking at the basic usage of Geoplotlib's plot methods for **DotDensity**, **Histograms**, and **Voronoi** diagrams. For this, we will make use of the data of various poaching incidents that have taken place all over the world:

1. Open the Jupyter Notebook **`exercise06.ipynb`** from the **`Lesson05`** folder to implement this exercise.

   To do that, you need to navigate to the path of this file. In the command-line terminal, type: **`jupyter-lab`**

2. By now, you should be familiar with the process of working with Jupyter Notebooks.

3. Open the **`exercise06.ipynb`** file.

4. As always, first, we want to import the dependencies that we will need. In this case, we will work without pandas, since **`geoplotlib`** has its own **`read_csv`** method that enables us to read a .csv file into a **`DataAccessObject`**:

   ```
   # importing the necessary dependencies
   import geoplotlib
   from geoplotlib.utils import read_csv
   ```

5. The data is being loaded in the same way as with the pandas **`read_csv`** method:

   ```
   dataset = read_csv('./data/poaching_points_cleaned.csv')
   ```

Note

The preceding dataset can be found here: https://bit.ly/2Xosg2b.

6. The dataset is stored in a **`DataAccessObject`** class that's provided by Geoplotlib. It does not have the same capabilities as pandas **DataFrames**. It's meant for the simple and quick loading of data so that you can create a visualization. If we print out this object, we can see the difference better. It gives us a basic overview of what columns are present and how many rows the dataset has:

```
# looking at the dataset structure
Dataset
```

The following figure shows the output of the preceding code:

```
DataAccessObject(['id_report', 'date_report', 'description', 'created_date', 'lat', 'lon'] x 268)
```

Figure 5.2: Dataset structure

As we can see in the preceding screenshot, the dataset consists of 268 rows and six columns. Each row is uniquely identified by **`id_report`**. The **`date_report`** column states on what date the poaching incident took place. On the other hand, the **`created_date`** column states the date on which the report was created. The description column provides the basic information about the incident. The **`lat`** and **`lon`** columns state the geographical location of the place where poaching took place.

7. Geoplotlib is compatible with pandas DataFrames as well. If you need to do some pre-processing with the data, it might make sense to use pandas right away:

```
# csv import with pandas
import pandas as pd
pd_dataset = pd.read_csv('./data/poaching_points_cleaned.csv')
pd_dataset.head()
```

The following figure shows the output of the preceding code:

	id_report	date_report	description	created_date	lat	lon
0	138	01/01/2005 12:00:00 AM	Poaching incident	2005/01/01 12:00:00 AM	-7.049359	34.841440
1	4	01/20/2005 12:00:00 AM	Poaching incident	2005/01/20 12:00:00 AM	-7.650840	34.480010
2	43	01/20/2005 12:00:00 AM	Poaching incident	2005/02/20 12:00:00 AM	-7.843202	34.005704
3	98	01/20/2005 12:00:00 AM	Poaching incident	2005/02/21 12:00:00 AM	-7.745846	33.948526
4	141	01/20/2005 12:00:00 AM	Poaching incident	2005/02/22 12:00:00 AM	-7.876673	33.690167

Figure 5.3: First five entries of the dataset

Note

*Geoplotlib requires your dataset to have **`lat`** and **`lon`** columns. These columns are the geographical data for latitude and longitude, and are used to determine how to plot.*

8. To start with, we'll be using a simple **DotDensityLayer** that will plot each row of our dataset as a single point on a map. Geoplotlib comes with the **`dot`** method, which creates this visualization without further configurations:

Note

*After setting up DotDensityLayer here, we need to call the **`show`** method, which will render the map with the given layer.*

```
# plotting our dataset with points
geoplotlib.dot(dataset)
geoplotlib.show()
```

The following figure shows the output of the preceding code:

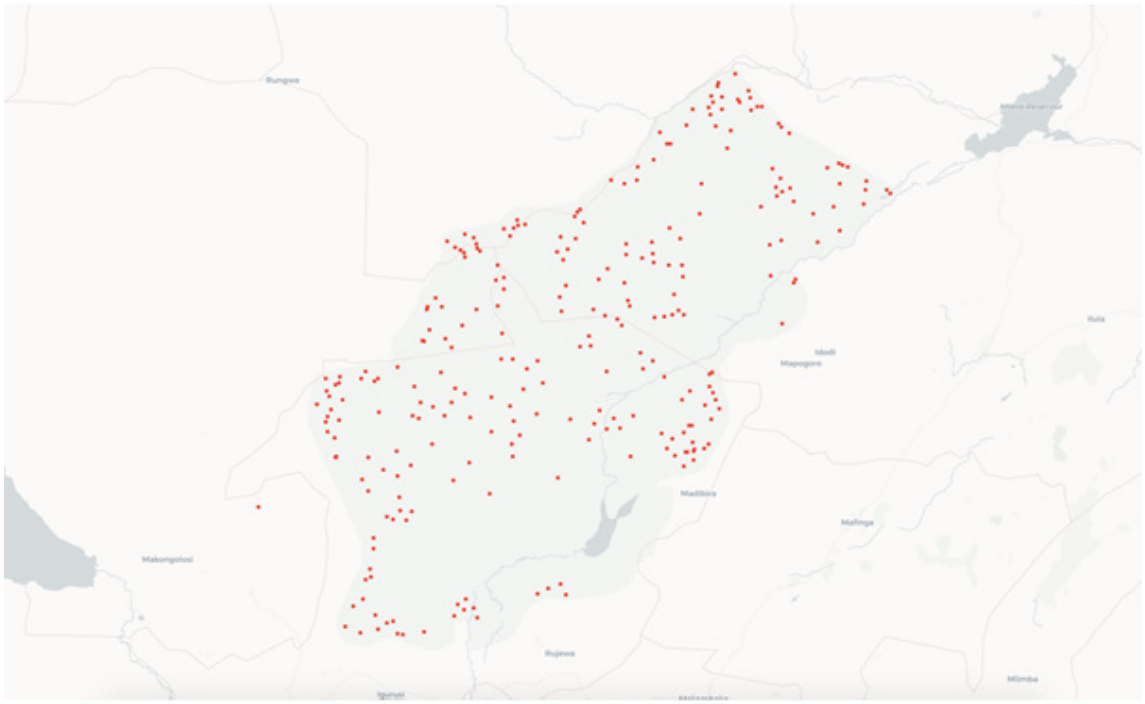

Figure 5.4: Dot-density visualization of poaching points

Only looking at the **`lat`** and **`lon`** values in the dataset won't give us a very good idea. We're not able to draw conclusions and get insights into our dataset without visualizing our data points on a map. When looking at the rendered map, we can see that there are some more popular spots with more dots and some less popular ones with fewer dots.

9. We now want to look at the point density some more. To better visualize the density, we have a few options. One of them is to use a **histogram plot**. We can define a **`binsize`**, which will allow us to set the size of the hist bins in our visualization. **Geoplotlib** provides the **`hist`** method, which will create a **Histogram Layer** on top of our map tiles:

```
# plotting our dataset as a histogram
geoplotlib.hist(dataset, binsize=20)
geoplotlib.show()
```

The following figure shows the output of the preceding code:

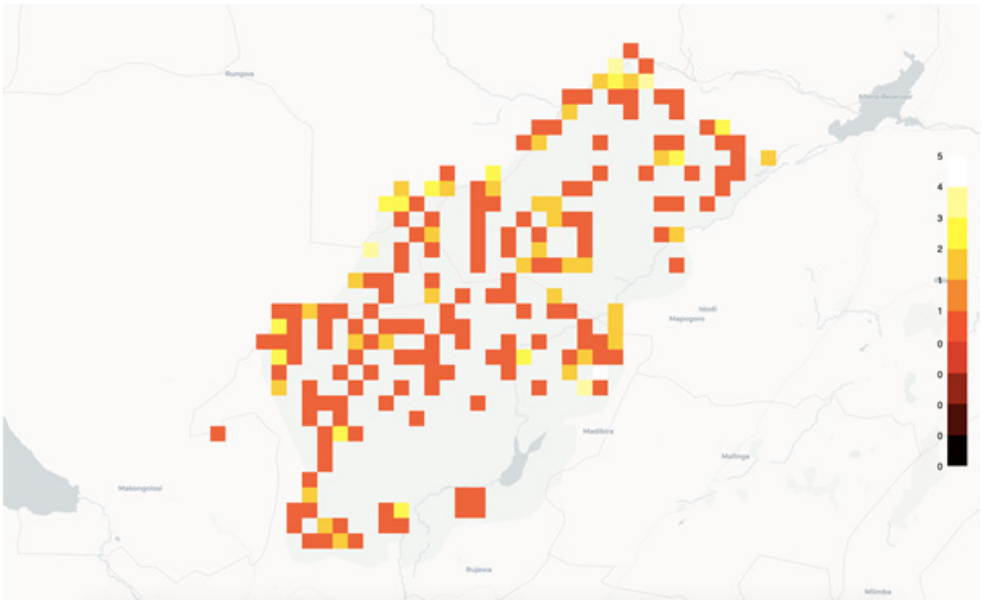

Figure 5.5: Histogram visualization of poaching points

Histogram plots give us a better understanding of the **density distribution** for our dataset. Looking at the final plot, we can see that there are some hotspots for poaching. This also highlights the areas without any poaching incidents.

10. **Voronoi plots** are also good for visualizing the density of data points. Voronoi introduces a little bit more complexity with several parameters such as **`cmap`**, **`max_area`**, and **`alpha`**. Here, **`cmap`** denotes the color of the map, **`alpha`** denotes the color of the alpha, and **`max_area`** denotes a constant that determines the color of the **Voronoi areas**. They are useful if you want to better fit your visualization into the data:

```
# plotting a voronoi map
geoplotlib.voronoi(dataset, cmap='Blues_r', max_area=1e5, alpha=255)
geoplotlib.show()
```

The following figure shows the output of the preceding code:

Figure 5.6: Voronoi visualization of poaching points

If we compare this Voronoi visualization with the histogram plot, we can see that one area that draws a lot of attention. The center right edge of the plot shows quite a large dark blue area with an even darker center, something that could've easily been overlooked with the histogram plot.

Congratulations! We just covered the basics of Geoplotlib. It has many more methods, but they all have a similar API that makes using the other methods simple. Since we looked at some of the very basic visualizations, it's now up to you to solve the first activity.

Activity 27: Plotting Geospatial Data on a Map

In this activity, we will take the previously learned skills of plotting data with Geoplotlib and apply them to the task of finding the dense areas within the cities in Europe that have a population of more than 100,000:

1. Open the Jupyter Notebook **`activity27.ipynb`** from the **`Lesson05`** folder to implement this activity.
2. Before we can start working with the data, we will need to import the dependencies.
3. Load the dataset using **`pandas`**.
4. After loading the dataset, we want to list all the datatypes that are present in it.
5. Once we can see that our datatypes are correct, we will map our **`Latitude`** and **`Longitude`** columns to **`lat`** and **`lon`**.
6. Now, plot the data points in a dot plot.
7. To get one step closer to solving our given task, we need to get the number of cities per country (the first 20 entries) and filter out the countries with a population greater than zero.
8. Plot the remaining data in a dot plot.
9. Again, filter your remaining data for cities with a population greater than 100,000.
10. To get a better understanding of the density of our data points on the map, we want to use a **Voronoi tessellation layer**.

11. Filter down the data even further to only cities in countries such as Germany and Great Britain.

12. Finally, use a **Delaunay triangulation layer** to find the most densely populated areas.

13. Observe the expected output of dot plot:

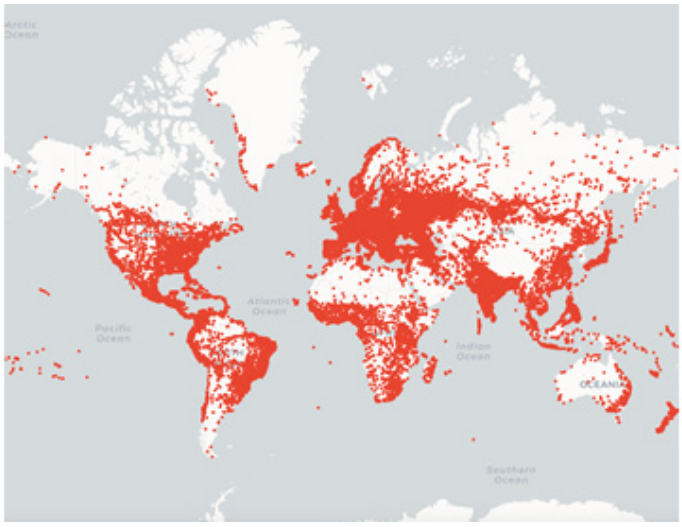

Figure 5.7: Dot-density visualization of the reduced dataset

The following is the expected output of the Voronoi plot:

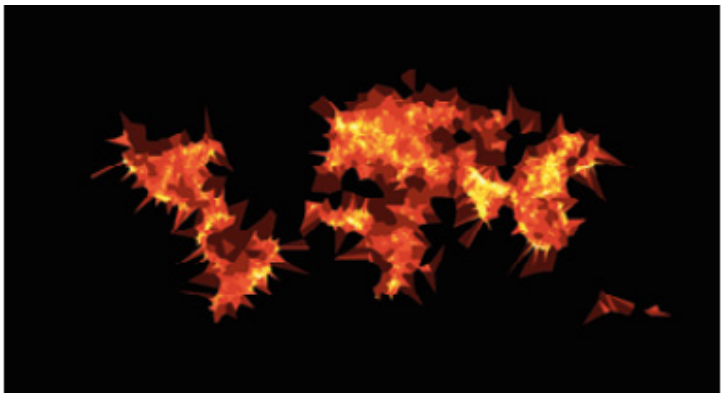

Figure 5.8: Voronoi visualization of densely populated cities

The following is the expected output of the Delaunay triangulation:

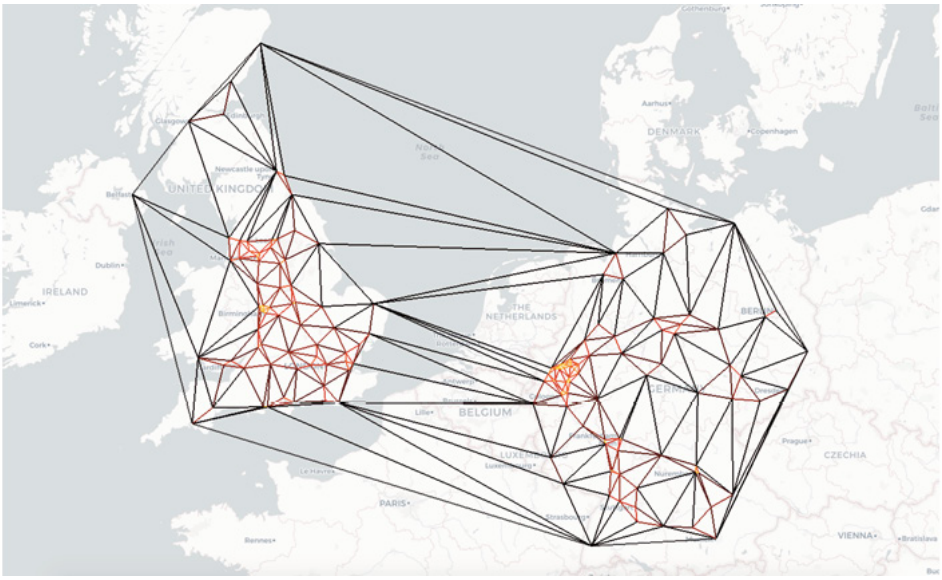

Figure 5.9: Delaunay triangle visualization of cities in Germany and Great Britain

Note:

The solution for this activity can be found on page 303.

Exercise 7: Choropleth Plot with GeoJSON Data

In this exercise, we not only want to work with **GeoJSON** data, but also want to see how we can create a **choropleth visualization**. They are especially useful for displaying statistical variables in shaded areas. In our case, the areas will be the outlines of the states of the USA. Let's create a choropleth visualization with the given GeoJSON data:

1. Open the Jupyter Notebook **`exercise07.ipynb`** from the **`Lesson05`** folder to implement this exercise.
2. Load the dependencies for this exercise:

```
# importing the necessary dependencies
import json
import geoplotlib
from geoplotlib.colors import ColorMap
from geoplotlib.utils import BoundingBox
```

3. Before we create the actual visualization, we need to understand the outlines of our dataset. Since the **`geojson`** method of Geoplotlib only needs a path to the dataset instead of a DataFrame or object, we don't need to load it. However, since we still want to see what kind of data we are handling, we must open the GeoJSON file and load it as a **`json`** object. Given that, we can then access its members by **simple indexing**:

```
# displaying one of the entries for the states
with open('data/National_Obesity_By_State.geojson') as data:
dataset = json.load(data)
first_state = dataset.get('features')[0]
# only showing one coordinate instead of all points
```

```
first_state['geometry']['coordinates'] = first_state['geometry']['coordinates'][0][0]
print(json.dumps(first_state, indent=4))
```

4. The following output describes one of the features that displays the general structure of a GeoJSON file. The properties of interest for us are the NAME, Obesity, and the geometry coordinates:


```
{
    "type": "Feature",
    "properties": {
        "OBJECTID": 1,
        "NAME": "Texas",
        "Obesity": 32.4,
        "Shape__Area": 7672329221282.43,
        "Shape__Length": 15408321.8693326
    },
    "geometry": {
        "type": "Polygon",
        "coordinates": [
            -106.623454789568,
            31.9140391520155
        ]
    }
}
```


Figure 5.10: General structure of the geojson file

Note

Geospatial applications prefer GeoJSON files for persisting and exchanging geographical data.

5. Depending on the information present in the GeoJSON file, we might need to extract some of it for later mappings. For the **obesity database**, we want to extract the names of all the states of the US. The following code does the same:

```
# listing the states in the dataset
with open('data/National_Obesity_By_State.geojson') as data:
dataset = json.load(data)
states = [feature['properties']['NAME'] for feature in dataset.get('features')]
print(states)
```

The following figure shows the output of the preceding code:


```
['Texas', 'California', 'Kentucky', 'Georgia', 'Wisconsin', 'Oregon', 'Virginia', 'Tennessee', 'Louisia
na', 'New York', 'Michigan', 'Idaho', 'Florida', 'Alaska', 'Montana', 'Minnesota', 'Nebraska', 'Washing
ton', 'Ohio', 'Illinois', 'Missouri', 'Iowa', 'South Dakota', 'Arkansas', 'Mississippi', 'Colorado', 'N
orth Carolina', 'Utah', 'Oklahoma', 'Wyoming', 'West Virginia', 'Indiana', 'Massachusetts', 'Nevada', '
Connecticut', 'District of Columbia', 'Rhode Island', 'Alabama', 'Puerto Rico', 'South Carolina', 'Main
e', 'Arizona', 'New Mexico', 'Maryland', 'Delaware', 'Pennsylvania', 'Kansas', 'Vermont', 'New Jersey',
'North Dakota', 'New Hampshire']
```


Figure 5.11: List of all cities in the United States

6. If your GeoJSON file is valid, meaning that it has the expected structure, you can then use the **`geojson`** method of Geoplotlib. By only providing the path to the file, it will plot the coordinates for each feature in a blue color by default:

```
# plotting the information from the geojson file
geoplotlib.geojson('data/National_Obesity_By_State.geojson')
geoplotlib.show()
```

After calling the **`show`** method, the map will show up with a focus on North America. In the following diagram, we can already see the borders of each state:

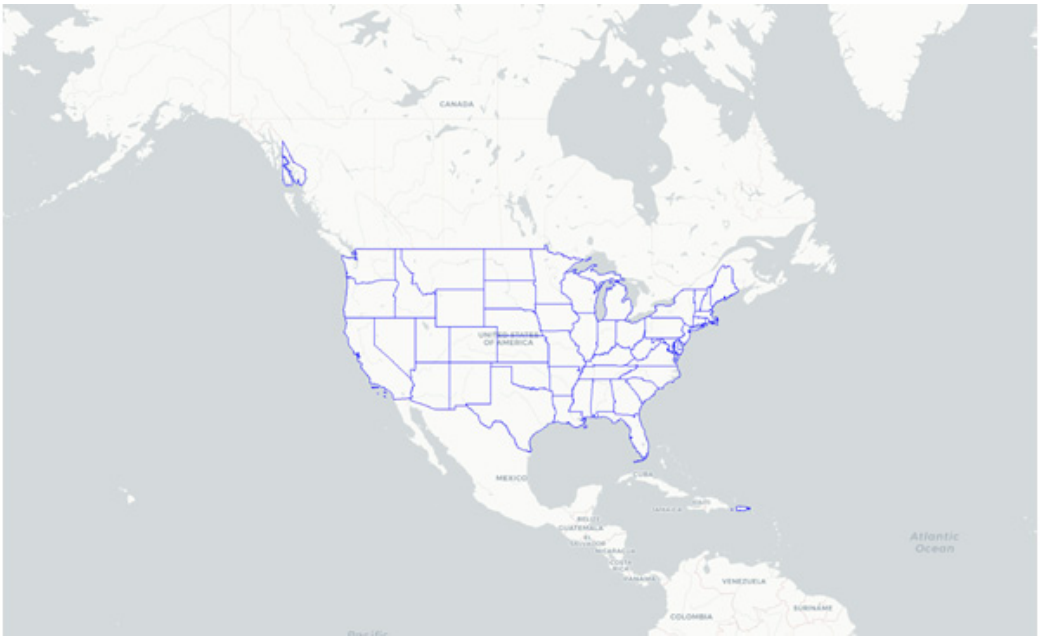

Figure 5.12: Map with outlines of the states plotted

7. To assign a color that represents the obesity of each state, we have to provide the **`color`** parameter to the **`geojson`** method. We can provide different types depending on the use case. Rather than assigning a single value to each state, we want the darkness to represent the percentage of obese people. To do this, we have to provide a method for the **`color`** property. Our method simply maps the **`Obesity`** attribute to a **`ColorMap`** class object with enough levels to allow good distinction:

```
# converting the obesity into a color
cmap = ColorMap('Reds', alpha=255, levels=40)
def get_color(properties):
return cmap.to_color(properties['Obesity'], maxvalue=40,scale='lin')
```

8. We then provide the color mapping to our **`color`** parameter. This, however, won't fill the areas. We therefore also have to set the **`fill`** parameter to **`True`**. In addition, we also want to keep the outlines of our states visible. Here, we can leverage the concept that **Geoplotlib** is layer-based, so we can simply call the same method again, providing a white color and setting the **`fill`** parameter to **`false`**. We also want to make sure that our view is displaying the country **`USA`**. To do this we, again, use one of the constants that are provided by Geoplotlib:

```
# plotting the shaded states and adding another layer which plots the state outlines
in white
# our BoundingBox should focus the USA
geoplotlib.geojson('data/National_Obesity_By_State.geojson', fill=True,
color=get_color)
geoplotlib.geojson('data/National_Obesity_By_State.geojson', fill=False, color=[255,
255, 255, 255])
geoplotlib.set_bbox(BoundingBox.USA)
geoplotlib.show()
```

9. After executing the preceding steps, the expected output is as follows:

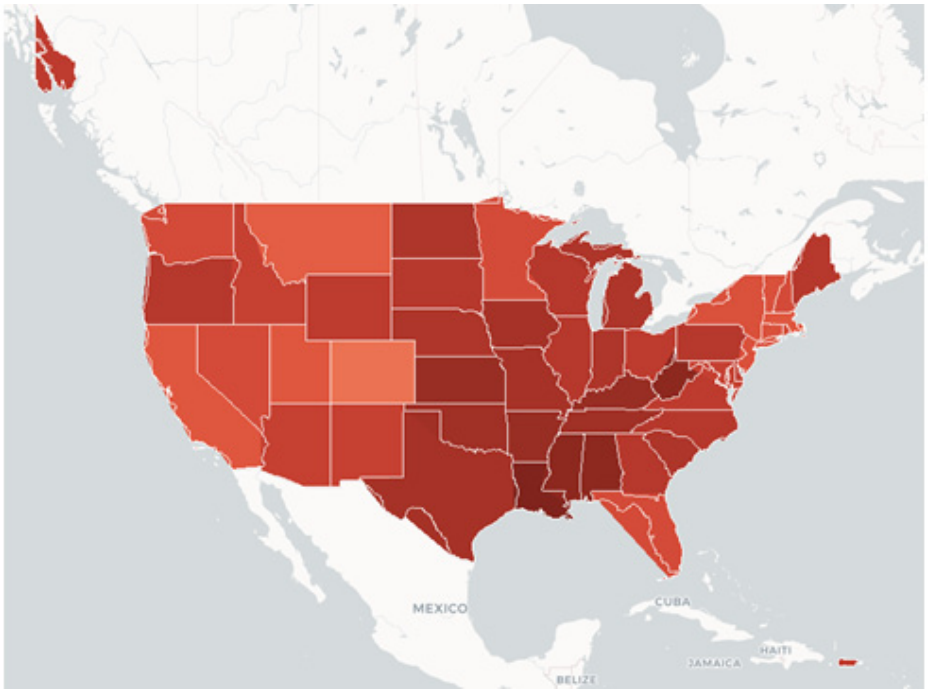

Figure 5.13: Choropleth visualization showing obesity in different states

A new window will open, displaying the country USA with the areas of its states filled with different shades of red. The darker areas represent higher obesity percentages.

Note

To give the user some more information for this plot, we could also use the ***`f_tooltip`*** *parameter to provide a tooltip for each state, thus displaying the name and the percentage of obese people.*

Congratulations! You've already built different plots and visualizations using Geoplotlib. In this exercise, we looked at displaying data from a GeoJSON file and creating a **choropleth plot**.

In the following topics, we will cover more advanced customizations that will give you the tools to create more powerful visualizations.

Tile Providers

Geoplotlib supports the usage of different **tile providers**. This means that any **OpenStreetMap** tile server can be used as a backdrop to our visualization. Some of the popular free tile providers are **Stamen Watercolor**, **Stamen Toner, Stamen Toner Lite, and DarkMatter**.

Changing the tile provider can be done in two ways:

- **Make use of built-in tile providers**

  Geoplotlib contains a few built-in tile providers with shortcuts. The following code shows how to use it:

  ```
  geoplotlib.tiles_provider('darkmatter')
  ```

- **Provide a custom object to the tiles_provider method**

By providing a custom object to Geoplotlib's **`tiles_provider()`** method, you will not only get the access to the **`url`** from which the map tiles are being loaded, but also see the **`attribution`** displayed in the lower right corner of the visualization. We are also able to set a distinct **caching directory** for the downloaded tiles. The following code shows how to provide a custom object:

```
geoplotlib.tiles_provider({
'url': lambda zoom, xtile, ytile:
'http://a.tile.stamen.com/watercolor/%d/%d/%d.png' % (zoom, xtile, ytile),
'tiles_dir': 'tiles_dir',
'attribution': 'Python Data Visualization | Packt'
})
```

The caching in **`tiles_dir`** is mandatory, since, each time the map is scrolled or zoomed into, we query new map tiles if they are not already downloaded. This can lead to the tile provider refusing your request due to many requests in a short period of time.

In the following exercise, we'll have a quick look at how to switch the map tile provider. It might not seem powerful at first, but it can take your visualizations to the next level if leveraged correctly.

Exercise 8: Visually Comparing Different Tile Providers

This quick exercise will teach you how to switch the map tile provider for your visualizations. Geoplotlib provides mappings for some of the available and most popular map tiles. However, we can also provide a custom object that contains the **`url`** of some tile providers:

1. Open the Jupyter Notebook **`exercise08.ipynb`** from the **`Lesson05`** folder to implement this exercise.

   To do that, you need to navigate to the path of this file and in the command-line terminal type: **`jupyter-lab`**

2. We won't use any dataset in this exercise, since we want to focus on the map tiles and tile providers. Therefore, the only import we need to do is **`geoplotlib`** itself:

   ```
   # importing the necessary dependencies
   import geoplotlib
   ```

3. We know that Geoplotlib has a layers approach to **plotting**. This means that we can simply display the map tiles without adding any plotting layer on top:

   ```
   # displaying the map with the default tile provider
   geoplotlib.show()
   ```

   The following figure shows the output of the preceding code:

Figure 5.14: World map with default tile provider

This will display a completely empty world map, since we haven't specified any tile provider. By default, it will use the **CartoDB Positron** map tiles.

4. Geoplotlib provides several **shorthand accessors** to common map tile providers. The `tiles_provider` method allows us to simply provide the name of the provider:

```
# using map tiles from the dark matter tile provider
geoplotlib.tiles_provider('darkmatter')
geoplotlib.show()
```

The following figure shows the output of the preceding code:

Figure 5.15: World map with darkmatter map tiles

In this example, we used the `darkmatter` map tiles. As you can see, those are very dark and will make your visualizations pop out.

Note

*We can also use different map tiles such as **watercolor**, **toner**, **toner-lite**, and **positron** in a similar way.*

5. When using tile providers that are not covered by **`geoplotlib`**, we can pass a **custom object** to the **`tiles_provider`** method. It maps the current viewport information to the **`url`**. The **`tiles_dir`** parameter defines where the tiles should be cached. When changing the **`url`**, you also have to change **`tiles_dir`** to see the changes immediately. The **`attribution`** gives you the option to display custom text in the right lower corner:

```
# using custom object to set up tile provider
geoplotlib.tiles_provider({
'url': lambda zoom, xtile, ytile: 'http://a.tile.openstreetmap.fr/hot/%d/%d/%d.png' % (zoom, xtile, ytile),
'tiles_dir': 'custom_tiles',
'attribution': 'Custom Tiles Provider - Humanitarian map style | Packt Courseware'
})
geoplotlib.show()
```

The following figure shows the output of the preceding code:

Figure 5.16: Humanitarian map tiles from custom tile providers object

Some map tile providers have strict request limits, so it might lead to warning messages if you're zooming too fast.

Congratulations! You now know how to change the tile provider to give your visualization one more layer of customizability. This also introduces us to another layer of complexity. It all depends on the concept of our final product and whether we want to use the "default" map tiles or some artistic map tiles.

The next topic will cover how to create custom layers that can go far beyond the ones we have described in this book. We'll look at the basic structure of the **`BaseLayer`** class and what it takes to create a custom layer.

Custom Layers

Now that we have covered the basics of visualizing geospatial data with the built-in layers, and the methods to change the tile provider, we will now focus on defining our own custom layers. **Custom layers** allow you to create more complex data visualizations. They also help with adding more interactivity and animation to them. Creating a custom layer starts by defining a new class that extends the **`BaseLayer`**

class that's provided by Geoplotlib. Besides the `__init__` method that initializes the class level variables, we also have to at least extend the `draw` method of the already provided `BaseLayer` class.

Depending on the nature of your visualization, you might also want to implement the **invalidate** method, which takes care of map projection changes such as zooming into your visualization. Both the `draw` and `invalidate` methods receive a `Projection` object that takes care of the latitude and longitude mapping on our two-dimensional viewport. These mapped points can be handed to an instance of a `BatchPainter` object that provides primitives such as points, lines, and shapes to draw those coordinates onto your map.

Note

Since Geoplotlib operates on OpenGL, this process is highly performant and can even draw complex visualizations quickly. For more examples on how to create custom layers, visit the following GitHub repository of Geoplotlib: https://github.com/andrea-cuttone/geoplotlib/tree/master/examples.

Activity 28: Working with Custom Layers

In this activity, we will take a look at creating a custom layer that will allow you to not only display geospatial data but also animate your data points over time. We'll get a deeper understanding of how Geoplotlib works and how the layers are created and drawn. Our dataset does not only contain **spatial** but also **temporal** information, which enables us to plot the flights over time on our map:

1. Open the Jupyter Notebook `activity28.ipynb` from the `Lesson05` folder to implement this activity.
2. First, make sure to import the necessary dependencies.
3. Load the `flight_tracking.csv` dataset using `pandas`.
4. Have a look at the dataset and its features.
5. Since our dataset has columns that are named `Latitude` and `Longitude` instead of `lat` and `lon`, rename those columns to their short versions.
6. Our custom layer will animate the flight data, which means that we need to work with the `timestamp` of our data. **Date** and **time** are two separate columns, so we need to merge these columns. Use the provided `to_epoch` method and create a new **timestamp** column.
7. Create a new `TrackLayer` that extends Geoplotlib's `BaseLayer`.
8. Implement the methods `__init__`, `draw`, and `bbox` for the `TrackLayer`. Use the provided `BoundingBox` to focus on Leeds when calling the `TrackLayer`.
9. After executing the preceding steps, the expected output is as follows:

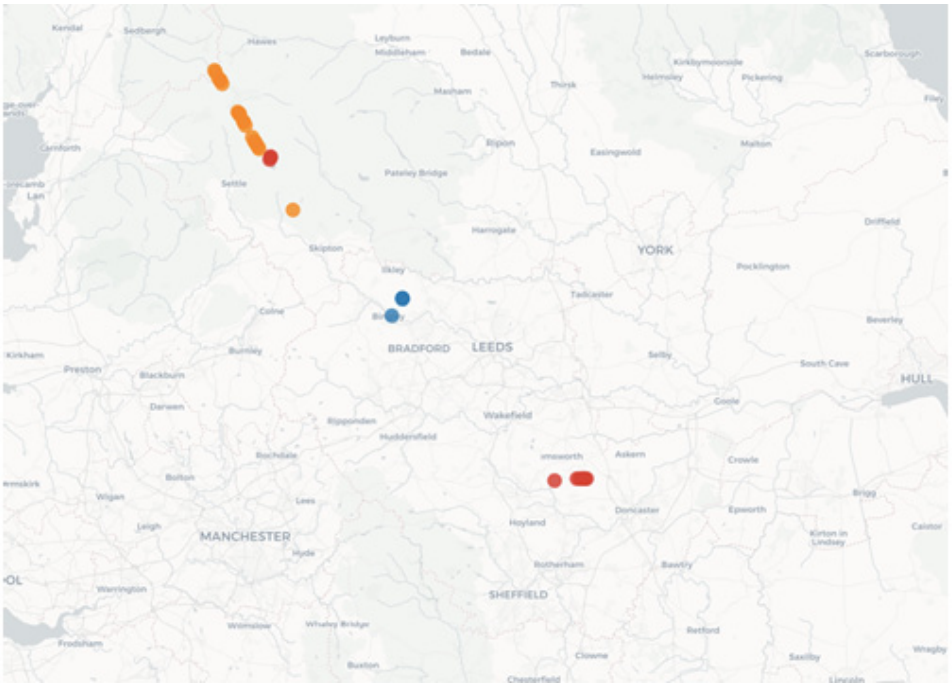

Figure 5.17: Flight tracking visualization focused on Leeds

Note:

The solution for this activity can be found on page 311.

Summary

In this chapter, we covered the basic and advanced concepts and methods of Geoplotlib. It gave us a quick insight into the internal processes and how to practically apply the library to our own problem statements. Most of the time, the built-in plots should suit your needs pretty well. Once you're interested in having animated or even interactive visualizations, you will have to create custom layers that enable those features.

In the following chapter, we'll get some hands-on experience with the Bokeh library and build visualizations that can easily be integrated into web pages. Once we have finished using Bokeh, we'll conclude this chapterware with a chapter that gives you the opportunity to work with a new dataset and a library of your choice so that you can come up with your very own visualization. This will be the last step in consolidating your journey in data visualization with Python.

Chapter 6

Making Things Interactive with Bokeh

Learning Objectives

By the end of this chapter, you will be able to:

- Use Bokeh to create insightful web-based visualizations
- Explain the difference between the two interfaces for plotting
- Identify when to use the Bokeh Server
- Create interactive visualizations

In this chapter, we will design interactive plots using the Bokeh library.

Introduction

Bokeh has been around since 2013, with version 1.0.4 being released in 2018. It targets modern web browsers to present interactive visualizations to users rather than static images. The following are some of the features of Bokeh:

- **Simple visualizations**: Through its different interfaces, it targets users of many skill levels, thus providing an API for quick and simple visualizations but also more complex and extremely customizable ones.
- **Excellent animated visualizations**: It provides high performance and therefore can work on large or even streaming datasets, which makes it the go-to choice for animated visualizations and data analysis.
- **Inter-visualization interactivity**: This is a web-based approach, where it's easy to combine several plots and create unique and impactful dashboards with visualizations that can be interconnected to create inter-visualization interactivity.
- **Supports multiple languages**: Other than Matplotlib and geoplotlib, Bokeh has libraries not only for Python but also JavaScript itself, and several other popular languages.
- **Mulitiple ways to perform a task**: The previously mentioned interactivity can be added in several ways. The simplest inbuilt way is the possibility to zoom and pan around in your visualization, and this already gives the user better control of what they want to see. Other than that, we can also empower the users to filter and transform the data.
- **Beautiful chart styling**: The tech stack is based on Tornado in the backend and powered by D3 in the frontend, which explains the beautiful default styling of the charts.

Since we are using Jupyter Notebook throughout this book, it's worth mentioning that Bokeh, including its interactivity, is natively supported in Notebook.

Concepts of Bokeh

The basic concept of Bokeh, in some ways, is comparable to that of Matplotlib. In Bokeh, we have a figure as our root element, which has sub-elements such as a title, an axis, and **glyphs**. Glyphs have to be added to a figure, which can take on different shapes such as circles, bars, and triangles to represent a figure. The following hierarchy shows the different concepts of Bokeh:

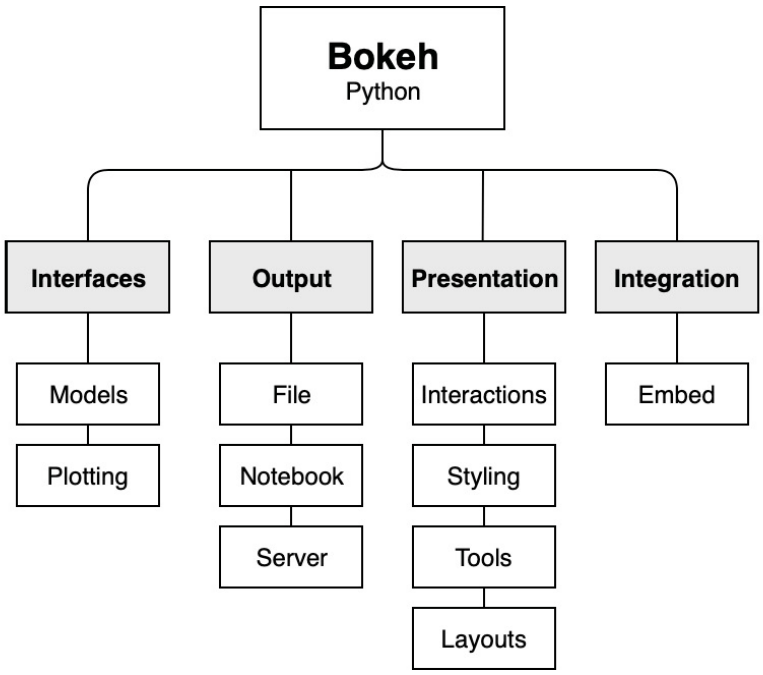

Figure 6.1: Concepts of Bokeh

Interfaces in Bokeh

The interface-based approach provides different levels of complexity for users that either simply want to create some basic plots with very few customizable parameters or the users who want full control over their visualizations and want to customize every single element of their plots. This layered approach is divided into two levels:

- **Plotting**: This layer is customizable.
- **Models interface**: This layer is complex and provides an open approach to designing charts.

Note

The models interfaces are the basic building blocks for all plots.

The following are the two levels of the layered approach in interfaces:

- **bokeh.plotting**

  This mid-level interface has a somewhat comparable API to Matplotlib. The workflow is to create a figure and then enrich this figure with different glyphs that render data points in the figure. Comparable to Matplotlib, the composition of sub-elements such as axes, grids, and the **Inspector** (they provide basic ways of exploring your data through zooming, panning, and hovering) is done without additional configuration.

  The important thing to note here is that, even though its setup is done automatically, we are able to configure those sub-elements. When using this interface, the creation of the scene graph, which is used by **BokehJS**, is handled automatically, too.

- **bokeh.models**

This low-level interface is composed of two libraries: the JavaScript library called **BokehJS**, which gets used for displaying the charts in the browser and the Python library, which provides the developer interface. Internally, the definition created in Python creates JSON objects that hold the declaration for the JavaScript representation in the browser.

The models interface exposes complete control over how Bokeh plots and **widgets** (that are elements that enable users to interact with the data displayed) are assembled and configured. This means that it is up to the developer to ensure the correctness of the **scene graph** (which is a collection of objects describing the visualization).

Output

Outputting Bokeh charts is straightforward. There are three ways this can be done, depending on your needs:

- The `.show()` method: The basic option is to simply display the plot in an HTML page. This is done with the `.show()` method.
- Inline `.show()` method: When working with Jupyter Notebook, the `.show()` method will allow you to display the chart inside your notebook (when using inline plotting).
- The `.output_file()` method: You're also able to directly save the visualization to a file without any overhead using the `.output_file()` method. This will create a new file at the given path with a given name.

The most powerful way of providing your visualization is through the use of the Bokeh Server.

Bokeh Server

As we mentioned before, Bokeh creates **scene graph** JSON objects that will be interpreted by the BokehJS library to create the visualization output. This process allows you to have a unified format for other languages to create the same Bokeh plots and visualizations, independent of the language used.

If we think one step further, what if we could also keep the visualizations in sync with one another? This would enable us to create way more complex visualizations and leverage the tooling that Python provides. We could not only filter data but also do calculations and operations on the server side, which would update the visualizations in real time.

In addition to that, since we'd have an entry point for data, we can create visualizations that get fed by streams instead of static datasets. This design enables us to have way more complex systems with even greater capabilities.

Looking at the scheme of this architecture, we can see that the documents are provided on the server side, then moved over to the browser client, which then inserts it into the BokehJS library. This insertion will trigger the interpretation by BokehJS, which will then create the visualization itself. The following diagram describes the working of the Bokeh Server:

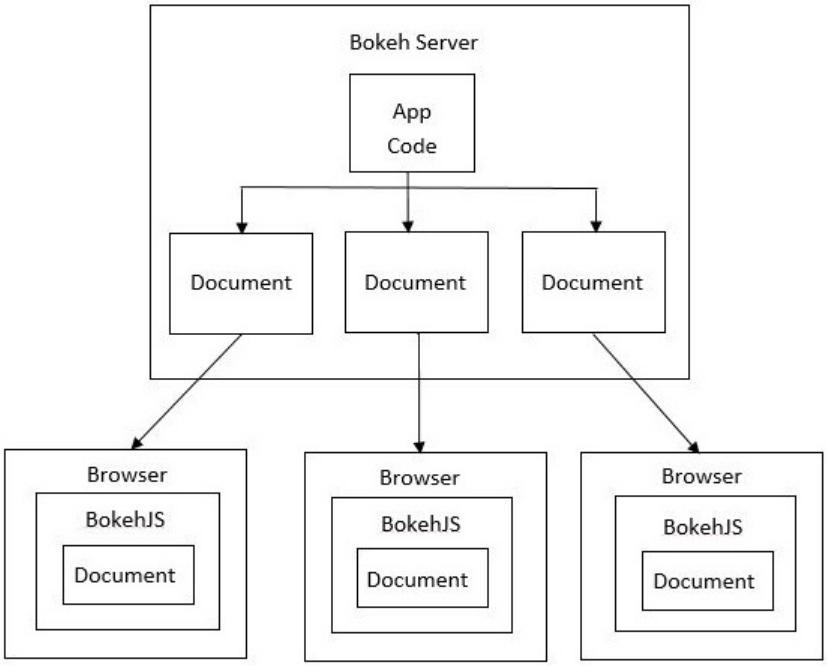

Figure 6.2: Workings of the Bokeh Server

Presentation

In Bokeh, presentations help make the visualization more interactive by using different features such as interactions, styling, tools, and layouts.

Interactions

Probably the most interesting feature of Bokeh is its interactions. There are basically two types of interactions: **passive** and **active**.

Passive interactions are actions that the users can take that neither change the data nor the displayed data. In Bokeh, this is called the **Inspector**. As we mentioned before, the inspector contains attributes such as zooming, panning, and hovering over data. This tooling allows the user to inspect its data further and might get better insights by only looking at a zoomed-in subset of the visualized data points.

Active interactions are actions that directly change the displayed data. This incorporates actions such as selecting subsets of data or filtering the dataset based on parameters. **Widgets** are the most prominent of active interactions, since they allow users to simply manipulate the displayed data with handlers. Widgets can be tools such as buttons, sliders, and checkboxes. Referencing back to the subsection about the output styles, those widgets can be used in both—the so-called standalone applications and the Bokeh Server. This will help us consolidate the recently learned theoretical concepts and make things clearer. Some of the interactions in Bokeh are tab panes, dropdowns, multi-selects, radio groups, text inputs, check button groups, data tables, and sliders.

Integrating

Embedding Bokeh visualizations can take two forms, as follows:

HTML document: These are the standalone HTML documents. These documents are very much self-contained.

Bokeh applications: They are backed by a Bokeh Server, which means that they provide the possibility to connect, for example: Python tooling for more advanced visualizations.

Bokeh is a little complex compared to Matplotlib with Seaborn and has its drawbacks like every other library, but once you have the basic workflow down, you're able to leverage the benefits that come with Bokeh, which are the ways you can extend the visual representation by simply adding interactivity features and giving power to the user.

Note

One interesting feature is the ***`to_bokeh`*** *method, which allows you to plot Matplotlib figures with Bokeh without configuration overhead. Further information about this method is available at the following link: https://bokeh.pydata.org/en/0.12.3/docs/user_guide/compat.html.*

In the following exercises and activities, we'll consolidate the theoretical knowledge and build several simple visualizations to understand Bokeh and its two interfaces. After we've covered the basic usage, we will compare the plotting and the **`models`** interface to see the difference in using them and work with widgets that add interactivity to the visualizations.

Note

All the exercises and activities in this chapter are developed using Jupyter Notebook and Jupyter Lab. The files can be downloaded from the following link: https://bit.ly/2T3Afn1.

Exercise 9: Plotting with Bokeh

In this exercise, we want to use the higher-level interface that is focused around providing a simple interface for quick visualization creation. Refer to the introduction to check back with the different interfaces of Bokeh. In this exercise, we will be using the **`world_population`** dataset. This dataset shows the population of different countries over the years. We will use the plotting interface to get some insights into the population densities of Germany and Switzerland:

1. Open the **`exercise09_solution.ipynb`** Jupyter Notebook from the **`Lesson06`** folder to implement this exercise. To do that, you need to navigate to the path of this file in the command-line terminal and type in **`jupyter-lab.`**

2. As we described previously, we will be using the plotting interface for this exercise. The only elements we have to import from plotting are the figure (which will initialize a plot) and the **`show`** method (which displays the plot):

```
# importing the necessary dependencies
import pandas as pd
from bokeh.plotting import figure, show
```

3. One thing to note is that if we want to display our plots inside a Jupyter Notebook, we also have to import and call the **`output_notebook`** method from the **`io`** interface of Bokeh:

```
# make bokeh display figures inside the notebook
from bokeh.io import output_notebook
output_notebook()
```

4. Use pandas to load our **`world_population`** dataset:

```
# loading the dataset with pandas
dataset = pd.read_csv('./data/world_population.csv', index_col=0)
```

5. A quick test by calling **head** on our DataFrame shows us that our data has been successfully loaded:

```
# looking at the dataset
dataset.head()
```

The following figure shows the output of the preceding code:

 Figure 6.3: Loading the top 5 rows of the world_population dataset using he head method

Figure 6.3: Loading the top five rows of the world_population dataset using he head method

6. To populate our x and y axis, we need to do some data extraction. The x-axis will hold all the years that are present in our columns. The y-axis will hold the population density values of the countries. We'll start with Germany:

```
# preparing our data for Germany
years = [year for year in dataset.columns if not year[0].isalpha()] de_vals = [dataset.loc[['Germany']][year] for year in years]
```

7. After extracting the wanted data, we can create a new plot by calling the Bokeh figure method. By providing parameters such as **title**, **x_axis_label**, and **y_axis_label**, we can define the descriptions displayed on our plot. Once our plot is created, we can add glyphs to it. In our example, we will use a simple line.By providing the **legend** parameter next to the x and y values, we get an informative legend in our visualization:

```
# plotting the population density change in Germany in the given years
plot = figure(title='Population Density of Germany', x_axis_label='Year', y_axis_label='Population Density')
plot.line(years, de_vals, line_width=2, legend='Germany')
show(plot)
```

The following figure shows the output of the preceding code:

 Figure 6.4: Creating a line plot from population density data of Germany

Figure 6.4: Creating a line plot from population density data of Germany

8. We now want to add another country. In this case, we will use **Switzerland**. We will use the same technique that we used with **Germany** to extract the data for **Switzerland**:

```
# preparing the data for the second country
ch_vals = [dataset.loc[['Switzerland']][year] for year in years]
```

9. We can simply add several layers of glyphs on to our figure plot. We can also stack different glyphs on top of one another, thus giving specific data-improved visuals. In this case, we want to add an orange line to our plot that displays the data from **Switzerland**. In addition to that, we also want to have circles for each entry in the data, giving us a better idea about where the actual data points reside. By using the same legend name, Bokeh creates a combined entry in the legend:

```
# plotting the data for Germany and Switzerland in one visualization,
# adding circles for each data point for Switzerland
plot = figure(title='Population Density of Germany and Switzerland', x_axis_label='Year', y_axis_label='Population Density')
```

```
plot.line(years, de_vals, line_width=2, legend='Germany')

plot.line(years, ch_vals, line_width=2, color='orange', legend='Switzerland')

plot.circle(years, ch_vals, size=4, line_color='orange', fill_color='white',
legend='Switzerland')

show(plot)
```

The following figure shows the output of the preceding code:

 Figure 6.5: Adding Switzerland to the plot

Figure 6.5: Adding Switzerland to the plot

10. When looking at a larger amount of data for different countries, it makes sense to have a plot for each of them separately. This can be achieved by using one of the layout interfaces. In this case, we are using the **`gridplot`**:

```
# plotting the Germany and Switzerland plot in two different visualizations

# that are interconnected in terms of view port

from bokeh.layouts import gridplot

plot_de = figure(title='Population Density of Germany', x_axis_label='Year',
y_axis_label='Population Density', plot_height=300)

plot_ch = figure(title='Population Density of Switzerland', x_axis_label='Year',
y_axis_label='Population Density', plot_height=300, x_range=plot_de.x_range,
y_range=plot_de.y_range)

plot_de.line(years, de_vals, line_width=2)

plot_ch.line(years, ch_vals, line_width=2)

plot = gridplot([[plot_de, plot_ch]])

show(plot)
```

The following figure shows the output of the preceding code:

 Figure 6.6: Using a gridplot to display the country plots next to each other

Figure 6.6: Using a gridplot to display the country plots next to each other

11. As the name suggests, we can arrange the plots in a grid. This also means that we can quickly get a vertical display when changing the two-dimensional list that was passed to the **`gridplot`** method:

```
# plotting the above declared figures in a vertical manner

plot_v = gridplot([[plot_de], [plot_ch]])

show(plot_v)
```

The following screenshot shows the output of the preceding code:

 Figure 6.7: Using the gridplot method to arrange the visualizations vertically

Figure 6.7: Using the gridplot method to arrange the visualizations vertically

Congratulations! We just covered the very basics of Bokeh. Using the **`plotting`** interface makes it really easy to get some quick visualizations in place. This really helps you understand the data you're working with.

This simplicity, however, is achieved by abstracting away complexity. We lose a lot of control by using the **`plotting`** interface. In the next exercise, we'll compare the **`plotting`** and **`models`** interfaces to show you how much abstraction is added to **`plotting`**.

Exercise 10: Comparing the Plotting and Models Interfaces

In this exercise, we want to compare the two interfaces: **plotting** and **models**. We will compare them by creating a basic plot with the high-level plotting interface and then recreate this plot by using the lower-level models interface. This will show us the differences between these two interfaces and give us a good direction for the later exercises to understand how to use the **`models`** interface:

1. Open the Jupyter Notebook **`exercise10_solution.ipynb`** from **`Lesson06`** to implement this exercise. To do that, once again, you need to navigate to the path of this file. In the command-line terminal, type **`jupyter-lab`**.

2. As we described previously, we will be using the **`plotting`** interface for this exercise. The only elements we have to import from plotting are the **`figure`** (which will initialize a plot) and the **`show`** method (which displays the plot):

   ```
   # importing the necessary dependencies
   import numpy as np
   import pandas as pd
   ```

3. One thing to note is that if we want to display our plots inside a Jupyter Notebook, we also have to import and call the **`output_notebook`** method from the **`io`** interface of Bokeh:

   ```
   # make bokeh display figures inside the notebook
   from bokeh.io import output_notebook
   output_notebook()
   ```

4. As we have done several times before, we will use pandas to load our **`world_population`** dataset:

   ```
   # loading the dataset with pandas
   dataset = pd.read_csv('./data/world_population.csv', index_col=0)
   ```

5. A quick test by calling **`head`** on our DataFrame shows us that our data has been successfully loaded:

   ```
   # looking at the dataset
   dataset.head()
   ```

   The following screenshot shows the output of the preceding code:

 Figure 6.8: Loading the top 5 rows of the world_population dataset using the head method

Figure 6.8: Loading the top five rows of the world_population dataset using the head method

6. In this part of the exercise, we will work with the previously seen **`plotting`** interface. As we saw before, we basically only need to import the **`figure`** to create, and **`show`** to display our plot:

   ```
   # importing the plotting dependencies
   from bokeh.plotting import figure, show
   ```

7. Our data stays the same for both plots, since we only want to change the way we create our visualization. We need the years present in the dataset as a list, the mean population density for the whole dataset for each year, and the mean population density per year for **`Japan`**:

```
# preparing our data of the mean values per year and Japan
years = [year for year in dataset.columns if not year[0].isalpha()]
mean_pop_vals = [np.mean(dataset[year]) for year in years]
jp_vals = [dataset.loc[['Japan']][year] for year in years]
```

8. When using the **`plotting`** interface, we can create a plot element that gets all the attributes about the plot itself, such as the title and axis labels. We can then use the plot element and "apply" our **`glyphs`** elements to it. In this case, we will plot the global mean with a line and the mean of **`Japan`** with crosses:

```
# plotting the global population density change and the one for Japan
plot = figure(title='Global Mean Population Density compared to Japan',
x_axis_label='Year', y_axis_label='Population Density')
plot.line(years, mean_pop_vals, line_width=2, legend='Global Mean')
plot.cross(years, jp_vals, legend='Japan', line_color='red')
show(plot)
```

The following screenshot shows the output of the preceding code:

Figure 6.9: Line plots comparing the global mean population density with that of Japan

As we can see in the preceding diagram, we have many elements already in place. This means that we already have the right x-axis labels, the matching range for the y-axis, and our legend is nicely placed in the upper-right corner without much configuration.

Using the models Interface

1. The **`models`** interface is of a much lower level compared to other interfaces. We can already see this when looking at the list of imports we need for a comparable plot. Looking through the list, we can see some familiar names such as **`Plot`**, **`Axis`**, **`Line`**, and **`Cross`**:

```
# importing the models dependencies
from bokeh.io import show
from bokeh.models.grids import Grid
from bokeh.models.plots import Plot
from bokeh.models.axes import LinearAxis
from bokeh.models.ranges import Range1d
from bokeh.models.glyphs import Line, Cross
from bokeh.models.sources import ColumnDataSource
from bokeh.models.tickers import SingleIntervalTicker, YearsTicker
from bokeh.models.renderers import GlyphRenderer
from bokeh.models.annotations import Title, Legend, LegendItem
```

2. Before we build our plot, we have to find out the **`min`** and **`max`** values for the y-axis since we don't want to have a too large or small range of values. We therefore get all the mean values for global and **`Japan`** without any invalid values and then get their smallest and largest values. Those values are then passed to the constructor of **`Range1d`**, which will give us a range that can later be used in the plot construction. For the x-axis, we have our list of years pre-defined:

```
# defining the range for the x and y axis

extracted_mean_pop_vals = [val for i, val in enumerate(mean_pop_vals) if i not in [0, len(mean_pop_vals) - 1]]

extracted_jp_vals = [jp_val['Japan'] for i, jp_val in enumerate(jp_vals) if i not in [0, len(jp_vals) - 1]]

min_pop_density = min(extracted_mean_pop_vals)

min_jp_densitiy = min(extracted_jp_vals)

min_y = int(min(min_pop_density, min_jp_densitiy))

max_pop_density = max(extracted_mean_pop_vals)

max_jp_densitiy = max(extracted_jp_vals)

max_y = int(max(max_jp_densitiy, max_pop_density))

xdr = Range1d(int(years[0]), int(years[-1]))

ydr = Range1d(min_y, max_y)
```

3. Once we have the **`min`** and **`max`** values for the y-axis, we can create two **`Axis`** objects that will be used to display the axis lines and the label for the axis. Since we also want ticks between the different values, we have to pass in a **`Ticker`** object that creates this setup for us:

```
# creating the axis

axis_def = dict(axis_line_color='#222222', axis_line_width=1,
major_tick_line_color='#222222',
major_label_text_color='#222222',major_tick_line_width=1)

x_axis = LinearAxis(ticker = SingleIntervalTicker(interval=10), axis_label = 'Year', **axis_def)

y_axis = LinearAxis(ticker = SingleIntervalTicker(interval=50), axis_label = 'Population Density', **axis_def)
```

4. Creating the title and plot itself is straightforward. We can pass a **`Title`** object to the title attribute of the **`Plot`** object:

```
# creating the plot object

title = Title(align = 'left', text = 'Global Mean Population Density compared to Japan')

plot = Plot(x_range=xdr, y_range=ydr, plot_width=650, plot_height=600,
title=title)
```

5. If we try to display our plot now using the show method, we will get an error, since we have no renderers defined at the moment. First, we need to add elements to our plot:

```
# error will be thrown because we are missing renderers that are created when adding elements
```

```
show(plot)
```

The following screenshot shows the output of the preceding code:

 Figure 6.10: Empty plot with title

Figure 6.10: Empty plot with title

6. When working with data, we always need to insert our data into a DataSource object. This can then be used to map the data source to the Glyph object that will be displayed in the plot:

```
# creating the data display
line_source = ColumnDataSource(dict(x=years, y=mean_pop_vals))
line_glyph = Line(x='x', y='y', line_color='#2678b2', line_width=2)
cross_source = ColumnDataSource(dict(x=years, y=jp_vals))
cross_glyph = Cross(x='x', y='y', line_color='#fc1d26')
```

7. When adding objects to the plot, you have to use the right **`add`** method. For layout elements such as the **`Axis`** objects, we have to use the **`add_layout`** method. **`Glyphs`**, which display our data, have to be added with the **`add_glyph`** method:

```
# assembling the plot
plot.add_layout(x_axis, 'below')
plot.add_layout(y_axis, 'left')
line_renderer = plot.add_glyph(line_source, line_glyph)
cross_renderer = plot.add_glyph(cross_source, cross_glyph)
```

8. If we now try to show our plot, we can finally see that our lines are in place:

```
show(plot)
```

The following screenshot shows the output of the preceding code:

 Figure 6.11: Models interface-based plot displaying the lines and axes

Figure 6.11: Models interface-based plot displaying the lines and axes

9. A few elements are still missing. One of them is the legend in the upper-right corner. To add a legend to our plot, we again have to use an object. Each **`LegendItem`** object will be displayed in one line in the legend:

```
# creating the legend
legend_items= [LegendItem(label='Global Mean', renderers=[line_renderer]),
LegendItem(label='Japan', renderers=[cross_renderer])]
legend = Legend(items=legend_items, location='top_right')
```

10. Creating the grid is straightforward: we simply have to instantiate two **`Gridobjects`** for the x and y axes. These grids will get the tickers of the previously created x and y axes:

```
# creating the grid
x_grid = Grid(dimension=0, ticker=x_axis.ticker)
y_grid = Grid(dimension=1, ticker=y_axis.ticker)
```

11. To add the last final touches, we, again, use the **`add_layout`** method to add the grid and the legend to our plot. After this, we can finally display our complete plot, which will look like the one we created in the first task, with only four lines of code:

```
# adding the legend and grids to the plot
plot.add_layout(legend)
plot.add_layout(x_grid)
plot.add_layout(y_grid)
show(plot)
```

The following screenshot shows the output of the preceding code:

 Figure 6.12: Full recreation of the visualization done with the plotting interface

Figure 6.12: Full recreation of the visualization done with the plotting interface

Congratulations! As you can see, the **`models`** interface should not be used for simple plots. It's meant to provide the full power of Bokeh to experienced users that have specific requirements that need more than the **`plotting`** interface. Having seen the **`models`** interface before will come in handy in our next topic, which is about widgets.

Adding Widgets

One of the most powerful features of Bokeh is its ability to use **widgets** to interactively change the data that's displayed in the visualization. To understand the importance of interactivity in your visualizations, imagine seeing a static visualization about stock prices that only show data for the last year. If this is what you specifically searched for, it's suitable enough, but if you're interested to see the current year or even visually compare it to recent years, those plots won't work and will add additional work, since you have to create them for every year. Comparing this to a simple plot that lets the user select the wanted date range, we can already see the advantages. There are endless options to combine widgets and tell your story. You can guide the user by restricting values and only displaying what you want them to see. Developing a story behind your visualization is very important, and doing this is much easier if the user has ways of interacting with the data.

Bokeh widgets work best when used in combination with the Bokeh server. However, using the Bokeh server approach is beyond the content of this book, since we would need to work with simple Python files and can't leverage the power of Python notebook. Instead, we will use a hybrid approach that only works with the older Jupyter Notebook.

Exercise 11: Basic Interactivity Widgets

This first exercise of the *Adding Widgets* topic will give you a gentle introduction into the different widgets and the general concept of how to use them in combination with your visualizations. We will look at basic widgets and build a simple plot that will display the first 25 data points for the selected stock. The displayed stock can be changed with a drop-down menu.

The dataset of this exercise is a **`stock_prices`** dataset. This means that we will be looking at data over a range of time. As this is a large and variable dataset, it will be easier to show and explain different widgets such as slider and dropdown on it. The dataset is available in the data folder of the GitHub repository; here is the link to it: https://bit.ly/2UaLtSV.

We will look at the different available widgets and how to use them before going in and building a basic plot with one of them. There are a few different options regarding how to trigger updates, which are also explained in the following steps. The widgets that will be covered in this exercise are explained in the following table:

 Figure 6.13: Some of the basic widgets with examples

Figure 6.13: Some of the basic widgets with examples

1. Open the **`exercise11_solution.ipynb`** Jupyter Notebook from the **`Lesson06`** folder to implement this exercise. Since we need to user Jupyter Notebook in this example, we will type in the following at the command line: **`jupyter notebook.`**

2. A new browser window will open that lists all the files in the current directory. Click on **`exercise11_solution.ipynb`**, which will open it in a new tab.

3. This exercise will first introduce you to the basic widgets before showing you how to create a basic visualization with them. Therefore, we will add more imports in suitable places throughout the code. For importing our dataset, we need pandas:

```
# importing the necessary dependencies
import pandas as pd
```

4. Again, we want to display our plots inside a Jupyter Notebook, so we have to import and call the **`output_notebook`** method from the **`io`** interface of Bokeh:

```
# make bokeh display figures inside the notebook
from bokeh.io import output_notebook
output_notebook()
```

5. After downloading the dataset and moving it into the data folder of this chapter, we can import our **`stock_prices.csv`** data:

```
# loading the Dataset with geoplotlib
dataset = pd.read_csv('./data/stock_prices.csv')
```

6. A quick test by calling **`head`** on our DataFrame shows us that our data has been successfully loaded:

```
# looking at the dataset
dataset.head()
```

The following screenshot shows the output of the preceding code:

 Figure 6.14: Loading the top 5 rows of the stock_prices dataset using the head method

Figure 6.14: Loading the top five rows of the stock_prices dataset using the head method

7. Since the date column has no information about the hour, minute, and second, we want to avoid displaying them in the visualization later on and simply display the year, month, and day. Therefore, we'll create a new column that holds the formatted short version of the date value. Note that the execution of the cell will take a moment, since it's a fairly large dataset. Please be patient:

```
# mapping the date of each row to only the year-month-day format
from datetime import datetime
def shorten_time_stamp(timestamp):
shortened = timestamp[0]
if len(shortened) > 10:
```

```
parsed_date=datetime.strptime(shortened, '%Y-%m-%d %H:%M:%S')
shortened=datetime.strftime(parsed_date, '%Y-%m-%d')
return shortened
dataset['short_date'] = dataset.apply(lambda x: shorten_time_stamp(x), axis=1)
```

8. Taking another look at our updated dataset, we can see a new column called **short_date** that holds the date without the hour, minute, and second information:

```
# looking at the dataset with shortened date
dataset.head()
```

The following screenshot shows the output of the preceding code:

 Figure 6.15: Dataset with the added short_date column

Figure 6.15: Dataset with the added `short_date` column

Looking at Basic Widgets

1. In this first task, the interactive widgets are added with the interact element of IPython. We have to specifically import them:

```
# importing the widgets
from ipywidgets import interact, interact_manual
```

2. We'll be using the "syntactic sugar" way of adding a decorator to a method, that is, by using annotations. This will give us an interactive element that will be displayed below the executable cell. In this example, we'll simply print out the result of the interactive element:

```
# creating a checkbox
@interact(Value=False)
def checkbox(Value=False):
print(Value)
```

The following screenshot shows the output of the preceding code:

Figure 6.16: Interactive checkbox that will switch from False to True if checked

Note

`@interact()` *is called a* ***decorator****, which wraps the annotated method into the interact component. This allows us to display and react to the change of the drop-down menu. The method will be executed every time the value of the dropdown changes.*

3. Once we have the first element in place, all the other elements are created the exact same way, just altering the data type of the arguments in the decorator.

4. Use the following code for the dropdown:

```
# creating a dropdown
```

```
options=['Option1', 'Option2', 'Option3', 'Option4']
@interact(Value=options)
def slider(Value=options[0]):
print(Value)
```

The following screenshot shows the output of the preceding code:

 Figure 6.17: Interactive dropdown

Figure 6.17: Interactive dropdown

Use the following code for the input text:

```
# creating an input text
@interact(Value='Input Text')
def slider(Value):
print(Value)
```

The following screenshot shows the output of the preceding code:

 Figure 6.18: Interactive text input

Figure 6.18: Interactive text input

Use the following code to apply multiple widgets:

```
# multiple widgets with default layout
options=['Option1', 'Option2', 'Option3', 'Option4']
@interact(Select=options, Display=False)
def uif(Select, Display):
print(Select, Display)
```

The following screenshot shows the output of the preceding code:

 Figure 6.19: Two widgets are displayed vertically by default

Figure 6.19: Two widgets are displayed vertically by default

Use the following code to apply an int slider:

```
# creating an int slider with dynamic updates
@interact(Value=(0, 100))
def slider(Value=0):
print(Value)
```

The following screenshot shows the output of the preceding code:

 Figure 6.20: Interactive int slider

Figure 6.20: Interactive int slider

Use the following code to apply an int slider that triggers upon releasing the mouse:

```
# creating an int slider that only triggers on mouse release
from ipywidgets import IntSlider
slider=IntSlider(min=0, max=100, continuous_update=False)
@interact(Value=slider)
def slider(Value=0.0):
print(Value)
```

The following screenshot shows the output of the preceding code:

 Figure 6.21: Interactive int slider that only triggers on mouse release

Figure 6.21: Interactive int slider that only triggers upon mouse release

Note:

Although the output of Figures 6.20 and 6.21 looks the same, in Figure 6.21, the slider triggers only upon mouse release.

5. If we don't want to update our plot every time we change our widget, we can also use the **`interact_manual`** decorator, which adds an execution button to the output:

```
# creating a float slider 0.5 steps with manual update trigger
@interact_manual(Value=(0.0, 100.0, 0.5))
def slider(Value=0.0):
print(Value)
```

The following screenshot shows the output of the preceding code:

 Figure 6.22: Interactive int slider with a manual update trigger

Figure 6.22: Interactive int slider with a manual update trigger

Note

*Compared to the previous cells, this one contains the **`interact_manual`** decorator instead of interact. This will add an execution button, which will trigger the update of the value instead of triggering with every change. This can be really useful when working with larger datasets, where the recalculation time would be large. Because of this, you don't want to trigger the execution for every small step but only once you have selected the right value.*

Creating a Basic Plot and Adding a Widget

In this task, we will create a basic visualization with the stock price dataset. This will be your first interactive visualization in which you can dynamically change the stock that is displayed in the graph. We will get used to one of the aforementioned interactive widgets: the drop-down menu. It will be the main point of interaction for our visualization:

1. To be able to create a plot, we first need to import the already-familiar figure and show methods from the plotting interface. Since we also want to have a panel with two tabs displaying different plot styles, we also need the **`Panel`** and **`Tabs`** classes from the **`models`** interface:

```
# importing the necessary dependencies

from bokeh.models.widgets import Panel, Tabs

from bokeh.plotting import figure, show
```

2. To structure our notebook better, we want to write an adaptable method that gets a subsection of stock data as an argument and builds a two-tab **`Pane`** object that lets us switch between the two views in our visualization. The first tab will contain a line plot of the given data, while the second one will contain a circle-based representation of the same data. A legend will display the name of the currently viewed stock:

```
# method to build the tab-based plot

def get_plot(stock):

stock_name=stock['symbol'].unique()[0]

line_plot=figure(title='Stock prices',

x_axis_label='Date', x_range=stock['short_date'],

y_axis_label='Price in $USD')

line_plot.line(stock['short_date'], stock['high'], legend=stock_name)

line_plot.xaxis.major_label_orientation = 1

circle_plot=figure(title='Stock prices', x_axis_label='Date',
x_range=stock['short_date'], y_axis_label='Price in $USD')

circle_plot.circle(stock['short_date'], stock['high'], legend=stock_name)

circle_plot.xaxis.major_label_orientation = 1

line_tab=Panel(child=line_plot, title='Line')

circle_tab=Panel(child=circle_plot, title='Circles')

tabs = Tabs(tabs=[ line_tab, circle_tab ])

return tabs
```

3. Before we can build our interaction, we have to get a list of all the stock names that are present in the dataset. Once we have done that, we can then use this list as an input for the interact element. With each interaction of the dropdown, our displayed data will then be updated. To keep it simple, we only want to display the first 25 entries of each stock in this task. By default, the stock of Apple should be displayed. Its symbol in the dataset is **AAPL**:

```
# extracing all the stock names

stock_names=dataset['symbol'].unique()
```

4. We can now add the drop-down widget in the decorator and call the method that returns our visualization in the show method with the selected stock. This will give us a visualization that is displayed in a pane with two tabs. The first tab will display an interpolated line and the second tab will display the values as circles:

```
# creating the dropdown interaction and building the plot

# based on selection
```

```
@interact(Stock=stock_names)
def get_stock_for(Stock='AAPL'):
stock = dataset[dataset['symbol'] == Stock][:25]
show(get_plot(stock))
```

The following screenshot shows the output of the preceding code:

 Figure 6.23: Line tab with the data of AAPL displayed

Figure 6.23: Line tab with the data of AAPL displayed

The following screenshot shows the output of the code in step 16:

 Figure 6.24: Circle tab with the data of AAPL displayed

Figure 6.24: Circle tab with the data of AAPL displayed

Note

We can already see that each date is displayed on the x-axis. If we want to display a bigger time range, we have to customize the ticks on our x-axis. This can be done using ticker objects.

Congratulations! We just covered the very basics of widgets and how to use them in Jupyter Notebook.

Note

If you want to learn more about using widgets and which widgets can be used in Jupyter, you can refer to these links: https://bit.ly/2Sx9txZ and https://bit.ly/2T4FcM1.

Activity 29: Extending Plots with Widgets

In this activity, you will combine what you have already learned about Bokeh. You will also need the skills you have acquired while working with pandas for additional DataFrame handling. We will create an interactive visualization that lets us explore the end results of the Olympics 2016 in Rio.

Our visualization will display each country that participated in a coordinate system where the x-axis represents the number of won medals and the y-axis represents the number of athletes. Using interactive widgets, we will be able to filter down the displayed countries in both the maximum amount of won medals and the maximum amount of athletes axes.

There are many options when it comes to choosing which interactivity to use. We will focus on having only two widgets to make it easier for you to understand the concepts. In the end, we will have a visualization that allows us to filter countries for the amount of medals and athletes they placed in the Olympics and upon hovering over the single data points, receive more information about each country:

1. Open the **`activity29.ipynb`** Jupyter Notebook from the **`Lesson06`** folder to implement this activity.
2. Don't forget to enable to notebook output using the **`bokeh.io`** interface. Import pandas and load the dataset. Make sure that the dataset is loaded by displaying the first five elements of the dataset.
3. Import **`figure`** and **`show`** from Bokeh and **`interact`** and **`widgets`** from **`ipywidgets`** to get started.

4. When starting to create our visualization, we have to import the tools we will be using. Import **`figure`** and **`show`** from Bokeh and the **`interact`** and **`widgets`** interfaces from **`ipywidgets`**.

5. After extracting the necessary data, we will set up the interaction elements. Scroll down until you reach the cell that says **`getting the max amount of medals and athletes of all countries`**. Extract those two numbers from the dataset.

6. After extracting the maximum numbers of medals and athletes, create widgets for **`IntSlider`** for the maximum number of athletes (orientation vertical) and **`IntSlider`** for the maximum number of medals (orientation horizontal).

7. The last step of preparation before implementing our plotting is to set up the **`@interact`** method, which will display the whole visualization. The only code we will write here is to **`show`** the return value of the **`get_plot`** method that gets all the interaction element values as parameters.

8. After implementing the decorated method, we can now move up in our notebook and work on the **`get_plot`** method.

9. First, we want to filter down our countries dataset that contains all the countries that placed athletes in the **`olympic games`**. We need to check whether they have less or equal number of medals and athletes than our max values passed as arguments.

10. Once we have a filtered down our dataset, we can create our DataSource. This DataSource will be used both for the tooltips and the printing of the circle glyphs.

11. After that, we will create a new plot using the figure method that has the following attributes: title as **`Rio Olympics 2016 - Medal comparison`**, x_axis_label as **`Number of Medals`**, y_axis_label as **`Num of Athletes.`**

12. The last step is to execute every cell starting from the **`get_plot`** cell to the bottom, again, making sure that all implementations are captured.

13. When executing the cell that contains the **`@interact`** decorator, you will see the scatter plot that displays a circle for every country displaying additional information such as the short code of the country, the amount of athletes, and the number of gold, silver, and bronze medals.

Note:

The solution for this activity can be found on page 315.

Summary

In this chapter, we have looked at another option for creating visualizations with a whole new focus: web-based Bokeh plots. We also discovered ways in which we can make our visualizations more interactive and really give the user the chance to explore data in a whole different way. As we mentioned in the first part of this chapter, Bokeh is a comparably new tool that empowers developers to use their favorite language to create easily portable visualizations for the web. After working with Matplotlib, Seaborn, geoplotlib, and Bokeh, we can see some common interfaces and similar ways to work with those libraries. After understanding the tools that are covered in this book, it will be simple to understand new plotting tools.

In the next and final chapter, we will introduce a new, not-yet-covered dataset, with which you will create a visualization. This last chapter will allow you to consolidate the concepts and tools that you have learned about in this book and further enhance your skills.

Chapter 7

Combining What We Have Learned

Learning Objectives

By the end of this chapter, you will be able to:

- Apply your skills in Matplotlib and Seaborn
- Create a time series with Bokeh
- Analyze geospatial data with geoplotlib

In this chapter, we will apply all the concepts that we have learned in all the previous chapters. We will use three new datasets in combination with practical activities for Matplotlib, Seaborn, geoplotlib, and Bokeh. We will conclude this chapter with a summary that recaps what we've learned throughout the book.

Introduction

To consolidate what we have learned, we will provide you with three sophisticated activities. Each activity uses one of the libraries that we have covered in this book. Every activity has a bigger dataset than we have used before in this book, which will prepare you for larger datasets.

Note

All activities will be developed in the Jupyter Notebook or Jupyter Lab. Please download the GitHub repository with all the prepared templates from https://bit.ly/2SswjqE.

Activity 30: Implementing Matplotlib and Seaborn on New York City Database

In this activity, we will visualize data about New York City (NYC) and compare it to the state of New York and the United States (US). The American Community Survey (ACS) Public-Use Microdata Samples (PUMS) dataset (one-year estimate from 2017) from https://www.census.gov/programs-surveys/acs/technical-documentation/pums/documentation.2017.html is used. For this activity, you can either use Matplotlib, Seaborn, or a combination of both.

In this activity, the datasets "New York Population Records" (**`./data/pny.csv`**) and "New York Housing Unit Records" (**`./data/hny.csv`**) are used. The first dataset contains information about the New York population, and the second dataset contains information about housing units. The dataset contains data for about 1% of the population and housing units. Due to the extensive amount of data, we do not provide the datasets for the whole of the US; instead, we will provide the required information related to the US if necessary. The **`PUMS_Data_Dictionary_2017.pdf`** PDF gives an overview and description of all variables. A further description of the codes can be found in **`ACSPUMS2017CodeLists.xls`**:

1. Open the **`activity30.ipynb`** Jupyter Notebook from the **`Lesson07`** folder to implement this activity.
2. Use pandas to read both .csv files located in the subdirectory **`data`**.

3. Use the given PUMA (public use microdata area code based on the 2010 Census definition, which are areas with populations of 100k or more) ranges to further divide the dataset into NYC districts (Bronx, Manhatten, Staten Island, Brooklyn, and Queens):

```
# PUMA ranges

bronx = [3701, 3710]

manhatten = [3801, 3810]

staten_island = [3901, 3903]

brooklyn = [4001, 4018]

queens = [4101, 4114]

nyc = [bronx[0], queens[1]]
```

4. In the dataset, each sample has a certain **`weight`** that reflects the **`weight`** for the total dataset. Therefore, we cannot simply calculate the median. Use the given **`weighted_median`** function in the following code to compute the median:

```
# Function for a 'weighted' median

def weighted_frequency(values, weights):

  weighted_values = []

  for value, weight in zip(values, weights):

    weighted_values.extend(np.repeat(value, weight))

  return weighted_values

def weighted_median(values, weights):

  return np.median(weighted_frequency(values, weights))
```

5. In this subtask, we will create a plot containing multiple subplots that visualize information with regard to NYC wages. Visualize the median household income for the US, New York, New York City, and its districts. Visualize the average wage by gender for the given occupation categories for the population of NYC:

```
occ_categories = ['Management,\nBusiness,\nScience,\nand Arts\nOccupations',
'Service\nOccupations',

'Sales and\nOffice\nOccupations', 'Natural Resources,\nConstruction,\nand
Maintenance\nOccupations',

'Production,\nTransportation,\nand Material Moving\nOccupations']

occ_ranges = {'Management, Business, Science, and Arts Occupations': [10, 3540],
'Service Occupations': [3600, 4650],

'Sales and Office Occupations': [4700, 5940], 'Natural Resources, Construction,
and Maintenance Occupations': [6000, 7630],

'Production, Transportation, and Material Moving Occupations': [7700, 9750]}
```

   Visualize the wage distribution for New York and NYC. Use the following yearly wage intervals: 10k steps between 0 and 100k, 50k steps between 100k and 200k, and >200k.

6. Use a tree map to visualize the percentage for the given occupation subcategories for the population of NYC:

```
occ_subcategories = {'Management,\nBusiness,\nand Financial': [10, 950],
'Computer, Engineering,\nand Science': [1000, 1965],
'Education,\nLegal,\nCommunity Service,\nArts,\nand Media': [2000, 2960],
'Healthcare\nPractitioners\nand\nTechnical': [3000, 3540],
'Service': [3600, 4650],
'Sales\nand Related': [4700, 4965],
'Office\nand Administrative\nSupport': [5000, 5940],
'': [6000, 6130],
'Construction\nand Extraction': [6200, 6940],
'Installation,\nMaintenance,\nand Repair': [7000, 7630],
'Production': [7700, 8965],
'Transportation\nand Material\nMoving': [9000, 9750]}
```

7. Use a heatmap to show the correlation between difficulties (self-care difficulty, hearing difficulty, vision, difficulty, independent living difficulty, ambulatory difficulty, veteran service-connected disability, and cognitive difficulty) and age groups (<5, 5-11, 12-14, 15-17, 18-24, 25-34, 35-44, 45-54, 55-64, 65-74, and 75+) in New York City.

Note:

The solution to this activity can be found on page 321.

Bokeh

Stock price data is one of the most interesting types of data for many people. When thinking about its nature, we can see that it is highly dynamic and constantly changing. To understand it, we need high levels of interactivity to not only look at the stocks of interest, but also to compare different stocks, see their traded volume, and the high/lows of the given dates and whether it rose or sunk the day before that.

Taking into account all of the aforementioned features, we need to use a highly customizable visualization tool. We also need the possibility to add different widgets to enable interactivity. In this activity, we will therefore use Bokeh to create a candle-stick visualization with several interactivity widgets to enable better exploration of our data.

Activity 31: Visualizing Bokeh Stock Prices

This activity will combine most of what you have already learned about Bokeh. You will also need the skills you have acquired while working with pandas. We will create an interactive visualization that displays a candlestick plot, which is often used when handling stock price data. We will be able to compare two stocks to each other by selecting them from dropdowns. A RangeSlider will allow us to restrict the displayed date range in the requested year 2016. Depending on what graph we choose, we will either see the candlestick visualization or a simple line plot displaying the volume of the selected stock:

1. Open the **`activity31.ipynb`** Jupyter Notebook from the **`Lesson07`** folder to implement this activity.

2. Don't forget to enable notebook output by using the **`bokeh.io`** interface. Import pandas and load the downloaded dataset. Make sure that the dataset is loaded by displaying the first five elements of the dataset.

3. We need to create a column in our DataFrame that holds the information from the date column without the hour, minute, and second information. Make sure your implementation works correctly by displaying the first five elements of the updated DataFrame.

4. There are many options when it comes to choosing which interactivity to use. Since the goal of this activity is to be able to compare two stocks with each other in terms of traded volume and the high/low and open/close prices over a time range, we will need widgets to select elements and a slider to select a given range. Considering that we have two options of display, we also need a way to select either one or the other.

5. Import **`figure`** and **`show`** from Bokeh and **`interact`** and **`widgets`** from **`ipywidgets`** to get started.

6. When starting to create our visualization, we have to import the tools we will be using. Import **`figure`** and **`show`** from Bokeh and the **`interact`** and **`widgets`** interface from **`ipywidgets`**.

7. Execute the cells from top to bottom until you reach the cell that has the comment **`#extracting the necessary data.`** Start your implementation there. Get the unique stock names from the dataset. Filter out the dates from 2016. Only get the unique dates from 2016. Create a list that contains the strings **`open-close`** and **`volume`**, which will be used for the radio buttons to switch between the two plots.

8. After extracting the necessary data, we will set up the interaction elements. Create widgets for the following: Dropdown for the first stock name (default value will be **`AAPL`**); Dropdown for the second stock name that will be compared to the first (default value will be **`AON`**); **`SelectionRangeSlider`** to select the range of dates we want to display in our plot (default values displayed will be 0 to 25); RadioButtons to choose between the candlestick plot and the plot that displays the traded volume (the default value will be **`open-close`**, which will display the candlestick plot.)

9. The last step of preparation before implementing our plot is to set up the **`@interact`** method that finally displays the whole visualization.

10. The only code we will write here is to **`show`** the return value of the **`get_plot`** method that gets all the interaction element values as parameters.

11. After implementing the decorated method, we can now move up in our notebook and work on the **`add_candle_plot`** method. Make sure to use the example in the Bokeh documentation as an outline. You can find the same in this link: https://bokeh.pydata.org/en/latest/docs/gallery/candlestick.html.

12. The next step will be to move on and implement the line plot in the cell that contains the **`get_plot`** method. Plot a line for the data from **`stock_1`** with a blue color. Plot a line for the data from **`stock_2`** with an orange color.

13. Before finalizing this activity, we want to add one more interactivity feature: muting different elements of our plot. This can be done by clicking on one of the displayed elements in the legend of the visualization. Of course, we first have to tell Bokeh what it should do. Read up on this at https://bokeh.pydata.org/en/latest/docs/user_guide/interaction/legends.html.

14. The last step is to execute every cell starting from the **`add_candle_plot`** cell to the bottom, again making sure that all implementations are captured.

15. When executing the cell that contains the **`@interact`** decorator, you will see the candlestick plot for the two, by default, selected, with the **`AAPL`** and **`AON`** stocks displayed.

Note:

The solution for this activity can be found on page 327.

Geoplotlib

The dataset that's used in this activity is of Airbnb, which is publicly available online. Accommodation listings have two predominant feature: latitude and longitude. Those two features allow us to create geo-spatial visualizations that gives us a better understanding of attributes such as the distribution of accommodations across each city.

In this activity, we will therefore use geoplotlib to create a visualization that maps each accommodation to a dot on a map, colored based on either the price or rating of that listing. The two attributes can be switched by pressing the left and right keys on the keyboard.

Activity 32: Analyzing Airbnb Data with geoplotlib

In this last activity for geoplotlib, we will use airbnb listing data to determine the most expensive and best-rated regions of accommodation in the New York area. We will write a custom layer with which we can switch between the price and the review score of each accommodation. In the end, we will be able to see the hotspots for the most expensive and best-rated accommodation across New York.

In theory, we should see a price increase the closer we get to the center of Manhatten. It will be very interesting to see whether the ratings for the given accommodations also increase the closer we get to the center of Manhatten:

1. Open the **`activity32.ipynb`** Jupyter Notebook from the **`Lesson07`** folder to implement this activity.
2. First, make sure you import the necessary dependencies.
3. Load the **`airbnb_new_york.csv`** dataset using pandas. If your system is a little bit slower, just use the **`airbnb_new_york_smaller.csv`** dataset with less data points.
4. Get a feeling for the dataset and the features it has by taking a look at it.
5. Since our dataset once again has columns that are named **`Latitude`** and **`Longitude`** instead of **`lat`** and **`lon`**, rename those columns to their short versions, which we need for geoplotlib.
6. In addition to that, we also want to clean and map our two major coumns: **`price`** and **`review_scores_rating`**. Fill the **`n/a`** values and create a new column called **`dollar_price`** that holds the price as a float.
7. Before we finally create the layer, we also want to trim down the number of columns our working dataset has. Create a subsection of the columns with **`id`**, **`latitude`** (as **`lat`**), **`longitude`** (as **`lon`**), **`price`** (in **`$`**), and **`review_scores_rating`**.
8. Create a new **`DataAccessObject`** with the newly created subsection of the dataset. Use it to plot out a dot map.
9. Create a new **`ValueLayer`** that extends the geoplotlib **`BaseLayer`**.
10. Given the data, we want to plot each point on the map with a color that is defined by the currently selected attribute, either **`price`** or **`rating`**.
11. To assign each point a different color, we simply paint each point separately. This is definitely not the most efficient solution, but it will do for now.
12. We will need the following instance variables: **`self.data`**, which holds the dataset; **`self.display`**, which holds the currently selected attribute name; **`self.painter`**, which holds an instance of the **`BatchPainter`** class; **`self.view`**,

which holds the **`BoundingBox`**; **`self.cmap`**, which holds a color map with the **`jet`** color schema; and an alpha of **`255`** and **`100`** levels.

13. Implement the **`__init__`**, **`invalidate`**, **`draw`**, and **`bbox`** method for the **`ValueLayer`**.
14. Use the provided **`BoundingBox`** that's focused on New York when calling the **`ValueLayer`**.

Note:

The solution for this activity can be found on page 336.

Summary

This chapter gave us a short overview and recap of everything that was covered in this bookware on the basis of three extensive practical activities. In *Chapter 1, The Importance of Data Visualization and Data Exploration*, we started with a Python library journey that we used as a guide throughout the whole bookware. We first talked about the importance of data and visualizing this data to get meaningful insights into it and gave a quick recap on different statistics concepts. In several activities, we learned how to import and handle datasets with Numpy and pandas. In *Chapter 2*, *All You Need to Know about Plots*, we discussed various visualizations plots/charts and which visualizations are best to display certain information. We mentioned the use case, design practices, and practical examples for each plot type.

In *Chapter 3, A Deep Dive into Matplotlib*, we thoroughly covered Matplotlib and started with the basic concepts. Next, a deep insight into the numerous possibilities to enrich visualizations with text was given. Emphasis was put on explaining almost all plotting functions Matplotlib offers using practical examples. Furthermore, we talked about different ways to create layouts. The chapter was rounded off with how to visualize images and write mathematical expressions. In *Chapter 4*, *Simplifying Visualizations Using Seaborn*, Seaborn was covered, which is built on top of Matplotlib and provides a higher-level abstraction to make insightful visualizations. With several examples, we showed you how Seaborn can simplify the creation of visualizations. We also introduced further plots, such as heatmaps, violin plots, and correlograms. Finally, we used Squarify to create tree maps.

Visualizing geospatial data was covered in *Chapter 5*, *Plotting Geospatial Data*, using geoplotlib. Understanding how geoplotlib is structured internally explained why we had to work with the pyglet library when adding interactivity to our visualizations. In the book of this chapter, we worked with different datasets and built both static and interactive visualizations for geospatial data. In *Chapter 6*, *Making Things Interactive with Bokeh*, we focused on working with Bokeh, which targets modern web browsers to present interactive visualizations. Starting with simple examples, we emphasized the biggest advantage of Bokeh, namely interactive widgets. We ended the book with this chapter, applying all the skills that we've learned about by using three real-life datasets.

Appendix

About

This section is included to assist the students to perform the activities in the course. It includes detailed steps that are to be performed by the students to achieve the objectives of the activities.

Chapter 1: The Importance of Data Visualization and Data Exploration

Activity 1: Using NumPy to Compute the Mean, Median, Variance, and Standard Deviation for the Given Numbers

Solution:

Let's use NumPy to calculate the mean, median, variance, and standard deviation:

1. Import the necessary libraries:

   ```
   # importing the necessary dependencies

   import numpy as np
   ```

2. Load the **normal_distribution.csv** dataset by using the **genfromtxt** method of NumPy:

   ```
   # loading the dataset

   dataset = np.genfromtxt('./data/normal_distribution.csv', delimiter=',')
   ```

3. First, we want to print a subset of the first two rows of the dataset:

   ```
   # looking at the first two rows of the dataset

   dataset[0:2]
   ```

The output of the preceding code is as follows:

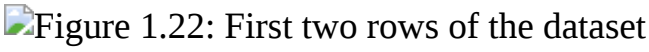
Figure 1.22: First two rows of the dataset

Figure 1.22: First two rows of the dataset

4. Once we know that our dataset has been successfully loaded, we can start solving our first task, which is calculating the mean of the third row.

   The third row can be accessed by indexing **dataset[2]**:

   # calculate the mean of the third row

   np.mean(dataset[2])

   The output of the preceding code is as follows:

   
   Figure 1.23: Mean of the third row

   Figure 1.23: Mean of the third row

5. The last element of an ndarray can be indexed the same way a regular Python List can be accessed. **dataset[:, -1]** will give us the last column of every row:

   # calculate the mean of the last column

   np.mean(dataset[:,-1])

   The output of the preceding code is as follows:

   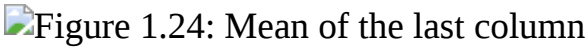
   Figure 1.24: Mean of the last column

   Figure 1.24: Mean of the last column

6. The double-indexing mechanism of NumPy gives us a nice interface to extract sub-selection. In this task, we are asked to get a sub-matrix of the first three elements of every row of the first three columns:

calculate the mean of the intersection of the first 3 rows and first 3 columns

np.mean(dataset[0:3, 0:3])

The output of the preceding code is as follows:

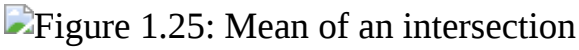
Figure 1.25: Mean of an intersection

Figure 1.25: Mean of an intersection

7. Moving to the next set of tasks that cover the usage of the median, we will see that the API is consistent between the different methods.:

   # calculate the median of the last row

   np.median(dataset[-1])

   The output of the preceding code is as follows:

   
   Figure 1.26: Median of the last row

   Figure 1.26: Median of the last row

8. Reverse indexing also works when defining a range. So, if we want to get the last three columns, we can use **dataset[:, -3:]**:

   # calculate the median of the last 3 columns

   np.median(dataset[:, -3:])

   The output of the preceding code is as follows:

   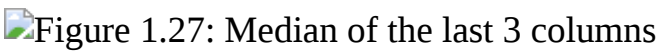
   Figure 1.27: Median of the last 3 columns

   Figure 1.27: Median of the last 3 columns

9. As we saw in the previous exercise, we can aggregate the values along an **axis**. If we want to calculate the rows, we use **axis=1**:

calculate the median of each row

np.median(dataset, axis=1)

The output of the preceding code is as follows:

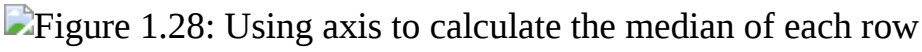
Figure 1.28: Using axis to calculate the median of each row

Figure 1.28: Using axis to calculate the median of each row

10. The last method we'll cover here is variance. Again, NumPy provides us with a consistent API, which makes doing this easy. To calculate the variance for each column, we have to use **axis 0**:

    # calculate the variance of each column

    np.var(dataset, axis=0)

    The output of the preceding code is as follows:

Figure 1.29: Variance across each column

11. When only looking at a very small subset of the matrix (2x2) elements, we can apply what we learned in the statistical overview to observe that the value is way smaller than the whole dataset:

    Note

    A small subset of a dataset does not display the attributes of the whole.

    # calculate the variance of the intersection of the last 2 rows and first 2 columns

    np.var(dataset[-2:, :2])

    The output of the preceding code is as follows:


Figure 1.30: Variance of a small subset of the dataset

Figure 1.30: Variance of a small subset of the dataset

12. The values of the variance might seem a little bit strange at first.

    You can always go back to the *Measures of Dispersion* topic to recap what you've learned so far.

    Just remember, variance is not the standard deviation:

    # calculate the standard deviation for the dataset

    np.std(dataset)

    The output of the preceding code is as follows:

     Figure 1.31: Standard deviation of the complete dataset

Figure 1.31: Standard deviation of the complete dataset

Congratulations! You've completed your first activity using NumPy. In the following activities, this knowledge will be further consolidated.

Activity 2: Indexing, Slicing, Splitting, and Iterating

Solution:

Let's use the features of NumPy to index, slice, split, and iterate ndarrays.

Indexing

1. Import the necessary libraries:

   # importing the necessary dependencies

   import numpy as np

2. Load the **normal_distribution.csv** dataset using NumPy. Make sure that everything works by having a look at the ndarray, like in the previous activity:

   # loading the Dataset

dataset = np.genfromtxt('./data/normal_distribution.csv', delimiter=',')

3. First, we want to use simple indexing for the second row, as we did in our first exercise. For a clearer understanding, all the elements are saved to a variable:

   # indexing the second row of the dataset (second row)

   second_row = dataset[1]

   np.mean(second_row)

   The output of the preceding code is as follows:

   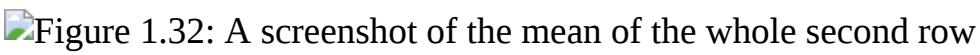 Figure 1.32: A screenshot of the mean of the whole second row

   Figure 1.32: A screenshot of the mean of the whole second row

4. Now, we need to reverse index the last row and calculate the mean of that row. Always remember that providing a negative number as the index value will index the list from the end:

   # indexing the last element of the dataset (last row)

   last_row = dataset[-1]

   np.mean(last_row)

   The output of the preceding code is as follows:

   Figure 1.33: Calculating the mean of the last row

5. Two-dimensional data can be accessed the same as with a Python List by using **[0][0]**, where the first pair of brackets accesses the row and the second one accesses the column.

   However, we can also use comma-separated notation, like **[0, 0]:**

   # indexing the first value of the second row (1st row, 1st value)

first_val_first_row = dataset[0][0]

np.mean(first_val_first_row)

The output of the preceding code is as follows:

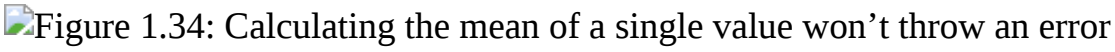
Figure 1.34: Calculating the mean of a single value won’t throw an error

Figure 1.34: Calculating the mean of a single value won't throw an error

6. The last value of the second last row can easily be accessed with reverse indexing. Remember that **-1** means the last element:

   # indexing the last value of the second to last row (we want to use the combined access syntax here)

   last_val_second_last_row = dataset[-2, -1]

   np.mean(last_val_second_last_row)

   The output of the preceding code is as follows:


Figure 1.35: Use of comma-separated notation

Figure 1.35: Use of comma-separated notation

Slicing

7. To create a 2x2 matrix that starts at the second row and second column, we use **[1:3, 1:3]**:

   # slicing an intersection of 4 elements (2x2) of the first two rows and first two columns

   subsection_2x2 = dataset[1:3, 1:3]

   np.mean(subsection_2x2)

   The output of the preceding code is as follows:

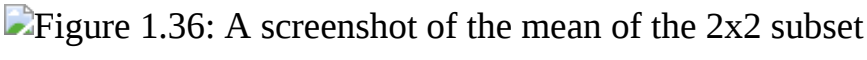
Figure 1.36: A screenshot of the mean of the 2x2 subset

Figure 1.36: A screenshot of the mean of the 2x2 subset

8. Introducing the second column into the indexing allows us to add another layer of complexity. The third value allows us to only select certain values like every other element by providing the value of 2. This means it skips the values between and only takes each second element from the used list. In this task, we want to have every other element, so we provide an indexing of **::2**, which, as we already discussed, will take every other element of the whole list:

   # selecting every second element of the fifth row

   every_other_elem = dataset[6, ::2]

   np.mean(every_other_elem)

   The output of the preceding code is as follows:

Figure 1.37: Selecting every other element of the seventh row

9. Negative numbers can also be used to reverse the elements in a slice:

   # reversing the entry order, selecting the first two rows in reversed order

   reversed_last_row = dataset[-1, ::-1]

   np.mean(reversed_last_row)

   The output of the preceding code is as follows:

Figure 1.38: Slice of the last row with the elements in reversed order

Splitting

1. Horizontally splitting our data can be done with the **hsplit** method. Note that if the dataset can't be split with the given number of slices, it will throw an

error:

splitting up our dataset horizontally on indices one third and two thirds

hor_splits = np.hsplit(dataset,(3))

2. We now need to split the first third into two equal parts vertically. There is also a **vsplit** method that does exactly this; it works the same as with **hsplit**:

   # splitting up our dataset vertically on index 2

   ver_splits = np.vsplit(hor_splits[0],(2))

3. When comparing the shapes, we can see that the subset has the required half of rows and the third half of columns:

   # requested subsection of our dataset which has only half the amount of rows and only a third of the columns

   print("Dataset", dataset.shape)

   print("Subset", ver_splits[0].shape)

   The output of the preceding code is as follows:

    Figure 1.39: Comparing the shapes of the original dataset and subset

Figure 1.39: Comparing the shapes of the original dataset and subset

Iterating

4. Looking at the given piece of code, we can see that the index is simply incremented with each element.

   This only works with one-dimensional data. If we want to index multi-dimensional data, this won't work:

   # iterating over whole dataset (each value in each row)

   curr_index = 0

   for x in np.nditer(dataset):

print(x, curr_index)

curr_index += 1

The output of the preceding code is as follows:

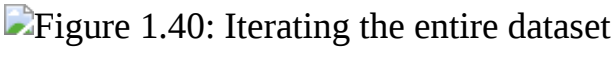

Figure 1.40: Iterating the entire dataset

5. **ndenumerate** is the right method to use for this task. In addition to the value, it returns the index. This works with multi-dimensional data, too:

   # iterating over whole dataset with indices matching the position in the dataset

   for index, value in np.ndenumerate(dataset):

   print(index, value)

   The output of the preceding code is as follows:

   

Figure 1.41: Enumerating the dataset with multi-dimensional data

Congratulations! We've already covered most of the basic data wrangling methods for NumPy. In the next activity, we'll take a look at more advanced features that will give you the tools to get better insights into your data.

Activity 3: Filtering, Sorting, Combining, and Reshaping

Solution:

Let's use the filtering features of NumPy for sorting, stacking, combining, and reshaping our data:

1. Import the necessary libraries:

   # importing the necessary dependencies

import numpy as np

2. Load the **normal_distribution.csv** dataset using NumPy. Make sure that everything works by having a look at the ndarray, like in the previous activity:

   # loading the Dataset

   dataset = np.genfromtxt('./data/normal_distribution.csv', delimiter=',')

   Filtering

3. Getting values greater than 105 can be done by supplying the condition in the brackets:

   # values that are greater than 105

   vals_greater_five = dataset[dataset > 105]

4. To use more complex conditions, we might want to use the **extract** method of NumPy. However, we can also have the same checks with the bracket notation:

   # values that are between 90 and 95

   vals_between_90_95 = np.extract((dataset > 90) & (dataset < 95), dataset)

5. The **where** method of **numpy** allows us to only get **indices (rows, cols)** for each of the matching values. In this task, we want to print them out nicely. We can combine **rows** with the respective **cols using List comprehension**. In this example, we have simply added the column to the respective row:

   # indices of values that have a delta of less than 1 to 100

   rows, cols = np.where(abs(dataset - 100) < 1)

   one_away_indices = [[rows[index], cols[index]] for (index, _) in np.ndenumerate(rows)]

   Note

List comprehensions are Python's way of mapping over data. They're a handy notation for creating a new list with some operation applied to every element of the old list.

For example, if we want to double the value of every element in this list, **list = [1, 2, 3, 4, 5]**, we would use list comprehensions like this:**doubled_list= [x*x for x in list]**. This would give us the following list: **[1, 4, 9, 16, 25]**. To get a better understanding of list comprehensions, please visit https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions.

Sorting

6. Sorting each row in our dataset works by using the **sort** method. As described in the notebook, this will always take the last axis, which in this case is the right one to sort per row:

   # values sorted for each row

   row_sorted = np.sort(dataset)

7. With multi-dimensional data, we can use the **axis** parameter to define which dataset should be sorted. **0**, in this case, means column-based sorting:

   # values sorted for each column

   col_sorted = np.sort(dataset, axis=0)

8. If we want to keep the order of our dataset and only want to know which indices the values in a sorted dataset would have, we can use **argsort**. In combination with fancy indexing, we can get access to sorted elements easily:

   # indices of positions for each row

   index_sorted = np.argsort(dataset)

Combining

9. Use combining features to add the second half of the first column back together, add the second column to our combined dataset, and add the third column to our combined dataset.

```
# split up dataset from activity03

thirds = np.hsplit(dataset, (3))

halfed_first = np.vsplit(thirds[0], (2))

# this is the part we've sent the client in activity03

halfed_first[0]
```

The output of the preceding code is as follows:


Figure 1.42: Splitting the dataset

Figure 1.42: Splitting the dataset

10. Depending on which way we want to combine our data, we have to either use **vstack** or **hstack**. Both take a list of datasets to stack together:

    ```
    # adding the second half of the first column to the data

    first_col = np.vstack([halfed_first[0], halfed_first[1]])
    ```

11. After vstacking the second half of our split dataset, we have one third of our initial dataset stacked together again. We now want to add the other two remaining datasets to our **first_col** dataset. We can do this by using the **hstack** method, which combines our already combined **first_col** with the second of the three split datasets:

    ```
    # adding the second column to our combined dataset

    first_second_col = np.hstack([first_col, thirds[1]])
    ```

12. To reassemble our initial dataset, one third is still missing. We can **hstack** the last one-third column onto our dataset, which is the same thing we did with our second-third column in the previous step:

    ```
    # adding the third column to our combined dataset

    full_data = np.hstack([first_second_col, thirds[2]])
    ```

Reshaping

13. The first subtask is to reshape our dataset into a single list. This is done using the **reshape** method:

    # reshaping to a list of values

    single_list = np.reshape(dataset, (1, -1))

14. If we supply a -1 for the dimension we don't know about, NumPy figures this dimension out itself:

    # reshaping to a matrix with two columns

    two_col_dataset = dataset.reshape(-1, 2)

Activity 4: Using pandas to Compute the Mean, Median, and Variance for the Given Numbers

Solution:

Let's use pandas' features such as mean, median, and variance to make some calculations on our data:

1. Import the necessary libraries:

   # importing the necessary dependencies

   import pandas as pd

2. After importing pandas, we can use the **read_csv** method to load the aforementioned dataset. We want to use the first column, containing the country names, as our index. We will use the **index_col** parameter for that. The full line should look like this:

   # loading the Dataset

   dataset = pd.read_csv('./data/world_population.csv', index_col=0)

3. First, we want to print a subset of the dataset with the first two rows. We can, again, use the Python List syntax to create a subset of the DataFrame of

the first two rows:

looking at the first two rows of the dataset

dataset[0:2]

The output of the preceding code is as follows:

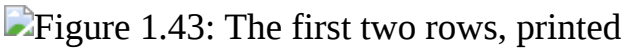
Figure 1.43: The first two rows, printed

Figure 1.43: The first two rows, printed

4. Once we know that our dataset has been successfully loaded, we can start solving the given tasks.

   The third row can be accessed by indexing **dataset.iloc[[2]]**. To get the mean of the country rather than the yearly column, we need to pass the **axis** parameter:

   # calculate the mean of the third row

   dataset.iloc[[2]].mean(axis=1)

   The output of the preceding code is as follows:

   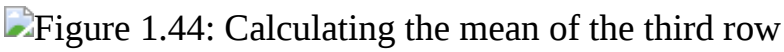
   Figure 1.44: Calculating the mean of the third row

   Figure 1.44: Calculating the mean of the third row

5. The last element of a DataFrame can, just like with NumPy ndarrays and Python Lists, be indexed using **-1** as the index. So, **dataset.iloc[[-1]]** will give us the last row:

   # calculate the mean of the last row

   dataset.iloc[[-1]].mean(axis=1)

   The output of the preceding code is as follows:

   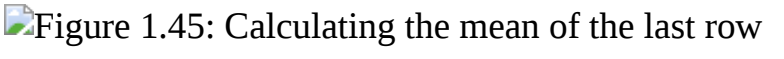
   Figure 1.45: Calculating the mean of the last row

Figure 1.45: Calculating the mean of the last row

6. Besides using **iloc** to access rows based on their index, we can also use **loc**, which works based on the index column. This can be defined by using **index_col=0** in the **read_csv** call:

   # calculate the mean of the country Germany

   dataset.loc[["Germany"]].mean(axis=1)

   The output of the preceding code is as follows:

Figure 1.46: Indexing a country and calculating the mean of Germany

7. Moving to the next set of tasks, which cover the usage of the median, we will see that the API is consistent between the different methods. This means that we have to keep providing **axis=1** to our method calls to make sure that we are aggregating for each country:

   # calculate the median of the last row

   dataset.iloc[[-1]].median(axis=1)

   The output of the preceding code is as follows:

   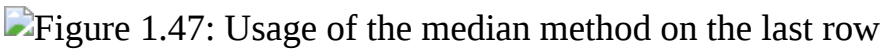
   Figure 1.47: Usage of the median method on the last row

Figure 1.47: Usage of the median method on the last row

8. Slicing rows in pandas is similar to doing so in NumPy. We can use reverse indexing to get the last three columns with **dataset[-3:]**:

   # calculate the median of the last 3 rows

   dataset[-3:].median(axis=1)

   The output of the preceding code is as follows:

   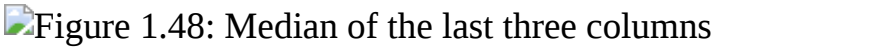
   Figure 1.48: Median of the last three columns

Figure 1.48: Median of the last three columns

9. When handling larger datasets, the order in which methods are executed matters. Think about what **head(10)** does for a moment. It simply takes your dataset and returns the first 10 rows in it, cutting down your input to the **mean()** method drastically.

   This will definitely have an impact when using more memory-intensive calculations, so keep an eye on the order:

   # calculate the median of the first 10 countries

   dataset.head(10).median(axis=1)

   The output of the preceding code is as follows:

   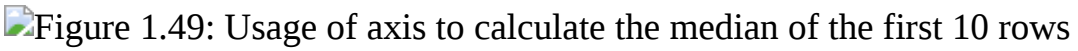
   Figure 1.49: Usage of axis to calculate the median of the first 10 rows

Figure 1.49: Usage of axis to calculate the median of the first 10 rows

10. The last method we'll cover here is the variance. Again, pandas provides us with a consistent API, which makes its usage easy. Since we only want to display the last five columns, we will make use of the **tail** method:

    # calculate the variance of the last 5 columns

    dataset.var().tail()

    The output of the preceding code is as follows:

    
    Figure 1.50: Variance among the last five columns

Figure 1.50: Variance among the last five columns

11. As mentioned in the introduction to pandas, it's interoperable with several features of NumPy.

    Here's an example of how to use NumPy's **mean** method with a pandas DataFrame. In some cases, NumPy has better functionality but pandas, with its DataFrames, has a better format:

```
# NumPy pandas interoperability

import numpy as np

print("pandas", dataset["2015"].mean())

print("numpy", np.mean(dataset["2015"]))
```

The output of the preceding code is as follows:

Figure 1.51: Using NumPy's mean method with a pandas DataFrame

Congratulations! You've completed your first activity with pandas, which showed you some of the similarities but also differences when working with NumPy and pandas. In the following activities, this knowledge will be consolidated. You'll also be introduced to more complex features and methods of pandas.

Activity 5: Indexing, Slicing, and Iterating Using pandas

Solution:

Let's use the indexing, slicing, and iterating operations to display the population density of Germany, Singapore, United States, and India for years 1970, 1990, and 2010.

Indexing

1. Import the necessary libraries:

   ```
   # importing the necessary dependencies

   import pandas as pd
   ```

2. After importing pandas, we can use the **read_csv** method to load the mentioned dataset. We want to use the first column, containing the country names, as our index. We will use the **index_col** parameter for that. The complete line should look like this:

   ```
   # loading the dataset
   ```

dataset = pd.read_csv('./data/world_population.csv', index_col=0)

3. To index the row with the **index_col** "United States", we need to use the **loc** method:

   # indexing the USA row

   dataset.loc[["United States"]].head()

   The output of the preceding code is as follows:

    Figure 1.52: Indexing United States with the loc method

   Figure 1.52: Indexing United States with the loc method

4. Reverse indexing can also be used with pandas. To index the second to last row, we need to use the **iloc** method, which accepts int-type data, meaning an index:

   # indexing the last second to last row by index

   dataset.iloc[[-2]]

   The output of the preceding code is as follows:

   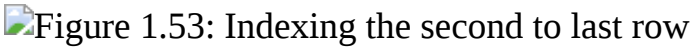 Figure 1.53: Indexing the second to last row

   Figure 1.53: Indexing the second to last row

5. Columns are indexed using their header. This is the first line of the CSV file. To get the column with the header **2000**, we can use normal indexing. Remember, the **head()** method simply returns the first five rows:

   # indexing the column of 2000 as a Series

   dataset["2000"].head()

   The output of the preceding code is as follows:

   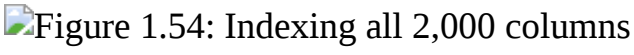 Figure 1.54: Indexing all 2,000 columns

Figure 1.54: Indexing all 2,000 columns

6. Since the double brackets notation again returns a DataFrame, we can chain method calls to get distinct elements. To get the population density of India in 2000, we first want to get the data for the year 2000 and only then select India using the **loc()** method:

   # indexing the population density of India in 2000 (Dataframe)

   dataset[["2000"]].loc[["India"]]

   The output of the preceding code is as follows:

   
   Figure 1.55: Getting the population density of India in 2000

Figure 1.55: Getting the population density of India in 2000

7. If we want to only retrieve a Series object, we have to replace the double brackets with single ones. This will give us the distinct value instead of the new DataFrame:

   # indexing the population density of India in 2000 (Series)

   dataset["2000"].loc["India"]

   The output of the preceding code is as follows:

   
   Figure 1.56: India’s population density in the year 2000

Figure 1.56: India's population density in the year 2000

Slicing

8. To create a slice with the rows 2 to 5, we have to use the **iloc()** method again. We can simply provide the same slicing syntax as NumPy:

   # slicing countries of rows 2 to 5

   dataset.iloc[1:5]

   The output of the preceding code is as follows:

Figure 1.57: The countries in rows 2 to 5

9. Using the **loc()** method, we can also access dedicated rows by their **index_col** (which was defined in the **read_csv** call). To get several rows in a new DataFrame, we can use the nested brackets to provide a list of elements:

   ```
   # slicing rows Germany, Singapore, United States, and India

   dataset.loc[["Germany", "Singapore", "United States", "India"]]
   ```

   The output of the preceding code is as follows:

   

   Figure 1.58: Slicing countries Germany, Singapore, United States, and India

10. Since the double bracket queries return new DataFrames, we can chain methods and therefore access distinct subframes of our data:

    ```
    # slicing a subset of Germany, Singapore, United States, and India

    # for years 1970, 1990, 2010 <

    country_list = ["Germany", "Singapore", "United States", "India"]

    dataset.loc[country_list][["1970", "1990", "2010"]]
    ```

    The output of the preceding code is as follows:

    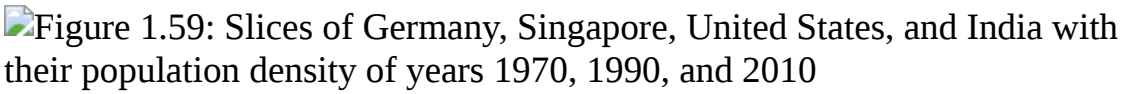

    Figure 1.59: Slices of Germany, Singapore, United States, and India with their population density of years 1970, 1990, and 2010

Iterating

11. To iterate our dataset and print out the countries up until **Angola**, we can use the **iterrows()** method. The index will be the name of our row, and the row will hold all columns:

    # iterating over the first three countries (row by row)

    for index, row in dataset.iterrows():

    # only printing the rows until Angola

    if index == 'Angola':

    break

    print(index, '\n', row[["Country Code", "1970", "1990", "2010"]], '\n')

    The output of the preceding code is as follows:

    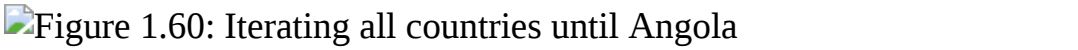 Figure 1.60: Iterating all countries until Angola

Figure 1.60: Iterating all countries until Angola

Congratulations! We've already covered most of the basic data wrangling methods using pandas. In the next activity, we'll take a look at more advanced features such as filtering, sorting, and reshaping to prepare you for the next chapter.

Activity 6: Filtering, Sorting, and Reshaping

Solution:

Let's use pandas to filter, sort, and reshape our data.

Filtering

1. Import the necessary libraries:

    # importing the necessary dependencies

    import pandas as pd

2. After importing pandas, we can use the **read_csv** method to load the aforementioned dataset. We want to use the first column, containing the country names, as our index. We will use the **index_col** parameter for that. The complete line should look like this:

   ```
   # loading the dataset

   dataset = pd.read_csv('./data/world_population.csv', index_col=0)
   ```

3. Instead of using the bracket syntax, we can also use the **filter** method to filter for specific items. Here, we also provide a list of elements that should be kept:

   ```
   # filtering columns 1961, 2000, and 2015

   dataset.filter(items=["1961", "2000", "2015"]).head()
   ```

   The output of the preceding code is as follows:

   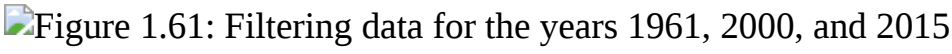
   Figure 1.61: Filtering data for the years 1961, 2000, and 2015

   Figure 1.61: Filtering data for the years 1961, 2000, and 2015

4. If we want to filter for specific values in a specific column, we can use conditions. To get all countries that had a higher population density than 500 in 2000, we simply pass this condition in brackets:

   ```
   # filtering countries that had a greater population density than 500 in 2000

   dataset[(dataset["2000"] > 500)][["2000"]]
   ```

   The output of the preceding code is as follows:

   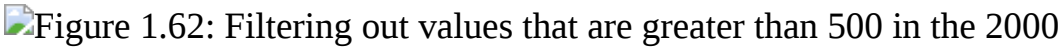
   Figure 1.62: Filtering out values that are greater than 500 in the 2000 column

   Figure 1.62: Filtering out values that are greater than 500 in the 2000 column

5. One powerful parameter of the **filter** method is **regex**. This allows us to search for arbitrary columns or rows (depending on the index given) that

match a certain **regex**. To get all the columns that start with 2, we can simply pass **^2**, which means it starts with **2**:

```
# filtering for years 2000 and later

dataset.filter(regex="^2", axis=1).head()
```

The output of the preceding code is as follows:


Figure 1.63: Retrieving all columns starting with 2

Figure 1.63: Retrieving all columns starting with 2

6. Using the **axis** parameter, we can decide on which dimension the filtering should happen. To filter the rows instead of the columns, we can pass **axis=0**. This will be helpful for situations like when we want to filter all the rows that start with **A**:

```
# filtering countries that start with A

dataset.filter(regex="^A", axis=0).head()
```

The output of the preceding code is as follows:

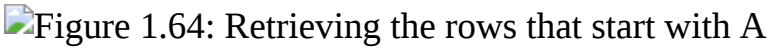
Figure 1.64: Retrieving the rows that start with A

Figure 1.64: Retrieving the rows that start with A

7. If we want all the rows or columns that contain some specific value or character, we can use the **like** query. For example, if we want to have only countries that contain the word "land", such as Switzerland:

```
# filtering countries that contain the word land

dataset.filter(like="land", axis=0).head()
```

The output of the preceding code is as follows:

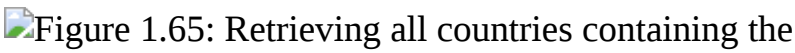
Figure 1.65: Retrieving all countries containing the word “land”

Figure 1.65: Retrieving all countries containing the word "land"

Sorting

8. Sorting in pandas can be done by using the **sort_values** or **sort_index** methods. If we want to get the countries with the lowest population density for a specific year, we can sort by this specific column:

   # values sorted by column 1961

   dataset.sort_values(by=["1961"])[["1961"]].head(10)

   The output of the preceding code is as follows:

   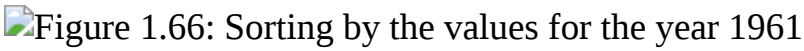 Figure 1.66: Sorting by the values for the year 1961

   Figure 1.66: Sorting by the values for the year 1961

9. Just for comparison, we want to do the same sorting for **2015**. This will give us some nice insights into our data. We can see that the order of the countries with the lowest population density changed a bit, but that the first three entries are unchanged:

   # values sorted by column 2015

   dataset.sort_values(by=["2015"])[["2015"]].head(10)

   The output of the preceding code is as follows:

    Figure 1.67: Sorting based on the values of 2015

   Figure 1.67: Sorting based on the values of 2015

10. The default sorting order is **ascending**. This means that we have to provide a separate parameter if we want to sort in **descending** order, showing us the biggest values first:

    # values sorted by column 2015 in descending order

    dataset.sort_values(by=["2015"], ascending=False)[["2015"]].head(10)

    The output of the preceding code is as follows:

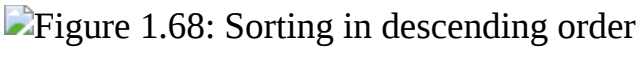

Figure 1.68: Sorting in descending order

Reshaping

11. As we mentioned before, reshaping data with pandas can be a very complex task, so we'll only do an easier reshape. We want to get DataFrames where the columns are **country codes** and the only row is the year **2015**. Since we only have one **2015** label, we need to duplicate it as many times as our dataset length. This will lead to every value receiving the 2015 row index:

    ```
    # reshaping to 2015 as row and country codes as columns

    dataset_2015 = dataset[["Country Code", "2015"]]

    dataset_2015.pivot(index=["2015"] * len(dataset_2015), columns="Country Code", values="2015")
    ```

    The output of the preceding code is as follows:

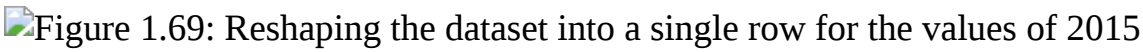

Figure 1.69: Reshaping the dataset into a single row for the values of 2015

You've now completed the topic about pandas, which concludes this chapter. We've learned about the basic tools that help you wrangle and work with data. pandas itself is an incredibly powerful and heavily used tool for data wrangling and understanding.

Chapter 2: All You Need to Know about Plots

Activity 7: Employee Skill Comparison

Solution:

1. Bar charts and radar charts are great for comparing multiple variables for multiple groups.

2. **Suggested response**: The bar chart is great for comparing the skill attributes of the different employees, but it is disadvantageous to get an overall impression for an employee. The radar chart is great for comparing values across both employees and skill attributes.
3. **Suggested response**:

   Bar chart: Addition of a title and labels, use of different colors

   Radar chart: Addition of a title and different colors for different employees

Activity 8: Road Accidents Occurring over Two Decades

Solution:

1. **Suggested response**: If we look at the Figure 2.20, we can see that for the year 2015, the number of accidents have reduced to 100 in the months of January and July. Around 300 accidents have occurred in the months of April and October.
2. **Suggested response**: If we look at the trend for each month, that is, January, April, July, and October for the past two decades, we can see a decreasing trend in the number of accidents taking place in January.

Design practices:

- Select colors and contrasts that will be easily visible to individuals with vision problems so that your plots are more inclusive.

Activity 9: Smartphone Sales Units

Solution:

1. **Suggested response**: If we compare the performance of each manufacturer in the third and fourth quarters, we come to the conclusion that Apple has performed exceptionally well. Their sales units have risen at a higher rate from the third quarter to the fourth quarter for both 2016 and 2017, when compared with that of other manufacturers.
2. **Suggested response**: If we look at the trends in the sales units of each manufacturer, we can see that after the third quarter of 2017, the sales units of all the companies except Xiaomi have shown an inconsistency. If we look

at the performance of Xiaomi, there has been an upward trend after the first quarter of the year 2017.

Activity 10: Frequency of Trains during Different Time Intervals

Solution:

1. **Suggested response**: If we focus on the preceding histogram, we can clearly identify that most trains arrive during 6 pm and 8 pm.
2. **Suggested response**: The histogram will look as follows if the number of trains between 4-6 pm are increased by 50:


Figure 2.45: Frequency of trains increased by 50 between 4-6 pm

Figure 2.45: Frequency of trains increased by 50 between 4-6 pm

Activity 11: Identifying the Ideal Visualization

Solution:

First visualization: **suggested response:**

1. The proposed visualization has multiple faults: first, a pie chart is supposed to show part-of-a-whole relations, which is not the case for this task since we only consider the top 30 YouTubers. Second, 30 values are too many to visualize within a pie chart. Third, the slices are not ordered according to their size. Also, it is difficult to quantify the slices as there is no unit of measurement specified. On the other hand, in the following horizontal bar chart, it is easier to tell the number of subscribers in millions for each YouTube channel:

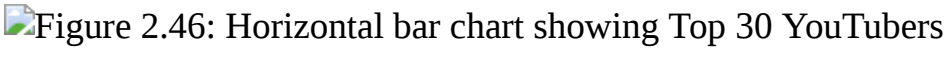
Figure 2.46: Horizontal bar chart showing Top 30 YouTubers

Figure 2.46: Horizontal bar chart showing Top 30 YouTubers

Second visualization: **suggested response:**

1. This is also an example of using the wrong chart type. A line chart was used to compare different categories that do not have any temporal relation.

Furthermore, information such as legends and labels are missing. The following figure shows how the data should had been represented using comparative bar chart:

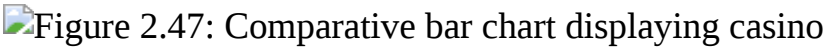
Figure 2.47: Comparative bar chart displaying casino data for two days

Figure 2.47: Comparative bar chart displaying casino data for two days

Chapter 3: A Deep Dive into Matplotlib

Activity 12: Visualizing Stock Trends by Using a Line Plot

Solution:

Let's visualize a stock trend by using a line plot:

1. Open the **activity12_solution.ipynb** Jupyter Notebook from the **Lesson03** folder to implement this activity.

   Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab**

2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   ```
   # Import statements

   import matplotlib.pyplot as plt

   import numpy as np

   import pandas as pd

   %matplotlib inline
   ```

3. Use pandas to read the data located in the data folder. The **read_csv()** function reads a .csv file into **DataFrame**:

   ```
   # load datasets
   ```

```
google = pd.read_csv('./data/GOOGL_data.csv')

facebook = pd.read_csv('./data/FB_data.csv')

apple = pd.read_csv('./data/AAPL_data.csv')

amazon = pd.read_csv('./data/AMZN_data.csv')

microsoft = pd.read_csv('./data/MSFT_data.csv')
```

4. Use **Matplotlib** to create a **line chart** that visualizes the closing prices for the past five years (whole data sequence) for all five companies. Add labels, titles, and a legend to make the visualization self-explanatory. Use the **plt.grid()** function to add a grid to your plot:

```
# Create figure

plt.figure(figsize=(16, 8), dpi=300)

# Plot data

plt.plot('date', 'close', data=google, label='Google')

plt.plot('date', 'close', data=facebook, label='Facebook')

plt.plot('date', 'close', data=apple, label='Apple')

plt.plot('date', 'close', data=amazon, label='Amazon')

plt.plot('date', 'close', data=microsoft, label='Microsoft')

# Specify ticks for x and y axis

plt.xticks(np.arange(0, 1260, 40), rotation=70)

plt.yticks(np.arange(0, 1450, 100))

# Add title and label for y-axis

plt.title('Stock trend', fontsize=16)

plt.ylabel('Closing price in $', fontsize=14)
```

```
# Add grid

plt.grid()

# Add legend

plt.legend()

# Show plot

plt.show()
```

Activity 13: Creating a Bar Plot for Movie Comparison

Solution:

Let's create a bar plot for comparing the ratings of different movies:

1. Open the **activity13_solution.ipynb** Jupyter Notebook from the **Lesson03** folder to implement this activity.

   Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab.**

2. Import the necessary modules and enable plotting within a Jupyter Notebook.;

   ```
   # Import statements

   import numpy as np

   import pandas as pd

   import matplotlib.pyplot as plt

   %matplotlib inline
   ```

3. Use pandas to read the data located in the data folder:

   ```
   # Load dataset

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

4. Use Matplotlib to create a visually appealing bar plot comparing the two scores for all five movies. Use the movie titles as labels for the x-axis. Use percentages in an interval of 20 for the y-axis and minor ticks in an interval of 5. Add a legend and a suitable title to the plot:

```
# Create figure

plt.figure(figsize=(10, 5), dpi=300)

# Create bar plot

pos = np.arange(len(movie_scores['MovieTitle']))

width = 0.3

plt.bar(pos - width / 2, movie_scores['Tomatometer'], width,
label='Tomatometer')

plt.bar(pos + width / 2, movie_scores['AudienceScore'], width,
label='Audience Score')

# Specify ticks

plt.xticks(pos, rotation=10)

plt.yticks(np.arange(0, 101, 20))

# Get current Axes for setting tick labels and horizontal grid

ax = plt.gca()

# Set tick labels

ax.set_xticklabels(movie_scores['MovieTitle'])

ax.set_yticklabels(['0%', '20%', '40%', '60%', '80%', '100%'])

# Add minor ticks for y-axis in the interval of 5

ax.set_yticks(np.arange(0, 100, 5), minor=True)

# Add major horizontal grid with solid lines
```

```
ax.yaxis.grid(which='major')

# Add minor horizontal grid with dashed lines

ax.yaxis.grid(which='minor', linestyle='--')

# Add title

plt.title('Movie comparison')

# Add legend

plt.legend()

# Show plot

plt.show()
```

5. Some functions require you to explicitly specify the **axes**. To get the reference to the current axes, use **ax = plt.gca()**. Ticks can be specified using **plt.xticks([xticks])** and **plt.yticks([yticks])** for the x-axis and y-axis, respectively. **Axes.set_xticklabels([labels])** and **Axes.set_yticklabels([labels])** can be used to set tick labels. To add minor y-ticks, use **Axes.set_yticks([ticks], minor=True)**. To add a horizontal grid for major ticks, use **Axes.yaxis.grid(which='major')**, and to add a dashed horizontal grid for minor ticks, use **Axes.yaxis.grid(which='minor', linestyle='--')**.

Activity 14: Creating a Stacked Bar Plot to Visualize Restaurant Performance

Solution:

Let's create a stacked bar chart to visualize the performance of a restaurant:

1. Open the **activity14_solution.ipynb** Jupyter Notebook from the **Lesson03** folder to implement this activity.

   Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab.**

2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   ```
   # Import statements

   import pandas as sb

   import numpy as np

   import matplotlib.pyplot as plt

   import seaborn as sns

   %matplotlib inline
   ```

3. Load the dataset:

   ```
   # Load dataset

   bills = sns.load_dataset('tips')
   ```

4. Use the given dataset and create a matrix where the elements contain the sum of the total bills for each day and smokers/non-smokers:

   ```
   days = ['Thur', 'Fri', 'Sat', 'Sun']

   days_range = np.arange(len(days))

   smoker = ['Yes', 'No']

   bills_by_days = [bills[bills['day'] == day] for day in days]

   bills_by_days_smoker = [[bills_by_days[day][bills_by_days[day]['smoker'] == s] for s in smoker] for day in days_range]

   total_by_days_smoker = [[bills_by_days_smoker[day][s]['total_bill'].sum() for s in range(len(smoker))] for day in days_range]

   totals = np.asarray(total_by_days_smoker)
   ```

   Here, the **asarray()** function is used to convert any given input into an array.

5. Create a stacked bar plot, stacking the summed total bills separated by smoker and non-smoker for each day. Add a legend, labels, and a title:

```
# Create figure

plt.figure(figsize=(10, 5), dpi=300)

# Create stacked bar plot

plt.bar(days_range, totals[:, 0], label='Smoker')

plt.bar(days_range, totals[:, 1], bottom=totals[:, 0], label='Non-smoker')

# Add legend

plt.legend()

# Add labels and title

plt.xticks(days_range)

ax = plt.gca()

ax.set_xticklabels(days)

ax.yaxis.grid()

plt.ylabel('Daily total sales in $')

plt.title('Restaurant performance')

# Show plot

plt.show()
```

Activity 15: Comparing Smartphone Sales Units Using a Stacked Area Chart

Solution:

Let's compare the sales units of smartphone manufacturers using a stacked area chart:

1. Open the **activity15_solution.ipynb** Jupyter Notebook from the **Lesson03** folder to implement this activity.

   Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab.**

2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   ```
   # Import statements

   import pandas as pd

   import numpy as np

   import matplotlib.pyplot as plt

   %matplotlib inline
   ```

3. Use pandas to read the data located in the data folder:

   ```
   # Load dataset

   sales = pd.read_csv('./data/smartphone_sales.csv')
   ```

4. Create a visually appealing stacked area chart. Add a legend, labels, and a title:

   ```
   # Create figure

   plt.figure(figsize=(10, 6), dpi=300)

   # Create stacked area chart

   labels = sales.columns[1:]

   plt.stackplot('Quarter', 'Apple', 'Samsung', 'Huawei', 'Xiaomi', 'OPPO',
   data=sales, labels=labels)

   # Add legend

   plt.legend()
   ```

```
# Add labels and title

plt.xlabel('Quarters')

plt.ylabel('Sales units in thousands')

plt.title('Smartphone sales units')

# Show plot

plt.show()
```

Activity 16: Using a Histogram and a Box Plot to Visualize the Intelligence Quotient

Solution:

Let's visualize the intelligence quotient of different groups using a histogram and a box plot:

1. Open the **activity16_solution.ipynb** Jupyter Notebook from the **Lesson03** folder to implement this activity.

   Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab.**

2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   ```
   # Import statements

   import numpy as np

   import matplotlib.pyplot as plt

   %matplotlib inline
   ```

3. Use the following code to generate a sample of IQ scores:

   ```
   # IQ samples
   ```

iq_scores = [126, 89, 90, 101, 102, 74, 93, 101, 66, 120, 108, 97, 98, 105, 119, 92, 113, 81, 104, 108, 83, 102, 105, 111, 102, 107, 103, 89, 89, 110, 71, 110, 120, 85, 111, 83, 122, 120, 102, 84, 118, 100, 100, 114, 81, 109, 69, 97, 95, 106, 116, 109, 114, 98, 90, 92, 98, 91, 81, 85, 86, 102, 93, 112, 76, 89, 110, 75, 100, 90, 96, 94, 107, 108, 95, 96, 96, 114, 93, 95, 117, 141, 115, 95, 86, 100, 121, 103, 66, 99, 96, 111, 110, 105, 110, 91, 112, 102, 112, 75]

4. Plot a histogram with 10 bins for the given IQ scores. IQ scores are normally distributed with a mean of 100 and a standard deviation of 15. Visualize the mean as a vertical solid red line, and the standard deviation using dashed vertical lines. Add labels and a title:

```
# Create figure

plt.figure(figsize=(6, 4), dpi=150)

# Create histogram

plt.hist(iq_scores, bins=10)

plt.axvline(x=100, color='r')

plt.axvline(x=115, color='r', linestyle= '--')

plt.axvline(x=85, color='r', linestyle= '--')

# Add labels and title

plt.xlabel('IQ score')

plt.ylabel('Frequency')

plt.title('IQ scores for a test group of a hundred adults')

# Show plot

plt.show()
```

5. Create a box plot to visualize the same IQ scores. Add labels and a title:

```
# Create figure
```

```
plt.figure(figsize=(6, 4), dpi=150)
# Create histogram
plt.boxplot(iq_scores)
# Add labels and title
ax = plt.gca()
ax.set_xticklabels(['Test group'])
plt.ylabel('IQ score')
plt.title('IQ scores for a test group of a hundred adults')
# Show plot
plt.show()
```

6. The following are IQ scores for different test groups:

```
group_a = [118, 103, 125, 107, 111, 96, 104, 97, 96, 114, 96, 75, 114,
107, 87, 117, 117, 114, 117, 112, 107, 133, 94, 91, 118, 110,
117, 86, 143, 83, 106, 86, 98, 126, 109, 91, 112, 120, 108,
111, 107, 98, 89, 113, 117, 81, 113, 112, 84, 115, 96, 93,
128, 115, 138, 121, 87, 112, 110, 79, 100, 84, 115, 93, 108,
130, 107, 106, 106, 101, 117, 93, 94, 103, 112, 98, 103, 70,
139, 94, 110, 105, 122, 94, 94, 105, 129, 110, 112, 97, 109,
121, 106, 118, 131, 88, 122, 125, 93, 78]
group_b = [126, 89, 90, 101, 102, 74, 93, 101, 66, 120, 108, 97, 98,
105, 119, 92, 113, 81, 104, 108, 83, 102, 105, 111, 102, 107,
103, 89, 89, 110, 71, 110, 120, 85, 111, 83, 122, 120, 102,
```

84, 118, 100, 100, 114, 81, 109, 69, 97, 95, 106, 116, 109,

114, 98, 90, 92, 98, 91, 81, 85, 86, 102, 93, 112, 76,

89, 110, 75, 100, 90, 96, 94, 107, 108, 95, 96, 96, 114,

93, 95, 117, 141, 115, 95, 86, 100, 121, 103, 66, 99, 96,

111, 110, 105, 110, 91, 112, 102, 112, 75]

group_c = [108, 89, 114, 116, 126, 104, 113, 96, 69, 121, 109, 102, 107,

122, 104, 107, 108, 137, 107, 116, 98, 132, 108, 114, 82, 93,

89, 90, 86, 91, 99, 98, 83, 93, 114, 96, 95, 113, 103,

81, 107, 85, 116, 85, 107, 125, 126, 123, 122, 124, 115, 114,

93, 93, 114, 107, 107, 84, 131, 91, 108, 127, 112, 106, 115,

82, 90, 117, 108, 115, 113, 108, 104, 103, 90, 110, 114, 92,

101, 72, 109, 94, 122, 90, 102, 86, 119, 103, 110, 96, 90,

110, 96, 69, 85, 102, 69, 96, 101, 90]

group_d = [93, 99, 91, 110, 80, 113, 111, 115, 98, 74, 96, 80, 83,

102, 60, 91, 82, 90, 97, 101, 89, 89, 117, 91, 104, 104,

102, 128, 106, 111, 79, 92, 97, 101, 106, 110, 93, 93, 106,

108, 85, 83, 108, 94, 79, 87, 113, 112, 111, 111, 79, 116,

104, 84, 116, 111, 103, 103, 112, 68, 54, 80, 86, 119, 81,

84, 91, 96, 116, 125, 99, 58, 102, 77, 98, 100, 90, 106,

109, 114, 102, 102, 112, 103, 98, 96, 85, 97, 110, 131, 92,

79, 115, 122, 95, 105, 74, 85, 85, 95]

7. Create a box plot for each of the IQ scores of different test groups. Add labels and a title:

   ```
   # Create figure

   plt.figure(figsize=(6, 4), dpi=150)

   # Create histogram

   plt.boxplot([group_a, group_b, group_c, group_d])

   # Add labels and title

   ax = plt.gca()

   ax.set_xticklabels(['Group A', 'Group B', 'Group C', 'Group D'])

   plt.ylabel('IQ score')

   plt.title('IQ scores for different test groups')

   # Show plot

   plt.show()
   ```

Activity 17: Using a Scatter Plot to Visualize Correlation between Various Animals

Solution:

Let's visualize correlation between various animals with the help of a scatter plot:

1. Open the **activity17_solution.ipynb** Jupyter Notebook from the **Lesson03** folder to implement this activity.

   Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab.**

2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   ```
   # Import statements
   ```

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

%matplotlib inline
```

3. Use pandas to read the data located in the data folder:

```
# Load dataset

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

4. The given dataset is not complete. Filter the data so that you end up with samples containing a body mass and a maximum longevity. Sort the data according to the animal class. Here, the **isfinite()** function checks for the finiteness of the given element:

```
# Preprocessing

longevity = 'Maximum longevity (yrs)'

mass = 'Body mass (g)'

data = data[np.isfinite(data[longevity]) & np.isfinite(data[mass])]

# Sort according to class

amphibia = data[data['Class'] == 'Amphibia']

aves = data[data['Class'] == 'Aves']

mammalia = data[data['Class'] == 'Mammalia']

reptilia = data[data['Class'] == 'Reptilia']
```

5. Create a scatter plot visualizing the correlation between the body mass and the maximum longevity. Use different colors for grouping data samples according to their class. Add a legend, labels, and a title. Use a log scale for both the x-axis and y-axis:

```
# Create figure
plt.figure(figsize=(10, 6), dpi=300)
# Create scatter plot
plt.scatter(amphibia[mass], amphibia[longevity], label='Amphibia')
plt.scatter(aves[mass], aves[longevity], label='Aves')
plt.scatter(mammalia[mass], mammalia[longevity], label='Mammalia')
plt.scatter(reptilia[mass], reptilia[longevity], label='Reptilia')
# Add legend
plt.legend()
# Log scale
ax = plt.gca()
ax.set_xscale('log')
ax.set_yscale('log')
# Add labels
plt.xlabel('Body mass in grams')
plt.ylabel('Maximum longevity in years')
# Show plot
plt.show()
```

Activity 18: Creating a Scatter Plot with Marginal Histograms

Solution:

1. Open the **activity18_solution.ipynb** Jupyter Notebook from the **Lesson03** folder to implement this activity.

Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab.**

2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   ```
   # Import statements

   import pandas as pd

   import numpy as np

   import matplotlib.pyplot as plt

   %matplotlib inline
   ```

3. Use pandas to read the data located in the data folder:

   ```
   # Load dataset

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

4. The given dataset is not complete. Filter the data so that you end up with samples containing a body mass and a maximum longevity. Select all the samples of the **aves** class and with a body mass smaller than 20,000:

   ```
   # Preprocessing

   longevity = 'Maximum longevity (yrs)'

   mass = 'Body mass (g)'

   data = data[np.isfinite(data[longevity]) & np.isfinite(data[mass])]

   # Sort according to class

   aves = data[data['Class'] == 'Aves']

   aves = data[data[mass] < 20000]
   ```

5. Create a figure with a constrained layout. Create a **gridspec** of size 4x4. Create a scatter plot of size 3x3 and marginal histograms of size 1x3 and

3x1. Add labels and a figure title:

```
# Create figure

fig = plt.figure(figsize=(8, 8), dpi=150, constrained_layout=True)

# Create gridspec

gs = fig.add_gridspec(4, 4)

# Specify subplots

histx_ax = fig.add_subplot(gs[0, :-1])

histy_ax = fig.add_subplot(gs[1:, -1])

scatter_ax = fig.add_subplot(gs[1:, :-1])

# Create plots

scatter_ax.scatter(aves[mass], aves[longevity])

histx_ax.hist(aves[mass], bins=20, density=True)

histx_ax.set_xticks([])

histy_ax.hist(aves[longevity], bins=20, density=True,
orientation='horizontal')

histy_ax.set_yticks([])

# Add labels and title

plt.xlabel('Body mass in grams')

plt.ylabel('Maximum longevity in years')

fig.suptitle('Scatter plot with marginal histograms')

# Show plot

plt.show()
```

Activity 19: Plotting Multiple Images in a Grid

Solution:

1. Open the **activity19_solution.ipynb** Jupyter Notebook from the **Lesson03** folder to implement this activity.

   Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab.**

2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   ```
   # Import statements

   import os

   import numpy as np

   import matplotlib.pyplot as plt

   import matplotlib.image as mpimg

   %matplotlib inline
   ```

3. Load all four images from the data folder:

   ```
   # Load images

   img_filenames = os.listdir('data')

   imgs = [mpimg.imread(os.path.join('data', img_filename)) for img_filename in img_filenames]
   ```

4. Visualize the images in a 2x2 grid. Remove the axes and give each image a label:

   ```
   # Create subplot

   fig, axes = plt.subplots(2, 2)

   fig.figsize = (6, 6)
   ```

```
fig.dpi = 150

axes = axes.ravel()

# Specify labels

labels = ['coast', 'beach', 'building', 'city at night']

# Plot images

for i in range(len(imgs)):

axes[i].imshow(imgs[i])

axes[i].set_xticks([])

axes[i].set_yticks([])

axes[i].set_xlabel(labels[i])
```

Chapter 4: Simplifying Visualizations Using Seaborn

Activity 20: Comparing IQ Scores for Different Test Groups by Using a Box Plot

Solution:

Let's compare IQ scores among different test groups using the Seaborn library:

1. Open the **activity20_solution.ipynb** Jupyter Notebook from the **Lesson04** folder to implement this activity. Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab**.
2. Import the necessary modules and enable plotting within a Jupyter Notebook.

```
%matplotlib inline

import numpy as np
```

```
import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns
```

3. Use the pandas **read_csv()** function to read the data located in the data folder:

```
mydata = pd.read_csv("./data/scores.csv")
```

4. Access the data of each test group in the column. Convert them into a list using the **tolist()** method. Once the data of each test group has been converted into a list, assign this list to variables of each respective test groups:

```
group_a = mydata[mydata.columns[0]].tolist()

group_b = mydata[mydata.columns[1]].tolist()

group_c = mydata[mydata.columns[2]].tolist()

group_d = mydata[mydata.columns[3]].tolist()
```

5. Print the variables of each group to check whether the data inside it is converted into a list. This can be done with the help of the **print()** function:

```
print(group_a)
```

Data values of Group A are shown in the following figure:

Figure 4.35: Values of Group A

```
print(group_b)
```

Data values of Group B are shown in the following figure:

Figure 4.36: Values of Group B

print(group_c)

Data values of Group C are shown in the following figure:

Figure 4.37: Values of Group C

print(group_d)

Data values of Group D are shown in the following figure:

Figure 4.38: Values of Group D

6. Once we have the data for each test group, we need to construct a DataFrame from this given data. This can be done with the help of the **pd.DataFrame()** function, which is provided by pandas.

   data = pd.DataFrame({'Groups': ['Group A'] * len(group_a) + ['Group B'] * len(group_b) + ['Group C'] * len(group_c) + ['Group D'] * len(group_d),

   'IQ score': group_a + group_b + group_c + group_d})

7. Now, since we have the DataFrame, we need to create a box plot using the **boxplot()** function that's provided by Seaborn. Within this function, we need to specify the titles for both the axes along with the DataFrame we are using. The title for the x-axis would be **Groups** and the title for the y-axis would be **IQ score**. As far as the DataFrame is concerned, we will pass **data** as a parameter. Here, **data** is the DataFrame that we obtained from the previous step.

   plt.figure(dpi=150)

   # Set style

   sns.set_style('whitegrid')

```
# Create boxplot

sns.boxplot('Groups', 'IQ score', data=data)

# Despine

sns.despine(left=True, right=True, top=True)

# Add title

plt.title('IQ scores for different test groups')

# Show plot

plt.show()
```

The **despine()** function helps in removing the top and right spines from the plot. Here, we have also removed the left spine. Using the **title()** function, we have set the title for our plot. The **show()** function helps to visualize the plot.

From Figure 4.8, we can conclude that by using a box plot, the IQ scores of Group A are better than the other groups.

Activity 21: Using Heatmaps to Find Patterns in Flight Passengers' Data

Solution:

Let's find the patterns in the flight passengers' data with the help of a heatmap:

1. Open the **activity21_solution.ipynb** Jupyter Notebook from the **Lesson04** folder to implement this activity. Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab**.
2. Import the necessary modules and enable plotting within a Jupyter Notebook:

```
%matplotlib inline

import numpy as np
```

```
import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns
```

3. Use the **read_csv()** function of pandas to read the data located in the data folder:

```
mydata = pd.read_csv("./data/flight_details.csv")
```

4. Now, we can use the **pivot()** function to provide meaningful row and column labels to our DataFrame:

```
data = mydata.pivot("Months", "Years", "Passengers")
```

5. Use the **heatmap()** function of the Seaborn library to visualize this data. Within this function, we pass parameters such as DataFrame and colormap. Since we got data from the preceding code, we will pass it as a **DataFrame** in the **heatmap()** function. Also, we will create our own colormap and pass it as a second parameter to this function.

```
sns.set()

plt.figure(dpi=150)

sns.heatmap(data, cmap=sns.light_palette("orange", as_cmap=True, reverse=True))

plt.title("Flight Passengers from 2001 to 2012")

plt.show()
```

From Figure 4.23, we can conclude that the number of flight passengers were highest in July 2012 and August 2012.

Activity 22: Movie Comparison Revisited

Solution:

Let's compare the movie scores for five different movies by using a bar plot that's been provided by Seaborn library:

1. Open the **activity22_solution.ipynb** Jupyter Notebook from the **Lesson04** folder to implement this activity. Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab**.
2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   ```
   %matplotlib inline

   import numpy as np

   import pandas as pd

   import matplotlib.pyplot as plt

   import seaborn as sns
   ```

3. Use the **read_csv()** function of pandas to read the data located in the data folder:

   ```
   mydata = pd.read_csv("./data/movie_scores.csv")
   ```

4. Construct a DataFrame from this given data. This can be done with the help of the **pd.DataFrame()** function, which is provided by pandas. The following code gives us a better idea of this:

   ```
   movie_scores = pd.DataFrame({"Movie Title": list(mydata["MovieTitle"]) *
   2,

   "Score": list(mydata["AudienceScore"]) + list(mydata["Tomatometer"]),

   "Type": ["Audience Score"] * len(mydata["AudienceScore"]) +
   ["Tomatometer"] * len(mydata["Tomatometer"])})
   ```

5. Make use of the **barplot()** function provided by Seaborn. Provide **Movies** and **Scores** as parameters so that their data is displayed on both axes. Provide **Type** as **hue**, on the basis of which the comparison needs to be made. The last parameter needs a DataFrame as input. Thus, we provide the **movie_scores** DataFrame, which we obtained from the previous step.

   The following code provides a better understanding of this:

   ```
   sns.set()
   ```

```
plt.figure(figsize=(10, 5), dpi=300)

# Create bar plot

ax = sns.barplot("Movie Title", "Score", hue="Type", data=movie_scores)

plt.xticks(rotation=10)

# Add title

plt.title("Movies Scores comparison")

plt.xlabel("Movies")

plt.ylabel("Scores")

# Show plot

plt.show()
```

We compared the ratings of AudienceScore and Tomatometer for 5 different movies and concluded that the ratings matched for the movie **The Martian**.

Activity 23: Comparing IQ Scores for Different Test Groups by Using a Violin Plot

Solution:

Let's compare IQ scores among different test groups using the Seaborn library:

1. Open the **activity23_solution.ipynb** Jupyter Notebook from the **Lesson04** folder to implement this activity. Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab**.
2. Import the necessary modules and enable plotting within a Jupyter Notebook:

```
%matplotlib inline

import numpy as np

import pandas as pd
```

```
import matplotlib.pyplot as plt

import seaborn as sns
```

3. Use the **read_csv()** function of pandas to read the data located in the data folder:

```
mydata = pd.read_csv("./data/scores.csv")
```

4. Access the data of each test group in the column. Convert them into a list using the **tolist()** method. Once the data of each test group has been converted into a list, assign this list to the variables of each respective test group.

```
group_a = mydata[mydata.columns[0]].tolist()

group_b = mydata[mydata.columns[1]].tolist()

group_c = mydata[mydata.columns[2]].tolist()

group_d = mydata[mydata.columns[3]].tolist()
```

5. Print the variables of each group to check whether the data inside it has been converted into a list. This can be done with the help of the **print()** function:

```
print(group_a)
```

Figure 4.39: Values of Group A

```
print(group_b)
```

Figure 4.40: Values of Group B

```
print(group_c)
```

Figure 4.41: Values of Group C

print(group_d)

Figure 4.42: Values of Group D

6. Once we get the data for each test group, we need to construct a DataFrame from this given data. This can be done with the help of the **pd.DataFrame()** function that's provided by pandas.

   data = pd.DataFrame({'Groups': ['Group A'] * len(group_a) + ['Group B'] * len(group_b) + ['Group C'] * len(group_c) + ['Group D'] * len(group_d),

   'IQ score': group_a + group_b + group_c + group_d})

7. Now, since we have the DataFrame, we need to create a violin plot using the **violinplot()** function that's provided by Seaborn. Within this function, we need to specify the titles for both the axes along with the DataFrame we are using. The title for the x-axis will be **Groups** and the title for the y-axis will be **IQ score**. As far as the DataFrame is concerned, we will pass **data** as a parameter. Here, **data** is the DataFrame that we obtained from the previous step.

   plt.figure(dpi=150)

   # Set style

   sns.set_style('whitegrid')

   # Create boxplot

   sns.violinplot('Groups', 'IQ score', data=data)

   # Despine

   sns.despine(left=True, right=True, top=True)

   # Add title

```
plt.title('IQ scores for different test groups')

# Show plot

plt.show()
```

The **despine()** function helps is removing the top and right spines from the plot. Here, we have also removed the left spine. Using the **title()** function, we have set the title for our plot. The **show()** function helps to visualize the plot.

We can conclude that the IQ scores of Group A are better than those for other groups.

Activity 24: Top 30 YouTube Channels

Solution:

Let's visualize the total number of subscribers and the total number of views for the top 30 YouTube channels by using the **FacetGrid()** function that's provided by the Seaborn library:

1. Open the **activity24_solution.ipynb** Jupyter Notebook from the **Lesson04** folder to implement this activity. Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab**.
2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   ```
   %matplotlib inline

   import numpy as np

   import pandas as pd

   import matplotlib.pyplot as plt

   import seaborn as sns
   ```

3. Use the **read_csv()** function of pandas to read the data located in the data folder:

mydata = pd.read_csv("./data/youtube.csv")

4. Access the data of each test group in the column. Convert them into a list by using the **tolist()** method. Once the data of each test group has been converted into a list, assign this list to variables of each respective test group.

   channels = mydata[mydata.columns[0]].tolist()

   subs = mydata[mydata.columns[1]].tolist()

   views = mydata[mydata.columns[2]].tolist()

5. Print the variables of each group to check whether the data inside it has been converted into a list. This can be done with the help of the **print()** function:

   print(channels)

   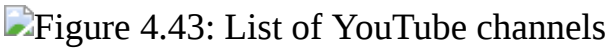
   Figure 4.43: List of YouTube channels

   Figure 4.43: List of YouTube channels

   print(subs)

   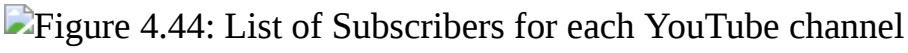
   Figure 4.44: List of Subscribers for each YouTube channel

   Figure 4.44: List of Subscribers for each YouTube channel

   print(views)

   
   Figure 4.45: List of Views for each YouTube channel

   Figure 4.45: List of Views for each YouTube channel

6. Once we get the data for **channels**, **subs**, and **views**, we need to construct a DataFrame from this given data. This can be done with the help of the **pd.DataFrame()** function that's provided by pandas.

   data = pd.DataFrame({'YouTube Channels': channels + channels, 'Subscribers in millions': subs + views, 'Type': ['Subscribers'] * len(subs) +

['Views'] * len(views)})

7. Now, since we have the DataFrame, we need to create a FacetGrid using the **FacetGrid()** function that's provided by Seaborn. Here, **data** is the DataFrame, which we obtained from the previous step.

   sns.set()

   g = sns.FacetGrid(data, col='Type', hue='Type', sharex=False, height=8)

   g.map(sns.barplot, 'Subscribers in millions', 'YouTube Channels')

   plt.show()

We can conclude that PewDiePie's YouTube channel has the highest number of subscribers, whereas T-Series has the highest number of views.

Activity 25: Linear Regression

Solution:

Let's visualize the linear relationship between maximum longevity and body mass in the regression plot by using the **regplot()** function that's provided by the Seaborn library:

1. Open the **activity25_solution.ipynb** Jupyter Notebook from the **Lesson04** folder to implement this activity. Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab**.
2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   %matplotlib inline

   import numpy as np

   import pandas as pd

   import matplotlib.pyplot as plt

   import seaborn as sns

3. Use the **read_csv()** function of pandas to read the data located in the data folder:

   mydata = pd.read_csv("./data/anage_data.csv")

4. Filter the data so that you end up with samples containing a body mass and a maximum longevity. Only consider samples for the **Mammalia** class and a body mass below 200,000. This preprocessing can be seen in the following code:

   longevity = 'Maximum longevity (yrs)'

   mass = 'Body mass (g)'

   data = mydata[mydata['Class'] == 'Mammalia']

   data = data[np.isfinite(data[longevity]) & np.isfinite(data[mass]) & (data[mass] < 200000)]

5. Once the preprocessing is done, we need plot the data using the **regplot()** function that's provided by the Seaborn library. In the following code, we have provided three parameters inside the **regplot()** function. The first two parameters are **mass** and **longevity**, wherein the body mass data will be shown in the x-axis and the maximum longevity data will be shown in the y-axis. In the third parameter, we need to provide a DataFrame called **data**, which we obtained from the previous step:

   # Create figure

   sns.set()

   plt.figure(figsize=(10, 6), dpi=300)

   # Create scatter plot

   sns.regplot(mass, longevity, data=data)

   # Show plot

   plt.show()

We can conclude that there is a linear relationship between body mass and maximum longevity for the **Mammalia** class.

Activity 26: Water Usage Revisited

Solution:

Let's visualize the water usage by using a tree map, which can be created with the help of the Squarify library:

1. Open the **activity26_solution.ipynb** Jupyter Notebook from the **Lesson 04** folder to implement this activity. Navigate to the path of this file and type in the following at the command-line terminal: **jupyter-lab**.
2. Import the necessary modules and enable plotting within a Jupyter Notebook:

   ```
   %matplotlib inline

   import numpy as np

   import pandas as pd

   import matplotlib.pyplot as plt

   import seaborn as sns

   import squarify
   ```

3. Use the **read_csv()** function of pandas to read the data located in the data folder:

   ```
   mydata = pd.read_csv("./data/water_usage.csv")
   ```

4. Create a list of labels by accessing each column from the preceding dataset. Here, the **astype('str')** function is used to cast the fetched data into a type string.

   ```
   # Create figure

   plt.figure(dpi=200)

   # Create tree map
   ```

```
labels = mydata['Usage'] + ' (' + mydata['Percentage'].astype('str') + '%)'
```

5. For creating a tree map visualization of given data, we make use of the **plot()** function of the **squarify** library. This function takes three parameters. The first parameter is the list of all percentages and the second parameter is the list of all labels, which we got in the previous step. The third parameter is the color that can be created by using the **light_pallete()** function of the Seaborn library.

```
squarify.plot(sizes=mydata['Percentage'], label=labels,
color=sns.light_palette('green', mydata.shape[0]))

plt.axis('off')

# Add title

plt.title('Water usage')

# Show plot

plt.show()
```

Chapter 5: Plotting Geospatial Data

Activity 27: Plotting Geospatial Data on a Map

Solution:

Let's plot the geospatial data on a map and find the densely populated areas for cities in Europe that have a population of more than 100k:

1. Open the **activity27.ipynb** Jupyter Notebook from the **Lesson05** folder to implement this activity.
2. Before we can start working with the data, we need to import the dependencies:

```
# importing the necessary dependencies

import numpy as np

import pandas as pd
```

import geoplotlib

3. Load the dataset using pandas:

   #loading the Dataset (make sure to have the dataset downloaded)

   Dataset = pd.read_csv('./data/world_cities_pop.csv', dtype = {'Region': np.str})

   Note

   If we import our dataset without defining the **dtype** of the **Region** column as a **String**, we will get a warning telling us that it has a mixed datatype. We can get rid of this warning by explicitly defining the type of the values in this column, which we can do by using the **dtype** parameter.

4. To see the **dtype** of each column, we can use the **dtypes** attribute of a DataFrame:

   # looking at the data types of each column

   Dataset.dtypes

   The following figure shows the output of the preceding code:

   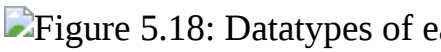 Figure 5.18: Datatypes of each column of the dataset

   Figure 5.18: Datatypes of each column of the dataset

   Note

   Here, we can see the datatypes of each column. Since the **String** type is not a primitive datatype, it's displayed as an object.

5. We use the **head()** method of pandas DataFrames to display the first five entries:

   # showing the first 5 entries of the dataset

   dataset.head()

The following figure shows the output of the preceding code:

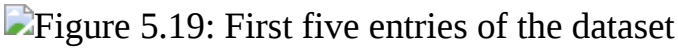
Figure 5.19: First five entries of the dataset

Figure 5.19: First five entries of the dataset

6. Most datasets won't be in the format that you desire. Some of them might have their **Latitude** and **Longitude** values hidden in a different column. This is where the data wrangling skills of *Chapter 1, The Importance of Data Visualization and Data Exploration*, are needed. For the given dataset, the transformations are easy – we simply need to map the **Latitude** and **Longitude** columns into the **lat** and **lon** columns, which are done by an assignment:

   ```
   # mapping Latitude to lat and Longitude to lon

   dataset['lat'] = dataset['Latitude']

   dataset['lon'] = dataset['Longitude']
   ```

7. Our dataset is now ready for the first plotting. We'll use a DotDensityLayer here, which is a good way to start and see all of our data points:

   ```
   # plotting the whole dataset with dots

   geoplotlib.dot(dataset)

   geoplotlib.show()
   ```

   The following figure shows the output of the preceding code:

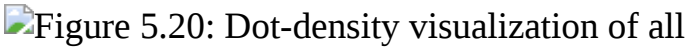
Figure 5.20: Dot-density visualization of all the cities

Figure 5.20: Dot-density visualization of all the cities

8. Before we start breaking down our data to get a better and more workable dataset, we want to understand the outlines of our full data. We'll do this by displaying the number of countries and the number of cities that our dataset holds:

   ```
   # amount of countries and cities
   ```

```
print(len(dataset.groupby(['Country'])), 'Countries')

print(len(dataset), 'Cities')
```

The following figure shows the output of the preceding code:

Figure 5.21: Grouping by countries and cities

9. To see each grouped element on its own, we use the **size()** method, which returns a Series object:

```
# amount of cities per country (first 20 entries)

dataset.groupby(['Country']).size().head(20)
```

The following figure shows the output of the preceding code:

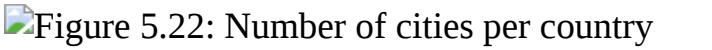

Figure 5.22: Number of cities per country

10. We also want to display the average amount of cities per country. Aggregation is an important concept in pandas that helps us in achieving this:

```
# average num of cities per country

dataset.groupby(['Country']).size().agg('mean')
```

The following figure shows the output of the preceding code:

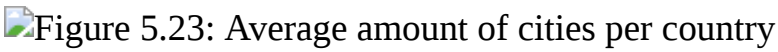

Figure 5.23: Average amount of cities per country

11. We now want to reduce the amount of data we are working with. One approach would be to remove all cities that don't have a population value:

Note

Breaking down and filtering your data is one of the most important aspects of getting good insights. Cluttered visualizations can hide information.

```
# filter for countries with a population entry (Population > 0)

dataset_with_pop = dataset[(dataset['Population'] > 0)]

print('Full dataset:', len(dataset))

print('Cities with population information:', len(dataset_with_pop))
```

The following figure shows the output of the preceding code:

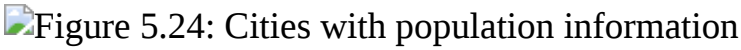
Figure 5.24: Cities with population information

Figure 5.24: Cities with population information

12. Displaying the first five items from the new dataset gives us a basic indication of what values will be present in the **Population** column:

```
# displaying the first 5 items from dataset_with_pop

dataset_with_pop.head()
```

The following figure shows the output of the preceding code:


Figure 5.25: First five items of the reduced dataset

Figure 5.25: First five items of the reduced dataset

13. Now. we can take a look at our reduced dataset with the help of a dot-densityplot:

```
# showing all cities with a defined population with a dot density plot

geoplotlib.dot(dataset_with_pop)

geoplotlib.show()
```

In the new dot plot, we can already feel some improvements in terms of clarity. We still have too many dots on our map. Given the activity definition, we can filter our dataset further by only looking at cities with a population of more than 100k.

14. To further filter our dataset, we can use the same approach that we used previously:

    # dataset with cities with population of >= 100k

    dataset_100k = dataset_with_pop[(dataset_with_pop['Population'] >= 100_000)]

    print('Cities with a population of 100k or more:', len(dataset_100k))

    The following figure shows the output of the preceding code:

    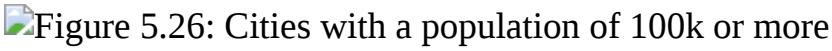 Figure 5.26: Cities with a population of 100k or more

    Figure 5.26: Cities with a population of 100k or more

15. In addition to just plotting our 100k dataset, we also want to fix our viewport to a specific bounding box. Since our data is spread across the world, we can use the in-built **WORLD** constant of the **BoundingBox** class:

    # displaying all cities >= 100k population with a fixed bounding box (WORLD) in a dot density plot

    from geoplotlib.utils import BoundingBox

    geoplotlib.dot(dataset_100k)

    geoplotlib.set_bbox(BoundingBox.WORLD)

    geoplotlib.show()

    The following figure shows the output of the preceding code:

     Figure 5.27: Dot-density visualization of cities with a population of 100k or more

Figure 5.27: Dot-density visualization of cities with a population of 100k or more

16. Comparing it with the previous plots, it gives us a better view on where large amounts of cities are with a population of more than 100k. We now want to find the areas of these cities that are the most densely packed. As we've seen in the previous exercise, **Voronoi** is convenient for such insights:

    ```
    # using filled voronoi to find dense areas

    geoplotlib.voronoi(dataset_100k, cmap='hot_r', max_area=1e3, alpha=255)

    geoplotlib.show()
    ```

    The resulting visualization is exactly what we were searching for. In the Voronoi plot, we can see clear tendencies. Germany, Great Britain, Nigeria, India, Japan, Java, the east coast of the USA, and Brazil stick out. We can now filter our data and only look at those countries to find the ones that are best suited.

    Note

    You can also create custom color map gradients with the **ColorMap** class.

17. The last step is to reduce our dataset to countries in Europe, such as Germany and Great Britain. We can use operators when supplying conditions to filter our data. This operator will allow us to filter for Germany and Great Britain at the same time:

    ```
    # filter 100k dataset for cities in Germany and GB

    dataset_europe = dataset_100k[(dataset_100k['Country'] == 'de') |
    (dataset_100k['Country'] == 'gb')]

    print('Cities in Germany or GB with population >= 100k:',
    len(dataset_europe))
    ```

    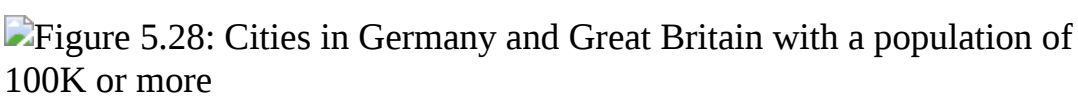
    Figure 5.28: Cities in Germany and Great Britain with a population of 100K or more

Figure 5.28: Cities in Germany and Great Britain with a population of 100K or more

18. Using Delaunay triangulation to find the area with the most densely packed cities, we can see one more hotspot. The areas around Cologne, Birmingham, and Manchester really stick out:

    #using Delaunay triangulation to find the most densely populated area

    geoplotlib.delaunay(dataset_europe, cmap='hot_r')

    geoplotlib.show()

    By using the **hot_r** color map, we can quickly get a good visual representation and make the areas of interest pop out.

Congratulations! You've completed your first activity using Geoplotlib. We made use of different plots to get the information we needed. In the following activities, we will look at some more custom features of Geoplotlib to change the map tiles provider and create custom plotting layers.

Activity 28: Working with Custom Layers

Solution:

Let's create a custom layer **that** will allow us to display geospatial data and animate the data points over time**:**

1. In this first step, we only need **pandas** for the data import:

    # importing the necessary dependencies

    import pandas as pd

2. Use the **read_csv** method of **pandas** to load the **.csv** file:

    # loading the dataset from the csv file

    dataset = pd.read_csv('./data/flight_tracking.csv')

    Note

    The preceding dataset can be found here: https://bit.ly/2DyPHwD.

3. We need to understand the structure of our dataset by looking at the provided features:

   # displaying the first 5 rows of the dataset

   dataset.head()

   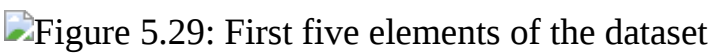
   Figure 5.29: First five elements of the dataset

   Figure 5.29: First five elements of the dataset

4. Remember that Geoplotlib needs **latitude** and **longitude** columns with the names **lat** and **lon**. Use the **rename** method provided in pandas to rename the columns:

   # renaming columns latitude to lat and longitude to lon

   dataset = dataset.rename(index=str, columns={"latitude": "lat", "longitude": "lon"})

5. Taking another look at the first five elements of the dataset, we can now observe that the names of the columns have changed to **lat** and **lon**:

   # displaying the first 5 rows of the dataset

   dataset.head()

   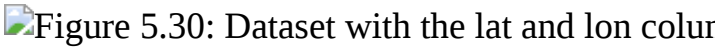
   Figure 5.30: Dataset with the lat and lon columns

   Figure 5.30: Dataset with the lat and lon columns

6. Since we want to get a visualization over time in this activity, we need to work with **date** and **time**. If we take a closer look at our dataset, it shows us that **date** and **time** are separated by two columns.

   We need to combine date and time into a timestamp using the **to_epoch** method. The following code does this:

   # method to convert date and time to an unix timestamp

   from datetime import datetime

```
def to_epoch(date, time):

try:

timestamp = round(datetime.strptime('{} {}'.format(date, time), '%Y/%m/%d %H:%M:%S.%f').timestamp())

return timestamp

except ValueError:

return round(datetime.strptime('2017/09/11 17:02:06.418', '%Y/%m/%d %H:%M:%S.%f').timestamp())
```

7. With the preceding method, we can now use the **apply** method provided by the pandas DataFrame to create a new column called **timestamp** that holds the unix timestamp:

```
# creating a new column called timestamp with the to_epoch method applied

dataset['timestamp'] = dataset.apply(lambda x: to_epoch(x['date'], x['time']), axis=1)
```

8. Taking another look at our dataset, we now have a new column that holds the unix timestamps:

```
# displaying the first 5 rows of the dataset

dataset.head()
```

Figure 5.31: Dataset with a timestamp column added

Figure 5.31: Dataset with a timestamp column added

Since our dataset is now ready to be used with all the necessary columns in place, we can start writing our custom layer. This layer will display each point once it reaches the **timestamp** that's provided in the dataset. It will be displayed for a few seconds before it disappears. We'll need to keep track of the current timestamp in our custom layer. Consolidating what we learned in the theoretical section of this topic, we have a **__init__** method that constructs our custom **TrackLayer**.

9. In our **draw** method, we filter our dataset for all the elements that are in the mentioned time range and use each element of the filtered list to display it on the map with color that's provided by the **colorbrewer** method.

   Since our dataset only contains data from a specific time range and we're always incrementing the time, we want to check whether there are still any elements with **timestamps** after the current timestamp. If not, we want to set our current timestamp to the earliest timestamp that's available in the dataset. The following code shows how we can create a custom layer:

   ```
   # custom layer creation

   import geoplotlib

   from geoplotlib.layers import BaseLayer

   from geoplotlib.core import BatchPainter

   from geoplotlib.colors import colorbrewer

   from geoplotlib.utils import epoch_to_str, BoundingBox

   class TrackLayer(BaseLayer):

   def __init__(self, dataset, bbox=BoundingBox.WORLD):

   self.data = dataset

   self.cmap = colorbrewer(self.data['hex_ident'], alpha=200)

   self.time = self.data['timestamp'].min()

   self.painter = BatchPainter()

   self.view = bbox

   def draw(self, proj, mouse_x, mouse_y, ui_manager):

   self.painter = BatchPainter()

   df = self.data.where((self.data['timestamp'] > self.time) &
   (self.data['timestamp'] <= self.time + 180))
   ```

```
for element in set(df['hex_ident']):

grp = df.where(df['hex_ident'] == element)

self.painter.set_color(self.cmap[element])

x, y = proj.lonlat_to_screen(grp['lon'], grp['lat'])

self.painter.points(x, y, 15, rounded=True)

self.time += 1

if self.time > self.data['timestamp'].max():

self.time = self.data['timestamp'].min()

self.painter.batch_draw()

ui_manager.info('Current timestamp: {}'.format(epoch_to_str(self.time)))

# bounding box that gets used when layer is created

def bbox(self):

return self.view
```

10. Since our dataset only contains data from the area around Leeds in the UK, we need to define a custom **BoundingBox** that focuses our view on this area:

    ```
    # bounding box for our view on Leeds

    from geoplotlib.utils import BoundingBox

    leeds_bbox = BoundingBox(north=53.8074, west=-3, south=53.7074 , east=0)
    ```

11. Geoplotlib sometimes requires you to provide a **DataAccessObject** instead of a pandas DataFrame. Geoplotlib provides a convenient method for converting any pandas DataFrame into a **DataAccessObject**:

    ```
    # displaying our custom layer using add_layer
    ```

```
from geoplotlib.utils import DataAccessObject

data = DataAccessObject(dataset)

geoplotlib.add_layer(TrackLayer(data, bbox=leeds_bbox))

geoplotlib.show()
```

Congratulations! You've completed the custom layer activity using Geoplotlib. We've applied several pre-processing steps to shape the dataset as we want to have it. We've also written a custom layer to display spatial data in the temporal space. Our custom layer even has a level of animation. This is something we'll look into more in the following chapter about Bokeh.

Chapter 6: Making Things Interactive with Bokeh

Activity 29: Extending Plots with Widgets

Solution:

1. Open the **activity29_solution.ipynb** Jupyter Notebook from the **Lesson06** folder to implement this activity.
2. Import the necessary libraries:

   ```
   # importing the necessary dependencies

   import pandas as pd
   ```

3. Again, we want to display our plots inside a Jupyter Notebook, so we have to import and call the **output_notebook** method from the **io** interface of Bokeh:

   ```
   # make bokeh display figures inside the notebook

   from bokeh.io import output_notebook

   output_notebook()
   ```

4. After downloading the dataset and moving it into the data folder of this chapter, we can import our **olympia2016_athletes.csv** data:

```
# loading the Dataset with geoplotlib

dataset = pd.read_csv('./data/olympia2016_athletes.csv')
```

5. A quick test by calling **head** on our DataFrame shows us that our data has been successfully loaded:

```
# looking at the dataset

dataset.head()
```

The following figure shows the output of the preceding code:

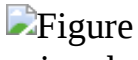
Figure 6.25: Loading the top five rows of the olympia2016_athletes dataset using the head method

Figure 6.25: Loading the top five rows of the olympia2016_athletes dataset using the head method

Building an Interactive Visualization

1. To create our visualization, we need some additional imports. We'll again import **figure** and show from the plotting interface, which will give us the tools we need to create our plot. The widgets, as we saw in the previous exercise, come from the **ipywidgets** library. Here, we'll be using **interact**, again, as **decorator**, and the **widgets** interface, which gives us access to different widgets:

```
# importing the necessary dependencies

from bokeh.plotting import figure, show

from ipywidgets import interact, widgets
```

2. Like in the previous exercises, we need to do some data extraction first. In this activity, we will need a list of unique countries from the dataset, the amount of athletes for each country, and the amount of medals won by each country, split into gold, silver, and bronze:

```
# extract countries and group Olympians by country and their sex

# and the number of medals per country by sex
```

```
countries = dataset['nationality'].unique()

athletes_per_country = dataset.groupby('nationality').size()

medals_per_country = dataset.groupby('nationality')['gold',
'silver','bronze'].sum()
```

3. Before we go in and implement the plotting for this visualization, we want to set up our widgets and the **@interact** method that will later display the plot upon execution. Execute this empty **get_plot()** method cell and then move on to the widget creation. We will implement this later.
4. In this activity, we will use two **IntSlider** widgets that will control the max numbers for the amount of athletes and/or medals a country is allowed to have in order to be displayed in the visualization. We need two values to set up the widgets: the maximum amount of medals of all the countries and the maximum amount of athletes of all the countries:

```
# getting the max amount of medals and athletes of all countries

max_medals = medals_per_country.sum(axis=1).max()

max_athletes = athletes_per_country.max()
```

5. Using those maximum numbers as the maximum for both widgets will give us reasonable slider values that are dynamically adjusted if we should increase the amount of athletes or medals in the dataset. We need two **IntSlider** objects that handle the input for our **max_athletes** and **max_medals**. To look like our actual visualization, we want to have the **max_athletes_slider** displayed in a vertical orientation and the **max_medals_slider** in a horizontal orientation.In the visualization, they should be displayed as **Max. Athletes** and **Max. Medals**:

```
# setting up the interaction elements

max_athletes_slider=widgets.IntSlider(value=max_athletes, min=0,
max=max_athletes, step=1, description='Max. Athletes:',
continuous_update=False, orientation='vertical', layout={'width': '100px'})

max_medals_slider=widgets.IntSlider(value=max_medals, min=0,
max=max_medals, step=1, description='Max. Medals:',
continuous_update=False, orientation='horizontal')
```

6. After setting up the widgets, we can implement the method that will be called with each update of the interaction widgets. As we saw in the previous exercise, we will use the **@interact** decorator for this. Instead of value ranges or lists, we will provide the variable names of our already created widgets in the decorator. Since we have already set up the empty method that will return a plot, we can call **show()** with the method call inside it to show the result once it is returned from the **get_plot** method:

   ```
   # creating the interact method

   @interact(max_athletes=max_athletes_slider,
   max_medals=max_medals_slider)

   def get_olympia_stats(max_athletes, max_medals):

   show(get_plot(max_athletes, max_medals))
   ```

7. As we mentioned in the previous exercise, we can also make use of the widgets that are described here: https://ipywidgets.readthedocs.io/en/stable/examples/Widget%20List.html.
8. Once you've built the widgets, upon execution, you will see them being displayed below the cell. We are now ready to scroll up and implement the plotting with Bokeh.
9. The two arguments we get passed are **max_athletes** and **max_medals**. Both of them are **int** values. First, we want to filter down our countries dataset that contains all the countries that placed athletes in the Olympic Games. We need to check whether they have less than or equal medals and athletes than our max values, passed as arguments. Once we have a filtered down dataset, we can create our DataSource. This DataSource will be used both for the tooltips and the printing of the circle glyphs.

   Note

   There is extensive documentation on how to use and set up tooltips that you can and use, which can be accessed from the following link: https://bokeh.pydata.org/en/latest/docs/user_guide/tools.html.

10. Create a new plot using the **figure** method that has the following attributes: title of '**Rio Olympics 2016 - Medal comparison**', **x_axis_label** of '**Number of Medals**', and **y_axis_label** of '**Num of Athletes**':

```
# creating the scatter plot

def get_plot(max_athletes, max_medals):

filtered_countries=[]

for country in countries:

if (athletes_per_country[country] <= max_athletes and

medals_per_country.loc[country].sum() <= max_medals):

filtered_countries.append(country)

data_source=get_datasource(filtered_countries)

TOOLTIPS=[ ('Country', '@countries'),('Num of Athletes', '@y'),('Gold',
'@gold'),('Silver', '@silver'),('Bronze', '@bronze')]

plot=figure(title='Rio Olympics 2016 - Medal comparison',
x_axis_label='Number of Medals', y_axis_label='Num of Athletes',
plot_width=800, plot_height=500, tooltips=TOOLTIPS)

plot.circle('x', 'y', source=data_source, size=20, color='color', alpha=0.5)

return plot
```

11. To display every country with a different color, we want to randomly create the colors with a six-digit hex code. The following method does this:

```
# get a 6 digit random hex color to differentiate the countries better

import random

def get_random_color():

return '%06x' % random.randint(0, 0xFFFFFF)
```

12. We will use a Bokeh **ColumnDataSource** to handle our data and make it easily accessible for our tooltip and glyphs. Since we want to display additional information in a tooltip, we need our DataSource to have the **color** field, which holds the required amount of random colors; the

countries field, which holds the list of filtered down countries; the **gold**, **silver**, and **bronze** fields, which hold the number of **gold**, **silver**, and **bronze** medals for each country, respectively; the **x** field, which holds the summed number of medals for each **country**; and the **y** field, which holds the number of athletes for each **country**:

```
# build the DataSource

def get_datasource(filtered_countries):

return ColumnDataSource(data=dict(

color=[get_random_color() for _ in filtered_countries],

countries=filtered_countries,

gold=[medals_per_country.loc[country]['gold'] for country in
filtered_countries],

silver=[medals_per_country.loc[country]['silver'] for country in
filtered_countries],

bronze=[medals_per_country.loc[country]['bronze'] for country in

x=[medals_per_country.loc[country].sum() for country in
filtered_countries],

y=[athletes_per_country.loc[country].sum() for country in
filtered_countries]

))
```

13. After our implementation is finished, we can execute the last cell with our **@interact** decorator once more. This time, it will display our scatter plot with our interactive widgets. We will see each country in a different color. Upon hovering over them, we will get more information about each country, such as its short name, number of athletes, and the amount of gold, silver, and bronze medals they earned. The resulting visualization should look as follows:

Figure 6.26: Final interactive visualization that displays the scatter plot

You've built a full visualization to display and explore data from the 2016 Olympics. We added two widgets to our visualization that allows us to filter down our displayed countries. As we mentioned before, when working with interactive features and Bokeh, you might want to read up about Bokeh Server a little bit more. It will give you more options to express your creativity by creating animated plots and visualizations that can be explored by several people at the same time.

Chapter 7: Combining What We Have Learned

Activity 30: Implementing Matplotlib and Seaborn on New York City Database

Solution:

1. Open the Jupyter Notebook **activity30_solution.ipynb** from the **Lesson07** folder to implement this activity. Import all the necessary libraries:

   ```
   # Import statements

   import pandas as pd

   import numpy as np

   import seaborn as sns

   import matplotlib

   import matplotlib.pyplot as plt

   import squarify

   sns.set()
   ```

2. Use pandas to read both **.csv** files located in the **data** subdirectory:

   ```
   p_ny = pd.read_csv('./data/pny.csv')
   ```

h_ny = pd.read_csv('./data/hny.csv')

3. Use the given PUMA (public use microdata area code based on the 2010 Census definition, which are areas with populations of 100,000 or more) ranges to further divide the dataset into NYC districts (Bronx, Manhatten, Staten Island, Brooklyn, and Queens):

   # PUMA ranges

   bronx = [3701, 3710]

   manhatten = [3801, 3810]

   staten_island = [3901, 3903]

   brooklyn = [4001, 4017]

   queens = [4101, 4114]

   nyc = [bronx[0], queens[1]]

   def puma_filter(data, puma_ranges):

   return data.loc[(data['PUMA'] >= puma_ranges[0]) & (data['PUMA'] <= puma_ranges[1])]

   h_bronx = puma_filter(h_ny, bronx)

   h_manhatten = puma_filter(h_ny, manhatten)

   h_staten_island = puma_filter(h_ny, staten_island)

   h_brooklyn = puma_filter(h_ny, brooklyn)

   h_queens = puma_filter(h_ny, queens)

   p_nyc = puma_filter(p_ny, nyc)

   h_nyc = puma_filter(h_ny, nyc)

4. In the dataset, each sample has a certain **weight** that reflects the **weight** for the total dataset. Therefore, we cannot simply calculate the median. Use the

given **weighted_median** function in the following code to compute the median:

```
# Function for a 'weighted' median

def weighted_frequency(values, weights):

  weighted_values = []

  for value, weight in zip(values, weights):

    weighted_values.extend(np.repeat(value, weight))

  return weighted_values

def weighted_median(values, weights):

  return np.median(weighted_frequency(values, weights))
```

5. In this subtask, we will create a plot containing multiple subplots that visualize information with regards to NYC wages. Visualize the median household income for the US, New York, New York City, and its districts. Visualize the average wage by gender for the given occupation categories for the population of NYC. Visualize the wage distribution for New York and NYC. Use the following yearly wage intervals: 10k steps between 0 and 100k, 50k steps between 100k and 200k, and >200k:

Lesson07/Activity30/activity30_solution.ipynb

```
# Data wrangling for median housing income

income_adjustement = h_ny.loc[0, ['ADJINC']].values[0] / 1e6

def median_housing_income(data):

//[…]

h_queens_income_median = median_housing_income(h_queens)

# Data wrangling for wage by gender for different occupation categories
```

occ_categories = ['Management,\nBusiness,\nScience,\nand Arts\nOccupations', 'Service\nOccupations',

'Sales and\nOffice\nOccupations', 'Natural Resources,\nConstruction,\nand Maintenance\nOccupations',

'Production,\nTransportation,\nand Material Moving\nOccupations']

//[…]

wages_female = wage_by_gender_and_occupation(p_nyc, 2)

Data wrangling for wage distribution

wage_bins = {'<$10k': [0, 10000], '$10-20k': [10000, 20000], '$20-30k': [20000, 30000], '$30-40k': [30000, 40000], '$10-20k': [40000, 50000],

'$50-60k': [50000, 60000], '$60-70k': [60000, 70000], '$70-80k': [70000, 80000], '$80-90k': [80000, 90000], '$90-100k': [90000, 100000],

'$100-150k': [100000, 150000], '$150-200k': [150000, 200000], '>$200k': [200000, np.infty]}

//[…]

wages_ny = wage_frequency(p_ny)

Create figure with four subplots

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(7, 10), dpi=300)

Median household income in the US

us_income_median = 60336

Median household income

ax1.set_title('Median Household Income', fontsize=14)

//[…]

ax1.set_xlabel('Yearly household income in $')

```
# Wage by gender in common jobs

ax2.set_title('Wage by Gender for different Job Categories', fontsize=14)

x = np.arange(5) + 1

//[…]

ax2.set_ylabel('Average Salary in $')

# Wage distribution

ax3.set_title('Wage Distribution', fontsize=14)

x = np.arange(len(wages_nyc)) + 1

width = 0.4

//[…]

ax3.vlines(x=9.5, ymin=0, ymax=15, linestyle='--')

# Overall figure

fig.tight_layout()

plt.show()
```

https://bit.ly/2StchfL

The following figure shows the output of the preceding code:


Figure 7.15: Wage statistics for New York City in comparison with New York and the United States

Figure 7.15: Wage statistics for New York City in comparison with New York and the United States

6. Use a tree map to visualize the percentage for the given occupation subcategories for the population of NYC:

Lesson07/Activity30/activity30_solution.ipynb

```
# Data wrangling for occupations

occ_subcategories = {'Management,\nBusiness,\nand Financial': [10, 950],

//[..]

def occupation_percentage(data):

percentages = []

overall_sum = np.sum(data.loc[(data['OCCP'] >= 10) & (data['OCCP'] <=
9750), ['PWGTP']].values)

for occ in occ_subcategories.values():

query = data.loc[(data['OCCP'] >= occ[0]) & (data['OCCP'] <= occ[1]),
['PWGTP']].values

percentages.append(np.sum(query) / overall_sum)

return percentages

occ_percentages = occupation_percentage(p_nyc)

# Visualization of tree map

plt.figure(figsize=(16, 6), dpi=300)

//[..]

plt.axis('off')

plt.title('Occupations in New York City', fontsize=24)

plt.show()
```

https://bit.ly/2StchfL

The following figure shows the output of the preceding code:

 Figure 7.16: Occupations in NYC

Figure 7.16: Occupations in NYC

7. Use a heatmap to show the correlation between difficulties (self-care difficulty, hearing difficulty, vision, difficulty, independent living difficulty, ambulatory difficulty, veteran service-connected disability, and cognitive difficulty) and age groups (<5, 5-11, 12-14, 15-17, 18-24, 25-34, 35-44, 45-54, 55-64, 65-74, and 75+) in New York City:

```
# Data wrangling for New York City population difficulties

difficulties = {'Self-care difficulty': 'DDRS', 'Hearing difficulty': 'DEAR',

'Vision difficulty': 'DEYE', 'Independent living difficulty': 'DOUT',

'Ambulatory difficulty': 'DPHY', 'Veteran service connected disability': 'DRATX',

'Cognitive difficulty': 'DREM'}

age_groups = {'<5': [0, 4], '5-11': [5, 11], '12-14': [12, 14], '15-17': [15, 17], '18-24': [18, 24], '25-34': [25, 34],

'35-44': [35, 44], '45-54': [45, 54], '55-64': [55, 64], '65-74': [65, 74], '75+': [75, np.infty]}

def difficulty_age_array(data):

array = np.zeros((len(difficulties.values()), len(age_groups.values())))

for d, diff in enumerate(difficulties.values()):

for a, age in enumerate(age_groups.values()):

age_sum = np.sum(data.loc[(data['AGEP'] >= age[0]) & (data['AGEP'] <= age[1]), ['PWGTP']].values)

query = data.loc[(data['AGEP'] >= age[0]) & (data['AGEP'] <= age[1]) & (data[diff] == 1), ['PWGTP']].values

array[d, a] = np.sum(query) / age_sum

return array
```

```
array = difficulty_age_array(p_nyc)

# Heatmap

plt.figure(dpi=300)

ax = sns.heatmap(array * 100)

ax.set_yticklabels(difficulties.keys(), rotation=0)

ax.set_xticklabels(age_groups.keys(), rotation=90)

ax.set_xlabel('Age Groups')

ax.set_title('Percentage of NYC population with difficulties', fontsize=14)

plt.show()
```

The following figure shows the output of the preceding code:

Figure 7.17: Percentage of NYC population with difficulties

Activity 31: Bokeh Stock Prices Visualization

Solution:

1. Open the **activity31_solution.ipynb** Jupyter Notebook from the **Lesson07** folder to implement this activity. Import the pandas library:

   ```
   # importing the necessary dependencies

   import pandas as pd
   ```

2. Display our plots inside a Jupyter Notebook. We have to import and call the **output_notebook** method from the **io** interface of Bokeh:

   ```
   # make bokeh display figures inside the notebook

   from bokeh.io import output_notebook
   ```

```
output_notebook()
```

3. After downloading the dataset and moving it into the data folder of this chapter, we can import our **stock_prices.csv** data:

```
# loading the Dataset with geoplotlib

dataset = pd.read_csv('./data/stock_prices.csv')
```

4. A quick test by calling head on our DataFrame shows us that our data has been successfully loaded:

```
# looking at the dataset

dataset.head()
```

The following figure shows the output of the preceding code:

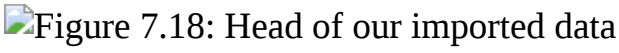
Figure 7.18: Head of our imported data

Figure 7.18: Head of our imported data

5. Since the **date** column has no information about the hour, minute, and second, we want to avoid displaying them in the visualization later on and simply display the year, month, and day. Therefore, we'll create a new column that holds the formatted short version of the date value. The execution of the cell will take a moment since it's a fairly large dataset. Please be patient:

```
# mapping the date of each row to only the year-month-day format

from datetime import datetime

def shorten_time_stamp(timestamp):

  shortened = timestamp[0]

  if len(shortened) > 10:

    parsed_date=datetime.strptime(shortened, '%Y-%m-%d %H:%M:%S')
```

```
    shortened=datetime.strftime(parsed_date, '%Y-%m-%d')
  return shortened
dataset['short_date'] = dataset.apply(lambda x: shorten_time_stamp(x), axis=1)
```

6. Taking another look at our updated dataset, we can see a new column called **short_date** that holds the date without the hour, minute, and second information:

```
# looking at the dataset with shortened date
dataset.head()
```

The following figure shows the output of the preceding code:

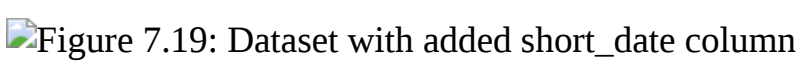

Figure 7.19: Dataset with added short_date column

Building an Interactive Visualization

1. To create our visualization, we need some additional imports. We'll again import **figure** and show this from the plotting interface, which will give us the tools we need to create our plot. The widgets, as we saw in the previous exercise, come from the **ipywidgets** library. Here, we'll be using **@interact**, again, as a decorator, and the **widgets** interface, which gives us access to the different widgets:

```
# importing the necessary dependencies
from bokeh.plotting import figure, show
from ipywidgets import interact, widgets
```

2. Before we go in and implement the plotting methods, we want to set up the interactivity widgets. Scroll down to the cell that says **#extracing the necessary data** before implementing the plotting. Still make sure that you execute the cells below that, even though this will simply pass and do nothing for now. We want to start implementing our visualization here. In

the following cells, we will extract the necessary data that will be provided to the widget elements. In the first cell, we want to extract the following information: a list of unique stock names that are present in the dataset, a list of all **short_dates** that are in 2016, a sorted list of unique dates generated from the previous list of dates from 2016, and a list with the values **open-close** and **volume**.

3. Once we have this information in place, we can start building our widgets:

```
# extracing the necessary data

stock_names=dataset['symbol'].unique()

dates_2016=dataset[dataset['short_date'] >= '2016-01-01']['short_date']

unique_dates_2016=sorted(dates_2016.unique())

value_options=['open-close', 'volume']
```

4. Given the extracted information from the preceding cell, we can now define **widgets** and provide the available options for it. As we mentioned in the introduction, we want to have several interactive features, including two dropdowns with which we can select two stocks that should be compared to each other. The first dropdown by default should have the AAPL stock selected, named **Compare:**, while the second dropdown by default should have the AON stock selected, named **to:**:

```
# setting up the interaction elements

drp_1=widgets.Dropdown(options=stock_names,

value='AAPL',

description='Compare:')

drp_2=widgets.Dropdown(options=stock_names,

value='AON', description='to:')
```

5. Then, we need **SelectionRange**, which will allow us to select a range of dates from the extracted list of unique 2016 dates. By default, the first 25 dates should be selected, named **From-To**. Make sure to disable the

continuous_update parameter. Adjust the layout width to **500px** to make sure that the dates are displayed correctly:

```
range_slider=widgets.SelectionRangeSlider(options=unique_dates_2016,

index=(0,25),

continuous_update=False,

description='From-To',

layout={'width': '500px'})
```

6. Add a **RadioButton** group that provides the **open-close** and **volume** options. By default, **open-close** should be selected, named **Metric**:

```
range_slider=widgets.SelectionRangeSlider(options=unique_dates_2016,

index=(0,25),

continuous_update=False,

description='From-To',

layout={'width': '500px'})

value_radio=widgets.RadioButtons(options=value_options,

value='open-close',

description='Metric')
```

Note

As we mentioned in the previous exercise, we can also make use of the widgets that are described here: https://bit.ly/2Te9jAf.

7. After setting up the widgets, we can implement the method that will be called with each update of the interaction widgets. As we saw in the previous exercise, we will use the **@interact** decorator for this.
8. Instead of value ranges or lists, we will provide the variable names of our already created widgets in the decorator. The method will get four

arguments: **stock_1**, **stock_2**, **date**, and **value**. Since we have already set up the empty method that will return the preceding plot, we can call **show()** with the method call inside to show the result once it is returned from the **get_stock_for_2016** method.

9. Once you've built the widgets, upon execution, you will see them being displayed below the cell:

```
# creating the interact method

@interact(stock_1=drp_1, stock_2=drp_2, date=range_slider,
value=value_radio)

def get_stock_for_2016(stock_1, stock_2, date, value):

show(get_plot(stock_1, stock_2, date, value))
```

10. We are now ready to scroll up and implement the plotting with Bokeh. At the moment, our **show()** in the last cell will not render any elements into our visualization. We will start with the so-called candle stick visualization, which is often used with stock price data.
11. The already defined method gets our **plot** object, a **stock_name**, a **stock_range** containing the data of only the selected date range that was defined with the widgets, and a color for the line. We will use those arguments to create the candle sticks. They basically contain a segment that creates the vertical line, and either a green or red vbar to color code whether the close price is lower than the open price. Once you have created the candles, we also want to have a continuous line running through the mean (**high**, **low**) point of each candle. So, you have to calculate the mean for every (high/low) pair and then plot those data points with a line with the given color:

```
def add_candle_plot(plot, stock_name, stock_range, color):

inc_1 = stock_range.close > stock_range.open

dec_1 = stock_range.open > stock_range.close

w = 0.5

plot.segment(stock_range['short_date'], stock_range['high'],
```

```
stock_range['short_date'], stock_range['low'],

color="grey")

plot.vbar(stock_range['short_date'][inc_1], w,

stock_range['high'][inc_1], stock_range['close'][inc_1],

fill_color="green", line_color="black",

legend=('Mean price of ' + stock_name), muted_alpha=0.2)

plot.vbar(stock_range['short_date'][dec_1], w,

stock_range['high'][dec_1], stock_range['close'][dec_1],

fill_color="red", line_color="black",

legend=('Mean price of ' + stock_name), muted_alpha=0.2)

stock_mean_val=stock_range[['high', 'low']].mean(axis=1)

plot.line(stock_range['short_date'], stock_mean_val,

legend=('Mean price of ' + stock_name), muted_alpha=0.2,

line_color=color, alpha=0.5)
```

Note

Make sure to reference the example provided in the Bokeh library here. You can adapt the code in there to our arguments: https://bokeh.pydata.org/en/latest/docs/gallery/candlestick.html.

12. After you are done implementing the **add_candle_plot** method, scroll down and run the **@interact** cell again. You will now see the candles being displayed for the two selected stocks. The last missing step is implementing the plotting of the lines if the **volume** value is selected.
13. One additional interaction feature is to have an interactive legend that allows us to **mute**, meaning gray out, each stock in the visualization:

Lesson07/Activity31/activity31_solution.ipynb

```
# method to build the plot

def get_plot(stock_1, stock_2, date, value):

//[..]

  plot.xaxis.major_label_orientation = 1

  plot.grid.grid_line_alpha=0.3

  if value == 'open-close':

    add_candle_plot(plot, stock_1_name, stock_1_range, 'blue')

    add_candle_plot(plot, stock_2_name, stock_2_range, 'orange')

  if value == 'volume':

  plot.line(stock_1_range['short_date'], stock_1_range['volume'],

    legend=stock_1_name, muted_alpha=0.2)

  plot.line(stock_2_range['short_date'], stock_2_range['volume'],

    legend=stock_2_name, muted_alpha=0.2,

    line_color='orange')

  plot.legend.click_policy="mute"

  return plot
```

https://bit.ly/2GRneWR

Note

To make our legend interactive please take a look at the documentation for the legend feature: https://bokeh.pydata.org/en/latest/docs/user_guide/interaction/legends.html.

After our implementation has finished, we can execute the last cell with our **@interact** decorator once more. This time, it will display our candlestick plot and once we switch to the volume RadioButton, we will see the volumes displayed that have been traded at the given dates. The resulting visualization should look somewhat like this:

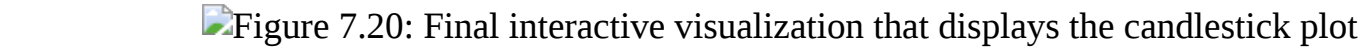
Figure 7.20: Final interactive visualization that displays the candlestick plot

Figure 7.20: Final interactive visualization that displays the candlestick plot

The following figure shows the final interactive visualization of volume plot:

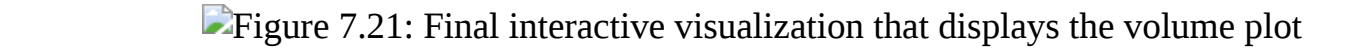
Figure 7.21: Final interactive visualization that displays the volume plot

Figure 7.21: Final interactive visualization that displays the volume plot

Congratulations!

You've built a full visualization to display and explore stock price data. We added several widgets to our visualization that allows us to select to be compared stocks, restrict the displayed data to a specific date range, and even display two different kind of plots.

As we mentioned before, when working with interactive features and Bokeh, you might want to read up about Bokeh Server a little bit more. It will give you more options to express your creativity by creating animated plots and visualizations that can be explored by several people at the same time.

Activity 32: Analyzing Airbnb Data with Geoplotlib

Solution:

1. Open the **activity032_solution.ipynb** Jupyter Notebook from the **Lesson07** folder to implement this activity. Import NumPy, pandas, and Geoplotlib first:

   ```
   # importing the necessary dependencies

   import numpy as np
   ```

```
import pandas as pd

import geoplotlib
```

2. Use the **read_csv** method of pandas to load the **.csv** file. If your computer is a little slow, use the smaller dataset:

```
# loading the Dataset

dataset = pd.read_csv('./data/airbnb_new_york.csv')

# dataset = pd.read_csv('./data/airbnb_new_york_smaller.csv')
```

3. Understand the structure of our dataset by looking at the provided features:

```
# print the first 5 rows of the dataset

dataset.head()
```

The following figure shows the output of the preceding code:

	id	listing_url	scrape_id	last_scraped	name	summary	space	description	experiences_offered	neighborhood_overview	...	requires_license	license	jurisdiction_names	instant_bookable	is_business_travel_ready
0	2515	https://www.airbnb.com/rooms/2515	20181206022948	2018-12-06	Stay at Chez Chic budget room #1	Step into our artistic spacious apartment and ...	-PLEASE BOOK DIRECTLY. NO NEED TO SEND A REQUE...	Step into our artistic spacious apartment and ...	none	NaN	...	f	NaN	NaN	f	f
1	21456	https://www.airbnb.com/rooms/21456	20181206022948	2018-12-06	Light-filled classic Central Park	An adorable, classic, clean, light-filled one-...	An adorable, classic, clean, light-filled one-...	An adorable, classic, clean, light-filled one-...	none	Diverse. Great coffee shops and restaurants, n...	...	f	NaN	NaN	f	f
2	2539	https://www.airbnb.com/rooms/2539	20181206022948	2018-12-06	Clean & quiet apt home by the park	Renovated apt home in elevator building.	Spacious, renovated, and clean apt home, one b...	Renovated apt home in elevator building. Spaci...	none	Close to Prospect Park and Historic Ditmas Park	...	f	NaN	NaN	f	f
3	2595	https://www.airbnb.com/rooms/2595	20181206022948	2018-12-06	Skylit Midtown Castle	Find your romantic getaway to this beautiful, ...	- Spacious (500+ft²), immaculate and nicely fu...	Find your romantic getaway to this beautiful, ...	none	Centrally located in the heart of Manhattan ju...	...	f	NaN	NaN	f	f
4	21644	https://www.airbnb.com/rooms/21644	20181206022948	2018-12-06	Upper Manhattan, New York	A great space in a beautiful neighborhood-min...	Nice room in a spacious pre-war apartment in u...	A great space in a beautiful neighborhood-min...	none	I love that the neighborhood is safe to walk a...	...	f	NaN	NaN	f	f

Figure 7.22: Displaying the first five elements of the dataset

4. Remember that Geoplotlib needs **latitude** and **longitude** columns with the names **lat** and **lon**. We will therefore add new columns for **lat** and **lon** and assign the corresponding value columns to it:

```
# mapping Latitude to lat and Longitude to lon

dataset['lat'] = dataset['latitude']

dataset['lon'] = dataset['longitude']
```

5. When creating a color map that changes color based on the price of an accommodation, we need a value that can easily be compared and checked whether it's smaller or bigger than any other listing. Therefore, we will create a new column called **dollar_price** that will hold the value of the price column as a float:

   ```
   # convert string of type $<numbers> to <nubmers> of type float

   def convert_to_float(x):

   try:

   value=str.replace(x[1:], ',', '')

   return float(value)

   except:

   return 0.0

   # create new dollar_price column with the price as a number

   # and replace the NaN values by 0 in the ratings column

   dataset['price'] = dataset['price'].fillna('$0.0')

   dataset['review_scores_rating'] = dataset['review_scores_rating'].fillna(0.0)

   dataset['dollar_price'] = dataset['price'].apply(lambda x: convert_to_float(x))
   ```

6. This dataset has 96 columns. When working with such a huge dataset, it makes sense to think about what data we really need and create a subsection of our dataset that only holds the data we need. Before we can do that, we'll take a look at all the columns that are available and an example for that column. This will help us decide what information is suitable:

   ```
   # print the col name and the first entry per column

   for col in dataset.columns:

   print('{}\t{}'.format(col, dataset[col][0]))
   ```

The following figure shows the output of the preceding code:

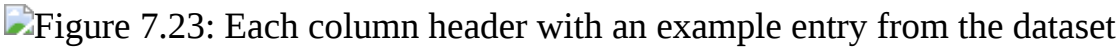
Figure 7.23: Each column header with an example entry from the dataset

Figure 7.23: Each column header with an example entry from the dataset

7. For now, we want to only use the fields that help us build the described visualization. Those fields are **id**, **latitude** (as **lat**), **longitude** (as **lon**), **price** (in **$**), and **review_scores_rating**:

   ```
   # create a subsection of the dataset with the above mentioned columns

   columns=['id', 'lat', 'lon', 'dollar_price', 'review_scores_rating']

   sub_data=dataset[columns]
   ```

8. Taking another look at our dataset, we now have a new column that holds the unix timestamps:

   ```
   # print the first 5 rows of the dataset

   sub_data.head()
   ```

   The following figure shows the output of the preceding code:

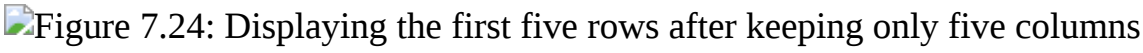
Figure 7.24: Displaying the first five rows after keeping only five columns

Figure 7.24: Displaying the first five rows after keeping only five columns

9. Even though we know that our data holds Airbnb listings for New York City, at the moment, we have no feeling about the amount, distribution, and character of our dataset. The simplest way to get a first glance at the data is to plot every listing with a simple dot map:

   ```
   # import DataAccessObject and create a data object as an instance of that class

   from geoplotlib.utils import DataAccessObject

   data = DataAccessObject(sub_data)
   ```

```
# plotting the whole dataset with dots

geoplotlib.dot(data)

geoplotlib.show()
```

The following figure shows the output of the preceding code:

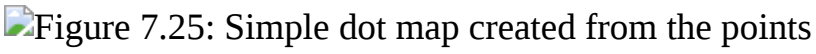
Figure 7.25: Simple dot map created from the points

Figure 7.25: Simple dot map created from the points

10. The last step is to write the custom layer. Here, we want to define a **ValueLayer** that extends the **BaseLayer** of Geoplotlib. For the mentioned interactive feature, we need an additional import. **pyglet** provides us with the option to act on key presses. Given the data, we want to plot each point on the map with a color that is defined by the currently selected attribute, either **price** or **rating**.
11. To avoid non-descriptive output, we need to also adjust the scale of our color map. Ratings are between 0 and 100, whereas prices can be much higher. Using a linear (**lin**) scale for the ratings and a logarithmic (**log**) scale for the price will give us much better insights into our data.
12. The view (bounding box) of our visualization will be set to New York and text information with the currently selected attribute will be displayed in the upper right corner:


Figure 7.26: jet color map scale

Figure 7.26: jet color map scale

13. To assign each point a different color, we simply paint each point separately. This is definitely not the most efficient solution, but it will do for now. We will need the following instance variables: **self.data**, which holds the dataset, **self.display,** which holds the currently selected attribute name, **self.painter,** which holds an instance of the **BatchPainter** class, **self.view,** which holds the **BoundingBox**, **self.cmap,** which holds a color map with the **jet** color schema, and an alpha of 255 and 100 levels.
14. Inside the **invalidate** method, which holds the logic of projection of the data to points on the map, we have to switch between the **lin** and **log** scales,

depending on the attribute that is currently selected. The color is then determined by placing the value between 0/1 and the maximum (**max_val**) value, which also has to be taken from the dataset based on what attribute is currently being displayed:

Lesson07/Activity32/activity32_solution.ipynb

```
# custom layer creation

import pyglet

import geoplotlib

//[..]

class ValueLayer(BaseLayer):

def __init__(self, dataset, bbox=BoundingBox.WORLD):

//[..]

def invalidate(self, proj):

# paint every point with a color that represents the currently selected attributes value

self.painter = BatchPainter()

max_val = max(self.data[self.display])

scale = 'log' if self.display == 'dollar_price' else 'lin'

for index, id in enumerate(self.data['id']):

//[..]

def draw(self, proj, mouse_x, mouse_y, ui_manager):

# display the ui manager info

ui_manager.info('Use left and right to switch between the displaying of price and ratings. Currently displaying: {}'.format(self.display))
```

```
self.painter.batch_draw()

def on_key_release(self, key, modifiers):

//[..]

# bounding box that gets used when layer is created

def bbox(self):

return self.view
```

https://bit.ly/2VoQveT

15. Since our dataset only contains data from New York, we want to set the view to New York in the beginning. Therefore, we need an instance of the **BoundingBox** class with the given parameters. In addition to a custom **BoundingBox**, we will use the **darkmatter** tile provider that we looked at in *Chapter 5, Plotting Geospatial Data*:

```
# bounding box for our view on New York

from geoplotlib.utils import BoundingBox

ny_bbox = BoundingBox(north=40.897994, west=-73.999040,
south=40.595581, east=-73.95040)

# displaying our custom layer using add_layer

geoplotlib.tiles_provider('darkmatter')

geoplotlib.add_layer(ValueLayer(data, bbox=ny_bbox))

geoplotlib.show()
```

16. After launching our visualization, we can see that our viewport is focused on New York. Every accommodation is displayed with one dot that is colored based on either its price, or upon clicking the right or left arrow, the rating. We can see that the general color gets closer to yellow/orange the closer we get to central Manhattan. On the other hand, in the ratings visualization, we can see that the accommodations in central Manhattan seem to be rated lower than the ones outside:

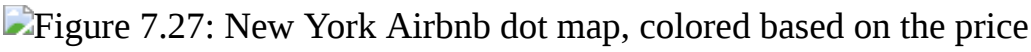

Figure 7.27: New York Airbnb dot map, colored based on the price

The following figure shows a dot map with color based on rating:

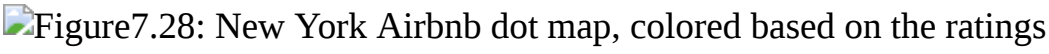

Figure7.28: New York Airbnb dot map, colored based on the ratings

Congratulations!

You've created an interactive visualization by writing your own custom layer to display and visualize price and rating information for Airbnb accommodations spread across New York. As we can now see, writing custom layers for Geoplotlib is a good approach for focusing on the attributes that you are interested in.

Thanks for picking up this course to enhance your data visualization skills with Python.