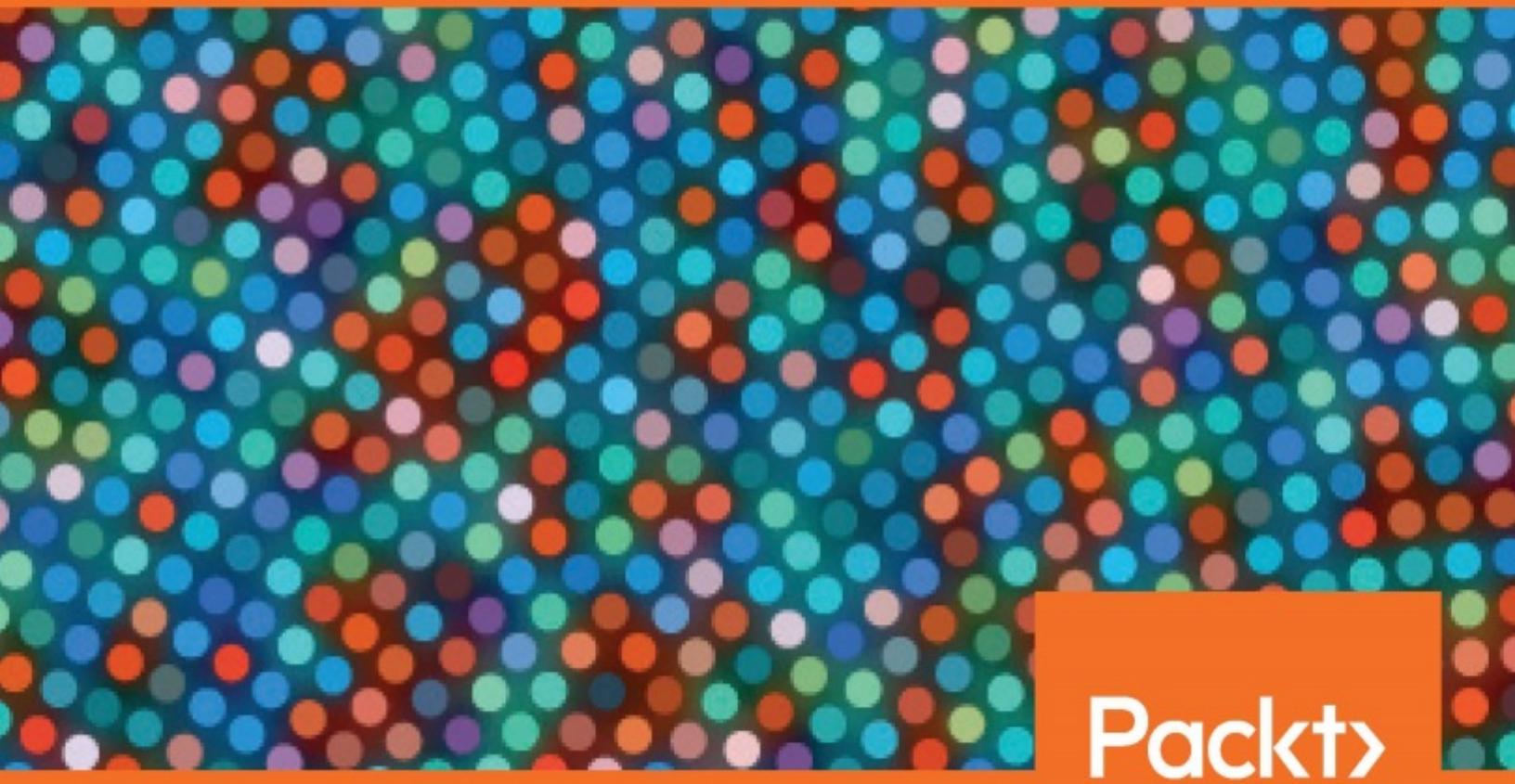


Hands-On Python for Finance

A practical guide to implementing financial analysis strategies using Python

The background of the book cover features a dense, abstract pattern of small, semi-transparent colored circles in shades of blue, green, red, and orange, creating a pixelated or dot-matrix effect.

Packt

www.packt.com

Krish Naik

Hands-On Python for Finance

A practical guide to implementing financial analysis strategies using Python

Krish Naik

Packt

BIRMINGHAM - MUMBAI

Hands-On Python for Finance

Copyright © 2019 Packt Publishing

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

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

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

Commissioning Editor: Sunith Shetty

Acquisition Editor: Nelson Morris

Content Development Editor: Chris D'cruz

Technical Editor: Dinesh Pawar

Copy Editor: Safis Editing

Language Support Editor: Mary McGowan

Project Coordinator: Hardik Bhinde

Proofreader: Safis Editing

Indexer: Rekha Nair

Graphics: Tom Scaria

Production Coordinator: Arvindkumar Gupta

First published: March 2019

Production reference: 1290319

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78934-637-4

www.packtpub.com



mapt.io

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

Why subscribe?

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

Packt.com

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

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

Contributors

About the author

Krish Naik works as a lead data scientist, pioneering in machine learning, deep learning, and computer vision, and is an artificial intelligence practitioner, an educator, and a mentor, with over 7 years' experience in the industry. He also runs a YouTube channel where he explains various topics on machine learning, deep learning, and AI with many real-world problem scenarios. He has implemented various complex projects involving complex financial data with predictive modeling, machine learning, text mining, and sentiment analysis in the healthcare, retail, and e-commerce domains. He has delivered over 30 tech talks on data science, machine learning, and AI at various meet-ups, technical institutions, and community-arranged forums.

I would like to thank God for helping me and guiding me throughout my book. I would most importantly like to thank my parents, siblings (Vish Naik and Reena Naik), friends, students, and colleagues (Deepak Jha and Sudhanshu), who inspired me with their wonderful ideas. Lastly, I would like to dedicate this book to my Dad. He is, and always has been, the backbone of my life.

About the reviewer

Arunkumar N T has attained an MSc (physics) and an MBA (finance), pursuing CMA and CS. He has over 20 years' experience of corporate life and 2 years' experience teaching MBA students. He is an entrepreneur and has previously worked for Airtel, Citi Finance, ICICI Bank, and many other companies.

I would like to thank my parents for their support and trust, in spite of repeated failures; my brothers, Anand and Prabhanjan, for their acknowledgment, love, and support; my wife, Bharathi, for her critical input; my kids, Vardhini and Charvangi, for their naughtiness and also would love to thank my gurus, the late Prof. Badwe and Prof. Sundararajan, for their constant encouragement and guiding me; and my friends, Dr. Sreepathi B and Anand K.

Packt is searching for authors like you

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

Table of Contents

- Title Page
- Copyright and Credits
 - Hands-On Python for Finance
- About Packt
 - Why subscribe?
 - Packt.com
 - Contributors
 - About the author
 - About the reviewer
 - Packt is searching for authors like you
 - Preface
 - Who this book is for
 - What this book covers
 - To get the most out of this book:
 - Download the example code files
 - Download the color images
 - Conventions used
 - Get in touch
 - Reviews
- 1. Section 1: Introduction to Python for Finance

- 1. Python for Finance 101
 - Concepts related to the financial market
 - Public finance
 - Corporate finance
 - Personal finance
 - Understanding the stock market:
 - The primary market
 - The secondary market
 - Comparing the primary market and the secondary market
 - Bonds
 - Types of funds
 - ETFs
 - Mutual funds
 - Hedge funds
- An introduction to derivatives
 - Forward contracts
 - Future contracts
 - Option contracts

- Swap contracts
- Stock splits and dividends
 - What are stock splits?
 - How do stock splits work?
 - Why would a company split a stock?
 - What are reverse stock splits?
 - The advantages of stock splits on dividends
 - A summary of stock splits
- Order books and short selling
 - Short selling
 - Why Python for finance?
 - What is Python?
 - Python for finance
- The Anaconda Python distribution
 - Why Anaconda?
 - Summary

2. Getting Started with NumPy, pandas, and Matplotlib

- Technical requirements
- A brief introduction to NumPy
 - NumPy arrays
 - NumPy's arange function
 - NumPy's zeros and ones functions
 - NumPy's linspace function
 - NumPy's eye function
 - NumPy's rand function
 - NumPy indexing
 - NumPy operations
- A brief introduction to pandas
 - Series
 - DataFrames
 - Selection and indexing in DataFrames
 - Different operations in pandas DataFrames
 - Data input and output operations with pandas
 - Reading CSV files
 - Reading Excel files
 - Reading from HTML files
- Matplotlib
 - The plot function
 - The xlabel function
 - The ylabel function
 - Creating multiple subplots using matplotlib
 - The pie function
 - The histogram function

Summary

Further reading

2. Section 2: Advanced Analysis in Python for Finance

3. Time Series Analysis and Forecasting

Technical requirements

pandas with time series data

Date time index

Time resampling

Timeshifts

Timeseries rolling and expanding using pandas

Introduction to StatsModel libraries

Error trend seasonality models

ETS decomposition

AutoRegressive integrated moving average model

Testing for stationarity

ARIMA code

Autocorrelation function

Partial autocorrelation (PACF)

Summary

4. Measuring Investment Risks

Technical requirements

Sources of financial data

Security risks and returns

Calculating a security's rate of return

Calculating the rate of return of a security in Python using simple
returns

Calculating a security's rate of return using logarithmic return

Measuring the risk of a security

Calculating the risk of a security in Python

Portfolio diversification

Covariance and correlation

Calculating the covariance and correlation

Summary

5. Portfolio Allocation and Markowitz Portfolio Optimization

Technical requirements

Portfolio allocation and the Sharpe ratio

Portfolio allocation and the Sharpe ratio with code

Portfolio optimization

Markowitz portfolio optimization

Assumptions of Markowitz's theory

Obtaining the efficient frontier in Python – part 1

Obtaining the efficient frontier in Python – part 2

Obtaining the efficient frontier in Python – part 3

Summary

6. The Capital Asset Pricing Model

Technical requirements

Understanding the CAPM

The beta of securities

Calculating the beta of a stock;

The CAPM formula

Calculating the expected return of a stock (using the CAPM)

Applying the Sharpe ratio in practice

Obtaining the Sharpe ratio in Python

Measuring alpha and verifying the performance of a portfolio manager

Different use cases of the CAPM using the scipy library

Summary

7. Regression Analysis in Finance

Technical requirements

The fundamentals of simple regression analysis

Running a regression model in Python

Simple linear regression using Python and scikit learn

Computing the slope and the intercepts

Multiple linear regression in Python using the scikit-learn library

Computing the slopes and intercepts

Evaluating the regression model using the R square value

Decision trees; a way to solve non-linear equations

Summary

3. Section 3: Deep Learning and Monte Carlo Simulation

8. Monte Carlo Simulations for Decision Making

Technical requirements

An introduction to Monte Carlo simulation

Monte Carlo simulation applied in the context of corporate finance;

Using Monte Carlo simulation to predict gross profit in Python;

Using Monte Carlo simulation to predict gross profit; part 1

Using Monte Carlo simulation to predict gross profit; part 2

Using Monte Carlo simulation to forecast stock prices in Python

Using Monte Carlo simulation to forecast stock prices; part 1

1

Using Monte Carlo simulation to forecast stock prices; part 2

Using Monte Carlo simulation to forecast stock prices; part 3

Summary

9. Option Pricing - the Black Scholes Model

Technical requirements

An introduction to the derivative contracts

- Forward contracts
- Future contracts
- Option contracts
- Swap contracts

Using the Black Scholes formula for option pricing

Calculating the price of an option using Black Scholes;

Using Monte Carlo in conjunction with Euler Discretization in Python

Summary

10. Introduction to Deep Learning with TensorFlow and Keras

- Technical requirements
- An overview of deep learning in the financial domain
- An introduction to neural networks
 - Neurons
 - Types of neural networks
 - The activation function
 - Types of activation functions
 - The sigmoid activation function
 - The tanh activation function
 - The ReLU activation function

- What is bias?
- How do neural networks learn?

- Gradient descent
- An introduction to TensorFlow
- Implementing linear regression using TensorFlow
- An introduction to Keras
- API categories in Keras
 - The sequential model API
 - The functional API
 - Model subclassing API

Summary

11. Stock Market Analysis and Forecasting Case Study

- Technical requirements
- LSTM RNN intuition
 - How does the RNN work?
 - Backpropagation through time
 - Problems with standard RNNs
 - Vanishing gradient problem in RNN
 - Why earlier layers of the RNN are important for us
 - What harm does it do to our model?
 - Exploding gradients in RNN
 - What are exploding gradients?
 - LSTM

Use case to predict stock prices using LSTM

Data preprocessing of the Google stock data

[Building the LSTM RNN](#)
[Compiling the LSTM RNN](#)
[Making predictions and visualizing data](#)
[Predicting wine sales using the ARIMA model](#)
[Summary](#)

12. What Is Next?

[A summary of different financial applications](#)
[Automating risk management](#)
[Real-time analytics](#)
[Managing customer data](#)
[Financial process automation](#)
[Financial security analysis](#)
[Credit score analysis and underwriting decisions](#)
[Automated algorithmic trading](#)
[Financial recommendations and predictions](#)
[Financial mergers and acquisitions](#)
[Additional research paper links for further reference](#)

[Summary](#)

Other Books You May Enjoy

[Leave a review - let other readers know what you think](#)

Preface

The main purpose of this book is to provide a detailed and intensive introduction to the use of Python in finance. It explores the key characteristics of this powerful and modern programming language to solve problems in finance and risk management.

This book will explain how to code in Python and how to apply these skills in the world of finance. It is both a programming and a finance book. It will provide hands-on experience of various data analysis techniques that are relevant for financial analysis in Python, and of machine learning using sklearn and various stats libraries.

This book explains techniques, from basic to advanced, for applying statistical methods that will be useful for data preprocessing and to predict some real-world scenarios, such as stock prediction and sales prediction. It covers model creation using deep learning techniques, which involve neural network and recurrent neural network concepts with preprocessed financial data including stock prices and sales data, and you will be able to create a predictive model.

This book will also deal with lots of projects related to financial data, and we will see how different machine learning and deep learning algorithms are applied, and how various information and insights are gained from the data in order to make predictions.

At the time of writing in 2019, this book will have the most up-to-date code on Python, along with the latest libraries and techniques used for preprocessing financial data. This book also covers creating models using the best machine learning and deep learning techniques with open source software libraries such as TensorFlow and Keras.

Who this book is for

This book targets the following audience:

- Aspiring data scientists
- Beginners at programming in Python
- Programmers who want to specialize in finance
- Everyone who wants to learn how to code and apply their skills in practice to finance
- Finance graduates and professionals who need to know more about how to apply their knowledge in Python
- The scientific community, working on modeling and simulations using Python as alternative to MATLAB, Mathematica, and R
- Domains of the financial and banking sectors, such as risk analysis, using Python for development, automation, and testing

This book is ideal for graduates who majored in finance, professionals working on financial projects, financial analysts, business analysts, portfolio analysts, quantitative analysts, risk managers, model validators, quantitative developers, and information systems professionals. This book will help the reader to use some of the best techniques available in Python, including machine learning and deep learning to make use of the neural network framework, along with open source software libraries such as TensorFlow and Keras, to gain lots of insights and create predictive models from financial data.

What this book covers

[Chapter 1](#), *Python for Finance 101*, explains key financial concepts. We will explore the various terms and keywords that are used in finance and the stock market in order to better understand the background of quantitative finance.

[Chapter 2](#), *Getting Started with NumPy, pandas, and Matplotlib*, explains how to use the different data analysis libraries such as NumPy, pandas, and matplotlib.

[Chapter 3](#), *Time Series Analysis and Forecasting*, explores time series analysis, which comprises methods for analyzing time series data to extract meaningful statistics and other characteristics of the data. Time series forecasting is using a model to predict future values based on previously observed values. We will focus on the data preprocessing techniques of time series data and use that data to make forecasts.

[Chapter 4](#), *Measuring Investments Risks*, explains that when we think about an investment, we have to consider both its potential advantages and its possible pitfalls. In this chapter, we will discuss how to measure investment risks.

[Chapter 5](#), *Portfolio Allocation and Markowitz Portfolio Optimization*, deals with Markowitz portfolio optimization. We will discuss how we can use Python for an implementation of Markowitz portfolio optimization.

[Chapter 6](#), *The Capital Asset Pricing Model*, discusses the capital asset pricing model (CAPM), which is a model that's used to determine a theoretically appropriate required rate of return of an asset in order to help you make decisions about adding assets to a diverse portfolio. This chapter shows how to create a financial dataset by using time series techniques, and we will use this dataset with the CAPM model.

[Chapter 7](#), *Regression Analysis in Finance*, covers the basics of simple linear regression—a tool commonly used in forecasting and financial analysis.

[Chapter 8](#), *Monte Carlo Simulations for Decision Making*, covers Monte Carlo simulation, which is a computerized mathematical technique that allows people

to account for risk in quantitative analysis and decision making. In this chapter, we will use Monte Carlo simulation to predict the gross profit of a company and to forecast stocks.

[Chapter 9](#), *Option Pricing – the Black Scholes Model*, covers derivatives, the Black Scholes formula for option pricing, and Euler Discretization.

[Chapter 10](#), *Introduction to Deep Learning with TensorFlow and Keras*, discusses neural network frameworks, along with the open source software libraries TensorFlow and Keras, which will help us to create a powerful model.

[Chapter 11](#), *Stock Market Analysis and Forecasting Case Study*, discusses the various use cases related to finance using deep learning techniques with the Keras library.

[Chapter 12](#), *What Is Next*, takes a look at the different use cases in which we can apply all the different techniques to do with Python, machine learning, and deep learning that we have learned throughout this book.

To get the most out of this book

The readers must know the basics of Python, and must install the Anaconda distribution, which is explained in the first chapter of the book.

Download the example code files

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

You can download the code files by following these steps:

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

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

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

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

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

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://www.packtpub.com/sites/default/files/downloads/9781789346374_ColorImages.pdf.

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "By default, the `num` value is `50`, but this can be changed."

A block of code is set as follows:

```
| In [12]:  
| #Example of a list  
| list_1 = [1,2,3]  
| #show  
| list_1
```

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

```
| $ pip install numpy
```

Bold: Indicates a new term, an important word, or words that you see onscreen.



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Reviews

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

For more information about Packt, please visit packt.com.

Section 1: Introduction to Python for Finance

In these chapters, we will look at some financial topics and terminology, and along with that we will see how we can use Python with finance. We will also see how to use some of the basic tools in Python such as NumPy, pandas, and matplotlib.

This section consists of the following chapters:

[Chapter 1](#), *Python for Finance 101*

[Chapter 2](#), *Getting Started with NumPy, pandas, and Matplotlib*

Python for Finance 101

This chapter will help us to understand some of the most common keywords that are used in the financial domain. We will look at various terms related to finance and the stock market that can give us a background understanding of quantitative finance. A lot of these ideas were developed in the 1970s, and they have completely changed the way in which people think about the stock market and the economy in general. In this chapter, we will learn about the many financial topics that relate to quantitative finance, such as stocks, sales, derivatives, and bonds, and then we will explore the various applications of the Python programming language in these fields.

The following topics will be covered in this chapter:

- Understanding the stock market
- Bonds
- Types of funds
- Derivative basics
- Stock splits and dividends
- Order books and short selling
- Why Python for finance?
- The Anaconda Python distribution

Concepts related to the financial market

Finance can be thought of as a technique money management. Financial activity is the use of strategies that people and associations use to deal with their budgetary issues. It is a discipline that typically involves the allocation of belongings and liabilities in situations of risk or uncertainty. People in finance usually rate assets primarily based on their risk, the returns that they're anticipating, and future predictions.

The financial domain can be divided into three subcategories, as follows:

- Public finance
- Corporate finance
- Personal finance

Public finance

Public finance involves looking at the role of the government or the authorities within the financial system. It includes the way in which governments secure and manage their revenues. It can also be thought of as a branch of political economy that studies the general economy of the public sector. Governments finance their expenditure through taxation (such as direct tax, indirect tax, corporation tax, and so on), the borrowing of funds by the general public sector, and the printing of cash in accordance with rules and laws.

Corporate finance

Corporate finance is the essential organ of an enterprise. It involves making decisions regarding the economics and funding of a company. The terms company finance and company financier deal with planning, transactions, or decision-making bodies that are responsible for raising capital that is used to create, broaden, grow, or acquire an organization. These choices are primarily based on the knowledge of numerous stakeholders.

Corporate finance departments are responsible for governing and handling their firm and the decisions made regarding capital investment. These decisions might include whether or not to raise capital through the use of equity, debt, or a hybrid of both.

Personal finance

The techniques or processes involved in handling money by an individual or a household is called personal finance. This process usually includes the following activities:

- Obtaining money in return for services that are provided
- Saving money for the future
- Investing money to get more returns
- Expenditure

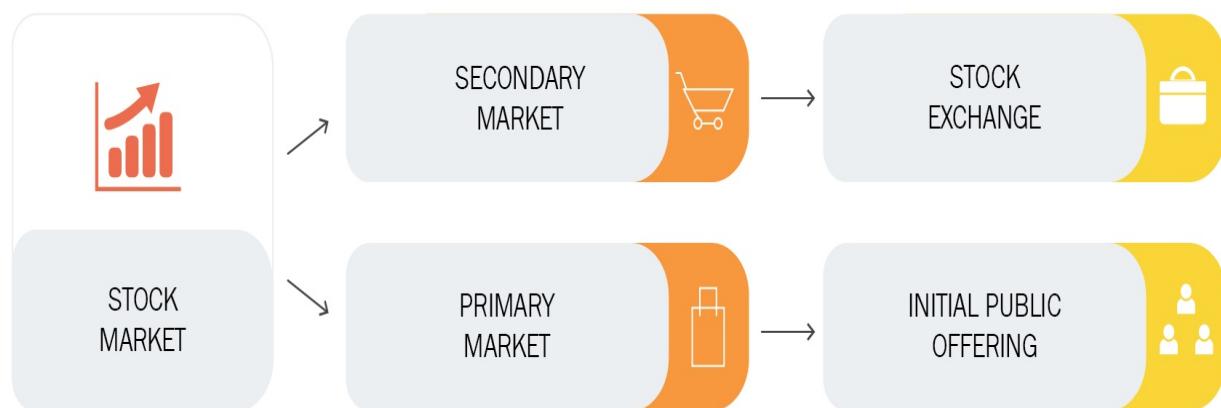
In the 21st century, banking has become a very important part of personal finance. Banks are the entities that actually provides the financial facilities that can help an individual, such as loan, insurance, and saving.

Understanding the stock market

The term stock market is analogous to the term share market, which is where shares are traded. The key distinction is that a stock market enables us to exchange a wide variety of financial items, such as securities, company shares, common assets, subordinate shares, bonds, or mutual funds. A share market solely permits the commercialism of shares.

The stock exchange is the most basic platform that provides services to trade the stocks and securities of different institutes. A stock can only be bought or sold if it's listed on an exchange. It is where stock purchasers and marketers from all over the world gather together.

The following diagram shows the various categories of the stock market:



There are two subcategories of stock market:

- The primary market
- The secondary market

The primary market

Whenever an organization or a company needs to raise some capital funds, they will usually take two different approaches:

- **Debt:** This basically involves borrowing money from another third-party entity. An entity may be a bank, a person, or another company. In this approach, the person or organization is required to pay back the third-party entity by a certain date, or after a certain number of years, with an interest rate added to the total amount, either annually or monthly.
- **Equity:** This involves raising capital for a company by providing ownership to a third-party entity. Here, a company issues some shares in the equity market and people pay to purchase them. An equity investor will have equal rights in the profit and loss of the company, based on the number of shares they own.

The primary market is where the trading of shares happens directly between the company and the investor. The capital amount that is received by the company after issuing new shares is used for expanding its business plan or setting up new ventures. In short, a company gets registered in the primary market to raise funds.

Shares sold by a company who haven't sold shares before are known as an **Initial Public Offering (IPO)**. The process by which the company issues new shares in the primary market is called **underwriting**, and is carried out by security dealers or underwriters. From a retail investor's perspective, making an investment within the primary market is the first step toward buying and selling stocks and shares.

Primary market features include the following:

- It is a market that issues the new shares of an organization for long-term capital, so it is often called a new issue market.
- Securities are issued by an organization directly to the buyers and not via any intermediaries.
- It provides a security certificate to investors as soon as they buy a new

share.

-  *Long-term external finance or loans from monetary establishments are not covered in the primary marketplace. Borrowers can also choose to go public, which means enhancing capital by changing private capital into public capital.*

The secondary market

When a company gets registered in the primary market, it basically means that they have been listed on the stock exchange. This is a virtually-assisted marketplace in which investors usually buy or sell shares. The stock exchange is an example of the secondary market. An investor who has purchased shares from the primary market is not always able to sell the shares back to the original organization. In such cases, they can use the stock exchange to find a suitable purchaser if they want to exit the investment, or if they are in need of some urgent cash. Secondary market transactions are transactions in which one investor buys stocks from another investor at the going market rate, or at a price that the two parties agree upon.

Secondary market features include the following:

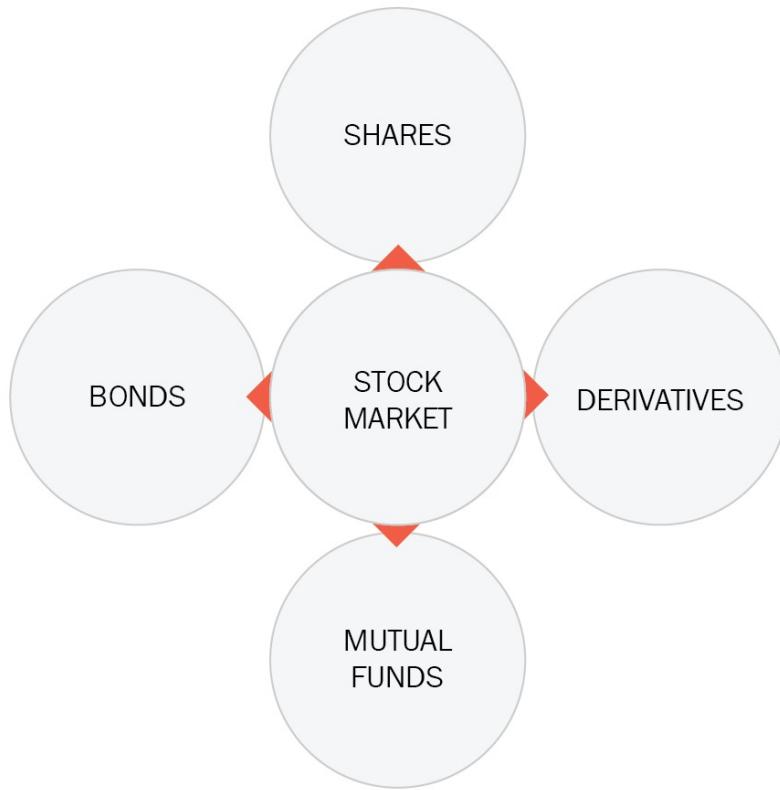
- It provides quick liquidity to sellers or investors who wants to sell their stocks or shares. For example, if an investor has 1,000 Google shares and they want to sell them quickly, they won't go to the CEO of Google, Google's board of directors, or any other stakeholders. Instead, they will sell their shares on the stock exchange.
- The cost of transactions in the secondary market is low, so it is a convenient tool for investors to buy or sell stock.

Comparing the primary market and the secondary market

Now that we have gained an understanding of both the primary and secondary market, let's take a look at some of the differences between them:

Primary market	Secondary market
In the primary market, new shares and bonds from companies are issued.	In the secondary market, shares and bonds that have already been issued are traded.
In the primary market, transactions usually happen between an investor and a company or organization.	In the secondary market, transactions usually happen between investors; the company is not involved.
In the primary market, the share price is determined by the company or the organization.	Here, the share price is determined by supply and demand on the stock exchange.

Let's now take a look at the key financial instruments that are exchanged:



We will talk more about stocks when we look at some use cases to do with the forecasting of stock prices in [Chapter 5, Portfolio Allocation and Markowitz Portfolio Optimization](#), and [Chapter 11, Stock Market Analysis and Forecasting Case Study](#).

Bonds

Bonds are debt instruments that are simple ways for the government and companies to borrow money. Governments and organizations can sell bonds to a large group of investors to raise the funds that are needed for operations and growth.

Bonds are simple instruments that have the following features:

- A fixed coupon rate
- A fixed maturity period
- A fixed principal

For example, let's imagine that company ABC has issued a bond with a principal amount of \$1,000. The maturity period is set to 10 years and the coupon rate, or the interest rate, is 7% per annum. This means that investors who buy the bond will pay \$1,000 for one bond and will get a return of 7% per annum for 10 years. At the end of 10 years, the investor will get back the full principal amount. If the investor wants to sell the bond within the 10-year maturity period, they can do this in the bond market. This is similar to the stock market, but it is used to buy and sell bonds before their maturity date. The prices of bonds change frequently, which is primarily due to the following two factors:

- **Interest rate risk:** This is also known as the market rate risk and refers to the propensity of bonds to change. Let's suppose that an investor has bought a 10-year maturity government bond with a face value of \$1,000, and an interest rate (or coupon rate) of 8%. After two years, the investor wants to sell the bond, but the interest rate for the other bonds is around 10%. It will be difficult for the investor to convince someone to purchase their bond with an interest rate of 8% when someone can buy new bonds with an interest rate of up to 10%. In this scenario, the investor will sell the bond at a discounted price, perhaps for around \$885.



An important point to note is that the bond price has an inverse relationship with the interest rate. When the interest rate increases, the bond price decreases.

- **Credit risk:** This type of risk usually happens when the organization that

has issued the bond is not performing well, so investors fear that the company may not be able to return the required payments. In this case, investors are likely to sell bonds at a discounted price to other investors.

Types of funds

There are three major types of funds:

- **Exchange Traded Funds (ETFs)**
- Mutual funds
- Hedge funds

These funds vary with regard to their fees, transparency, rules, and regulations.

ETFs

ETFs are funds that are made up of a basket of assets, bonds, and commodities. Their holdings are completely public and transparent, and individuals can buy and trade these kinds of funds. Typically, people investing in ETFs are interested in having a diverse portfolio, which refers to the list of stocks and bonds purchased, and want to keep their investment in an ETF for a longer period of time. The return on investment of an ETF depends on the rise in the index or the commodity value. One of the most common ETFs is the Spider (SPY), which tracks the S&P 500 index.

Mutual funds

Mutual funds are created by a group of investors coming together and pooling a certain amount of money to buy stocks, bonds, or both. These funds are managed by a professional fund manager, who aims to build up a portfolio in accordance with a particular investment objective. Investments are often spread across a wide range of different industries, such as IT, telecommunications, or infrastructure. This ensures that the risk is controlled, because the price of the different stocks will not move in the same direction and in the same proportion at the same time.

The most basic structure of mutual funds are units. Mutual funds allocate units to investors based on the amount of money invested. Investors in mutual funds are usually called **unit holders**.

The discussion in this section will be continued in [Chapter 6, *The Capital Asset Pricing Model*](#), where we will be implementing some of these concepts and looking at some examples using Python.

Hedge funds

Hedge funds are also known as pooled funds and are gathered from many high-net-worth individuals. Hedge funds are aggressively managed with the aid of the fund manager and are used in both domestic and international markets, with the aim of generating high returns. It is important to note that hedge funds are generally only accessible to accredited investors, as they require fewer **Securities and Exchange Commission (SEC)** regulations than other funds.

An introduction to derivatives

Derivatives are contracts whose value is based on an underlying asset. An underlying asset can take many forms, such as stocks, bonds, commodities, currencies, interest rates, and market indexes. Derivatives have two main uses:

- **Using derivatives to hedge against risk:** This generally refers to the practice of using derivatives for the objective of minimizing risk in the physical market.
- **Speculation on derivatives:** This is motivated by profit rather than a desire to mitigate risk.

Basically, while hedgers seek to limit risk by using derivatives as insurance policies (thereby indirectly increasing profitability), speculators are directly driven by the opportunity for profit.

There are four main types of derivatives:

- Forward contracts
- Future contracts
- Options contracts
- Swap contracts

Forward contracts

A forward contract is a tweaked contract between two gatherings, where settlement happens on a particular date in the future at a cost that has been incurred today. Forward contracts are not traded on the standard stock exchange and, as a result, they are not standardized, making them particularly useful for hedging. The primary highlights of forward contracts include the following:

- They are reciprocal contracts and are consequently presented to counter party risk.
- Each agreement is handcrafted, and consequently is unique with regard to the measure of the contract, the lapse date, the asset type, and the asset quality.
- The agreement must be settled by a conveyance of the benefit on the lapse date.

Future contracts

A future contact is a contract that is used to sell or buy an underlying asset, such as stocks, bonds, or commodities, at a specified time. A future contract is a lawful consent to purchase or offer a specific commodity or asset at a cost and a predetermined time. Future contracts are similar to forward contracts, but they are standardized and regulated so that they may be traded on a future exchange. They are often used to speculate on commodities.

The purchaser of a future contract assumes the commitment of purchasing the underlying assets when the future contract terminates. The seller of the future contract assumes the commitment of providing the underlying asset on the expiry date. Every future contract has the following features:

- A purchaser
- A vendor
- A cost
- An expiry date

Option contracts

An option contract is a contract that gives someone the right, but not the obligation, to buy (call) or sell (put) security or another financial asset. A call option gives the purchaser the privilege of purchasing the asset at a given cost. This is called the **strike price**. While the holder of the call option has the privilege of requesting an offer from the seller, the vendor, or the seller, has the right to sell, but not necessarily the commitment to do so. If a purchaser wants to purchase the underlying asset, the merchant needs to offer it, the merchants don't have an obligation to do so.

Similarly, a put option gives the purchaser the privilege of selling the asset to the seller at the strike price. Here, the purchaser has the privilege to offer, and the seller has the commitment to purchase. In every option contract, the privilege to exercise the option is vested with the purchaser of the agreement. The seller of the agreement has the right to sell but not the commitment to do so. As the seller of the agreement bears the commitment, they pay a cost, called a **premium**.

Swap contracts

A swap contract is used for the exchange of one cash flow for another set of future cash flows. Swaps refer to the exchange of one security for another, based on different factors.

Why use swap contracts? The main advantages of swap contracts are as follows:

- **Converting financial exposure:** Swap contracts can be used to convert currencies. They can be used to obtain debt financing in the swapped currency at a reduced interest rate brought about by the comparative advantages that each counter-party has in their national capital market, and/or the benefits of hedging long-run exchange rate exposure.
- **Comparative advantages:** This implies that an arbitrage opportunity exists because of a mispricing of the default risk premiums on different types of debt instruments. Swap contracts also allow us to speculate on interest rates, and currencies.

We will continue our discussion on derivatives in [Chapter 5, Portfolio Allocation and Markowitz Portfolio Optimization](#), [Chapter 6, The Capital Asset Pricing Model](#), and [Chapter 9, Option Pricing – Black Scholes Model](#), where we will be implementing portfolio optimization using Python.

Stock splits and dividends

This section of the chapter deals with the following sub-topics:

- What are stock splits?
- How do stock splits work?
- Why would a company split a stock?
- What are reverse stock splits?
- The advantages of stock splits on dividends.
- A summary of stock splits.

What are stock splits?

All organizations that are traded on an open market have a specific number of shares or stock offers that have been issued and acquired by financial specialists. When an organization chooses to split its stock, it increases its number of stocks by issuing extra shares to its current shareholders. The decision to split a stock to expand the quantity of offers by proportionately decreasing the face value is made by the board of directors of an organization.

Usually, companies split their stock using a split ratio. The most commonly used split ratios are 3 to 1, 2 to 1, and 3 to 2, even though any other combination is also possible.

How do stock splits work?

Suppose a share has a face value of \$50 and the company wants to split each share into five shares; this is a 1 to 5 stock split. Each share will be divided into five shares with a face value of \$10. Now, imagine that an organization chooses to use a 2 to 1 stock split. This fundamentally implies that the investors get two extra shares of stock for every one officially held.

If an organization has 10 million offers and chooses to order a 3 to 1 stock split, it will end up with 30 million shares after the split. The estimation of each offer will be reduced by 33.33%. In a 3 to 1 stock split, the cost of the shares changes in line with the fact that 3 shares are the same as the first estimation of a solitary share before the split. While a stock split will build the aggregate number of shares for a given organization, it won't influence the organization's market capitalization, which is the aggregate market estimation of its offers.

Why would a company split a stock?

There are a few reasons why an organization may seek a stock split. If an organization has seen its share cost increase to the extent that investors are being valued out, it might choose to split its stock to make individual shares appear to be more moderately priced. This method is especially valuable when the share price for a particular organization has risen more than those of comparative organizations inside a given division or industry.

Another reason why an organization may split its stock is to do with liquidity. On the off chance that an organization's stocks are viewed as more reasonable, financial specialists may see them as being easier to sell, and therefore, less risky. Also, a stock split can be characteristic of development for any given organization, which is a positive sign for financial specialists. Another reason is that a stock split can actually make stock costs rise. Once a stock turns out to be more moderate and investors buy the stocks, the prices for that stock will increase as a result. When this happens, the cost of the stock will also increase.

What are reverse stock splits?

While a customary stock split expands an organization's aggregate number of shares, a reverse split does the opposite. In a reverse stock split, the aggregate number of offers decreases, which makes the cost of individual offers go up. Reverse stock splits are particularly important when an organization is in danger of being delisted for falling beneath the specific cost for each offer. However, a few organizations utilize reverse stock splits to depict themselves as more respectable players in the market by bragging about higher offer costs.

The advantages of stock splits on dividends

The advantages of stock splits are as follows:

- When the stock is split, investors get extra shares. However, the face value of each share decreases proportionally.
- A stock split creates more liquidity in the share market.
- After a stock split, the share capital and reserves stay the same, in absolute dollar terms, as before.
- If there are any bonus shares, the investors get an extra share of the face value in a predecided proportion as an incentive.
- The main role of issuing bonuses is to remunerate investors by issuing a couple of additional shares.
- When the stock is split, the different shares are immediately reflected in investors' demat accounts the day after the split, and investors can sell these shares without the risk of losing money from their additional shares if the stock price goes down.

A summary of stock splits

Splitting your stock doesn't transform a business or its valuation; it simply increases the quantity of shares and makes each share worth less. Investors get a similar aggregate dividend for the shares, but with less cash coming from each individual share.

A similar rationale applies to different measurements of the cost of a stock. For instance, in the event that you have an option to purchase a specific stock for \$100 per share before 2019, and the stock experiences a 2 to 1 split in 2017, you can choose to purchase twice the same number of offers for \$50. If an organization is anticipated to procure \$2 per share before a 2 to 1 split, it is then anticipated to win \$1 per share if no fundamental changes are made to the business.

Order books and short selling

In order to understand order books, let's consider an example. Person A has to buy or sell a particular stock. Usually, they will take the following steps:

1. Person A logs into their brokerage account (for example, Robinhood, E* Trade, or Ameritrade)
2. They click on the stock that they want to buy or sell
3. They either pay or receive money

Let's think about what actually happens when someone clicks the buy/sell button. The first thing that happens is that the order gets placed. The order includes the following information:

- **Buy or sell:** This indicates whether the person wants to buy or sell the stock.
- **Symbol:** This term usually indicates the code of the stock company that the user wants to buy or sell (such as AAPL or GOOGL).
- **Number of shares:** This field basically indicates the number of shares that we want to sell or buy.
- **Limit or market:** These are the types of order. The maximum price for the order is set by the limit price. If the limit price is not reached in the market, the order usually does not get executed. Market orders are exchanges that are intended to execute as fast as possible at the present or the market cost.
- **Price:** This is only needed for limited orders.

An example of an order would be *BUY, AAPL, 200, Market*, or *BUY, GOOGL, 200, Limit, 1200*.

Short selling

On the most fundamental level, short selling is making a forecast that a stock will go down as opposed to up. It refers to a process in which short sellers borrow an amount of shares or stocks from a broker's account and sell them at the market price. Then, after selling the shares, the goal is basically to re-buy those shares in the future when the prices are down. Finally, the borrowed shares are returned to the broker. Short sellers have to trust that they can benefit from the contrast between the returns from the short sale and the cost of purchasing back the offers, which is often referred to as **short covering**.

Let's consider an example. If a short seller borrows 1,000 shares of \$5 and they sell them at the market price, they will end up with \$5,000 in their account. If the stock's share price reduces to \$3 per share, the short seller has an option to buy back the 1,000 shares at a cost of \$3,000, which gives them a profit of \$2,000 (\$5,000 minus \$3,000) from the trade.

Why Python for finance?

Python for finance is primarily used for quantitative and qualitative analysis, and risk marketing. It has many important applications, including stock market analysis and predictions. Many deep learning and machine learning techniques are also usually carried out with the help of Python. It comes with a number of useful libraries, such as `numpy`, `pandas`, and `sklearn`, which play a very important role in the analysis phase.

What is Python?

Python is a general-purpose, dynamic, high-level, and interpreted programming language. It supports the object-oriented paradigm and it has many important features, such as inheritance, function overriding, function overloading, and abstraction. It is simple to learn and provides lots of high-level data structures.

The following lists the important features of Python:

- Python is simple to learn, read, and write.
- Python is a very good example of a **Free/Libre and Open Source Software (FLOSS)**, which means we can freely distribute copies of this software. Anybody can read its source code and modify it.
- Python is a high-level language. We don't need to bother about low-level details such as memory allocation while writing Python script.
- The Python language has portability, which means it is supported by many platforms, such as Linux, Windows, FreeBSD, Macintosh, Solaris, BeOS, OS/390, PlayStation, and Windows CE.
- Python also supports different programming paradigms. It supports procedure-oriented programming as well as object-oriented programming.
- Python is extensible. Python code can invoke the C and C++ libraries, and can integrate with Java and .NET components.
- The Python language is freely available at the official web address: <https://www.python.org/>.

Python for finance

The financial industry has seen an exponential increase in the amount of data in use. Companies have had to adapt to various administrative and regulatory requirements, meaning that time to market and cost productivity have become key achievement drivers for any monetary institution. Various advanced insights have been available to help with this process for quite a while. Prior to the 1980s, banking and finance were much more difficult to understand.

Lately, software engineering has joined forces with the financial industry in the act of purchasing and offering monetary resources to create benefits. Financial exchanges have become overwhelmed by computers, and various complex algorithms are in charge of settling split-second exchange decisions quicker than humans.

Python has become a very popular language due to its association with rapidly growing fields such as data science, machine learning, and deep learning. Researchers and scientists throughout the world are working on new and innovative ideas using concepts from statistics that integrate well with Python. The recent development of deep learning with Python has caused this language to reach new heights and achieve things that were previously impossible. A lot of AI products are currently being developed using Python.

Python has great potential in the financial domain. Libraries such as NumPy, SciPy, and pandas provide extraordinary insights into data analysis. Likewise, since Python requires less code, the technique by which Python utilizes these libraries is simpler and can be coordinated with the different programming languages and multiple platforms. Python also has a very big programming community that never hesitates to contribute useful information, leading to the growth of Python libraries.

Python can be used in the financial domain in the following scenarios:

- In numerous investment banks, Python is the fundamental language used to construct trade managing platforms. It is also used to create risk management and pricing platforms.

- It can also be utilized for risk administration, backtesting quantitative models, investigating information, exchanging platforms, and automating complex tasks.
- Python can tackle various difficulties raised by the monetary business related to investigation, control, consistency, and information.
- Finance professionals who work in the field of data science and data analytics use R and Python to perform analysis on various datasets. Python has become the language of choice for data analysis and it has been gathering a lot of interest among the community of finance professionals.

The Anaconda Python distribution

Anaconda is an open source package manager that consists of various **Integrated Development Environments (IDEs)**, such as Spyder, Jupyter Notebook, and R Studio, which are tools that provide a platform to write the programming code in Python and R.

Anaconda is an AI/ML enablement platform that allows associations or organizations to create, administer, and computerize AI, ML, and information science from computers or laptops through training and production. It gives industries a chance to scale from individual researchers to join groups of thousands, and to go from a single server to a huge number of hubs or nodes for model training and deployment.

Anaconda has a collection of more than 700 open source packages and it is available in both free and paid versions. The Anaconda distribution ships with the conda command-line utility. You can learn more about Anaconda and conda by reading the Anaconda documentation pages at the following link: <https://anaconda.com/>.

Why Anaconda?

The main reasons why we use Anaconda are given here:

- It provides user-level installation capabilities with respect to Python versions, which basically means that the user can create different environments with different Python versions in the same Anaconda tool.
- The user can update the package and the new libraries by using a simple single command that begins with conda.
- There is no risk or chance of corrupting the system libraries.
- Anaconda has a portability feature; it can be installed on multiple platforms, such as Windows, Linux, and macOS.
- Anaconda has the support of a very huge base of libraries, such as tensorflow, numpy, pandas, and matplotlib.

In Anaconda, we are going to use Jupyter Notebook and Spyder IDE.

Throughout this book, we will be using the Anaconda command prompt to install packages or libraries if they are not installed with the Anaconda tool by default.

The following screenshot shows what Anaconda Navigator looks like after you open it from the start menu:

ANACONDA NAVIGATOR

[Sign in to Anaconda Cloud](#)

Home

Environments

Learning

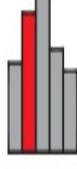
Community

Documentation

Developer Blog

[Twitter](#) [YouTube](#) [GitHub](#)

Applications on myenv Channels Refresh

 Notebook 5.7.4 Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.	 Qt Console 4.4.3 PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.	 Spyder 3.3.2 Scientific Python Development Environment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features
Launch	Launch	Launch
 Glueviz 0.13.3 Multidimensional data visualization across files. Explore relationships within and	 JupyterLab 0.35.3 An extensible environment for interactive and connectable computing based on the	 Orange 3 3.19.0 Component based data mining framework. Data visualization and data analysis for
Launch	Launch	Launch

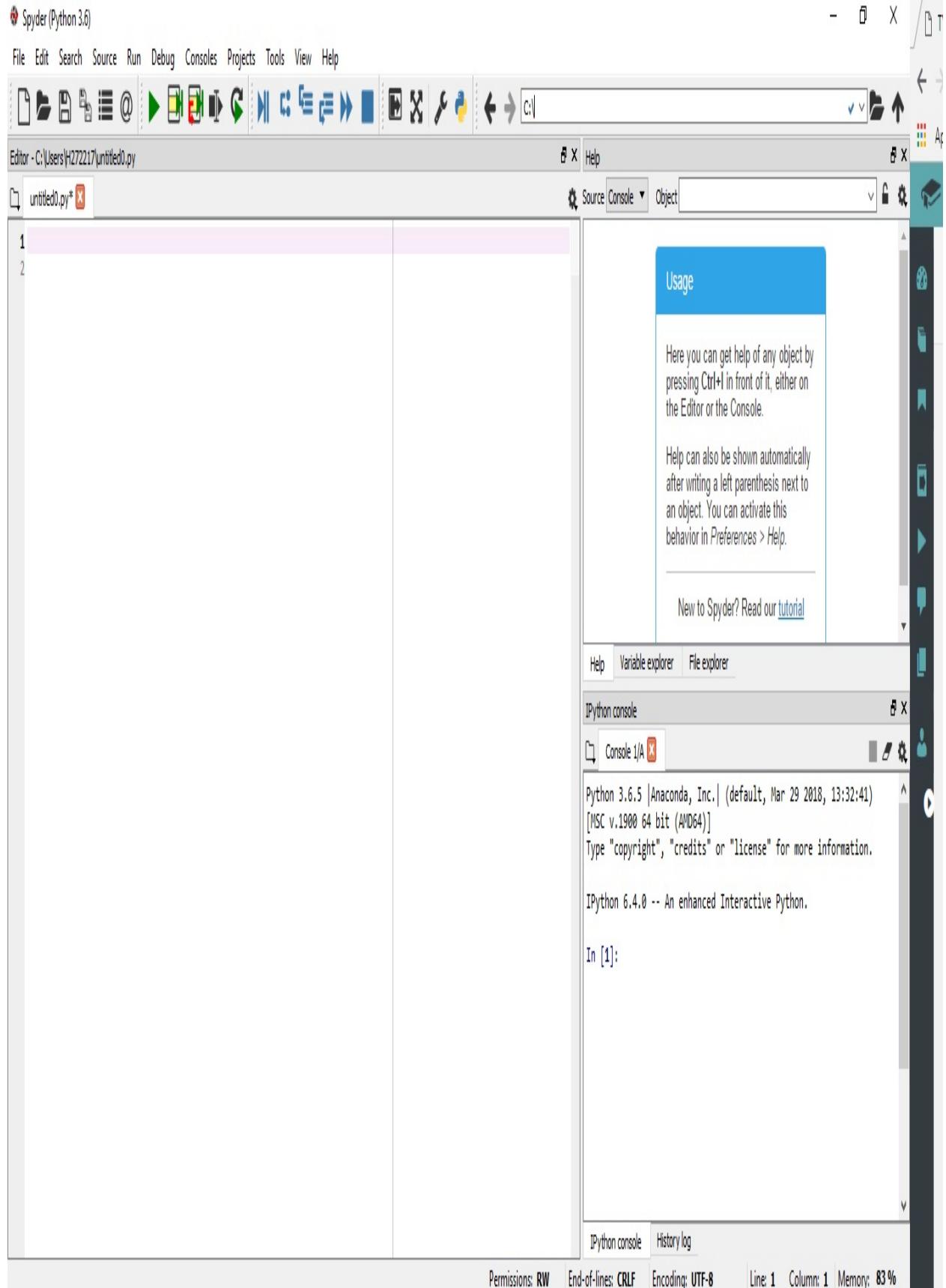
In the preceding screenshot, you can click on Launch to open Jupyter Notebook or the Spyder IDE.

The following screenshot shows Jupyter Notebook:

The screenshot shows the Jupyter Notebook interface. At the top, there is a logo and navigation links for "Files", "Running", and "Clusters". On the right side, there are "Quit" and "Logout" buttons. Below the header, a message says "Select items to perform actions on them." with buttons for "Upload", "New", and a refresh icon. A file list table is displayed, showing the following entries:

	Name	Last Modified	File size
0	/		
<input type="checkbox"/>	3D Objects	3 months ago	
<input type="checkbox"/>	Contacts	a month ago	
<input type="checkbox"/>	Desktop	3 minutes ago	
<input type="checkbox"/>	Documents	a day ago	
<input type="checkbox"/>	Downloads	8 hours ago	

The Spyder IDE looks as follows:



Summary

In this chapter, we looked at some key financial terms that are very important for qualitative and quantitative analysis in finance. We gained an understanding of various concepts to do with the financial markets, including different categories of finance, key financial instruments such as shares or stocks, and also where we can actually trade these stocks, that is, the primary market, and the secondary market. We then looked at the various types of funds and how they are available on various stock exchanges, such as the S&P 500. We also looked at why Python is the best language when financial data is involved.

In the upcoming chapters, we will learn about how to use Python to implement various concepts and case studies. In the next chapter, we will be introduced to some common libraries, such as `numpy`, `pandas`, and `matplotlib`, which will basically help us to handle financial and time-series data, which is the most important step in data preprocessing.

Getting Started with NumPy, pandas, and Matplotlib

In the previous chapter, we looked at some key financial topics. In this chapter, we will take a look at various data analysis libraries that we will use alongside Python.

As the financial domain deals with lots of data that may be in different formats and structures, we often have to analyze, preprocess, and visualize it in order to understand it efficiently. The NumPy, pandas, and matplotlib libraries are very important libraries that are used in Python for data analysis and data preprocessing.

The following topics will be covered in this chapter:

- NumPy
- pandas
- Matplotlib

So, let's get started.

Technical requirements

In this chapter, we will be using the Jupyter Notebook. You also need to install the NumPy, pandas, and matplotlib libraries if they are not already installed.

The code files for this chapter can be found at <https://github.com/PacktPublishing/Han ds-on-Python-for-Finance/tree/master/Chapter%202>.



Another prerequisite of this chapter is that you have Anaconda installed on your computer.

A brief introduction to NumPy

In this section, we will briefly discuss NumPy, its uses, and how to install it on your computer. NumPy is a linear algebra library for Python. All other libraries in the PyData ecosystem rely on NumPy, as it is one of the fundamental building blocks.

Since NumPy has bindings to C libraries, it is incredibly fast. PyData is a result of conferences organized by NumFOCUS, a not-for-profit organization that supports open source software. They frequently hold gatherings in Silicon Valley, Boston, NYC, and, most recently, in London. A significant number of the meeting coordinators are based in Austin, TX, at the Continuum Analytics organization, which was established by Travis Oliphant and Peter Wang. Leah Silen is the fundamental coordinator of the gatherings, but there are also volunteer coordinators who help with organizing the conferences.

PyData refers to the group that is primarily involved in using Python and its various libraries for data preprocessing and data analysis. It is more business-centered than SciPy, which is made up of the company Enthought and is more focused on academic applications. The two networks have a lot in common, but you'll discover more finance-related themes with PyData.

To install NumPy, follow these steps:

1. It is highly recommended that you install Python using the Anaconda distribution to make sure that all the underlying dependencies (such as linear algebra libraries) sync up with conda installation.
2. If you already have Anaconda, install NumPy by going to your Terminal or Anaconda Command Prompt and typing the following command:

```
|     conda install numpy
```

3. If you do not have the Anaconda distribution and you have installed Python manually, type the following command from the Terminal:

```
|     pip install numpy
```

Now that we have installed NumPy, let's discuss the following topics:

- NumPy arrays
- NumPy indexing
- Various NumPy operations

NumPy arrays

NumPy is a library that is used in the Python programming language to create single and multidimensional arrays and matrices. It also supports various built-in functions that can perform high-level mathematical functions and operations on these arrays.

There are two main variants of NumPy arrays, as follows:

- Vectors
- Matrices

Vectors are arrays that are strictly one-dimensional, whereas matrices are two-dimensional. They can still only have, however, one row or column.

In this section, we are going to learn about the various ways to create NumPy arrays using Python and the NumPy library. We are going to use the Jupyter Notebook to show the programming code.

To begin using the NumPy library, we need to import it, as follows:

```
| In [1]: import numpy as np
```

Let's create a NumPy array using a Python object. First, we will create a list and then convert it to a NumPy array:

```
| In [12]:  
| #Example of a list  
| list_1 = [1,2,3]  
| #show  
| list_1  
|  
| Out[12]:  
| [1, 2, 3]
```

We can also assign this array to a variable and use it where necessary:

```
| In [14]:  
| #Assign array to variable  
| list_array=np.array(list_1)  
| #Print list_array  
| print(list_array)
```

```
| Out[14]
| [1 2 3]
```

We can cast a normal Python list into an array. Currently, this is an example of a one-dimensional array. We can also create an array by directly converting a list of lists or a nested list:

```
| In [15]:
| nested_list= [[1,2,3],[4,5,6],[7,8,9]]
| #show
| nested_list
|
| Out[15]:
| [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Let's now take a look at some of the built-in methods to generate arrays using NumPy.

NumPy's arange function

`arange` is a built-in function provided by the NumPy library to create an array with even-spaced elements according to a predefined interval. The syntax for NumPy's `arange` function is as follows:

```
| arange([start,] stop[, step,], dtype=None)
```

The following examples demonstrate the basic usage of the `arange` function:

```
| In [9]: np.arange(0,10,2)
| Out[9]: array([ 0,  2,  4,  6,  8, 10])
```

NumPy's zeros and ones functions

The `zeros` and `ones` functions are built-in functions used to create arrays of zeros or ones; the syntax is as follows:

```
| np.zeros(shape)
| np.ones(shape)
```

The following examples show us how to create an array using `zeros` and `ones`:

```
In [24]:
np.zeros(3)

Out[24]:
array([ 0.,  0.,  0.])

In [26]:
np.zeros((5,5))

Out[26]:
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```



```
In [27]:
np.ones(3)

Out[27]:
array([ 1.,  1.,  1.])

In [28]:
np.ones((3,3))

Out[28]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

NumPy's linspace function

The `linspace` function is a built-in function that returns an evenly-spaced number over a specified interval. `linspace` requires three important attributes: `start`, `stop`, and `num`. By default, the `num` value is `50`, but this can be changed.

The syntax for NumPy's `linspace` function is as follows:

```
np.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
Docstring: Return evenly spaced numbers over a specified interval

In [6]:
np.linspace(0,10,3)

Out[6]:
array([ 0.,  5., 10.])

In [7]:
np.linspace(0,10,50)
```

The following output essentially returns 50 evenly-spaced numbers over a specified interval between 0 and 10:

```
Out[7]:
array([ 0.          ,  0.20408163,  0.40816327,  0.6122449 ,  0.81632653,
       1.02040816,  1.2244898 ,  1.42857143,  1.63265306,  1.83673469,
       2.04081633,  2.24489796,  2.44897959,  2.65306122,  2.85714286,
       3.06122449,  3.26530612,  3.46938776,  3.67346939,  3.87755102,
       4.08163265,  4.28571429,  4.48979592,  4.69387755,  4.89795918,
       5.10204082,  5.30612245,  5.51020408,  5.71428571,  5.91836735,
       6.12244898,  6.32653061,  6.53061224,  6.73469388,  6.93877551,
       7.14285714,  7.34693878,  7.55102041,  7.75510204,  7.95918367,
       8.16326531,  8.36734694,  8.57142857,  8.7755102 ,  8.97959184,
       9.18367347,  9.3877551 ,  9.59183673,  9.79591837, 10.        ])
```

NumPy's eye function

The `eye` function is a built-in function in NumPy that is used to create an identity matrix. An identity matrix is a type of matrix in which all the diagonal elements are one, and the remaining elements are zero:

```
In [8]:  
np.eye(4)  
  
Out[8]:  
array([[1., 0., 0., 0.],  
       [0., 1., 0., 0.],  
       [0., 0., 1., 0.],  
       [0., 0., 0., 1.]])
```

NumPy's rand function

NumPy has a built-in function called `rand` to create an array with random numbers. The `rand` function creates an array of a specified shape and populates it with random samples from a uniform distribution over [0, 1]. This essentially means that a random value between 0 and 1 will be selected, as shown in the following code block:

```
In [47]:  
np.random.rand(2)  
  
Out[47]:  
array([ 0.11570539,  0.35279769])  
  
In [46]:  
  
np.random.rand(5,5)  
  
Out[46]:  
  
array([[ 0.66660768,  0.87589888,  0.12421056,  0.65074126,  0.60260888],  
       [ 0.70027668,  0.85572434,  0.8464595 ,  0.2735416 ,  0.10955384],  
       [ 0.0670566 ,  0.83267738,  0.9082729 ,  0.58249129,  0.12305748],  
       [ 0.27948423,  0.66422017,  0.95639833,  0.34238788,  0.9578872 ],  
       [ 0.72155386,  0.3035422 ,  0.85249683,  0.30414307,  0.79718816]])
```

The `randn` function creates an array of a specified shape and populates it with random samples from a standard normal distribution:

```
In [1]:  
import numpy as np  
In [2]:  
np.random.randn(2)  
  
Out[2]:  
array([ 0.1426585 , -0.79882962])  
  
In [3]:  
np.random.randn(5,5)  
  
Out[3]:  
array([[ -0.31525094, -0.76859012,  0.72035964,  0.7312833 , -0.57112783],  
       [ 0.47523585,  0.18562321, -1.42741078, -0.50190548,  0.39230943],  
       [ -0.06597815, -0.92100907,  0.27146975, -0.84471005, -0.09242036],  
       [ -1.70155241, -0.79810538,  0.04569422,  0.1908103 ,  0.15467256],  
       [ 0.36371628, -0.39255851,  0.02732152, -1.62381529,  0.42104139]])
```

The `randint` function creates an array by returning random integers, from a low value (inclusive) to a high value (exclusive). It is a built-in function of the `random` module in Python 3. The `random` module provides access to various useful

functions, one of which is `randint()`.

The syntax of the `randint()` function is as follows:

```
| randint(start, end)
```

The following are the parameters required inside the `randint()` function:

```
| (start, end) : Both of them must be integer type values.
```

The following is the value returned from the `randint()` function:

```
| A random integer within the given range as parameters.
```

The following are the errors and exceptions usually given by the `randint()` function based on the input:

```
| ValueError : Returns a ValueError when floating  
| point values are passed as parameters.  
TypeError : Returns a TypeError when anything other than  
  
In [6]:  
np.random.randint(1,100)  
  
Out[6]:  
1  
  
In [7]:  
np.random.randint(1,100,10)  
  
Out[7]:  
array([95, 11, 47, 22, 63, 84, 16, 91, 67, 94])  
  
Out[11]:  
array([0, 2, 4, 6, 8])
```

Now let's discuss NumPy indexing. Here, indexing will help us to retrieve the elements of the array in a much faster way.

NumPy indexing

In this section, we are going to see how we can select elements or a group of elements from an array. We are also going to look at the indexing and selection mechanisms. First, let's create a simple one-dimensional array and look at how the indexing mechanism works:

```
In [2]:  
import numpy as np  
  
In [3]:  
#Creating a sample array  
arr_example = np.arange(0,11)  
  
In [4]:  
#Show the array  
arr_example  
  
Out[4]:  
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In the preceding code, we created an array with the `arr` variable name. Let's now see how we can select elements using indexing:

```
In [5]:  
#Get a value at an index  
arr_example[8]  
  
Out[5]:  
8  
  
In [6]:  
#Get values in a range  
arr_example[1:5]  
  
Out[6]:  
array([1, 2, 3, 4])  
  
In [7]:  
#Get values in a range  
arr_example[0:5]  
  
Out[7]:  
array([0, 1, 2, 3, 4])
```

We can also update a range of array values by using broadcasting techniques:

```
In [8]:  
#Setting a value with index range (Broadcasting)  
arr_example[0:5]=100
```

```
#Show  
arr_example  
  
Out[8]:  
array([100, 100, 100, 100, 100, 5, 6, 7, 8, 9, 10])
```

Let's take a look at how the indexing mechanism works for a two-dimensional array.



The general format for a two-dimensional array is either `array_2d[row][col]` or `array_2d[row, col]`. It is recommended to use comma notation for clarity.

The first step is to create a two-dimensional array and use the same indexing mechanism with slicing techniques to retrieve the elements from the array:

```
In [4]:  
arr_2d = np.array([[5,10,15],[20,25,30],[35,40,45]])  
  
#Show  
arr_2d  
  
Out[4]:  
array([[ 5, 10, 15],  
       [20, 25, 30],  
       [35, 40, 45]])  
  
In [5]:  
#Indexing row  
arr_2d[1]  
  
Out[5]:  
array([20, 25, 30])  
  
In [6]:  
# Format is arr_2d[row][col] or arr_2d[row,col]  
  
# Getting individual element value  
arr_2d[1][0]  
  
Out[6]:  
20  
  
In [7]:  
# Getting individual element value  
arr_2d[1,0]  
  
Out[7]:  
20
```

The slicing technique helps us to retrieve elements from an array with respect to their indexes. We need to provide the indexes to retrieve the specific data from the array. Let's take a look at a few examples of slicing in the following code:

```
In [8]:  
# 2D array slicing technique  
arr_2d[:2,1:]
```

```
Out[8]:  
array([[10, 15],  
       [25, 30]])  
  
In [9]:  
arr_2d[2]  
  
Out[9]:  
array([35, 40, 45])  
  
In [10]:  
#Shape bottom row  
arr_2d[2,:]  
  
Out[10]:  
array([35, 40, 45])
```

NumPy operations

We can easily perform arithmetic operations using a NumPy array. These operations may be array operations, array arithmetic, or scalar operations:

```
In [3]:  
import numpy as np  
arr_example = np.arange(0,10)  
  
In [4]:  
arr_example + arr_example  
  
Out[4]:  
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])  
  
In [5]:  
arr_example * arr_example  
  
Out[5]:  
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])  
  
In [6]:  
arr_example - arr_example  
  
Out[6]:  
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

NumPy has many universal array functions that are essential mathematical techniques that anyone can use to perform arithmetic operations:

```
In [10]:  
#Taking Square Roots  
np.sqrt(arr_example)  
  
Out[10]:  
array([0.          , 1.          , 1.41421356, 1.73205081, 2.          ,  
      2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.          ])  
  
In [11]:  
#Calculating exponential (e^)  
np.exp(arr_example)  
  
Out[11]:  
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,  
      5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,  
      2.98095799e+03, 8.10308393e+03])  
  
In [12]:  
np.max(arr_example) #same as arr.max()  
  
Out[12]:  
9
```

A brief introduction to pandas

In the previous section, we discussed the NumPy library, its built-in functions, and its applications. We are now going to move on to discussing the pandas library. The pandas library is very powerful, and is one of the most important tools for data analysis and data preprocessing. It is an open source library that is built on top of NumPy. It provides many important features, such as fast data analysis, data cleaning, and data preparation. We provide data as input to our machine learning or deep learning models for training purposes. pandas has high performance and high productivity; it also has many built-in visualization features. One of the most important attributes of the pandas library is that it can work with data from a variety of data sources, such as **comma-separated value (CSV)** files, HTML, JSON, and Excel.

In order to use the pandas library, we will need to install it by going to the Anaconda prompt, the Command Prompt, or the Terminal, and typing in the following command:

```
| conda install pandas
```

We can also use the `pip install` command to install the pandas library:

```
| pip install pandas
```

The following topics will be covered in this section:

- Series
- DataFrames
- pandas operations
- pandas data input and output

Series

Series is the first datatype that we will be discussing in pandas. A series is a one-dimensional array with custom labels or indexes that have the ability to hold different types of data, such as integer, string, float, and Python objects.

A simple series in pandas can be created using the following constructor:

```
| pandas.Series( data, index, dtype, copy)
```

The following table is a detailed explanation of the parameters used inside series:

Serial number	Parameters	Description
1	data	Data can be provided in the form of lists, n -dimensional arrays, or constants.
2	index	The index values must be provided as unique values. They should be of the same length as the <code>data</code> value provided in the first parameter. If no index is passed, a series usually takes up <code>numpy.arange(n)</code> as default, as discussed in the <i>NumPy</i> section of Chapter 1, Python for Finance 101 .
3	<code>dtype</code>	<code>dtype</code> stands for datatype. The default value for the <code>dtype</code> parameter is <code>None</code> . If <code>dtype</code> is <code>None</code> , the datatype will be inferred.
	<code>copy</code>	This allows you to copy data; the default value is

4

false.

A series can be created by using a variety of inputs, such as the following:

- An array input
- A dictionary input
- A constant or scalar input

Let's consider how we can create series using the Python and pandas libraries. First, we will import the NumPy and pandas libraries:

```
In [1]:  
import numpy as np  
import pandas as pd
```

We can create series using various datatypes, such as lists, arrays, dictionaries, and constants:

```
In [2]:  
labels_example = ['a', 'b', 'c']  
list_example = [10, 20, 30]  
dictionary_example = {'a':10, 'b':20, 'c':30}  
arr_example = np.array([10, 20, 30])
```

Series can be also be created from a list, as shown in the following code block:

```
In [3]:  
pd.Series(data=list_example)
```

The output is as follows:

```
Out[3]:  
0    10  
1    20  
2    30  
dtype: int64
```

In the following code, we are passing `list_example`, the indexes are set to `labels_example`, and we are converting them into series:

```
In [4]:  
pd.Series(data=list_example, index=labels_example)
```

The output is as follows:

```
| Out[4]:  
| a    10  
| b    20  
| c    30  
| dtype: int64
```

The following example shows how to create a series from an array:

```
| In [5]:  
| pd.Series(arr_example)
```

The output is as follows:

```
| Out[5]:  
| 0    10  
| 1    20  
| 2    30  
| dtype: int32
```

In the following example, we have passed `array_example` as our data and `labels_example` as our index, and converted them into series:

```
| In [6]:  
| pd.Series(arr_example, labels_example)
```

The output is as follows:

```
| Out[6]:  
| a    10  
| b    20  
| c    30  
| dtype: int32
```

The following example shows how to create a series from a dictionary:

```
| In [7]:  
| d.Series(dictionary_example)
```

The output is as follows:

```
| Out[7]:  
| a    10  
| b    20  
| c    30  
| dtype: int64
```

pandas series can also hold various objects, including built-in Python functions such as `data`, as demonstrated in the following code:

```
| In [8]:  
| pd.Series(data=labels_example)
```

The output is as follows:

```
| Out[8]:  
| 0    a  
| 1    b  
| 2    c  
| dtype: object
```

In the following code, we are adding built-in functions, such as `print` and `len`, and converting into series:

```
| In [10]:  
| # Even functions (although unlikely that you will use this)  
| pd.Series([print,len])
```

The output is as follows:

```
| Out[10]:  
| 0    <built-in function print>  
| 1    <built-in function len>  
| dtype: object
```

Now that we have seen how to create series using different datatypes, we should also learn how to retrieve values in the series using indexes. pandas makes use of index values, which allow fast retrieval of data; this is shown in the following example:

```
| In [14]:  
| series1 = pd.Series([1,2,3,4],index = ['A','B','C','D'])  
| series1
```

The output is as follows:

```
| Out[15]:  
| A    1  
| B    2  
| C    3  
| D    4  
| dtype: int64
```

The following code shows additional examples of how to create a series:

```
| In [16]:  
| series2 = pd.Series([1,2,5,4],index = ['B','C','D','E'])  
| In [17]:  
| series2
```

The output is as follows:

```
| Out[17]:  
| B    1  
| C    2  
| D    5  
| E    4  
| dtype: int64
```

The following code demonstrates how we can retrieve the series values by using indexing:

```
| In [19]:  
| # Retrieving through index  
| series1[1]  
  
| Out[19]:  
| 2
```

Different arithmetic operations can be performed on series with respect to values. In the following code, we will subtract the series:

```
| In [23]:  
| series1-series2
```

The output is as follows:

```
| Out[23]:  
| A    NaN  
| B    1.0  
| C    1.0-1.0  
| E    NaN  
| dtype: float64
```

The following code helps us to add two series – `series1` and `series2`:

```
| In [24]:  
| series1 + series2
```

The output is as follows:

```
| Out[24]:  
| A    NaN  
| B    3.0  
| C    5.0  
| D    9.0  
| E    NaN  
| dtype: float64
```

We can even multiply two or any number of series; the code is as follows:

```
| In [25]:
```

```
| series1 * series2
```

The output is as follows:

```
| Out[25]:  
| A      NaN  
| B      2.0  
| C      6.0  
| D     20.0  
| E      NaN  
| dtype: float64
```



NaN values are displayed here because the indexes in the two series don't match. If the index present in `series1` is not present in the variables of `series2`, the value displays a `NaN` value for that particular index.

DataFrames

A DataFrame is the most popular tool that we will use in the pandas library. A DataFrame is essentially a data structure, usually with a two-dimensional shape, where the data is in a tabular format, consisting of rows and columns. DataFrames are the workhorse of pandas and are directly inspired by the R programming language. You can think of a DataFrame as a bunch of series objects put together, sharing the same index.

Here are the most important features of DataFrames:

- The data in the columns is of different datatypes.
- They are mutable and can be changed with respect to their size.
- A lot of complex arithmetic operations can be performed on the rows and columns.
- All the rows and columns can be accessed via a unique feature called an **axis**.

A pandas DataFrame can be created using the following constructor:

```
| pandas.DataFrame( data, index, columns, dtype, copy)
```

The parameters of the constructor are as follows:

Serial number	Parameter and description
1	<code>data</code> : The <code>data</code> parameter can be provided in the form of lists, n -dimensional arrays, series, constants, or other DataFrames. This parameter essentially forms the values of the columns.
2	<code>index</code> : The <code>index</code> value is provided for the row labels. These labels act as an index for the complete rows, which are made of various columns provided by the <code>data</code> parameters.

3	<code>columns</code> : This parameter is to provide the column names or headings.
4	<code>dtype</code> : This is the datatype of each column.
5	<code>copy</code> : This parameter is used for copying data; the default value is FALSE.



Usually, `data` parameter values are given in the form of a two-dimensional array in the shape of (n, m) . The second parameter, index length, should be equal to the n value, while the column length should be equal to the m value.

A DataFrame can be created using various different types of input, such as lists, dictionaries, NumPy arrays, series, or other DataFrames. Let's consider some examples; the first step is to import the NumPy and pandas libraries, as we will need to create some arrays and DataFrames. The remaining tasks are quite simple; we need to add the elements provided in the syntax.

Let's import the library, as follows:

```
In [5]:
import pandas as pd
import numpy as np
```

We will also be importing the `randn` library, which will help us to create random numbers:

```
In [6]:
from numpy.random import randn
np.random.seed(55)
```

The following code helps us to create a DataFrame:

```
In [7]:
dataframe = pd.DataFrame(randn(4,5),index=['P','Q','R','S'],columns=['A','B','C','D','E']
In [8]:
dataframe.head()
```

The output is as follows:

Out[8]:

	A	B	C	D	E
P	-1.623731	-0.101784	-1.809791	0.262654	0.259953
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

Selection and indexing in DataFrames

This section is about using the selection and indexing mechanism to retrieve elements from DataFrames. The selection and indexing mechanism works in pretty much the same way as we saw in NumPy, but there are some syntax changes. There will be some properties, such as `loc` and `iloc` in pandas, that are not present in NumPy. In the following code, we will retrieve the elements from the DataFrame:

```
| In [9]:  
| dataframe['A']
```

The output is as follows:

```
| Out[9]:  
| P    -1.623731  
| Q    -0.381086  
| R    1.656445  
| S    1.368799  
| Name: A, dtype: float64
```

We can also pass a list of column names, as follows:

```
| In [10]:  
| # Passing a list of column names  
| dataframe[['A', 'B']]
```

The output is as follows:

Out[10]:

	A	B
P	-1.623731	-0.101784
Q	-0.381086	-0.002290
R	1.656445	-1.189009
S	1.368799	0.258169

The columns in the DataFrames are essentially a combination of series:

```
| In [12]:
```

```
| type(dataframe['A'])  
| Out[12]:  
| pandas.core.series.Series
```

A new column can be created using various arithmetic operations:

```
| In [14]:  
| dataframe['F'] = dataframe['A'] + dataframe['B']  
| In [15]:  
| dataframe
```

The output is as follows:

Out[15]:

	A	B	C	D	E	F
P	-1.623731	-0.101784	-1.809791	0.262654	0.259953	-1.725515
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100	-0.383376
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873	0.467436
S	1.368799	0.258169	0.702352	0.888382	0.722220	1.626967

Similarly, DataFrames have a built-in function to drop a column:

```
| In [16]:  
| dataframe.drop('F', axis=1)
```

The output is as follows:

Out[16]:

	A	B	C	D	E
P	-1.623731	-0.101784	-1.809791	0.262654	0.259953
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

From the following code, you can see that even though we have dropped the `F` column, we still see the column being displayed:

```
In [17]:  
dataframe
```

The output is as follows:

Out[17]:

	A	B	C	D	E	F
P	-1.623731	-0.101784	-1.809791	0.262654	0.259953	-1.725515
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100	-0.383376
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873	0.467436
S	1.368799	0.258169	0.702352	0.888382	0.722220	1.626967

So, how is this possible? `drop` is a very risky operation, as it allows us to drop or delete a column. In order to prevent us from doing this by mistake, there is an extra parameter, `inplace`, which is usually used to ensure that the user really wants to do a `drop` operation; let's take a look at an example:

```
In [17]:  
dataframe.drop('F', axis=1, inplace=True)
```

```
In [19]:  
dataframe
```

The output is as follows:

Out[19]:

	A	B	C	D	E
P	-1.623731	-0.101784	-1.809791	0.262654	0.259953
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

When the `axis` value is `1`, the columns are usually dropped. Similarly, to drop a row with a particular index, the `axis` value is set to `0`:

```
In [24]:  
dataframe.drop('P', axis=0, inplace=True)  
dataframe
```

The output is as follows:

Out[24]:	A	B	C	D	E
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

There are more built-in properties available for DataFrames that are to do with selecting rows and columns:

- **Selecting rows in DataFrames:** The built-in properties used for selecting rows are `dataframe.loc` and `dataframe.iloc`. In the first property, we usually provide the index name, while in the latter, we provide the index number.

The following syntax is used for selecting the rows of a DataFrame:

```
In [32]:  
dataframe.loc['Q']
```

The output is as follows:

```
Out[32]:  
A    -0.381086  
B    -0.002290  
C     0.341615  
D     0.897572  
E    -0.361100  
Name: Q, dtype: float64
```

Alternatively, we can select a value based on its position, instead of its label:

```
In [30]:  
dataframe.iloc[0]
```

The output is as follows:

```
Out[30]:  
A    -0.381086  
B    -0.002290  
C     0.341615  
D     0.897572  
E    -0.361100
```

```
|   Name: Q, dtype: float64
```

Here, the output returns the complete row with respect to the column values.

- **Selecting a subset of rows and columns in a DataFrame:** The same built-in property, `dataframe.iloc`, is used to select subsets of rows and columns in a DataFrame. This selection requires the use of the slicing operation that we discussed in the *NumPy* section in the previous chapter; the code is as follows:

```
In [33]:  
# Select all the rows and columns  
dataframe.iloc[:]
```

The output is as follows:

Out[33]:

	A	B	C	D	E
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

In the following code, we will skip the last column from the DataFrame:

```
In [38]:  
#Skip the last column from the dataframe  
dataframe.iloc[:, :-1]
```

The output is as follows:

Out[38]:

	A	B	C	D
Q	-0.381086	-0.002290	0.341615	0.897572
R	1.656445	-1.189009	1.666429	-2.003439
S	1.368799	0.258169	0.702352	0.888382

In the following code, we will select the subset of `Q` and `R` rows along with the `B` and `C` columns:

```
| In [38]:  
| #Select subset Q and R rows along with B and C columns  
| dataframe.iloc[:2,1:3]
```

The output is as follows:

Out[40]:

	B	C
Q	-0.002290	0.341615
R	-1.189009	1.666429

In the following code, we will select the subset of rows and columns using `dataframe.loc`:

```
| In [41]:  
| # Select subset of rows and column using dataframe.loc  
| dataframe.loc['Q','A']  
  
| Out[41]:  
| -0.3810863829200849  
  
| In [42]:  
| dataframe  
| dataframe.loc[['Q','R'],['B','C']]
```

The output is as follows:

Out[42]:

	B	C
Q	-0.002290	0.341615
R	-1.189009	1.666429

DataFrames also allow conditional selection, if we provide the conditions in brackets:

```
| In [43]:  
| dataframe
```

The output is as follows:

Out[43]:

	A	B	C	D	E
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

The following code returns `True` if greater than zero:

```
In [43]:  
dataframe>0
```

The output is as follows:

Out[44]:

	A	B	C	D	E
Q	False	False	True	True	False
R	True	False	True	False	False
S	True	True	True	True	True

The following code will return values that are greater than 1:

```
In [46]:  
#return the values where the value> 0 else returns NaN  
dataframe[dataframe>0]
```

The output is as follows:

Out[46]:

	A	B	C	D	E
Q	NaN	NaN	0.341615	0.897572	NaN
R	1.656445	NaN	1.666429	NaN	NaN
S	1.368799	0.258169	0.702352	0.888382	0.72222

The following code returns the complete rows and columns where the value of the A column is greater than 0:

```
| In [47]:  
| #Returns the complete rows and columns where the value of columns A > 0  
| dataframe[dataframe['A']>0]
```

The output is as follows:

Out[47]:

	A	B	C	D	E
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

If we need to apply multiple conditions within the DataFrame, we use | and & with parentheses:

```
| In [51]:  
| dataframe[(dataframe['A']>0) & (dataframe['B'] > 0)]
```

The output is as follows:

Out[51]:

	A	B	C	D	E
S	1.368799	0.258169	0.702352	0.888382	0.72222

There is also a built-in function that can reset the index in the DataFrame to the default index, as in NumPy (such as 0, 1, 2, 3, ..., n):

```
| In [52]:  
| dataframe
```

The output is as follows:

Out[52]:

	A	B	C	D	E
Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
S	1.368799	0.258169	0.702352	0.888382	0.722220

The `reset_index()` method reset the index to default index as shown here:

```
In [53]:  
# reset_index() reset to default 0,1...n index  
dataframe.reset_index()
```

The output is as follows:

Out[53]:

	index	A	B	C	D	E
0	Q	-0.381086	-0.002290	0.341615	0.897572	-0.361100
1	R	1.656445	-1.189009	1.666429	-2.003439	-0.477873
2	S	1.368799	0.258169	0.702352	0.888382	0.722220

Different operations in pandas DataFrames

There are many different operations and built-in functions available in pandas DataFrames. These are very useful for data analysis and data preprocessing. We can create a DataFrame and apply various built-in functions to make different tasks easier. Let's now create a DataFrame and try different operations on it:

```
| In [4]:  
| #Creating a dataframe with dictionary items within it as key value pairs. The keys are t  
| import pandas as pd  
| df = pd.DataFrame({'col1':[10,11,12,13], 'col2':[100,200,300,400], 'col3':['abc','def','ghi']})  
| df.head()
```

The output for the preceding code will be as follows:

Out[4]:

	col1	col2	col3
0	10	100	abc
1	11	200	def
2	12	300	ghi
3	13	400	xyz

Let's take a closer look at some of the built-in functions:

- `unique()`: The `unique` function returns unique values from a column or a row in the form of an array.
- `nunique()`: The `nunique` function provides the number of unique elements in a row or column.
- `value_counts()`: The `value_counts` function provides the number of elements present in a row or column.

Let's now take a look at the following code:

```
| In [5]:
```

```
| df['col2'].unique()  
|  
| Out[5]:  
| array([100, 200, 300, 400], dtype=int64)  
|  
| In [6]:  
| df['col2'].nunique()  
|  
| Out[6]:  
| 4  
|  
| In [7]:  
| df['col2'].value_counts()
```

The output is as follows:

```
| Out[7]:  
| 100    1  
| 400    1  
| 300    1  
| 200    1  
| Name: col2, dtype: int64
```

A DataFrame also provides an option to apply an operation within a function to all the elements in the DataFrame. Let's define a function and see how we can apply it to a DataFrame:

```
| In [20]:  
| def squareofanumber(value):  
|     return value**2  
|  
| In [21]:  
| df['col1']  
|  
| Out[21]:  
| 0    10  
| 1    11  
| 2    12  
| 3    13  
| Name: col1, dtype: int64  
|  
| In [22]:  
| #apply() function applies the function definition in all the elements  
| from a column  
| df['col1'].apply(squareofanumber)  
|  
| Out[22]:  
| 0    100  
| 1    121  
| 2    144  
| 3    169  
| Name: col1, dtype: int64
```

The `apply()` function applies the function to all of the elements in the DataFrame.



Make sure the function definition is compatible with the values of the DataFrame with respect to the datatypes used, or it may produce a runtime error.

Data input and output operations with pandas

The pandas library provides a lot of built-in functions to read data from different data sources, such as CSV, Excel, JSON, and HTML. These features have made pandas the favorite library of many data scientists and machine learning developers.

Reading CSV files

pandas provides a built-in function, `read_csv()`, to read data from a CSV data source. The output returned from this function is essentially a DataFrame. First, we need to import the pandas library, as follows:

```
In [15]:  
import pandas as pd
```

The function used to read the CSV file is `read_csv()`:

```
In [16]:  
dataframe = pd.read_csv('Customers.csv')  
  
In [17]:  
dataframe.head()
```

The output is as follows:

	CustomerID	Gender	Age	Annual Income (k\$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

We can convert this DataFrame back into the CSV file using the `to_csv()` function:

```
In [18]:  
type(dataframe)  
  
Out[18]:  
pandas.core.frame.DataFrame  
  
In [19]:  
dataframe.to_csv('customer_copy.csv', index=False)
```



The CSV file is saved in the same location as that of the current file. It can be downloaded



from the GitHub repository here: <https://goo.gl/DJxn9x>.

Reading Excel files

pandas also has a built-in `read_excel()` function, which is used to read data from Excel files. The output that is returned is also a DataFrame. The parameter that is usually provided in the `read_excel()` function is the name of the Excel file and the sheet name:

```
In [3]:  
import pandas as pd  
  
In [9]:  
df=pd.read_excel('Sample.xlsx',sheet_name='Sample')  
df.head()
```

The output for the preceding code is as follows:

Out[9]:

	Row ID	Order Priority	Discount	Unit Price	Shipping Cost	Customer ID	Customer Name	Ship Mode	Customer Segment	Product Category	...	Region	State or Province
0	20847	High	0.01	2.84	0.93	3	Bonnie Potter	Express Air	Corporate	Office Supplies	...	West	Washington
1	20228	Not Specified	0.02	500.98	26.00	5	Ronnie Proctor	Delivery Truck	Home Office	Furniture	...	West	California
2	21776	Critical	0.06	9.48	7.29	11	Marcus Dunlap	Regular Air	Home Office	Furniture	...	East	New Jersey
3	24844	Medium	0.09	78.69	19.99	14	Gwendolyn F Tyson	Regular Air	Small Business	Furniture	...	Central	Minnesota
4	24846	Medium	0.08	3.28	2.31	14	Gwendolyn F Tyson	Regular Air	Small Business	Office Supplies	...	Central	Minnesota

5 rows × 25 columns

The `to_excel()` function is used to convert the DataFrame back into the Excel file. It is stored in the same location that we are currently working in:

```
In [10]:
```

```
| df.to_excel('Excel_Sample.xlsx', sheet_name='Sheet1')
```



The sample Excel file can be found at the GitHub link mentioned in the Technical requirements section.

Reading from HTML files

pandas also provides a built-in function to read data from HTML files. This function reads the table tabs from the HTML and returns a DataFrame object; an example of this is as follows:

```
In [12]:  
df = pd.read_html('http://www.fdic.gov/bank/individual/failed/banklist.html')  
  
In [14]:  
df[0].head()
```

The output for the preceding code is as follows:

Bank Name	City	ST	CERT	Acquiring Institution	Closing Date	Updated Date
0 Washington Federal Bank for Savings	Chicago	IL	30570	Royal Savings Bank	December 15, 2017	February 21, 2018
1 The Farmers and Merchants State Bank of Argonia	Argonia	KS	17719	Conway Bank	October 13, 2017	February 21, 2018
2 Fayette County Bank	Saint Elmo	IL	1802	United Fidelity Bank, fsb	May 26, 2017	July 26, 2017
3 Guaranty Bank, (d/b/a BestBank in Georgia & Mi...)	Milwaukee	WI	30003	First-Citizens Bank & Trust Company	May 5, 2017	March 22, 2018
4 First NBC Bank	New Orleans	LA	58302	Whitney Bank	April 28, 2017	December 5, 2017



In this example, the `read_html()` function is used to read from the HTML URL. It returns a DataFrame object.

Matplotlib

Matplotlib is a very advanced data visualization library that is used with Python. It was created by John Hunter; he created it to try to replicate the plotting capabilities of MATLAB (another programming language) in Python. If you happen to be familiar with MATLAB, matplotlib will feel natural to you. It is an excellent two-dimensional and three-dimensional graphics library for generating scientific figures.

The major advantages of matplotlib include the following:

- It is generally very easy to get started for simple and complex plots.
- It provides features that support custom labels and texts.
- It provides great control of each and every element in a plotted figure.
- It provides high-quality output with many image formats.
- It provides lots of customizable options.

Matplotlib allows you to create reproducible figures programmatically. So, let's learn how to use it! Before continuing with this chapter, I encourage you to explore the official matplotlib web page here: <http://matplotlib.org/>.

In order to use the matplotlib library, we will need to install matplotlib by going to the Anaconda prompt, the Command Prompt, or the Terminal and typing in the following command:

```
| conda install matplotlib
```

Alternatively, we can use the following command:

```
| pip install matplotlib
```

Let's now move on and take a look at some examples of how to use matplotlib to visualize data. The first step is to import the library, as follows:

```
| In [1]:  
|     import matplotlib.pyplot as plt
```

You'll also need to use the following line to see plots in the Notebook:

```
| In [2]:  
| %matplotlib inline
```

We are going to use some NumPy examples to create some arrays and learn how to plot them using the matplotlib library. Let's create an array using the `numpy.linspace` function, as follows:

```
| In [6]:  
| import numpy as np  
| import numpy as np  
| arr1= np.linspace(0, 10, 11)  
| arr2 = arr1 ** 2  
  
| In [7]:  
| arr1  
  
| Out[7]:  
| array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])  
  
| In [8]:  
| arr2  
  
| Out[8]:  
| array([ 0.,  1.,  4.,  9., 16., 25., 36., 49., 64., 81., 100.])
```

Here, two arrays, `arr1` and `arr2`, have been created using NumPy. Let's now take a look at some matplotlib commands to create a visualization graph.

The plot function

The `plot` function is used for plotting y versus x in the form of points, markers, or lines. The syntax of the `plot` function is as follows:

In []: `plt.plot()`

In [20]: Signature: `plt.plot(*args, **kwargs)`
Docstring:
Plot y versus x as lines and/or markers.
Call signatures::

`plot([x], y, [fmt], data=None, **kwargs)`
`plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)`

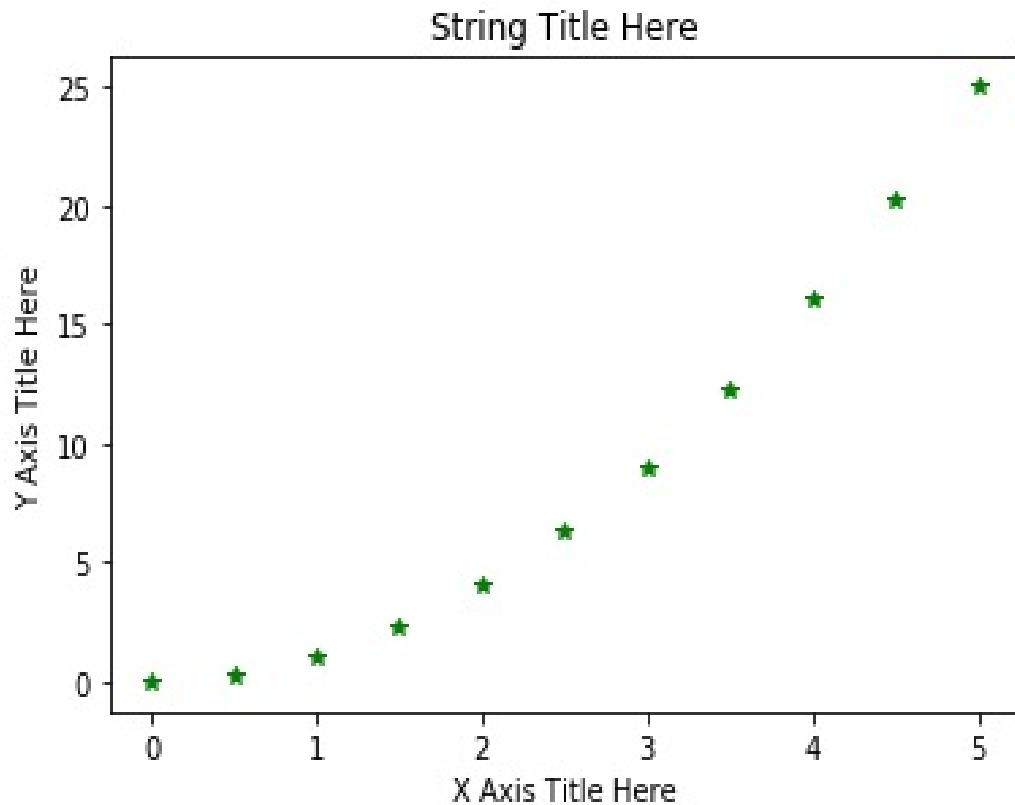
The coordinates of the points or line nodes are given by *x*, *y*.

Usually, the first and second parameters are the arrays that we need to plot. There is a list of parameters that can be given to the `plot` function, such as the line color and the pixel size of the line. We can create a very simple line plot using the following code:

```
In [9]:  
# 'g' is the color green  
plt.plot(a,b, 'g*',label="Example 2")  
plt.xlabel('X Axis Title Here')  
plt.ylabel('Y Axis Title Here')  
plt.title('String Title Here')
```

The output is as follows:

```
Out[9]: Text(0.5,1,'String Title Here')
```



The third parameter inside the `plot` function is essentially the color parameter, which is used to provide a color for the plotted points.



To find out more about the different parameters and features inside the various built-in functions provided by matplotlib, I encourage you to explore the official matplotlib web page here: <http://matplotlib.org/>.

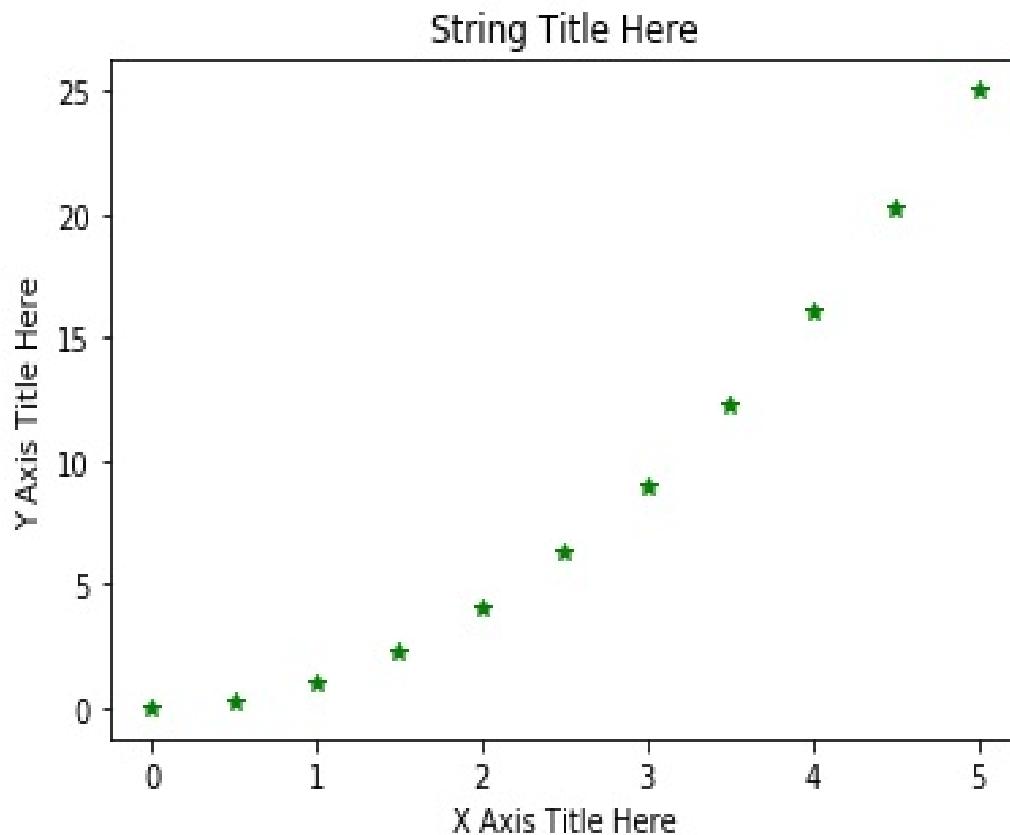
The xlabel function

In the preceding example, we used the `xlabel` built-in function to provide a label to the `x` axis. This helps you to provide your own custom label values for the `x` axis. We can create a very simple line plot using the following code:

```
In [9]:  
# 'g' is the color green  
plt.plot(a,b, 'g*',label="Example 2")  
plt.xlabel('X Axis Title Here')  
plt.ylabel('Y Axis Title Here')  
plt.title('String Title Here')
```

The output is shown here:

Out[9]: `Text(0.5,1,'String Title Here')`



In the preceding example, we provided the `xlabel` value as `X Axis Title Here`.

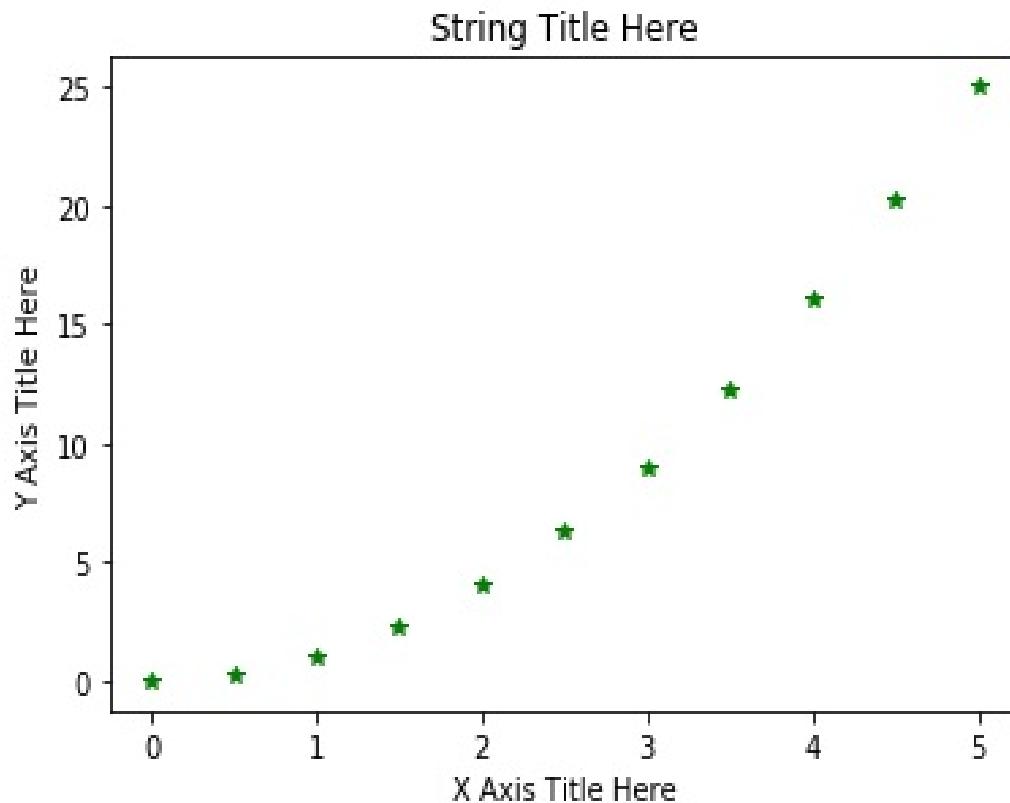
The ylabel function

Similarly, there is another built-in function, `plt.ylabel()`, which we can use to provide a custom label for the *y* axis. We can create a very simple line plot using the following code:

```
In [9]:  
# 'g' is the color green  
plt.plot(a,b, 'g*',label="Example 2")  
plt.xlabel('X Axis Title Here')  
plt.ylabel('Y Axis Title Here')  
plt.title('String Title Here')
```

The output is shown here:

Out[9]: `Text(0.5,1,'String Title Here')`



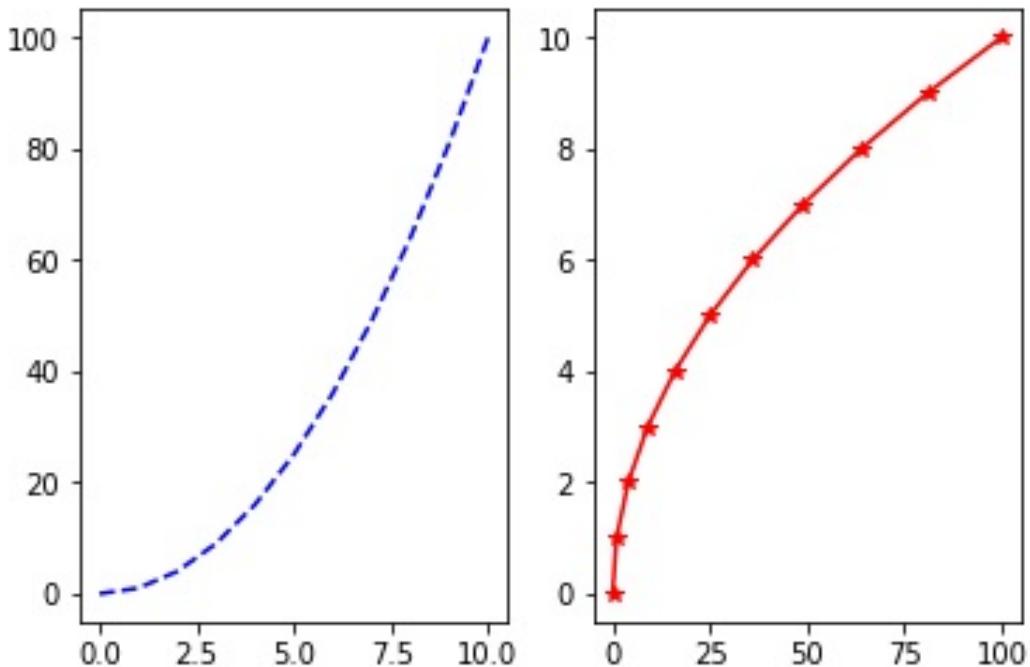
In the preceding example, we provided the value of `ylabel` as `Y Axis Title Here`.

Creating multiple subplots using matplotlib

We can also create multiple subplots using the `subplot` function, which is provided in the `matplotlib` library; let's take a look at an example:

```
In [14]:  
import numpy as np  
a= np.linspace(0, 10, 11)  
b = a**2  
  
In [15]:  
# plt.subplot(nrows, ncols, plot_number)  
plt.subplot(1,2,1)  
plt.plot(a, b, 'b--') # More on color options later  
plt.subplot(1,2,2)  
plt.plot(b, a, 'r*-');
```

The output is as follows:



The `matplotlib` library also provides various advanced methods, such as `histogram` and `pie`, which can provide better visualization of data.

The pie function

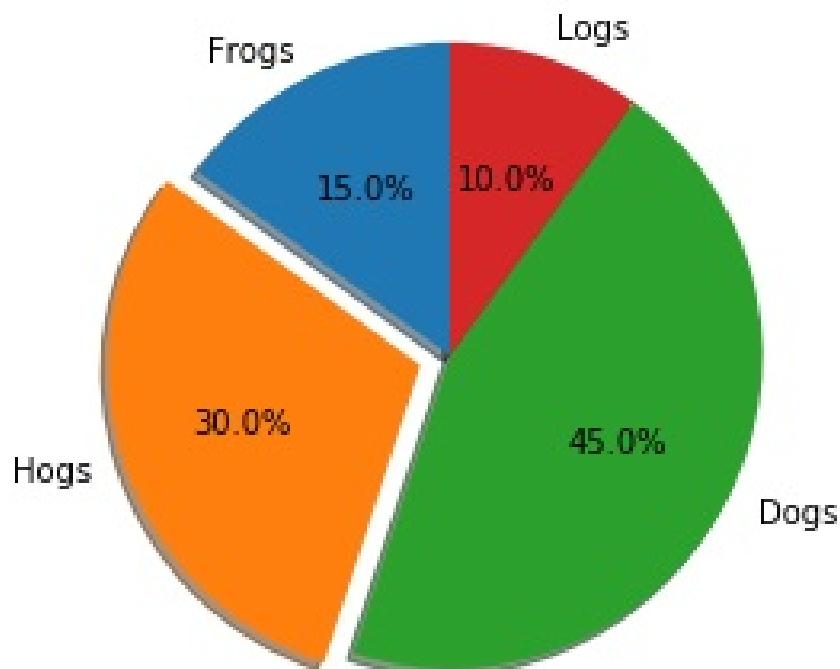
The `pie` function helps to categorize a group of entities so that they can be represented visually. In short, a `pie` function helps to plot a pie chart; the syntax is as follows:

```
|matplotlib.pyplot.pie(x, explode=None, labels=None, colors=None, autopct=None, pctdistar
```

Let's take a look at an example through the following code:

```
In [16]:  
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'  
sizes = [15, 30, 45, 10]  
explode = (0, 0.1, 0, 0) # only "explode" the 2nd slice (i.e. 'Hogs')  
  
fig1, ax1 = plt.subplots()  
ax1.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',  
         shadow=True, startangle=90)  
ax1.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.  
  
plt.show()
```

The output is shown here:



The histogram function

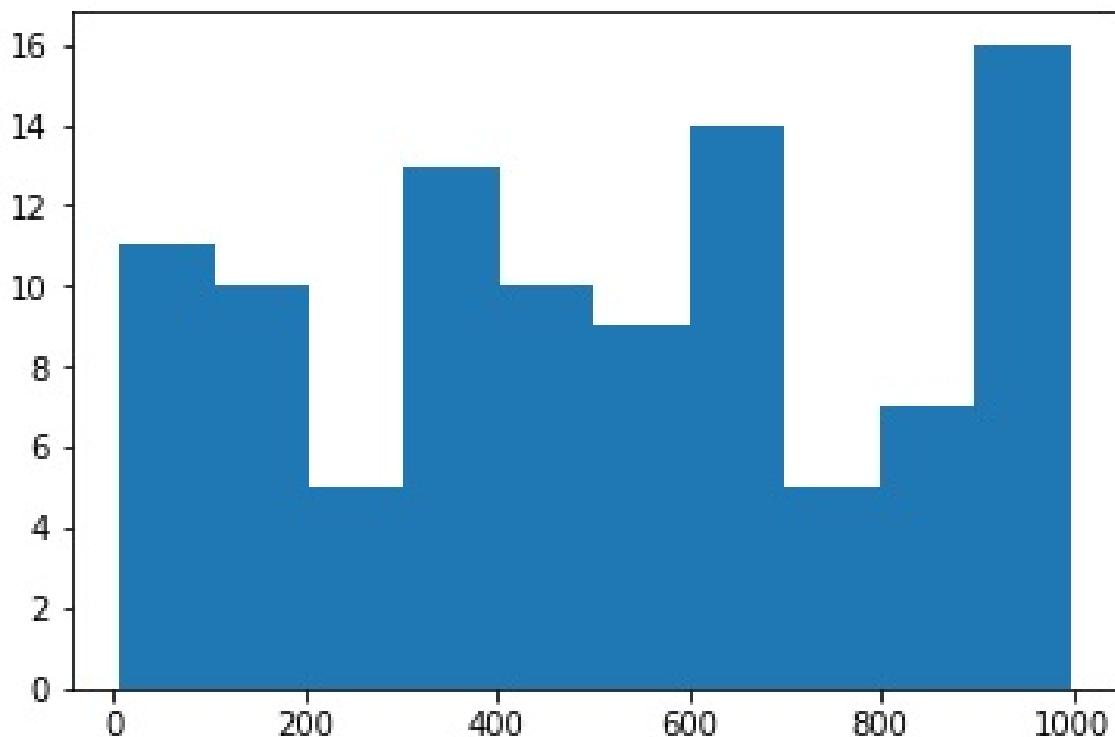
The `histogram` function in matplotlib plots histograms. The return value is a tuple, such as `n`, `bins`, and `patches`, or `[n0, n1, ...], bins, and [patches[0], patches[1],...]`, if the input contains multiple data values; let's take a look at the following example:

```
In [18]:  
from random import sample  
data = sample(range(1, 1000), 100)  
plt.hist(data)
```

The output is as follows:

```
Out[18]:  
(array([11., 10., 5., 13., 10., 9., 14., 5., 7., 16.]),  
 array([ 7., 105.9, 204.8, 303.7, 402.6, 501.5, 600.4, 699.3, 798.2,  
        897.1, 996.]),  
<a list of 10 Patch objects>)
```

The following graph demonstrates the output generated:





Please refer to the main documentation website (<https://matplotlib.org>) for more information about matplotlib.

Summary

In this chapter, we discussed the NumPy, pandas, and matplotlib libraries. We used the NumPy library to create multidimensional arrays. It provides various functions that we can apply to arrays to perform complex arithmetic tasks. We also learned how to retrieve elements from an array using indexing techniques for both single-dimensional and multidimensional arrays.

Then, we discussed the pandas library. In the pandas library, we looked at series and DataFrames, which provide various built-in functions to read data from data sources such as CSV files, Excel files, and HTML files. The pandas library provides us with a convenient way to carry out data preprocessing, which is the backbone of data analysis.

Finally, we explored the matplotlib library. Here, we found an efficient way of visualizing data, which is also an important part of data analysis and data preprocessing. Visualization helps us to understand our data more efficiently and to make better decisions.

All these libraries help us to carry out efficient data analysis, data preprocessing, and data visualization tasks, which represent the core of any problem statements that we solve using Python. In the next chapter, we will learn how to use these libraries to solve some real-world problems.

Further reading

It is always a good idea to read the official documentation of the libraries discussed to gain additional knowledge.

You can refer to the following links for the documentation of each library:

- **NumPy:** <https://docs.scipy.org/doc/>
- **pandas:** <https://pandas.pydata.org/pandas-docs/stable/>
- **Matplotlib:** <https://matplotlib.org>

Section 2: Advanced Analysis in Python for Finance

In this section, we will deep dive into various topics, such as time series financial data, portfolio optimization, capital asset pricing model, and regression analysis. In these topics, we will see how to use Python and the tools available in Python to implement these techniques.

This section consists of the following chapters:

[Chapter 3](#), *Time Series Analysis and Forecasting*

[Chapter 4](#), *Measuring Investment Risks*

[Chapter 5](#), *Portfolio Allocation and Markowitz Portfolio Optimization*

[Chapter 6](#), *The Capital Asset Pricing Model*

[Chapter 7](#), *Regression Analysis in Finance*

Time Series Analysis and Forecasting

In the previous chapter, we discussed various important libraries, such as NumPy, pandas, and matplotlib. We also looked at a few specific code examples. In this chapter, we are going to discuss time series analysis and forecasting. Time series data is data that is usually linked to a fixed interval of time, such as the sales data of a company, or stock prices.

Time series data is usually represented using line charts, which allow us to understand the patterns of the data that's been plotted. Time series data is used in various fields, including statistics, control engineering, astronomy, communications engineering, econometrics, mathematical finance, weather forecasting, earthquake prediction, electroencephalography, signal processing, pattern recognition, and in any applied science or engineering domain.

The following topics will be covered in this chapter:

- pandas with time series data
- Introducing the StatsModels library
- Introducing errors, trends, and seasonality with code
- The **AutoRegressive Integrated Moving Average (ARIMA)** model for time series forecasting

Let's get started.

Technical requirements

In this chapter, we will be using the Jupyter Notebook for coding purposes. We will be using the pandas library and the StatsModels library for time series analysis and time series forecasting. The GitHub repository that contains the code for this chapter, is available at: <https://github.com/PacktPublishing/Hands-on-Python-for-Finance/tree/master/Chapter3>.

pandas with time series data

After having read the previous chapter, we now have a better understanding of how to use pandas with various datasets. Let's now see how we can use pandas with time series data.

Time series analysis refers to the methods that are used for carefully studying time series data in order to extract meaningful statistical information and other important features.

Time series forecasting is a technique that's used to predict future outcomes based on previously observed time series data.



In this chapter, all the datasets we will be working on will be in the form of time series, with a date time index and a corresponding value. We will be learning about pandas time series features so that we can work with this kind of data.

In this section, we are going to cover the following topics:

- Date time index
- Time resampling
- Timeshifts
- Rolling and expanding

Date time index

In this section, we will initially be discussing Python's date time inbuilt function. We will then move on to discussing the various pandas inbuilt functions for creating and working with the date time index. Let's take a look at some examples.

To begin with, let's import all the necessary libraries—NumPy, pandas, and matplotlib—as shown in the following code segment:

```
In [1]:  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt
```

Now, let's import the `datetime` library. This library is the Python inbuilt library that allows us to create a date time stamp or a specific date object. The `datetime` library is imported as follows:

```
In [2]:  
from datetime import datetime
```

Let's take a look at how we can create a date time object. Initially, we will create variables for the year, the month, the day, and the time:

```
In [3]:  
year = 2018  
month = 9  
day = 24  
hour = 12  
minute = 30  
second = 15
```

We then convert this into a date time function, as shown here:

```
In [4]:  
date = datetime(year,month,day)  
date  
  
Out[5]:  
datetime.datetime(2018, 9, 24, 0, 0)
```

The variables for the year, the month, the day, and the time have been created to illustrate the order of arguments that are used inside the date time function. Since

we have not provided a parameter for the hours or the minutes inside the `datetime` function, the default value is set to 0. If you want to provide the hour, the minute, and the second as well, we can do this as follows:

```
In [8]:  
date_time = datetime(year,month,day,hour,minute,second)  
date_time  
  
Out[8]:  
datetime.datetime(2018, 9, 24, 12, 30, 15)
```

We can now see the values for the hours, the minutes, and the seconds.

The date time object has various attributes that help us retrieve information from the date time variables, as shown in the following code segment. The following code displays the day of the date time object:

```
In [12]:  
date_time.day  
  
Out[12]:  
24
```

The following code displays the hour of the date time variable:

```
In [14]:  
date_time.hour  
  
Out[14]:  
12
```

The following code displays the minute value from the date time variable:

```
In [15]:  
date_time.minute  
  
Out[15]:  
30
```

The following code displays the year from the date time variable:

```
In [16]:  
date_time.year  
  
Out[16]:  
2018
```

The preceding examples use various attributes, such as the day, the hour, the minute, or the year to provide information about the date time variable.



If you want more information about the various inbuilt properties, please go to the Python official documentation provided at the following link: <https://docs.python.org/2/library/datetime.html>.

Up until now, we have been looking at the basic date time object that is provided by Python. We will now go ahead and discuss how we can use the pandas library to handle the date time object and convert it into a date time index. Usually, in a real-world scenario, most financial data will have a date time object in its dataset. This data can be retrieved through an API or a CSV file, which can be read or retrieved with the help of pandas. The syntax for the pandas date time index is as follows:

```
| pandas.DateTimeIndex( data, copy, start, end)
```

The following table provides a description of the parameters:

Serial no.	Parameter and its description
1	<ul style="list-style-type: none">• <code>data</code>: This is a one-dimensional array and is optional. Date time data is used for creating the indexes.
2	<ul style="list-style-type: none">• <code>copy</code>: A Boolean value. This allows us to create a copy of the input data.
3	<ul style="list-style-type: none">• <code>freq</code>: This parameter should be a pandas offset or strings and is optional.
4	<ul style="list-style-type: none">• <code>start</code>: This parameter is a date that is a date time object and is optional. If the data is <code>None</code>, <code>start</code> is used as the start point to generate regular timestamp data.
5	<ul style="list-style-type: none">• <code>end</code>: This parameter is a date that is a date time object and is optional. If <code>periods</code> is <code>None</code>, the generated index will extend to the first matching time, on or just past the <code>end</code> argument.

6

- `closed`: This parameter is a string and the default is `None`. This makes the interval closed with respect to the given frequency to the left, right, or both sides (`None`).

We usually deal with time series as an index when working with a pandas dataframe that's been obtained from some sort of financial API. Fortunately, pandas has a lot of functions and methods for working with time series data. Let's take a look at an example of how we can convert a Python date time object into a pandas date time index.

First, we will create some date time variables using the Python `datetime` library, as shown here:

```
In [28]:  
# Create an example datetime list  
list_date = [datetime(2018, 9, 24), datetime(2016, 9, 25)]  
list_date  
  
Out[28]:  
[datetime.datetime(2018, 9, 24, 0, 0), datetime.datetime(2016, 9, 25, 0, 0)]
```

The next step is to use a pandas date time index to convert the preceding data into a date time index, as shown here:

```
In [29]:  
# Converted to a Date time index  
dt_date = pd.DatetimeIndex(list_date)  
dt_date  
  
Out[29]:  
DatetimeIndex(['2018-09-24', '2016-09-25'], dtype='datetime64[ns]', freq=None)
```

Here, we can see that the return type of `list_date` was `datetime.datetime`. In the next line, it is converted into a date time index. Now, let's create a dataframe with the date time index:

```
In [30]:  
# Considering some random data  
data = np.random.randn(2,2)  
print(data)  
cols = ['X', 'Y']
```

```
| Out[30]:  
| [[-0.0747333  0.21118015]  
| [-0.5697708 -0.36354259]]
```

Then, we will create a dataframe using the pandas library, as shown here:

```
| In [31]:  
| df = pd.DataFrame(data, dt_date, cols)  
| df
```

The output is as follows:

	X	Y
2018-09-24	-0.074733	0.211180
2016-09-25	-0.569771	-0.363543

In the preceding code, we created a date time index and provided it as a parameter in the pandas dataframe as an index. We then created a dataframe with the indexes of the date time. This is the format in which we will see a lot of financial data in our dataset.

Time resampling

In this section, we will be discussing time resampling. We usually get financial data that has a date time index on a smaller time scale (such as data that is sampled every day or every hour). However, it is often a good idea to aggregate the data based on a particular frequency, such as every month, every quarter, or every year.

Another approach would be to use `groupby`, but the group operation is not smart enough to understand concepts such as business quarters or the start of a business year. Fortunately, pandas has other inbuilt functions that help us implement frequency sampling. To get started with time resampling, we will be considering the stock prices of the Walmart dataset. You can find the CSV in the chapter 3 folder of the GitHub link we mentioned previously. This dataset is taken from Yahoo Finance.

To begin with, let's import all the necessary libraries, such as `numpy`, `pandas`, and `matplotlib`, as shown in the following code segment:

```
In [2]:  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
%matplotlib inline
```

Now, let's read the Walmart CSV dataset using the `pandas read_csv` function, as shown here:

```
In [3]:  
df = pd.read_csv('walmart_stock.csv')  
In [4]:  
df.head()
```

The output is as follows:

	Date	Open	High	Low	Close	Volume	/
0	2012-01-03	59.970001	61.060001	59.869999	60.330002	12668800	5
	2012-						

1	01-04	60.209999	60.349998	59.470001	59.709999	9593300	5
2	2012-01-05	59.349998	59.619999	58.369999	59.419998	12768200	5
3	2012-01-06	59.419998	59.450001	58.869999	59.000000	8069400	5
4	2012-01-09	59.029999	59.549999	58.919998	59.180000	6679300	5

From the preceding output, we can see that the first column is the date column, which has to be set as an index. If you take a look at the type of the `Date` column, we can see that it is of the object type.

We can see the datatypes of all the columns using the following code:

```
In [5]:
df.info()

Out[5]:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1258 entries, 0 to 1257
Data columns (total 7 columns):
Date      1258 non-null object
Open       1258 non-null float64
High       1258 non-null float64
Low        1258 non-null float64
Close      1258 non-null float64
Volume     1258 non-null int64
Adj Close   1258 non-null float64
dtypes: float64(5), int64(1), object(1)
memory usage: 68.9+ KB
```

From the preceding output, the `Date` column is not a date time object. We need to convert the `Date` column into a date time object in order to set it as an index. The simplest way to do this is by using the pandas `to_datetime` function, as shown in the following code segment:

```
In [6]:
df['Date'] = df['Date'].apply(pd.to_datetime)

In [7]:
df.info()

Out[7]:
<class 'pandas.core.frame.DataFrame'>
```

```

RangeIndex: 1258 entries, 0 to 1257
Data columns (total 7 columns):
Date      1258 non-null datetime64[ns]
Open       1258 non-null float64
High       1258 non-null float64
Low        1258 non-null float64
Close      1258 non-null float64
Volume     1258 non-null int64
Adj Close   1258 non-null float64
dtypes: datetime64[ns](1), float64(5), int64(1)
memory usage: 68.9 KB

```

Let's check the first five rows of the dataframe:

```
In [8]:
df.head()
```

The output is as follows:

	Date	Open	High	Low	Close	Volume	Adj Close
0	2012-01-03	59.970001	61.060001	59.869999	60.330002	12668800	59.970001
1	2012-01-04	60.209999	60.349998	59.470001	59.709999	9593300	60.209999
2	2012-01-05	59.349998	59.619999	58.369999	59.419998	12768200	59.349998
3	2012-01-06	59.419998	59.450001	58.869999	59.000000	8069400	59.419998
4	2012-01-09	59.029999	59.549999	58.919998	59.180000	6679300	59.029999

Once we convert it into a date time object, we can see the type of the `Date` column: it is a date time object. We can now set the date time as an index by using the `set_index` pandas inbuilt function, as shown in the following code segment:

```
In [9]:
df.set_index('Date', inplace=True)
df.head()
```

The output is as follows:

Date	Open	High	Low	Close	Volume	Adj Close
------	------	------	-----	-------	--------	-----------

2012-01-03	59.970001	61.060001	59.869999	60.330002	12668800	52.61
2012-01-04	60.209999	60.349998	59.470001	59.709999	9593300	52.07
2012-01-05	59.349998	59.619999	58.369999	59.419998	12768200	51.82
2012-01-06	59.419998	59.450001	58.869999	59.000000	8069400	51.45
2012-01-09	59.029999	59.549999	58.919998	59.180000	6679300	51.61

Resampling is usually applied to a date time index, which was the reason why we converted the Date column into a date time index. Let's proceed and see how we can apply the date time index.

The resampling syntax is usually as follows:

```
| DataFrame.resample(rule, how=None, axis=0, fill_method=None, closed=None, label=None, cc
```

The following are the important parameters that are used in the `resample()` method:

<ul style="list-style-type: none"> • <code>rule</code> <p>This is the offset string or object representing the target conversion.</p> <ul style="list-style-type: none"> • <code>axis</code> <p>This can have values that are either <code>0</code> or <code>1</code> and the default value is <code>0</code>.</p> <ul style="list-style-type: none"> • <code>closed: {'right', 'left'}</code> <p>This indicates which side of the bin interval is closed. The default is <code>left</code> for all frequency offsets</p>
--

Parameters:

except for `M`, `A`, `Q`, `BM`, `BA`, `BQ`, and `W`, which all have a default of right. These letters refer to different frequencies, more details of which can be found in the list following this table.

- `label: {'right', 'left'}`

This indicates which bin edge label should be used to label the bucket with. The default is "left" for all frequency offsets except for `M`, `A`, `Q`, `BM`, `BA`, `BQ`, and `W`, which all have a default of right.

- `convention: {'start', 'end', 's', 'e'}`

This is used for `PeriodIndex` only. It controls whether to use the start or end of the rule. The `PeriodIndex` helps us to display the dataframe with respect to the starting month of a year or the end month of a year.

- `kind: {'timestamp', 'period'}`, optional

We can pass the timestamp to convert the resulting index to a date time index or period to convert it to a period index. By default, the input representation is retained.

- `loffset : timedelta`

This adjusts the resampled time labels.

- `base : int`, default 0

This refers to the origin of the aggregated intervals for frequencies that evenly subdivide a day. For example, for a frequency of five minutes, the base could range from zero to four. The default value is zero.

- `on : string`, optional

	<p>This indicates which column to use for resampling instead of the index for a dataframe. A column must be a date time object. This is new in version 0.19.0.</p> <ul style="list-style-type: none"> • <code>level: string or int, optional</code> <p>For a multiindex, this indicates which column to use for resampling instead of the index. The level must be a date time object. It is new in version 0.19.0.</p>
Returns:	Resampler object.

The first parameter in the resample function is `rule`. The `rule` parameter is used to indicate how we want to resample the date time index. It basically applies a `groupby` method that is specific to time series data. The following are the various options that can be provided for the rule:

Rule	Description
B	Business day frequency
D	Calendar day frequency
W	Weekly frequency
M	Month end frequency
SM	Semi-month end frequency (15 th and end of month)
BM	Business month end frequency
CBM	Custom business month end frequency
MS	Month start frequency
SMS	Semi-month start frequency (1st and 15th)
BMS	Business month start frequency
CBMS	Custom business month start frequency

Q	Quarter end frequency
BQ	Business quarter end frequency
A	Year end frequency
BA	Business year end frequency
AS	Year start frequency
BAS	Business year start frequency
BH	Business hour frequency
H	Hourly frequency
T, min	Minutely frequency
S	Secondly frequency
L, ms	Milliseconds
U, us	Microseconds
N	Nanoseconds



You can read the pandas documentation for more information on the rules: <http://pandas.pydata.org/pandas-docs/stable/timeseries.html>.

Let's apply a new resampling rule to the previously discussed dataset. In the following example, we will apply rule A, which specifies year end frequency, to resample our data, as shown here:

```
In [10]: df.resample(rule='A')
Out[10]: DatetimeIndexResampler [freq=<YearEnd: month=12>, axis=0, closed=right, label=right, con
```

The return type of the resample method is `DatetimeIndexResample`. After applying the resample method, we should apply an aggregate function to group the data. In the following example, we will apply a mean function to group the resampled data:

```
In [11]: # To find the yearly mean
df.resample(rule='A').mean()
```

The output is as follows:

Date	Open	High	Low	Close	Volume	Adj Close
2012-12-31	67.158680	67.602120	66.786520	67.215120	9.239015e+06	67.158680
2013-12-31	75.264048	75.729405	74.843055	75.320516	6.951496e+06	75.264048
2014-12-31	77.274524	77.740040	76.864405	77.327381	6.515612e+06	77.274524
2015-12-31	72.569405	73.064167	72.034802	72.491111	9.040769e+06	72.569405
2016-12-31	69.481349	70.019643	69.023492	69.547063	9.371645e+06	69.481349

From the preceding example, after applying rule `A` and the aggregate function `mean`, we get the result of the yearly mean, or the average values of `Open`, `High`, `Low`, `close`, and `volume` of the Walmart stock prices.

Similarly, we can apply various other rules, such as **weekly frequency (W)**, **calendar day frequency (D)**, and **business month end frequency (BM)**, as shown in the following code segment:

```
In [13]:
# Weekly frequency Means
df.resample(rule='W').mean().head()
```

The output is as follows:

Date	Open	High	Low	Close	Volume	Adj Close
2012-01-08	59.737499	60.120	59.1450	59.615000	10774925.0	51.99561
2012-01-15	59.298000	59.680	59.0700	59.332001	6983580.0	51.74878
2012-01-22	60.085000	60.530	59.8975	60.369999	8506200.0	52.65412
2012-01-29	61.080000	61.510	60.7220	61.090000	6827240.0	53.28209

2012-02-05	61.702001	62.084	61.2480	61.762000	8693500.0	53.86820
-------------------	-----------	--------	---------	-----------	-----------	----------

The following code provides us with the calendar day frequency means:

```
In [14]:
# Calendar day frequency Means
df.resample(rule='D').mean().head()
```

The output is as follows:

Date	Open	High	Low	Close	Volume	Ad.
2012-01-03	59.970001	61.060001	59.869999	60.330002	12668800.0	52.0
2012-01-04	60.209999	60.349998	59.470001	59.709999	9593300.0	52.0
2012-01-05	59.349998	59.619999	58.369999	59.419998	12768200.0	51.8
2012-01-06	59.419998	59.450001	58.869999	59.000000	8069400.0	51.4
2012-01-07	NaN	NaN	NaN	NaN	NaN	NaN

The following code provides us with the business month end frequency means:

```
In [15]:
# Business month end frequency Means
df.resample(rule='BM').mean().head()
```

The output is as follows:

Date	Open	High	Low	Close	Volume	Ad.
2012-01-31	60.159000	60.572000	59.803000	60.235500	8.178850e+06	52.0
2012-02-29	60.935500	61.224000	60.582500	60.898000	9.965725e+06	52.0
2012-						

03-30	60.309546	60.642273	60.101818	60.433637	8.446464e+06	E
2012-04-30	60.073000	60.553000	59.727000	60.149000	1.258940e+07	E
2012-05-31	61.173182	61.771819	60.985455	61.456363	1.123173e+07	E

In the 14th line of code, where we applied the calendar day frequency rule, we can see that one row has NaN values. This is because the stock prices are not available for Saturdays and Sundays.

We can also use various aggregate functions, such as `max()`, `min()`, or `std()`, as shown in the following code segment:

```
| In [16]:  
| yearly frequency max  
| df.resample(rule='A').max().head()
```

The output is as follows:

	Open	High	Low	Close	Volume	Adj Close
Date						
2012-12-31	77.599998	77.599998	76.690002	77.150002	38007300	68.56
2013-12-31	81.209999	81.370003	80.820000	81.209999	25683700	73.92
2014-12-31	87.080002	88.089996	86.480003	87.540001	22812400	81.70
2015-12-31	90.800003	90.970001	89.250000	90.470001	80898100	84.91
2016-12-31	74.500000	75.190002	73.629997	74.300003	35076700	73.23

The following code shows how we can apply the `min()` aggregate function:

```
| In [17]:  
| #yearly frequency min
```

```
| df.resample(rule='A').min().head()
```

The output is as follows:

	Open	High	Low	Close	Volume	Adj C
Date						
2012-12-31	57.590000	58.430000	57.180000	57.360001	2904800	50.363
2013-12-31	68.190002	68.669998	67.720001	68.300003	2094900	61.039
2014-12-31	72.269997	73.099998	72.269997	72.660004	2491800	66.531
2015-12-31	56.389999	57.060001	56.299999	56.419998	2482800	53.975
2016-12-31	60.500000	61.490002	60.200001	60.840000	4234400	58.691

The following code help us get the standard deviation with yearly frequency:

```
In [18]:  
#yearly frequency standard deviation  
df.resample(rule='A').std().head()
```

The output is as follows:

	Open	High	Low	Close	Volume	Adj C
Date						
2012-12-31	6.089182	6.108822	6.017926	6.043120	4.668863e+06	5.697
2013-12-31	3.204078	3.181319	3.190823	3.189005	2.784862e+06	3.204
2014-12-31	3.314478	3.377809	3.275304	3.324339	2.605779e+06	3.411
2015-12-31	9.702563	9.730657	9.674375	9.724255	6.314777e+06	8.707
2016-12-31	3.033227	2.884394	3.088522	2.955446	4.322518e+06	3.303

Timeshifts

Usually, when we are predicting prices or stocks, certain models or algorithms that we use require us to move our data forward or backward a certain amount of time steps. Fortunately, pandas provides us with a lot of inbuilt tools to implement this functionality. This movement with respect to time steps is called timeshifts.

Let's go ahead and see how we can use timeshifts with the help of pandas. In this section, we will be reading the same Walmart stock prices dataset. We'll start by importing the libraries, as shown in the following code segment:

```
In [5]:  
import pandas as pd  
  
import matplotlib.pyplot as plt  
%matplotlib inline
```

Let's read the `walmart_stock.csv` file, as shown in the following code segment:

```
In [6]:  
dataframe = pd.read_csv('walmart_stock.csv', index_col='Date')  
dataframe.index = pd.to_datetime(dataframe.index)
```

We can use the `head()` function to see the first five records of the dataframe, as shown in the following code segment:

```
In [7]:  
dataframe.head()
```

The output is as follows:

	Open	High	Low	Close	Volume	Adj Close
Date						
2012-01-03	59.970001	61.060001	59.869999	60.330002	12668800	52.61
2012-01-04	60.209999	60.349998	59.470001	59.709999	9593300	52.07
2012-01-05	59.349998	59.619999	58.369999	59.419998	12768200	51.82

2012-01-06	59.419998	59.450001	58.869999	59.000000	8069400	51.45
2012-01-09	59.029999	59.549999	58.919998	59.180000	6679300	51.61

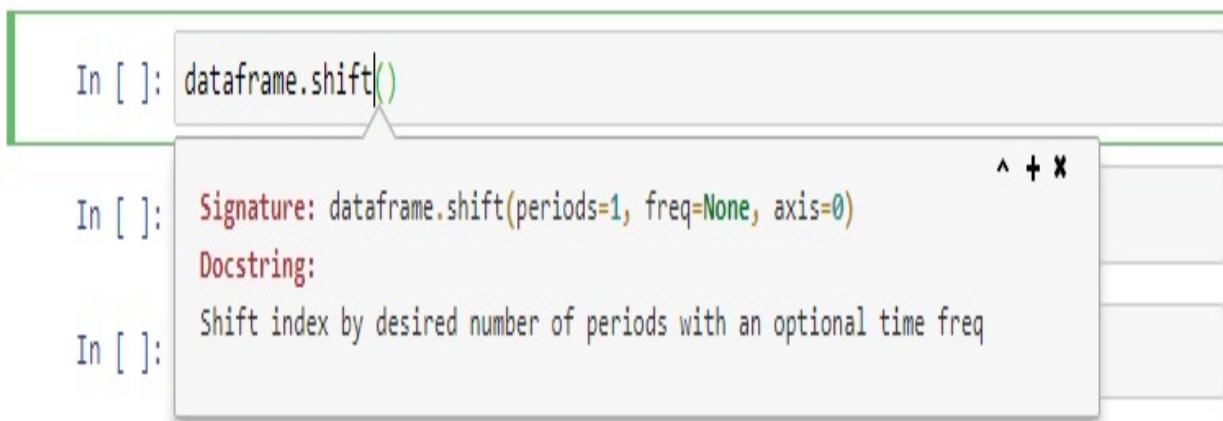
To see the last five records of the dataframe, we will use the `tail()` function, as shown in the following code segment:

```
| In [8]:  
| dataframe.tail()
```

The output is as follows:

	Open	High	Low	Close	Volume	Adj C
Date						
2016-12-23	69.430000	69.750000	69.360001	69.540001	4803900	69.032
2016-12-27	69.300003	69.820000	69.250000	69.699997	4435700	69.191
2016-12-28	69.940002	70.000000	69.260002	69.309998	4875700	68.804
2016-12-29	69.209999	69.519997	69.120003	69.260002	4298400	68.754
2016-12-30	69.120003	69.430000	68.830002	69.120003	6889500	68.615

In the preceding code, the `head()` function displays the top five records from the dataset, whereas the `tail` function displays the last five records. Let's take a look at the basic syntax of the `shift` function:



The shift function helps us shift the index by a desired number of periods with an optional frequency. The following are the various parameters of the shift function:

Parameters	Descriptions
periods	This is usually an integer value. It specifies the number of periods or indexes to move. It can be positive or negative.
frequency	This is usually a date offset, time delta, or time rule. It is optional.
axis	The axis value is zero for the index and one for the column.

Let's implement the Timeshift operation using the `shift` method, as shown in the following code segment:

```
| In [11]:  
| dataframe.shift(periods=1).head()
```

The output is as follows:

	Open	High	Low	Close	Volume	Ad.
Date						
2012-01-03	NaN	NaN	NaN	NaN	NaN	Nal
2012-	59.970001	61.060001	59.869999	60.330002	12668800.0	52.0

01-04						
2012-01-05	60.209999	60.349998	59.470001	59.709999	9593300.0	52.0
2012-01-06	59.349998	59.619999	58.369999	59.419998	12768200.0	51.8
2012-01-09	59.419998	59.450001	58.869999	59.000000	8069400.0	51.4

In the preceding example, we provided the period value as one, so the dataframe is shifted downward by one time step. For this reason, the first row is made up of all NaN values.

Similarly, we can also provide a negative value to the `periods` parameter to shift the dataset upward. In this case, the last row will be appended as NaN:

```
| In [13]:  
| dataframe.shift(periods=-1).tail()
```

The output is as follows:

	Open	High	Low	Close	Volume	Adj
Date						
2016-12-23	69.300003	69.820000	69.250000	69.699997	4435700.0	69.1
2016-12-27	69.940002	70.000000	69.260002	69.309998	4875700.0	68.8
2016-12-28	69.209999	69.519997	69.120003	69.260002	4298400.0	68.7
2016-12-29	69.120003	69.430000	68.830002	69.120003	6889500.0	68.6
2016-12-30	NaN	NaN	NaN	NaN	NaN	NaN

The next parameter we will be discussing is frequency. Frequency is basically a date offset, time delta, or time rule. Let's consider an example to shift the dataset by a month and then by a year, as shown in the following code segment:

```
In [18]:
# Shift everything forward one month
dataframe.tshift(periods=1,freq='M').head()
```

The output is as follows:

	Open	High	Low	Close	Volume	Adj Close
Date						
2012-01-31	59.970001	61.060001	59.869999	60.330002	12668800	52.61
2012-01-31	60.209999	60.349998	59.470001	59.709999	9593300	52.07
2012-01-31	59.349998	59.619999	58.369999	59.419998	12768200	51.82
2012-01-31	59.419998	59.450001	58.869999	59.000000	8069400	51.45
2012-01-31	59.029999	59.549999	58.919998	59.180000	6679300	51.61

```
In [19]:
# Shift everything forward one year
dataframe.tshift(periods=1,freq='Y').head()
```

The output is as follows:

	Open	High	Low	Close	Volume	Adj Close
Date						
2012-12-31	59.970001	61.060001	59.869999	60.330002	12668800	52.61
2012-12-31	60.209999	60.349998	59.470001	59.709999	9593300	52.07
2012-12-31	59.349998	59.619999	58.369999	59.419998	12768200	51.82
2012-12-31	59.419998	59.450001	58.869999	59.000000	8069400	51.45
2012-12-31	59.029999	59.549999	58.919998	59.180000	6679300	51.61

The shift function will be handy when we implement the forecasting technique in later subsections.

Timeseries rolling and expanding using pandas

In this section, we will be discussing the built-in pandas rolling methods. We can use these methods to create a rolling mean based on a given time period. Let's discuss what a rolling method can be used for. Usually, the financial data that is generated daily is noisy. We can use the rolling mean, also known as the moving average, to get more information about the general trend of the data.

When using the pandas built-in rolling methods, we will provide a time period window and use the data within that to calculate various statistical measures, such as the mean, the standard deviation, and other mathematical concepts, including autoregression and the moving average. Let's implement the pandas rolling methods in our Jupyter Notebook. We will be using the same Walmart stock dataset:

```
| In [1]:  
| import pandas as pd  
| import matplotlib.pyplot as plt  
| %matplotlib inline
```

Let's start by reading the dataset:

```
| In [2]:  
| datafram = pd.read_csv('walmart_stock.csv',index_col='Date',parse_dates=True)
```

The top five records of the dataframe can be seen by using the `head()` function, as shown in the following code segment:

```
| In [3]:  
| datafram.head()
```

The output is as follows:

	Open	High	Low	Close	Volume	Adj Close
Date						
2012-01-03	59.970001	61.060001	59.869999	60.330002	12668800	52.61

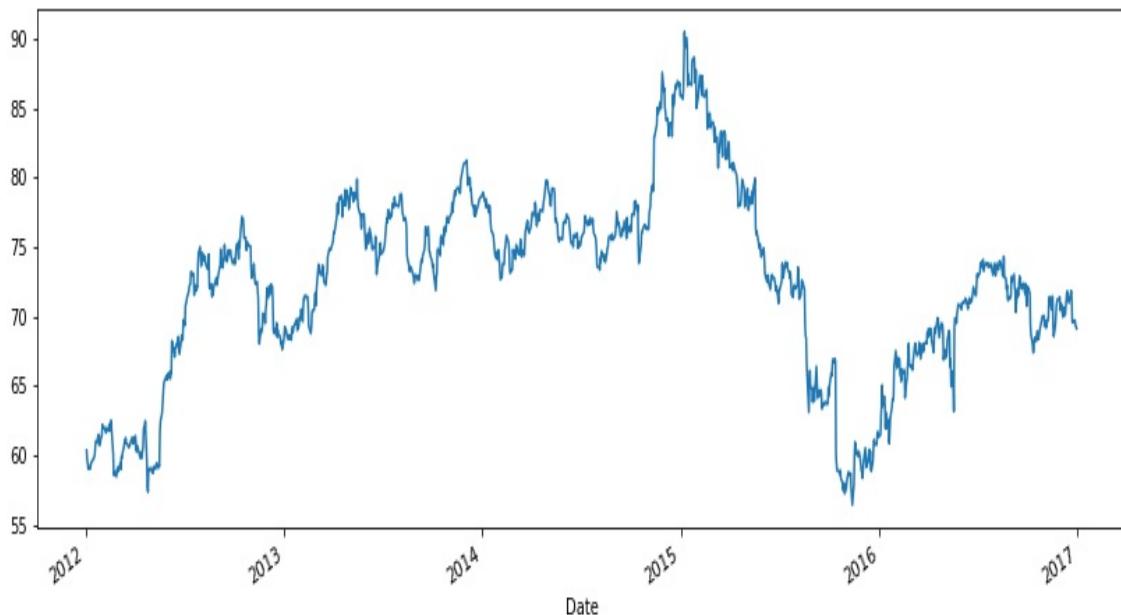
2012-01-04	60.209999	60.349998	59.470001	59.709999	9593300	52.07
2012-01-05	59.349998	59.619999	58.369999	59.419998	12768200	51.82
2012-01-06	59.419998	59.450001	58.869999	59.000000	8069400	51.45
2012-01-09	59.029999	59.549999	58.919998	59.180000	6679300	51.61

We will be using the `close` column of the stock dataset and plotting it, as shown in the following code segment:

```
In [6]:  
dataframe['Close'].plot(figsize=(14,5))
```

The output is as follows:

Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x8fdca20>



We can see that the preceding diagram consists of lots of noise. We try to find the average by week using the moving average or rolling mean, which can be implemented using the pandas library. To begin with, let's see the basic syntax of

the rolling function that's provided by pandas:

In []: `dataframe.rolling()`

Signature: `dataframe.rolling(window, min_periods=None, center=False, win_type=None, on=None, axis=0, closed=None)`

Docstring:

Provides rolling window calculations.

The important parameters of the rolling function are highlighted in the following table:

Parameters	Description
<code>window: int or offset</code>	This parameter is basically the size of the moving window. It refers to the number of rows of a dataset used for calculating the statistics, such as the mean and the standard deviation.
<code>min_periods: int</code>	This refers to the minimum number of rows in a window that are usually required to have a value. By default, the value is one.
<code>center: Boolean value</code>	This is used to position the label at the center of the window.



For more detailed information regarding the rolling function, please refer to the following link: <https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.rolling.html>.

Now, let's implement the rolling function with the size of the window set to seven days and apply an aggregate function such as mean, as shown in the following code segment:

```
In [8]:  
# rolling with windows size of every 7 days  
dataframe.rolling(7).mean().head(10)
```

The output is as follows:

	Open	High	Low	Close	Volume	/

Date						
2012-01-03	NaN	NaN	NaN	NaN	NaN	F
2012-01-04	NaN	NaN	NaN	NaN	NaN	F
2012-01-05	NaN	NaN	NaN	NaN	NaN	F
2012-01-06	NaN	NaN	NaN	NaN	NaN	F
2012-01-09	NaN	NaN	NaN	NaN	NaN	F
2012-01-10	NaN	NaN	NaN	NaN	NaN	F
2012-01-11	59.495714	59.895714	59.074285	59.440000	9.007414e+06	E
2012-01-12	59.469999	59.744285	59.007143	59.321429	8.231357e+06	E
2012-01-13	59.322857	59.638571	58.941428	59.297143	7.965071e+06	E
2012-01-17	59.397143	59.708571	59.105714	59.358572	7.355329e+06	E

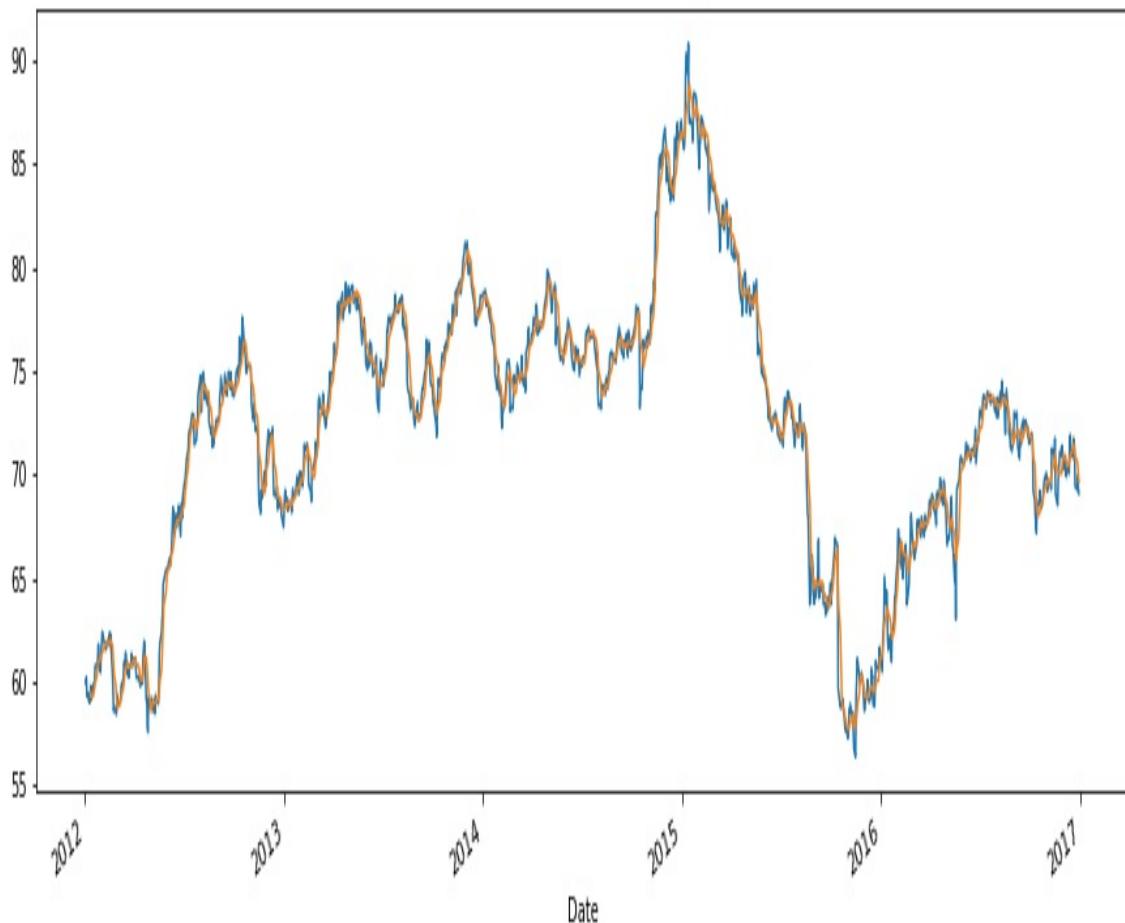
From the preceding output, we can see that the first six rows have values of NaN. This is because the size of the window is seven. Therefore, the record of the seventh row is the average of first six rows. Similarly, the eighth row is the average of the previous six rows. The other rows are also averaged in this way.

By using the rolling method with a particular window size, the data becomes less noisy and more reflective of the trend than the actual data. Let's plot the preceding data with respect to the `open` and `close` columns, as shown in the following code segment:

```
In [12]:
dataframe['Open'].plot()
dataframe.rolling(window=7).mean()['Close'].plot(figsize=(15,5))
```

The output is as follows:

```
Out[12]: <matplotlib.axes._subplots.AxesSubplot at 0x8e1ccc0>
```



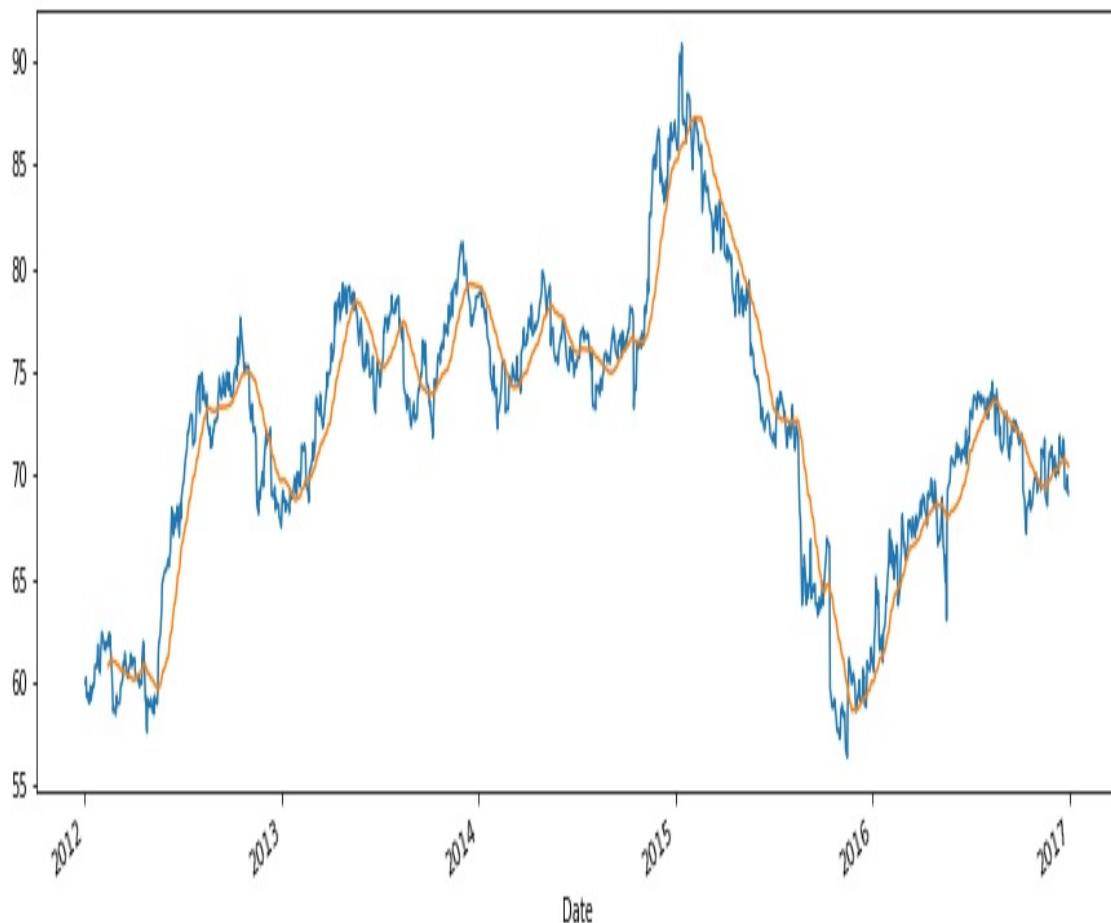
The `open` column (which has lots of variations) has been plotted using the noisy data from the stock price dataset, whereas the `close` column (the orange line) has been plotted using the `rolling` function, a window size of seven, and the mean as the aggregate function. We can see that the orange line is less noisy than the blue line, meaning we are able to capture the trend more efficiently.

Let's change the window size value to 30, which is a month instead of a week, and plot the stock dataset again:

```
In [11]:  
dataframe['Open'].plot()  
dataframe.rolling(window=30).mean()['Close'].plot(figsize=(15,5))
```

The output is as follows:

Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x8e90d68>



Now, let's move on and discuss the expanding function that's provided by the pandas library. The syntax of the expanding function is shown in the following screenshot:

In []: `dataframe.expanding()`

In []: `Signature: dataframe.expanding(min_periods=1, center=False, axis=0)`

In []: `Docstring:`

Provides expanding transformations.

The important parameters are as follows:

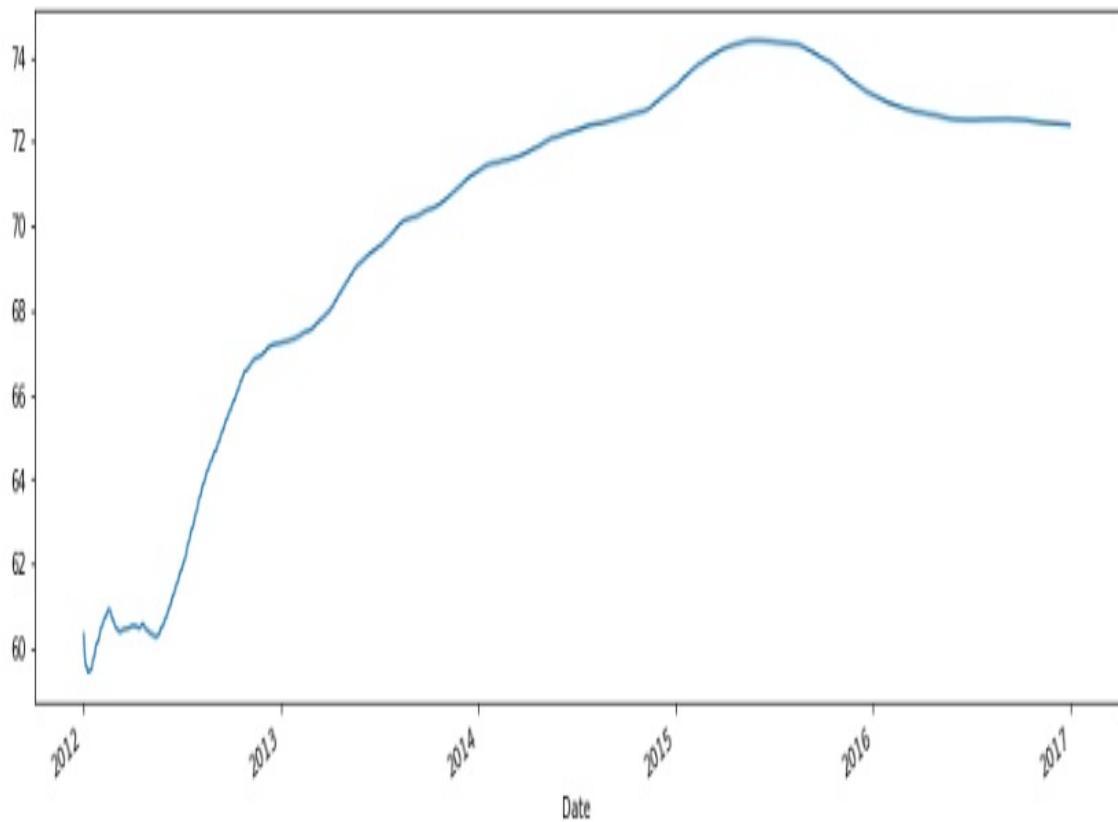
Parameters	Description
<code>min_periods</code>	The minimum number of records or observations required to have a value (otherwise the result is NaN).
<code>center</code>	Places the labels at the center of the window.
<code>axis</code>	The value of the axis can be either <code>0</code> or <code>1</code> . The default value is <code>0</code> .

The expanding function, along with the aggregate function mean, works as a rolling function in that it computes the average mean of all the previous stock values at specific time steps. With the expanding function, we will be able to understand whether the trend of the stock prices is increasing, decreasing, or stationary. The following is an example of an expanding function:

```
In [15]:  
#specify a minimum number of periods  
dataframe['Close'].expanding(min_periods=1).mean().plot(figsize=(16,5))
```

The output is as follows:

Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x91035c0>



Introduction to StatsModel libraries

In this section, we will be discussing the StatsModels library. The StatsModels library is a very important library in Python for working with time series data. It is inspired by the R statistical programming language. The StatsModels library consists of a huge range of statistical tests and mapping and plotting functions for different types of datasets and estimators.

StatsModels usually come installed with the Anaconda environment. If you want to install it manually, however, open the Anaconda prompt, or the cmd Command Prompt, and type the following command:

```
| conda install statsmodels
```

If you have opened the cmd Command Prompt, then type the following command:

```
| pip install statsmodels
```

Let's move on and take a look at how we can use the `statsmodels` library for various important activities, such as forecasting.



For additional information on the `statsmodels` library, please go through the `statsmodels` documentation at the following link: <http://www.statsmodels.org/stable/index.html>.

Error trend seasonality models

In this section, we will be discussing the **error trend seasonality (ETS)** models. The ETS models usually consist of the following:

- Exponential smoothing
- Trend method models
- ETS decomposition

Before moving on, let's discuss the key components of time series analysis, which are given here:

- Trend
- Seasonality
- Cyclical patterns
- Irregular patterns

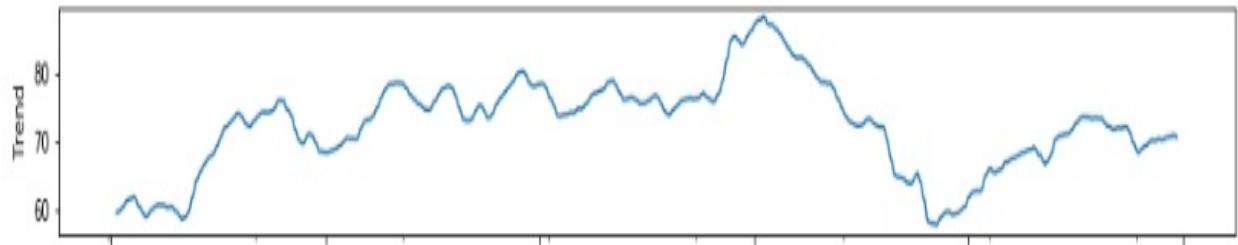
All time series data will follow the pattern of one of these components.

Trend

A trend pattern in time series data usually has the following characteristics:

- A gradual shift or movement to a relatively higher or lower value over a long period of time
- When the time series analysis shows a general pattern that is upward, we call it an uptrend
- When the trend pattern exhibits a general pattern that is downward, we call it a downtrend
- If there is no trend, we call it a horizontal or stationary trend

The following is an example of a trend pattern in the form of a graph:



The x axis basically represents the time.

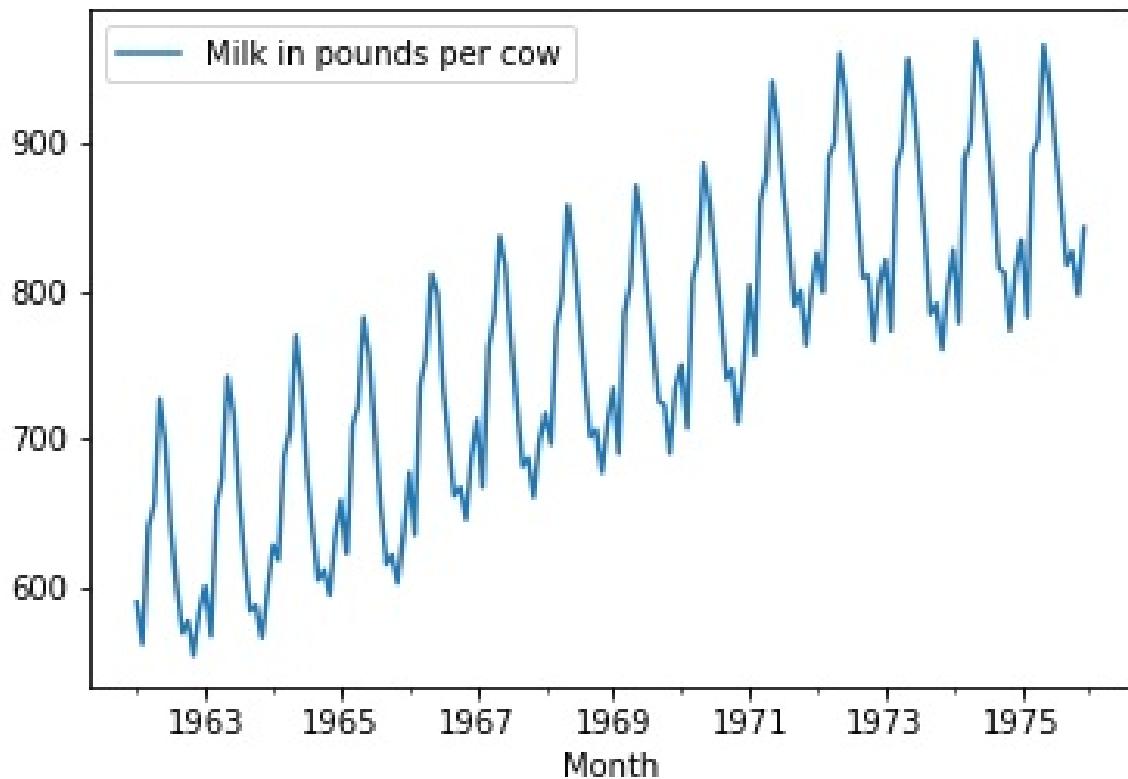
Seasonality

A seasonal pattern in time series data usually has the following characteristics:

- Upward or downward swings
- A repeating pattern within a fixed time period
- It is usually observed within one year

An example of a seasonal pattern would be if you live in a country with cold winters and hot summers, the amount you spend on air conditioning is likely to be high in summer and low in winter.

The following is an example of a seasonal pattern in the form of a graph:



Cyclical patterns

A cyclical pattern of time series data usually has the following characteristics:

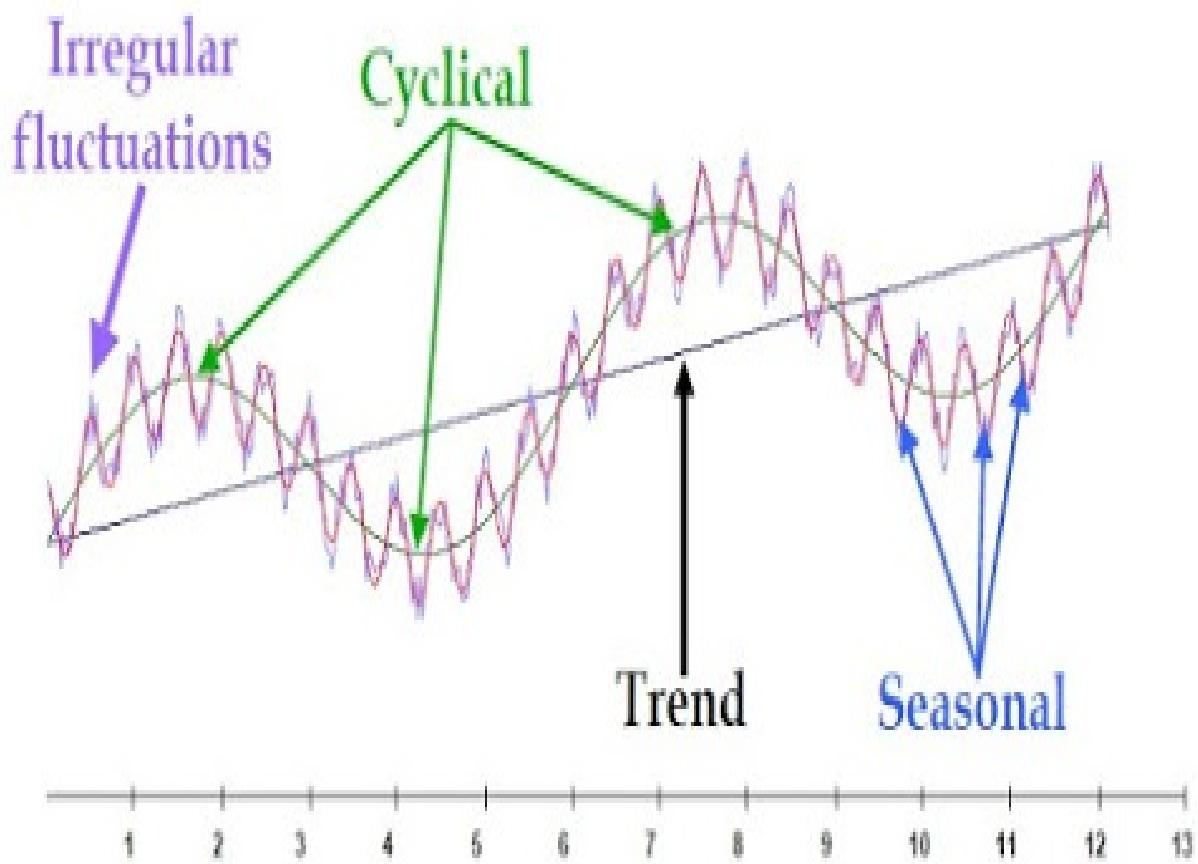
- It has repeating up and down movements
- It usually goes on over a year
- It does not have a fixed period
- It is much harder to predict

Irregular patterns

An irregular pattern usually has the following characteristics:

- It is erratic, unsystematic, and has residual fluctuations
- It has a short duration and does not repeat
- It is caused by random variations or unforeseen events
- It has a lot of white noise

The following diagram may help you visually understand all of the different components that have been discussed:



ETS decomposition

ETS decomposition models help us to visualize time series data with respect to these components. Let's take a look at an example with respect to the same Walmart stock dataset that we have used previously in this chapter. We will start by importing the basic libraries and reading the Walmart dataset, as shown here:

```
In [1]:  
import pandas as pd  
  
import matplotlib.pyplot as plt  
%matplotlib inline  
  
In [2]:  
dataframe = pd.read_csv('walmart_stock.csv', index_col='Date')  
dataframe.index = pd.to_datetime(dataframe.index)
```

We can get the top five records of the dataframe, as shown in the following code segment:

```
In [3]:  
dataframe.head()
```

The output is as follows:

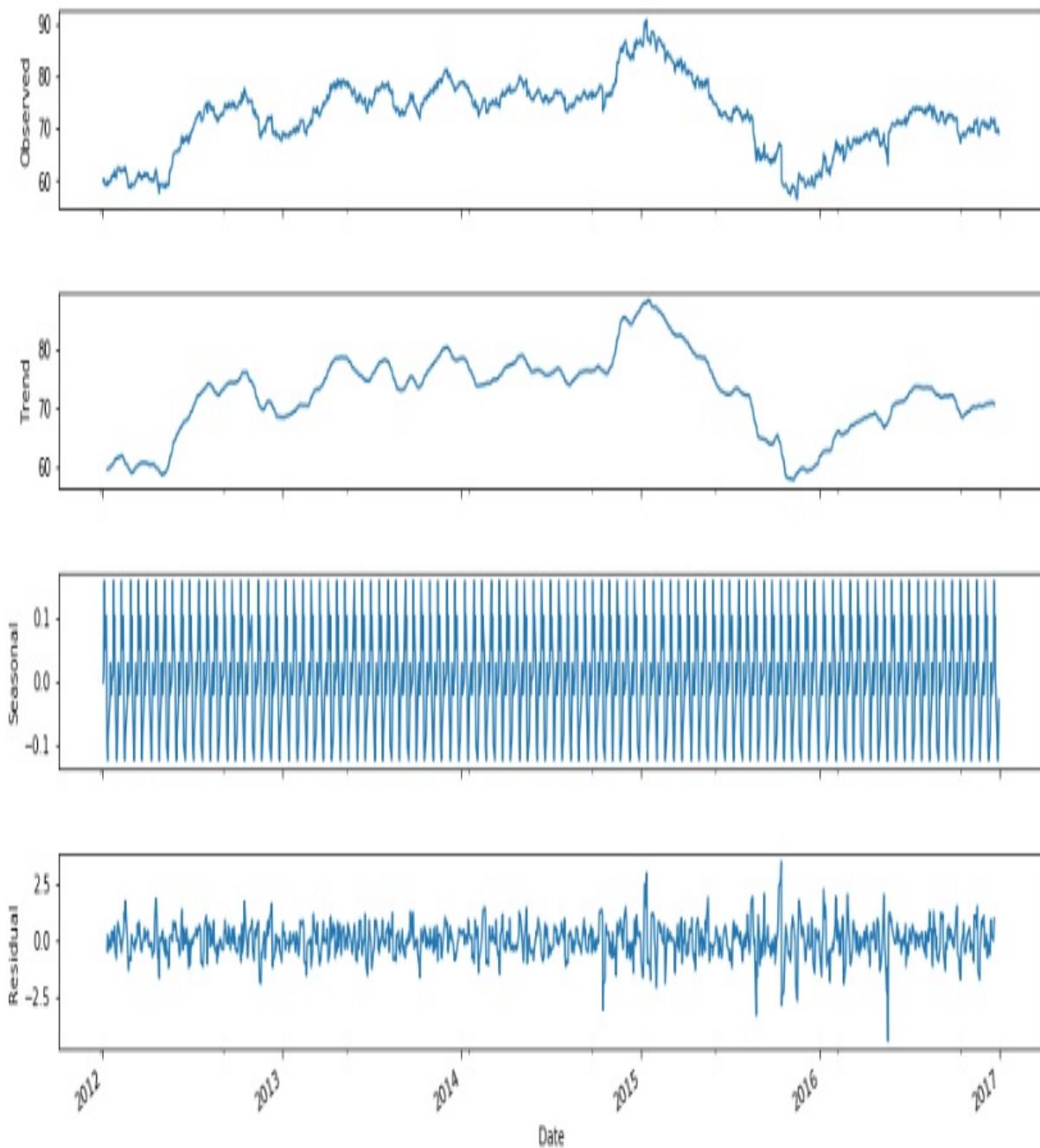
	Open	High	Low	Close	Volume	Adj Close
Date						
2012-01-03	59.970001	61.060001	59.869999	60.330002	12668800	52.61
2012-01-04	60.209999	60.349998	59.470001	59.709999	9593300	52.07
2012-01-05	59.349998	59.619999	58.369999	59.419998	12768200	51.82
2012-01-06	59.419998	59.450001	58.869999	59.000000	8069400	51.45
2012-01-09	59.029999	59.549999	58.919998	59.180000	6679300	51.61

To apply the ETS decomposition, we need to use the `statsmodels.tsa.seasonal` library and import the `seasonal_decompose` class, as shown in the following code segment:

```
In [5]:  
from statsmodels.tsa.seasonal import seasonal_decompose  
decomposition = seasonal_decompose(dataframe['Open'], freq=12)  
fig = plt.figure()  
fig = decomposition.plot()  
fig.set_size_inches(15, 8)
```

The output is as follows:

<Figure size 432x288 with 0 Axes>



From the preceding output, we can see the important components of the time series data, such as trends, seasonality, residuals (the error difference), and the observed data. This ETS decomposition helps us get more insights about the stock data to find the pattern and the trend of the stock prices. It also provides

information such as whether or not the data is seasonal.

AutoRegressive integrated moving average model

In this section, we are going to discuss the **AutoRegressive integrated moving average (ARIMA)** model, which is a very popular and widely used statistical tool for implementing time series forecasting in Python. It uses the `statsmodels` library to achieve forecasting. All of the topics we covered in the previous section will also be used in this technique when we are implementing the forecasting.

An ARIMA model is a statistical tool that's used for analyzing, gaining meaningful insights into, and forecasting, time series data. The key components of an ARIMA model are given here:

- **AutoRegression (AR)**
- **Integration (I)**
- **Moving Average (MA)**

All of these key components combine together to form an ARIMA model. Let's discuss these key components in detail.

AR

An AR model is a model that uses the dependent relationship between observed time series data and a number of lagged observations. In an AR model, the value of a variable in one period is related to its values in previous periods. Usually, the AR model is denoted by $AR(p)$ with p lags. It is given by the following equation:

$$y_t = \mu + \sum_{i=1}^p y_i y_{t-i} + \epsilon_t$$

Here, μ is a constant, y_p is the coefficient for the lagged variable, y_t is the

dependent variable at time t , y_{t-i} is the independent variable at a previous time period, and ϵ_t is the error at time t .

If we consider the value of p as one, AR(1) is given by the following equation:

$$y_t = \mu + y_{t-1} + \epsilon_t = \mu + y(L)y_t + \epsilon_t \text{ or}$$

$$(1 - yL)y_t = \mu + \epsilon_t$$

I

Whenever we apply an ARIMA model on time series data for time series forecasting, we usually need stationary time series data. If the data is not stationary, we use a technique called **differencing** to make the data stationary. Differencing basically means subtracting an observation from an observation of the previous step. Stationary data is a set of data with a mean and variance that does not change over time and that does not have trends. We can check whether a set of data is stationary using the Dickey Fuller test. If we find that it isn't stationary, we can stationarize it using high-order differencing. Let's discuss the Dickey Fuller Test for stationarity.

Let's assume we have an AR(1) model. The model is non-stationary, which means that a unit is present if $|p| = 1$.

The mathematical equation is as follows:

$$y_t = \rho y_{t-1} + \epsilon_t$$

$$y_t - y_{t-1} = \rho y_{t-1} - y_{t-1} + \epsilon_t$$

$$\Delta y_t = (\rho - 1)y_{t-1} + \epsilon_t = y y_{t-1} + \epsilon_t$$

We can estimate the preceding model for stationarity by testing the significance of the gamma (γ) coefficient:

- If the null hypothesis is not rejected, $\gamma^*=0$, then $y(t)$ is not stationary
- Difference the variable and repeat the test to see whether the differenced variable is stationary
- If the null hypothesis is rejected, then $y(t)$ is stationary

Testing for stationarity

We will be using the Dickey Fuller test to test the null hypothesis, which is denoted as H0. This indicates that the time series data has a unit root, which indicates that it is not stationary. If we reject the null hypothesis, we should use the alternate hypothesis, which is denoted as H1. This indicates that the time series data has no unit root and is basically stationary. We decide between H0 and H1 based on the p-value returned by the Dickey Fuller Test:

- A p-value that is typically smaller than, or equal to, 0.05 indicates strong evidence against the null hypothesis. In this case, we accept the alternate hypothesis, which indicates that the time series data is stationary.
- A p-value that is more than 0.05 indicates weak evidence against the null hypothesis. In this case, we fail to reject the null hypothesis and we accept the null hypothesis, which indicates that the time series data is non-stationary.

We will be discussing the Dickey Fuller Test in more detail once we implement the code.

Moving Average (MA)

The moving average model usually accounts for the possibility of a relationship between a variable and the residuals from the previous periods. The moving average is usually denoted by MA (q), where q is the lag for the residuals of the previous period. The basic equation of the moving average is as follows:

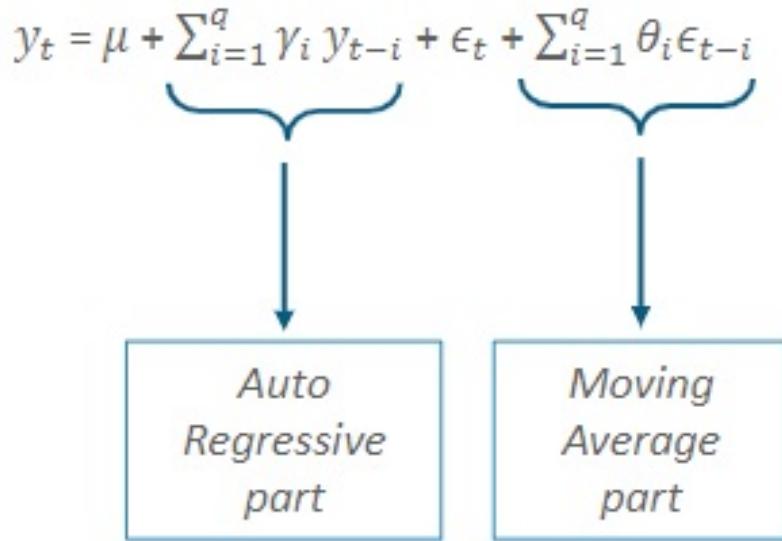
$$y_t = \mu + \epsilon_t + \sum_{i=1}^q \theta_i \epsilon_{t-i}$$

Here, θ_q is the coefficient for the lagged error term in time $t-q$.

If we consider the q value as one, the equation of the moving average MA (1) is as follows:

$$y_t = \mu + \epsilon_t + \theta_i \epsilon_{t-i}$$

The Arima model is basically the combination of all of these components for the forecasting prediction. If we combine the autoregressive model, AR(p), and the moving average model, MA (q), we get the notation of ARIMA (p,i,q). The equation is as follows:



ARIMA (p,i,q) basically denotes an ARIMA model with p autoregressive lags, q moving average lags, and a difference in the order of i . For the mathematical representation of ARIMA, we use the i notation for the differencing value. When we implement this with the help of Python, however, we will be using the d notation.

To select the values of p and q for the autoregressive and moving average models, we will have to use the concepts of **AutoCorrelation Function (ACF)** and **Partial Auto Correlation Function (PACF)**. ACF and PACF will be discussed later on as we move ahead with the implementation of the forecasting technique using ARIMA.

Let's go ahead and implement the Arima model with Python and the Statsmodels library. In this example, we are going to use the same Walmart stock price dataset. We will be using the ARIMA model to carry out the forecasting prediction for the `open` column.

ARIMA code

The general process for the ARIMA model when used for forecasting is as follows:

1. The first step is to visualize the time series data to discover the trends and find out whether the time series data is seasonal.
2. As we know, to apply the ARIMA model, we need to use stationary data. The second step, therefore, is to convert the non-stationary data into stationary data using the Dickey Fuller Test.
3. We then select the p and q values for $ARIMA(p,i,q)$ using ACF and PACF.
4. The next step is to construct the ARIMA model.
5. Finally, we use the model for the prediction.

Let's begin by importing the necessary libraries and reading the dataset, as shown in the following code segment:

```
In [1]:  
import numpy as np  
import pandas as pd  
import statsmodels.api as sm  
  
import matplotlib.pyplot as plt  
%matplotlib inline  
  
In [3]:  
df = pd.read_csv('walmart_stock.csv')
```

We can check the top five records of the dataframe, as shown in the following code segment:

```
In [4]:  
df.head()
```

The output is as follows:

	Date	Open	High	Low	Close	Volume	A
0	2012-01-03	59.970001	61.060001	59.869999	60.330002	12668800	5

1	2012-01-04	60.209999	60.349998	59.470001	59.709999	9593300	52.07
2	2012-01-05	59.349998	59.619999	58.369999	59.419998	12768200	51.82
3	2012-01-06	59.419998	59.450001	58.869999	59.000000	8069400	51.45
4	2012-01-09	59.029999	59.549999	58.919998	59.180000	6679300	51.61

Then, we are going to follow the same steps we used in the previous section on time series data. First, we need to convert the `Date` column into a date time object using pandas. After that, we set this date column as the index of the dataframe, as shown in the following code segment:

```
In [5]:  
df['Date'] = pd.to_datetime(df['Date'])  
  
In [6]:  
df.set_index('Date', inplace=True)  
df.head()
```

The output is as follows:

Date	Open	High	Low	Close	Volume	Adj Close
2012-01-03	59.970001	61.060001	59.869999	60.330002	12668800	52.61
2012-01-04	60.209999	60.349998	59.470001	59.709999	9593300	52.07
2012-01-05	59.349998	59.619999	58.369999	59.419998	12768200	51.82
2012-01-06	59.419998	59.450001	58.869999	59.000000	8069400	51.45
2012-01-09	59.029999	59.549999	58.919998	59.180000	6679300	51.61

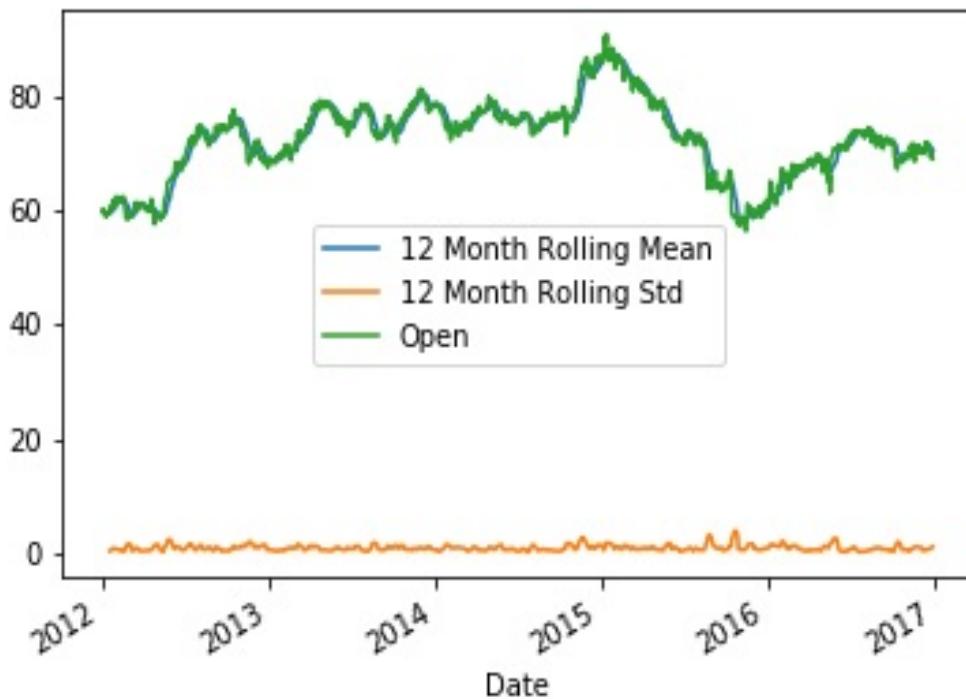
As we can see from the output, the `Date` column is set as an index in the dataframe.

Let's see the rolling mean and the rolling standard deviation of the `open` column of the time series dataset that we discussed in the previous section:

```
In [8]:  
timeseries = df['Open']  
  
In [9]:  
timeseries.rolling(12).mean().plot(label='12 Month Rolling Mean')  
timeseries.rolling(12).std().plot(label='12 Month Rolling Std')  
timeseries.plot()  
plt.legend()
```

The output is as follows:

Out[9]: <matplotlib.legend.Legend at 0xb17edd8>



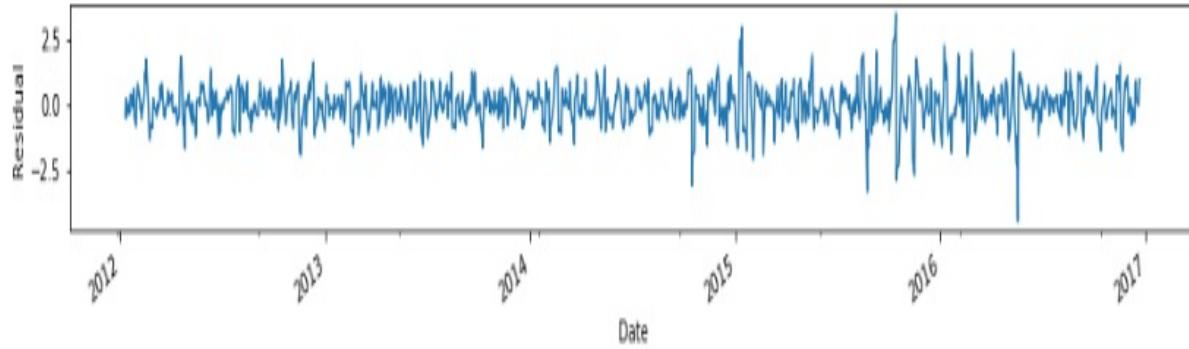
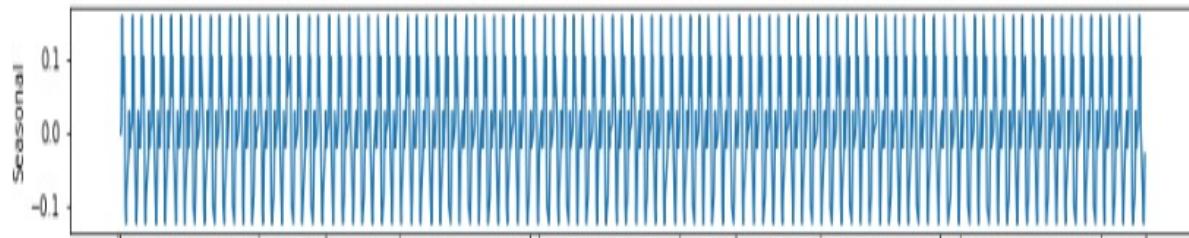
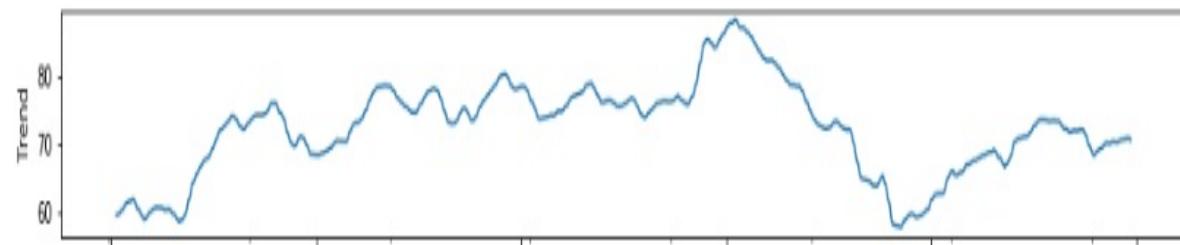
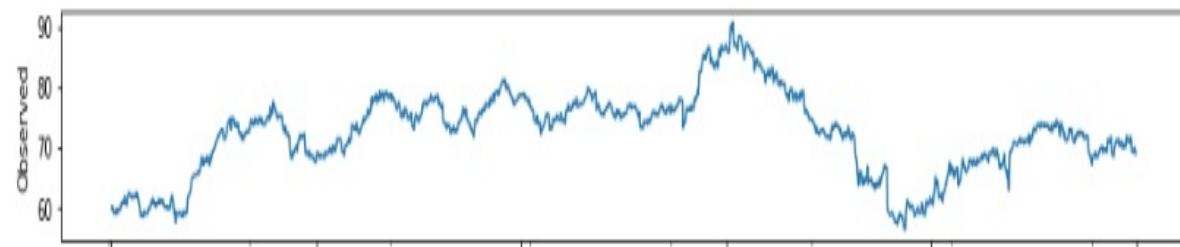
The next step is to use the ETS decomposition method to visualize the general trend of the data, as we discussed in the earlier section:

```
In [11]:  
from statsmodels.tsa.seasonal import seasonal_decompose  
decomposition = seasonal_decompose(df['Open'], freq=12)
```

```
| fig = plt.figure()  
| fig = decomposition.plot()  
| fig.set_size_inches(15, 8)
```

The output is as follows:

<Figure size 432x288 with 0 Axes>



From the ETS decomposition, we can see that the Walmart stock prices follow a seasonal pattern and that there are both uptrends and downtrends in the trending pattern. The next step is to find out whether the dataset is stationary. To do this, we will define a method to check whether the time series data is stationary using the Dickey Fuller library, which is called `adfuller`. This is present in the `statsmodels` library. Based on the p value that's returned, we will decide whether the data is stationary or not. First, we will import the `adfuller` library or class and create a method, as shown in the following code segment:

```
In [14]:  
from statsmodels.tsa.stattools import adfuller  
  
In [15]:  
# Store in a function for later use!  
def adf_check(time_series):  
    """  
    Pass in a time series, returns ADF report  
    """  
    result = adfuller(time_series)  
    print('Augmented Dickey-Fuller Test:')  
    labels = ['ADF Test Statistic', 'p-value', '#Lags Used', 'Number of Observations Used']  
  
    for value,label in zip(result,labels):  
        print(label+' : '+str(value) )  
  
    if result[1] <= 0.05:  
        print("strong evidence against the null hypothesis, reject the null hypothesis.  
    else:  
        print("weak evidence against null hypothesis, time series has a unit root, indic
```

Once we create the method, we can pass the dataframe with `open` to the method and see whether or not the data is stationary, as shown in the following code segment:

```
In [15]:  
adf_check(df['Open'])  
  
Out[15]:  
Augmented Dickey-Fuller Test:  
ADF Test Statistic : -2.315173149148315  
p-value : 0.1671210162134677  
#Lags Used : 11  
Number of Observations Used : 1246  
weak evidence against null hypothesis, time series has a unit root, indicating it is nor
```

From the preceding output, we can see that the value of p is larger than 0.05, so we decide that the data is not stationary. To make the data stationary, we will follow the differencing technique, which we have already discussed. In differencing, the first difference of a time series is the series of changes from one period to the next. We take away this change by using a shift operation, which

we can do easily with pandas. We can continue to take away the second difference, the third difference, and so on, until the data is stationary. We can do this easily with pandas. You can continue to take the second difference, third difference, and so on, until your data is stationary:

```
| In [16]:  
| df['Open First Difference'] = df['Open'] - df['Open'].shift(1)  
  
| In [17]:  
| df['Open First Difference'].head()  
  
| Out[17]:  
| Date  
| 2012-01-03      NaN  
| 2012-01-04    0.239998  
| 2012-01-05   -0.860001  
| 2012-01-06    0.070000  
| 2012-01-09   -0.389999  
| Name: Open First Difference, dtype: float64
```

After the first differencing, we will pass the new dataset column, `open First Difference`, to the same method of the Dickey Fuller Test to see whether the data is stationary:

```
| In [18]:  
| adf_check(df['Open First Difference'].dropna())  
  
| Out[18]:  
| Augmented Dickey-Fuller Test:  
| ADF Test Statistic : -10.395143169790536  
| p-value : 1.9741449125945693e-18  
| #Lags Used : 10  
| Number of Observations Used : 1246  
| strong evidence against the null hypothesis, reject the null hypothesis. Data has no uni
```

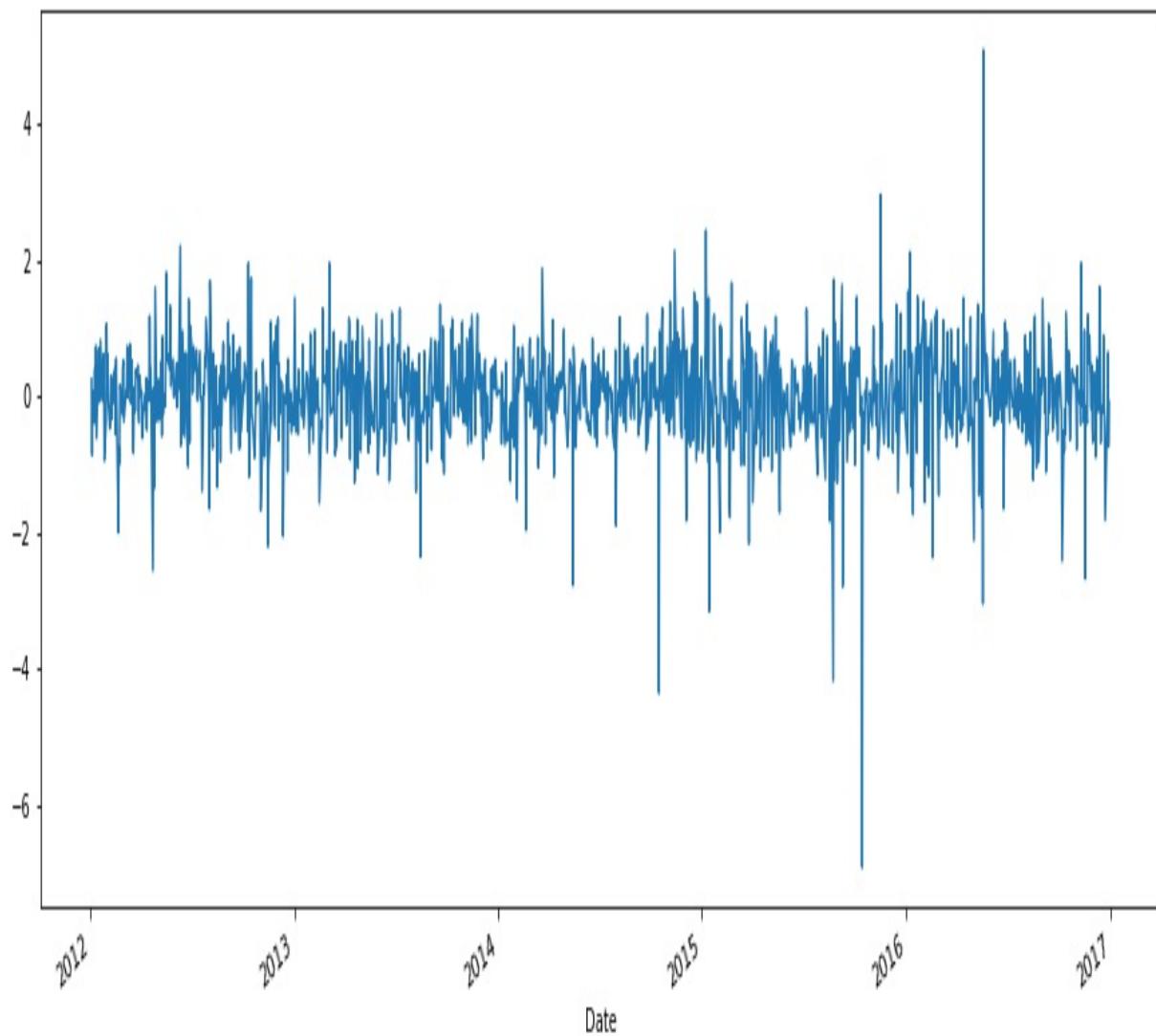
As you can see in the preceding output, we are now getting a p value that is much less than 0.05, so we can now consider the dataset as stationary. We then set the differencing (d) value in the Arima (p,d,q) model as one, as we have only observed one difference to establish whether the data is stationary.

If we try to plot the `open First Difference`, we will see the stationary pattern, as shown here:

```
| In [19]:  
| df['Open First Difference'].plot()
```

The output is as follows:

```
<matplotlib.axes._subplots.AxesSubplot at 0xb468ac8>
```



So far, we have found out the value of d in the ARIMA (p,d,q) model, which is the number of times we have to carry out the differencing. We now need to find the values of p and q , which are the lags for the AR and the MA models.

To find the p and q parameters, we will be using ACF and PACF.

Autocorrelation function

An autocorrelation plot (also known as a correlogram) shows the correlation of the series with itself, lagged by x time units. The y axis is the correlation, while the x axis is the number of time units of the lag.

Imagine taking a time series of length T , copying it, and deleting the first observation of copy #1 and the last observation of copy #2. Now, you have two series of length $T-1$, for which you calculate a correlation coefficient. This is the value of the vertical axis at $x=1$ in your plots. It represents the correlation of the series lagged by one time unit. You go on and do this for all possible time lags x , and this defines the plot.

Autocorrelation interpretation

The actual interpretation of an autocorrelation plot and how it relates to ARIMA models can get a bit complicated, but there are some basic common methods we can use. Our main priority here is to try and figure out whether we will use the AR, MA, or both components of the ARIMA model, as well as how many lags we should use. In general, you would use either AR or MA; using both is less common.

- If the autocorrelation plot shows a positive autocorrelation at the first lag (lag-1), then we should use the AR terms in relation to the lag
- If the autocorrelation plot shows a negative autocorrelation at the first lag, then we should use the MA terms

Partial autocorrelation (PACF)

In general, a partial correlation is a conditional correlation. It is the correlation between two variables under the assumption that we know and take into account the values of another set of variables. For instance, consider a regression context in which y is the response variable and x_1 , x_2 , and x_3 are predictor variables. The partial correlation between y and x_3 is the correlation between the variables that specifies how both y and x_3 are related to x_1 and x_2 .

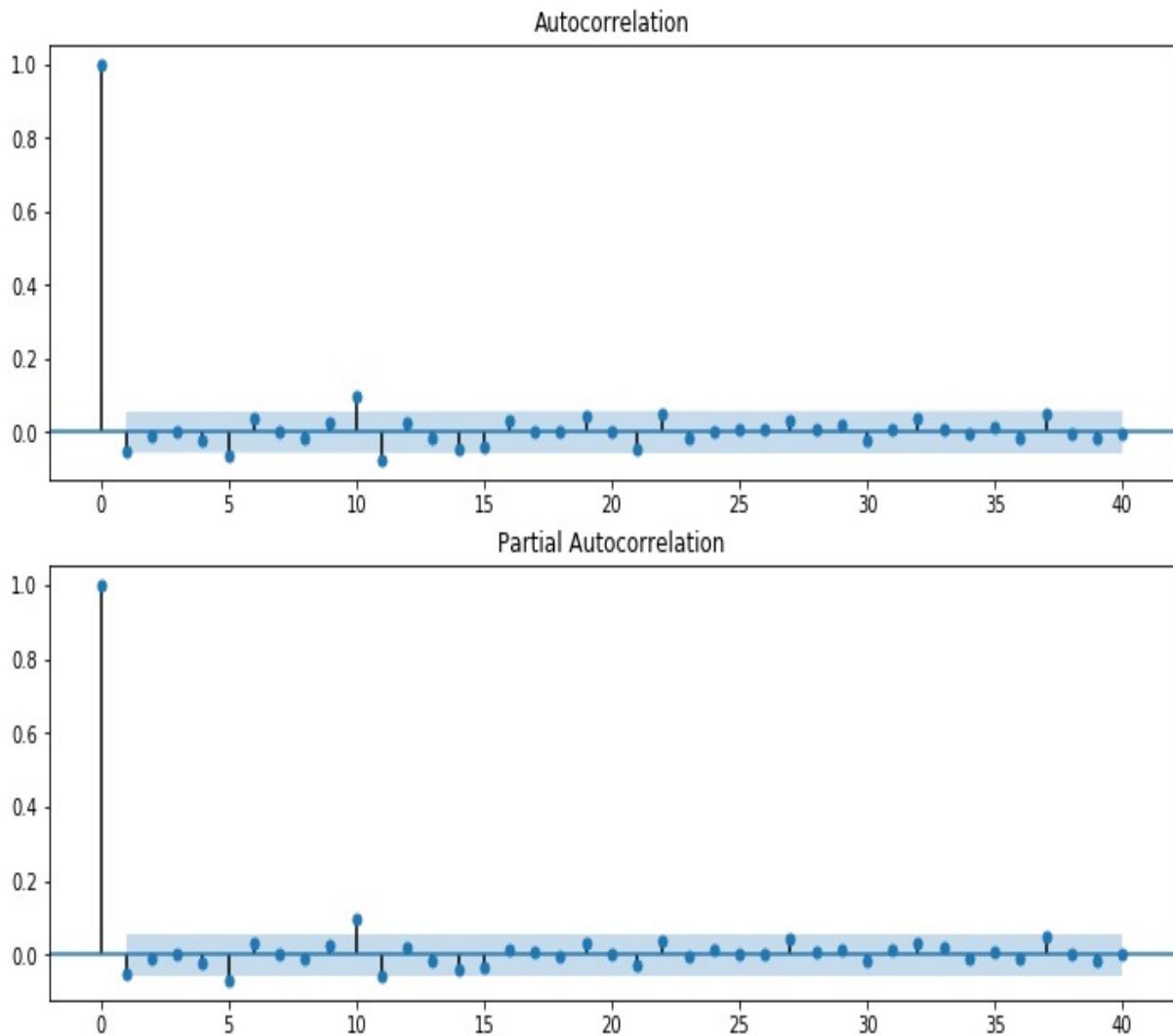
Formally, this relationship is defined as follows:

$$\frac{\text{covariance}(y, x_3 | x_1, x_2)}{\sqrt{\text{variance}(y | x_1, x_2) \text{Variance}(x_3 | x_1, x_2)}}$$

Now, let's plot the final ACF and PACF plots to select the p and q values, as shown in the following code segment:

```
In [26]:  
fig = plt.figure(figsize=(12,8))  
ax1 = fig.add_subplot(211)  
fig = sm.graphics.tsa.plot_acf(df['Open First Difference'].iloc[13:], lags=40, ax=ax1)  
ax2 = fig.add_subplot(212)  
fig = sm.graphics.tsa.plot_pacf(df['Open First Difference'].iloc[13:], lags=40, ax=ax2)
```

The output is as follows:



Let's think about how we can select the values for p and q .

- p : The number of lag observations included in the model. This is the number before the first inverted bar in the ACF (we begin counting from zero).
- d : The number of times that the raw observations are differenced, also called the degree of differencing.
- q : The size of the moving average window, also called the order of moving average. This is the number before the first inverted bar in the PACF (again, we begin counting from zero).

From the preceding diagram, we can see that the value of both p and q is 0. We will provide these parameters when we call the ARIMA model.

Let's see how we can import the ARIMA model. We will be using the seasonal ARIMAX model to predict the future of the time series data:

```
| # For non-seasonal data  
| from statsmodels.tsa.arima_model import ARIMA
```

```
In [28]: # I recommend you glance over this!
```

```
#  
help(ARIMA)
```

```
Help on class ARIMA in module statsmodels.tsa.arima_model:
```

```
class ARIMA(ARMA)  
    Autoregressive Integrated Moving Average ARIMA(p,d,q) Model  
  
    Parameters  
    -----  
    endog : array-like  
        The endogenous variable.  
    order : iterable  
        The (p,d,q) order of the model for the number of AR parameters,  
        differences, and MA parameters to use.  
    exog : array-like, optional  
        An optional array of exogenous variables. This should *not* include a  
        constant or trend. You can specify this in the `fit` method.  
    dates : array-like of datetime, optional  
        An array-like object of datetime objects. If a pandas object is given  
        for endog or exog, it is assumed to have a DateIndex.  
    freq : str, optional  
        The frequency of the time-series. A Pandas offset or 'B', 'D', 'W',  
        'M', 'A', or 'Q'. This is optional if dates are given.  
  
    Notes  
    -----  
    If exogenous variables are given, then the model that is fit is
```

One thing to note is that when we visualized the stock dataset using the ETS decomposition, we were able to find the seasonal pattern in the time series.

Usually, when we find a seasonal pattern, we should use another model, which is called the seasonal ARIMAX model. The basic difference between the seasonal ARIMAX model and the ARIMA model is that we need to provide an extra parameter for the seasonal ARIMAX model, which is `seasonal_order`. This parameter will be in tuple form, (p,d,q,S) , where p and q are the lags for the autoregressive and moving average models, d is the differencing value, and S is the number of months in a year, which is 12.

Now, let's take a look at how to create a seasonal ARIMAX model:

```
In [30]:  
# We have seasonal data!(P,D,Q,S)  
model = sm.tsa.statespace.SARIMAX(df['Open'],order=(0,1,0), seasonal_order=(0,1,0,12))  
results = model.fit()  
print(results.summary())
```

The output is as follows:

Statespace Model Results

Dep. Variable:	Open	No. Observations:	1258
Model:	SARIMAX(0, 1, 0)x(0, 1, 0, 12)	Log Likelihood	-1827.327
Date:	Sun, 07 Oct 2018	AIC	3656.654
Time:	23:16:03	BIC	3661.781
Sample:	0	HQIC	3658.581
	- 1258		
Covariance Type:	opg		

	coef	std err	z	P> z	[0.025	0.975]
sigma2	1.1025	0.025	44.135	0.000	1.054	1.151

Ljung-Box (Q):	356.60	Jarque-Bera (JB):	952.04
Prob(Q):	0.00	Prob(JB):	0.00
Heteroskedasticity (H):	1.66	Skew:	-0.25
Prob(H) (two-sided):	0.00	Kurtosis:	7.26

Warnings:

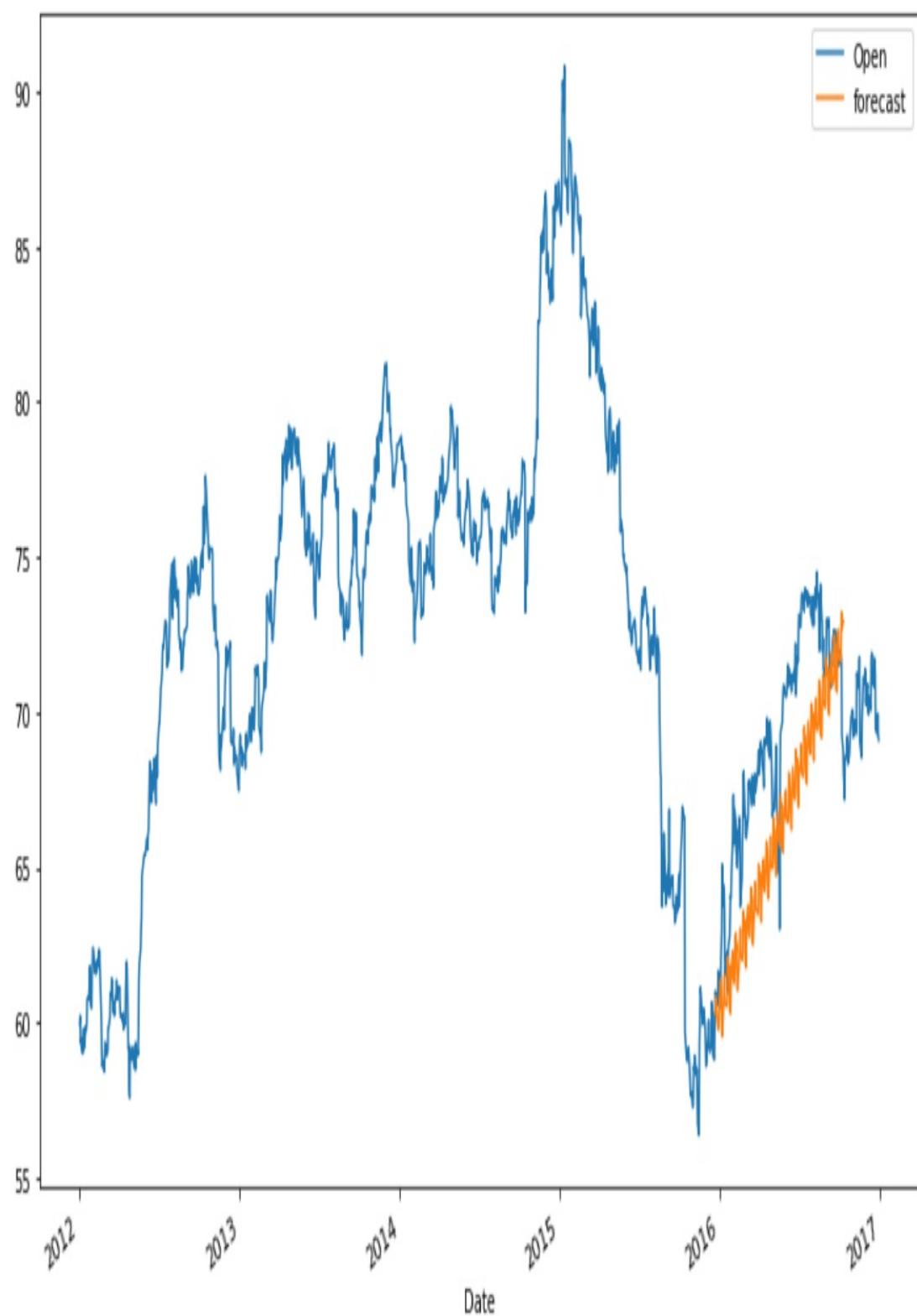
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

After we create the model, we need to make a prediction and compare the predicted data and the real data by plotting it. To do this, we use the predict method, which is present in the seasonal ARIMAX model. We provide the row number from which the prediction needs to be started. We can change the value to make different predictions. The code for thus is as follows:

```
| In [36]:  
| df['forecast'] = results.predict(start = 1000, end= 1200, dynamic= True)  
| df[['Open', 'forecast']].plot(figsize=(12,8))
```

The output is as follows:

Out[36]: <matplotlib.axes._subplots.AxesSubplot at 0xf1cacf8>



In the preceding prediction, we can see that the seasonal ARIMAX model provides a better prediction as it is able to determine the trend regarding how stock prices move.

Summary

In this chapter, we have discussed various topics related to time series analysis, such as timeshifts, date time index, time resampling, and time series rolling. These are very important when we implement forecasting or predictions using the *ARIMA* (p,d,q) model. We have also discussed the important components of time series data, which include trends, seasonality, cyclical, and irregularity. After that, we looked at the ARIMA model and discussed its main components, which were autoregressive (p), integration (d), and moving average (q). We also explored the ETS decomposition and found that stock prices have a seasonal pattern.

Finally, we implemented the seasonal ARIMAX model using Python code and the `statsmodels` library. We were then able to see a stock data prediction.

In the next chapter, we will be discussing how to measure investments risks.

Measuring Investment Risks

In this chapter, we will be discussing how to measure investment risks. When we think about an investment, we have to consider both its potential advantages and its possible downfalls. In other words, we should bear in mind both the profit that will be made if everything goes well, and the potential losses that might occur if the investment is unsuccessful.

In this chapter, we will use a very structured approach. We will explain concepts from a theoretical perspective first, in order to understand their meaning and purpose, and then we'll look at some actual calculations using Python. In the first few sections of this chapter, we will focus on the rates of return of an investment. We will start by learning how to calculate the rate of return of a single security, then we'll continue by calculating the rate of return of a portfolio of securities. After that, we'll explain stock indices and how they are composed. We'll also learn about risk and the statistical measures that allow us to calculate the standard deviation and variance of risk. Following this, we'll examine the relationship between two securities, also known as correlation, and we will learn how to calculate the correlation and covariance of two securities.

This chapter focuses on the following topics:

- Calculating the return of a single security
- Calculating the return of a portfolio of securities
- Risk, standard deviation, and variance
- Correlation and covariance

Technical requirements

In this chapter, we will be using the Jupyter Notebook for coding purposes. We will also be using the pandas and NumPy libraries. The code is provided in the GitHub repository that can be found at the following link: <https://github.com/PacktPublishing/Hands-on-Python-for-Finance/tree/master/Chapter%204>.

Before moving on, let's take a look at the different sources from which we can extract financial data.

Sources of financial data

Let's look at how to use financial data from a technical perspective. The data you'll be using for financial analysis, or for any other type of analysis, is likely to come from one of two sources: a web server or your computer. In practice, in order to access data stored on a web server, you'll need to connect to its **application programming interface (API)**. In our case, we will need a financial data API. We can also call these online financial data sources; examples of these include the IEX, Morningstar, Alpha Vantage, and the Quandl API.

An important characteristic of these servers is that they provide up-to-date data. For this reason, you'll need an internet connection to access these APIs. The pandas datareader is an example of a module that will help you retrieve data from financial data sources and prepare it for analysis. Apart from extracting online data, you may use information that is stored as certain types of files on your computer. One file format that every programmer or analyst needs to know how to work with is the **comma-separated values (CSV)** file. We used this type of file in the previous chapter for our stock forecasting using the ARIMA model. It is important to know that neither of these two approaches are straightforward. In the next few sections, we will clarify how these two methods work and we will introduce you to the relevant programming tools. This will be a valuable addition to your skill set.

It is usually much harder to clean and organize your data than to conduct the subsequent business or financial analysis. At first glance, it might seem like a better idea to use APIs as these provide the better data, and all you need is an internet connection. So, why bother using CSV files at all? Firstly, web services are prone to breaking down for unknown periods of time. Secondly, it is possible that a certain API may contain only part of the data that you need for proper financial analysis. Currently, there's no free API out there that provides financial data that is as rich as you might want. Some APIs offer single or multiple stock data, while others provide foreign stock data or data to do with market indices. Note that an analyst may often need to work with all of these. Furthermore, you can sometimes only connect to an API if you use Python

3, and not Python 2.

Taking these drawbacks into consideration, most of the time, we will be using various CSV files that are present in the GitHub repository of this chapter. Although the information contained in the CSV file may not be up to date, it will be the most appropriate as it is the richest and easiest-to-handle data that we can provide. This will allow you to focus on coding, financial theory, and the relation between the two, as we have already mentioned. If you insist on working with up-to-date financial data, however, it is suggested that you use the Morningstar or Alpha Vantage API if you are using Python 2 or Python 3. You should only use IEX if you are on Python 3. Let's now go ahead and look at some examples of how we can extract or retrieve financial data from various sources.

In order to read the financial data from the API, we will be using the `pandas-datareader` library. To use this, we need to install this library; type the following command in the Anaconda Command Prompt:

```
| conda install pandas-datareader
```

Once you install the package, we will be reading from various financial APIs, such as IEX and Morningstar. First, we will import the `pandas` and `pandas_datareader` libraries. While using this API, we will retrieve details related to Apple, which is denoted as AAPL in the stock index:

```
In [5]:  
import pandas as pd  
pd.core.common.is_list_like = pd.api.types.is_list_like  
from pandas_datareader import data as wb
```

Make sure you execute the second line, otherwise you may get an error. This line indicates that we should *bypass* the version control of your package manager. We will be using the `DataReader` function to read from the financial API. The basic syntax of this function is given as follows:

```
In [28]: GOOGL = wb.DataReader('GOOGL', data_source='morningstar', start='2015-1-1')
```

```
In [29]: Signature: wb.DataReader(name, data_source=None, start=None, end=None, retry_count=3, pau
```

```
se=0.001, session=None, access_key=None)
```

```
Out[29]: Docstring:
```

```
Imports data from a number of online sources.
```

The following are the details of some of the specified parameters:

Parameters	Description
name	This is the name of the dataset. Some data sources, such as Google or FRED, will accept a list of names.
datasource	This is the name of the financial API, such as Morningstar or IEX.
start	This parameter specifies where we want to start reading the financial data from.
end	This parameter specifies where we should stop reading the financial data.

Let's now go ahead and see whether we can retrieve the data from the financial API. The code to do this is as follows:

```
In [9]: AAPL_IEX = wb.DataReader('AAPL', data_source='iex', start='2015-1-1')  
Out[9]: 5y
```

The following code helps us to see the top five records of the `AAPL_IEX` dataframe:

```
In [10]: AAPL_IEX.head()
```

The output is as follows:

Out[10]:

	open	high	low	close	volume
date					
2015-01-02	104.2471	104.2939	100.4662	102.3192	53204626
2015-01-05	101.3459	101.6828	98.6506	99.4367	64285491
2015-01-06	99.7082	100.5411	97.9206	99.4461	65797116
2015-01-07	100.3258	101.2617	99.8532	100.8406	40105934
2015-01-08	102.2257	104.9584	101.7296	104.7151	59364547

In the preceding code, our goal was to extract data from the IEX data source about AAPL from January 1, 2015. The `DataReader` function allows us to do this in one row. The `wb` alias uses this `DataReader` function and we specify three important parameters. The first one is the ticker of the Apple stock, AAPL. Second, we select the data source, Morningstar. Finally, we specify the start date, which is January 1, 2015. With respect to these parameters, we get the preceding output.

We can use the same function to retrieve details about the stock price of Google using the `GOOGL` ticker, as shown in the following code block:

```
In [12]:  
GOOGL_IEX = wb.DataReader('GOOGL', data_source='iex', start='2015-1-1')  
  
Out[12]:  
5y
```

The following code helps us to see the top five records:

```
In [13]:  
GOOGL_IEX.head()
```

The output is as follows:

Out[13]:

	open	high	low	close	volume
date					
2015-01-02	532.60	535.8000	527.88	529.55	1327870
2015-01-05	527.15	527.9899	517.75	519.46	2059119
2015-01-06	520.50	521.2100	505.55	506.64	2731813
2015-01-07	510.95	511.4900	503.65	505.15	2345875
2015-01-08	501.51	507.5000	495.02	506.91	3662224

The output that we see indicates that we can retrieve a maximum of the past five years of data.

We can also use different data sources, such as Morningstar, to retrieve financial data, as shown here:

```
| In [20]:  
| AAPL = wb.DataReader('AAPL', data_source='morningstar', start='2015-1-1')
```

The following code helps us to see the top five records:

```
| In [22]:  
| AAPL.head()
```

The output is as follows:

Out[22]:

		Close	High	Low	Open	Volume
Symbol	Date					
AAPL	2015-01-01	110.38	110.38	110.380	110.38	0
	2015-01-02	109.33	111.44	107.350	111.39	53204626
	2015-01-05	106.25	108.65	105.410	108.29	64285490
	2015-01-06	106.26	107.43	104.630	106.54	65797116
	2015-01-07	107.75	108.20	106.695	107.20	40105934

The following code helps us to read the dataset from the Morningstar data source:

```
In [28]:  
GOOGL = wb.DataReader('GOOGL', data_source='morningstar', start='2015-1-1')
```

The following code helps us to see the top five records:

```
In [29]:  
GOOGL.head()
```

The output is as follows:

Out[29]:

		Close	High	Low	Open	Volume
Symbol	Date					
GOOGL	2015-01-01	530.7	530.7	530.7	530.7	0
	2015-01-02	529.6	535.8	527.9	532.6	1327870
	2015-01-05	519.5	528.0	517.8	527.2	2059119
	2015-01-06	506.6	521.2	505.6	520.5	2731813
	2015-01-07	505.2	511.5	503.7	511.0	2345875

Security risks and returns

One of the most important questions asked in this domain is, why do different investments have different levels of risk and profitability? The answer to this question can be easily understood if we compare stocks and bonds. Let's consider an example: government bonds conventionally offer an average rate of return of 3%. Historically, there have been very few cases of governments going bankrupt and not recompensing what's owed to investors. If an investor buys a government debt, they can be certain that one year from now they will obtain their initial investment plus the expected interest. While this investment isn't risk-free, the risk is contained. Equity shares have a higher rate of return—approximately 6%. However, they are associated with much more frequent fluctuations and price changes, as different variables influence a company's share price.

The art of finance isn't about maximizing an investor's returns in a year. It is about making carefully considered decisions that take into account both risk and return, and optimizing an investment portfolio. Now that we have understood the importance of risk and return, we are ready to learn how to calculate the two parameters.

Calculating a security's rate of return

The main goal of every investor is to earn a good rate of return on his investment. Imagine that you bought a share of Apple's equity at the beginning of the year. Back then, Apple shares were trading at \$105, whereas at the end of the year, their price increased to \$160. Taking into account any transaction costs and dividends paid by the company throughout the year, if you sell that share, you will receive \$116, so you've made a profit of \$11.

To decide whether we should be happy with this profit, we should compare our investment in Apple with other investments. All we have to do is to calculate a percentage rate of the return of the investment. Once we do that, we can compare it to other investments. The rate of return of an investment is given by the following formula:

$$\text{Rate of Return} = \frac{\text{End Price} - \text{Beginning Price}}{\text{Beginning Price}}$$

This is as follows:

$$\frac{\text{End Price} - \text{Beginning Price}}{\text{Beginning Price}} = \frac{\text{End Price}}{\text{Beginning Price}} - 1$$

In our case, this will be \$116 minus \$105, divided by \$105. This gives us a 10.5% rate of return. This computation of the rate of return is called a **simple rate of return**. If we assume that Apple paid a \$2 dividend at the end of the year, the rate of return calculation becomes the following:

$$\frac{(\$116 + \$2) - \$105}{\$105} = 12.4\%$$

We can calculate the logarithmic return of the investment as follows:

$$\ln\left(\frac{\text{End Price}}{\text{Beginning Price}}\right)$$

$$\log\left(\frac{\$116}{\$105}\right) = \log(\$116) - \log(\$105) = 10\%$$

The value of 10% was attained after multiplying the value with 100. Note that mathematicians writing $\log x$ usually mean $\log ex$, also called $\ln x$. The reason we use log is specified next.

It is important to be consistent with the way that we calculate returns. If we choose to calculate simple returns, we have to do this for all further financial calculations. Similarly, if we decide to calculate log returns, we should use only log returns. There isn't a general rule as to the method we should use, but most econometricians believe that a simple rate of returns is preferable when you have to deal with multiple assets over the same timeframe and a logarithmic rate of returns is preferable when you make calculations about a single asset over a period of time.

We should always remember the timeframe for which we've calculated a rate of return. In our example, we have calculated a yearly rate of return, because the Apple stock was held for a year and then sold. Typically, investors use daily, monthly, quarterly, or yearly returns. The most popular timeframe is annually. We can easily convert daily, monthly, and quarterly returns to yearly returns using the following basic formula:

$$\text{annual return} = [(\text{daily return} + 1)^{365}] * 100$$

 Converts daily, monthly, and quarterly returns to yearly

In the next section, we will calculate the rate of returns of a security using Python.

Calculating the rate of return of a security in Python using simple returns

In the previous section, we learned how to calculate the rate of return of a security using simple rate of returns and logarithmic rate of returns from a theoretical perspective. Let's now move on and see how we can calculate the simple rate of return for a security using Python. In this section, we are going to use the stock prices of Microsoft (MSFT). This dataset was downloaded from Yahoo! Finance for the period between 2000 and 2017. First, we will import the necessary libraries such as `numpy`, `pandas`, and `matplotlib`, as shown in the following code block:

```
| In [1]:  
|     import numpy as np  
|     import pandas as pd  
|     import matplotlib.pyplot as plt
```

Then, we are going to read the Microsoft stock prices dataset. The dataset is available in the GitHub repository of this chapter: <https://github.com/PacktPublishing/Hands-on-Python-for-Finance/tree/master/Chapter%204>.

Let's read the dataset, as shown in the following code block:

```
| In [3]:  
|     MSFT = pd.read_csv('MSFT_stock.csv', index_col = 'Date')
```

The following code helps us to see the top five records of the `MSFT` dataframe:

```
| In [4]:  
|     MSFT.head()
```

The output is as follows:

Date	Open	High	Low	Close	Adj Close	Volume
1999-12-31	58.75000	58.8750	58.1250	58.37500	38.771053	12517600

2000-01-03	58.68750	59.3125	56.0000	58.28125	38.708794	53228400
2000-01-04	56.78125	58.5625	56.1250	56.31250	37.401215	54119000
2000-01-05	55.56250	58.1875	54.6875	56.90625	37.795563	64059600
2000-01-06	56.09375	56.9375	54.1875	55.00000	36.529484	54976600

The following code helps us to see the tail, or the last five records, of the `MSFT` dataframe:

```
| In [5]:  
| MSFT.tail()
```

The output is as follows:

Date	Open	High	Low	Adj	Close	Volu
2017-12-06	81.550003	83.139999	81.430000	82.779999	82.779999	2616
2017-12-07	82.540001	82.800003	82.000000	82.489998	82.489998	2318
2017-12-08	83.629997	84.580002	83.330002	84.160004	84.160004	2448
2017-12-11	84.290001	85.370003	84.120003	85.230003	85.230003	2012
2017-12-12	85.309998	86.050003	85.080002	85.529999	85.529999	7805

As discussed in the previous section, the simple rate of returns of the security is as follows:

$$\frac{\text{End Price} - \text{Beginning Price}}{\text{Beginning Price}} = \frac{\text{End Price}}{\text{Beginning Price}} - 1$$

The Python code for calculating the simple rate of returns is shown here:

```
In [6]:
MSFT['simple_return'] = (MSFT['Close'] / MSFT['Close'].shift(1)) - 1
print (MSFT['simple_return'].head(10))

Out[6]:
Date
1999-12-31      NaN
2000-01-03   -0.001606
2000-01-04   -0.033780
2000-01-05    0.010544
2000-01-06   -0.033498
2000-01-07    0.013068
2000-01-10    0.007291
2000-01-11   -0.025612
2000-01-12   -0.032571
2000-01-13    0.018901
Name: simple_return, dtype: float64 of
```

Here, we are calculating the simple returns on a day-to-day basis using the `shift` function. The value we provided is `1`. The data series shown in the output is as expected: it exhibits the percentage daily change of the closing price. On most days, the number is lower than 1%. Significant movements of a company's stock price are not an everyday occurrence. Note that the first value of the series is **not a number (NaN)**. This makes sense, as there is no lag for our first observation.

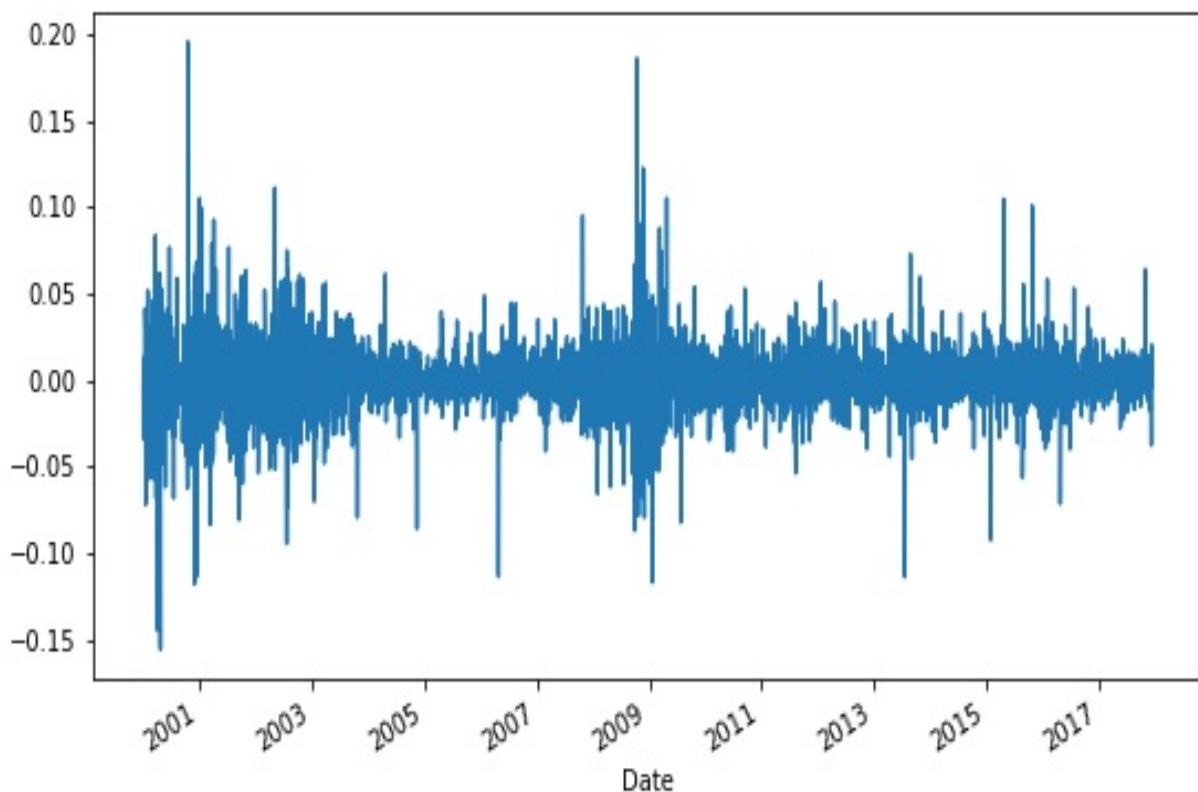
Let's plot these transformations using the `matplotlib` library. Before plotting it, however, we have to convert the index, which is in string format, to datetime, using `pandas`, as follows:

```
In [13]:
MSFT.index=pd.to_datetime(MSFT.index)
```

Then, we will plot the simple returns field calculated for the security, as shown here:

```
In [14]:
MSFT['simple_return'].plot(figsize=(9,5))
plt.show()
```

The output is as follows:



The preceding graph shows the daily simple returns that were observed throughout the defined period. As you can see, the most significant daily observations were mainly negative. In finance, it is common to see negative returns that have a much higher magnitude than positive returns. Usually, positive returns accumulate over time and stock prices increase, but when things do go wrong, stock prices tend to fall very quickly. As you can see from the graph, in 2001, Microsoft stocks experienced a loss of more than 15% in one day.

However, an investor who is interested in buying a stock and holding it in the long run is mainly interested in the average rate of return that the stock will have. For this reason, we calculate the mean return of the `MSFT` throughout the period under analysis. To do this, we will apply a `mean` function that calculates the average daily rate of returns. This function is available in pandas, as in the following code:

```
In [16]:  
MSFT_average_return=MSFT['simple_return'].mean()  
MSFT_average_return  
Out[16]:
```

```
| 0.00027
```

The output is a really small number, much smaller than 1%, which makes it very difficult to interpret. We might prefer to find the **average annual rate of return**. The data that we have extracted, however, is not composed of 365 days observations per year. It excludes non-trading days, such as Saturdays, Sundays, and bank holidays. The number of trading days actually comes to between 250 and 252 days. For now, let's use the number 250. The next step is to multiply the average daily return by 250, which will give us a close approximation of the actual average return per year. This value will be easier to understand than the previous one. The code to do this is shown as follows:

```
In [17]:  
MSFT_average_return=MSFT['simple_return'].mean()*250  
MSFT_average_return  
  
Out[17]:  
0.06820496450977334
```

We will multiply this value by 100 and add a %, as follows:

```
In [19]:  
print(str(MSFT_average_return * 100)+ '%')  
  
Out[19]:  
6.820496450977334%
```

In this section, we looked at how to estimate the simple market returns of a given stock, how to plot it, and how to turn this into a meaningful value that is easy to understand and interpret.

Calculating a security's rate of return using logarithmic return

Just as a reminder, the logarithm rate of return is given by the following formula:

$$\ln\left(\frac{\text{End Price}}{\text{Beginning Price}}\right)$$

Let's see how we can implement this with Python. We will be using the same libraries as we did for simple returns: `pandas`, `matplotlib`, and `numpy`. We will read the same Microsoft stock prices dataset and apply the preceding log formula using Python, as shown here:

```
In [1]:  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
  
In [2]:  
MSFT = pd.read_csv('MSFT_stock.csv', index_col = 'Date')
```

We apply the logarithmic formula previously mentioned using the `log` function from the `numpy` package and taking the `close` value of a given observation for a certain trading day and dividing it by the `close` value of the day before, as shown here:

```
In [3]:  
MSFT['log_return'] = np.log(MSFT['Close'] / MSFT['Close'].shift(1))  
print (MSFT['log_return'])
```

The output is as follows:

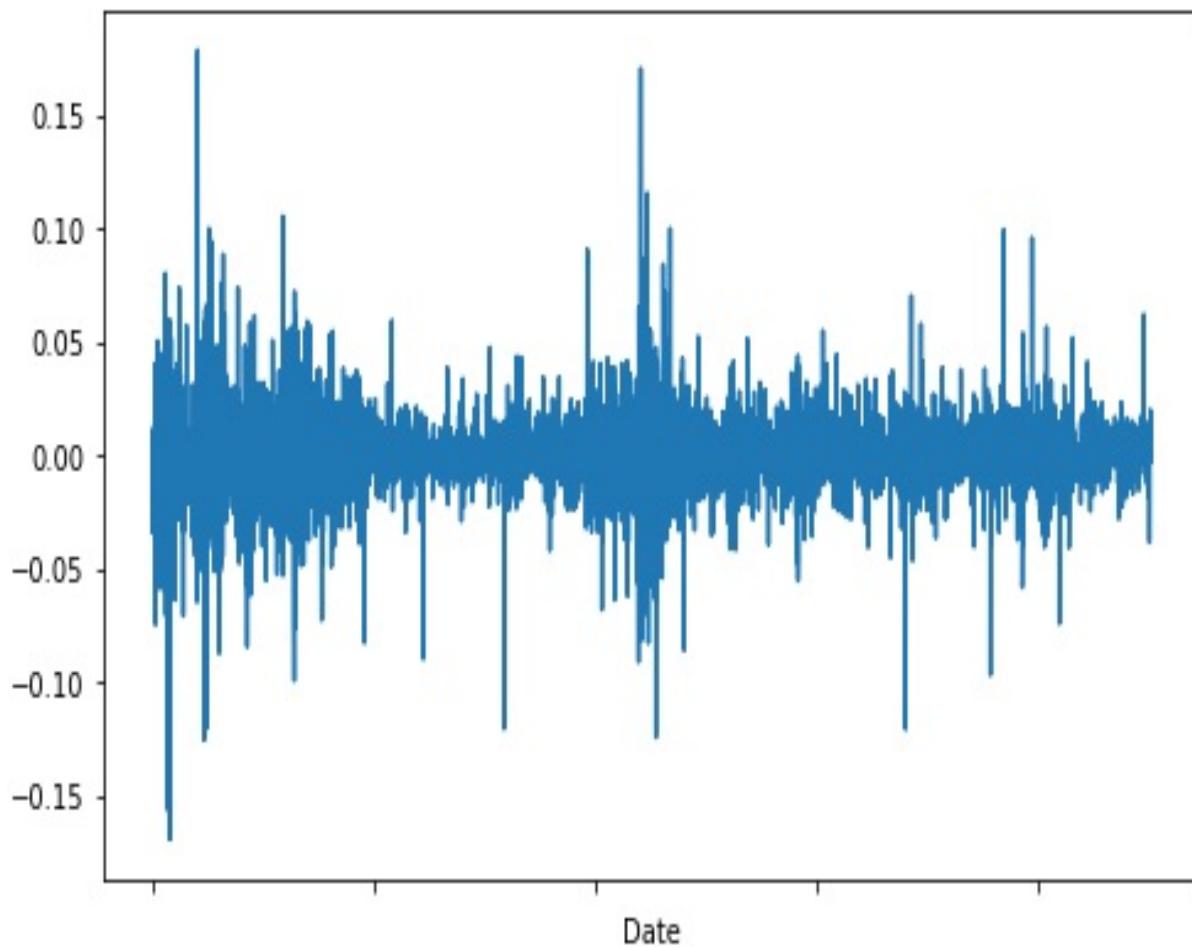
Date	
1999-12-31	NaN
2000-01-03	-0.001607
2000-01-04	-0.034364
2000-01-05	0.010489
2000-01-06	-0.034072
2000-01-07	0.012984
2000-01-10	0.007265
2000-01-11	-0.025946
2000-01-12	-0.033114
2000-01-13	0.018725
2000-01-14	0.040335
2000-01-18	0.026917
2000-01-19	-0.074817
2000-01-20	-0.009390
2000-01-21	-0.021455
2000-01-24	-0.024391
2000-01-25	0.015314
2000-01-26	-0.034006
2000-01-27	-0.006309
2000-01-28	-0.005076
2000-01-31	-0.003824
2000-02-01	0.050431
2000-02-02	-0.020860
2000-02-03	0.027516
2000-02-04	0.027953
2000-02-07	0.000586
2000-02-08	0.030594

From this, we obtain a series and, again, most of the values are less than 1%.
Let's plot the graph:

```
In [4]:  
MSFT.index=pd.to_datetime(MSFT.index)  
MSFT['log_return'].plot(figsize=(8, 5))
```

```
| plt.show()
```

The output is as follows:



When we plot the newly obtained data on a graph, we see that the log rate of return ostensibly resembles the simple rate of return. However, when we calculate the average daily and annual log returns, we obtain a percentage that is significantly smaller than the one we obtained for simple returns.

Let's calculate the average mean and the annual log return:

```
In [6]:  
log_return_d = MSFT['log_return'].mean()  
log_return_d  
  
Out[6]:  
8.45835836828664e-05
```

The following code is used to calculate the annual log returns, considering 250

as the number of working days in a year:

```
In [8]:  
log_return_a = MSFT['log_return'].mean() * 250  
log_return_a  
  
Out[8]:  
0.0211458959207166
```

The following code is used to convert the annual log to a percentage by multiplying it by 100:

```
In[9]:  
print(str(log_return_annually * 100) + '%')  
  
Out[9]:  
2.11458959207166%
```

To conclude, it is preferable to use simple returns when we calculate the returns of multiple securities in the same period. On the other hand, log returns are a better choice when we have only one security and we want to calculate the returns over multiple time periods.

Measuring the risk of a security

Investors prefer good rates of return and they don't like risk uncertainty, because this means that there is a chance that they'll lose a large portion of their money. As mentioned previously, risk and return are the two most important dimensions in investment decision making. It is, therefore, easy to understand why we should spend a significant portion of time learning how to measure and forecast security risks. Let's think about how we can define risk for an investor. If you invest \$1,000 of your money in a stock that's trading on the stock exchange and you know that on average this stock earns 15%, you might want to know what numbers the average is calculated from. Let's consider two different scenarios:

- In one year, the stock earned 14%. In the following year, it earned 16%. In the final two years observed, it earned 13% and 17% respectively.
- In one year, the stock earned 50%. In the following two years, it earned -20%. In the final year observed, it earned 50%.

There is a big difference between these two sets of data. In the first case, you can be certain that your money will earn an amount that is more or less in line with what you expect. Things might be slightly better or slightly worse, but the rate of return will always be between 13% and 17%. In the second set of data, however, although the average return is the same, there is a huge variability from one year to the next. An investor will be unsure about what might happen next. If they invested their money over the second and third observed years, they would have lost 40% of their initial investment.

This shows that *variability* plays an important role in the world of finance. It is the best measure of risk. The stock market is volatile and is likely to surprise investors, both positively and negatively. Investors, however, don't like surprises and are much more sensitive to the possibility of losing their initial investment. Most people prefer to have a good idea about the rate of return they can expect from a security, or a portfolio of securities, and do their best to reduce the risk that they are exposed to. Our goal is to measure the risk faced by investors and try to reduce it as much as possible.

The most commonly used statistical measures, variance, and

standard deviation, can help us a great deal when quantifying the risk associated with the dispersion of the likely outcome.

The variance of a security measures the dispersion of a set of data points around their mean value. It is equal to the following equation:

$$s^2 = \frac{\sum(x - \bar{x})^2}{n - 1}$$

The squared variance is equal to the sum of the squares of the difference between a data point x and the mean divided by the total number of data points minus one. If we take the square root of the variance, we get the standard deviation of this sample of observations:

$$s = \sqrt{s^2}$$

Let's go ahead and calculate the stock's variance and standard deviation using Python in the next section.

Calculating the risk of a security in Python

In this section, we will look at how to calculate a security risk. We will use the `Adjusted Close` column of MSFT and AAPL and we will use the variance and standard variance. First, we will import the same libraries: `numpy`, `pandas`, and `matplotlib`:

```
| In [1]:  
| import numpy as np  
| import pandas as pd  
| import matplotlib.pyplot as plt
```

Then, we will import the dataset using `pandas`:

```
| In [2]:  
| security_data = pd.read_csv('MSFT_AAPL_stock.csv', index_col='Date')
```

After that, we will examine the behavior of the two stocks over the past 17 years by retrieving data from December 31, 1999:

```
| In [3]:  
| security_data.index=pd.to_datetime(security_data.index)  
  
| In [4]:  
| security_data.head()
```

The output is as follows:

	MSFT	AAPL
Date		
1999-12-31	38.965767	3.303425
2000-01-03	38.903194	3.596616
2000-01-04	37.589046	3.293384
2000-01-05	37.985374	3.341579
2000-01-06	36.712940	3.052405

The standard deviation of a company's returns can also be called the **risk** or **volatility**. A stock whose returns show a large deviation from its mean is said to be more volatile.

Let's take a look at which company's stocks are riskier or more volatile. First, we take the logarithmic returns, because we will examine each company separately in the given timeframe. This approach will tell us more about the behavior of the stock:

```
| In [7]:  
| security_returns = np.log(security_data / security_data.shift(1))
```

We can use the head function to see the top 20 records, as shown in the following code:

```
| In [9]:  
| security_returns.head(20)
```

The output is as follows:

	MSFT	AAPL
Date		
1999-12-31	NaN	NaN
2000-01-03	-0.001607	0.085034
2000-01-04	-0.034364	-0.088078
2000-01-05	0.010489	0.014528
2000-01-06	-0.034072	-0.090514
2000-01-07	0.012983	0.046281
2000-01-10	0.007264	-0.017745
2000-01-11	-0.025946	-0.052505
2000-01-12	-0.033114	-0.061847
2000-01-13	0.018725	0.104069

2000-01-14	0.040335	0.037405
2000-01-18	0.026918	0.034254
2000-01-19	-0.074817	0.024942
2000-01-20	-0.009390	0.063071
2000-01-21	-0.021455	-0.019461
2000-01-24	-0.024391	-0.046547
2000-01-25	0.015314	0.054934
2000-01-26	-0.034007	-0.018545
2000-01-27	-0.006309	-0.001703
2000-01-28	-0.005076	-0.079191

We will store this data in a variable called **security returns**. This newly created variable has two columns, each of which contains the log returns of MSFT and AAPL, respectively. This allows us to obtain the mean and the standard deviation of the two stocks for the dataframe.

Then, we apply a `mean` function from NumPy, as follows:

```
In [10]:
security_returns['MSFT'].mean()

Out[10]:
0.00011508712580239439
```

From this function, we obtain a small number that equals the daily average return. Let's analyze this value by multiplying it by the number of trading days in a year, 250, as follows:

```
In [11]:
security_returns['MSFT'].mean()*250

Out[11]:
0.028771781450598596
```

The output obtained is a value just under 3%, which is the annual rate of return. The same Pythonic logic must be applied to calculate the volatility of the

company's stock. The method that we will be using is called **standard deviation**:

```
In [12]:  
security_returns['MSFT'].std()  
  
Out[12]:  
0.019656948849403607
```

After calculating the standard deviation, we will multiply the output by the square root of 250, which represents the trading days. This square root of 250 is taken because the standard deviation is the square root of the variance:

```
In [18]:  
security_returns['MSFT'].std()*250**0.5  
  
Out[18]:  
0.31080365106770774
```

We then repeat the same procedure for AAPL; we get a lower mean but a higher volatility percentage:

```
In [14]:  
security_returns['AAPL'].mean()  
  
Out[14]:  
0.000864342049190355
```

The following code help us to calculate for the annual rate of return:

```
In [15]:  
security_returns['AAPL'].mean()*250  
  
Out[15]:  
0.21608551229758874  
  
In [16]:  
security_returns['AAPL'].std()  
  
Out[16]:  
0.027761934312069386  
  
In [19]:  
security_returns['AAPL'].std()*250**0.5  
  
Out[19]:  
0.4389547233905951
```

It will be easier for us to interpret the results if we quoted the two means in the two standard deviations next to each other. To do this, we can print the equations of the two annual means and standard deviations, as shown in the following code:

```
In [20]:  
print(security_returns['MSFT'].mean()*250)  
print(security_returns['AAPL'].mean()*250)
```

The output for this is as follows:

```
Out[20]:  
0.028771781450598596  
0.21608551229758874
```

The following code helps us to see the top five records:

```
In [22]:  
security_returns.head(5)
```

The output is as follows:

	MSFT	AAPL
Date		
1999-12-31	NaN	NaN
2000-01-03	-0.001607	0.085034
2000-01-04	-0.034364	-0.088078
2000-01-05	0.010489	0.014528
2000-01-06	-0.034072	-0.090514

```
In [23]:  
security_returns[['MSFT', 'AAPL']].mean()*250  
  
Out[23]:  
MSFT    0.028772  
AAPL    0.216086  
dtype: float64  
  
In [24]:  
security_returns[['MSFT', 'AAPL']].mean()*250**0.5  
  
Out[24]:  
MSFT    0.001820  
AAPL    0.013666  
dtype: float64
```

Stocks with a higher expected return often attract more buyers. The Microsoft rate of return is slightly higher, but this comes at the expense of a higher volatility.

Portfolio diversification

In this section, we will talk about one of the most important concepts in finance: the relationship between financial securities. It is reasonable to expect the prices of shares in a stock exchange to be influenced by the same factors. The most obvious example of this is the development of the economy. In general, favorable macro-economic conditions facilitate the business of all companies. When people have jobs and money in their pockets, they will spend more. Companies benefit from this as their revenues increase.

In a tough economy, consumers reduce their spending and buy mainly products of primary importance. Because stock prices are determined by profits, whenever the economy is doing well, stock prices are higher. Investors will pay a high price for profitable companies. In times of recession, companies profits are lower and share prices fall significantly. However, different industries are influenced in different ways. Let's consider an example; imagine that we have stocks in two industries: one in the automobile sector and one in the food chain sector. Which do you think will suffer more in an economic crisis? The answer is the stock held in the automobile sector. People can't stop buying food and groceries, but they can easily postpone buying a new car and continue driving an old vehicle. Therefore, the state of the economy impacts different industries in different ways. An investor should always focus on building a portfolio of stocks that involves different sectors.

Let's consider another example to further understand stock portfolios: let's suppose that we own one stock of a tech company, such as Facebook. We then want to buy some shares of a second company. We can either choose LinkedIn or Walmart. For the purposes of this example, let's assume that we expect the same rate of return from both shares. Which one would you choose? The right answer is Walmart, because Walmart operates in a different industry and this gives you protection as an investor. If things don't go well in the internet domain, spending decreases and competition becomes stronger.

Technology changes rapidly and any sector can face problems. If this is the case, we would still have the Walmart share, which would not suffer in the same way.

The same concept is valid for the retail sector: if Walmart isn't performing as well as expected, we'd still have our Facebook share, which operates in a different industry.

By buying the shares of two companies operating in the same industry, our portfolio will be exposed to an excessive risk for the same level of expected return. There is clearly a relationship between the prices of different companies. It is very important to understand what causes this relationship and how to use this measurement to build optimal investment portfolios.

Covariance and correlation

Now that we know that it is reasonable to expect a relationship between the returns of different stocks, we have to learn how to quantify this relationship. Let's take a look at an example about the factors that determine the price of a house. One of the main factors is the size of the house: the larger it is, the more expensive it is. There is clearly a relationship between these two factors. Statisticians uses the term **correlation** to measure this relationship. Usually, this correlation output of the calculation lies in the interval from -1 to +1.

To understand this concept better, let's take a look at the formula that allows us to calculate the covariance between two variables:

$$\sigma_{xy} = \frac{(x - \bar{x}) * (y - \bar{y})}{n - 1}$$

Here, \bar{x} is the mean of factor x (size) and \bar{y} is the mean of factor y (price). n is the number of records, which contains the combination of x and y . The correlation coefficient measures the relationship between the two variables.

The output of the covariance specifies the following information:

- If the covariance is more than zero, the two variables move in the same direction.
- If the covariance is less than zero, the two variables move in the opposite direction.
- If the covariance is equal to zero, the two variables are independent.

Let's now discuss correlation; correlation is the concept, and correlation coefficient is the measure of correlation. It has values that range from -1 to 1, where -1 or 1 indicate a perfect linear association and 0 indicates no linear relationship. The relationship between the two variables, or the two factors, becomes easy to interpret. Let's consider the previous example again. While it is true that the price of a house is directly proportionate to the size of the house, in

reality, several variables have an impact, such as their location and their year of construction. There is a similar story when we compare company shares. There are several variables that determine share prices, such as industry growth, revenue growth, profitability, regulatory environment, and growth perspective. The more similar the context in which two companies operate, the more correlation there will be between their share prices, as they will be influenced by the same, or similar, factors.

Usually, correlation is given by the following formula:

$$\rho_{xy} = \frac{(x - \bar{x}) * (y - \bar{y})}{\sigma_x \sigma_y}$$

$$\rho_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

There are three different types of correlation output, as follows:

- **Perfect correlation:** A correlation of a value of one, also known as perfect positive correlation, means that the entire variability of the second variable is explained by the first variable. For example, house prices are directly proportionate to house size if we are considering house size as the only parameter affecting house price.
- **No correlation:** A correlation of a value of zero between two variables means they are independent from each other. For example, we will expect a correlation of zero between the price of coffee in Brazil and the price of houses in the USA. These two variables have nothing in common.
- **Negative correlation:** We can either have a perfect negative correlation of a value of -1, or, as is much likelier, an imperfect negative correlation of a value between -1 and 0. For example, there might be a negative correlation between the profits of a company producing ice cream and a company selling umbrellas. More ice cream is sold when the weather is good, while more umbrellas are sold when it's rainy. This is an example of a situation where the prices of two companies are influenced by the same variable, but the variable impacts their businesses in a different way.

Using the preceding formula, we can calculate a correlation coefficient manually. In the next section, we will learn how to calculate the covariance and

the correlation between the prices of AAPL and MSFT quickly and easily using Python.

Calculating the covariance and correlation

Let's take a look at how we can apply this knowledge in Python. Now that you know what covariance is, you should be able to grasp the concept of a covariance matrix more easily. A covariance matrix is a representation of the way that two or more variables relate to each other, as shown in the following formula:

$$\text{Covariance Matrix} : \Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1I} \\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2I} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{I1} & \sigma_{12} & \cdots & \sigma_I^2 \end{bmatrix}$$

The covariance between a variable and itself is the variance of that variable. Along the main diagonal, we have the variances of different variables. The rest of the table should be filled with the covariances between them. In our case, the variables are the prices of two stocks. We will therefore expect a two-by-two covariance matrix with the variances of each stock along the main diagonal, and the covariance between the two stocks displayed in the other two cells. In Python, there is no need to do mathematical calculations manually. The `var()` method calculates the variance of the object for us. We will use this function to calculate the variances of the stock prices of Microsoft and Apple.

The code is as follows:

```
In [8]:  
MSFT = security_returns['MSFT'].var()  
MSFT  
  
Out[8]:  
0.00038639563806806983
```

The following code helps us to calculate the variance of the Apple stock:

```
In [9]:
```

```

AAPL = security_returns['AAPL'].var()
AAPL

Out[9]:
0.0007707249967476555

```

After that, we make these values annual, as shown in the following code block:

```

In [10]:
MSFT = security_returns['MSFT'].var() * 250
MSFT

Out[10]:
0.09659890951701745

In [11]:
AAPL = security_returns['AAPL'].var()*250
AAPL

Out[11]:
0.19268124918691387

```

In order to compute the pairwise covariance of the column of the stock price, we will use the `cov()` function, which is available in the `pandas` library; the code for this is as follows:

```

In [12]:
cov_matrix = security_returns.cov()
cov_matrix

```

The output is as follows:

MSFT	AAPL	
MSFT	0.000386	0.000218
AAPL	0.000218	0.000771

Next, we analyze the result to obtain the annual covariance matrix, as shown here:

```

In [13]:
cov_matrix = security_returns.cov()*250
cov_matrix

```

The output is as follows:

	MSFT	AAPL
--	-------------	-------------

MSFT	0.096599	0.054592
AAPL	0.054592	0.192681

Let's examine this matrix cell by cell, starting from the top-left corner. The top-left corner has the same values as that of the MSFT annualized variance value, which was computed by the `var()` function mentioned previously. The `cov()` method is useful because it allows us to obtain the other numbers easily.

Let's now calculate the correlation with the help of the `corr()` method, which is available in pandas, as shown here:

```
In[14]:  
corr_matrix = security_returns.corr()  
corr_matrix
```

The output is as follows:

Out[14]:	
	MSFT AAPL
	MSFT 1.000000 0.400153
	AAPL 0.400153 1.000000

The `corr()` computes the correlation of columns pairwise. From the preceding output, we can see that the first diagonal element is 1, because we divide the variances of MSFT and AAPL by the same values. It makes sense that the movement of the stock is perfectly correlated with itself. The other two cells in the correlation table contain the same number. This tells us that the stock returns of the two companies are weakly correlated. Be very careful here: this is not the correlation between the price of the two equities. This often creates confusion. When carrying out financial analysis, the correlation between prices and the correlation between returns may show different values. They usually have different implications and it is important to respect this distinction. The correlation between the returns, which is the same as the correlation between the rate of returns, reflects the dependence between prices at different times and focuses on the returns of your portfolio rather than on the stock price levels.

Summary

In this chapter, we have looked at a wide range of concepts regarding measuring investment risks. We started by exploring the various sources of a financial dataset and looked at where we can download datasets from. We then learned about both the simple rate of returns for a security and the logarithmic rate of returns, and saw how we can compute these values using Python on the dataset of the adjusted column of the stocks of Microsoft and Apple. After that, we looked at how to calculate the risk of a security both theoretically and practically, using Python. Finally, we explored the concept of portfolio diversification and discussed the concepts of covariance and correlation to find the variance between the stock prices of the two companies, Microsoft and Apple.

In the next chapter, we will be discussing portfolio allocation and Markowitz portfolio optimization, which will help us to allocate and optimize our portfolios efficiently.

Portfolio Allocation and Markowitz Portfolio Optimization

This chapter deals primarily with Markowitz portfolio optimization. In 1952, Harry Markowitz published a paper that turned out to be one of the most significant academic breakthroughs ever seen in the world of finance. His work changed the way financiers look at their portfolios and formalized methods previously suggested by some practitioners.

In this chapter, we are going to discuss portfolio allocation. We are also going to consider various important terms that will be useful when we are allocating assets in a portfolio using Python. Also, we will be implementing Markowitz portfolio optimization with Python.

The following are the topics that we will cover in this chapter:

- Sharpe ratio
- Portfolio allocation
- Portfolio optimization
- Markowitz portfolio optimization theory
- Obtaining the efficient frontier in Python – part 1
- Obtaining the efficient frontier in Python – part 2
- Obtaining the efficient frontier in Python – part 3

Technical requirements

In this chapter, we will be using Jupyter Notebook for coding purposes. We will be using both the `pandas` library and the `quandl` library. In order to install the `quandl` library, open the Anaconda Command Prompt and type the following command:

```
| conda install -c anaconda quandl
```

The GitHub repository for this chapter can be found at <https://github.com/PacktPublishing/Hands-on-Python-for-Finance/tree/master/Chapter%205>.

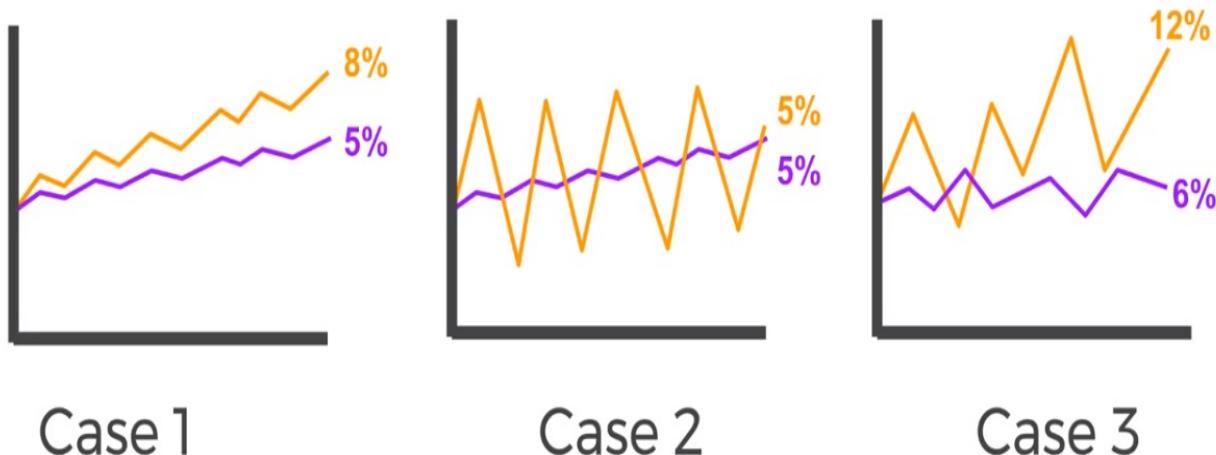
Portfolio allocation and the Sharpe ratio

Usually, when doing algorithmic trading, you're not just dealing with a single stock, you're dealing with a portfolio. A **portfolio** is a set of allocations in a variety of securities. As an example, let's consider a scenario where we have 20% of our portfolio in Apple stocks, 30% in Walmart stocks, and 50% in Google stocks. These percentages are defined as weights. Let's review some of the key statistics related to a stock portfolio:

- **Daily Returns:** The percent returned from one day to the next for each stock
- **Cumulative return:** The amount returned after an entire time period
- **Average daily returns:** The mean of the daily returns
- **Standard daily returns:** The standard deviation of the daily returns

Another critical statistics measure is the Sharpe ratio, which is named after William Sharpe. Let's take a look at what the Sharpe ratio is and why it is necessary.

Imagine we are comparing two portfolios in three different cases. In each case, we are trying to decide which is better:



In the preceding diagram, we have two different portfolios. If we look at **Case 1**, the x axis represents the time period and the y axis represents the total price of the portfolio, with the percentage indicating how much you actually get in return. In the first case, we can see that the top line returns 8%, while the bottom line returns 5%. If we were to decide which portfolio was better in **Case 1**, we would say the upper line, as it has a higher rate of returns.

In **Case 2**, both portfolios return the same value, which is 5%. If you were to choose which portfolio was better in **Case 2**, you might opt for the flatter line, because this one is a lot more stable and has less volatility. The other line from **Case 2** is much more volatile, which means it has a higher risk. Similarly, in **Case 3**, we might say that the stock with a 6% return is more stable, while the stock that has a 12% return is more volatile. We have to decide whether to go for the riskier stock, which will potentially bring higher gains, or to go for the stock with the lower returns but with lower volatility.

The Sharpe ratio allows us to use math to quantify relationships between the mean daily returns and the volatility (or the standard deviation) of the daily returns. It is a measure for calculating a risk-adjusted return and has become the industry standard for this calculation. It was developed by Nobel Laureate William F. Sharpe. The mathematical formula of the Sharpe ratio is as follows:

$$S = \left(\frac{R_p - R_f}{\sigma_p} \right)$$

Here, R_p is the expected portfolio return, R_f is the risk-free return, and σ_p is the standard deviation of the portfolio.

Portfolio allocation and the Sharpe ratio with code

Let's now take a look at a few examples using Python. In this section, we are going to use the stock datasets of a range of tech companies, including Apple, Cisco, IBM, and Amazon. First, we will import the necessary libraries, `pandas` and `quandl`, which will help us to retrieve the stock data of a company from an API.

Quandl is a platform for financial, economic, and alternative data that serves investment professionals. Quandl sources data from over 500 publishers. All Quandl's data is accessible through an API. This is possible through packages for multiple programming languages, including R, Python, Matlab, Maple, and Stata.

```
In[4]:  
import pandas as pd  
import quandl
```

We will retrieve the past five years' worth of stock data for the four companies using the `quandl` API, as follows:

```
In[7]:  
start_date = pd.to_datetime('2013-01-01')  
end_date = pd.to_datetime('2018-01-01')  
  
In[8]:  
aapl_stock = quandl.get('WIKI/AAPL.11', start_date=start_date, end_date=end_date)  
cisco_stock = quandl.get('WIKI/CSCO.11', start_date=start_date, end_date=end_date)  
ibm_stock = quandl.get('WIKI/IBM.11', start_date=start_date, end_date=end_date)  
amzn_stock = quandl.get('WIKI/AMZN.11', start_date=start_date, end_date=end_date)
```

We have to use the `quandl.get()` function to extract the stock information. After extracting the information, we can save it as a CSV file, as follows:

```
In[11]:  
aapl_stock.to_csv('AAPL_CLOSE')  
cisco_stock.to_csv('CISCO_CLOSE')  
ibm_stock.to_csv('IBM_CLOSE')  
amzn_stock.to_csv('AMZN_CLOSE')
```



Please refer the documentation of the `quandl` library for more information: <https://www.quandl.com/>

We also have the CSV files that are provided in the GitHub repository. We can use this data and read the CSV file if Quandl doesn't work due to firewall issues:

```
aapl_stock = pd.read_csv('AAPL_CLOSE', index_col='Date', parse_dates=True)
cisco_stock = pd.read_csv('CISCO_CLOSE', index_col='Date', parse_dates=True)
ibm_stock = pd.read_csv('IBM_CLOSE', index_col='Date', parse_dates=True)
amzn_stock = pd.read_csv('AMZN_CLOSE', index_col='Date', parse_dates=True)
```

First, we will review some important metrics. Let's see whether we can get a value for the cumulative daily returns. We will do this with respect to all the stocks that we have extracted, as follows:

```
In[14]:
for df_stock in (aapl_stock,cisco_stock,ibm_stock,amzn_stock):
    df_stock['Normalize Return'] = df_stock['Adj. Close']/df_stock.iloc[0]['Adj. Close']

In[15]:
aapl_stock.head()
```

The output of the preceding code is as follows:

Out[15]:

	Adj. Close	Normalize Return
Date		
2013-01-02	71.195748	1.000000
2013-01-03	70.296565	0.987370
2013-01-04	68.338996	0.959875
2013-01-07	67.937002	0.954228
2013-01-08	68.119845	0.956797

The `Normalize Return` column created gives us the cumulative daily returns. Let's now consider a portfolio that is allocated to the four companies in the following way:

- 30% is allocated to Apple

- 20% is allocated to Google/Alphabet
- 40% in Amazon
- 10% is allocated to IBM

We will multiply the `Normalize Return` column with these allocations, as follows:

```
In[21]:  
for df_stock,allocation in zip([aapl_stock,cisco_stock,ibm_stock,amzn_stock],[.3,.2,.4,.1]):  
    df_stock['Allocation'] = df_stock['Normalize Return']*allocation
```

A new `Allocation` column has been created with respect to the allocation of the portfolio.

The following code is used to display the top five records of the `aapl_stock` dataframe:

```
In[22]:  
aapl_stock.head()
```

The preceding code produces the following output:

Out[22]:

Date	Adj. Close	Normalize Return	Allocation
2013-01-02	71.195748	1.000000	0.300000
2013-01-03	70.296565	0.987370	0.296211
2013-01-04	68.338996	0.959875	0.287962
2013-01-07	67.937002	0.954228	0.286269
2013-01-08	68.119845	0.956797	0.287039

Now, suppose we have invested \$100,000 in this portfolio. We can take this into account as follows:

```
In[23]:  
for df_stock in [aapl_stock,cisco_stock,ibm_stock,amzn_stock]:  
    df_stock['Position Values'] = df_stock['Allocation']*100000
```

The following code helps us to see the top five records of the `aapl_stock` dataframe:

```
| In[24]:  
| aapl_stock.head()
```

The preceding code produces the following output:

Out[24]:

	Date	Adj. Close	Normalize Return	Allocation	Position Values
	2013-01-02	71.195748	1.000000	0.300000	30000.000000
	2013-01-03	70.296565	0.987370	0.296211	29621.108136
	2013-01-04	68.338996	0.959875	0.287962	28796.240643
	2013-01-07	67.937002	0.954228	0.286269	28626.850992
	2013-01-08	68.119845	0.956797	0.287039	28703.895962

As you can see in the preceding output, `Position values` specify how much money we allocate to the stocks. You can also see that on January 2, 2013, we initially invested \$30,000. The very next day, this value went down to \$29,621.10 and we continued to drop until the 8th. We can actually see how much money is in our portfolio as the days go by.

Let's now create a larger portfolio dataframe that essentially has all of these positioned values for all of our stocks.

We are going to concatenate all the `Position values` of all the stocks, as follows:

```
| In[25]:  
| portfolio_val = pd.concat([aapl_stock['Position Values'], cisco_stock['Position Values'],
```

The concatenated results are as follows:

```
| In[26]:  
| portfolio_val
```

The output of the preceding code is as follows:

Out[10]:

Date	Position Values	Position Values	Position Values	Position Values
2013-01-02	30000.000000	20000.000000	40000.000000	10000.000000
2013-01-03	29621.108136	20108.161259	39779.984721	10045.470444
2013-01-04	28796.240643	20139.528024	39519.225872	10071.509075
2013-01-07	28626.850992	19951.130777	39346.065699	10433.298356
2013-01-08	28703.895962	19970.501475	39291.061879	10352.493102

From the preceding output, we can see how our \$100,000 is distributed in this portfolio and how this changes over time. We can see how much money we have in our entire portfolio. Currently, all the columns are named `Position values`. Let's rename the columns to give us a clear understanding of which column relates to which stock:

In[27]:

```
portfolio_val.columns = ['AAPL Pos', 'CISCO Pos', 'IBM Pos', 'AMZN Pos']
```

The following code helps us to see the top five records of the `portfolio_val` dataframe:

In[28]:

```
portfolio_val.head()
```

The output of the preceding code is as follows:

Out[28]:

	AAPL Pos	CISCO Pos	IBM Pos	AMZN Pos
Date				
2013-01-02	30000.000000	20000.000000	40000.000000	10000.000000
2013-01-03	29621.108136	20108.161259	39779.984721	10045.470444
2013-01-04	28796.240643	20139.528024	39519.225872	10071.509075
2013-01-07	28626.850992	19951.130777	39346.065699	10433.298356
2013-01-08	28703.895962	19970.501475	39291.061879	10352.493102

Let's calculate the sum of all the position values on a day-to-day basis and plot this on a graph. This will give us an idea about whether our total money will increase in the future. To do this, we will create a new column, `Total Pos`, which will have the total sum of all the portfolios on a day-to-day basis.

Let's begin by importing the libraries, as follows:

```
In[29]:  
import matplotlib.pyplot as plt  
%matplotlib inline
```

The following code is used to create a new column, `Total Pos`, which will have the total sum of all the portfolios on a day-to-day basis:

```
In[30]:  
portfolio_val['Total Pos'] = portfolio_val.sum(axis=1)
```

The following code helps us to see the top five records of the portfolio dataframe:

```
In[31]:  
portfolio_val.head()
```

The output of the preceding code is as follows:

Out[31]:

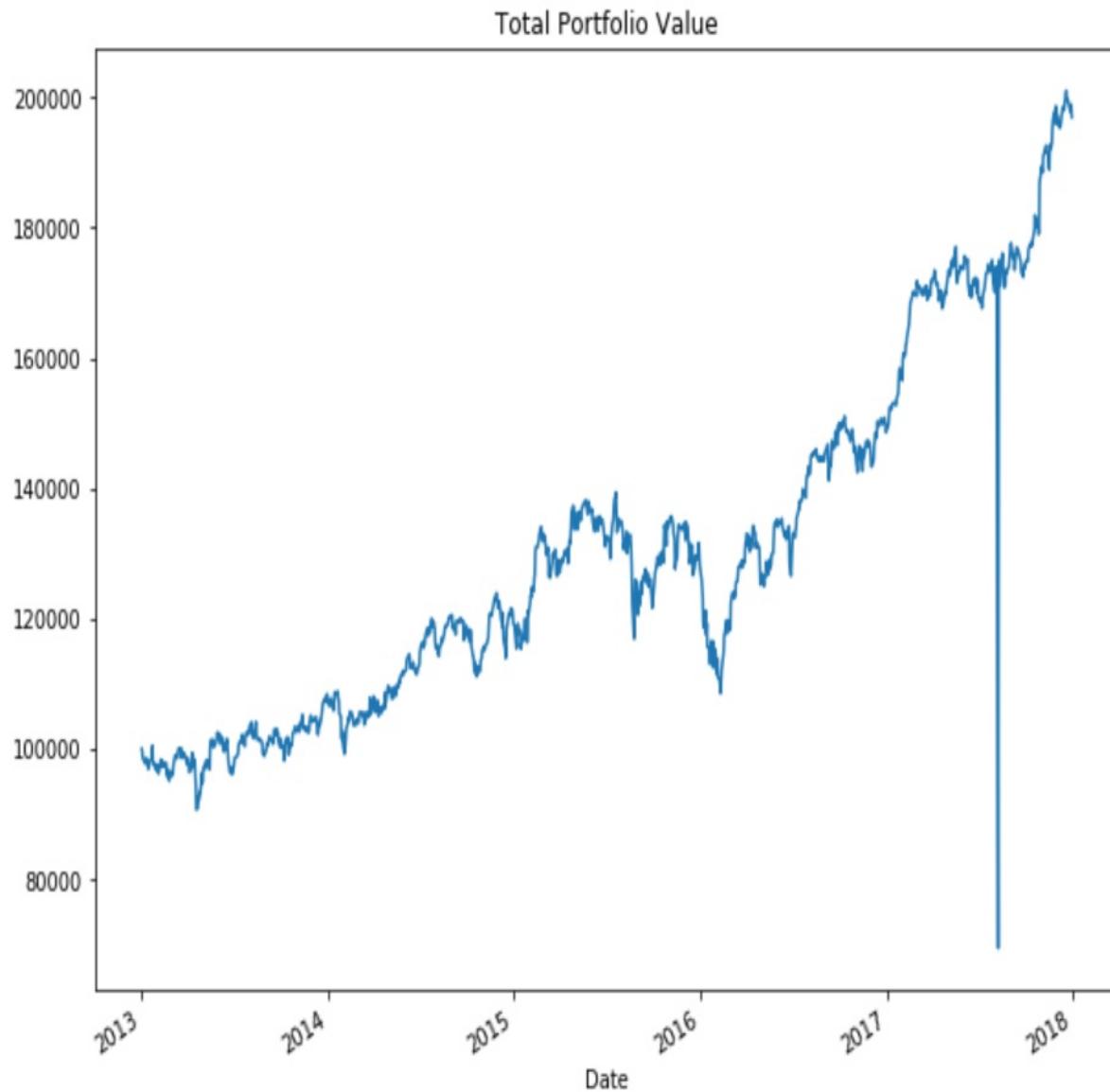
	AAPL Pos	CISCO Pos	IBM Pos	AMZN Pos	Total Pos
Date					
2013-01-02	30000.000000	20000.000000	40000.000000	10000.000000	100000.000000
2013-01-03	29621.108136	20108.161259	39779.984721	10045.470444	99554.724560
2013-01-04	28796.240643	20139.528024	39519.225872	10071.509075	98526.503613
2013-01-07	28626.850992	19951.130777	39346.065699	10433.298356	98357.345824
2013-01-08	28703.895962	19970.501475	39291.061879	10352.493102	98317.952418

Let's plot a figure using Matplotlib for the Total Pos column and see the changes day by day:

In[32]:

```
portfolio_val['Total Pos'].plot(figsize=(10,8))plt.title('Total Portfolio Value')
```

The output is shown in the following screenshot:

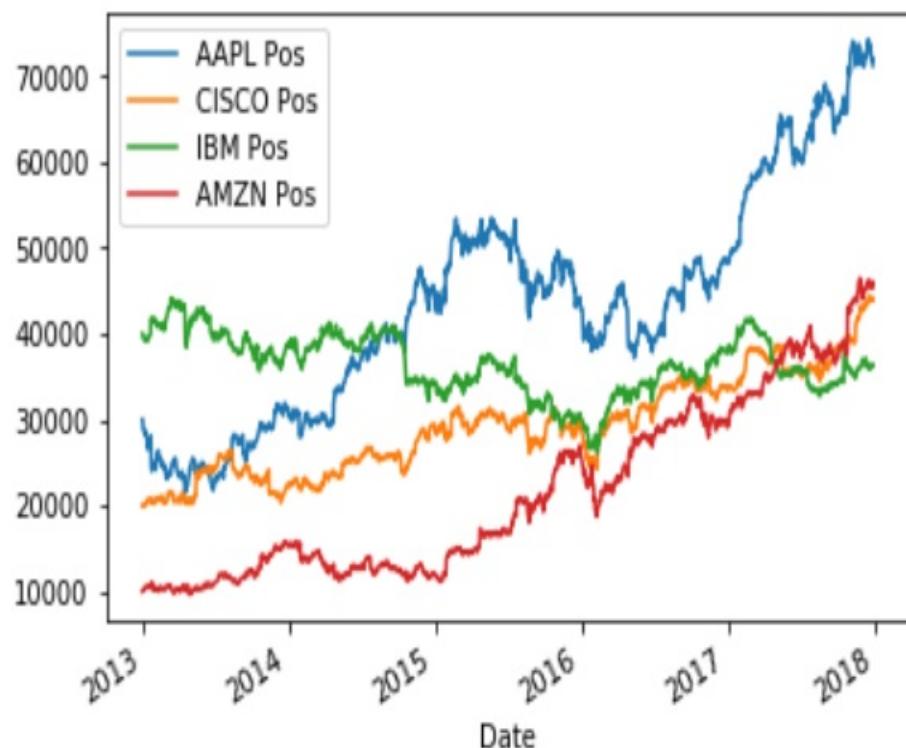


The preceding graph gives us the total sum of all the money invested in the portfolio and the changes that happen on a daily basis. If we need a better plotted diagram to further understand the changes that each company undergoes on a daily basis, we can create one, as follows:

```
| In[33]:  
| portfolio_val.drop('Total Pos',axis=1).plot(kind='line')
```

The output is shown in the following screenshot:

Out[33]: <matplotlib.axes._subplots.AxesSubplot at 0x19b1cba98d0>



Portfolio optimization

In this section, we are going to discuss portfolio optimization. We now have a useful metric, the Sharpe ratio, to evaluate portfolio allocations against each other. Let's consider how to optimize these portfolio holdings. One way to do this is by checking a bunch of random allocations and seeing which one has the best Sharpe ratio. We have powerful computers and we can use NumPy to generate random numbers, so this technique, sometimes known as **Monte Carlo Simulation**, works well. In a Monte Carlo Simulation, we randomly assign a weight to each security in our portfolio. We then calculate its mean daily return and the standard deviation of its daily return. This allows us to calculate the Sharpe ratio for thousands of randomly-selected allocations. We can then plot the allocations on a chart showing returns versus volatility, which is computed by the Sharpe ratio. However, guessing and checking is not very efficient. Instead of this, we can use math to figure out the optimal Sharpe ratio for any given portfolio.

In order to fully understand the optimization algorithm, we first need to understand a concept known as **minimization**. Let's say we have two simple equations:

- $y = x^2$
- $y = (2-x)^2$

Which value of x will minimize the value of y ?

In the first equation, $x=0$ is the value that minimizes the first equation. In the second equation, $x=2$ is the value that minimizes the equation. This idea of using a minimizer will allow us to build an optimizer. There are mathematical ways of finding this minimum value. Usually, for complex equations such as an optimizer, we can use the `scipy` library to do this math for us.

In the context of our portfolio optimization, we actually want to maximize our Sharpe ratio. Remember that we're trying to figure out which stock allocation will give us the best Sharpe ratio. We want to implement an inverse Sharpe ratio and try to minimize the optimizer, which will end up being the best Sharpe ratio,

just in reverse. In other words, we are raising a minimizer in order to calculate the optimal allocation. We will use SciPy's built-in optimization algorithms to calculate the optimal weight allocation for our portfolio and then we are going to optimize it based on the Sharpe ratio.

In the next section, we are going to walk through these steps using Python. We are also going to look at Markowitz portfolio optimization.

Markowitz portfolio optimization

According to Markowitz, investors shouldn't put all their eggs in one basket. Markowitz proved the existence of an efficient set of portfolios that optimize an investor's return according to the amount of risk they are willing to accept. One of the most important concepts he developed was that investments in multiple securities shouldn't be analyzed separately, but should be considered as a portfolio. The financier must understand how different securities in a portfolio interact with each other. Markowitz' model is based on analyzing various risks and returns and finding the relationship between them using various statistical tools. This analysis is then used to select stocks in a portfolio in an efficient manner, which leads to much more efficient portfolios. Individuals vary widely in their risk tolerance and asset preferences. Their means, expenditures, and investment requirements vary greatly. Because of this, portfolio selection is not a simple choice of any one security or securities, but a subtle selection of the right combination of securities.

In this chapter, we have learned how to measure the relationship between two securities by calculating their covariance. This is precisely what Markowitz suggested. Through a combination of securities with low correlation, investors can optimize their returns without assuming additional risk. Markowitz assumes investors are rational and risk-averse, so they are interested in earning higher returns and prefer avoiding additional risk. This leads to the conclusion that for any level of risk, investors will be interested only in the portfolio with the highest expected return. Usually, when we talk about risk, there are two different types:

- **Systematic risk:** This is the uncertainty that is characteristic of the entire market. Systematic risk is made up of the day-to-day changes in stock prices and is caused by the events that affect all companies.
- **Unsystematic risk:** These are company- or industry-specific risks that can be smoothed out through diversification.

If a portfolio contains two stocks, its risks will be a function of the variances of the two stocks and of the correlation between them. The mathematical formula

of the risks of these stocks is as follows:

$$(w_1 \sigma_1 + w_2 \sigma_2)^2 = w_1^2 \sigma_1^2 + 2w_1 \sigma_1 w_2 \sigma_2 \rho_{12} + w_2^2 \sigma_2^2$$

The equation is described here:

- w_1 : Weight of security 1
- w_2 : Weight of security 2
- σ_1 : Standard deviation of security 1
- σ_2 : Standard deviation of security 2
- ρ_{12} : Correlation between security 1 and 2

Let's now think about a simple example. We're going to look at six different portfolios with two different stocks. The allocations of the two stocks in each portfolio are presented in the following screenshot:

	Share A	Share B
Expected return	7%	9%
Standard deviation	5%	8%
Correlation	30%	

The following quantitative values show the basics weightage along with expected returns in the portfolio:

	Weight A	Weight B	Expected Return	Standard Deviation
Portfolio 1	100%	0%	7.00%	5%
Portfolio 2	80%	20%	7.40%	5%
Portfolio 3	60%	40%	7.80%	5%
Portfolio 4	40%	60%	8.20%	6%
Portfolio 5	20%	80%	8.60%	7%
Portfolio 6	0%	100%	9.00%	8%

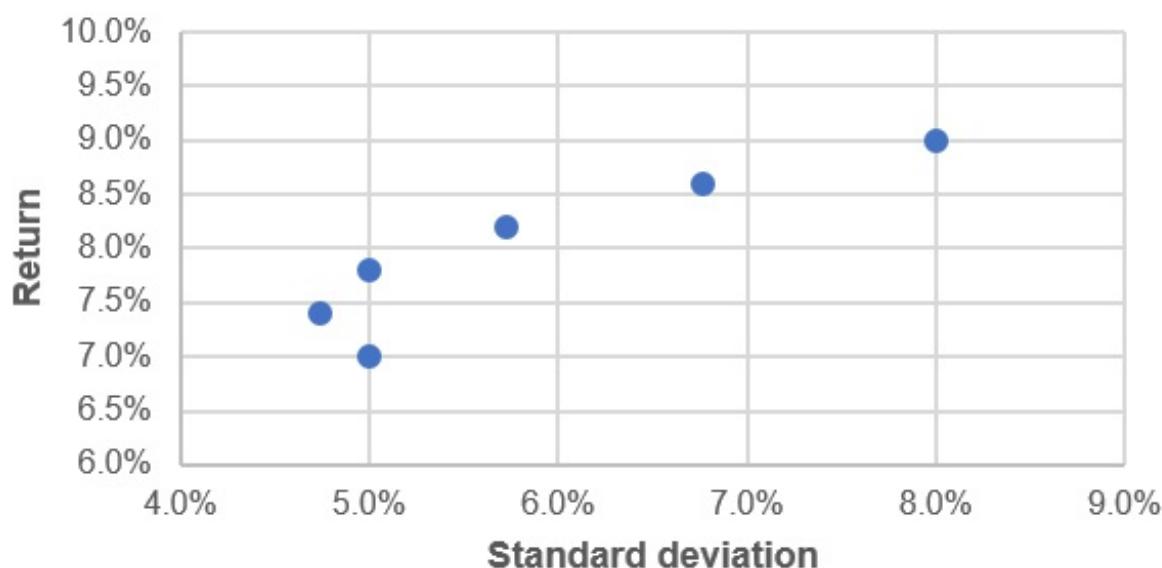
From these figures, we can see that the first portfolio is made up of 100% of stock A and 0% of stock B. The second portfolio is made up of 80% of stock A

and 20% of stock B. The third portfolio is made of 60% of stock A and 40% of stock B. The fourth portfolio is made up of 40 % of stock A and 60% of stock B. The fifth portfolio has 20 % of stock A and 80 % of stock B, while the sixth portfolio is made up of 100% of stock B.

With respect to these allocations, we can calculate the expected return and the standard deviation.

The standard deviation is the risk of the portfolio. It is plotted on the horizontal axis, while the expected returns are plotted on the vertical axis:

Portfolios 1-6: Markowitz Shape



The preceding diagram specifies the typical shape of a Markowitz efficient frontier. This is precisely what Markowitz suggests. There is a set of efficient portfolios that can provide a higher expected rate of return for the same or an even-lower risk. The starting point of the frontier represents the minimum variance portfolio, the lowest risk an investor could bear. The points under the efficient frontier represent inefficient portfolios. We can use this frontier to find an alternative portfolio with a greater expected return for the same level of standard deviation.



To see the calculation, there is an Excel sheet uploaded in the GitHub repository <https://github.com/PacktPublishing/Hands-on-Python-for-Finance/tree/master/Chapter%205>, which includes the graph.

Assumptions of Markowitz's theory

The Markowitz portfolio theory is based on the following assumptions:

- Investors are rational and act in a way to increase a given level of income.
- Financial specialists have free access to up-to-date data and revise this data based on the returns and risk.
- The market sectors are effective and assimilate the data rapidly and flawlessly.
- Financial specialists are risk-averse and attempt to limit risk and amplify return.
- Speculators make choices in light of the expected returns and the standard deviation from the means of these profits.
- Investors would always choose a higher return over a lower return.

Usually, investors select a portfolio that they consider to be efficient. This is the case if no other portfolio or asset offers a return higher than the selected one with the same or a lower level of risk. Usually, investors use portfolio diversification in order to achieve a portfolio that has a low level of risk. This reduces unsystematic risk.

In order to build an efficient set of portfolios, according to Markowitz, we need to consider the following important parameters:

- The expected return
- The variability of the returns, measured by comparing the standard deviation to the mean
- The covariance or variance of one asset's return to the returns of other assets

In general, the higher the expected return, the lower the standard deviation or variance. The lower the correlation, the lower the risk for the investor.

Regardless of the risk of the individual securities in isolation, the total risk of the portfolio of all securities may be lower if the covariance of their returns is negative or negligible. In the next section, we will examine more stocks and find the frontier using Python.

Obtaining the efficient frontier in Python – part 1

In this section, we will learn how to calculate the efficient frontier of a group of portfolios composed of two assets: Walmart and Facebook. These two datasets can be found in the GitHub repository.

To start, import the necessary libraries and read the dataset, as follows:

```
In[3]:  
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
%matplotlib inline  
companies = ['WMT', 'FB']  
df = pd.read_csv('Walmart_FB_2014_2017.csv', index_col='Date')
```

The following code helps us to see the top five records of the dataframe:

```
In[2]:  
df.head()
```

The preceding code generates the following output:

Out[2]:

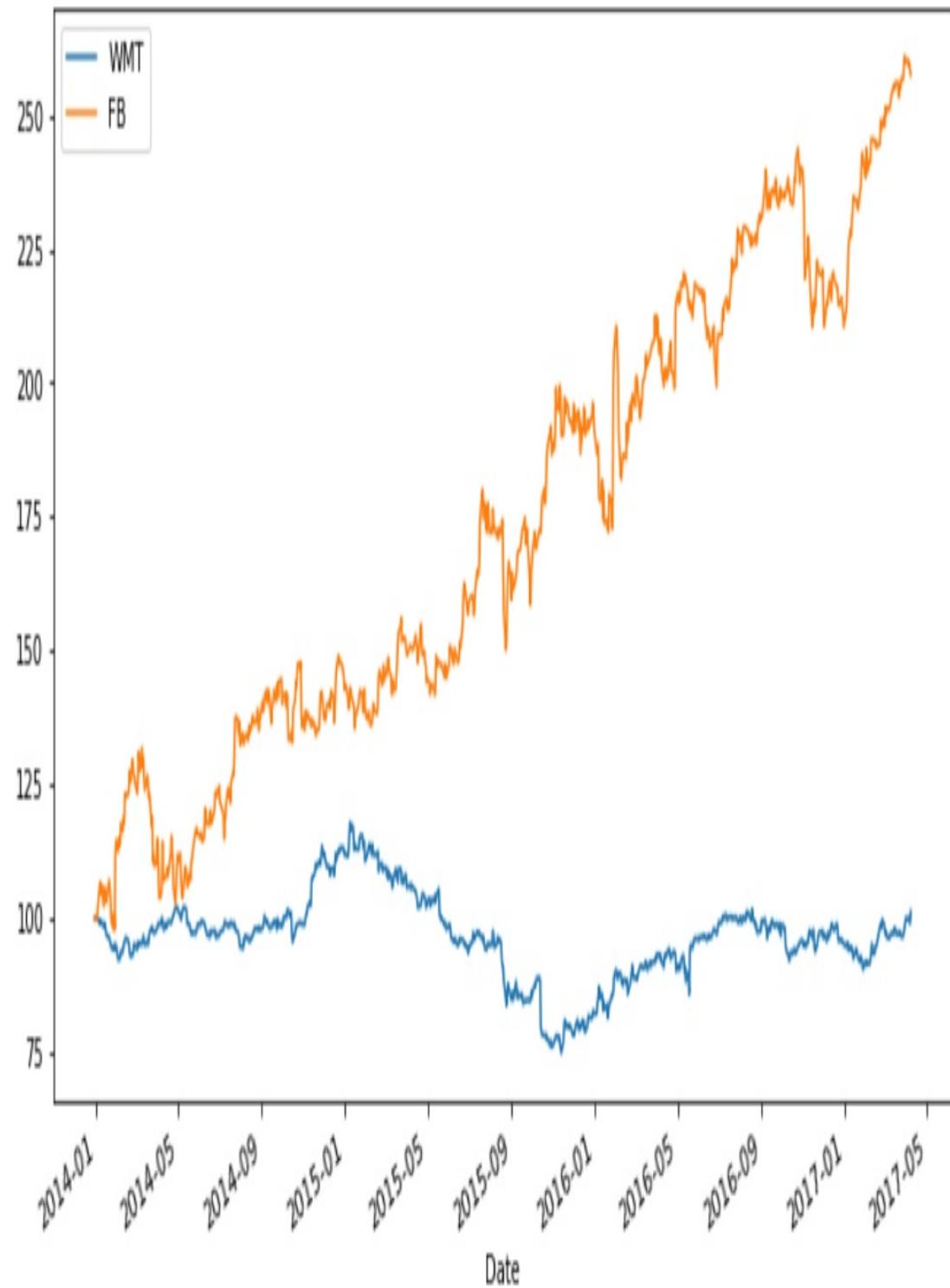
	WMT	FB
Date		
2013-12-31	71.126411	54.650002
2014-01-02	71.325287	54.709999
2014-01-03	71.090279	54.560001
2014-01-06	70.692558	57.200001
2014-01-07	70.909485	57.919998

We will normalize all the data to the value 100 for plotting the portfolio data on a graph to see how the two assets have been performing throughout the period of interest:

```
In[16]:  
df.index=pd.to_datetime(df.index)  
  
In[17]:  
(df / df.iloc[0] * 100).plot(figsize=(11, 6))
```

The output of the preceding code is shown in the following screenshot:

Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x2559dfa4da0>



If we want to obtain an efficient frontier composed of these assets or stocks, we need to compute their logarithmic returns. This was discussed in [Chapter 4, Measuring Investment Risks](#). The code for the logarithmic returns of the preceding assets is as follows:

```
| In[18]:  
| log_returns = np.log(df/ df.shift(1))
```

The following code helps us to read the top five records of the `log_returns` dataset:

```
| In[20]  
| log_returns.head()
```

The output generated is as follows:

Out[20]:

	Date	WMT	FB
	2013-12-31	NaN	NaN
	2014-01-02	0.002792	0.001097
	2014-01-03	-0.003300	-0.002745
	2014-01-06	-0.005610	0.047253
	2014-01-07	0.003064	0.012509

After that, we need to calculate the mean, covariance, and correlation matrices of the Walmart (`WMT`) and Facebook (`FB`) datasets, as follows:

```
| In[21]:  
| #Calculating Mean  
| log_returns.mean() * 250
```

The preceding code generates the following output:

```
Out[21]: WMT      0.003551
          FB       0.287439
          dtype: float64
```

The covariance is calculated using the following code:

```
In[22]:
# Calculating Covariance
log_returns.cov() * 250
```

The preceding code generates the following output:

Out[22]:

	WMT	FB
WMT	0.031544	0.008777
FB	0.008777	0.082786

The correlation is calculated using the following code:

```
In[23]:
# Calculating Correlation
log_returns.corr()
```

The preceding code generates the following output:

Out[23]:

	WMT	FB
WMT	1.000000	0.171746
FB	0.171746	1.000000

Note that the mean and the covariance have been multiplied by 250, because there are 250 working days in a year. From the correlation matrix, we see that both the stocks are averagely correlated. The next step is to address portfolio optimization from a coding perspective. First, we will create a variable that will

carry the number of assets in our portfolio. This variable is important, as we will be using it in our formulas so they can respond to a change in the number of assets that make up the portfolio. It is equal to the number of elements in the asset list. We can obtain this number with the help of the `len` function, as follows:

```
| In[27]:  
| num_assets = len(companies)  
  
| In[28]  
| num_assets
```

The output generated is as follows:

```
| Out[28]:  
| 2
```

Here, we are considering two assets: Facebook and Walmart. Remember that the portfolio does not need to be equally weighted. Create a variable called `weights`. Let this contain as many randomly-generated values as there are assets in your portfolio. Don't forget that these values should be neither smaller than 0 nor equal to or greater than 1.

The next step is to create two random variables using the `random` function available in the `numpy` library:

```
| In[30]:  
| arr = np.random.random(2)  
| arr
```

The output generated is as follows:

```
| Out[30]:  
| array([0.85216426, 0.73407425])
```

The following snippets helps us to add the initialized values in the array:

```
| In[31]:  
| arr[0]+arr[1]
```

The preceding code generates the following output:

```
| Out[31]:
```

```
| 1.5862385086575679
```

By randomly initializing two values, we sometimes find out that the sum of the initialized values is more than one.

To prevent this, we will use the following technique:

```
| In[32]:  
| weights = np.random.random(num_assets)  
| weights /= np.sum(weights)  
| weights
```

The output generated is as follows:

```
| Out[32]:  
| array([0.50354655, 0.49645345])
```

This gives us the weights of the assets. Using this technique, the sum of the values will always be equal to one. These steps are really important for the next section.

Obtaining the efficient frontier in Python – part 2

Let's push the analysis a few steps further. We will now write the formula for the expected portfolio return. This is given by the sum of the weighted average log returns, which is specified as follows:

```
| In[33]:  
| np.sum(weights * log_returns.mean())* 250
```

The output generated is as follows:

```
| Out[33]:  
| 0.1444881608787945
```

The following code will provide us with the expected portfolio variance and the volatility:

- The following code helps us to see the portfolio variance:

```
| In[34]:  
| # Expected Portfolio Variance  
| np.dot(weights.T,np.dot(log_returns.cov()* 250,weights))
```

- The output generated is as follows:

```
| Out[34]:  
| # Expected Portfolio Variance  
| 0.032790341551572906
```

- The following code helps us to see the portfolio volatility:

```
| In[35]:  
| # Expected Portfolio Volatility  
| np.sqrt(np.dot(weights.T,np.dot(log_returns.cov()* 250,weights)))
```

- The output generated is as follows:

```
| Out[35]:
```

```
| 0.18108103586950486
```

We will need the formulas for the return and the volatility in the simulation of the portfolio's mean-variance combinations. We will now create a graph, where 1,000 mean-variance simulations will be plotted. Pay attention here: we are not considering 1,000 different investments composed of different stocks, we are considering 1,000 combinations of the same two assets, Walmart and Facebook. In other words, we are simulating 1,000 combinations of their weight values. Among these 1,000 combinations, we will probably have one portfolio composed of 1% of Walmart stocks and 99% of Facebook stocks and vice versa. The idea is to compare the two and see which one is more efficient. As mentioned, our goal is to create a graph that visualizes the hypothetical portfolio returns versus the volatilities.

We will therefore, need two objects that we can use to store this data. The portfolio returns start as an empty list. We want to fill this with randomly-generated expected returns. We apply a `for` loop 1,000 times for the portfolio volatilities, as shown in the following code:

```
In[36]:  
portfolio_returns=[]  
portfolio_volatilities=[]  
for x in range(1000):  
    weights=np.random.random(num_assets)  
    weights/=np.sum(weights)  
    portfolio_returns.append(np.sum(weights*log_returns.mean())*250)  
    portfolio_volatilities.append(np.sqrt(np.dot(weights.T,np.dot(log_returns.cov(),weights  
portfolio_returns,portfolio_volatilities
```

The output generated is shown in the following screenshot:

```
Out[36]: ([0.23902430167315575,
 0.18944123445616523,
 0.17196869286950392,
 0.006685968278870664,
 0.10990479521109868,
 0.16054894843385578,
 0.05824036464673471,
 0.215181628263505,
 0.2780682371520503,
 0.18729193595657573,
 0.17271604822735367,
 0.1636115801655414,
 0.12794174155284807,
 0.01249290607471217,
 0.07250592422643425,
 0.20991492017689484,
 0.13979037716301046,
 0.09211161986228816,
 0.13765751454800465,
```

In the loop, we are generating two weights, whose sums equal one. We need two weights, because, as we already mentioned, the portfolios are composed of two assets. As the number of assets increases, the number of weights also increases. We are using the `append` method, which will add each newly-generated portfolio return value to the list of portfolio returns. This operation will be repeated for each pass of the loop, 1,000 times until the portfolio returns list accumulates 1,000 observations. We repeat the same procedure for the portfolio volatilities. We will use the `append` method and apply the formula for standard deviation. The output of the preceding code is a list and is hard to manipulate, so we convert it into a NumPy array, as follows:

```
In[37]:
portfolio_returns=np.array(portfolio_returns)
portfolio_volatilities=np.array(portfolio_volatilities)
portfolio_returns,portfolio_volatilities
```

The output generated is shown in the following screenshot:

```
Out[37]: (array([0.2390243 , 0.18944123, 0.17196869, 0.00668597, 0.1099048 ,  
0.16054895, 0.05824036, 0.21518163, 0.27806824, 0.18729194,  
0.17271605, 0.16361158, 0.12794174, 0.01249291, 0.07250592,  
0.20991492, 0.13979038, 0.09211162, 0.13765751, 0.18910214,  
0.08262875, 0.17548627, 0.23482999, 0.04977502, 0.12902758,  
0.07148237, 0.20572807, 0.15286035, 0.26246527, 0.19205318,  
0.15882209, 0.14039463, 0.06006665, 0.20189459, 0.12109478,  
0.07986211, 0.28593727, 0.15261084, 0.20015148, 0.18161396,  
0.10940569, 0.25770711, 0.02861408, 0.21241679, 0.18253775,  
0.19472238, 0.12674706, 0.16290408, 0.19506096, 0.12189415,  
0.17456375, 0.24339411, 0.20517286, 0.20562258, 0.1807871 ,  
0.15753684, 0.04511083, 0.14570486, 0.12292448, 0.13974987,  
0.00435762, 0.20417156, 0.17877689, 0.25769413, 0.04446349,  
0.13165945, 0.06924839, 0.03088512, 0.28478665, 0.12669177,  
0.2450802 , 0.07410808, 0.13628399, 0.11371631, 0.16358597,  
0.17437709, 0.07099607, 0.05968673, 0.27346702, 0.04829892,  
0.13314473, 0.04618645, 0.15861978, 0.1500662 , 0.1096527 ,  
0.05046059, 0.1190782 , 0.09149969, 0.20351993, 0.13591701,  
0.26443873, 0.18947447, 0.18606405, 0.14420315, 0.13771723,  
0.11729746. 0.17779084. 0.13504813. 0.16130019. 0.19122557.)
```

After converting this into an array, we will be able to plot the efficient frontier easily. This will be covered in the next section.

Obtaining the efficient frontier in Python – part 3

We just did most of the important coding. We will now implement the output of the Frontier. First, we will create a dataframe object that contains two columns: one for returns and another one for the respective volatilities. We will call this object `portfolios`. Its data will be composed of a dictionary with the `Return` and `Volatility` keys. These keys will be automatically assigned column names. The data contained in the portfolio returns and portfolio volatilities arrays will constitute the two key-value pairs and will fill the two columns:

```
In[38]:  
portfolios=pd.DataFrame({'Return': portfolio_returns, 'Volatility':portfolio_volatilities})
```

Let's check the head and the tail of the `portfolios` dataframe:

```
In[40]:  
portfolios.head()
```

The preceding code generates the following output:

Out[40]:

	Return	Volatility
0	0.239024	0.015538
1	0.189441	0.013149
2	0.171969	0.012424
3	0.006686	0.011145
4	0.109905	0.010596

The last five records of the portfolios can be seen as follows:

```
| In[41]:  
| portfolios.tail()
```

The output generated is as follows:

Out[41]:

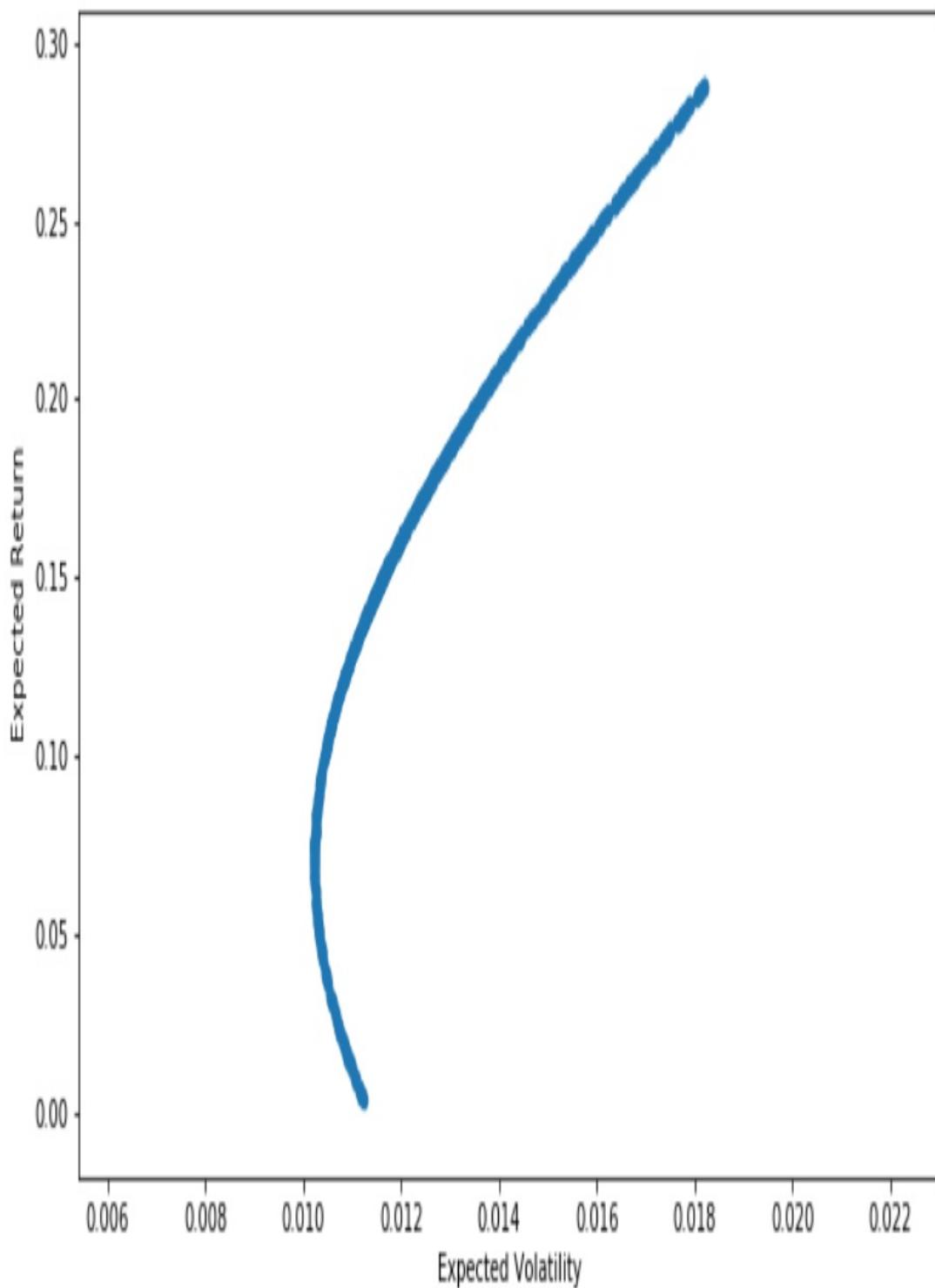
	Return	Volatility
995	0.273539	0.017409
996	0.102647	0.010477
997	0.175654	0.012571
998	0.014974	0.010930
999	0.062073	0.010251

We now need to plot data on a graph with an *x* axis that corresponds to the `Volatility` column and a *y* axis that corresponds to the `Return` column. An important specification is what kind of graph we want to insert. In our case, we will need a scatter plot. We will also adjust the figure size and provide some labels for the *x* axis and the *y* axis:

```
| In[42]:  
| portfolios.plot(x='Volatility',y='Return',kind='scatter',figsize=(10,6))  
| plt.xlabel('Expected Volatility')  
| plt.ylabel('Expected Return')
```

The output is shown in the following screenshot:

Out[42]: Text(0, 0.5, 'Expected Return')



In the preceding output, we see that the curve resembles the Markowitz frontier curve. Go through the notebook file to solidify your understanding of the tools we have presented in this chapter.

Summary

In this chapter, we learned about various important concepts, such as portfolios, portfolio allocation, and the Sharpe ratio. We looked at the different things to remember when we are considering investing in portfolios or selecting different assets or stocks in a portfolio. We also learned about the types of risks that we have when we select the different stocks or assets in a portfolio. We looked at the changes in the amount of money invested in various assets in a portfolio on a day-to-day basis, which helps us to determine whether we are gaining or losing money from our investments.

Then we discussed Markowitz portfolio optimization, which, in turn, helps us compare various portfolios with different combinations of stocks and different allocation percentages. We also learned how we can create an efficient frontier graph, which helps us understand which portfolios are less risky.

In the next chapter, we will discuss the capital-asset pricing model, which is a model used to determine the theoretically-appropriate required rate of return of an asset, to make decisions about adding assets to a well-diversified portfolio.

The Capital Asset Pricing Model

In finance, the **Capital Asset Pricing Model (CAPM)** is a model that is used to determine the rate of return of any assets; it helps us determine whether we can add the assets to create a diversified portfolio. This model usually divides the assets based on sensitivity and risk, which is represented by β in the financial industry. It also considers the expected returns of the market and theoretical risk-free assets.

We will cover the following topics in this chapter:

- Understanding the CAPM
- Understanding and calculating the beta of a security
- Calculating the beta of a stock
- The CAPM formula
- Calculating the expected return of stock
- Understanding the Sharpe ratio

Technical requirements

In this chapter, we will be using Jupyter Notebook for coding purposes. We will be using the pandas, NumPy, matplotlib, and SciPy libraries.

The GitHub repository for this chapter can be found at <https://github.com/PacktPublishing/Hands-on-Python-for-Finance/tree/master/Chapter%206>.

Understanding the CAPM

In 1960, a meeting was held between Harry Markowitz and William Sharpe. Markowitz offered the 26-year-old Sharpe an opportunity to continue his research on how various stocks functioned together. Sharpe wrote his thesis on this subject and created one of the most important models in financial history.

The premise of CAPM is not that different to the one created by Markowitz. Investors are risk-averse. They prefer earning a higher return but are also cautious about the risks they might face and want to optimize their portfolios in terms of both risk and return. Investors are unwilling to buy anything other than an optimal portfolio that optimizes their expected returns and standard deviation.

William Sharpe introduced the concept of a market portfolio as a bundle of all possible investments in the world, both bonds and stocks. The risk-return combination of this portfolio is superior to that of any other portfolio. The expected return of the market portfolio coincides with the expected return of the market and, because this is a diversified portfolio, it makes sense that it is optimal in terms of risk. It contains no idiosyncratic risk. The only risk faced by investors who own the market portfolio is systematic risk. We would expect this portfolio to lie somewhere on the efficient frontier as it is an efficient portfolio – the most efficient, in fact, of all portfolios.

Sharpe's CAPM contains the following:

- **Market portfolio:** A bundle of all possible investments in the world (both bonds and stocks). The risk-return combination of this portfolio is superior that of any other portfolio.
- **Risk-free asset:** An investment with no risk (zero standard deviation). It has a positive rate of return, but zero risk associated with it.
- **Beta coefficient:** This helps us to quantify the relationship between a security and the overall market portfolio.
- **Capital market line:** This is the line that connects the risk-free rate and is a tangent to the efficient frontier. The point at which the capital market line is a tangent to the efficient frontier is the market portfolio.

This looks like a simple extension of the ideas introduced by Markowitz. However, the CAMP assumes the existence of a risk-free asset and an investment with no risk and zero standard deviation. It has a positive rate of return but zero risk associated with it. There are people who are so risk-averse that they prefer to be 100% certain that their investment has no risk. These people will be willing to accept a lower rate of return.

The market portfolio has a lower expected rate of return because, in an efficient market, investors are compensated for the additional risks they are willing to bear once they own an efficient portfolio. They usually can't arbitrage the system and earn a higher expected return for the same level of risk. This means that a higher rate of returns means more risk and a lower level of risk means lower returns. If we have no risk, this translates to the lowest level of expected return, which is why we can expect the risk-free rate to provide a low level of expected returns.

Rational investors will form their portfolios by considering both the risk-free stocks and how much they are going to invest in the market portfolio. This question really depends on how much they want to earn.

Every investor can invest in a combination of risk-free assets and the market portfolio, according to their particular appetite for risk. If they are willing to risk more, they will hold a greater portion of the market portfolio. If they are unwilling to risk their money, they will buy more of the risk-free assets and less of the market portfolio. Investors who are interested in high expected returns will be able to borrow money and invest it in the market portfolio, going even further along the capital market line.

In the next section, we will discuss the relationship between the securities of individuals and the market portfolio. This will move us one step closer to learning how to build expectations about the prices of real-world assets.

The beta of securities

In this section, we will introduce the concept of beta, which is one of the main pillars of the CAPM. Beta helps us to quantify the relationship between the security and the overall market portfolio. Remember the market portfolio is made up of all the securities in the market: securities with low expected returns and securities with high expected returns. If there is an economic crisis, it is reasonable to expect the prices of most assets that make up the market portfolio to decrease and the market portfolio to experience a negative rate of return of, for example, -5%. In this case, investors can't protect themselves through diversification, because this is a systemic risk.

However, some securities in the market portfolio are less risky. They have a lower standard deviation and will decrease in value less than the market standard. Let's imagine that we have stock *A* and stock *B*. Stock *A* is less volatile than the market portfolio and so we only lose 3%. Stock *A* is an example of a security in the market portfolio that is less risky in times of volatility. The value of stock *B*, however, decreases by 7%, which is more than stock *A*. We can therefore consider stock *B* to be more volatile than stock *A*. Since stock *B* is volatile, when the economy recovers from the crisis, the market portfolio will do well and stock *B* may earn us a higher rate of return than stock *A*.

This example allows us to see that stocks might have a different behavior regarding their overall market performance. Some stocks are safer and will lose less and earn less than the market portfolio, while other stocks are riskier and will do well when the economy flourishes, but will perform poorly in times of crisis. This is precisely where beta (β) comes in handy. Beta allows us to measure the relationship between a stock and the market portfolio. It can be calculated as the covariance between the stock and the market divided by the variance of the market:

$$\beta_p = \frac{Cov(r_p, r_b)}{Var(r_b)}$$

The preceding formula measures the market risk, which cannot be avoided

through diversification. The riskier a stock, the higher its beta.

Let's look at a few important points that will help us understand beta:

- A beta of 0 means the stock has no relationship to the market.
- Betas that are below 1 are called defensive, because if the market does poorly, these stocks will typically lose less.
- Stocks with a beta of 1 will perform in the same way as the market.
- Stocks with beta of more than 1 are riskier than the market. They may do better than the market when the economy flourishes and lose more when it goes down.

We can understand the concept of beta by considering real-life examples. Some companies, such as Walmart, are less dependent on the economic cycle. Their clients must buy food and household products. In times of crisis, they may buy less food, but they can't give up food completely. An automobile company, however, is more dependent. During a recession, people are likely to prefer to use their old vehicles instead of buying a new vehicle. Car makers can therefore expect a significant slump in their revenues.

These examples basically illustrate how a recession might have a different impact on different businesses. The beta of these companies indicates whether they are riskier than the market.

Calculating the beta of a stock

In this section, we will see how we can find the beta value of a stock. The beta is typically measured using data from the past five years. We will be using data from Microsoft and the S&P 500 for the period between January 1, 2012 and December 31, 2016. We will calculate the beta of Microsoft and we will approximate the development of the market with the S&P 500.

Initially, we will import all the libraries that are required, such as `numpy` and `pandas`. We import the dataset of the Microsoft and S&P 500 as follows. The S&P index is given by the `^GSPC`. The S&P 500 is a capitalization-weighted index, and is associated with many ticker symbols, such as `^GSPC`, `INX`, and `$SPX`, depending on market or website. The S&P 500 differs from the Dow Jones Industrial Average and the NASDAQ Composite index because of its diverse constituency and weighting methodology.

```
In [1]:  
import numpy as np  
import pandas as pd  
data = pd.read_csv('MSFT_S&P.csv', index_col = 'Date')  
data
```

The preceding code displays the following output:

Date	MSFT	<code>^GSPC</code>
2012-01-03	23.031868	1277.060059
2012-01-04	23.573898	1277.300049
2012-01-05	23.814798	1281.060059

2012-01-06	24.184755	1277.810059
2012-01-09	23.866419	1280.699951
2012-01-10	23.952452	1292.079956
2012-01-11	23.849216	1292.479980
2012-01-12	24.090115	1295.500000
2012-01-13	24.305206	1289.089966
2012-01-17	24.313807	1293.670044

We need two values: the covariance between Microsoft and the S&P 500, and the variance of the S&P 500. We start by calculating the logarithmic returns and storing them in a `sec_returns` variable, as follows:

```
In [2]:
sec_returns = np.log( data / data.shift(1) )
```

Once we've done that, we can create a covariance matrix between MSFT and the market, as follows:

```
In [3]:
cov = sec_returns.cov() * 250
cov
```

The output of the preceding code is as follows:

	MSFT	^GSPC
MSFT	0.053781	0.018208
^GSPC	0.018208	0.016361

The `iloc` method allows us to obtain the covariance between the MSFT and S&P market as a float, as follows:

```
In [4]:
cov_with_market = cov.iloc[0,1]
cov_with_market

Out[4]:
0.01820843191039893
```

The preceding output is the nominator required to calculate the beta value, which is equal to the variance of the market. In our case, we can use a new variable called `market_var`, which contains the annual variance of the S&P 500:

```
In [5]:
market_var = sec_returns['^GSPC'].var() * 250
market_var

Out[5]:
0.016360592699269063
```

Finally, we calculate the beta value of these stocks by using the formula discussed in the previous section:

```
In [6]:
MSFT_beta = cov_with_market / market_var
MSFT_beta

Out[6]:
```

| 1.112944515219942

The beta value that we get is greater than one. This means that the MSFT stock is riskier than the market. It might do better than the market when the economy flourishes and lose more when it goes down.

The next step is to verify whether our calculations are correct. We are currently using data from Yahoo Finance. Let's now go to Yahoo Finance to check whether the beta value we obtained is credible. We can allow ourselves a 2 - 3% difference, because the data and the estimation methods may differ slightly, but the difference shouldn't be any bigger than that.

Once we go to the Yahoo finance website, search for the MSFT stock and check the quoted beta coefficient, as shown in the following screenshot:

Microsoft Corporation (MSFT)

Add to watchlist

NasdaqGS - NasdaqGS Real Time Price. Currency in USD

Quote Lookup



107.72 +0.21 (+0.20%)

Buy

Sell

At close: 4:00PM EST

Summary Chart Conversations

Statistics

Historical Data

Profile

Financials

Analysis

Options

Holders

Sustainability

Currency in USD

Valuation Measures

Trading Information

Get live quotes and news on new tabs

Market Cap (intraday)⁵

826.88B

Stock Price History

Enterprise Value³

782.77B

Beta (3Y Monthly)

1.09

Trailing P/E

50.57

52-Week Change³

27.14%

Forward P/E¹

21.37

S&P500 52-Week Change³

5.55%

PEG Ratio (5 yr expected)¹

1.77

52 Week High³

116.18

Price/Sales (ttm)

7.49

52 Week Low³

80.70

We can see that the beta value calculated for three years by Yahoo Finance is 1.09, which is similar to the value we obtained. This validates the beta coefficient that we calculated.

The CAPM formula

In the previous sections, we imagined that we were living in a world where all investors are rational, risk-averse, and willing to optimize their investment portfolios. We introduced the reader to the concept of a market portfolio and a risk-free asset. We also stated that investors make their decisions based on their risk appetite: those seeking higher expected returns will allocate a greater portion of their money to the market portfolio and less to the risk-free assets. Finally, we introduced the concept of beta, which measures how securities are expected to perform with respect to the entire market. We are now going to introduce the capital asset pricing model. The formula of the capital asset pricing model is as follows:

$$ri = rf + \beta im(rm - rf)$$

The CAPM formula has the following important parameters:

- **rf (Risk-free):** The risk-free rate of return. This is the minimum amount of compensation an investor would expect from an investment.
- **βim (Beta coefficient):** The beta value between the stock and the market. This coefficient specifies how risky a specific asset is with respect to the market.
- **rm (Market risk premium):** The market return. The amount of compensation an investor would expect when buying the market portfolio.

Let's take a look how the CAPM formula works. We start with a risk-free rate of return (rf), which is the bare minimum an investor would accept in order to buy a security. Since the investor must be compensated for the risk they're taking, we need the equity premium component given by the expected return of the market portfolio minus the risk-free rate. It is reasonable to expect this incentive since the security has a risk attached to it. This is the most widely used model by practitioners.

Calculating the expected return of a stock (using the CAPM)

We have covered a lot of concepts related to the CAPM. We have also discussed the beta coefficient and the CAPM formula. Let's now look at how we can compute the expected returns of a stock using the CAPM. First of all, we need to calculate the beta coefficient before we can apply it to the CAPM formula. Here, we will use Python and implement all the formulas of CAPM using Python. We will be using the same dataset that we used to calculate the beta coefficient and follow the same process as before:

```
In [2]:  
import numpy as np  
import pandas as pd  
  
data = pd.read_csv('MSFT_S&P.csv', index_col = 'Date')  
data  
  
sec_returns = np.log( data / data.shift(1) )  
cov = sec_returns.cov() * 250  
cov_with_market = cov.iloc[0,1]  
market_var = sec_returns['^GSPC'].var() * 250  
  
MSFT_beta = cov_with_market / market_var
```

From the preceding code, we get the beta value, which is stored in the `MSFT_beta` variable.

Let's take another look at the CAPM formula:

$$ri = rf + \beta im(rm - rf)$$

We can get an approximation of the risk-free trade value from Bloomberg's website. Assuming a risk-free rate of 2.5% and a risk premium of 5%, we now need to estimate the expected return of the Microsoft (MSFT) stock. We will use the CAPM formula and create an `MSFT_er` variable, which gives the expected return of the MSFT stock, as follows:

```
In [3]:  
MSFT_er = 0.025 + MSFT_beta * 0.05
```

```
| MSFT_er  
|  
| Out[3]:  
| 0.0806472257609971
```

The value we get is 8.06%. This is the return on the investment we might expect when buying the MSFT stock. This technique can be applied for any listed company that you are interested in.

Applying the Sharpe ratio in practice

Now that we know how to calculate a stock's return and assess its risk through its variance and standard deviation, we're ready to take another look at the Sharpe ratio. We discussed this topic in the previous chapter, but we will go into more detail in this section. As mentioned earlier, rational investors want to maximize their returns and are risk averse, which means they want to minimize the risk they face and invest in less volatile securities. They want less uncertainty and more clarity about an investment's rate of return. Rational investors are afraid that a risky investment would cause significant losses and want to invest their money in securities that are less volatile. It becomes obvious that the two dimensions must be combined.

We need a measure of the risk-adjusted return, a tool that will allow us to compare different securities. This is where the Sharpe ratio comes in. It is a great way to make comparisons between stocks and portfolios and decide which one is preferable in terms of risk and return.

The Sharpe ratio is given by the following formula:

$$S = \left(\frac{R_p - R_f}{\sigma_p} \right)$$

Here, R_p is the expected portfolio return of the stock, p ; R_f is the risk-free return; and σ_p is the portfolio standard deviation of the stock, p .

In the numerator of the preceding formula, we have the excess return of a stock. We have to subtract the risk-free rate (R_f) from the expected portfolio return (R_p). In the denominator, we have the standard deviation of the stock. If we increase the stock's expected rate of return, its Sharpe ratio becomes higher. Logically, if we increase the stock's standard deviation, its Sharpe ratio becomes lower.

This helps us to compare financial securities and portfolios quickly. It also allows us to understand whether a certain investment fund is performing in a satisfactory manner on a risk-adjusted basis. However, we should be aware that

this often comes at the expense of a riskier portfolio.

Obtaining the Sharpe ratio in Python

We will now continue our analysis of the Microsoft stock with regards to the S&P 500, which will represent the market. We will create a variable named `sharpe`, and this variable will be assigned with the following formula:

$$sharpe = \frac{\overline{r_{msft}} - r_f}{\sigma_{msft}}$$

We have to subtract the risk-free rate from the expected rate of return of the stock. We calculated the expected returns in a previous section and got a `MSFT_er` value of 0.0806. We can use the `MSFT_er` variable directly as the risk-free rate of Microsoft (r_{msft}). We also assumed the risk-free rate (r_f) to be 2.5%, which is equal to 0.025. To calculate the standard deviation of MSFT stock, we will use the `std()` method to obtain the volatility and annualize it by multiplying it by the square root of 250. The complete code to calculate the Sharpe ratio can be written as follows:

```
In [2]:  
Sharpe = (MSFT_er - 0.025) / (sec_returns['MSFT'].std() * 250 ** 0.5)  
Sharpe  
  
Out[2]:  
0.23995502423552612
```

The output is approximately 24%. We can use this ratio when we want to compare different stocks in stock portfolios.

Measuring alpha and verifying the performance of a portfolio manager

So far, we have learned about regression, the Markowitz efficient frontier of investment portfolios, and William Sharpe's capital asset pricing model. We have also learned how to calculate a portfolio's risk, return variance, and covariance. We know what beta is and how it can be interpreted in the context of Sharpe's capital market line.

In this section, we'll add another tool to our arsenal of financial knowledge. We'll learn how to interpret the intercept of the CAPM model, **alpha (α)**. In the world of finance and investments, alpha is often thought of as a measure of the performance of a fund manager.

The standard CAPM assumes an efficient financial market and an alpha of 0.

The following is the formula for the CAPM:

$$ri = \alpha + rf + \beta im(rm - rf)$$

Given that the beta (βim) multiplied by the equity risk ($rm-rf$) premium gives us the compensation for the risk that's been taken with the investment, alpha shows us how much we get without bearing the extra risk. A great portfolio manager is someone who outperforms the market and can achieve a high alpha. Conversely, a poor investment professional may obtain a negative alpha, meaning they underperformed with respect to the market.

In order to outperform the market, a portfolio manager can't simply invest in a diversified portfolio and sit back and relax. They have to select the winning portfolio of stocks. This is easier said than done, but investment professionals have a profound knowledge of certain industries, companies, and the stock market. They hand-pick successful investments and use active trading strategies. The term **active trading** means that an investor will try to take advantage of short-term price changes. We don't simply buy a stock portfolio that replicates the S&P 500.

We have to pick a portfolio of stocks that will win. Because other investments will eventually catch up and the price of these shares will normalize, we will have to sell the shares and select a new group of winning companies. This is active trading. The typical time period of active investments depends on the situation but ranges from a day to several months. Active trading is the basis on which investment professionals justify their fees.

To recap, there are two different types of trading:

- **Passive trading:** This trading technique involves investing in a market index (the S&P 500) and waiting patiently.
- **Active trading:** This trading involves adjusting investment portfolios on a frequent basis, based on their growth.

If an investment professional charges you 1% of the invested amount, they need to outperform the market by over 1%, otherwise you'd be better off investing in a passive fund that charges you a tiny portion of the invested income. Don't forget that the alpha is only comparable when the risk profile of the investments being compared is similar. We shouldn't compare a portfolio of small illiquid companies operating in an emerging market with a well-diversified portfolio of bluechip companies based in the US. Many investment managers struggle to produce an alpha value that is consistently over zero. Many investors are aware that well-diversified and passive index-tracking funds that charge low fees are very efficient, and are starting to consider these as a solid alternative to active management strategies.

William Sharpe believes that investors should bet on the efficiency of markets and shouldn't bother paying high management fees, given that only 2% funds have a positive alpha that is consistently different than zero.

Let's recap the ideas presented in this section.

The alpha is given by the following formula:

$$ri = \alpha + rf + \beta im(rm - rf)$$

In the world of finance and investments, alpha is often thought of as a measure of the performance of a fund manager has been. The standard CAPM assumes an

efficient financial market and an alpha of 0. Given that the beta multiplied by the equity risk premium gives us the compensation for risk that's been taken with the investment, the alpha shows us how much return we get without bearing extra risk. A great portfolio manager, someone who outperforms the market, can achieve a high alpha. Conversely, a poor investment professional may obtain a negative alpha, meaning they underperformed the market.

There are various different types of investment strategy:

Type of correlation	Description
Passive investing	This consists of buying a portfolio of assets and holding it for a long time regardless of the short-term macroeconomic developments.
Active investing	This refers to frequent trading based on the expectations of macroeconomic and company-specific developments.
Arbitrage trading	This is where we find pricing discrepancies on the market and exploit them in order to make a profit without assuming additional risk.
Value investing	This is where we invest in specific companies, in the hope that they will outperform their peers.

Different use cases of the CAPM using the `scipy` library

In this section, we will take a look at how we can implement the CAPM using the `scipy` library.

We will begin importing the `scipy` library, as follows:

```
| In [2]:  
|     from scipy import stats
```

Then, we will import `pandas` and `pandas_datareader`, which will be used to pull the dataset from the API, as follows:

```
| In [17]:  
|     import pandas as pd  
| In [13]:  
|     import pandas_datareader as web
```

After that, we import the data with the help of `pandas_datareader`. The index we have provided is basically the `SPY`, which is an ETF. In this context, index basically means a market:

```
| In [14]:  
|     spy_etf = web.DataReader('SPY', 'google')
```

We use the `info` function to see the details of the columns extracted, as follows:

```
| In [21]:  
|     spy_etf.info()  
  
| Out[21]:  
  
<class 'pandas.core.frame.DataFrame'>  
DatetimeIndex: 1897 entries, 2010-01-04 to 2017-07-18  
Data columns (total 5 columns):  
Open      1879 non-null float64  
High      1879 non-null float64  
Low       1879 non-null float64
```

```
| Close      1897 non-null float64  
| Volume    1897 non-null int64  
| dtypes: float64(4), int64(1)  
| memory usage: 88.9 KB
```

We can also see the first five rows of the dataset retrieved from the API using the `head()` function:

```
| In [22]:  
| spy_etf.head()
```

The preceding code generates the following output:

Date	Open	High	Low	Close	Volume
2010-01-04	112.37	113.39	111.51	113.33	118944541
2010-01-05	113.26	113.68	112.85	113.63	111579866
2010-01-06	113.52	113.99	113.43	113.71	116074402
2010-01-07	113.50	114.33	113.18	114.19	131091048
2010-01-08	113.89	114.62	113.66	114.57	126402764

If you take a look at SPY ETF, we have about 2,000 entries, each of which have open, high, low, and close columns. These results are actually indicators, rather than the actual S&P 500. SPY ETF is an exchange-traded fund that represents

the S&P 500.

Let's now create a start date and an end date:

```
In [18]:  
start = pd.to_datetime('2010-01-04')  
end = pd.to_datetime('2017-07-18')
```

Now, let's say that our portfolio strategy is to invest entirely in Apple stocks. We will use `pandas_datareader` and retrieve the Apple stock with the preceding start and end dates from Google Finance, as follows:

```
In [24]:  
aapl = web.DataReader('AAPL', 'google', start, end)
```

Once we have executed the preceding statement, we can check out the head of the dataframe, as follows:

```
In [27]:  
aapl.head()
```

The preceding code generates the following output:

Date	Open	High	Low	Close	Volume
2010-01-04	30.49	30.64	30.34	30.57	123432050
2010-01-05	30.66	30.80	30.46	30.63	150476004
2010-01-06	30.63	30.75	30.11	30.14	138039594
2010-01-07	30.25	30.29	29.86	30.08	119282324

2010-01-08	30.04	30.29	29.87	30.28	111969081
------------	-------	-------	-------	-------	-----------

The CAPM basically states that there should be a relationship between the performance of Apple's stocks and the overall market performance. Here, the market data is the data that we extracted for SPY, which is an **Exchange-traded Fund (ETF)**. Let's visualize this data. Import the `matplotlib` library, as follows:

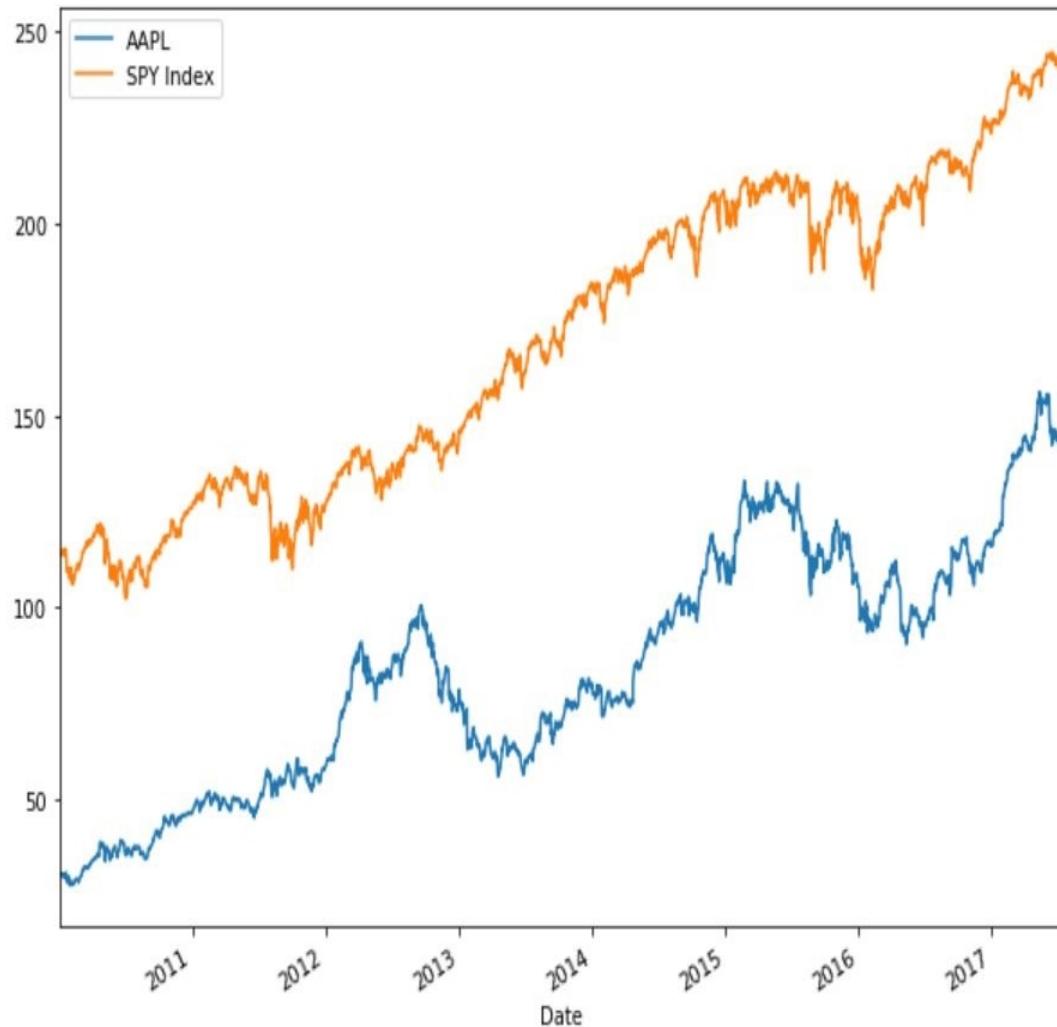
```
In [28]:  
import matplotlib.pyplot as plt  
%matplotlib inline
```

We will be plotting the `close` column of the `aapl` dataframe along with the `close` column of the `spy_etf` dataframe. We will consider a plot size of (8, 5), as follows:

```
In [29]:  
aapl['Close'].plot(label='AAPL', figsize=(10,8))  
spy_etf['Close'].plot(label='SPY Index')  
plt.legend()
```

The output of the preceding code is shown in the following screenshot:

```
Out[29]: <matplotlib.legend.Legend at 0x1c4e9221cf8>
```



From the preceding output, we can see Apple and the index of the S&P market 500, or an exchange-traded fund, which essentially mimics its behavior. We now need to find the beta and the alpha, using the CAPM, so that we can compare Apple and the S&P ETF. To do this, we are going to return the cumulative returns.

The cumulative returns for Apple and the SPY ETF can be calculated as follows:

```
In [31]:  
aapl['Cumulative'] = aapl['Close']/aapl['Close'].iloc[0]  
spy_etf['Cumulative'] = spy_etf['Close']/spy_etf['Close'].iloc[0]
```

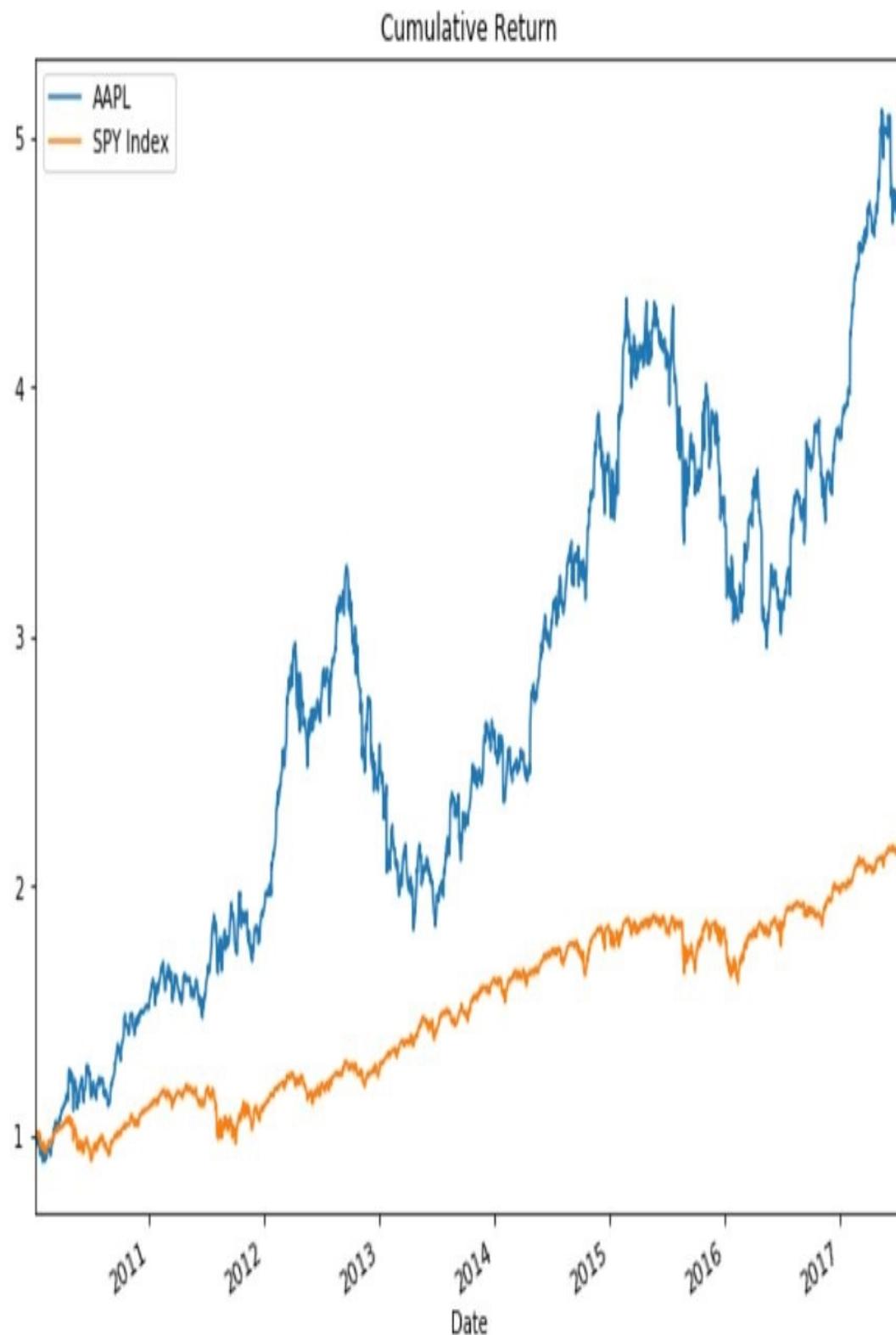
We basically divide all the `close` values with the close price on the very first day.

We create a new column called `cumulative` in both the `aapl` and `spy_etf` dataframes. We now plot these cumulative values, as follows:

```
In [33]:  
aapl['Cumulative'].plot(label='AAPL', figsize=(10,8))  
spy_etf['Cumulative'].plot(label='SPY Index')  
plt.legend()  
plt.title('Cumulative Return')
```

The output of the preceding code is shown in the following screenshot:

Out[33]: <matplotlib.text.Text at 0x1c4e97460b8>



This output shows the cumulative return. If we invested \$1 in 2011, we can see that we could have got better returns if we had invested in Apple. Let's now check out the daily returns for these two stocks:

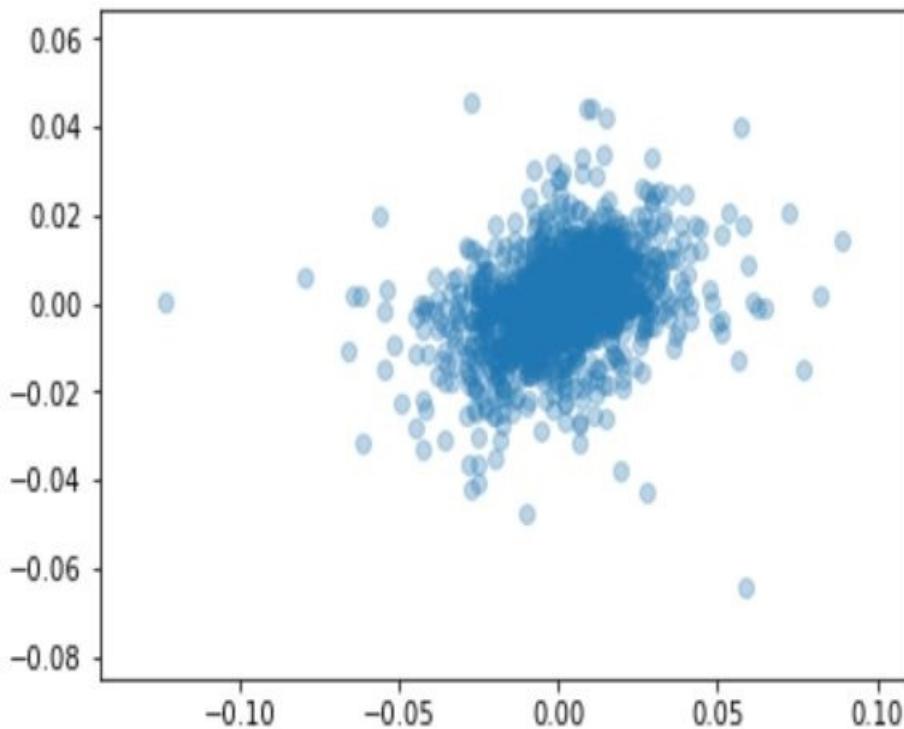
```
In [41]:  
aapl['Daily Return'] = aapl['Close'].pct_change(1)  
spy_etf['Daily Return'] = spy_etf['Close'].pct_change(1)
```

Here, we use the `pct_change()` function, where we provide one day as the parameter and create a new column called `Daily Return` for both `aapl` and `spy_etf`. Next, we will take the scatter plot and plot the daily return values of both `aapl` and `spy_etf` to check whether there is any correlation. Here, we also provide the value of `alpha`, which is equal to `0.3`:

```
In [45]:  
plt.scatter(aapl['Daily Return'], spy_etf['Daily Return'], alpha=0.3)
```

The output of the preceding code is shown in the following screenshot:

Out[45]: <matplotlib.collections.PathCollection at 0x1c4ea1ec518>

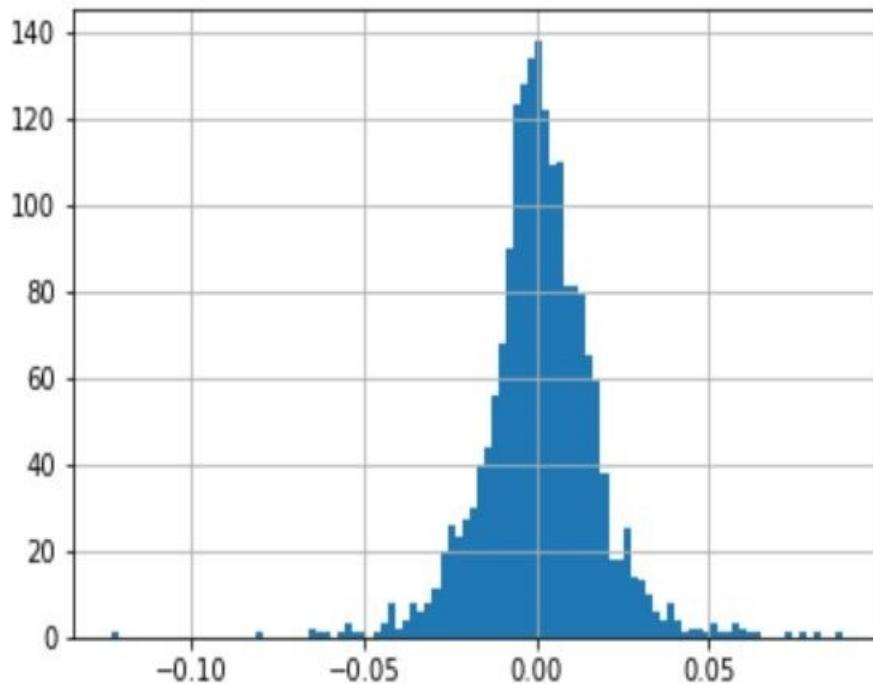


From the preceding output, we can see that there is some correlation. We can also plot `Daily Returns` with the help of the histogram to see the highest frequency returns, as follows:

```
| In [46]:  
| aapl['Daily Return'].hist(bins=100)
```

The output of the preceding code is shown in the following screenshot:

Out[46]: <matplotlib.axes._subplots.AxesSubplot at 0x1c4ea1a4908>

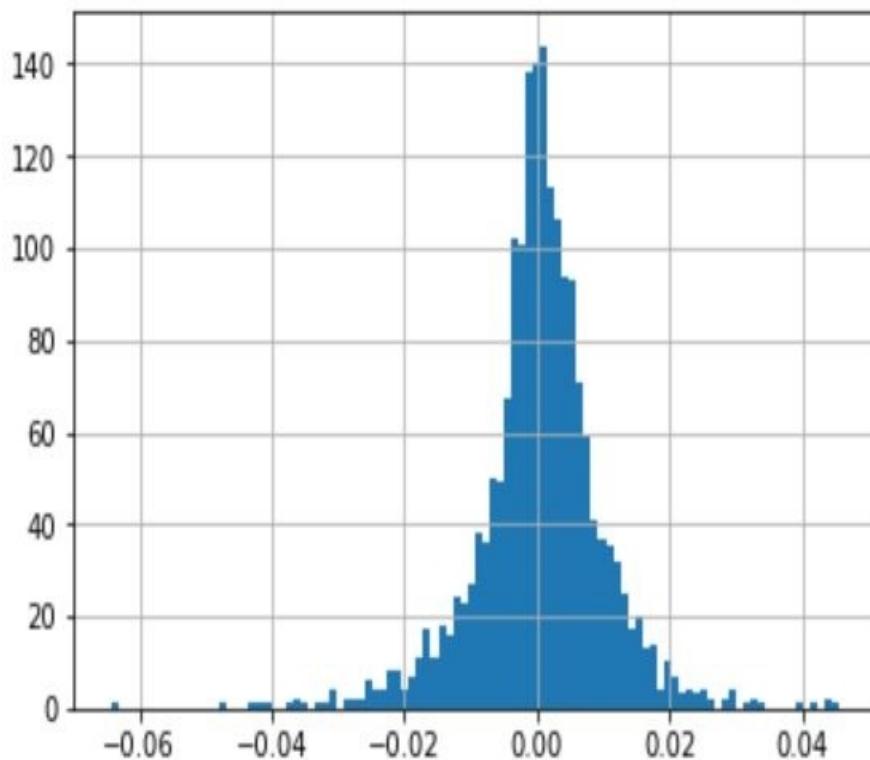


We can also take a look at the histogram of the `spy_etf Daily Return`:

```
| In [48]:  
| spy_etf['Daily Return'].hist(bins=100)
```

The output of the preceding code is shown in the following screenshot:

```
Out[48]: <matplotlib.axes._subplots.AxesSubplot at 0x1c4e97c9cc0>
```



Let's calculate the beta and alpha values. The following code uses a tuple unpacking, which returns four different values. We are going to use the `stats.linregress()` method, as follows:

```
| beta,alpha,r_value,p_value,std_err = stats.linregress(aapl['Daily Return'].iloc[1:],spy_
```

The following codes helps us to see the beta value:

```
In [38]:  
beta  
Out[38]:  
0.19423150396392763  
  
In [39]:  
alpha  
Out[39]:  
0.00026461336993233316
```

```
In [40]:  
r_value  
Out [40]:  
0.33143080741409325
```

Summary

In this chapter, we looked at the capital asset pricing model, which was used to determine the rate of return of an asset. We also discussed how to calculate the beta of the security, which helps us to determine which stock is riskier compared to the market. Then, we saw how to measure an alpha value and how to verify the performance of the portfolio. We also learned about the various use cases of the capital asset pricing model using the `scipy` library.

In the next chapter, we will discuss regression analysis in the financial domain.

Regression Analysis in Finance

In this chapter, we will be discussing regression analysis and how it plays a very important role in finance. In statistical modeling, regression analysis is a set of statistical processes for estimating relationships between variables. It includes many techniques for modeling and analyzing several variables when the focus is on the relationship between a dependent variable and one or more independent variables.

Regression models (both linear and non-linear) are used to predict real values, such as a salary. If your independent variable is time, then you are forecasting future values, otherwise your model is predicting present but unknown values. In this chapter, we will look at the basic concepts of simple linear regression, where we will examine the relationship between two variables, and then we will move on to a more complex regression technique called multiple or multivariate linear regression in which we will be examining the relationship between more than two variables.

We will also learn how we can design a predictive model that will able to predict an output based on the relationship between these independent and dependent features.

In this chapter, we will focus on the following topics:

- The fundamentals of simple regression analysis
- Running a regression model in Python
- Learning how to distinguish a good regression model
- Computing the intercept and beta (coefficient)
- Evaluating the model using an R square value
- Implementing decision trees to solve non-linear equations

Technical requirements

In this chapter, we will be using Jupyter Notebook for coding purposes. We will also be using the pandas, NumPy, matplotlib, and SciPy libraries.

The GitHub repository for this chapter can be found at <https://github.com/PacktPublishing/Hands-on-Python-for-Finance/tree/master/Chapter%207>.

The fundamentals of simple regression analysis

Regression analysis is one of the most frequently used tools in the world of finance. It quantifies the relationship between a variable called a **dependent variable** and one or more explanatory variables, which are also called **independent variables**. Regression analysis is handy when we want to forecast a future dependent variable with the help of patterns from our historical data.

Let's consider an example to do with house prices. Usually, the larger the house, the more expensive it is.

In this example, we are considering two variables:

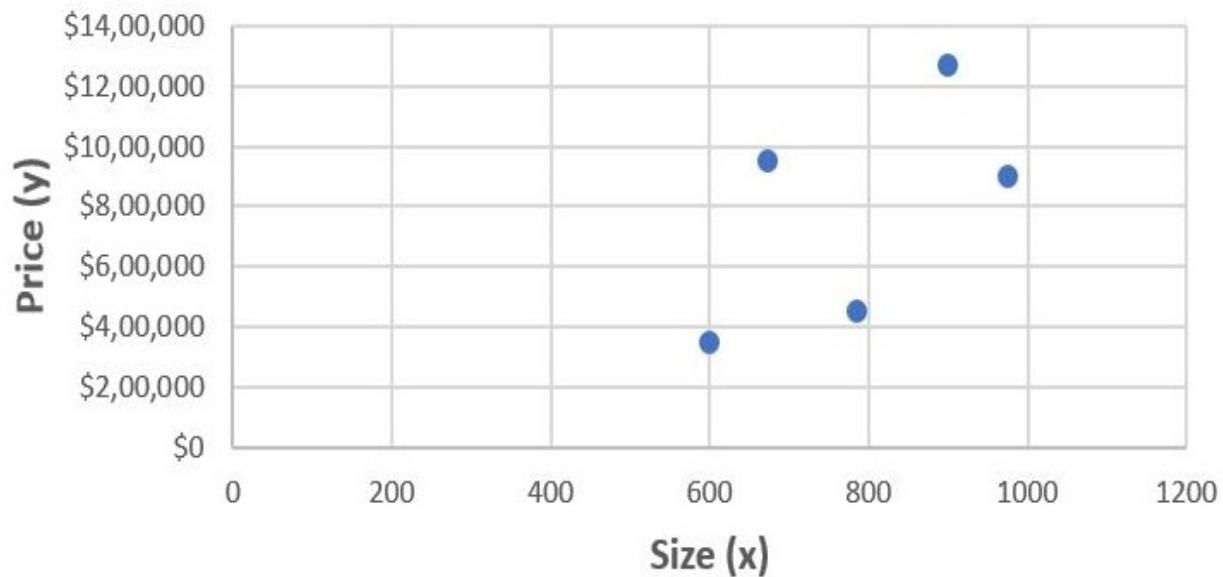
- The price of the house
- The size of the house

Here, the explanatory or independent variable is the size of the house, as this helps us explain why certain houses cost more, and the dependent variable is the price of the house, which is completely dependent on the size of the house. Based on the size of the house, this value varies, so we know for a fact that there is a relationship between the two variables. If we know the value of the explanatory variable, which is the size of the house, we can determine the expected value of the dependent variable, which is the house price.

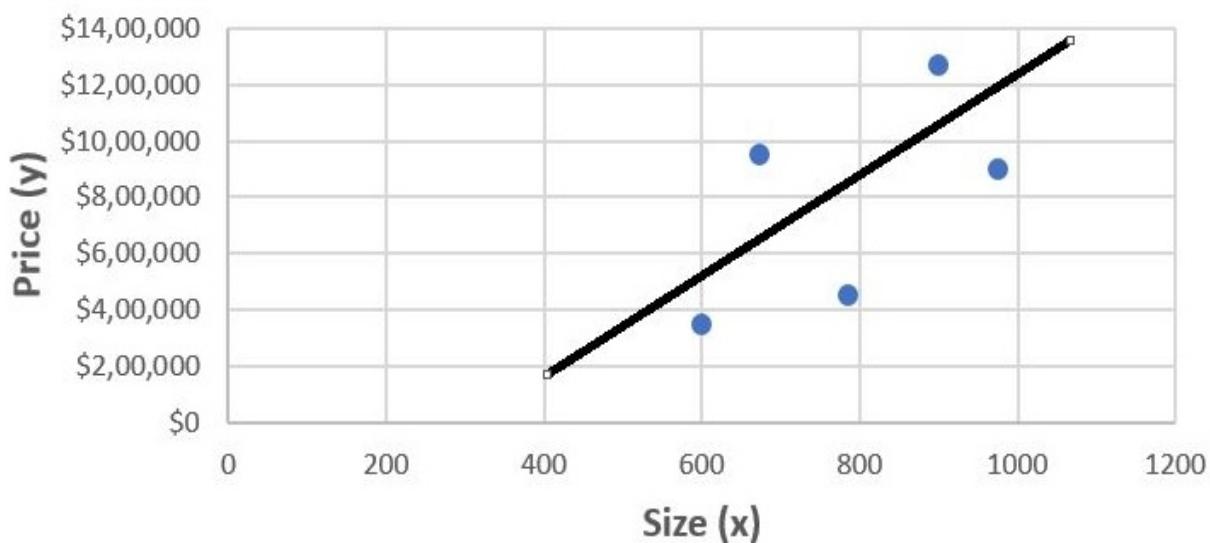
There are many other factors that determine house prices. In this case, we are only using size, but this is not the only variable in real life. If we use only one independent variable in a regression, this is known as **simple regression**, while regressions with more than one variable are called **multivariate regression** or **multi-linear regression**.

Let's start by considering a simple regression with two variables, x and y . Here, y indicates the house price and x indicates the house size. The data with the size and the price of the house can be found in the Excel sheet called `Housing-Data.xlsx`, which has been uploaded to the GitHub link specified in the *Technical*

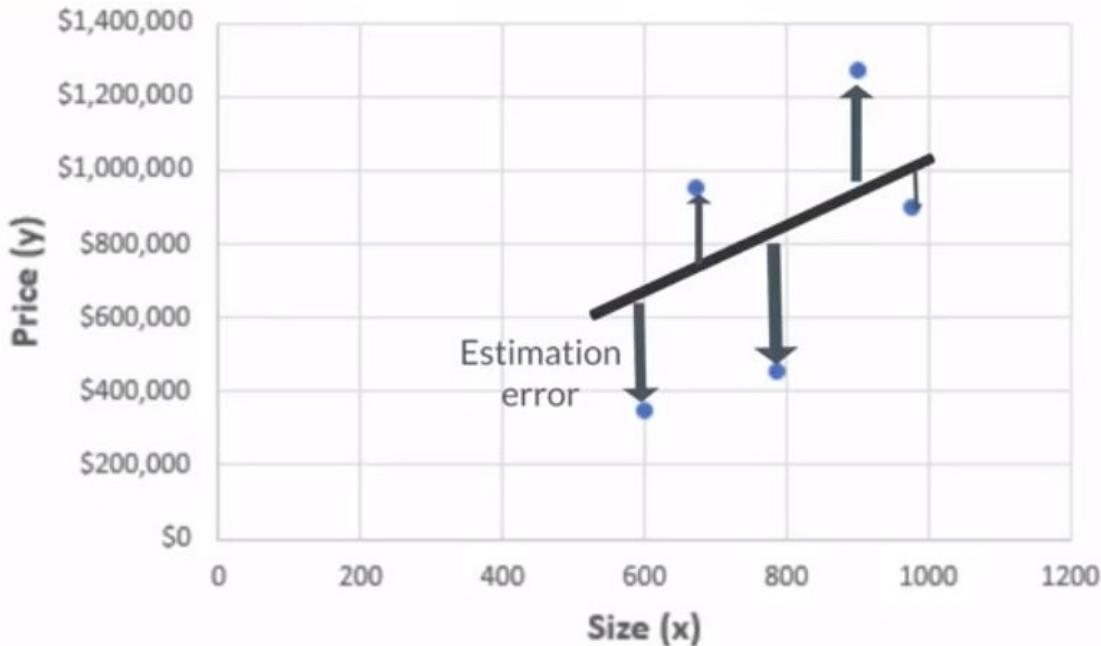
requirements section. We can plot the x and y variables in a graph, as shown here:



The preceding diagram contains information about the actual size and prices of houses. We can easily see that the two variables are connected: larger houses have higher prices. Regression analysis assumes the existence of a linear relationship between the two variables. We can draw a line of best fit that can help us describe the relationship between all the data points, as shown in the following plot:



To determine the line of best fit that will help us describe the relationship between house prices and house size, we need to find a line that minimizes the error between the lines and the actual observation, as shown here:



From the preceding diagram, observe how the different observations tend to deviate from the line. A linear regression will calculate the error that's observed when using different lines, allowing us to determine which contains the least error. Each deviation from the line is an error because it deviates from the prediction that the line would have provided.

The general equation of a straight line, which is also known as a linear equation, is given as follows:

$$y = mx + b$$

m → Slope (or Gradient) b → Y Intercept

Let's describe the preceding equation:

- **m:** This is the slope of the line

- **b:** This is the **y** intercept

Given that this is a linear equation and its output is a line, we would expect to obtain an equation with a very similar shape to the following. In this equation, we will be considering $\beta_0=b$ and $\beta_1=m$:

$$y=\beta_0+\beta_1 x_1$$

As we move ahead, we will learn about the financial meaning of the alpha (β_0) and the beta coefficients (β_1). However, first, let's take a look at the different regression techniques and the basic mathematical equation that's used to represent these techniques:

Type of regression	Description
Simple regression	$y=\beta_0+\beta_1 x_1+\epsilon_i$ Regression analysis assumes the existence of a linear relationship between the two variables. One straight line is the best fit and can help us describe the rapport between all the data points we see in the plot.
Multivariate regression	$y=\beta_0+\beta_1 x_1+\beta_2 x_2+\beta_3 x_3+\epsilon_i$ By considering more variables in the regression equation, we'll improve its explanatory power and provide a better idea of the circumstances that determine the development of the variable we are trying to predict.

Running a regression model in Python

So far, we have learned about some of the fundamental concepts behind implementing linear regression problems in finance. Now, let's look at how we can implement regression analysis using Python.

In Python, there are modules that help run a huge variety of regression problems. In this section, we will learn about using Python for both simple linear regression and multiple linear regression. We are going to consider a housing dataset and predict houses prices using regression analysis by considering various features that affect the price of houses.

Simple linear regression using Python and scikit learn

To begin, we are going to import a couple of modules and libraries, as shown here:

```
In[1]:  
import numpy as np  
import pandas as pd  
  
from scipy import stats  
import statsmodels.api as sm  
  
import matplotlib.pyplot as plt
```

In the preceding code, we imported `numpy`, `pandas`, `scipy`, `statsmodels`, and the `matplotlib` library. The next step is to import the `Housing.xlsx` dataset, on which we are going to apply the regression analysis, as shown here:

```
In[2]:  
data = pd.read_excel('Housing.xlsx')
```

The top five records of the dataframe are extracted:

```
In[3]:  
data.head()
```

The output looks as follows:

	House Price	House Size (sq.ft.)	State	Number of Rooms	Year of Construction
0	1116000	1940	IN	8	2002
1	860000	1300	IN	5	1992
2	818400	1420	IN	6	1987
3	1000000	1680	IN	7	2000
4	640000	1270	IN	5	1995

We will be considering the first two columns in the preceding dataset, which are `House Price` and `House size (sq.ft.)`. We will divide the dataset into independent

features and dependent features, as shown here:

```
| In[4]:  
| X = data['House Size (sq.ft.)']  
| Y = data['House Price']
```

The house size is the independent feature and is stored in the `x` variable, whereas `House Price` is the dependent feature and is stored in the `y` variable.

The following are the top five records in the `x` variable:

```
| In[5]:  
| X.head()
```

The output looks as follows:

```
0    1940  
1    1300  
2    1420  
3    1680  
4    1270  
Name: House Size (sq.ft.), dtype: int64
```

The following are the top five records in the `y` variable:

```
| In[6]:  
| Y.head()
```

The output looks as follows:

```
0    1116000  
1    860000  
2    818400  
3    1000000  
4    640000  
Name: House Price, dtype: int64
```

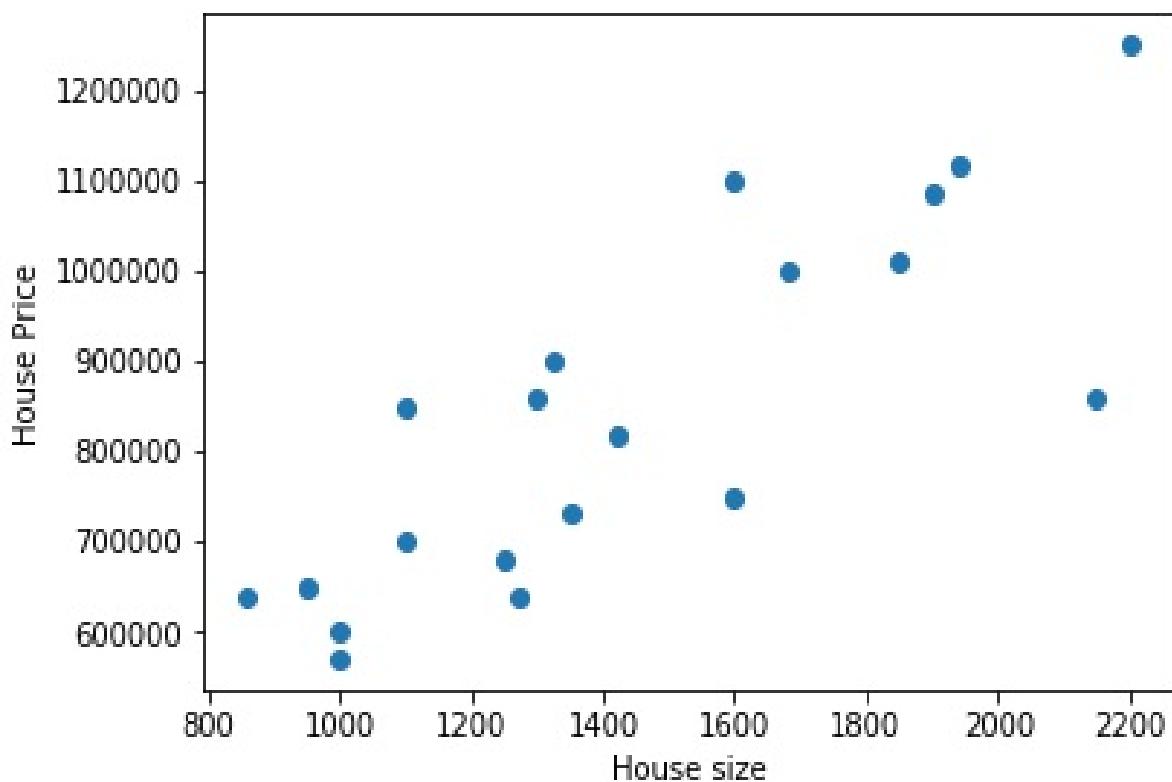
The `matplotlib` library helps us to project our data in a graph.

We will be using the scatter plot, which will help us to scatter the data in a chart so that we can visualize it, as shown here:

```
| In[9]:  
| plt.scatter(X,Y)
```

```
| plt.xlabel("House size")
| plt.ylabel("House Price")
| plt.show()
```

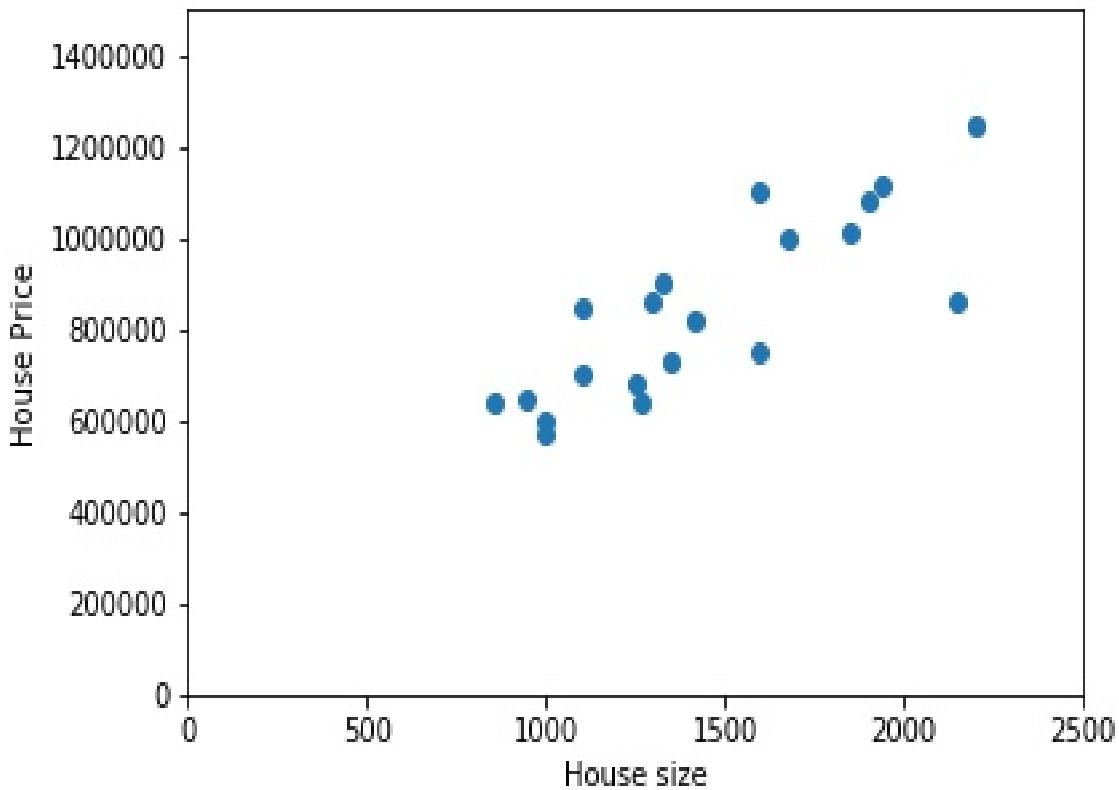
Here is the output:



These scattered points show the house price with respect to the various house sizes. First, we need to set the limits of the *x* label and the *y* label limit, as we can see that the house size in the *x* axis starts from 800, whereas the house price in the *y* axis starts from around 500,000. This can be done using `plt.axis()`:

```
In[10]:
plt.scatter(X,Y)
plt.axis([0, 2500, 0, 1500000])
plt.xlabel("House size")
plt.ylabel("House Price")
plt.show()
```

In the preceding code, we are limiting the *x*-axis value from a range of 0 to 2,500, whereas we are limiting the *y* axis to a range of 0 to 1,500,000. This will give us enough space to plot our observation, as shown here:



We can now see that the output looks much better. We can see that even the smallest of houses in our sample cost a lot of money and we can get a better idea of the size-to-price ratio of our data.

Now that we have our independent features (X) and our dependent features (Y), to create a regression model, we will first split the independent and dependent features into a training dataset and a test dataset. We will use the former to train our linear regression model and the latter to check the accuracy of the linear regression model.

To split the independent and dependent features, we will use the Scikit learn library. Scikit-learn (formerly scikits.learn) is a free machine learning library for the Python programming language. It features various classification, regression, and clustering algorithms, including support vector machines, random forests, gradient boosting, k-means, and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries, NumPy and SciPy.

To use the scikit learn library to split our data, we need to import the `sklearn` library:

```
| In[11]:  
| from sklearn.model_selection import train_test_split
```

After importing, we split the independent and dependent features into a training dataset and a test dataset:

```
| In[12]:  
| X_train,X_test,y_train,y_test=train_test_split(X,Y,test_size=0.2,random_state=0)
```

Here, `X_train` and `y_train` are the training datasets that we will be using to train our regression model. The `X_test` data will be used to test the accuracy of our regression model, which will later be compared with `y_test`.

We will also be carrying out an additional step, which is converting `X_train`, `y_train`, `X_test`, and `y_test` into two-dimensional arrays, as shown here:

```
| In[13]:  
| X_train=np.array(X_train).reshape(-1,1)  
| y_train=np.array(y_train).reshape(-1,1)  
| X_test=np.array(X_test).reshape(-1,1)  
| y_test=np.array(y_test).reshape(-1,1)
```

Finally, we apply the linear regression model using the `scikit.learn` library:

```
| In[14]:  
| from sklearn.linear_model import LinearRegression  
| linregression=LinearRegression()  
| linregression.fit(X_train,y_train)
```

In the first line of the preceding code, we import the linear regression model from the scikit learn library. In the next line, we initialize an object of the `LinearRegression` model, which will be trained using the `X_train` and `y_train` datasets with the `fit()` method. Once this code is executed, we get the following output:

```
| Out[14]:  
| LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,  
| normalize=False)
```

If we see the preceding output, we can consider our model ready and can test our `X_test` data to see the prediction of our model, as shown here:

```
| In[15]:  
| y_pred=linregression.predict(X_test)  
| y_pred
```

`y_pred` has the following predicted values:

```
|Out[15]:  
|array([[634973.36884605],  
|       [772590.58762453],  
|       [752930.98494189],  
|       [890548.20372037]])
```

Computing the slope and the intercepts

We have now created a linear regression model that can predict the price of a house whenever the size of the house is given as an input to the model. As we mentioned previously, simple linear regression is represented by the following equation:

$$y = \beta_0 + \beta_1 x_1 + \varepsilon_i$$

Let's describe this equation:

- β_0 is the intercept
- β_1 is the slope

We can see the intercept from the preceding simple linear regression model, as shown here:

```
| In[16]:  
| linregression.intercept_
```

The output is as follows:

```
| Out[16]:  
| array([261440.9178759])
```

From the preceding code, we can see that `intercept_` is the keyword that's used to see the intercept values.

Similarly, we can see the slope from the simple linear regression model:

```
| In[17]:  
| linregression.coef_
```

The output is as follows:

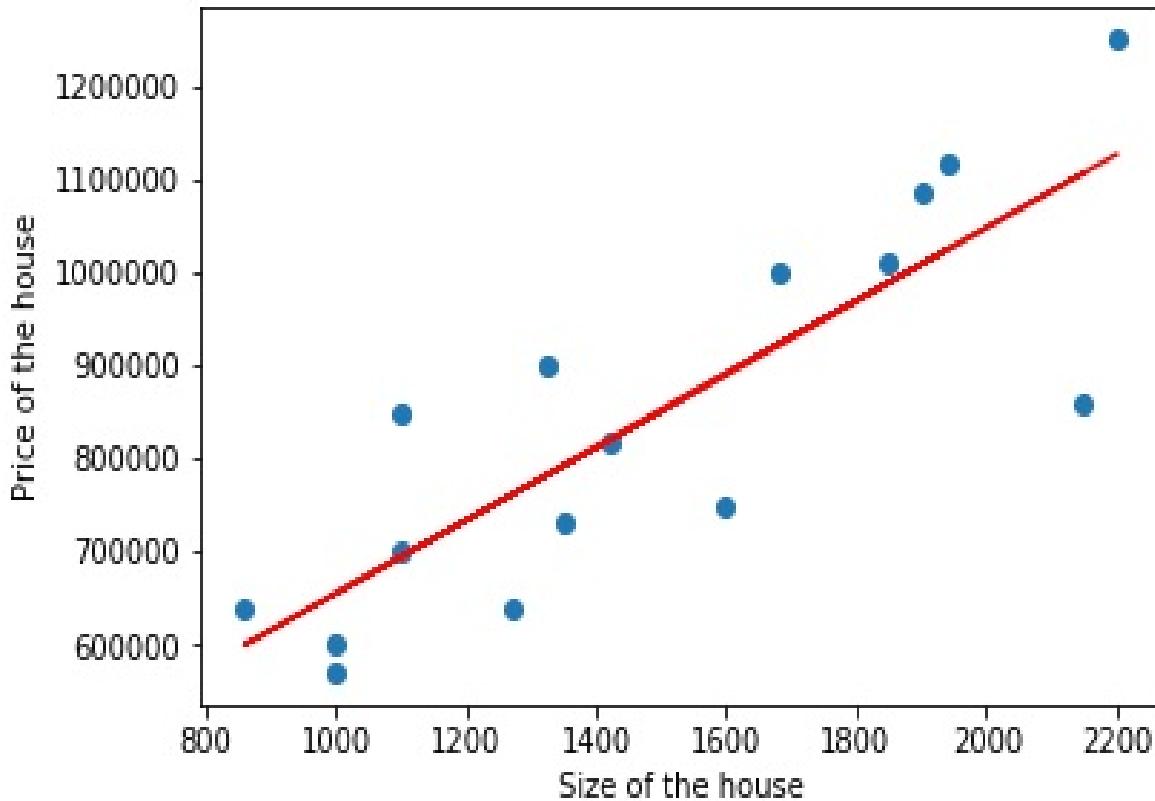
```
| Out[17]:  
| array([[393.19205365]])
```

The preceding output provides us with the slope value.

We would also like to visualize how the the line of best fit, which is represented by the $y = \beta_0 + \beta_1 x_1 + \epsilon_i$ equation, is displayed. Here, we will use the matplotlib library to see the line:

```
In[18]:  
import matplotlib.pyplot as plt  
  
plt.scatter(X_train,y_train)  
plt.plot(X_train,linregression.predict(X_train),'r')  
plt.show()
```

The output looks as follows:



The straight line that we can see in the graph is plotted using the intercept and the slope value that we calculated using the simple regression model.

In this section, we discussed the simple linear regression technique, in which we only had one independent feature. In the next section, we will discuss the multiple or multivariate regression technique.

Multiple linear regression in Python using the scikit-learn library

If we have more than one independent feature, we can create a regression model to carry out future predictions. This regression model is called a multiple linear regression model. If we have three independent features, for example, this is represented by the $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \epsilon_i$ equation, where the independent features are x_1 , x_2 , and x_3 . Here, β_1 , β_2 , and β_3 are the slopes or the coefficients and β_0 is the intercept.

We will consider a new dataset in this section, which is `startups_Invest.csv`. This dataset has five features, as shown here:

A	B	C	D	E	F
1	R&D Spend	Administration	Marketing Spend	State	Profit
2	165349.2	136897.8	471784.1	New York	192261.83
3	162597.7	151377.59	443898.53	California	191792.06
4	153441.51	101145.55	407934.54	Florida	191050.39
5	144372.41	118671.85	383199.62	New York	182901.99
6	142107.34	91391.77	366168.42	Florida	166187.94
7	131876.9	99814.71	362861.36	New York	156991.12
8	134615.46	147198.87	127716.82	California	156122.51
9	130298.13	145530.06	323876.68	Florida	155752.6
10	120542.52	148718.95	311613.29	New York	152211.77
11	123334.88	108679.17	304981.62	California	149759.96
12	101913.08	110594.11	229160.95	Florida	146121.95
13	100671.96	91790.61	249744.55	California	144259.4
14	93863.75	127320.38	249839.44	Florida	141585.52
15	91992.39	135495.07	252664.93	California	134307.35
16	119943.24	156547.42	256512.92	Florida	132602.65
17	114523.61	122616.84	261776.23	New York	129917.04
18	78013.11	121597.55	264346.06	California	126992.93

The dataset consists of data related to 50 start-up companies that have invested some amount of money in various internal departments, such as R&D, marketing, or administration, from various states. Based on this expenditure, the company has achieved some profit, which is specified in the Profit column. Our goal will be to create a multiple linear regression model that will be able to predict the profit based on the expenditure in the R&D, marketing, and administration departments from various states. R&D Spend, Marketing Spend, Administration, and State are the independent features in this example, while Profit is the dependent feature.

We will perform the following steps to create a multiple linear regression model

using Python and the sklearn library:

1. Import the required libraries:

```
In[1]:  
import numpy as np  
import matplotlib.pyplot as plt  
import pandas as pd
```

2. Read and split our dataset into dependent and independent features:

```
In[2]:  
dataset = pd.read_csv('Startups_Invest.csv')  
X = dataset.iloc[:, :-1]  
y = dataset.iloc[:, 4]
```

Here, x represents the independent features and y is the dependent feature. The following code helps us see the top five records of the x variables:

```
In[3]  
X.head()
```

The output looks as follows:

	R&D Spend	Administration	Marketing Spend	State
0	165349.20	136897.80	471784.10	New York
1	162597.70	151377.59	443898.53	California
2	153441.51	101145.55	407934.54	Florida
3	144372.41	118671.85	383199.62	New York
4	142107.34	91391.77	366168.42	Florida

Similarly, the following code helps us see the top five records of the y variable:

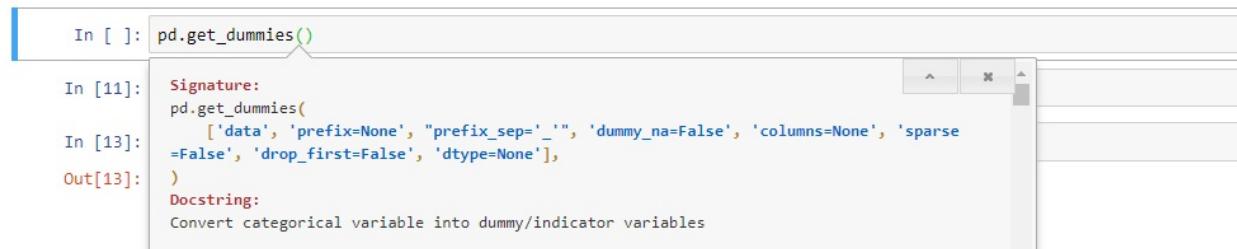
```
In[4]:  
y.head()
```

The output looks as follows:

```
0    192261.83
1    191792.06
2    191050.39
3    182901.99
4    166187.94
Name: Profit, dtype: float64
```

In the next step, we can observe that the `state` column is basically a categorical feature, so we need to convert the categorical feature into dummy variables using pandas. Our final regression model will not be able to understand categorical features, since they are in the forms of strings. The regression model usually understands mathematical calculations using various algorithms, so we need to convert the categorical variables into dummy variables. To do this, we will be using `pandas.get_dummies()`.

The following is the syntax of the `get_dummies()` function:



In []: `pd.get_dummies()`

In [11]: `Signature:`
`pd.get_dummies(`
 `['data', 'prefix=None', "prefix_sep=' '", 'dummy_na=False', 'columns=None', 'sparse`
`=False', 'drop_first=False', 'dtype=None'],`
`)`
`Docstring:`
`Convert categorical variable into dummy/indicator variables`

The parameters can be described as follows:

- `data`: This can be an array, a series, or a DataFrame.
- `prefix`: This can be a string or a dictionary of strings. By default, it is `None`.
- `columns`: This is a list of the columns that have been specified.
- `drop_first`: This is a Boolean value and by default it is a `False` value. We need to specify whether to get $K-1$ dummies out of the K categorical levels by removing the first level.

The following is the code that helps us to convert the `state` column into dummy variables:

```
In[5]:
State=pd.get_dummies(X.iloc[:,3],drop_first=True)
```

Inside the `get_dummies` function, we have provided the State column using `iloc`

indexing and we have set the `drop_first` parameter as `True` to just get only K-1 dummies out of the K categorical variables from the `state` column. Here, K is the total number of categorical variables inside the State column. Currently, we have three unique States, which are New York, California, and Florida. We choose only K-1 categories because this is the minimum number required to represent all K categories. We know that if the state is neither Florida nor New York, for example, as is the case in the second record in the following output, it must be California.

The following code helps us to see the top 10 records of the dummy variables:

```
| In[6]:  
| State.head(10)
```

The output looks as follows:

	Florida	New York
0	0	1
1	0	0
2	1	0
3	0	1
4	1	0
5	0	1
6	0	0
7	1	0
8	0	1
9	0	0

Since we have converted the State column into categorical features, we will be deleting the State column, as it is no longer required. This can be done as follows:

```
| In[7]:  
| X.drop('State', axis=1, inplace=True)
```

Let's check the `x` independent variable to see whether the `State` column has successfully been dropped:

```
| In[8]:  
| X.head()
```

The output is as follows:

	R&D Spend	Administration	Marketing Spend
0	165349.20	136897.80	471784.10
1	162597.70	151377.59	443898.53
2	153441.51	101145.55	407934.54
3	144372.41	118671.85	383199.62
4	142107.34	91391.77	366168.42

We can see that the `State` column has been dropped successfully.

We now need to concatenate the dummy variables that were created from the `State` feature, which is stored in the `state` variable. To concatenate them, we use the `concat()` function that's present in the pandas library. The code is as follows:

```
| In[9]:  
| X=pd.concat([X,state],axis=1)
```

Now, let's check the updated independent feature, `x`, as follows:

```
| In[10]:  
| X.head()
```

The output is as follows:

	R&D Spend	Administration	Marketing Spend	Florida	New York
0	165349.20	136897.80	471784.10	0	1
1	162597.70	151377.59	443898.53	0	0
2	153441.51	101145.55	407934.54	1	0
3	144372.41	118671.85	383199.62	0	1
4	142107.34	91391.77	366168.42	1	0

We can see that the dummy variable columns have been added to the `x` variable. With this, we have completed the data-preprocessing step. The next step is to divide the independent features, `x`, and the dependent feature, `y`, into training and test data using the scikit learn library.

First, we import the sklearn library:

```
| In[11]:  
| from sklearn.model_selection import train_test_split
```

The following code splits the independent and dependent features into training and test data:

```
| In[12]:  
| X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state
```

Finally, we apply the linear regression model by using the scikit learn library:

```
| In[14]:  
| from sklearn.linear_model import LinearRegression  
| linregression=LinearRegression()  
| linregression.fit(X_train,y_train)
```

In the first line of the preceding code, we import the linear regression model from the scikit learn library. In the next line, we initialize an object of the `LinearRegression` model, which will be trained using the `X_train` and the `y_train` dataset with the `fit()` method. Once this code is executed, we get the following output:

```
| Out[15]:  
| LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,  
| normalize=False)
```

If we see the preceding output, we can consider our model ready. We can test our `X_test` data and see the prediction of our model, as shown here:

```
| In[15]:  
| y_pred=linregression.predict(X_test)  
| y_pred
```

`y_pred` gives the following predicted values:

```
| Out[15]:  
| array([103015.20159796, 132582.27760816, 132447.73845174, 71976.09851258,  
|        178537.48221055, 116161.24230165, 67851.69209676, 98791.73374687,  
|        113969.43533012, 167921.0656955 ])
```

Computing the slopes and intercepts

The intercept can be found by using the following code:

```
| In[16]:  
| linregression.intercept_
```

The output is as follows:

```
| Out[16]:  
| 42554.16761773238
```

The coefficients, or the slope, can be found as follows:

```
| In[17]:  
| linregression.coef_  
  
| Out[17]:  
| array([ 7.73467193e-01,  3.28845975e-02,  3.66100259e-02, -9.59284160e+02,  
|        6.99369053e+02])
```

From the preceding output, we can see that we have five different slope values for five different independent features. Note that we have also converted our state feature into two columns with dummy variables. In total, we have five different features. We have determined the slope or coefficient for each of these.

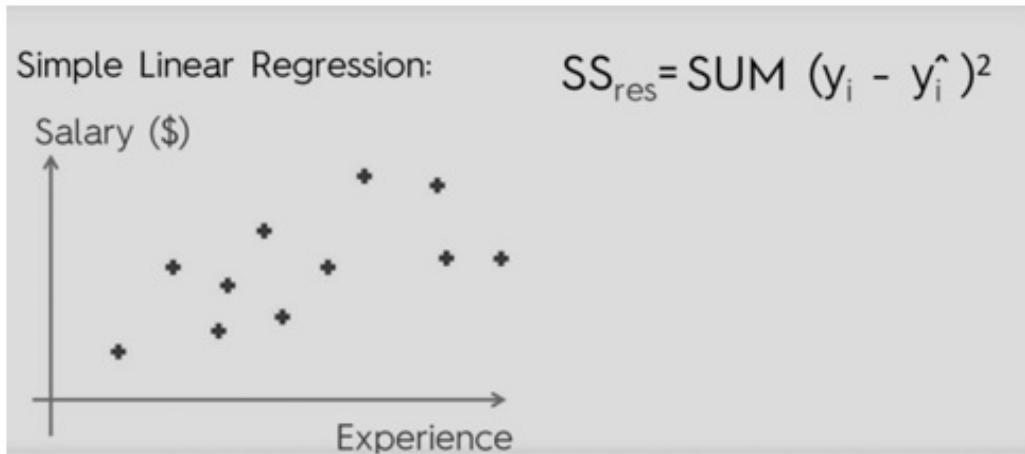
Evaluating the regression model using the R square value

R square is a very important technique to evaluate how good your model is. Usually, the value of R square ranges between zero and one. The closer the value of R square is to one, the better your model.

R square is calculated using two parameters:

- The square sum of the residuals (the error), denoted as SSres
- The square sum of the mean, denoted as SStot

Let's explore the square sum of the residuals. To understand this better, consider the following example:



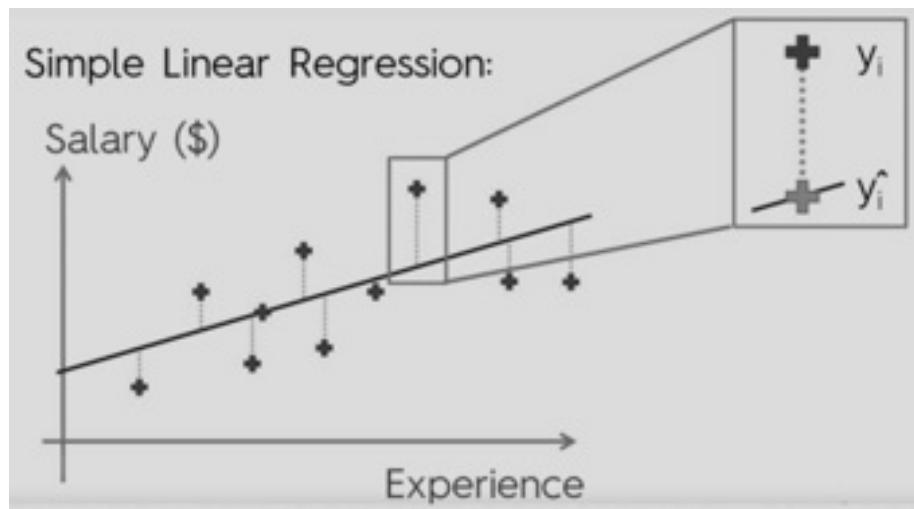
In the preceding diagram, we have a dataset that shows salary versus experience. We can see that the salary increases with years of experience. Now, let's find the line of best, which is represented by the following equation:

$$y = b_0 + b_1 * x$$

↓

$$\text{Salary} = b_0 + b_1 * \text{Experience}$$

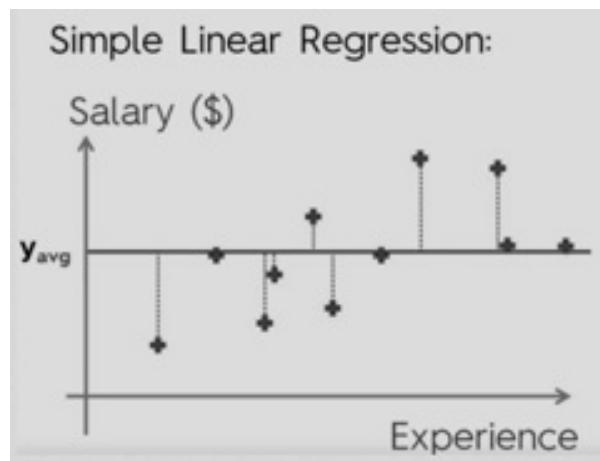
After we find the line of best using the preceding equation, it can be plotted as follows:



SS_{res} is the summation of the difference between the real and the predicted output, as shown here:

$$SS_{res} = \text{SUM } (y_i - \hat{y}_i)^2$$

Similarly, SS_{tot} is the summation of the difference between the real and the mean of the real output, as shown here:



The following equation shows us the SS_{tot}:

$$SS_{tot} = \text{SUM } (y_i - y_{avg})^2$$

Finally, the R square value for a regression model is given by the following formula:



Let's find the R square value with Python code for the `startups_invest.csv` file that we created earlier. Import the `r2_score` library from `sklearn`, as shown here:

```
In[18]:  
from sklearn.metrics import r2_score
```

Initialize `r2_score` and provide the `y_test` and `y_pred` parameters to get the r square value:

```
In[19]:  
score=r2_score(y_test,y_pred)  
score
```

```
| Out[19]  
| 0.93470684732824227
```

Here, we get an output of 0.93, which is very close to 1. This indicates that the regression model that we created is a very good model.

Now, let's explore another machine learning algorithm: decision trees.

Decision trees – a way to solve non-linear equations

Decision trees are also supervised machine learning algorithms and they can be used to solve both linear and non-linear regression and classification problems. The main aim of the decision tree is to predict targeted values or classes based on the inferences and the rules that are created by learning from the training data.

A decision tree algorithm usually solves a problem by constructing a tree representation. Every internal node that is present inside the tree represents an independent feature, and the leaf node corresponds to a target variable or the dependent variable.

The decision tree uses a divide-and-conquer strategy to select the leaf nodes and create a decision tree. Here is the pseudo algorithm to implement the decision tree:

1. Select the best independent feature or the attribute using a concept called **entropy** and **information gain**, then select it as the root node.
2. Split or divide the dataset into subsets. Subsets should be made in such a way that all the values are split, based on the data belonging to the attributes.
3. Repeat steps 1 and 2 on every subset until we get the root node or the leaf node.

The algorithm that is used to construct a decision tree is called an ID3 algorithm.

The ID3 algorithm has the following steps:

```
split(node, {examples}):
```

1. A<- is the best attribute for splitting the examples of the node
2. Select the decision attribute for this node, <-A
3. For each value of A, create a new child node

4. Split training {examples} to child nodes

5. For each child node/subset:

```
|   if the subset is pure: Stop the splitting  
|   else: Split(child_node, {subset})
```

Let's explore the concepts of entropy and information gain.

Entropy helps us to measure the purity of the splitting based on the attributes selected. Entropy is given by the following formula:

Entropy

- **Entropy:** $H(S) = - p_{(+)} \log_2 p_{(+)} - p_{(-)} \log_2 p_{(-)}$ bits
 - S ... subset of training examples
 - $p_{(+)} / p_{(-}$... % of positive / negative examples in S
- **Interpretation:** assume item X belongs to S
 - how many bits need to tell if X positive or negative
- **impure (3 yes / 3 no):**
$$H(S) = -\frac{3}{6} \log_2 \frac{3}{6} - \frac{3}{6} \log_2 \frac{3}{6} = 1 \text{ bits}$$
- **pure set (4 yes / 0 no):**
$$H(S) = -\frac{4}{4} \log_2 \frac{4}{4} - \frac{0}{4} \log_2 \frac{0}{4} = 0 \text{ bits}$$

If the entropy value is nearer to 0, we would be considering the split based on the attribute as a pure subset.

After calculating the Entropy for each and every attributes or the independent features we can calculate the **information gain**. Information gain is used to select the attributes that need to be selected when the decision tree is created.

Information gain is given by the following formula:

$$Gain(T, X) = Entropy(T) - Entropy(T, X)$$

Let's implement the same dataset example that we discussed in the *Multiple linear regression in Python using the scikit-learn library* section.

We will consider a new dataset in this section, which is `startups_Invest.csv`. This dataset has five features, as shown here:

	A	B	C	D	E	F
1	R&D Spend	Administration	Marketing Spend	State	Profit	
2	165349.2	136897.8	471784.1	New York	192261.83	
3	162597.7	151377.59	443898.53	California	191792.06	
4	153441.51	101145.55	407934.54	Florida	191050.39	
5	144372.41	118671.85	383199.62	New York	182901.99	
6	142107.34	91391.77	366168.42	Florida	166187.94	
7	131876.9	99814.71	362861.36	New York	156991.12	
8	134615.46	147198.87	127716.82	California	156122.51	
9	130298.13	145530.06	323876.68	Florida	155752.6	
10	120542.52	148718.95	311613.29	New York	152211.77	
11	123334.88	108679.17	304981.62	California	149759.96	
12	101913.08	110594.11	229160.95	Florida	146121.95	
13	100671.96	91790.61	249744.55	California	144259.4	
14	93863.75	127320.38	249839.44	Florida	141585.52	
15	91992.39	135495.07	252664.93	California	134307.35	
16	119943.24	156547.42	256512.92	Florida	132602.65	
17	114523.61	122616.84	261776.23	New York	129917.04	
18	78013.11	121597.55	264346.06	California	126992.93	

The dataset consists of data related to 50 start-up companies who have invested some amount of money in various internal departments, such as R&D, marketing, and administration, from various states. Based on this expenditure, the company has achieved some profit, which is specified in the `Profit` column. Our goal will be to create a multiple linear regression model that will be able to predict the Profit based on the expenditure in the R&D, marketing, and administration departments from various states. `R&D Spend`, `Marketing Spend`, `Administration`, and `State` are the independent features in this example, while `Profit` is the dependent feature.

We will perform the following steps to create a multiple linear regression model using Python and the `sklearn` library:

1. Import the required libraries:

```
In[1]:
import numpy as np
import matplotlib.pyplot as plt
```

```
| import pandas as pd
```

2. Read and split our dataset into dependent and independent features:

```
| In[2]:  
| dataset = pd.read_csv('Startups_Invest.csv')  
| X = dataset.iloc[:, :-1]  
| y = dataset.iloc[:, 4]
```

Here, x represents the independent features and y is the dependent feature. The following code helps us see the top five records of the x variable:

```
| In[3]:  
| X.head()
```

The output is as follows:

	R&D Spend	Administration	Marketing Spend	State
0	165349.20	136897.80	471784.10	New York
1	162597.70	151377.59	443898.53	California
2	153441.51	101145.55	407934.54	Florida
3	144372.41	118671.85	383199.62	New York
4	142107.34	91391.77	366168.42	Florida

Similarly, the following code helps us see the top five records of the y variable:

```
| In[4]:  
| y.head()
```

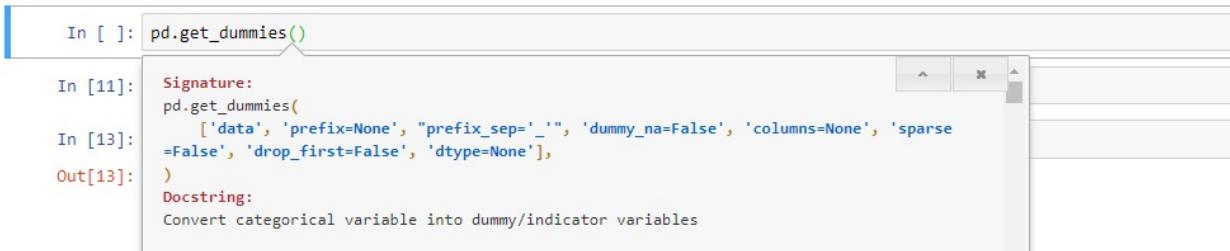
The output is as follows:

```
0    192261.83  
1    191792.06  
2    191050.39  
3    182901.99  
4    166187.94  
Name: Profit, dtype: float64
```

The `State` column is a categorical feature, so we need to convert the categorical

feature into dummy variables using pandas. Our final regression model will not be able to understand categorical features, since they are in the form of strings. The regression model usually understands mathematical calculations using various algorithms, so we need to convert the categorical variables into dummy variables. To do this, we will be using a pandas function, `get_dummies()`.

Here is the syntax of the `get_dummies()` function:



The screenshot shows a Jupyter Notebook interface. In the code editor, the command `pd.get_dummies()` is typed. A tooltip or info box is displayed over the command, providing the following information:

- In [11]:** `pd.get_dummies()`
- Signature:** `pd.get_dummies(data, prefix=None, prefix_sep=' ', dummy_na=False, columns=None, sparse=False, drop_first=False, dtype=None)`
- Docstring:** Convert categorical variable into dummy/indicator variables

The parameters can be described as follows:

- `data`: This can be an array, a series, or a DataFrame.
- `prefix`: This can be a string or a dictionary of strings. By default, it is `None`.
- `columns`: This is a list of the columns specified.
- `drop_first`: This is a Boolean value and by default it is a `False` value. We need to specify whether to get K-1 dummies out of the K categorical levels by removing the first level.

Here is the code that helps us to convert the `state` column into dummy variables:

```
| In[5]:  
| State=pd.get_dummies(X.iloc[:,3],drop_first=True)
```

Inside the `get_dummies` function, we have provided the State column using `iloc` indexing and have set the `drop_first` parameter as `True` to get only K-1 dummies out of the K categorical variables from the State column. Here, K is the total number of categorical variables inside the `state` column. Currently, we have three unique States, which are New York, California, and Florida. We choose only K-1 categories because this is the minimum number required to represent all K categories. We know that if the state is neither Florida nor New York, for example, as is the case in the second record in the following output, it must be California.

The following code helps us to see the top 10 records of the dummy variables:

```
| In[6]:  
| State.head(10)
```

The output is as follows:

	Florida	New York
0	0	1
1	0	0
2	1	0
3	0	1
4	1	0
5	0	1
6	0	0
7	1	0
8	0	1
9	0	0

Since we have converted the `State` column into categorical features, we will be deleting the `State` column, as it is no longer required. This can be done as follows:

```
| x.drop('State', axis=1, inplace=True)
```

Let's check the `x` independent variable to see whether the `State` column has successfully been dropped:

```
| In[8]:  
| x.head()
```

The output is as follows:

	R&D Spend	Administration	Marketing Spend
0	165349.20	136897.80	471784.10
1	162597.70	151377.59	443898.53
2	153441.51	101145.55	407934.54
3	144372.41	118671.85	383199.62
4	142107.34	91391.77	366168.42

We can see that the State column has been dropped successfully.

We need to concatenate the dummy variables that were created from the state feature, which is stored in the state variable. To concatenate them, we use the concat() function that's present in the pandas library. The code is as follows:

```
| In[9]:  
| X=pd.concat([X,State],axis=1)
```

Let's check the updated independent feature, x:

```
| In[10]:  
| X.head()
```

The output looks as follows:

	R&D Spend	Administration	Marketing Spend	Florida	New York
0	165349.20	136897.80	471784.10	0	1
1	162597.70	151377.59	443898.53	0	0
2	153441.51	101145.55	407934.54	1	0
3	144372.41	118671.85	383199.62	0	1
4	142107.34	91391.77	366168.42	1	0

We can see that the dummy variable columns have been added to the x variable. With this, we have completed the data-preprocessing step. The next step is to

divide the independent features, `x`, and the dependent feature, `y`, into training and test data using the scikit learn library.

First, we import the `sklearn` library, as shown here:

```
| In[11]:  
| from sklearn.model_selection import train_test_split
```

The following code splits the independent and dependent features into training and test data:

```
| In[12]:  
| X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=42)
```

Finally, we apply the decision tree regressor model by using the scikit learn library:

```
| In[14]:  
| from sklearn.tree import DecisionTreeRegressor  
| decisionregressor=DecisionTreeRegressor()  
| decisionregressor.fit(X_train,y_train)
```

Here, we import the decision tree regressor and then we initialize an object and fit the object with the `X_train` and `y_train` values. The output is as follows:

```
| Out[14]:  
| DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,  
|   max_leaf_nodes=None, min_impurity_decrease=0.0,  
|   min_impurity_split=None, min_samples_leaf=1,  
|   min_samples_split=2, min_weight_fraction_leaf=0.0,  
|   presort=False, random_state=None, splitter='best')
```

The next step is that we do the prediction using the `DecisionTree` regressor object. We will be using the `predict` method:

```
| In[15]:  
| y_pred=decisionregressor.predict(X_test)  
| y_pred
```

The output of `y_pred` is shown here:

```
| Out[15]:  
| array([101004.64, 141585.52, 141585.52, 78239.91, 182901.99, 105733.54,  
|       71498.49, 97427.84, 105733.54, 156122.51])
```

Check the r square value, which indicates how good our model is. Import the `r2_score` library:

```
| In[16]:  
| from sklearn.metrics import r2_score
```

Use this `r2_score` library to find the `r2_score` value:

```
| In[17]:  
| score=r2_score(y_test,y_pred)  
| score
```

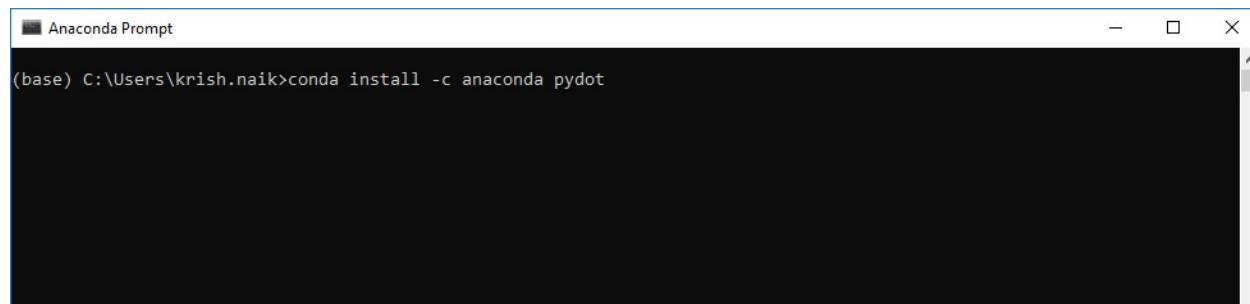
The output is as follows:

```
| Out[17]:  
| 0.9751842669296727
```

The preceding `r2_score` is very close to 1 and hence it is a very good model.

Now, we will check out the visualization of the decision tree using some of the libraries that are present in Python.

Install the `pydot` library to visualize the decision tree. Open the Anaconda prompt and type the following command:



Import the following libraries:

```
| In[19]:  
| from IPython.display import Image  
| from sklearn.externals.six import StringIO  
| from sklearn.tree import export_graphviz  
| import pydot
```

These libraries are used to visualize the decision trees.

After importing, we will consider all the independent features on which the decision tree was applied. Use the following code:

```
| In[20]:  
| features = list(X_train)  
| features
```

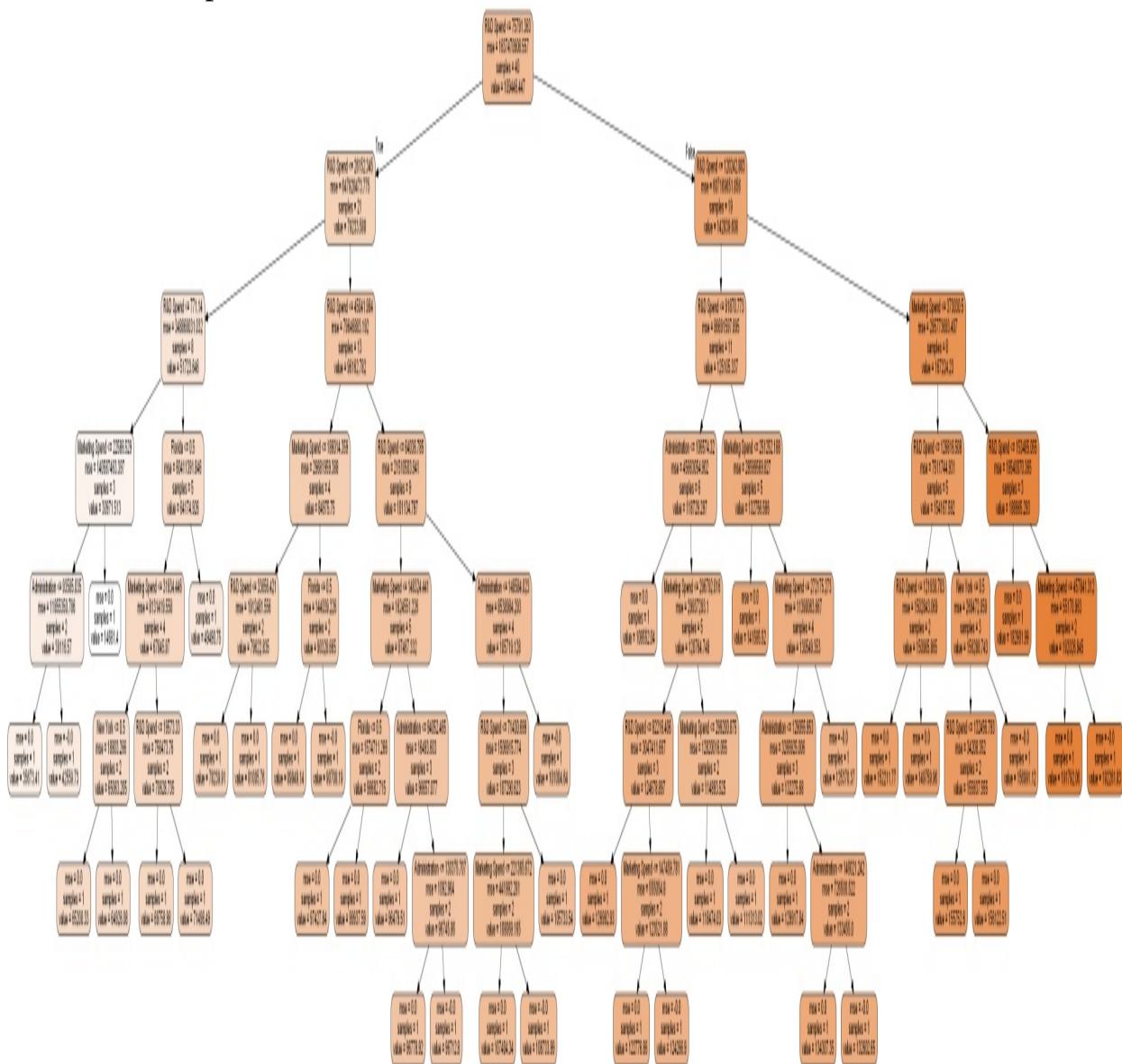
The output is as follows:

```
| Out[20]:  
| ['R&D Spend', 'Administration', 'Marketing Spend', 'Florida', 'New York']
```

This code helps us to create the decision tree graph in Jupyter Notebook:

```
| In[21]:  
| dot_data = StringIO()  
| export_graphviz(decisionregressor, out_file=dot_data, feature_names=features, filled=True,  
| graph = pydot.graph_from_dot_data(dot_data.getvalue())  
| Image(graph[0].create_png())
```

Here is the output:



The preceding graph shows the decision tree that was created by the decision tree regressor. The diagram will be more visible in the Jupyter Notebook.

We have many other machine learning algorithms, such as `RandomForestRegressor` and `XGBoostRegressor`, that perform much better than the decision tree.

Summary

In this chapter, we explored linear regression, which includes both simple linear regression and multivariate linear regression. We also discussed how we can compute the intercept and the coefficient, which is indicated as beta. Based on these parameters, we were able to find the line of best fit, which represents our predicted output. Finally, after we created the regression model, we saw how to evaluate the regression model using R square intuition, which included both the **Square Sum of the Residuals (SSres)** and the **Square Sum of the mean (SStot)**.

We saw how we can implement the decision tree regressor and perform the prediction, and we looked at how we can visualize the decision tree that was created by `DecisionTreeRegressor`.

In the next chapter, we will discuss the Monte Carlo simulation and decision making.

Section 3: Deep Learning and Monte Carlo Simulation

In this section, we will discuss Monte Carlo simulation and explore deep learning techniques implemented using neural networks. These topics will be implemented using Python and some of the libraries that are available in Python. The libraries that we will be using include the StatModel library, TensorFlow, and Keras.

This section consists of the following chapters:

[Chapter 8](#), *Monte Carlo Simulations for Decision Making*

[Chapter 9](#), *Option Pricing – the Black Scholes Model*

[Chapter 10](#), *Introduction to Deep Learning with TensorFlow and Keras*

[Chapter 11](#), *Stock Market Analysis and Forecasting Case Study*

[Chapter 12](#), *What Is Next?*

Monte Carlo Simulations for Decision Making

Monte Carlo simulation is a computerized mathematical technique to generate random data based on a known distribution for numerical experiments. This method is applied to risk quantitative analysis and decision-making problems. It is used by professionals in a variety of different industries, including finance, project management, energy, manufacturing, engineering, research and development, insurance, oil and gas, and transportation.

Monte Carlo simulation was first used by scientists working on the atom bomb in 1940 (<https://www.solver.com/press/monte-carlo-methods-led-atomic-bomb-may-be-your-best-bet-business-decision-making>). It is now used in situations in which we need to make estimates for uncertain decisions, such as weather forecast predictions. In this chapter, we will be using Monte Carlo simulation for predicting the gross profit of a company and for forecasting stocks.

In this chapter, we will be focusing on the following topics:

- An introduction to Monte Carlo simulation
- Using Monte Carlo simulation to predict gross profit in Python
- Using Monte Carlo simulation to forecast stock prices in Python

Technical requirements

In this chapter, we will be using the Jupyter Notebook for coding purposes. We will also be using the pandas, NumPy, matplotlib, and SciPy libraries.

The GitHub repository for this chapter can be found at the following link: <https://github.com/PacktPublishing/Hands-on-Python-for-Finance/tree/master/Chapter%208>.

An introduction to Monte Carlo simulation

Monte Carlo simulation is an important tool that has a variety of applications in the world of business and finance. When we run a Monte Carlo simulation, we are interested in observing the different possible realizations of a future event. What happens in real life is just one of the possible outcomes of any event. Let's consider an example. If a basketball player shoots a free throw at the end of a game and the game is tied, there are two possibilities:

- They score and win the game
- They don't score

However, this doesn't tell us much about the player's chances of scoring the free throw. This is where Monte Carlo simulation comes in handy. We can use past data to create a simulation that is in fact nothing but a new set of fictional, but sensible, data.

These realizations are generated by observing the distribution of historical data and calculating its mean and variance. The new dataset that is generated is larger than the initial dataset and it allows us to see what would have happened if the player had to shoot the final free throw 1,000 times, for example. This information is valuable, as it allows us to consider the probability of different outcomes, thereby helping us to make an informed decision.

Monte Carlo simulations are used in corporate finance, investment valuation, asset management, risk management, estimating insurance liabilities, pricing of options, and other derivatives. The large level of uncertainty in finance makes Monte Carlo a valuable tool that improves the decision-making process when several random variables are in play. As we move on, we will learn about the mechanics of a Monte Carlo simulation and how it can be applied in the world of finance, looking in particular at two practical examples.

Monte Carlo simulation applied in the context of corporate finance

Finance managers face a wide variety of tough tasks in their work. They must forecast the development of revenues, the costs of goods sold, and the operating expenses. Each value is affected by many factors, the behavior of which can be considered random. This is the reason that Monte Carlo simulation, which reduces the complexity and improves the decision-making process, can be helpful.

Let's consider an example to see how Monte Carlo can be applied in a corporate finance context. Usually, the revenue of any company for the current year is equal to last year's revenue, multiplied by one plus the growth rate of the revenue. This is represented by the following equation:

$$\text{Current revenue} = \text{Last year's revenue} * (1 + \text{year-on-year growth rate})$$

Usually, the value of last year's revenue is not available. The random variable, which can have any value in this case, is the revenue growth rate. The computer software would allow us to simulate or predict the development of the revenue, say, 1,000 times, and help us to obtain an idea of the average, the maximum, and the minimum values of the expected revenue figure.

The following diagram show how we can calculate the current revenue:

Current Revenues = Last Year Revenues * (1 + y-o-y growth rate)



Run 1,000 simulations
of growth rate



Get an idea of average, maximum
and minimum values

udemy

This can be really useful for a finance manager, as it would allow them to understand the overall direction of where the company is heading and the maximum and the minimum amount of revenue that they can expect. The two parameters for revenue growth and standard deviation could be obtained by looking at historical data or arbitrarily chosen data.

The logic behind determining the **cost of goods sold (COGS)** and the **operating expenses (Opex)** is almost the same; both COGS and Opex are expenditures that companies incur when running their businesses. However, there is one main difference, which is that the expenses are segregated on the income statement. Opex and COGS measure different ways in which resources are spent in the process of running a company.

We need to represent COGS and Opex as a percentage of the revenue and then model how the percentage of the revenue changes over time. This would allow us to obtain figures to do with COGS and Opex that are in line with the firm's expected revenue, as both COGS and Opex depend on the revenue the firm makes. If the revenues are low, the value of COGS will be lower, as fewer products will be produced. When we deduct COGS from the revenue of a company, you obtain its gross profit. In the next section, we will learn how to estimate a firm's gross profit with a Monte Carlo simulation.

Using Monte Carlo simulation to predict gross profit in Python

In this section, we will see how we can use the Monte Carlo computer simulation to project a company's future revenue and expense using Python. This section will be divided into two parts.

Using Monte Carlo simulation to predict gross profit – part 1

We will start by importing two important libraries that will help us with standard financial analysis. These two libraries are `numpy` and `matplotlib`. We will import them as follows:

```
In[1]:  
import numpy as np  
import matplotlib.pyplot as plt
```

Our goal in this exercise is to predict the firm's future gross profit. We will need the values of the expected revenue and the expected COGS. We will start by performing 1,000 simulations of the company's expected revenue.

Let's assume that we have a value for last year's revenue and we that have an idea about the revenue growth rate we can expect. The expected revenue for this year, therefore, is \$170 million, the standard deviation of which would be \$20 million. To simplify things, let's work in millions of dollars. We are going to create two variables. The first is `rev_m`, which refers to the mean of the revenue, and is assigned the value `170`. The other variable will be `rev_std`, which refers to the standard deviation of the revenue, and is assigned the value `20`. The next important value we need to specify is the number of iterations we intend to produce, which is 1,000. This is defined in the `iterations` variable. Let's create these three variables, as shown here:

```
In[2]:  
rev_m = 170  
rev_std = 20  
iterations = 1000|
```

From the preceding code, we can see that `rev_m` is assigned to `170`, `rev_std` is assigned to `20`, and `iterations` is assigned to `1000`.

The next line of code will produce a simulation of the future revenues. We will apply NumPy's random normal distribution generator. The arguments we provide in the NumPy random function are the expected mean of the revenue (`rev_m`), the standard deviation (`rev_std`), and the number of iterations we would like to

perform (`iterations`). The code is as follows:

```
In[3]:  
rev = np.random.normal(rev_m, rev_stdev, iterations)  
rev
```

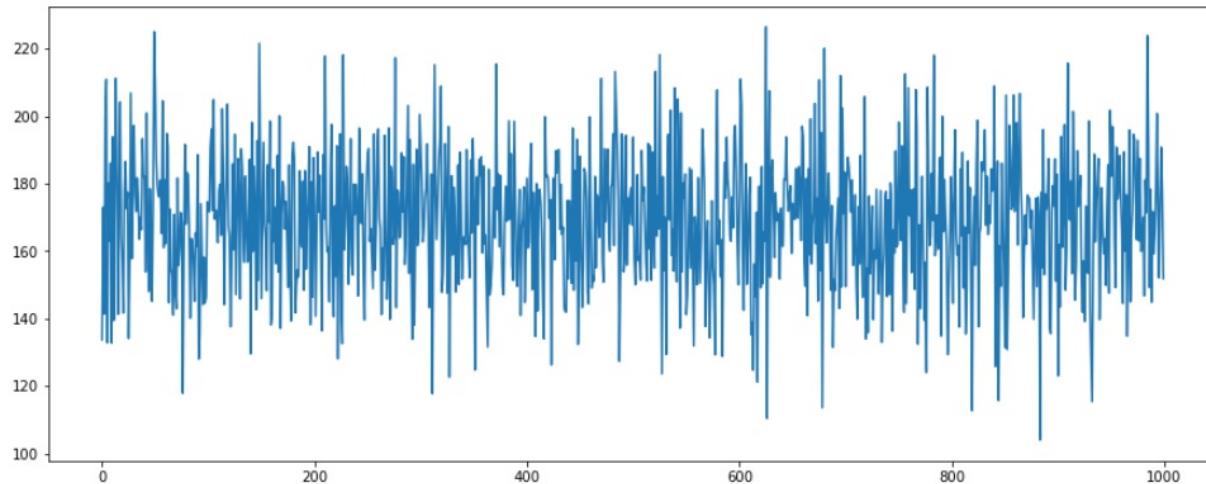
We will get the following output:

```
array([ 133.79693835,  172.97868561,  141.43723108,  199.23821641,  
       210.8605806 ,  132.90910444,  180.08930132,  163.10107918,  
       186.05902844,  132.85090004,  193.80007128,  139.52190143,  
       180.12598103,  211.19083272,  181.58539953,  141.33382792,  
       177.75007975,  204.16769908,  164.34016401,  156.41764005,  
       141.73981877,  177.61608059,  186.57000934,  172.55932015,  
       177.41917122,  134.17937117,  156.94424026,  206.83338497,  
       157.77576601,  184.66683437,  197.22894686,  176.79242667,  
       171.7299541 ,  181.68384217,  176.2924201 ,  163.50211098,  
       170.02954262,  166.38374969,  193.25911787,  182.17944735,  
       182.10794939,  153.91226572,  200.88724167,  168.37079575,  
       148.13079166,  178.4406263 ,  166.52532845,  145.19754625,  
       173.03396581,  224.96209297,  209.81061555,  187.54479113,  
       179.44246608,  176.51511245,  176.08070867,  181.03316917,  
       165.26930604,  204.47021424,  161.05455959,  181.13559136,  
       162.26881501,  194.84456269,  191.07437943,  144.77394228,  
       172.44369555,  154.95962652,  153.02147588,  141.10778724,  
       170.92063558,  148.14042234,  142.96519071,  181.65309926,  
       155.76306277,  160.70494417,  171.40170232,  168.60294914,
```

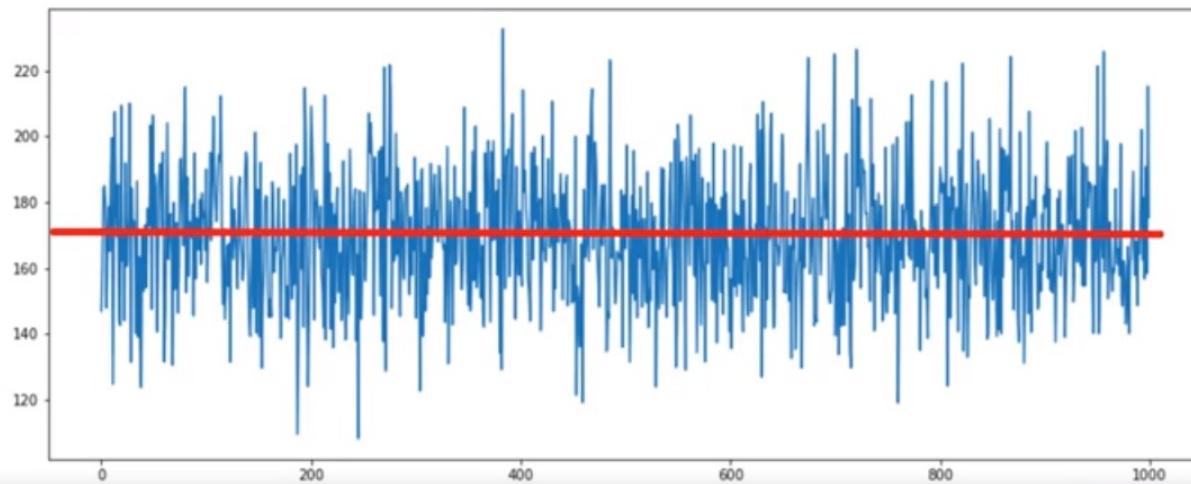
From the preceding output, we can see that we have created 1,000 random values. Most of these values are close to the mean we selected. In the next step, we will try to plot these observations and see their distributions using the following code:

```
In[4]:  
plt.figure(figsize=(15, 6))  
plt.plot(rev)  
plt.show()
```

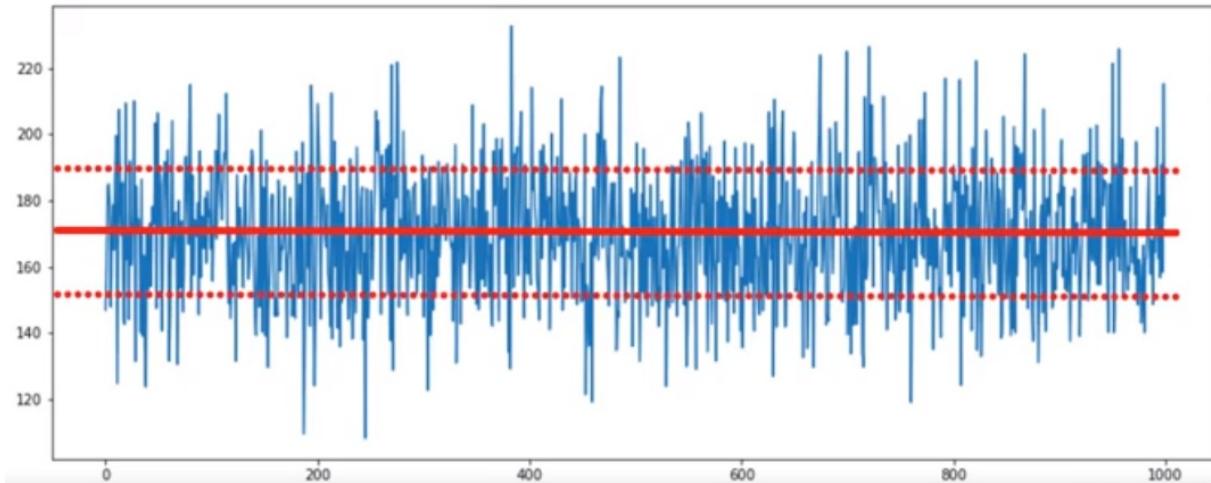
The output is as follows:



From the preceding output, we can see that the simulated values are centered on the mean of 170, which is represented in the y axis of the graph. The value of 170 is shown in the following graph:



We can see that all data points fall in the region of 150 to 190, as shown here:



The values of the 150 and 190 range are within the first standard deviation of the mean, which is 170.

This is how we can interpret the distribution of the expected revenue and find out the impact of COGS. Let's see how we can simulate this development. Let's assume that we are experienced in this business and able to tell that the typical COGS amount is approximately 60% of the company's annual revenue.

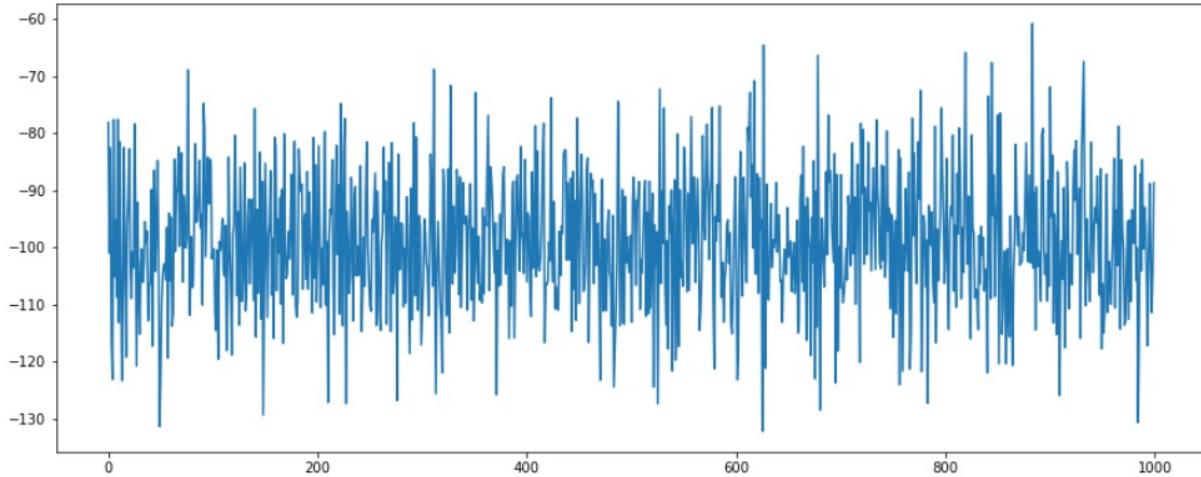
To work this out, we need to know the previous values of COGS. Let's say that COGS was once equal to 55% of the revenue and then it went up to 62%, before going up again to 63%, and then finally coming back down to 55%. It is reasonable to determine that this is a normal distribution with a mean of 60% and that this estimation would have a standard deviation of 10%. It doesn't matter that COGS was actually deviating with 6% of the revenue value. To set the distribution, the distribution should deviate by 10% from the mean or the average of COGS.

COGS is the money spent, so we need to put a minus sign first, and then the expression must reflect the multiplication of the revenue by 60%. We need to pay attention here, because what comes next is crucial for this analysis. We will not simulate COGS 1,000 times. This is because this simulation has already been done for the revenue in line `In[3]` of the code, and we have already generated 1,000 revenue data points. We must assign a random COGS value to each one of these points. COGS is a percentage of the revenue, which is why the revenue value we obtained must be multiplied by a value extracted from a random normal distribution with a mean of 0.6 and a standard deviation of 0.1. Had we put 0.6 directly, this would have meant that COGS always equals 60% of the revenue, and this is not always the case. The percentage will probably vary, and we have decided that the standard deviation is equal to 10%. NumPy allows us to incorporate the expected deviation, so we will take advantage of this in the following code:

```
In[5]:  
COGS = - (rev * np.random.normal(0.6, 0.1))  
  
plt.figure(figsize=(15, 6))  
plt.plot(COGS)  
plt.show()
```

When we plot the results on a graph, we see the typical behavior of the normal

distributions. The normal distribution basically follows a bell-curved shape:



It is interesting that if you re-run the COGS approximation, you will not always get the same mean value for the observations.

Let's find the mean:

```
| In[6]:  
| COGS.mean()
```

The output is as follows:

```
| Out[6]:  
| -98.787602431688725
```

The standard deviation can be computed as follows:

```
| In[7]:  
| COGS.std()  
|  
| Out[7]:  
| 11.589576145999802
```

It is important that the deviation of COGS is around 10% of its mean at all times.

In this part, we have simulated all the variables. In the next part, we will see how we can calculate the future gross profit.

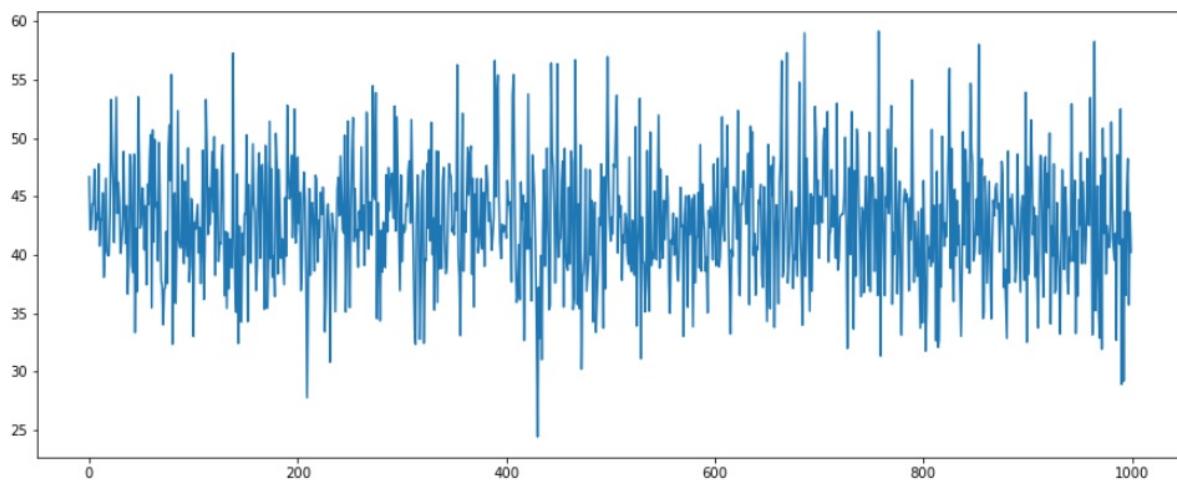
Using Monte Carlo simulation to predict gross profit – part 2

Computing the gross profit is the objective of the second part of this lesson. We have generated 1,000 potential values for both the revenue and COGS. Calculating the gross profit requires us to combine these values.

The code is as follows. Here, we are combining the revenue and COGS and plotting the graph:

```
In[8]:  
Gross_Profit = rev + COGS  
Gross_Profit  
  
plt.figure(figsize=(15, 6))  
plt.plot(Gross_Profit)  
plt.show()
```

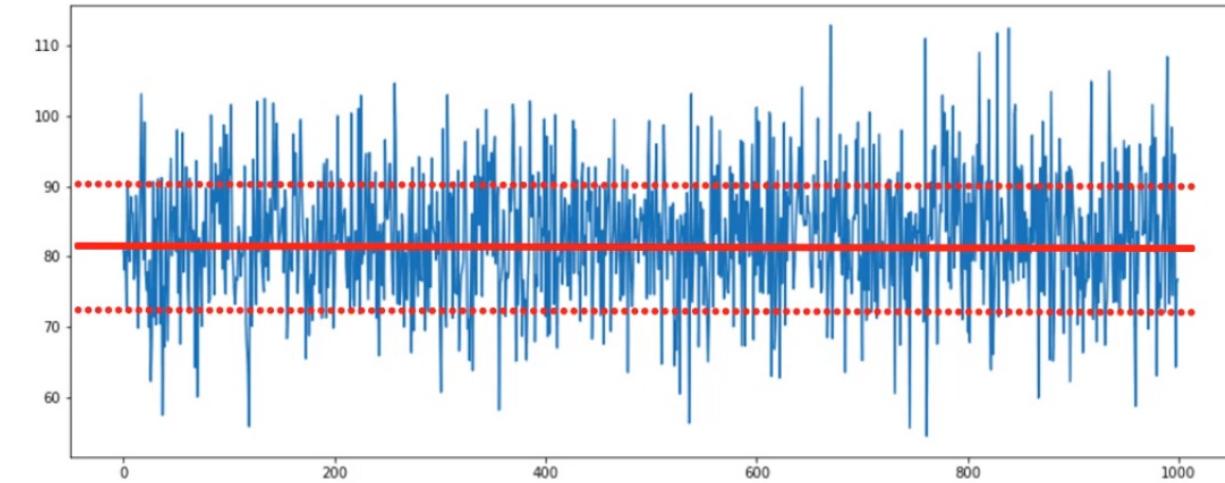
The output is as follows:



From the preceding output, we can agree that the distribution is normally distributed with a mean value that is equal to the difference between the means of the revenues and the mean of COGS.

With the help of the `max` and `min` functions, it is easy to obtain the biggest and the

smallest potential values of gross profit:



The following code helps us to find the maximum gross profit:

```
| In[9]:  
| max(Gross_Profit)
```

The output is as follows:

```
| Out[9]:  
| 59.175304646744934
```

We can also find the minimum gross profit, as follows:

```
| In[10]:  
| min(Gross_Profit)  
  
| Out[10]:  
| 24.388073332020554
```

The `mean()` and `std()` methods can provide an output that is equal to the mean and the standard deviation of the gross profit, respectively. The mean can be computed as follows:

```
| In[11]:  
| Gross_Profit.mean()
```

Therefore, the mean of `Gross_Profit` is as follows:

```
| Out[11]:  
| 42.783571075126019
```

Similarly, the standard deviation can be calculated as follows:

```
| In[12]:  
| Gross_Profit.std()
```

The standard deviation of the `Gross_Profit` is as follows:

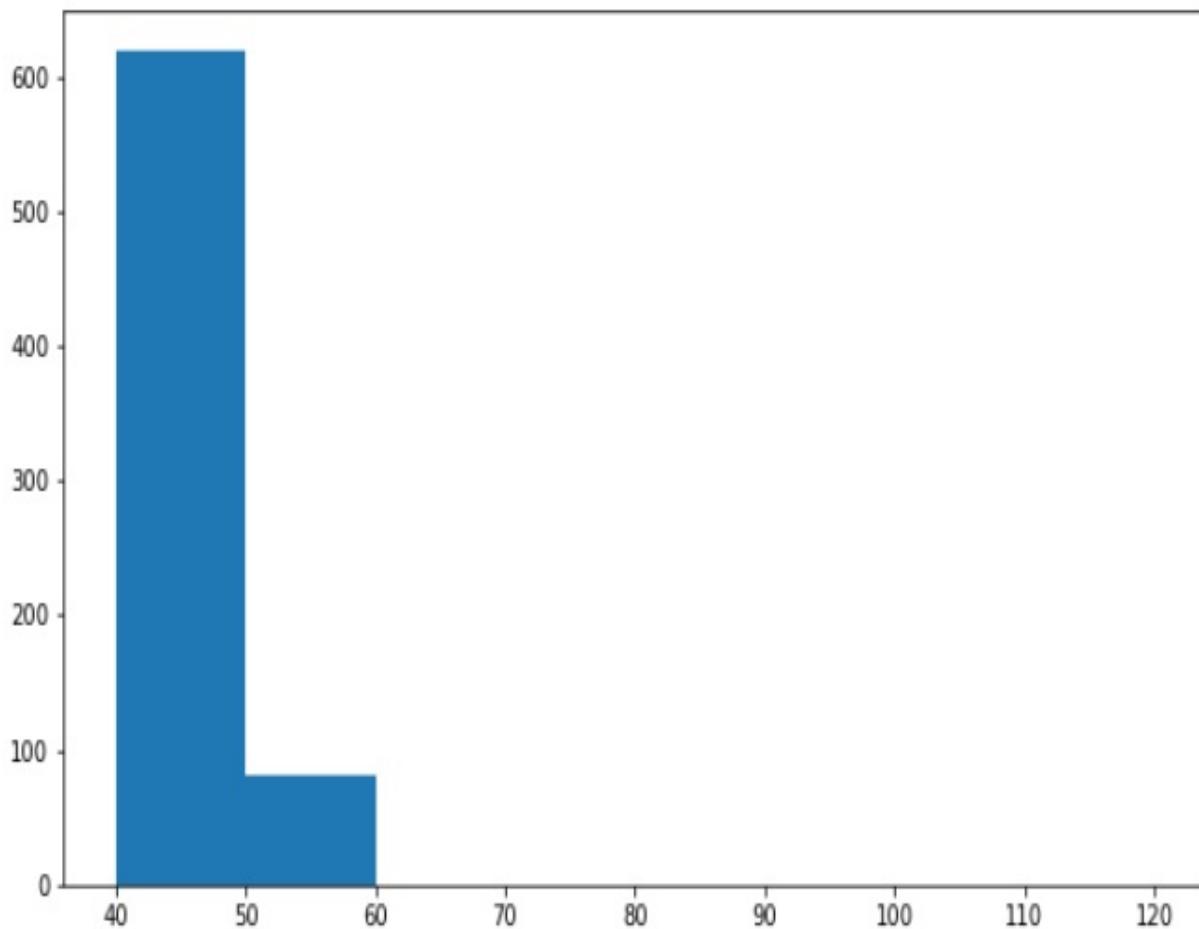
```
| Out[12]:  
| 5.1828669493999255
```

While we have completed our original task here, it is always a good idea to plot these results on a histogram. A histogram is a graph that helps you to identify the distribution of your output. The histogram syntax to be implemented is similar to the one we used for a regular plot. The histogram is created using the `hist()` function, which is present inside the `matplotlib` library. The syntax is as follows:

```
| In[13]:  
| plt.figure(figsize=(10, 6));  
| plt.hist(Gross_Profit, bins = [40, 50, 60, 70, 80, 90, 100, 110, 120]);  
| plt.show()
```

Here, `hist()` takes two parameters. One is the data, which is basically the gross profit, and the other is the bins. These are the chunks in which the data in the plot will be divided. There are basically two ways to use bins. One way is to create a list whose elements separates the bins along the `x` axis. If our list contains the numbers 40, 50, 60, and so on, up to 120, then we will see the observations between 40 and 50 grouped in one bin and the observations between 50 and 60 grouped in another bin, and so on.

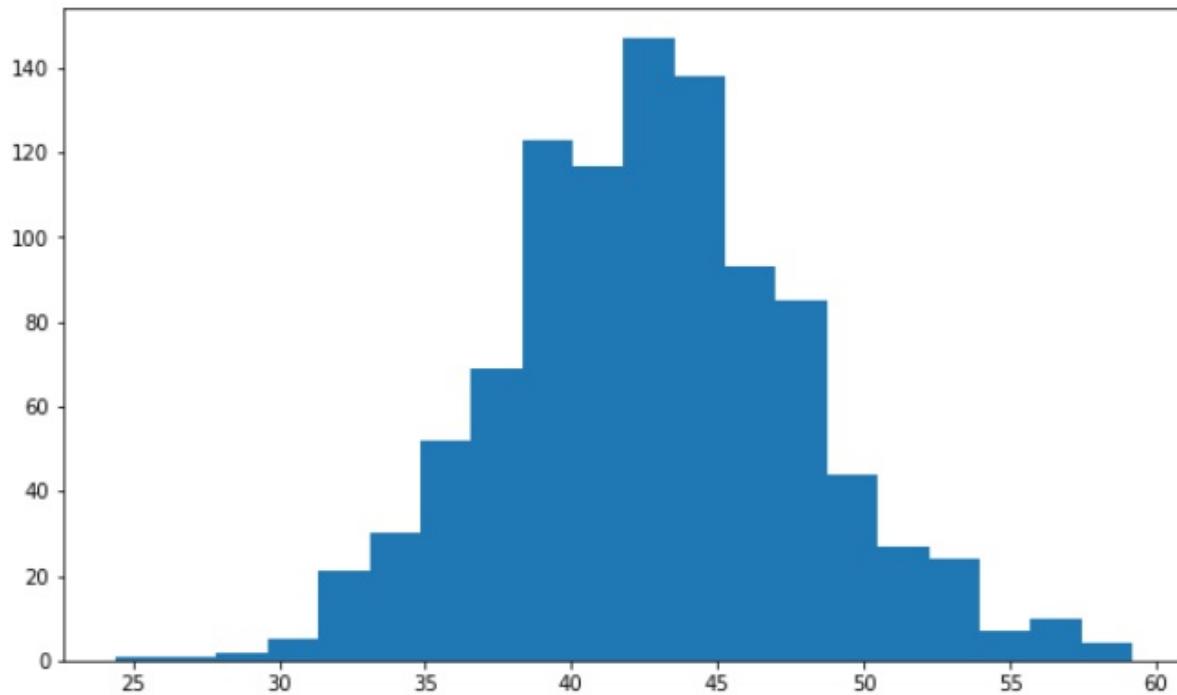
The histogram is as shown here:



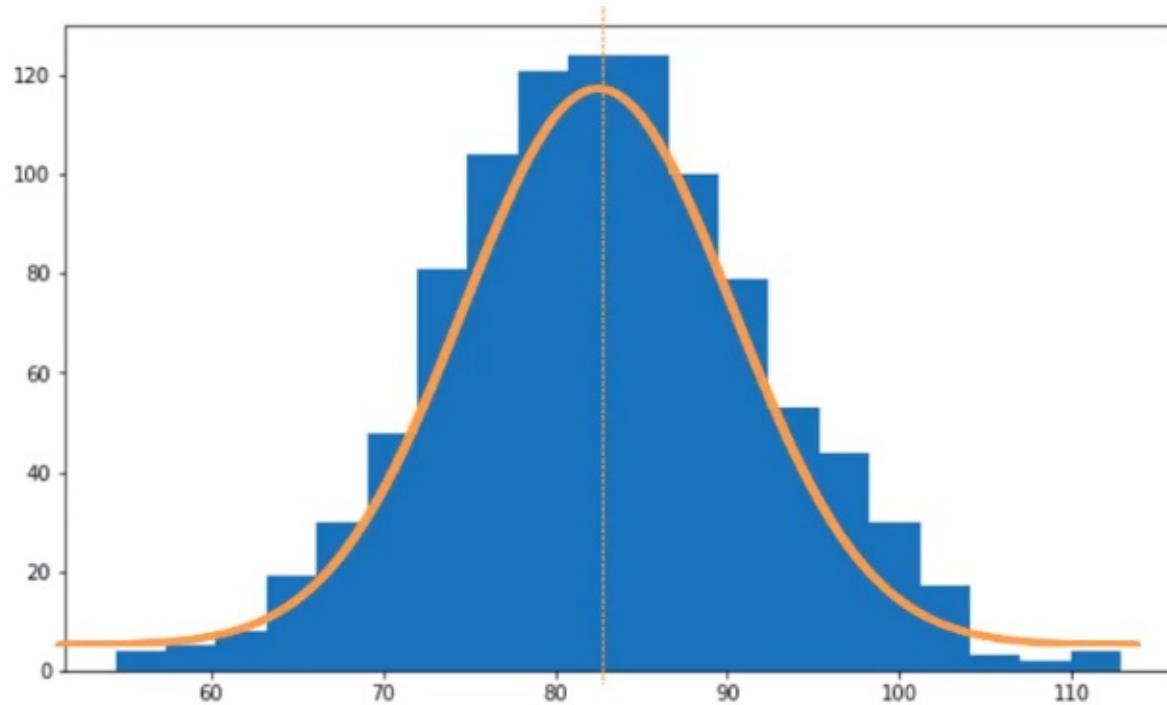
Here, the bins are too big and it is hard to see whether our data is normally distributed. Instead, we will assign the number of bins directly, as follows:

```
In[14]:  
plt.figure(figsize=(10, 6));  
plt.hist(Gross_Profit, bins = 20);  
plt.show()
```

The following output is obtained:



If we ask for 20 bins, the `hist()` function will automatically split our data into 20 equal intervals. This time, it will be obvious that the numbers are normally distributed, as shown in the following diagram:



This is a great example of a tool that is useful for a finance manager working in a big company, although there are complex distributions or procedures out there in the market. It is this kind of analysis that can help users to predict future revenues, costs, and margins.

Using Monte Carlo simulation to forecast stock prices in Python

In this section, we will see how a Monte Carlo simulation can be used to model the development of asset prices, such as stocks. The price of an equity share is something we have observed in the previous chapters. However, the future development of that share is currently unknown. The only information that we have is about past prices. Tomorrow, the company's shares could go up or down, who knows.

Take a look at the following formula, which is used to predict stock prices:

$$\begin{aligned} \text{Price Today} &= \text{Price Yesterday} * e^r \\ r &: \text{Log return of share price between yesterday and today} \end{aligned}$$

The preceding formula basically says that the price of a share today is equal to the price of the same share yesterday multiplied by e^r , where r is the log return of the share.

Remember the algebra principle, according to which e^r gives us the number we are taking a logarithm for. Here, r is equal to the natural logarithm of today's price divided by yesterday's price.

The following diagram shows how to calculate a share price:

$$\text{Price Today} = \text{Price Yesterday} \times e^r$$

r : log return of share price between yesterday and today

$$e^{\ln(x)} = x$$

$$\ln(\text{price today} / \text{price yesterday})$$

$$\text{Price Today} = \text{Price Yesterday} \times e^{\ln(\text{price today} / \text{price yesterday})}$$

This is an equation we should feel confident with, because it allows us to depict today's stock price as a function of yesterday's stock price and the daily returns.

We usually know yesterday's stock price, but we do not know the value of r as it is a random variable. To determine the value of r , we will use a new concept called **Brownian motion**, which allows us to model this kind of randomness. The formula of Brownian motion is made up of two components:

- Drift
- Volatility (random variable)

Drift is the direction in which the rates of returns have headed in the past. It is the expected daily return of the stock and it is the best approximation about the future we have.

We will start by calculating the periodic daily returns of stocks over the historical period. We only have to take a natural logarithm of the ratio between the current and previous price. Once we have calculated the daily returns in the historical period, we can easily calculate their average standard deviation and variance. This would allow us to calculate the drift component, which is given

by the following formula:

$$drift = u - \frac{1}{2} \cdot var$$

The second component of a Brownian motion is the random variable, which is given by the following formula:

$$\text{Random Variable} = \sigma \times Z(\text{Rand}(0;1))$$

The random variable is given by the stock historical volatility specified by σ , which is multiplied by Z , which is in turn multiplied by a random number between 0 and 1. The random number from 0 to 1 represents a percentage.

If we assume that the expected future returns are distributed normally, Z multiplied by the percentage between 0 to 1 would give us the number of standard deviations away from the mean:



Number of Standard Deviations
away from the mean

$$\text{Random variable} = \sigma \times Z(\text{Rand}(0; 1))$$

We can select random variables using the preceding equation because statisticians have calculated the distance between the mean and the events that have a given probability of occurring between 0 and 1. Suppose, for example, that from the preceding equation, the distance between the means and events with a probability of less than 99.7% has a standard deviation of 3.

This means that the equation of the price of a stock for today is yesterday's price, multiplied by e to the power of the drift plus the random value, as shown here:

$$\text{Price Today} = \text{Price yesterday} * e^{(\mu - \frac{1}{2}\sigma^2) + \sigma Z[\text{Rand}(0;1)]}$$

If we repeat this calculation 1,000 times, we will be able to simulate the development of tomorrow's stock price and assets. The likelihood is that these will follow a certain pattern. This is a great way to assess the upside and the downside of the investment, as we have obtained an upper and a lower bound when performing the Monte Carlo simulation.

These are the mechanics you need to understand when using Monte Carlo for asset pricing. In the next section we will apply this technique in Python.

Using Monte Carlo simulation to forecast stock prices – part 1

In this section, we will be gaining further knowledge on how to implement Monte Carlo simulations in the real world. Here, we will be looking at how we can run or implement simulations, which, in turn, will help us to predict the stock price of any company.

First, we will import all the important libraries that we have already used, as shown here:

```
| In[1]:  
| import numpy as np  
| import pandas as pd  
| import matplotlib.pyplot as plt  
| from scipy.stats import norm  
| %matplotlib inline
```

The company dataset that we will be using for our analysis will be the Microsoft (MSFT) dataset. The timeframe under consideration includes the past 17 years of data, starting from December 31, 1999 to October 18, 2017. We will be forecasting the future stock prices of MSFT in this exercise.

Let's read the dataset using the `read_csv()` function, as shown here:

```
| In[2]:  
| data = pd.read_csv('MSFT_2000.csv', index_col = 'Date')
```

As we mentioned in the previous section, we will estimate the historical log returns of the MSFT stock. There is a built-in method in pandas called `pct_change()`, which helps us to find the simple returns of the MSFT dataset. We can create a formula for the log returns by using the NumPy log function and add one to the simple returns, as shown here:

```
| In[3]:  
| log_returns = np.log(1 + data.pct_change())
```

We'll take a look at the last five records of the `log_returns`, as shown here:

```
| In[4]
```

```
| log_returns.tail()
```

The output is as follows:

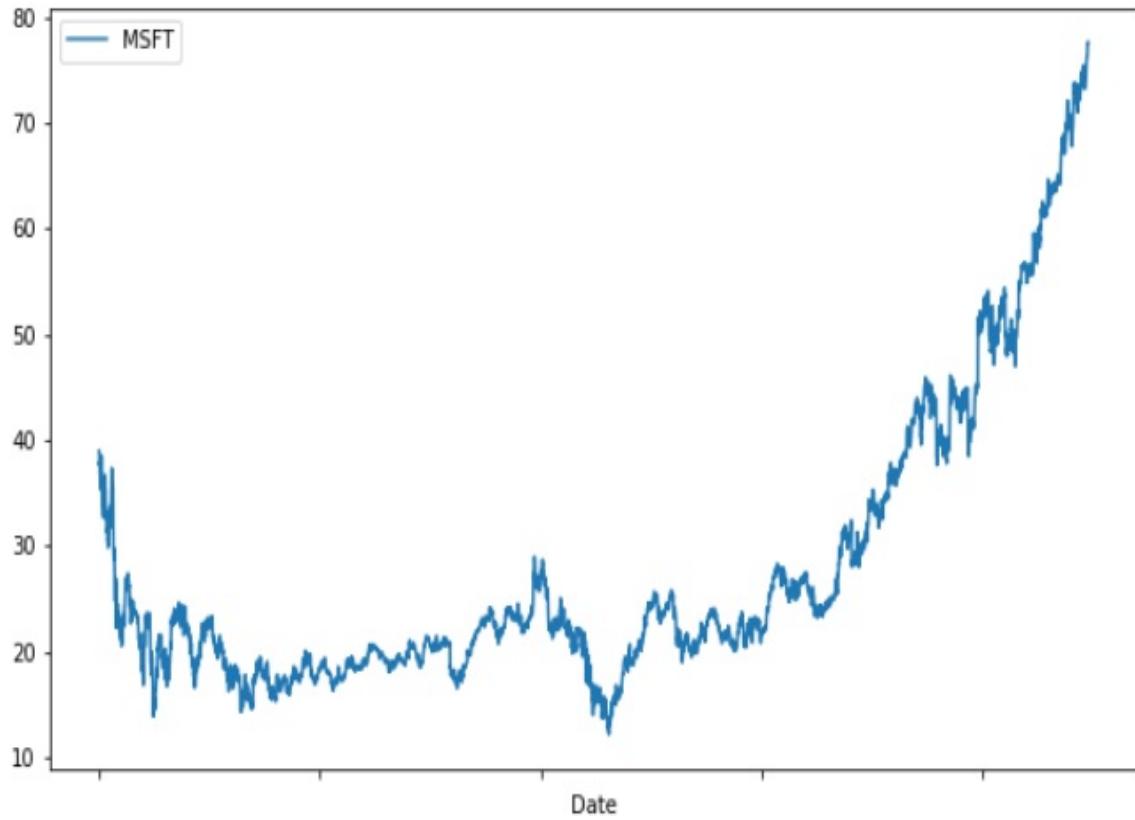
Out[4]:

MSFT	
Date	
2017-10-12	0.009118
2017-10-13	0.004786
2017-10-16	0.002063
2017-10-17	-0.000773
2017-10-18	0.000258

In the next plot, we will plot the MSFT data graph using matplotlib, as shown here:

```
| In[5]:  
| data.plot(figsize=(10, 6));
```

The output looks like there is a gradual increase in the stock price over time:

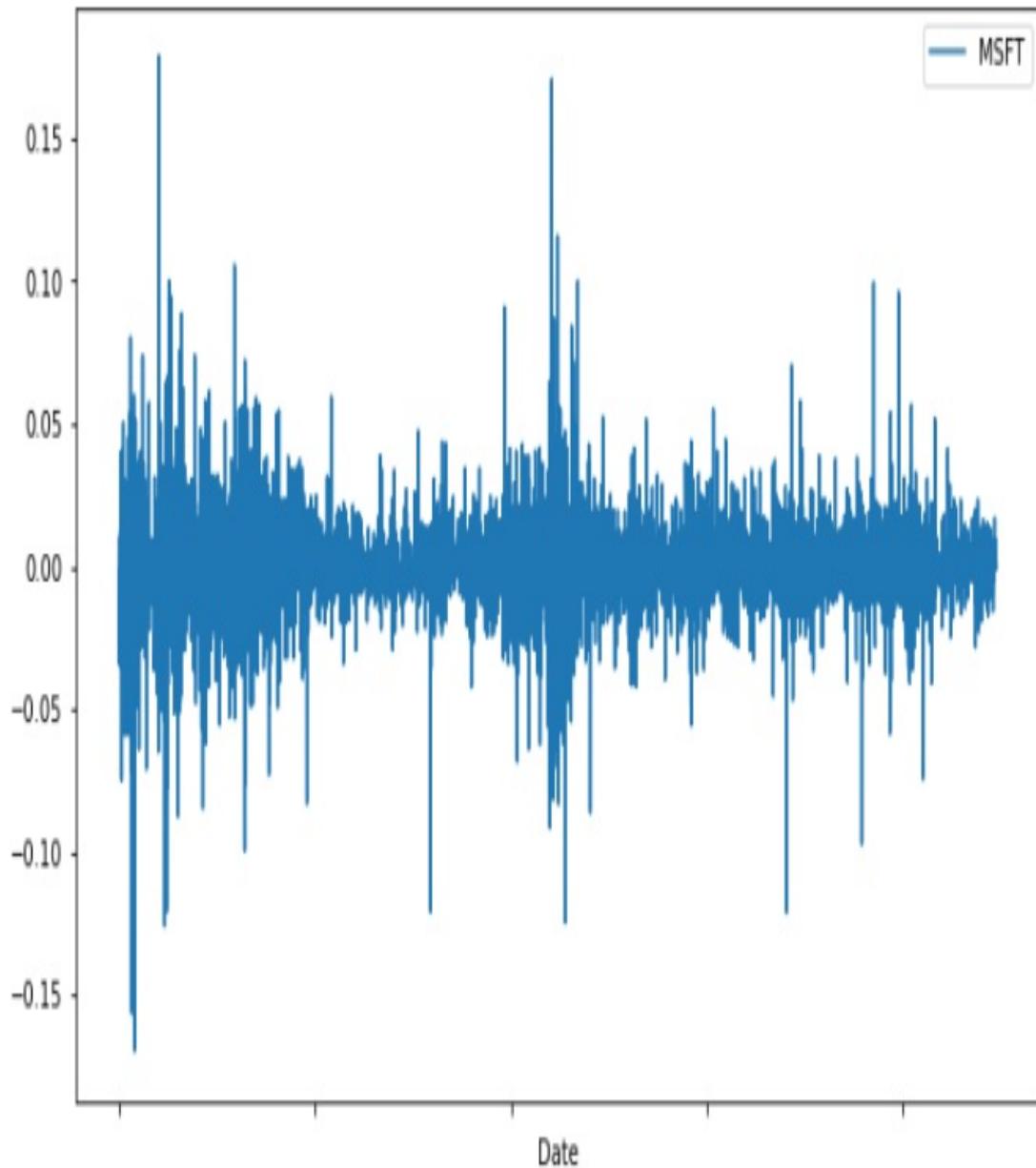


In the next step, we will plot the log returns that were assigned to the `log_returns` variables, as follows:

```
| In[6]:  
| log_returns.plot(figsize = (10, 6))
```

The plotted graph is as shown here:

Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x1d0b07039e8>



The preceding output tells us that the returns are normally distributed and have stable means.

Now, we will explore their mean and variance, as these are essential to calculate the Brownian motion that we discussed earlier in this chapter.

The following is the code to calculate the mean:

```
In[7]:  
u = log_returns.mean()  
u
```

The output is as follows:

```
Out[7]:  
MSFT    0.000154  
dtype: float64
```

The following code is used for computing the variance:

```
In[8]:  
var = log_returns.var()  
var
```

The output is as follows:

```
Out[8]:  
MSFT    0.000376  
dtype: float64
```

We will now be computing the drift component that we discussed earlier. This is the best approximation of the future rates of returns of the stock. The formula to use here is u , which equals the average log return, minus half of its variance, as shown here:

$$drift = u - \frac{1}{2} \cdot var$$

Let's compute the drift with the help of the following code:

```
In[9]:  
drift = u - (0.5 * var)  
drift  
  
Out[9]:  
MSFT    -0.000034  
dtype: float64
```

We have obtained a small drift value, which we don't need to worry about. We will do this exercise without annualizing our indicators because we will try to predict the MSFT daily stock price. Next, we will find the standard deviation of the log returns, as shown here:

```
In[10]:  
stdev = log_returns.std()  
stdev  
  
Out[10]:  
MSFT    0.019397  
dtype: float64
```

We have already discussed that the Brownian motion comprises the sum of the drift and standard deviation adjusted by e to the power of r , as shown here:

$$\text{Brownian motion} : r = \text{drift} + \text{stdev} * e^r$$

In this section, we have looked at calculating the Brownian motion, along with the drift for our simulation, which is our first component. In the upcoming section, we will run the simulation, which will allow us to find the company's future stock prices, which will be our second component in our Monte Carlo simulation.

Using Monte Carlo simulation to forecast stock prices – part 2

In the previous section, we obtained the values of the drift and standard deviation, which we need to calculate the daily returns.

We will be using a `type()` function, which allows us to check the `drift` variable datatype and see it as a pandas series.

The code to check the type of `drift` variable that was computed in the previous section is as follows:

```
| In[10]:  
| type(drift)
```

The output is as follows:

```
| Out[10]:  
| pandas.core.series.Series
```

Similarly, to use the type of the `stdev` variable, we use the following code:

```
| In[11]:  
| type(stdev)
```

The output is as follows:

```
| Out[11]:  
| pandas.core.series.Series
```

The reason why we are trying to see the type is so that we can proceed further with our task. To do this, we should convert these values into NumPy arrays. Here, we know that the `numpy.array()` method can already do this task for us. The code is as follows:

```
| In[12]:  
| np.array(drift)  
|  
| Out[12]:  
| array([-3.42521946e-05])
```

Alternatively, we can also convert these values into arrays using the `values` property, as shown here:

```
| In[13]:  
| drift.values
```

The output is as follows:

```
| Out[13]:  
| array([-3.42521946e-05])
```

We will perform the same step for `std_dev`, as shown here:

```
| In[15]:  
| stdev.values
```

The output is as follows:

```
| Out[14]:  
| array([0.01939682])
```

The second component of the Brownian motion is basically a random variable, indicated by `z`, which is given by the distance between the events and the mean. This is usually expressed as the number of standard deviations. We will be using the `scipy norm.ppf()` function to obtain this result. The code is as follows:

```
| In[16]:  
| norm.ppf(0.95)
```

The output is as follows:

```
| Out[16]:  
| 1.6448536269514722
```

The preceding code specifies that if an event has a 95% chance of occurring, the distance between this event and the mean will be approximately 1.65 standard deviations.

To complete the second component, we will need to initialize some random variables. We will be using the well-known NumPy `rand()` function to create a random array, as shown here:

```
| In[17]:  
| x = np.random.rand(10, 2)  
| x
```

The output is as follows:

```
| Out[17]:  
| array([[0.56555911, 0.30563333],  
|        [0.2558629 , 0.3623618 ],  
|        [0.02093475, 0.78152537],  
|        [0.51250654, 0.20669873],  
|        [0.11159959, 0.92041264],  
|        [0.57354439, 0.90275911],  
|        [0.50732264, 0.58218642],  
|        [0.13884827, 0.73142924],  
|        [0.10923014, 0.48959636],  
|        [0.74980671, 0.54223763]])
```

Corresponding to each of the randomly generated probabilities, we include this random element within the distribution to obtain the distance from the mean, as shown in the following code:

```
| In[18]:  
| norm.ppf(x)
```

The output is as follows:

```
| Out[18]:  
| array([[ 0.45015915,  0.19631405],  
|        [-1.0100133 , -0.35284644],  
|        [-1.40043117,  0.25628267],  
|        [-0.77801174,  0.80260848],  
|        [ 0.53589663, -0.27688877],  
|        [-0.44906061,  1.57629762],  
|        [-0.28410626, -0.48493221],  
|        [ 0.04759264,  0.92631139],  
|        [ 0.45262652,  0.70276892],  
|        [ 0.59550801,  0.04143592]])
```

The first number from the first row of the output corresponds to the first probability from the first row of the x matrix or array. Similarly, the second element corresponds to the second probability, as shown in the x matrix, and so on.

Now, the whole expression that corresponds to z is as follows:

```
| In[19]:  
| z = norm.ppf(np.random.rand(10,2))  
| z
```

The output is as follows:

```
| Out[19]:  
| array([[ 0.45015915,  0.19631405],  
|        [-1.0100133 , -0.35284644],  
|        [-1.40043117,  0.25628267],
```

```
[[-0.77801174,  0.80260848],
 [ 0.53589663, -0.27688877],
 [-0.44906061,  1.57629762],
 [-0.28410626, -0.48493221],
 [ 0.04759264,  0.92631139],
 [ 0.45262652,  0.70276892],
 [ 0.59550801,  0.04143592]])
```

The array, which is created from the preceding code, will create probabilities using the random function, which in turn is converted from the mean 0 and the number of standard deviations from the mean. The probabilities that are created assume values of the standard normal distribution.

Once these tools are built and have calculated all the necessary variables, we will be ready to calculate the daily returns.

Initially, we will create a time interval variable and assign it to 1,000, so that we will be forecasting the stock price for the upcoming 1,000 days. Another variable we will create is `iterations`, and we will set the value to `10`, which means that we will produce a series of 10 future stock predictions. The code to do this is as follows:

```
In[20]:  
t_intervals = 1000  
iterations = 10
```

Now, let's get back to the equation that we discussed at the start of this section. The daily returns and the r value equation are given as follows:

daily_returns= e^r
 $r=drift + stdev * z$

The code is as follows:

```
In[21]:  
daily_returns = np.exp(drift.values + stdev.values * norm.ppf(np.random.rand(t_intervals  
daily returns
```

Here, `z` is `norm.ppf(np.random.rand(t_intervals, iterations))`.

The output is as follows:

```
Out[22]:  
array([[ 1.01591081,  0.9794886 ,  1.00048003,  ...,  1.02416   ,  1.02649606,  
       1.00107475],  
      [ 1.02952747,  0.98716138,  1.00885144,  ...,  1.00744178,  0.99612322,
```

```
0.9957195 ],  
[1.03545845, 1.00564673, 1.00229303, ..., 1.05751804, 0.99713746,  
0.99991494],  
...  
[0.99215031, 0.9734628 , 1.01616243, ..., 1.01639533, 0.9727786 ,  
1.01131246],  
[1.00229691, 0.98945243, 1.00550888, ..., 0.99096372, 0.98795269,  
1.00527073],  
[0.98754594, 0.98971277, 1.02871545, ..., 1.01032874, 0.97908139,  
1.02108459]])
```

We use the preceding equation to get an array of a size of (1,000, 10), where 1,000 is the number of rows and 10 is the number of columns. This means that we have 10 sets of 1,000 random future stock prices.

Using Monte Carlo simulation to forecast stock prices – part 3

In this section, we need to create a price list. Each price must equal the price of the product that was observed the previous day, as shown here:

$$\begin{aligned} S_t &= S_0 \cdot \text{daily_return}_t \\ S_{t+1} &= S_t \cdot \text{daily_return}_{t+1} \\ &\dots \\ S_{t+999} &= S_{t+998} \cdot \text{daily_return}_{t+999} \end{aligned}$$

Once we calculate the price of the stock at the date specified by t , we can predict the price of the stock at the date specified by $t+1$. This process will then continue 1,000 times, giving us predictions of the stocks for 1,000 days in the future.

We have already created a matrix that contains the daily returns (`daily_returns`), so the `daily_returns` variable is available. The question remains, however, whether the first price in the list will be 0 or \$1 million.

To make useful predictions, the first stock price in our list must be considered or initialized as the last one in the dataset. This is called the **current market price**. We will call this variable `s0` and it will contain the stock price of the starting day, specified as t_0 . The last piece of data can be retrieved using the `iloc` operator that's present in the pandas library, as shown here:

```
| In[21]:  
| s0 = data.iloc[-1]  
| s0
```

The output is as follows:

```
| Out[21]:  
| MSFT    77.610001  
| Name: 2017-10-18, dtype: float64
```

The preceding price is the first stock price that we will enter in the list. Then we

will insert the other stock prices. This list should have the same size as that of the `daily_returns` array. The stock price list will be equal to that of the `daily_returns` array size.

We will be using a method which is `zeros_like()` within the NumPy library, which will be used to create an array. The code is as follows:

```
In[22]:  
price_list = np.zeros_like(daily_returns)  
price_list
```

The output is as follows:

```
Out[22]:  
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],  
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],  
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],  
       ...,  
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],  
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],  
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

From the preceding output, we have obtained an array of 1,000 by 10 elements, similar to the dimensions of the daily returns, that we can now fill with zeros. This step is necessary to replace all the zero values with the expected stock prices. First, we must set the first row of our price list to s_0 . Note that s_0 will be the initial value for each of the 10 iterations we intend to generate. The code is as follows:

```
In[23]:  
price_list[0] = s0  
price_list
```

The output is as follows:

```
Out[23]:  
array([[77.610001, 77.610001, 77.610001, ..., 77.610001, 77.610001,  
       77.610001],  
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,  0.        ,  
       0.        ],  
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,  0.        ,  
       0.        ],  
       ...,  
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,  0.        ,  
       0.        ],  
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,  0.        ,  
       0.        ],  
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,  0.        ,  
       0.        ]])
```

We can now create our stock price list, wherein we will run a loop from day 1 to the 1000th day. The expected price of the stock on any day t will be calculated as stock price at $t-1$ times the daily returns observed on day t. The code is as follows:

```
In[24]:  
for t in range(1, t_intervals):  
    price_list[t] = price_list[t - 1] * daily_returns[t]  
price_list
```

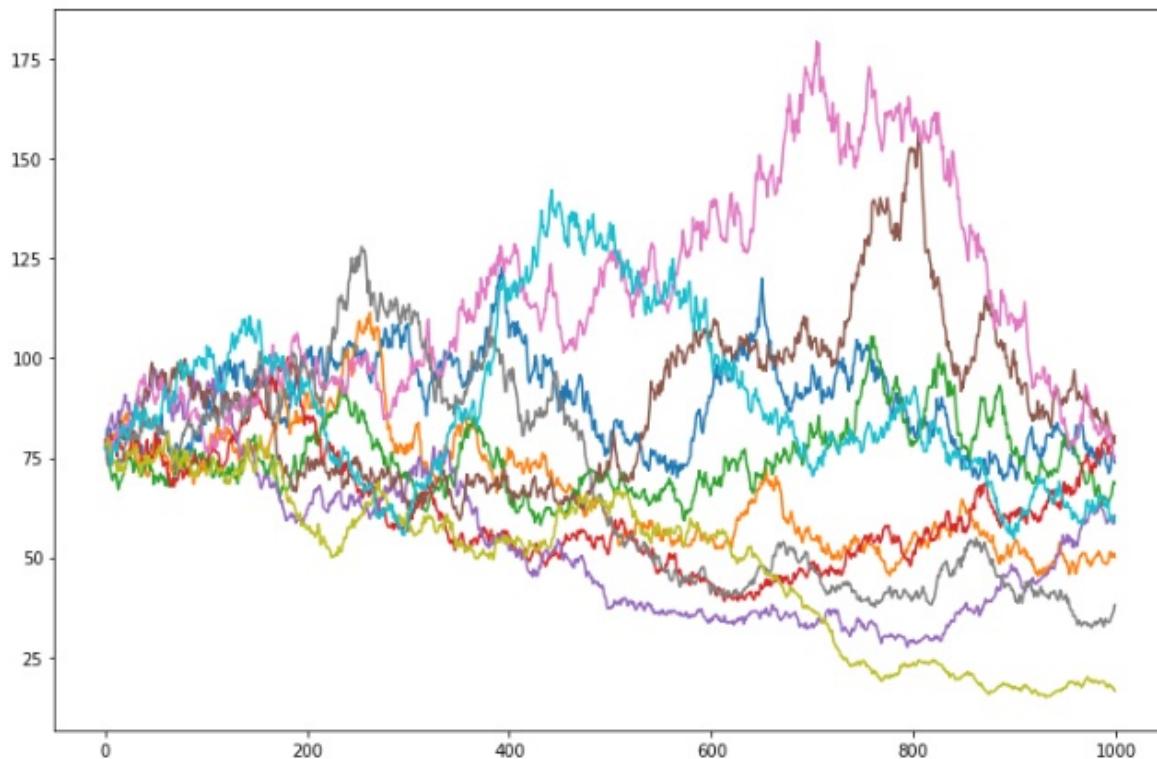
The output is as follows:

```
Out[25]:  
array([[77.610001 , 77.610001 , 77.610001 , ..., 77.610001 ,  
       77.610001 , 77.610001 ],  
      [78.79399319, 78.15736447, 78.47241385, ..., 78.81123575,  
       78.52399318, 77.26364522],  
      [81.97306464, 77.27177019, 76.58029917, ..., 78.41463375,  
       78.38948596, 77.02231333],  
      ...,  
      [75.265363 , 50.36677456, 67.54915702, ..., 35.97898365,  
       17.18563963, 60.4218888 ],  
      [74.6533371 , 51.01256219, 68.96932526, ..., 36.47692199,  
       17.16277676, 59.86800869],  
      [73.82356378, 50.14445155, 68.51996472, ..., 38.06104907,  
       16.57770549, 59.01295086]])
```

Now, let's plot this price list using matplotlib, as shown here:

```
In[26]:  
plt.figure(figsize=(10,6))  
plt.plot(price_list);
```

The output is as follows:



From the preceding output, we will obtain 10 possible paths of the expected stock price of Microsoft (MSFT), starting from the last day present in the dataset. We call these trends iterations, since we iterated through the provided formula 10 times. Here, we have the paths that we simulated.

In this chapter, we got into a lot of technical language, which involved a lot of advanced concepts. This is the kind of topic, however, that you need to master to get into the field of finance or data science. I would personally suggest that you go through Jupyter carefully so that you can get an upper hand when you are carrying out this kind of work.

Summary

In this chapter, we have seen and understood the importance of Monte Carlo simulation and how it is helpful in predicting the gross profit of a company based on COGS, both intuitively and practically, with Python. We also understood how to forecast stock prices using Monte Carlo simulation. We used the Microsoft stock dataset and implemented the forecasting technique using Python and Monte Carlo simulation.

In the next chapter, we will look at the option pricing Black Scholes model to price various derivatives.

Option Pricing - the Black Scholes Model

The Black Scholes formula is one of the most popular financial instruments in use. It was derived by Fisher Black, Myron Scholes, and Robert Merton in 1973, and since then it has become the primary tool for derivative pricing.

The original framework that was developed by the three scientists considered the pricing of a European call or put option and assumed that there were efficient markets, an absence of transaction costs, no dividend payments, and a known volatility and risk-free rate. Some of these hypotheses can be used to calculate an option price in practice. In this chapter, we are going to understand derivatives and look at the Black Scholes formula for option pricing and Euler Discretization.

In this chapter, we will be focusing on the following topics:

- An introduction to derivative contracts
- Using the Black Scholes formula for option pricing with Python
- Using Monte Carlo in conjunction with Black Scholes
- Using Monte Carlo in conjunction with Euler Discretization in Python

Technical requirements

In this chapter, we will be using the Jupyter Notebook for coding purposes. We will also be using the pandas, NumPy, matplotlib, and SciPy libraries.

The GitHub repository for this chapter can be found at the following link: <https://github.com/PacktPublishing/Hands-on-Python-for-Finance/tree/master/Chapter%209>.

An introduction to the derivative contracts

A derivative is a financial instrument whose price is determined, or derived, based on the development of one or more underlying assets, such as stocks, bonds, interest rate commodities, and exchange rates. It is a contract involving at least two parties and describes how and when the two parties will exchange payments. Some derivative contracts are traded in regulated markets, while others that are traded over the counter are not regulated.

Derivatives traded in regulated markets have a uniform contractual structure and are much simpler to understand. Originally, derivatives were used as a hedging instrument. Companies interested in buying these contracts were mostly concerned about protecting their investments.

However, with time, a great deal of innovation was introduced to the scene. So-called financial engineering was applied and new types of derivatives appeared. Nowadays, there are many types of derivatives. Some are rather complicated and difficult to understand, even for those with a lot of experience in finance.

There are three groups of people who tend to be interested in dealing with derivatives:

- People interested in hedging their investments
- Speculators
- Arbitrageurs

In this chapter, we will learn about the four main types of financial derivatives and how they function:

- Forward
- Future
- Option
- Swap

Forward contracts

A forward contract is a tweaked contract between two gatherings, where settlement happens on a particular date in the future at a cost incurred today. Forward contracts are not traded in the standard stock exchange and, as a result, they are not standardized, making them particularly useful for hedging. The primary characteristics of forward contracts are as follows:

- They are reciprocal contracts and are consequently presented to the opposite party
- Each agreement is hand-crafted and is therefore unique in regard to the measure of the contract, the lapse date, the asset type, and the asset quality
- The agreement must be settled by conveyance of the benefit on the lapse date

Future contracts

A future contact is a contract that is used to buy or sell underlying assets, such as stocks, bonds, or commodities at a specified time. A future contract is a lawful consent to purchase or offer a specific commodity or asset at a cost at a predetermined time later on. Future contracts are similar to forward contracts, but they are standardized and regulated so that they may be traded in the future. They are often used to speculate on commodities.

The purchaser of a future contract assumes the commitment of purchasing the underlying assets when the future contract terminates. The seller of the future contract assumes the commitment of providing the underlying asset on the expiry date. Every future contract has the following features:

- A purchaser
- A vendor
- A cost
- An expiry date

Option contracts

An option contract is a contract that gives someone the right, but not the obligation, to buy (call) or sell (put) security or an other financial asset. A call option gives the purchaser the privilege of purchasing the asset at a given cost, called the **strike price**. While the holder of the call option has the privilege of requesting an offer from the seller, the vendor or the seller has the right to sell but not necessarily the commitment to do so. If a purchaser wants to purchase the underlying asset, the merchant needs to offer it, but they don't have the obligation to do so.

Similarly, a put option gives the purchaser the privilege of selling the asset at the strike price to the purchaser. Here, the purchaser has the privilege to offer, and the seller has the commitment to purchase. In every option contract, the privilege to exercise the option is vested with the purchaser of the agreement. The seller of the agreement has the right to sell, but not the commitment to do so. As the seller of the agreement bears the commitment, they are paid a cost, called a premium.

Swap contracts

A swap contract is used for the exchange of one cash flow for another set of future cash flows. A swap refers to the exchange of one security for another, based on different factors.

The main advantages of swap contracts are listed here:

- **Converting financial exposure:** Swap contracts can be used to convert currencies to obtain debt financing at a reduced interest rate. This is brought about by the comparative advantages that each counter-party has in their national capital market, and/or the benefits of hedging long-run exchange rate exposure.
- **Comparative advantages:** The different types of debt instruments show that there exists an arbitrage opportunity due to mispricing of the default risk premiums.
- **Speculations:** Swap contracts allow us to speculate on interest rates, currencies, and so on.

Using the Black Scholes formula for option pricing

The Black Scholes formula calculates the value of an option. An option is the freedom to choose whether to acquire a stock. The holder of the option may decide if they want to buy the stock, but they may also decide that they are better off without doing so. This freedom is valuable to every investor, so it has a price.

Here is a diagram representing the payoff of a call option:



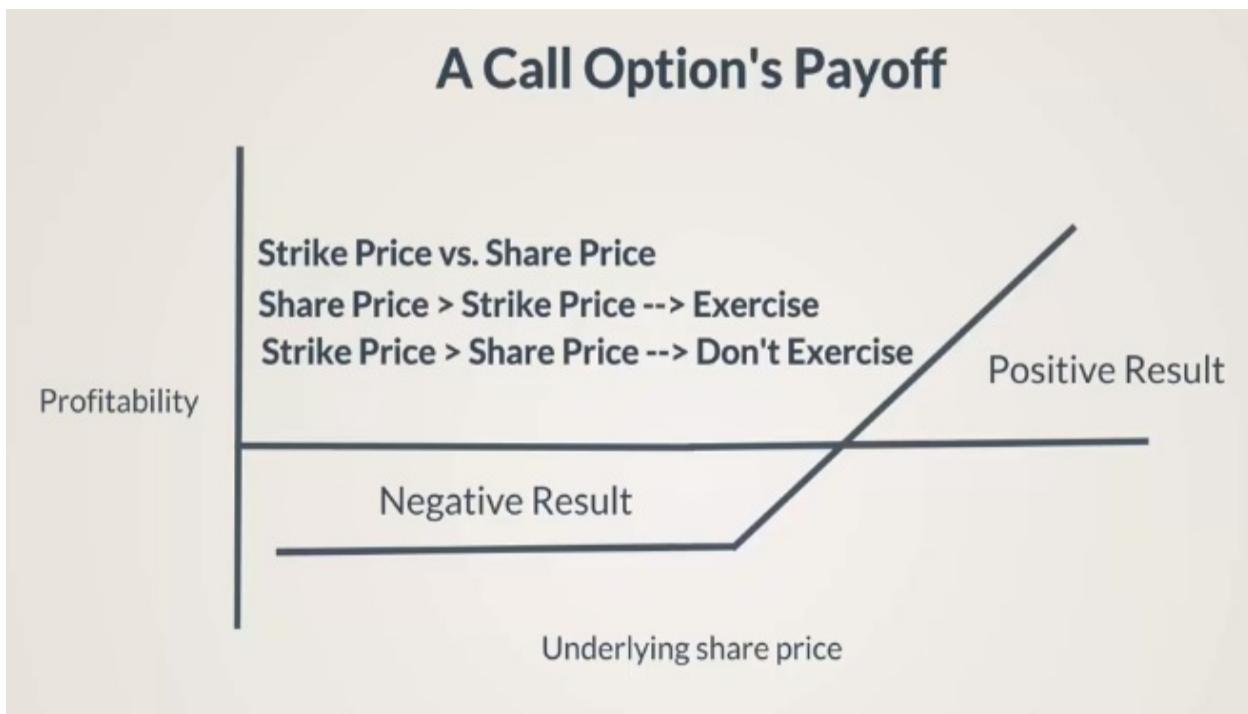
The y axis shows the profitability of an investor who buys the call option, while the x axis shows the development of the underlying share price for which the investor has an option. When the option expires, the owner will compare the strike price and the actual market price of the underlying share.

The strike price is the price at which the derivative can be bought or exercised. This term is most commonly used to describe the index and the stock options. For call options, the strike price is the price at which the stocks or the derivatives can be bought by the buyer until the expiration date. For put options, the strike

price is the price at which shares can be sold by the option buyer.

If the strike price is lower than the market price, the owner of the option will exercise it. Conversely, if the strike price is higher than the market price, they won't exercise the option, because they must buy the share at a higher price than its market price.

The profitability of the investor therefore looks as follows:



At first, the investor buys the option. They pay the money and so their profitability is negative. Then, when the expiration date comes around, they are able to use this option if the price of the underlying share is higher than the strike price. Even if the market price is higher than the strike price, this doesn't mean the investor will profit from the deal. They need a price that is significantly higher to reach a break-even point and profit from the deal.

The Black Scholes formula provides an intuitive way to calculate the option prices. The formula is as given here:

$$C(S, t) = N(d_1)S - N(d_2)Ke^{-r(T-t)}$$

$$d_1 = \frac{1}{s\sqrt{T-t}} \left[\ln\left(\frac{S}{K}\right) + \left(r + \frac{S^2}{2}\right)(T-t) \right]$$

$$d_2 = d_1 - s\sqrt{T-t}$$

Here, we have the following:

- S = the current stock price
- K = the option strike price
- T = the time when the option was exercised
- t = the time until the option expires
- r = the risk-free interest rate
- s = the sample standard deviation
- N = the standard normal distribution

- e = the exponential term
- C = the call premium

The first component, d_1 , shows us how much we are going to get if the option is exercised. The second component, d_2 , is the amount we must pay when exercising the option. If we have all the inputs necessary to calculate d_1 and d_2 , we shouldn't have a problem in obtaining these values and applying them in the preceding formulas.

This is the key to understanding the formulas. In the first component, to find d_1 , we are multiplying the probability of exercising the option by the amount received when the option is exercised. In the second component, to find d_2 , we subtract the value of the amount we must pay to exercise the option. All this is done if the development of the stock's share price follows a normal distribution.

In short, the Black Scholes formula calculates the value of a call by taking the difference between the amount you get if you exercise the option, minus the amount you have to pay if you exercise the option.

Calculating the price of an option using Black Scholes

Our goal in this section will be to calculate the price of a call option using Python. We will apply the Black Scholes formula that we introduced in the previous section.

Let's begin by importing the necessary libraries, as shown in the following code:

```
In[1]:  
import numpy as np  
import pandas as pd  
from pandas_datareader import data as wb  
from scipy.stats import norm
```

Now, we will create two functions that will allow us to calculate d_1 and d_2 , which we need to apply the Black Scholes formula and price a call option. The parameters to include are the current stock (S), the strike price (K), the risk-free interest rate (r), the standard deviation ($stdev$), and the time (t) measured in years.

The following are the equations of the two functions, d_1 and d_2 :

$$d_1 = \frac{\ln(\frac{S}{K}) + (r + \frac{stdev^2}{2})t}{s \cdot \sqrt{t}}$$
$$d_2 = d_1 - s \cdot \sqrt{t} = \frac{\ln(\frac{S}{K}) + (r - \frac{stdev^2}{2})t}{s \cdot \sqrt{t}}$$

The code is as follows:

```
In[2]:  
def d1(S, K, r, stdev, T):  
    return (np.log(S / K) + (r + stdev ** 2 / 2) * T) / (stdev * np.sqrt(T))  
  
def d2(S, K, r, stdev, T):  
    return (np.log(S / K) + (r - stdev ** 2 / 2) * T) / (stdev * np.sqrt(T))
```

The difference between d_1 and d_2 is that in d_1 , we add the variance divided by 2, denoted as $(r + stdev **2/2)$, while in d_2 , we must subtract it, denoted as $(r - stdev **2/2)$.

To apply the Black Scholes formula, we won't need the PPF distribution we used previously when we forecasted the future stock prices. Instead, we will need the cumulative normal distribution. The reason this is needed is that it shows us how the data accumulates over time. Its output can never be below zero or above one.

We will use the SciPy library and the `cdf()` function, which will take a value from the data as an argument and show us what portion of the data lies below that value.

For instance, an argument of `0` will lead to an output of `0.5`, as shown in the following code:

```
| In[3]:  
| norm.cdf(0)
```

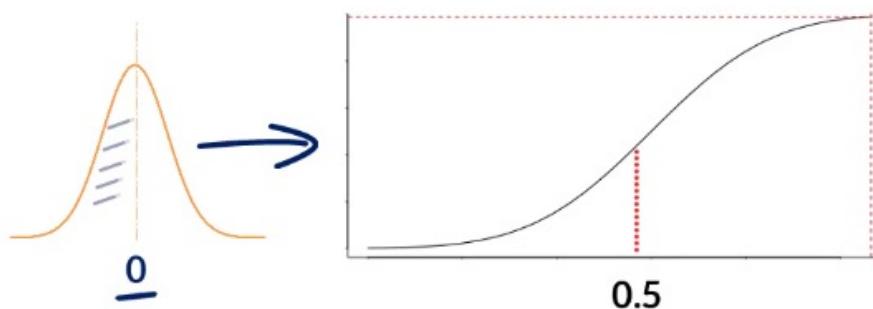
The output is as follows:

```
| Out[3]:  
| 0.5
```

Since zero is the mean of the standard normal distribution, half the data lies below this value:

Cumulative Distribution Function (cdf)

shows how the data accumulates in time



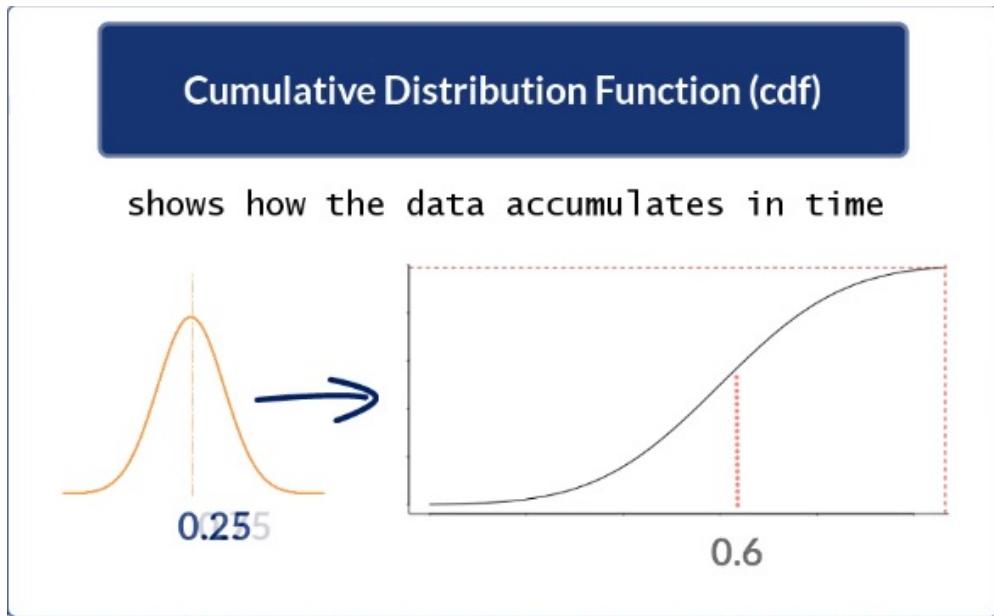
Now, let's give an argument of 0.25 :

```
| In[4]:  
| norm.cdf(0.25)
```

The output is as follows:

```
| Out[4]:  
| 0.5987063256829237
```

This can be represented as follows:



If we give a big argument, such as 9 , we expect to find the largest data point in our set:

```
| In[6]:  
| norm.cdf(9)
```

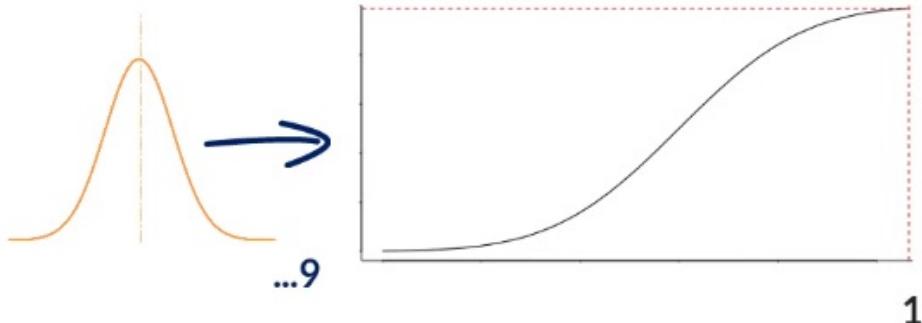
The output is as follows:

```
| Out[6]:  
| 1.0
```

The following diagram gives us a better representation of this:

Cumulative Distribution Function (cdf)

shows how the data accumulates in time



We can now introduce the Black Scholes function. It will have the same parameters as d_1 and d_2 :

$$C = SN(d_1) - Ke^{-rt} N(d_2)$$

The code is as follows:

```
In[7]:  
def BSM(S, K, r, stdev, T):  
    return (S * norm.cdf(d1(S, K, r, stdev, T))) - (K * np.exp(-r * T) * norm.cdf(d2
```

We have now defined all the necessary functions. Let's apply them to the Procter and Gamble dataset, which consists of the stock prices of the Procter and Gamble company. We will first read the dataset using the pandas `read_csv` function, as shown here:

```
In[8]:  
data = pd.read_csv('PG_2007_2017.csv', index_col = 'Date')
```

We will then use the `iloc` operator with the argument as `-1` to get the last record. We will store the last record in the variable `s`, as shown here:

```
In[9]:  
S = data.iloc[-1]  
S  
  
Out[9]:  
PG      88.118629  
Name: 2017-04-10, dtype: float64
```

Another argument we can extract from the data is the standard deviation. In our case, we will use an approximation of the standard deviation of the logarithmic returns of this stock, as shown here:

```
| In[10]:  
| log_returns = np.log(1 + data.pct_change())
```

Then, we will find the standard deviation, as shown here:

```
| In[11]:  
| stdev = log_returns.std() * 250 ** 0.5  
| stdev
```

The output is as follows:

```
| Out[11]:  
| PG      0.176109  
| dtype: float64
```

We can now calculate the price of the call option. We will stick to a risk-free interest rate (r) of 2.5%, corresponding to the yield of a 10 year government bond. Let's assume that the strike price (K) equals \$110 and that the time (T) is 1 year. Let's define these variables, as follows:

```
| In[12]:  
| r = 0.025  
| K = 110.0  
| T = 1
```

At this stage, we have the values for all five parameters we are interested in. If we use them in the three functions we created, we will be able to determine the value for d_1 , d_2 , and the `bsm` function, which are the components of the Black Scholes formula, as shown here:

```
| In[13]:  
| d1(S, K, r, stdev, T)
```

The output is as follows:

```
| Out[13]:  
| PG      -1.029416  
| dtype: float64
```

The same can be applied for d_2 , as shown here:

```
| In[14]:  
| d2(S, K, r, stdev, T)
```

The output is as follows:

```
In[14]:  
PG    -1.205525  
dtype: float64
```

The same can be applied for `BSM`, as shown here:

```
In[15]:  
BSM(S, K, r, stdev, T)  
  
Out[15]:  
PG    1.132067  
Name: 2017-04-10, dtype: float64
```

The call price option from the preceding output is close to \$1 and 13 cents. We were able to successfully price the call option. Is it possible to have a call option price that is much lower than the actual stock price? The value of the option depends on multiple parameters, such as the strike price, the time when we are exercising the option, the maturity of the option, and the market volatility. It is not directly proportional to the price of the security.

If you rerun the same code with different values for the time to maturity period, the standard deviation, or the strike price, you will obtain a different option price. In the next section, we will see how we can calculate the price of a stock option in a more sophisticated way.

Using Monte Carlo in conjunction with Euler Discretization in Python

In this section, we will be looking at some more advanced features of finance and mathematics. We will use the Euler Discretization technique to calculate the call option. This is a more sophisticated way of calculating the call option than the techniques we discussed in the previous section. To compare the two approaches, we will use the same dataset we used in the previous section, which is the Procter and Gamble dataset for the time period from January 1, 2007 until March 21, 2017.

To begin with, let's import all the necessary libraries, as shown here:

```
| In[1]:  
| import numpy as np  
| import pandas as pd  
| from pandas_datareader import data as web  
| from scipy.stats import norm  
| import matplotlib.pyplot as plt  
| %matplotlib inline
```

The next step is to read the dataset using the pandas `read_csv()` method, as shown here:

```
| In[2]:  
| data = pd.read_csv('PG_2007_2017.csv', index_col = 'Date')
```

In the next step, we calculate the log returns, as shown here:

```
| In[3]:  
| log_returns = np.log(1 + data.pct_change())
```

In Euler Discretization, the methods and formula we will apply to compute the call option price are different. We want to run a huge number of experiments to make sure that the price we pick is the most accurate one.

Monte Carlo simulation can provide us with thousands of possible call option prices. We can then average the pay off. The trick lies in the formula we will use to calculate the future prices, which is as follows:

$$S_t = S_{t-1} \cdot e^{((r - \frac{1}{2} \cdot stdev^2) \cdot \delta_t + stdev \cdot \sqrt{\delta_t} \cdot Z_t)}$$

This is another version of a Brownian motion. The formula and the approach employed are what is known as Euler Discretization. The following are the parameters of the preceding equation:

- S_t = the stock price for day t
- S_{t-1} = the stock price observed on the previous day, $t-1$

Let's go through the steps that will allow us to assign values in this long formula.

The first thing that we will do is assign a risk-free interest rate (r). The initialization is shown here:

```
| r = 0.025
```

Here, we assign the value of the risk-free interest rate as 2.5%. Next, we can obtain the standard deviation of the log returns and store it in a `stdev` variable, as shown here:

```
| In[5]:
| stdev = log_returns.std() * 250 ** 0.5
| stdev
```

The output is as follows:

```
| Out[5]:
| PG    0.176109
| dtype: float64
```

We can verify that the `stdev` variable is a series using the following code:

```
| In[6]:
| type(stdev)
```

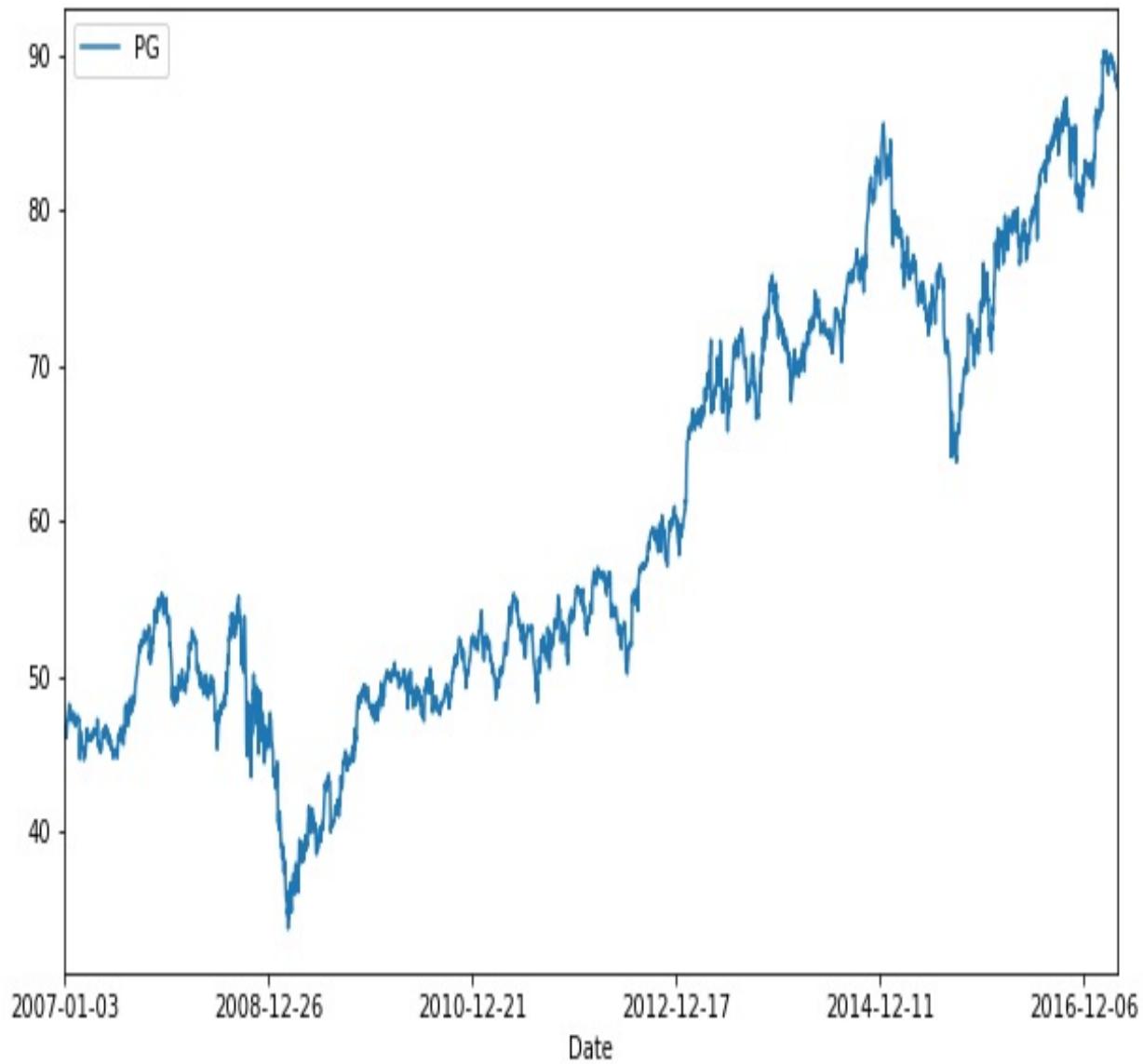
The output is as follows:

```
| Out[6]:
| pandas.core.series.Series
```

Let's plot and check the data of the Proctor and Gamble dataset, as shown here:

```
| In[6]:
| data.plot(figsize=(10, 6));
```

The output is as follows:



We intend to use the `stdev` variable in an array operation afterwards, so it is necessary to convert the `stdev` variable, which is a series, into an array, by using values, as shown here:

```
| In[7]:  
| stdev = stdev.values  
| stdev
```

The output is as follows:

```
| Out[7]:
```

```
| array([ 0.17610875])
```

The preceding output basically tells us that `stdev` is now a NumPy array.

We will consider the time (T) to be 1 year, since we are forecasting the prices for 1 year ahead. The number of time intervals (`t_intervals`) must correspond to the number of trading days in a year, which is 250. The initialization is shown in the following code:

```
In[8]:  
T = 1.0  
t_intervals = 250  
delta_t = T / t_intervals  
  
iterations = 10000
```

From the preceding code, we have created two more variables: `delta_t`, which is the fixed time interval, and the 10,000 simulations or iterations for simulating `z`, which is the random component.

The code is shown here:

```
In[9]:  
Z = np.random.standard_normal((t_intervals + 1, iterations))
```

The random component `z` will be a matrix with random components drawn from a standard normal distribution, which is a normal distribution with a mean of 0 and a standard deviation of 1.

The dimension of the matrix will be defined by the number of time intervals augmented by 1 (`t_intervals + 1`) and the number of iterations (`iterations`).

The code is as follows:

```
In[10]:  
S = np.zeros_like(Z)
```

In this step, we can create an empty array, `s`, of the same dimensions as the random component `z`. For this, we use the `np.zeros_like()` method, as shown in the preceding code.

Then, we will use the `iloc` operator with the argument as `-1` to get the last record. We will store the last record in the variable `s0`, as shown in the preceding code:

```
| In[11]:  
| S0 = data.iloc[-1]  
| S[0] = S0
```

As well as changing the formula, which differs from the one we used for calculating the future stock prices, the remaining steps are almost identical to the Monte Carlo technique we looked at in the previous chapter.

We loop from 1 to the number of intervals ($t_intervals + 1$) and we create a whole stock price matrix with the dimensions of ($t_intervals + 1$) and the number of iterations, as shown here:

```
| In[12]:  
| for t in range(1, t_intervals + 1):  
|     S[t] = S[t-1] * np.exp((r - 0.5 * stdev ** 2) * delta_t + stdev * delta_t ** 0.5 * z
```

We can see the value inside s , as follows:

```
| In[13]:  
| S
```

The output is as follows:

```
| Out[13]:  
| array([[ 88.118629 ,  88.118629 ,  88.118629 ,  ...,  88.118629 ,  
|        88.118629 ,  88.118629 ], [ 88.81636639,  90.61639177,  89.03898033,  ...,  87.30375254,  
|        87.5583302 ,  88.80166136], [ 86.90009657,  91.62555653,  89.63268689,  ...,  88.33679858,  
|        87.62299858,  91.35123942],  
|        ...,[ 84.3786526 ,  107.72224183,  80.82567392,  ...,  125.62838195,  
|        89.32732597,  105.18086595], [ 84.3872724 ,  108.28857142,  81.12333622,  ...,  125.84132277,  
|        90.90533155,  105.51622291], [ 83.67055401,  107.96691607,  81.7082641 ,  ...,  125.94172348,  
|        91.80990436,  104.04308705]])
```

We can check the dimension by using the `shape` property, as shown here:

```
| In[14]:  
| S.shape
```

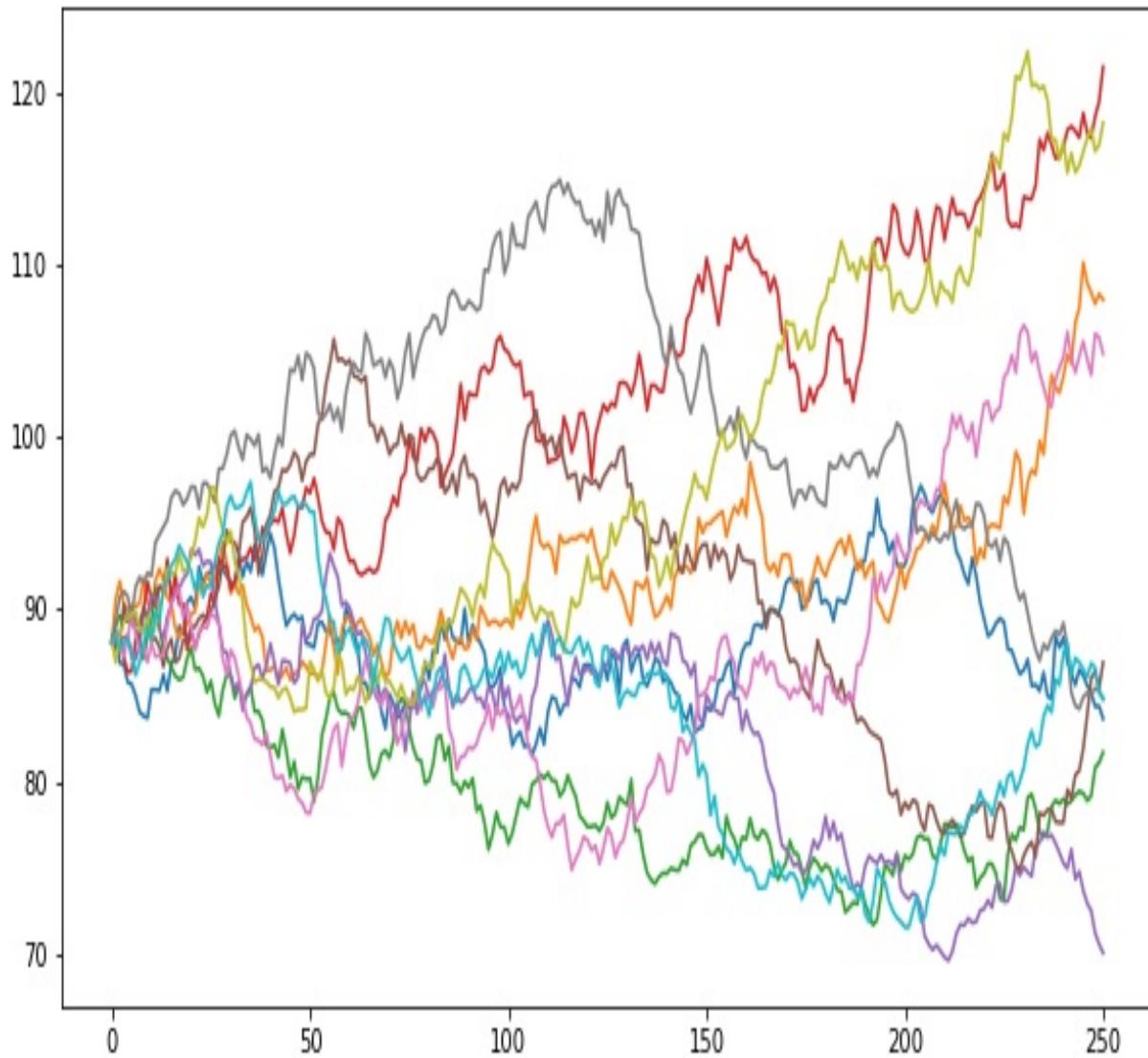
The output is as follows:

```
| Out[14]:  
| (251L, 10000L)
```

To plot only 10 simulations, we can use `matplotlib`. The code is shown here:

```
In[15]:  
plt.figure(figsize=(10, 6))  
plt.plot(S[:, :10]);
```

The output is as follows:



Our simulation is complete and now we will go ahead and compute the call option price.

Let's see what the payoff is like for a call option. At a certain point in time, we will exercise our right to buy the option if the stock price minus the strike price is greater than zero. We will not exercise our right to buy if the difference is a negative number.

So, the value of the option depends on the chance of the difference between the stock price (S) and the strike price (K) being positive and, in particular, how positive it is expected to be.

To work this out, we can use a NumPy method called `maximum()`, which will create an array that contains either zeros or the numbers equal to the differences. We call p , which represents the payoff. The code is as follows:

```
| In[16]:  
| p = np.maximum(S[-1] - 110, 0)  
| p
```

The output is as follows:

```
| Out[16]:  
| array([ 0.,  0.,  0., ...,  0.,  0.,  0.])
```

We can also check the shape of p , as shown here:

```
| In[17]:  
| p.shape
```

The dimensions are as follows:

```
| Out[17]:  
| (10000L,)
```

The formula to discount the average of the payoff p is shown here:

$$C = \frac{e^{(-rT)} \cdot \sum p_i}{iterations}$$

This corresponds to the exponential of the r and T multiplied by the sum of the computed payoffs, divided by the number of iterations we choose, which is 10,000.

The preceding equation can be written in Python, as shown here:

```
| In[18]:  
| C = np.exp(-r * T) * np.sum(p) / iterations  
| C
```

The output is the price of the call option:

```
| Out[18]:
```

| 1.159769289926591

It is important to compare whether this number differs substantially from the one we obtained with the Black Scholes formula in the previous section. From the Black Scholes formula, we got a call option value of 1.13. This is very close to the value of 1.15, which we got by using Euler Discretization. This is proof that the choice of computation method can lead to insignificant but not unimportant differences in the outcome.

We are now armed with enough pythonic skills to conduct these kinds of studies on our own.

Summary

In this chapter, we have looked at various concepts, such as derivative contracts, the Black Scholes formula, and Euler Discretization. We used these techniques to calculate the call option price, which in turn can be used in derivative contracts. We also implemented this practically by using Python. We then looked at the difference between the call option price that's calculated using the different techniques.

In the next chapter, we are going to introduce deep learning techniques, which can be used to solve problems such as stock prediction, sales prediction, and more. We will look at different frameworks, such as TensorFlow and Keras, which help us to build a basic architecture called a neural network. We will be looking at a famous neural network architecture called a recurrent neural network, which is a very important technique that works well on sequences of data.

Introduction to Deep Learning with TensorFlow and Keras

Deep learning is an emerging and promising technology for data modelling. It helps us to get meaningful insights from data, which can, in turn, help in making predictions. The financial domain is the ideal area for the application of deep learning and machine learning. In this chapter, we are going to see how deep learning can be used in finance. We will also look at neural networks and how they work. We will also be solving a regression problem using libraries such as TensorFlow.

In this chapter, we will be focusing on the following topics:

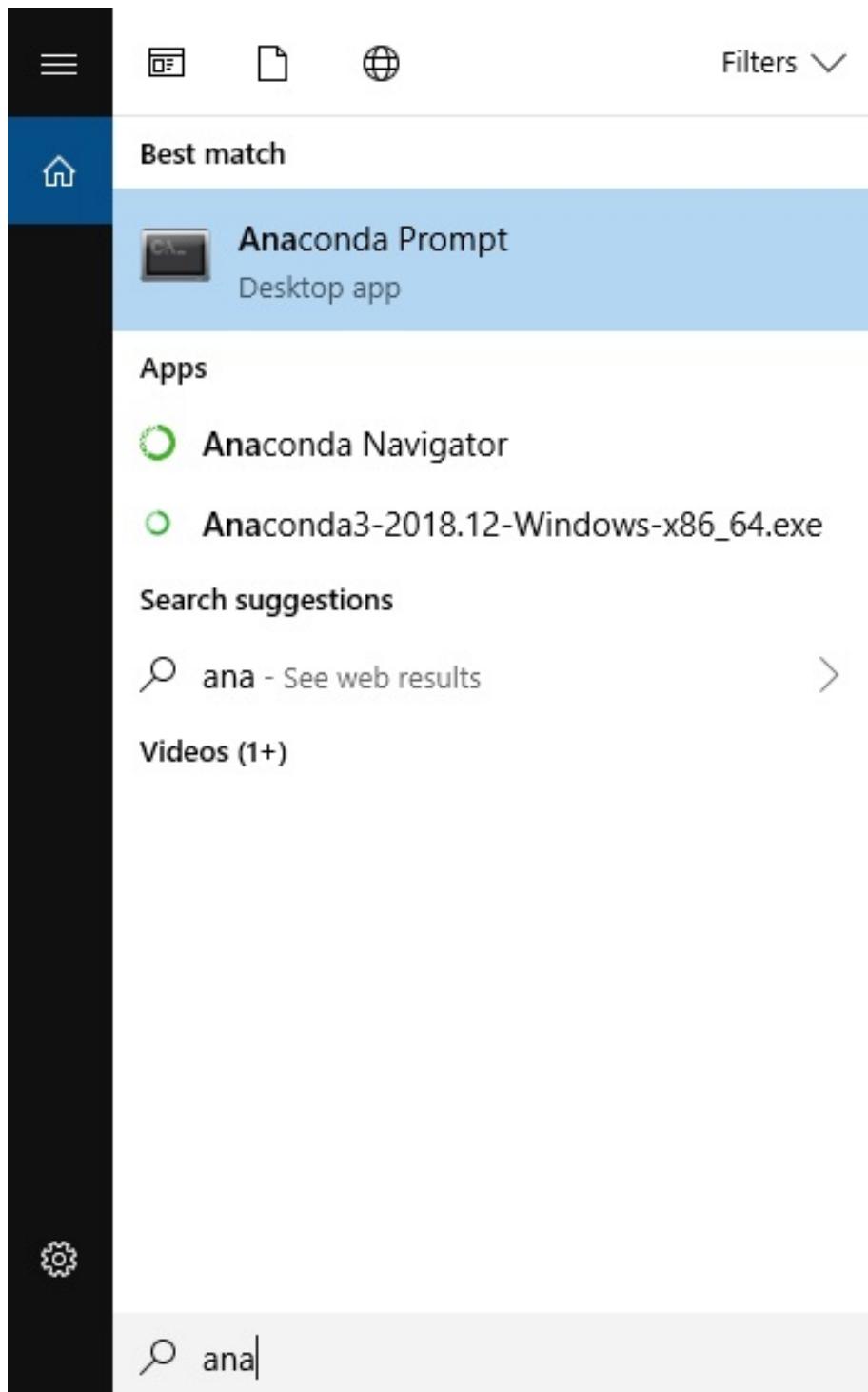
- An overview of deep learning in the financial domain
- An introduction to neural networks
- An introduction to TensorFlow
- An implementation of linear regression using TensorFlow
- An introduction to Keras

Technical requirements

In this chapter, we will be using the Jupyter Notebook for coding. We will be using the pandas, NumPy, and matplotlib libraries, as well as the TensorFlow and Keras libraries.

Here are the installation instructions for TensorFlow and Keras:

1. Go to the Anaconda Command Prompt from the start menu, as shown here:



2. In the Anaconda prompt, type the following command and press *Enter*:

```
| conda install tensorflow
```

After TensorFlow is successfully installed, we need to install the Keras

library.

3. Type the following command in the Anaconda prompt and press *Enter*:

```
| conda install keras
```

The GitHub repository for this chapter can be found at <https://github.com/PacktPublishing/Hands-on-Python-for-Finance/tree/master/Chapter%2010>.

An overview of deep learning in the financial domain

As we know, the human brain is one of the most powerful learning tools on the planet. Recent research in computer science has been looking at whether we can recreate the technique by which the human brain learns. Scientists have managed to implement this mechanism in machines. This process is known as deep learning.

The purpose of deep learning is to mimic the human brain. It uses a neural network architecture that is based on the idea of neurons being the basic units of the human brain. Let's consider an example: as humans, it is very easy for us to determine whether an animal is a dog or a cat. When we see an animal, our sensory organs (our eyes, in this case) receive the signal. This signal is then passed through a large number of neurons, a structure known as the neural network. As this happens, the signal is processed and the resulting information is sent to the brain. In this section, our aim is to create a machine or model that is able to replicate the working of a neural network with respect to deep learning using the same methodology that our brain uses to solve problems concerned with financial data.

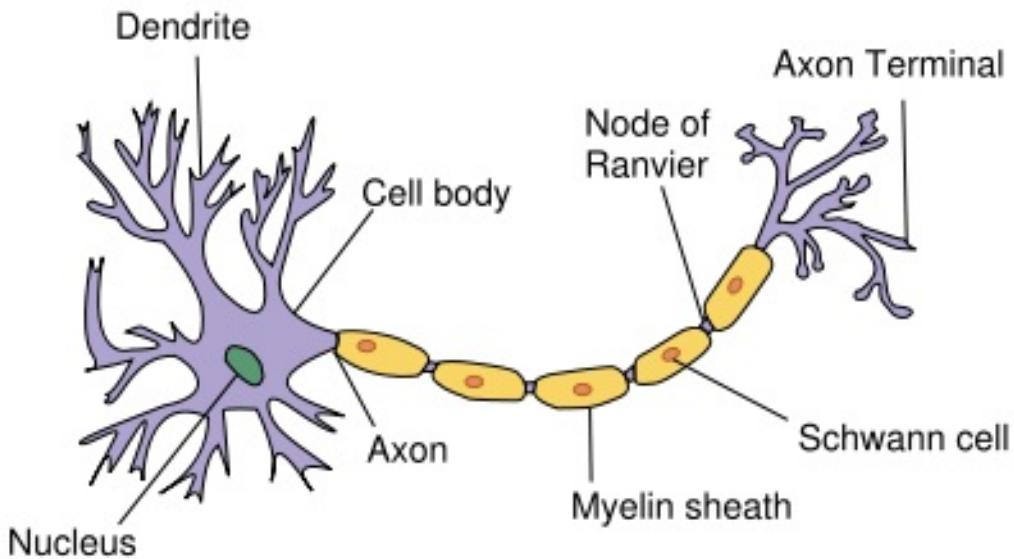
An introduction to neural networks

In this section, we will discuss neural networks. A neural network in computing is a complex model that is inspired by the way in which neural networks that are present in the human brain digest and process information. Neural networks have had a significant impact on research and development in areas such as speech recognition, natural language processing, predictive modelling, and computer vision.

Neurons

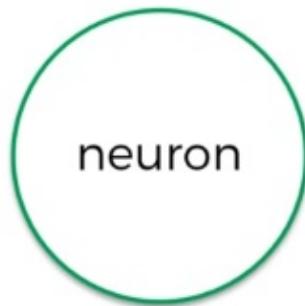
In this section, we are going to discuss neurons, which are the basic building blocks of the neural network. They are also called **units** or **nodes**.

Let's take a look at the basic structure of a biological neuron:

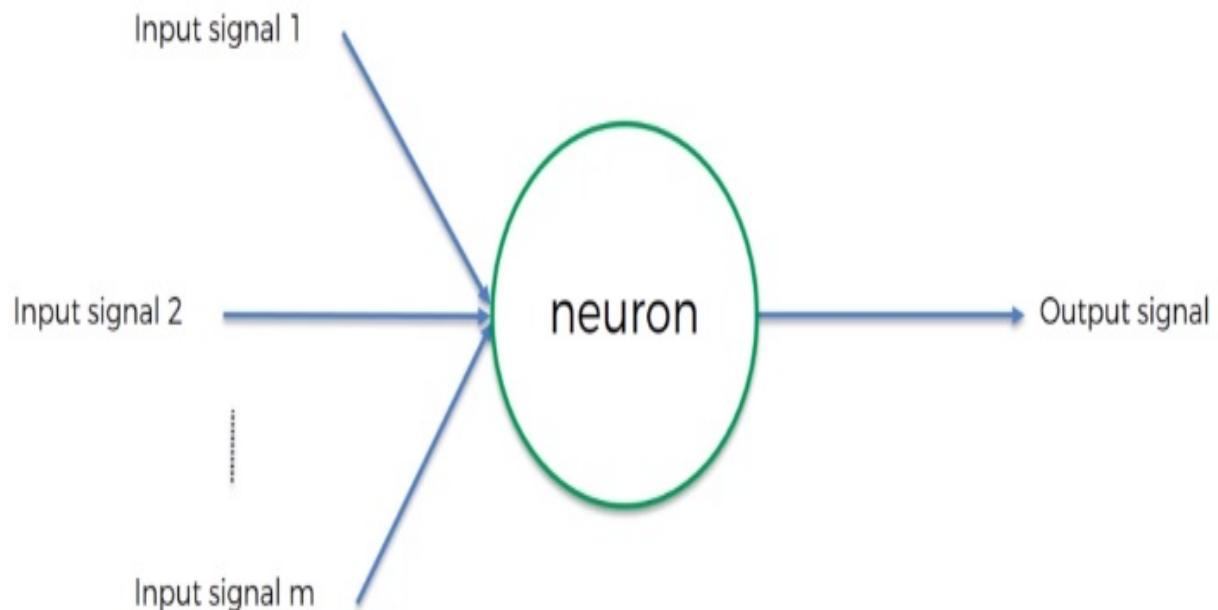


The main component of a neuron is a **Nucleus**. It has some branches at the top that are called **Dendrites**. The body of the neuron is called an **Axon**, and we also have another structure in the tail, which is called an **Axon Terminal**. The key point to understand here is that neurons by themselves are pretty useless. They can be compared to ants: a single ant is unable to create an anthill by itself, but many ants together can do this easily. Similarly, if a lot of neurons are connected together, they can carry out impressive work. The **Dendrite** and **Axon** allow the neurons to work together. The **Dendrites** receive signals and the **Axons** transmit them.

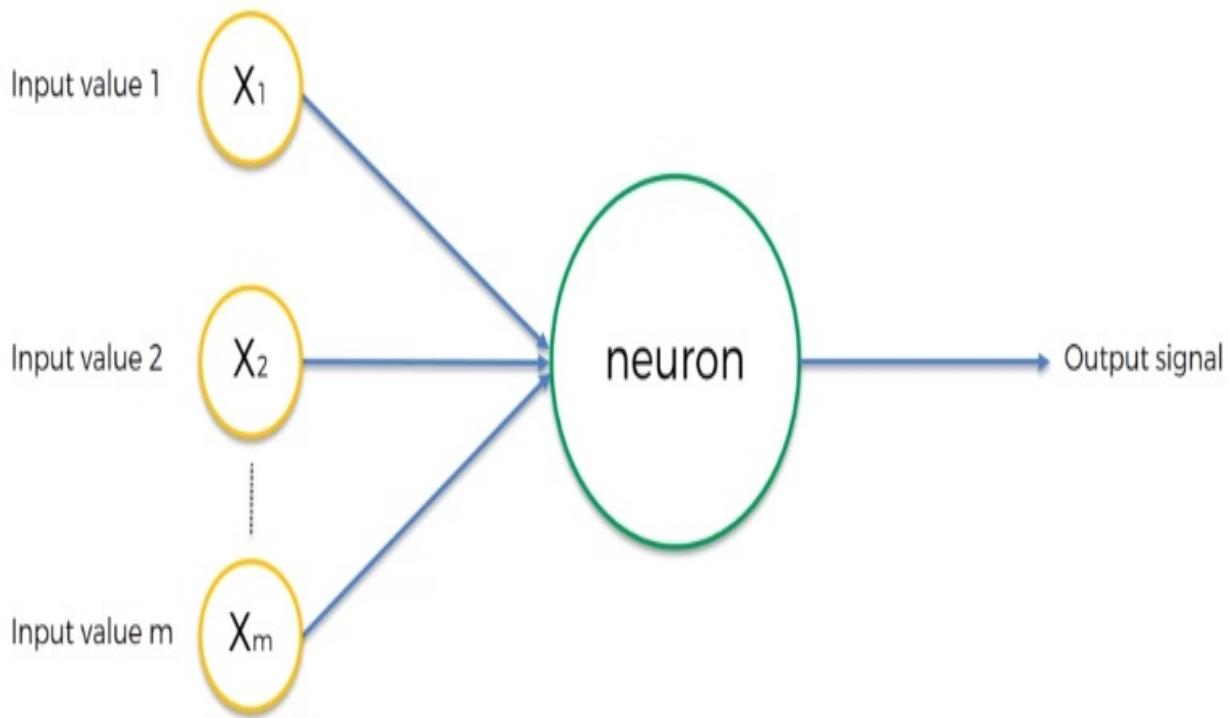
A neuron in a machine is called a node. It can be represented as follows:



The neuron gets some input signals and produces an output signal, as shown in the following diagram:



We can also represent the input signal as a node, as shown in the following diagram:



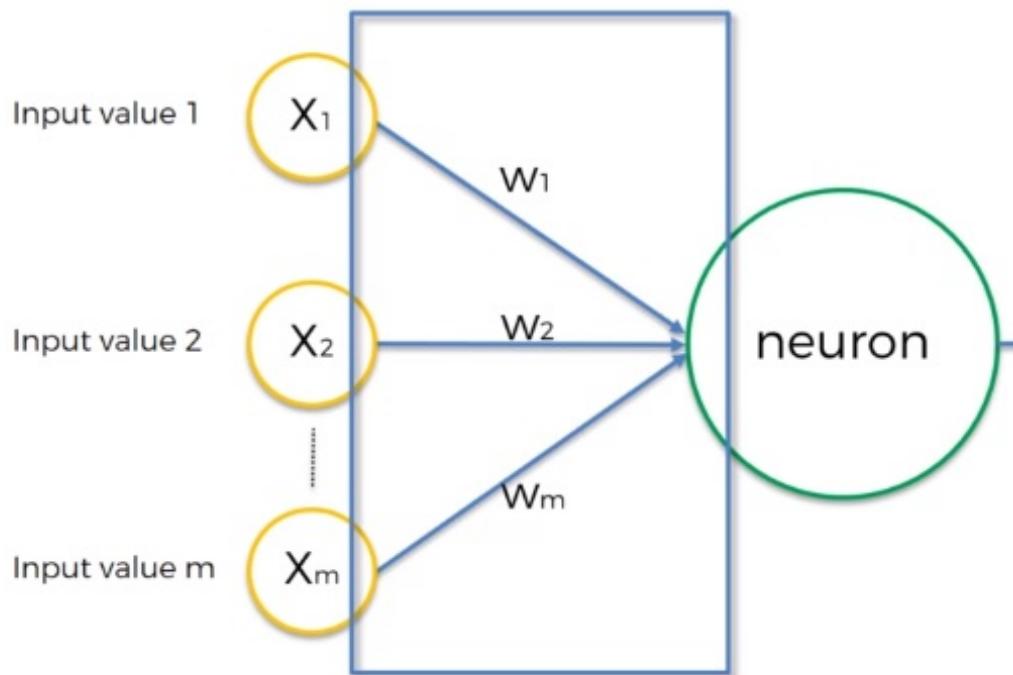
In the preceding diagram, all the nodes that represent the input are called the **input layer**. These inputs represent some information that the neurons receive. The layer in which the neuron is present, which is receiving the input, is called the **hidden layer**. The input layer can be thought of as your senses in the human brain analogy. Any information that comes from your sensory organs is an input, and it is processed by the neurons. We can't really see how the information is processed, so we consider these neurons to exist in hidden layers.

In machine learning and deep learning, we consider these input values as the independent features that are passed to the neurons. The neurons carry out some processing and provide an output. This output is sent to the other neuron, and this step continues until we get a final output. Let's take a look at the different layers in more detail:

- **Input layer:** The inputs in the input layer are the independent features or variables. One important thing to note is that these variables are all from a single observation. We can think of this layer as one row of our dataset. Let's look at an example: imagine we need to predict the price of a house based on the size and number of bedrooms in the house. The size and the number of bedrooms are the independent features or variables.

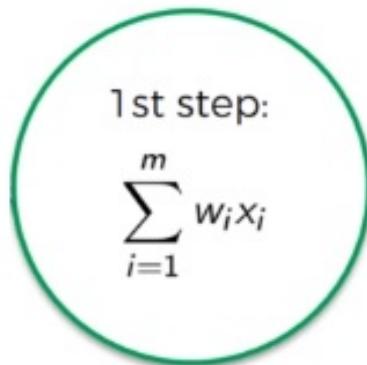
- **Hidden layer:** The hidden layer consists of the neurons that will receive the signal and provide an output.
- **Output layer:** The output layer provides the output value of the neuron from the hidden layer. The output values can be one of the following types:
 - **Continuous value (regression):** In the case of a continuous output, we will have one output
 - **Binary values (binary classification):** In the case of binary classification, we will have two outputs
 - **Categorical values (multi-class classification):** In this case, we will have multiple outputs

Each input that the neuron or the node receives is assigned or associated with a weight (w). Weights are crucial to the functioning of a neural network because this is how neural networks learn. By adjusting the weights, the neural network decides in every single case which signals are important and which signals are not important. The weights are assigned as follows:



The next important thing to understand is what happens inside a neuron. There are two main activities that take place:

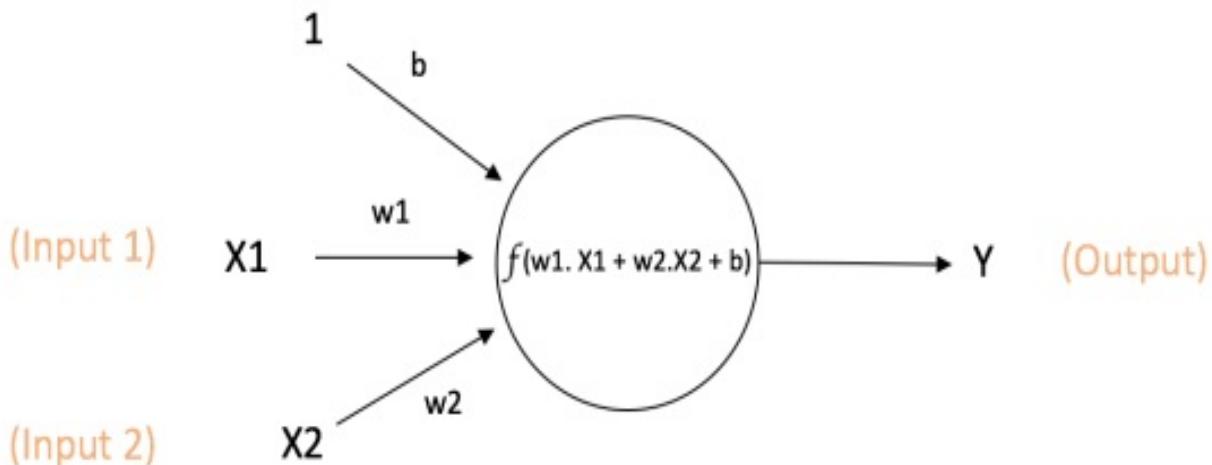
- The neuron finds the weighted sum of all input and weights, which is represented by the following equation:



- The neuron applies a function, (f), called an **activation function**, on the weighted sum of all the input. Based on this function, the neuron will either pass a signal or not. The equation is as follows:

$$\text{Output of the neuron} = Y = f(w_1 \cdot X_1 + w_2 \cdot X_2 + b)$$

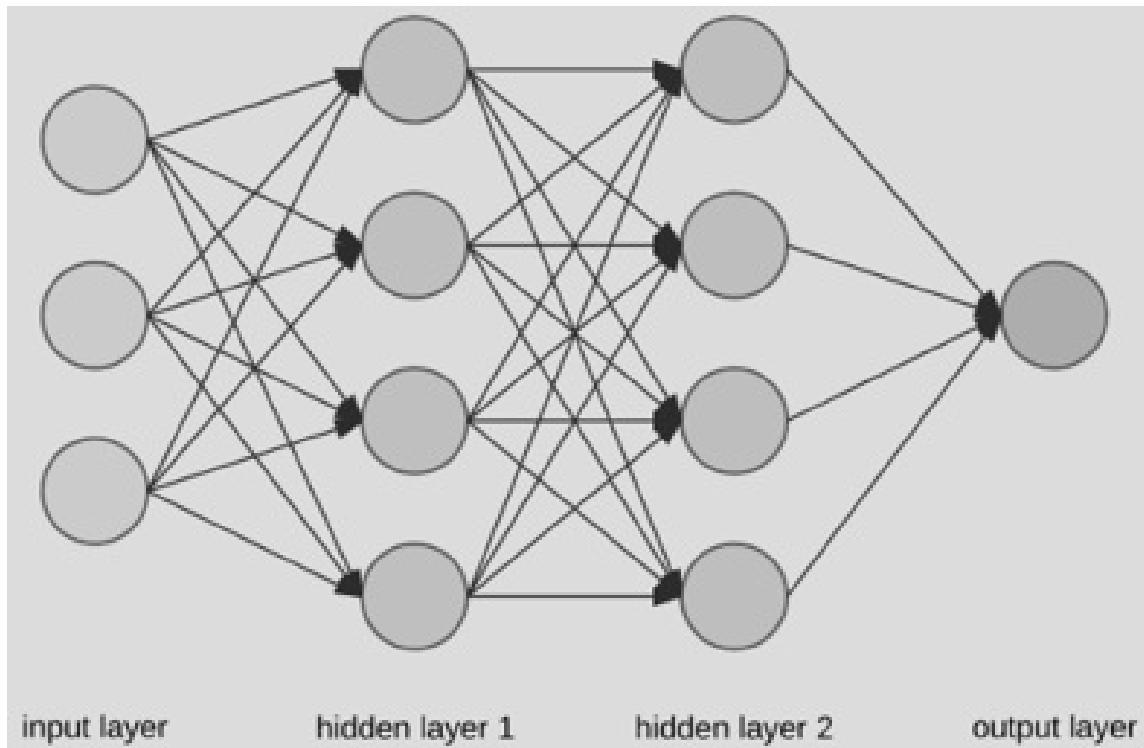
Diagrammatically, this can be represented as follows:



This activates or deactivates the neuron, and the weights get adjusted through the process of learning. When we train the neural network, we adjust the weights to get the right output. The preceding perceptron or single neural network takes two inputs: **X1** and **X2**. The perceptron has some associated weights: **w1** and **w2**. There is also another input that is specified by **b**, which is called **bias**.

A network with many hidden layers and many neurons in each hidden layer is

called a **multilayer neural network**. Diagrammatically, these can be represented as follows:



Types of neural networks

There are three main types of neural network:

- **Artificial Neural Network (ANN):** An ANN is a multi-layer neural network that can be used very efficiently in complex problems. It can solve regression, binary classification, and multi-class classification problems. This is one of the simplest neural networks, and it is also known as a feed-forward neural network. The data passes in one direction from the units or input nodes until it reaches the output node. This is called forward propagation. As the input, along with the weight, gets passed through the different layers, an activation function is also applied.
- **Convolutional Neural Network (CNN):** A CNN is a multi-layer neural network that contains a special layer, called a convolution layer. The convolution layer uses a convolution operation, which helps make the network deeper, with fewer parameters. Because of this, CNNs work very well for images, videos, natural language processing, and recommender systems. CNNs are used in agricultural sectors and health sectors, among others.
- **Recurrent Neural Network (RNN):** An RNN is a type of ANN in which the output that is predicted is fed back to the input. This helps to predict the outcome of that layer. Usually, the first layer is the same as that of the feed-forward neural network, which is the product of the weights and the input. The main functionality of an RNN begins from the subsequent layer. With every time step, t , passed, the RNN remembers the output of the previous time steps.

When we talk about bias, we are referring to bias on a per-neuron basis. We can think of each neuron as having its own bias term, so the entire network will be made up of several biases. The values that are assigned to the biases learn, just like the weights. The weights are usually updated by the gradient descent through back-propagation during training. Gradient descent is also learning and updating the biases. We will be learning about gradient descent in the *Gradient descent* section. First, let's take a look at the activation function.

The activation function

Applying the activation function is a very important step in a neural network. It brings non-linear properties to the neural network. If we do not apply an activation function, the output of the neuron will just be a simple linear function. We discussed linear regression in [Chapter 7, Regression Analysis in Finance](#). A linear equation can be easily solved, but they have a very limited power to learn more complex patterns. We want our neural network to solve any problem, no matter how complex it is. The neural network will not be able to learn or model other complex input, such as images, audio, text data, or videos, without using an activation function.

Types of activation functions

There are three common activation functions:

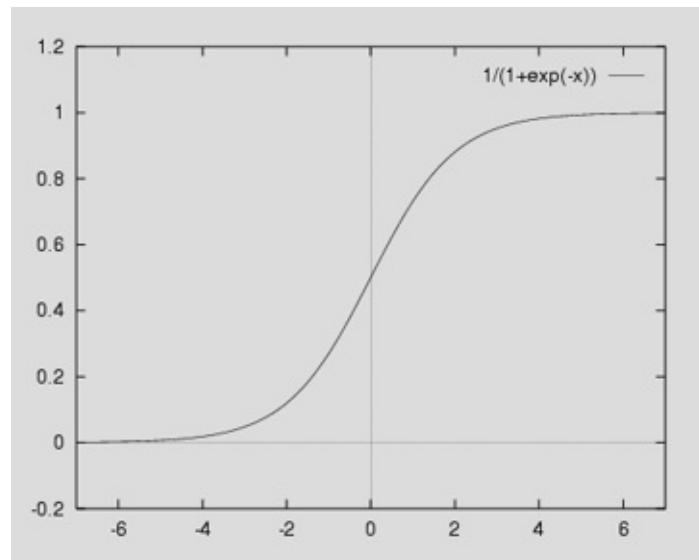
- Sigmoid activation function
- Tanh activation function
- **Rectified linear unit (ReLU)**

The sigmoid activation function

The sigmoid activation equation is represented by the following equation:

$$f(x) = 1 / (1 + \exp(-x))$$

This activation usually transforms the value of the weighted sum of the input and the weights to a value between 0 and 1. The equation can be represented as follows:



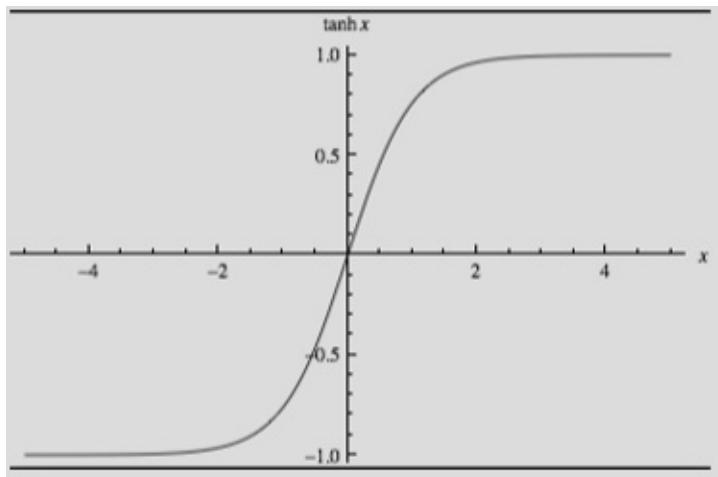
This activation function is usually used in the output layer of the neural network when we are trying to solve a binary classification problem.

The tanh activation function

This activation function is also called a hyperbolic tangent function. It is represented by the following equation:

$$f(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

This activation transforms the value of the weighted sum of the input and the weights to a value between -1 and 1. It is better than the sigmoid activation function, since the optimization process is much easier. The equation can be represented as follows:

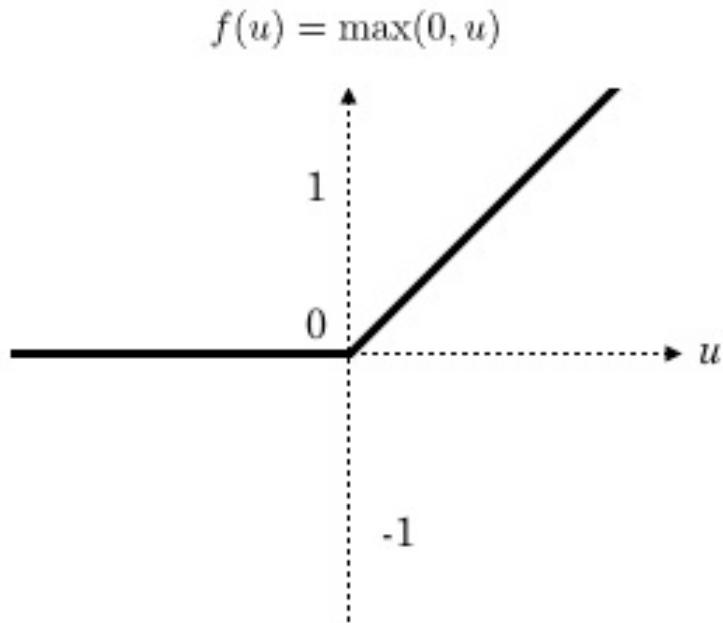


The ReLu activation function

This is currently the most popular activation function. It was also recently proven that this activation function significantly outperformed tanh in convergence. It is mathematically represented as follows:

$$F(x) = \max(0, x). \text{ If } x < 0, F(x) = 0, \text{ and if } x \geq 0, F(x) = x.$$

The graph is as follows:



Relu, however, has its own limitations. It can only be used with the hidden layers of a neural network. Therefore, for all the output layers, the common activation function that's used is sigmoid, while ReLu is used for the hidden layers. Let's move on and understand how neural networks work.

What is bias?

When we talk about bias, we are referring to bias on a per-neuron basis. We can think of each neuron as having its own bias term, so that the entire network will be made up of several biases. The values that get assigned to the biases also get updated, just like the weights. The weights are usually updated by the gradient descent through back-propagation during training. Gradient descent is also learning and updating the biases. We will be learning about gradient descent in the *Gradient descent* section. First, let's take a look at the activation function.

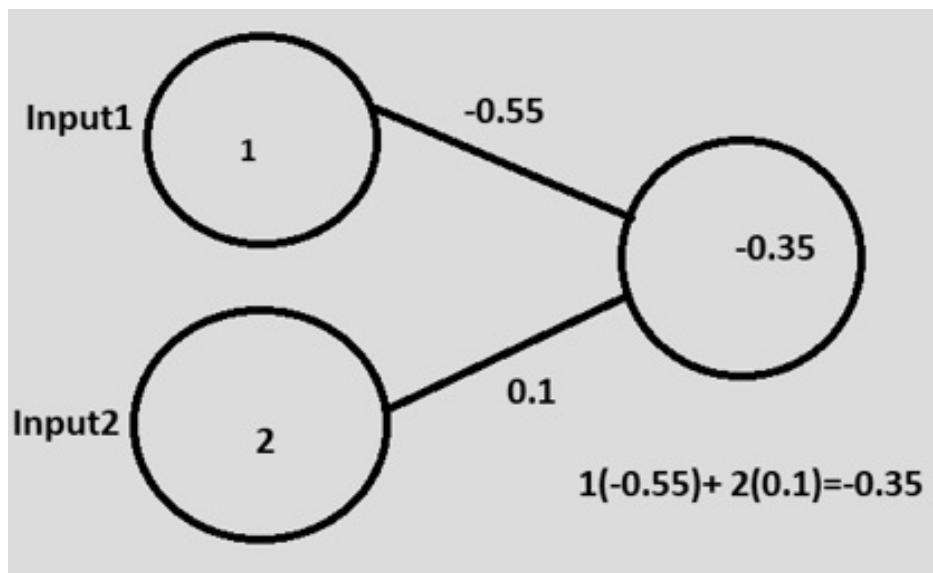
We can think of the bias of each neuron as having a role similar to that of a threshold. The bias value determines whether the activation output from a neuron is propagated through the network. In other words, the bias determines whether or not or by how much a neuron will be activated or fired. The addition of these biases ends up increasing the flexibility of the model to fit the given input data.

Each neuron receives a weighted sum of input from the previous layers, and then that weighted sum gets passed through an activation function. This is where the bias is added. The equation is as follows:

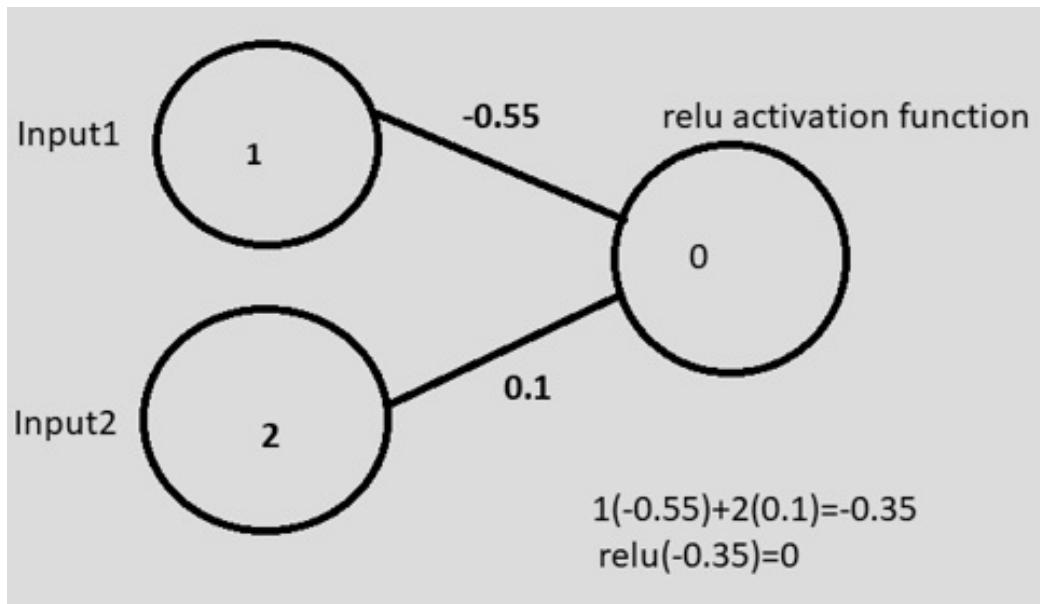
$$\text{Output of the neuron} = Y = f(w_1 * X_1 + w_2 * X_2 + b)$$

Rather than passing the weight directly to the activation function, we pass the weighted sum plus the bias term to the activation function instead.

To illustrate the importance of bias, let's consider an example. Think about a single neural network with two inputs and one neuron. The weight initialized and the input have the values that are shown in the following diagram:



As the inputs are forwarded to the neuron, they are multiplied by their respective assigned weights, which are $1(-0.55) + 2(0.1) = -0.35$. Then, we pass this result to the ReLu activation function. We know that the value of the ReLu activation function for any given input will be the maximum of either 0 or the input itself. In our case, the maximum of -0.35 and 0 is 0:



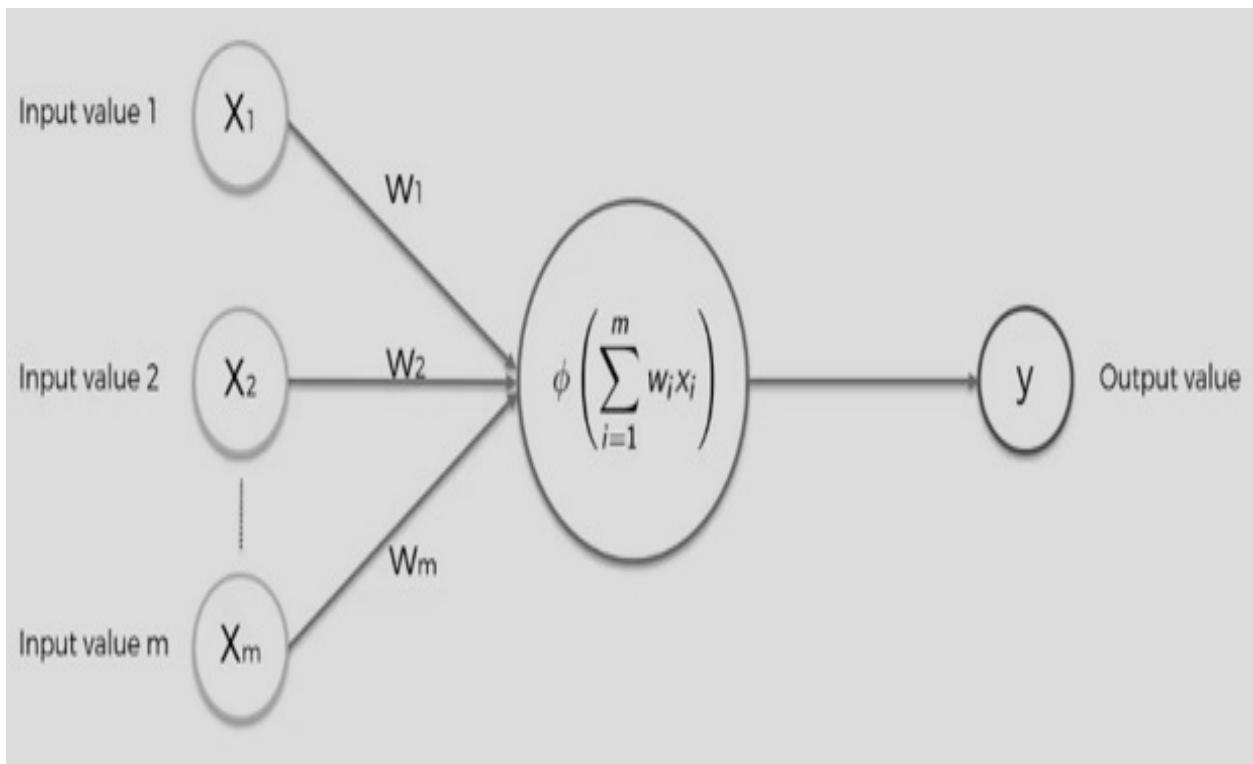
Now, since the output is 0, this neuron is considered not to be activated. This is always the case with ReLu when the weighted sum of the input is less than or equal to 0. No information from these non-activated neurons will be passed to the rest of the network. Essentially, 0 is the threshold here for the weighted sum in determining whether a neuron is activated.

Let's say we want to shift this threshold. Instead of 0, let's say we want a neuron to fire if its input is greater than or equal to -1. This is where bias comes into play. The bias is added to the weighted sum before being passed to the activation function. The value we assign to the bias is the opposite of the threshold value. If we want our threshold to move from 0 to -1, the bias will be the opposite of -1, which is just 1. Now, our weighted sum is $0.35 + 1$, which is 0.65. Passing this to the ReLu activation function gives us the maximum of 0.65 and 0, which is 0.65. The neuron that didn't fire before is now considered to be firing. The model has a bit more flexibility in fitting the data, since it now has a broader range of values that it considers activated or not.

How do neural networks learn?

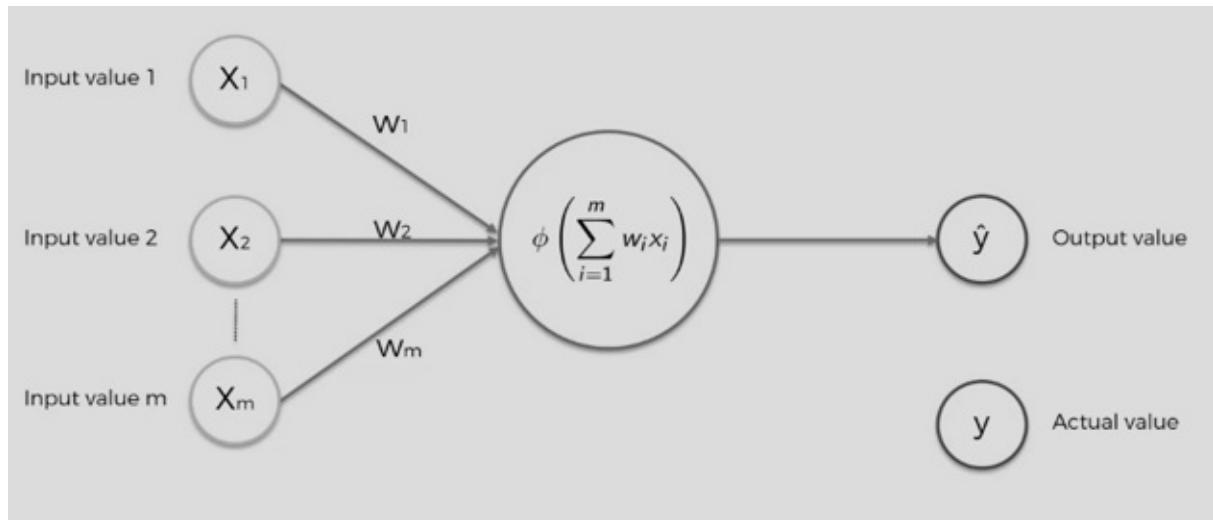
Now that we have explored the various components of a neural network, we will discuss how it learns. There are two fundamental approaches to get a program to do what you want it to do. The first method is hardcoding, where you actually tell the program specific rules and what outcomes you want, guiding it and accounting for all the different options that the program has to deal with. The other method is to use neural networks, where we provide the program with the ability to understand what it needs to do on its own. We provide the input and we request the output, and then we let the neural network figure the rest out on its own.

Our goal is to create a network that learns on its own. Let's consider the following diagram of a neural network:

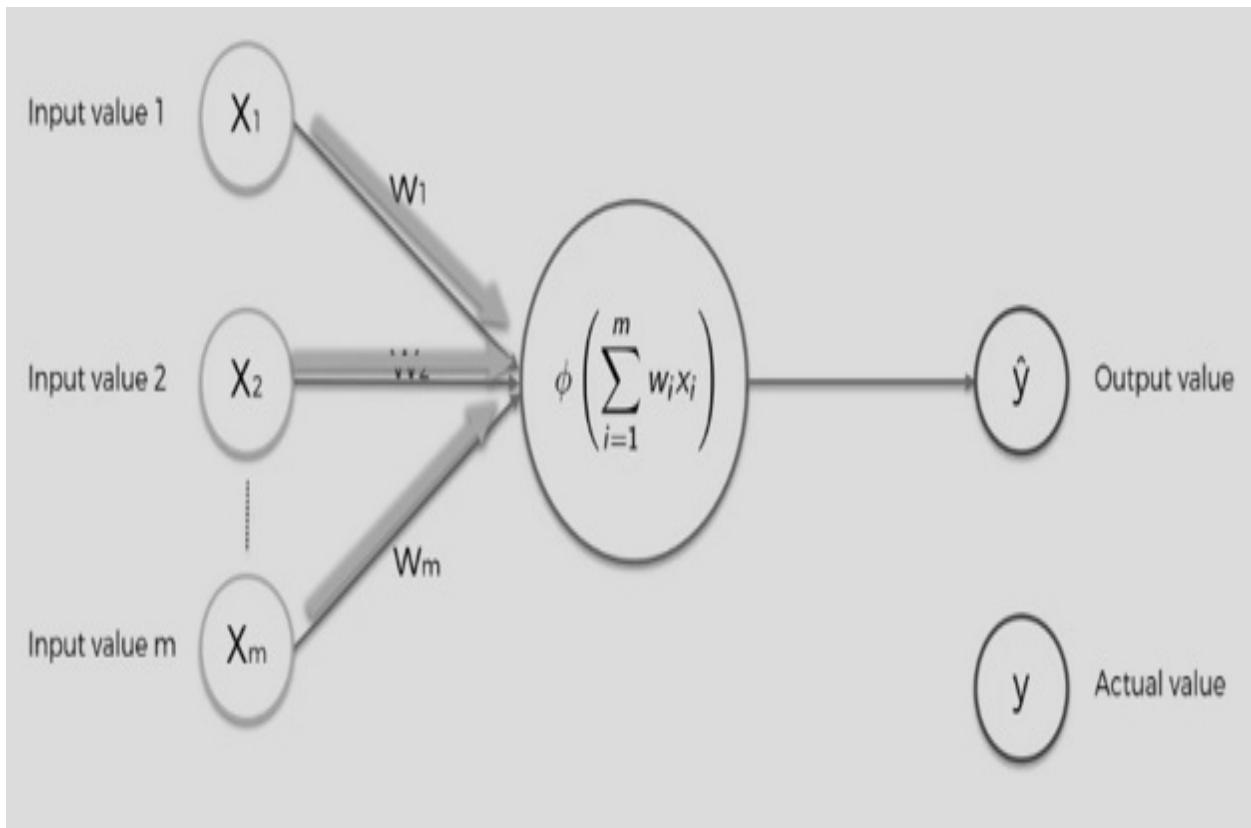


In the preceding diagram, we have a basic neural network, which is called a **perceptron**. The perceptron is a single-layer feed-forward neural network. Before we proceed, we need to adjust the output value. Currently, the output is

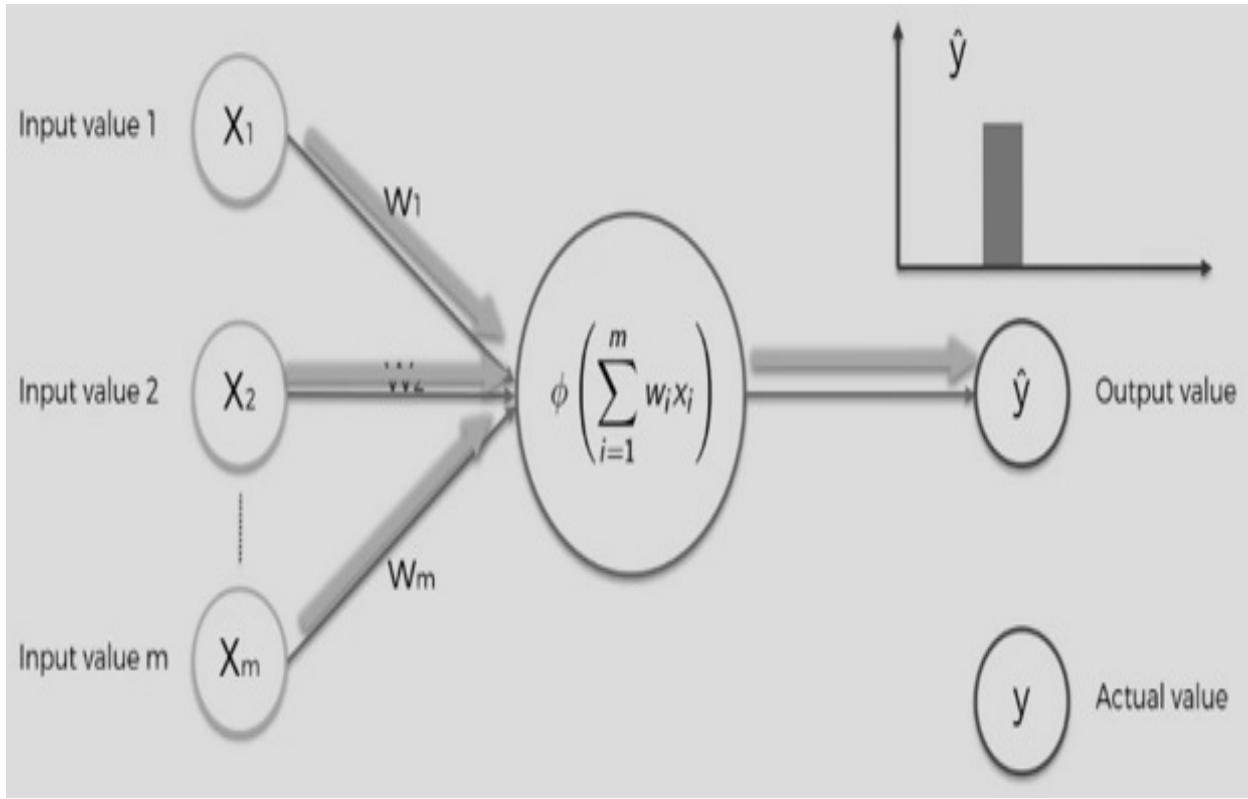
represented as y , but we will write it as \hat{y} here to indicate that it is a predicted value rather than the actual value:



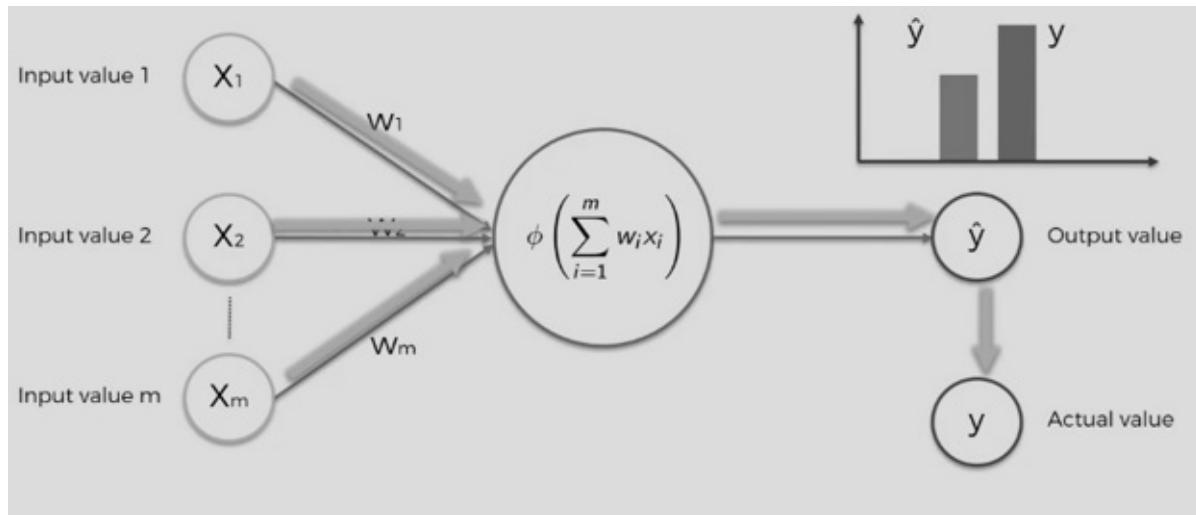
The perceptron was first invented in 1957 by Frank Rosenblatt. Let's think about how our perceptron learns. Let's say that we have some input values that have been supplied to the neuron or the perceptron:



Then, the activation function is applied. We have an output that we are going to plot on a chart, as shown here:



Now, if we need to make the neural network learn, we need to compare the output value (\hat{y}) to the actual value (y) that we want the neural network to achieve. If we plot this on a graph, we will see that there is a difference:

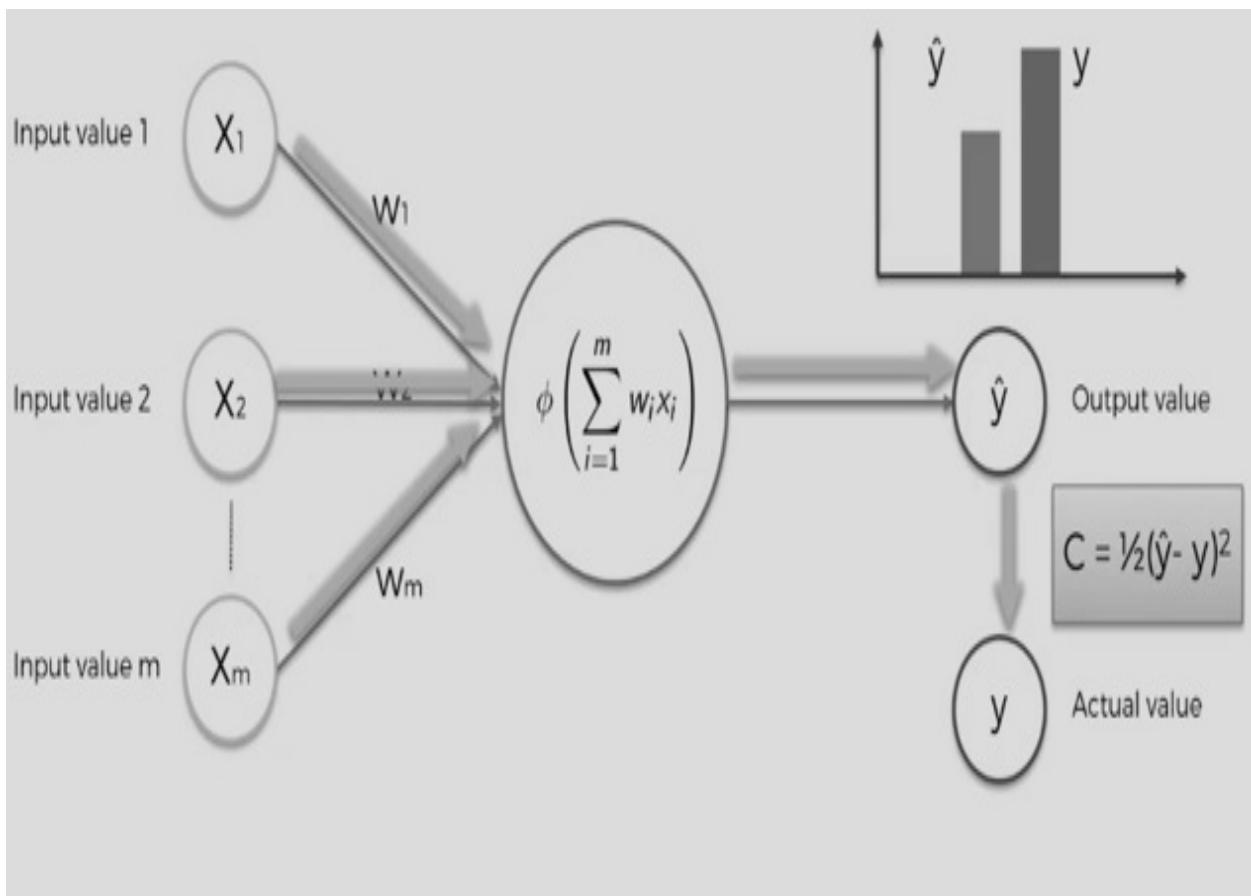


We will now try to calculate the function called the **cost function**, C , which is represented as follows:

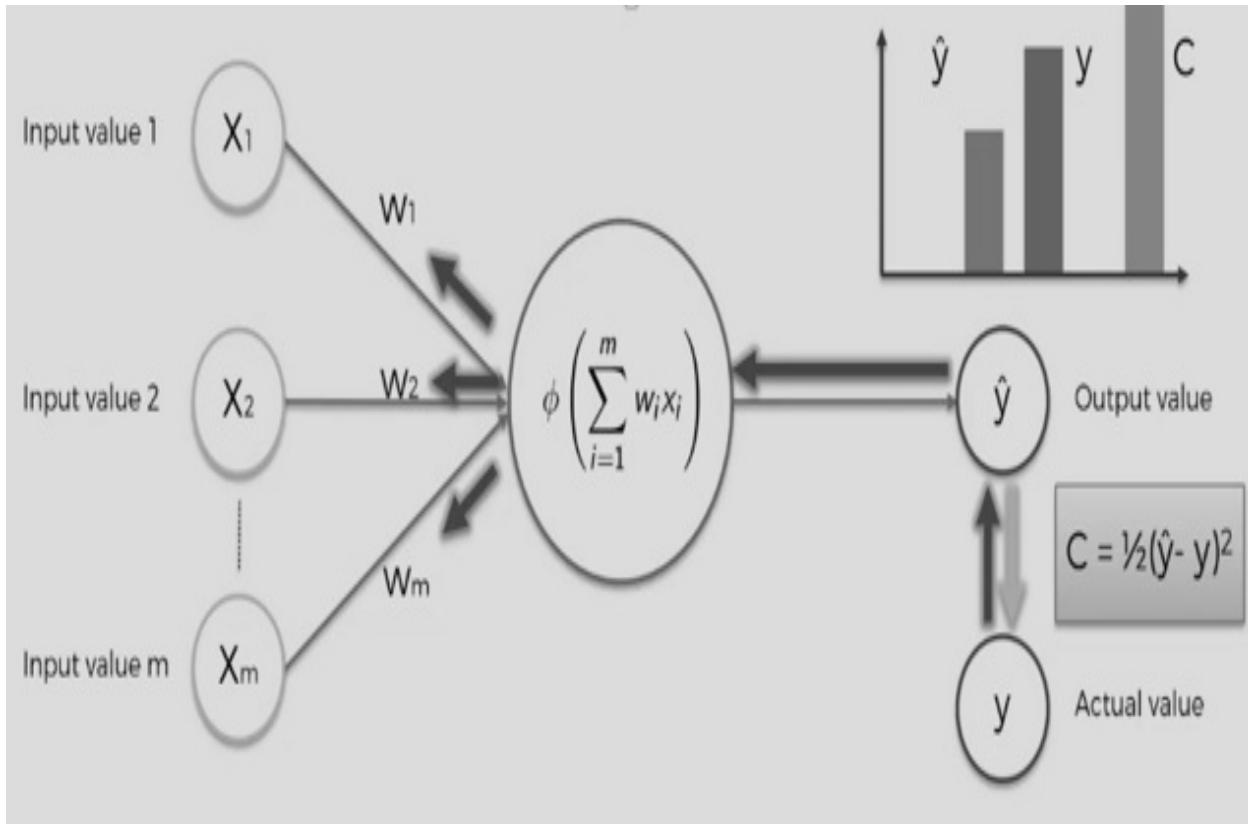
$$C = \frac{1}{2}(\hat{y}-y)^2$$

The cost function can be thought of as one half of the squared difference between the actual value and the predicted value. There are many other cost functions that we can use, but this is the most common one. It tells us about the error that can be found in our prediction. Our goal is to minimize the cost function.

Here is a diagram that indicates the cost function:



Once we compare the cost function, which is the error, we will feed this information back to the neural network. As the information goes into the neuron, all the weights are updated:



The only thing that we control in this neural network is the weights ($w_1, w_2, w_3, \dots, w_m$). After the weights have been updated, we will resend the input and the whole process will be repeated. We will compute the weighted sum of all the input and weights, and then pass this through the activation function. We will then compute the cost function again. After this, we send the information again, the weights are updated, and so on. This training is continued until our cost function is minimized.

Currently, we only have one input that is passed to a neuron. In a real scenario, however, we are likely to have many records. All of these input records need to be fed to the neural network.

Let's introduce a new term, called an **epoch**. One epoch is when we go through the complete dataset and train our neural network on all of the rows. In one epoch, for each and every input record, we get a predicted value, which is then compared to the actual value. Based on the difference between the actual and the predicted values, we can calculate the cost function, which is the sum of all of the squared differences between the predicted value and the actual value, as shown here:

$$C = \sum \frac{1}{2}(\hat{y} - y)^2$$

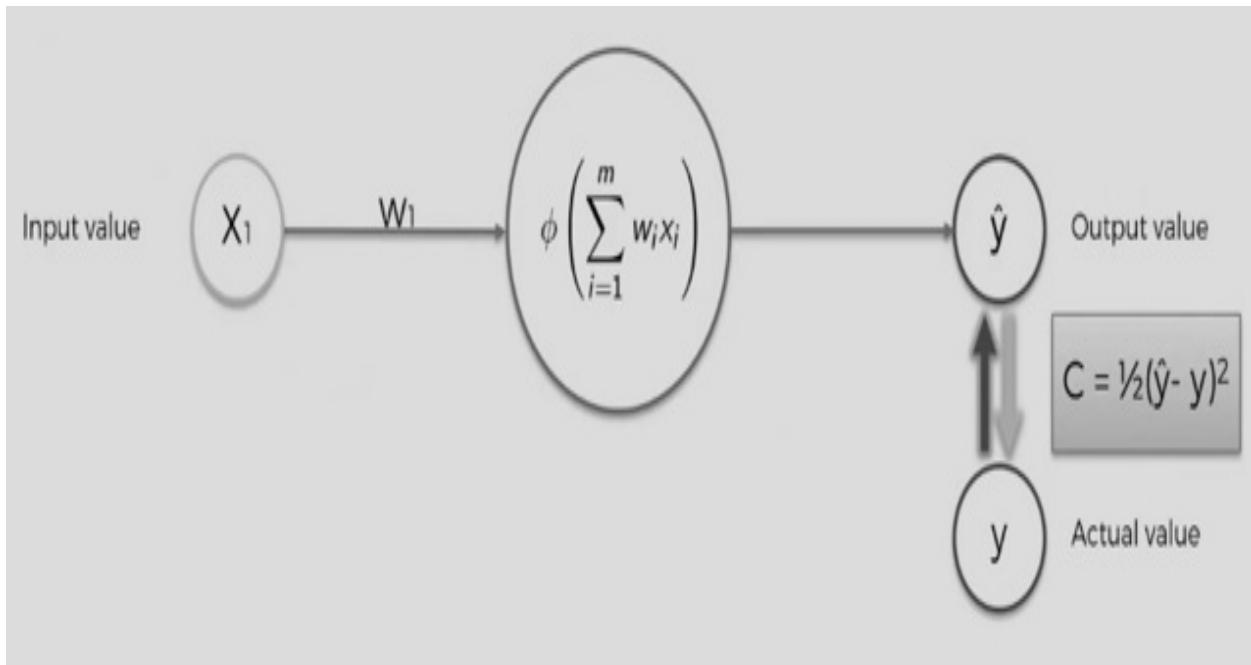
Once we have the full cost function, the neural network goes back and updates the weights ($w_1, w_2, w_3, \dots, w_m$). This step is called **back-propagation**.

Next, we are going to run all the input records again. We will feed every single row into the neural network, find out the cost, and then update the weight and do this whole process again. The final goal is to minimize the cost function.

Gradient descent

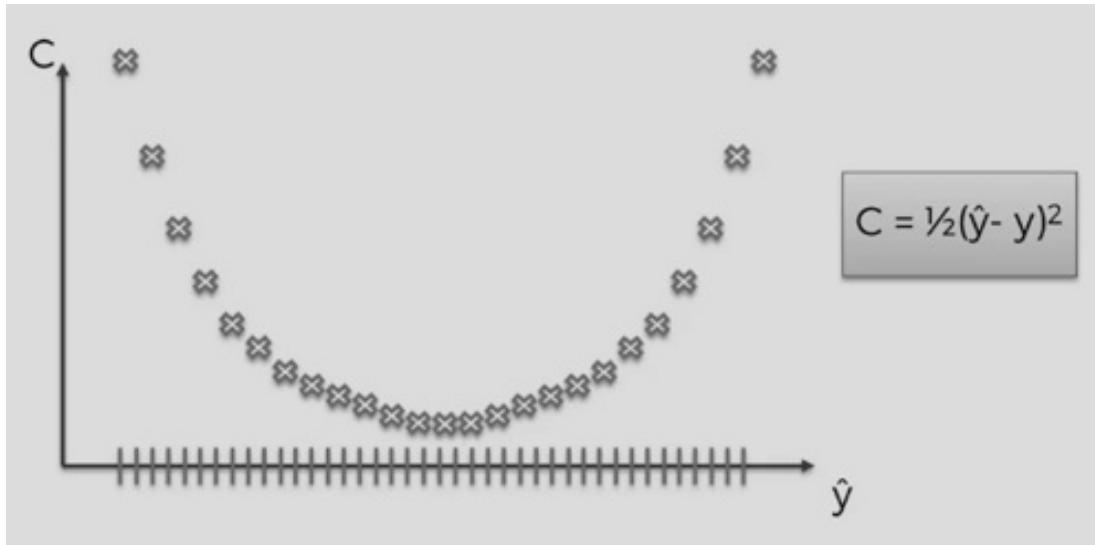
In the previous section, we learned that for a neural network to learn, we need back-propagation. In this section, we will learn how the weights are adjusted.

Let's consider a very simple version of a neural network, a perceptron, or a single feed-forward network:



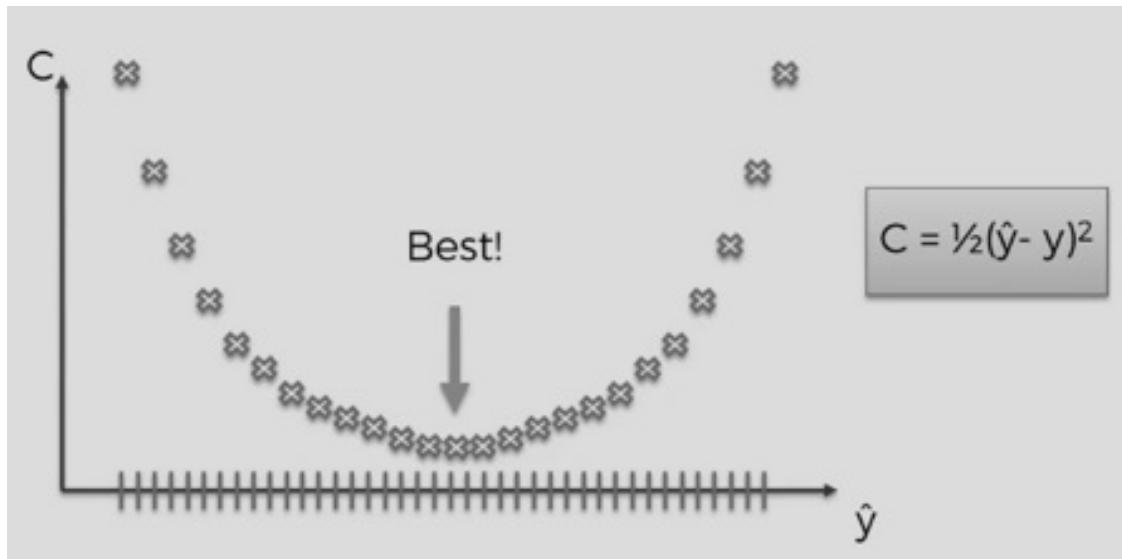
Again, here, we have the whole action in process. The neuron receives an input and then we get the product of the weights, w , and the input, x . Later, an activation function is applied, so we get \hat{y} . We compare the predicted value with the actual value and calculate the cost function, C .

To minimize the cost function, we could use a brute-force approach, where we just take lots of different possible weights and look at them to see which one looks best. Let's say that we try to compute the cost function for 1,000 different weights and plot them on a graph:



The preceding graph indicates the cost function value in the y axis and \hat{y} , which is the predicted value based on different weights in the x axis.

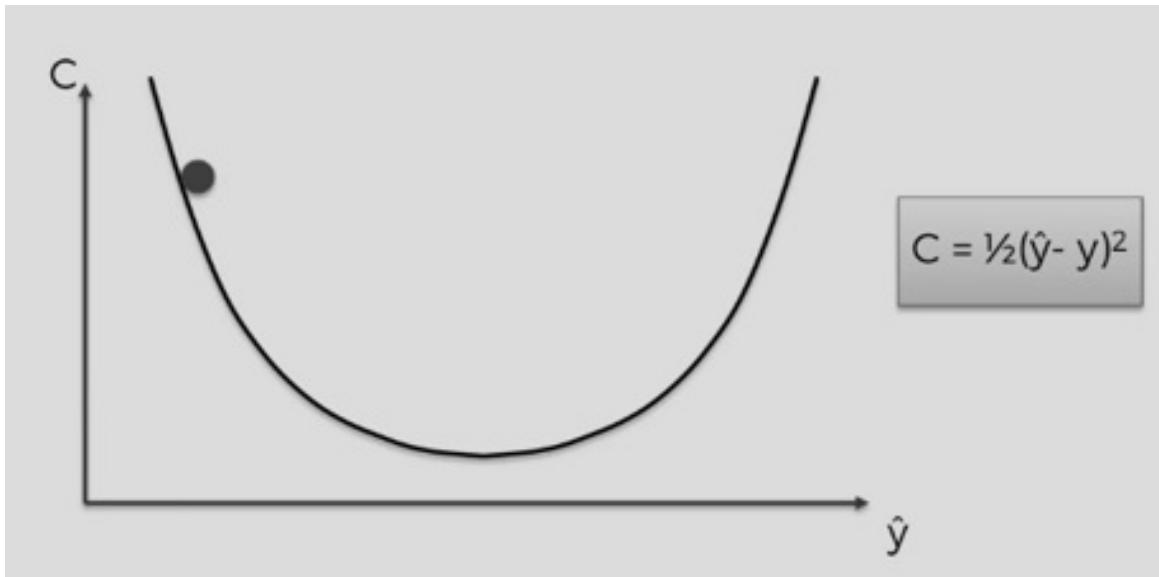
We can see that there is a global minima, which is indicated as follows:



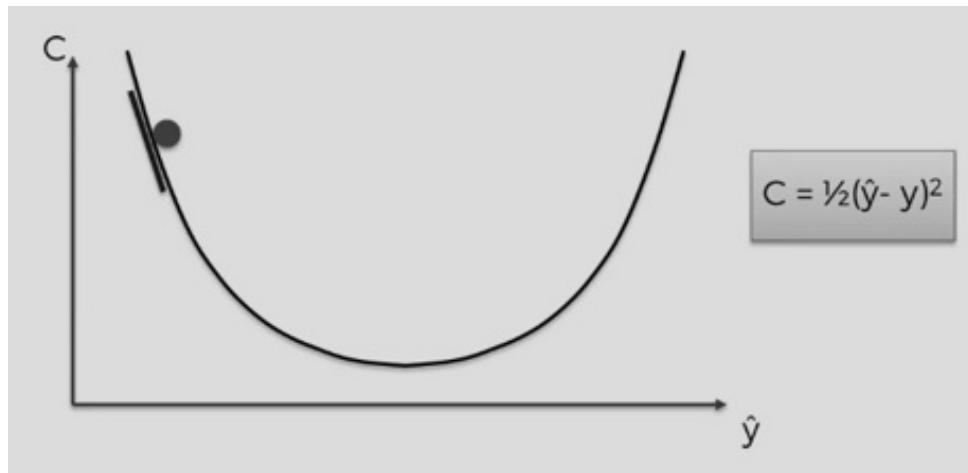
We usually do not follow the brute-force approach, because, as the number of weights increases, the number of connections to the neurons also increases. We may face the curse of dimensionality as a result, which is where the performance and accuracy of the model decrease as the number of features or independent features increases.

Instead of this, we can use a technique called gradient descent. This helps us to

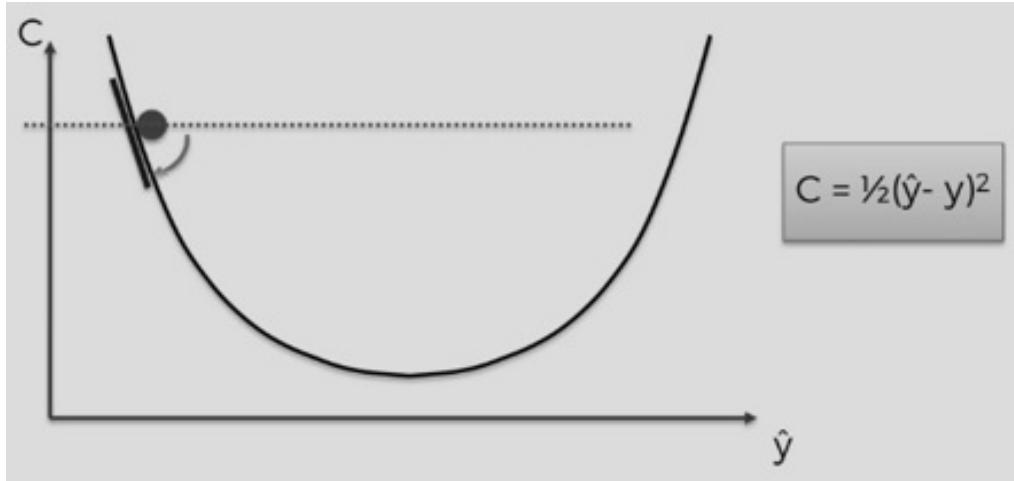
find the best point in the preceding graph without initializing many weights. Let's consider the same cost function, C , and see how we can quickly find the best point in the graph. First, we will initialize some weights. We will get a predicted output as \hat{y} and the cost function, as shown here:



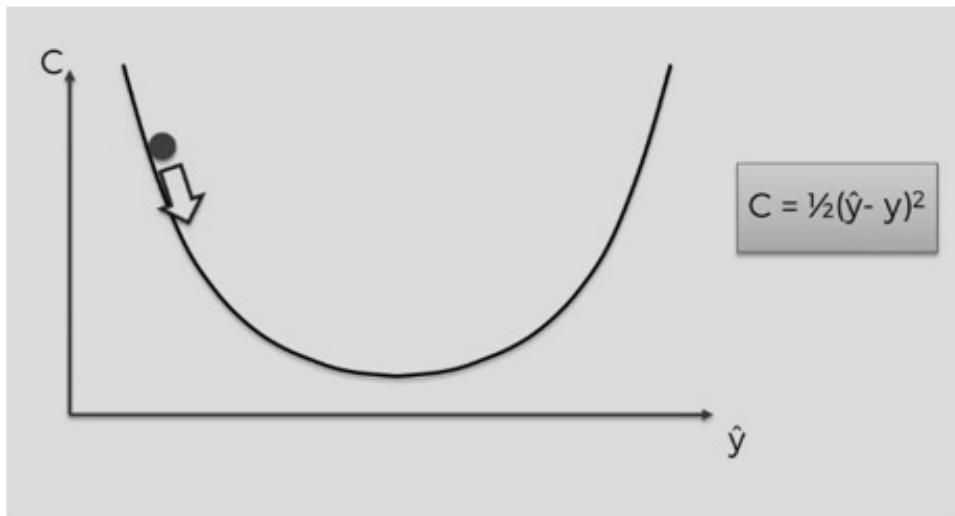
Now, we are going to look at the angle of our cost function at the following point. This is called the gradient and we find it using differentiation:



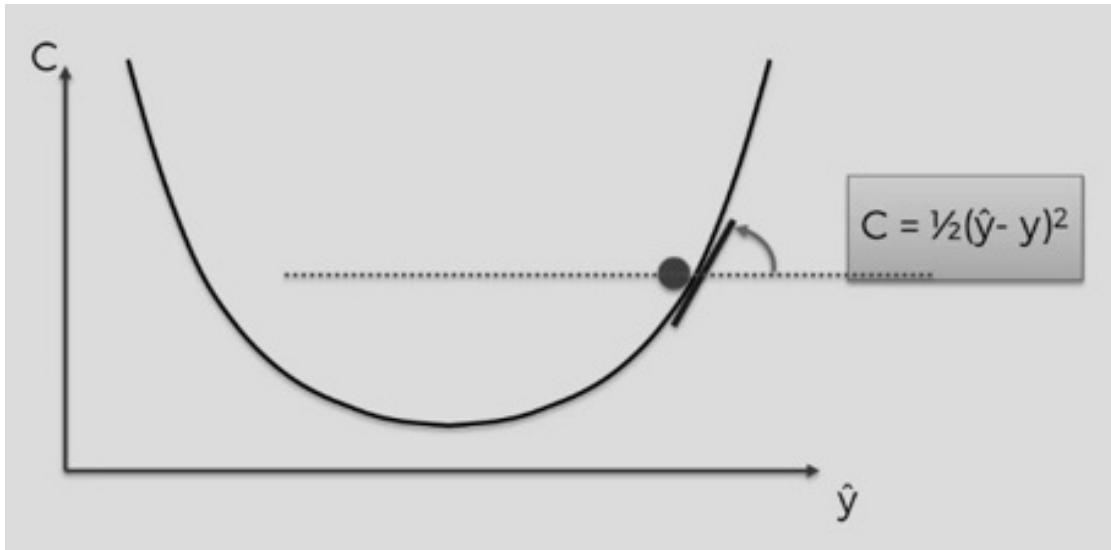
The reason we differentiate is basically to find out the slope in that specific point and see whether it is negative or positive. The following diagram indicates that the slope is negative:



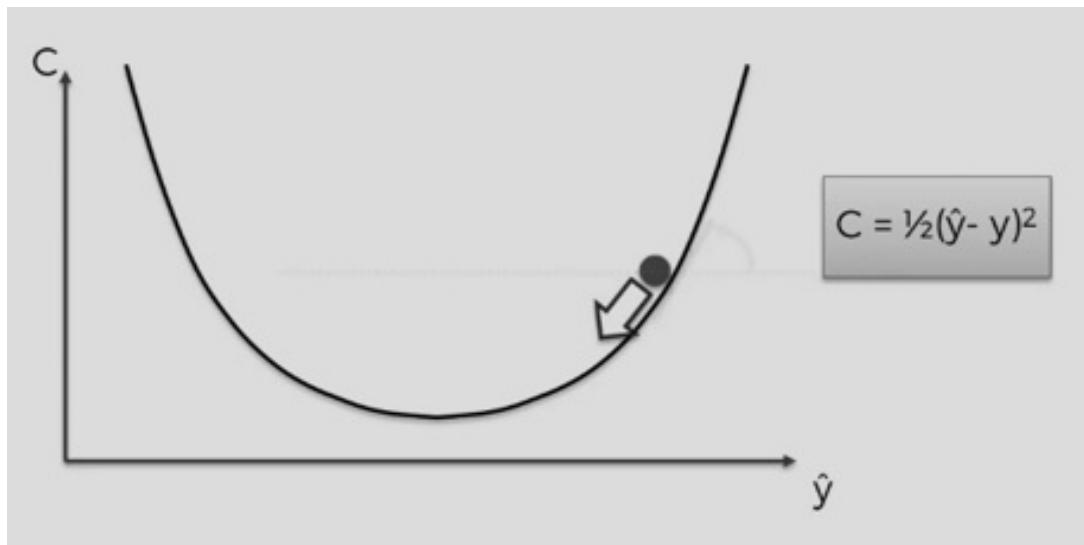
If the slope is negative, as it is in this case, this means we are going downhill. Therefore, we need to move to the right:



The following diagram indicates that the slope is positive:

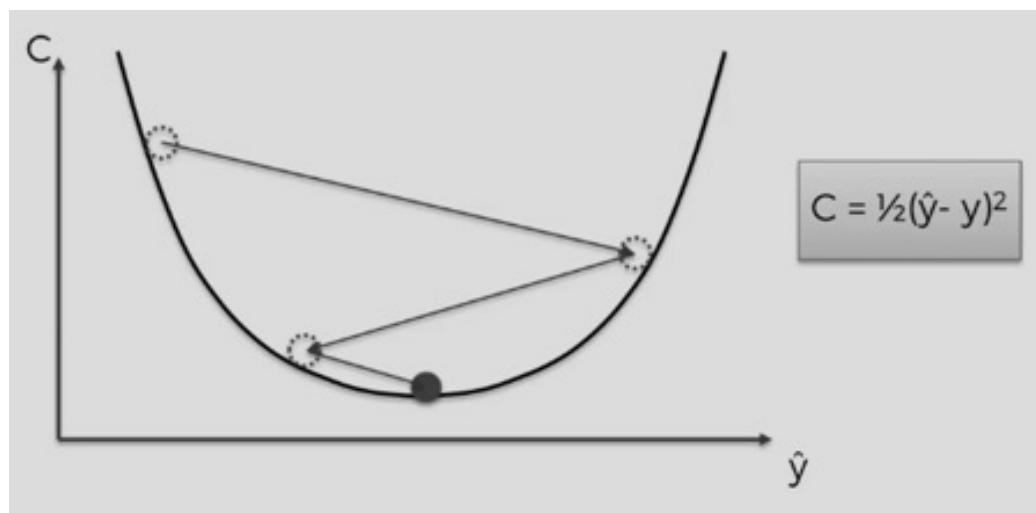


If the slope is positive, this means the line is uphill, so we have to move to the right, as shown here:



We now need to consider a concept called the **learning rate**. Usually, the learning rate is initialized with a small value between 0.01 and 0.0001. The learning rate is the rate at which we want the neural network to converge to reach the point of best fit. If we initialize a higher value for the learning rate, it may never converge to the point of best fit. This value needs to be carefully selected.

After continuing this process and moving downward, we will reach the global minimal point, and the graph will be as follows:



An introduction to TensorFlow

TensorFlow is an open source deep learning framework that was released by Google Brain Team in November 2015. TensorFlow was created based on Google's own proprietary machine learning framework, DistBelief. TensorFlow usually supports multiple core CPUs and faster GPUs with good processing power by using the Nvidia CUDA library. TensorFlow is compatible with all 64-bit operating systems, such as Linux or macOS. It also supports mobile operating systems, such as Android and iOS.

The simple architecture of TensorFlow allows for the easy division of computation between the CPU and the GPU. Its computations are expressed as computation graphs, which are usually created using the neural network architecture, which performs various operations on multi-dimensional arrays. These are referred to as **tensors**.

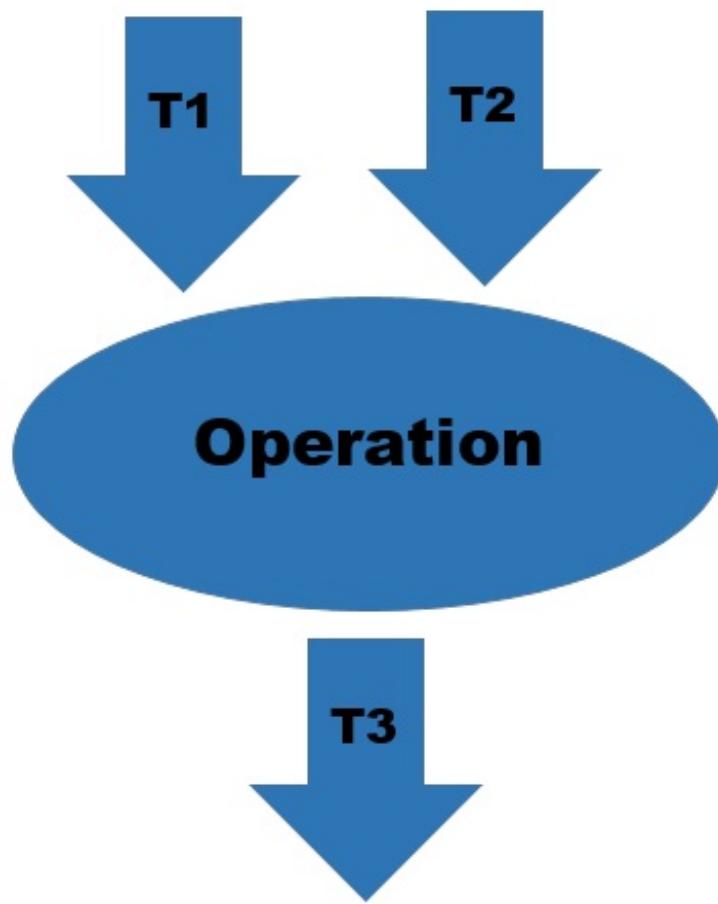
A tensor is a mathematical representation of a physical entity, which is characterized by a magnitude and a direction. TensorFlow provides primitives that define functions on tensors and automatically compute their derivatives.

TensorFlow exposes various **Application Programming Interfaces (APIs)**. There are two main types of APIs:

- **Low-level APIs:** This API provides complete programming control. It is usually recommended for machine learning researchers and provides a fine level of control over the deep learning models. TensorFlow Core is the low-level API of TensorFlow.
- **High-level APIs:** These are built on top of TensorFlow Core. They are easier to understand and use than TensorFlow Core, which is the low-level API. These high-level APIs make repetitive tasks easier and more consistent between different users. `Tensorflow.contrib.learn` is an example of a high-level API.

TensorFlow provides computation using stateful dataflow graphs, as shown here:

TensorFlow stateful Graph



Here, **T1** and **T2** are the input tensors and **T3** is the resultant tensor. The node represents the operation and the edges represent the multi-dimensional data arrays (called tensors) communicating between them. There are two main types of TensorFlow Core programs:

- **Building a computational graph:** A series of TensorFlow operations arranged into a graph of nodes.
- **Running a computational graph:** To evaluate the nodes, we must run the computational graph within a session. A session encapsulates the control and the state of the TensorFlow runtime.

Let's look at an example of how we can add two numbers using TensorFlow. Follow these steps:

1. Import the TensorFlow library:

```
In[1]:  
# importing tensorflow  
import tensorflow as tf
```

2. Let's create the nodes that form the input. Currently, we have two inputs as we want to do an addition operation:

```
In[2]:  
# Creating nodes in computation graph  
# Constant node takes no inputs, and it outputs a value it stores internally.  
# We can also specify the data type of output tensor using dtype argument.  
node1 = tf.constant(5, dtype=tf.int32)  
node2 = tf.constant(6, dtype=tf.int32)
```

Here, we initialized both the input as constant values, in this case, 5 and 6. This will prevent changes being made to the input as the program is executed.

3. Create a third node, which will consist of the operation, which is addition. The parameters will be `node1` and `node2`:

```
In[3]:  
# Operation performed on the node1 and node2.  
node3 = tf.add(node1, node2)
```

4. Create a TensorFlow session object:

```
In[4]:  
# Creating Tensorflow sessions object.  
sess = tf.Session()
```

5. Evaluate the whole computation graph, which is based on the input nodes. The add operation will be performed and we will get an output:

```
In[5]:  
# Evaluating the computation graph.  
print("Sum of node1 and node2 is:", sess.run(node3))
```

The output of the preceding operation is shown here:

```
Out[5]:  
Sum of node1 and node2 is: 11
```

6. Close the session that we have created, indicating the end of the program. The code is shown here:

```
In[6]:  
## Closing session  
sess.close()
```

Now, let's discuss the two main types of tensor objects in a graph, which are variables and placeholders:

- **Variables:** TensorFlow variables are completely different from constants. They need to be initialized and their values change as their execution passes through the computation graph. Whenever we create a session, these TensorFlow variables need to be initialized. The variable's ID is usually created with the following syntax:

```
In[1]:  
import tensorflow as tf  
w = tf.Variable(False)  
type(w)
```

In the preceding code, we defined a Boolean variable using TensorFlow and the datatype of the variable, as shown here:

```
Out[1]:  
tensorflow.python.ops.variables.RefVariable
```

- **Placeholders:** TensorFlow placeholders are usually initialized as empty and are used to feed the input data once the computation graph is ready to be executed in a session:

```
In[1]:  
import tensorflow as tf  
w = x = tf.placeholder(tf.float32, [None, 784])  
type(w)
```

In the preceding code, we defined a placeholder using TensorFlow and the datatype of the variable:

```
Out[1]:  
tensorflow.python.ops.placeholder.Placeholder
```

We will now discuss how we can implement linear regression using TensorFlow

Implementing linear regression using TensorFlow

In this section, we will look at an example of implementing a linear regression neural network using TensorFlow:

1. Import the `numpy` and `matplotlib` libraries. The `numpy` library will help us to create a dataset. The code is as follows:

```
In[1]:  
import numpy as np  
import matplotlib.pyplot as plt  
%matplotlib inline
```

2. Create a million equally spaced points between 0 to 10 using the NumPy `linspace()` function. This data will be our input data. The code is as follows:

```
In[2]:  
# 1 Million Points  
x_data = np.linspace(0.0,10.0,1000000)
```

3. Add some noise in the dataset so that it looks like a real-world problem. Here, we are using the `randn()` function that is present in NumPy. The code is as follows:

```
In[3]:  
noise = np.random.randn(len(x_data))
```

4. Use the $y = mx + b + \text{noise_levels}$ equation to create the dependent feature, which can be represented as `y_true`:

```
In[4]:  
# y = mx + b + noise_levels  
b = 5  
  
y_true = (0.5 * x_data) + 5 + noise
```

5. Take `x_data`, which is the independent feature, and `y_true`, which is the dependent feature, and combine them into a dataframe. The code is as follows:

```
In[5]:  
my_data = pd.concat([pd.DataFrame(data=x_data,columns=['X Data']),pd.DataFrame(d
```

6. The following code helps us see the top five records of the dataset:

```
| In[6]:  
| my_data.head()
```

The output is as follows:

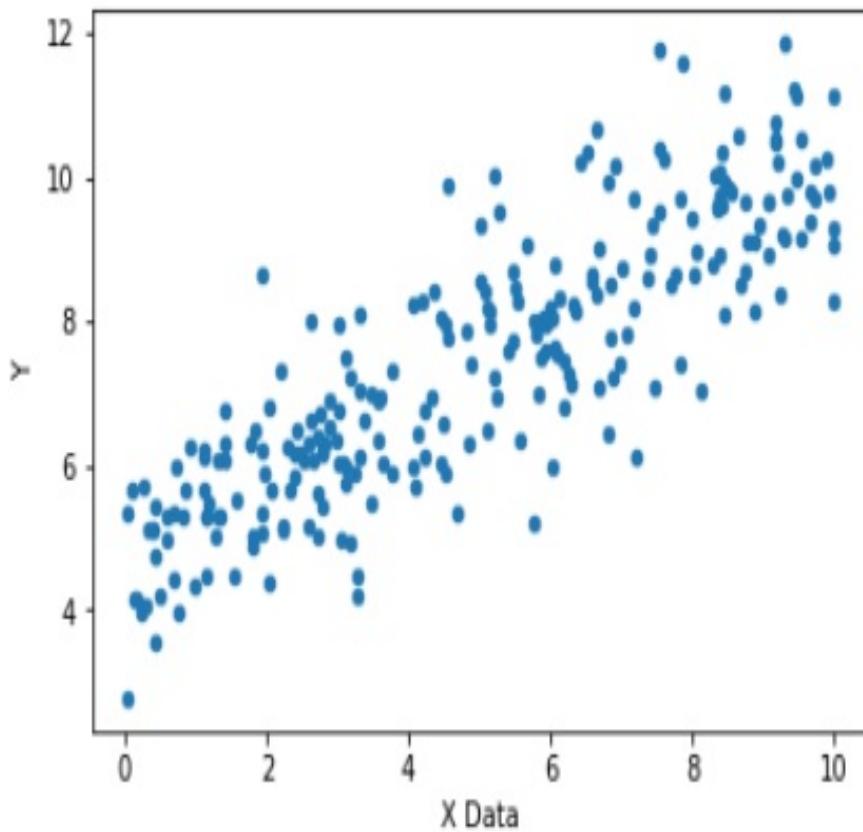
	Out[6]:	X Data	Y
0	0.00000	4.763854	
1	0.00001	4.058862	
2	0.00002	5.625209	
3	0.00003	4.555293	
4	0.00004	3.655031	

7. Let's consider the top 250 records and plot the dataset:

```
| In[7]:  
| my_data.sample(n=250).plot(kind='scatter',x='X Data',y='Y')
```

The output is as follows:

```
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x17c74679390>
```



Once we have created a dataset, we need to create a linear regression neural network using TensorFlow.

8. Import TensorFlow:

```
| In[8]:  
|     import tensorflow as tf
```

9. Initialize the `batch_size` variable to 8:

```
| In[9]:  
|     batch_size = 8
```

10. We will be using the linear regression equation, $y = mx + c$. Create variables using TensorFlow. The code is as follows:

```
| In[10]:  
|     ## Variables  
|     m = tf.Variable(0.5)
```

```
|     b = tf.Variable(1.0)
```

11. Create placeholders for the input and output variables. The code is as follows:

```
| In[11]:  
| xph = tf.placeholder(tf.float32, [batch_size])  
| yph = tf.placeholder(tf.float32, [batch_size])
```

12. Create the computation graph:

```
| In[12]:  
| y_model = m*xph + b
```

13. Create the cost function, usually represented by $C = 1/2(\hat{y}-y)^2$, with the following TensorFlow code:

```
| error = tf.reduce_sum(tf.square(yph-y_model))
```

Here, `tf.square()` is a built-in function of TensorFlow that will square the difference between `yph`, which is the actual value, and `y_model`, which is the predicted value.

14. Create an optimizer. We will be using `GradientDescentOptimizer`, which will help us find the global minima point, which basically indicates when and where we have to stop the training of the neural network. The code is as follows:

```
| In[14]:  
| optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)  
| train = optimizer.minimize(error)
```

15. If we want to run the TensorFlow code, we need to initialize all the variables using the `tf.global_variables_initializer()` function. The code is as follows:

```
| init = tf.global_variables_initializer()
```

16. Create a session to help us run the complete computation graph:

```
| In[16]:  
| with tf.Session() as sess:  
|  
|     sess.run(init)  
|  
|     batches = 1000  
|  
|     for i in range(batches):
```

```
    rand_ind = np.random.randint(len(x_data), size=batch_size)
    feed = {xph:x_data[rand_ind], yph:y_true[rand_ind]}
    sess.run(train, feed_dict=feed)
model_m, model_b = sess.run([m, b])
```

Within this session, we will pass our input data, which is `x_data`, in the `x_ph` placeholder, and the true output, which is `y_true`, in the `y_ph` placeholder. We will run this for 1,000 epochs. One epoch is basically the combination of one forward and one backward propagation, which will result in updating the weights of the neural network.

Once the preceding code is executed, we can get the coefficients that are indicated by m , and the bias, b , which is the intercept, by using the following code:

```
| In[17]:  
| model_m
```

The output of the coefficient is as follows:

```
| Out[17]:  
| 0.48660713
```

The intercept is present in the `b` variable:

```
| In[18]:  
| model_b
```

The output is as follows:

```
| Out[18]:  
| 4.8790665
```

Once we get the coefficient, which is stored in the `model_m` variable, and the intercept, which is stored in `model_b`, we can plot the graph of the line of best fit. The code for this is as follows:

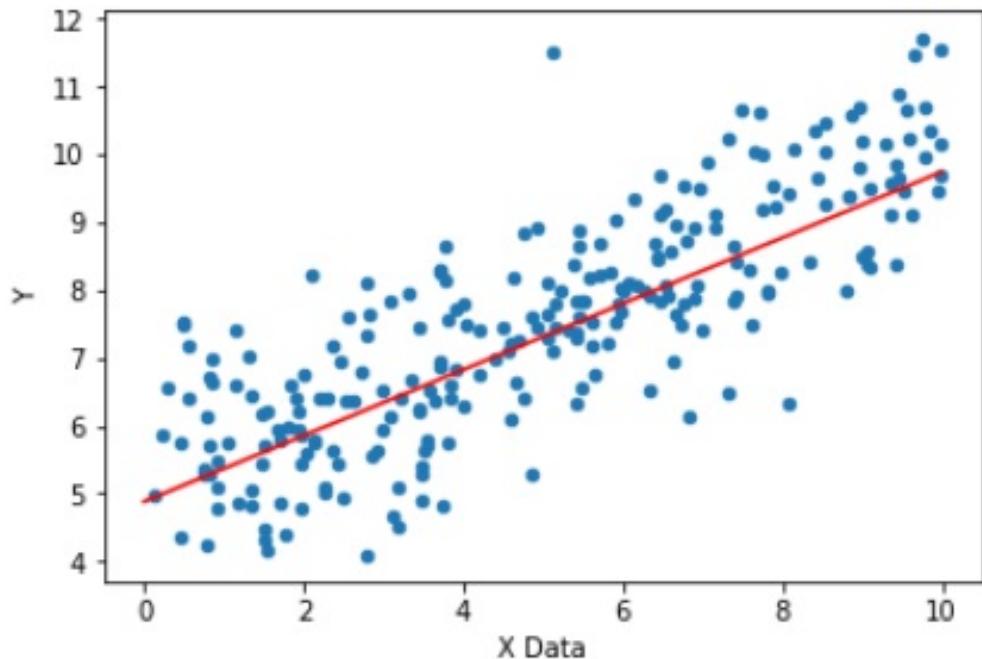
```
| In[19]:  
| y_hat = x_data * model_m + model_b
```

Here, `y_hat` is the value that's predicted by the neural network when we use the coefficient and the bias in the equation. The line of best fit can be plotted using the following code, in which we are using the `matplotlib plot()` function:

```
In[20]:  
my_data.sample(n=250).plot(kind='scatter',x='X Data',y='Y')  
plt.plot(x_data,y_hat,'r')
```

The output is as follows:

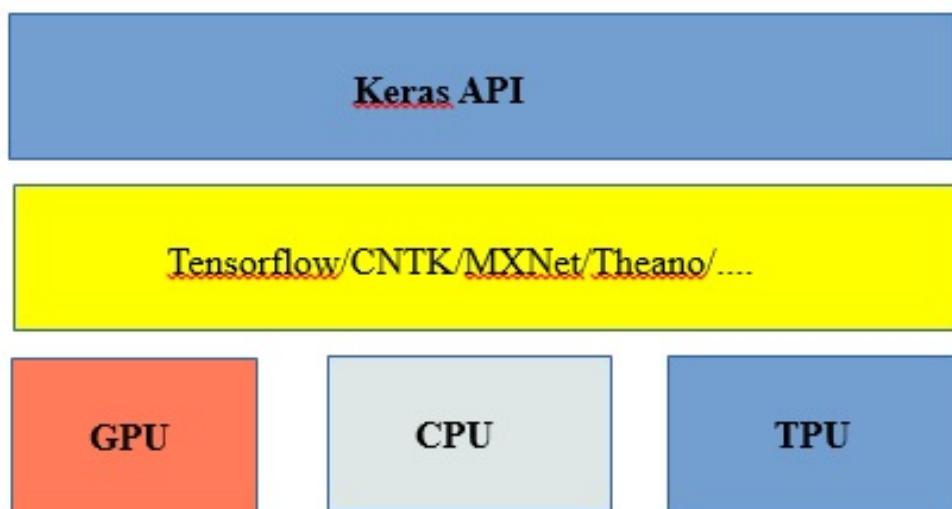
```
Out[20]: [<matplotlib.lines.Line2D at 0x17c78e0dfd0>]
```



The linear line that we can see in the preceding graph is the line of best fit.

An introduction to Keras

Keras is a high-level neural network API that is capable of running on top of TensorFlow, Theano, and CNTK. It enables fast experimentation through a high-level, user-friendly, modular, and extensible API. Keras can also be run on both CPU and GPU. We can say that Keras is a wrapper on top of other powerful libraries, such as TensorFlow, Theano, and CNTK. This structure is shown in the following diagram:



Keras is part of the TensorFlow core, which makes it TensorFlow's preferred high-level API. It provides lots of features that will help to implement deep learning models very quickly and it is really helpful in research and development. Since it is a wrapper on the top of TensorFlow, it provides lots of built-in functions that make it easier to access the TensorFlow APIs.

Keras was developed and maintained by François Chollet, a Google engineer.

API categories in Keras

With the help of the Keras API, we can create neural networks easily. Keras supports different types of API usage. The Keras API internally calls the backend TensorFlow or the Theano API, which helps us create the computation graphs of the neural network architecture. In the following sections, we'll look at the different APIs that are present in Keras.

The sequential model API

The sequential model helps us create a feed-forward neural network model. We can add hidden layers and neurons. It is used to create computation graphs in the form of a neural network architecture. As we are adding different neurons and hidden layers, it provides us with the facility to add an activation function, such as ReLu or sigmoid.

Let's consider a neural network in which we want to create an input layer with 10 inputs, a hidden layer with 20 neurons, and an output layer with two outputs, which is like a binary classification.

To begin, we need to import the Keras library, because all the APIs are present within this library. We will also be importing the layers library present in Keras, which will help us to create the input layer, the hidden layer, and the output layer.

Let's import both the libraries:

```
In[1]:  
import keras  
from keras import layers  
  
Out[1]:  
Using TensorFlow backend.
```

In this step, we will initialize the sequential model since we are creating a feed-forward neural network. The code is shown here:

```
In[2]:  
model = keras.Sequential()
```

In the next step, we will start adding the input layer and the first hidden layer. We will be using the `add()` function to add layers inside the sequential model. To add layers, we will be using the `layer.Dense()` function. The following is the syntax of the `layer.Dense()` function:

```
Init signature:  
layers.Dense(  
    ['units', 'activation=None', 'use_bias=True', "kernel_initializer='glorot_uniform'",  
)
```

The following are the parameters that are used:

```
# Arguments
    units: Positive integer, dimensionality of the output space.
    activation: Activation function to use
        (see [activations](../activations.md)).
        If you don't specify anything, no activation is applied
        (ie. "linear" activation: `a(x) = x`).
    use_bias: Boolean, whether the layer uses a bias vector.
    kernel_initializer: Initializer for the `kernel` weights matrix
        (see [initializers](../initializers.md)).
    bias_initializer: Initializer for the bias vector
        (see [initializers](../initializers.md)).
    kernel_regularizer: Regularizer function applied to
        the `kernel` weights matrix
        (see [regularizer](../regularizers.md)).
    bias_regularizer: Regularizer function applied to the bias vector
        (see [regularizer](../regularizers.md)).
    activity_regularizer: Regularizer function applied to
        the output of the layer (its "activation").
        (see [regularizer](../regularizers.md)).
    kernel_constraint: Constraint function applied to
        the `kernel` weights matrix
        (see [constraints](../constraints.md)).
    bias_constraint: Constraint function applied to the bias vector
        (see [constraints](../constraints.md)).

# Input shape
nD tensor with shape: `batch_size, ..., input_dim`.
The most common situation would be
a 2D input with shape `(batch_size, input_dim)`.

# Output shape
nD tensor with shape: `batch_size, ..., units`.
For instance, for a 2D input with shape `(batch_size, input_dim)`,
the output would have shape `(batch_size, units)`.
```

The code to use the `Dense()` function is as follows:

```
In[3]:
model.add(layers.Dense(10, activation='relu', input_shape=(10,)))
```

In the preceding code, the first parameter is the number of inputs. The `input_shape` parameter defines the shape of the input and is usually a scalar value. The `activation` parameter is used to provide the activation function that's used.

We are considering 10 inputs here, so we add 10 neurons in the first hidden layer. We will apply the ReLu activation function.

Finally, we will add the output layer:

```
In[3]:
## Output Layer
model.add(layers.Dense(2, activation='softmax'))
```

In the preceding code, we added an output layer with two output. The activation function that's used is the sigmoid activation function, since it is a binary classification.

Finally, we can see a summary of the model using the following code:

```
| In[4]:  
| model.summary()
```

The output is as follows:

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 10)	110
dense_2 (Dense)	(None, 2)	22
<hr/>		
Total params: 132		
Trainable params: 132		
Non-trainable params: 0		
<hr/>		

The total parameters, 132, is the number of weights and biases that we are using in this neural network.

The functional API

This is suitable for multi-input, multi-output, and arbitrary static graph topologies:

```
In[1]:  
import keras  
from keras import layers  
  
inputs = keras.Input(shape=(10,))  
x = layers.Dense(20, activation='relu')(x)  
x = layers.Dense(20, activation='relu')(x)  
outputs = layers.Dense(10, activation='softmax')(x)  
  
model = keras.Model(inputs, outputs)  
model.fit(x,y, epochs=10, batch_size=32)
```

Model subclassing API

This helps you to achieve maximum flexibility, allowing you to create your own fully-customizable models by subclassing the model class and implementing your own forward pass in the `call` method:

```
In[1]:  
class CustomModel(keras.Model):  
  
    def __init__(self):  
  
        super(CustomModel, self).__init__()  
        self.dense1 = layers.Dense(20, activation='relu')  
        self.dense2 = layers.Dense(20, activation='relu')  
        self.dense3 = layers.Dense(10, activation='softmax')  
  
    def call(self, inputs):  
        x = self.dense1(x)  
        x = self.dense2(x)  
        return self.dense3(x)  
  
model = CustomModel()  
model.fit(x, y, epochs=10, batch_size=32)
```

Summary

In this chapter, we discussed deep learning and its uses in financial case studies. We also discussed neural networks and looked closely at how they work. We then learned about activation functions, such as sigmoid, tanh, and ReLu. After that, we looked at bias, which is another important parameter, before exploring different types of neural networks, including ANNs, CNNs, and RNNs. We discussed TensorFlow, an open source library that was developed by Google. We used TensorFlow to solve a practical problem in which we implemented a linear regression neural network model. We finished by looking at another library, called Keras, which is a wrapper on top of TensorFlow.

In the next chapter, we will discuss how we can use LSTM RNNs for stock predictions.

Stock Market Analysis and Forecasting Case Study

In this chapter, we will discuss various use cases related to finance using deep learning techniques and the Keras library. In the previous chapters, we focused on implementing use cases with Python, along with statistics using the StatModels library. We will initially go ahead with an in-depth discussion of the **Long Short Term Memory (LSTM)** version of the **Recurrent Neural Network (RNN)**. Part of the RNN has already been covered in previous chapters. In this chapter, we will deep dive into the architecture of the LSTM RNN, which is a variant of RNN. We need to understand this architecture since we will be solving the use cases with these architectures using Keras as a wrapper, and TensorFlow as the backend library.

In this chapter, we will discuss the following topics:

- LSTM RNN
- Predicting and forecasting the stock market price using LSTM – case study 1
- Predicting and forecasting wine sales using the Arima model – case study 2

Technical requirements

In this chapter, we will use Jupyter Notebook for coding purposes. We will also use pandas, NumPy, and matplotlib. Along with these libraries, we will use libraries such as TensorFlow and Keras.

You can refer to the technical requirements of the previous chapter to install TensorFlow and Keras.

The GitHub repository for this chapter can be found at <https://github.com/PacktPublishing/Hands-on-Python-for-Finance/tree/master/Chapter%2011>.

LSTM RNN intuition

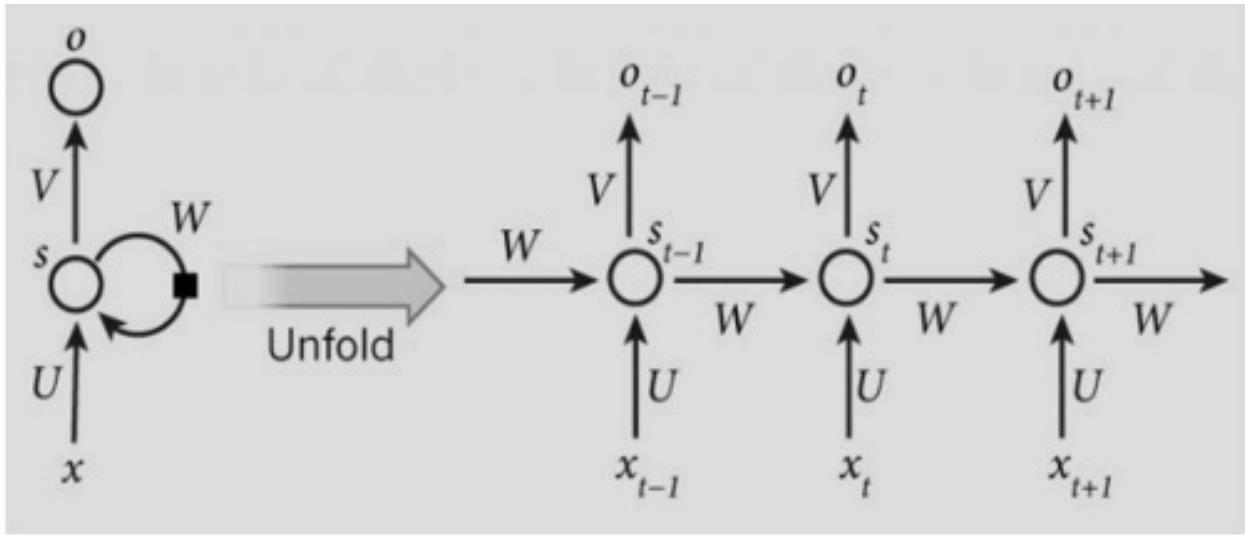
In this section, we will learn about the the most complex and cutting edge areas of the deep learning technique: RNN. In the previous chapter, we covered a basic introduction to RNN. RNN has a very good architecture that helps the neural network work efficiently with financial data. RNN has also become an efficient neural network architecture for use cases involving **natural language processing (NLP)**. Some devices, such as Alexa, Google Assistant, and Apple's Siri, work on NLP concepts, along with this neural network.

When we compare an **Artificial Neural Network (ANN)** to an RNN, we can see that ANNs work with vectors of input data, while the RNN works with the sequence of input data. This is possible because of the memory in the RNN. RNN has been really successful since it is able to handle sequential data efficiently when considering time as an important factor, compared to the ANN, which has no notion of time when it is training with the input data.

This means that an RNN will be able to remember the output of the previous time steps when it is getting trained, and this output is simultaneously fed along with the new input to the RNN. The RNN is able to remember the previous output because of the memory that is part of the architecture present. RNN considers both context units based on what they've seen previously and the current input.

In a traditional feedforward neural network, all the input data and output data is independent of each other. Suppose that in the stock prediction use case, if you want to predict the next day stock, it is always best to know what the previous day's stock prices were. The RNN is defined as recurrent because they do a similar task for every sequence of data, with the output being dependent on the previous output computation. There is also another way to define RNNs, which is that they have a memory that saves information from the previous output calculations.

The following diagram shows the architecture of RNN:



The preceding diagram shows how an RNN is being unfolded into a full network. The left side of the diagram indicates x as our input and o as our output. Furthermore, the s symbol indicates the neurons in the hidden layer. There is a self loop on the hidden neurons that indicates that the output of the previous time step $t-1$ is also provided as an input in the current time step t . For example, if the sequence of the input is the six days of stock opening price data, then the network would unfold itself to six layered neural networks, which refers to one layer for each day's opening stock price. However, usually, with respect to the stock prediction use case, we require many days previous data to predict the next day prediction more accurately. A conventional RNN will not work properly, because it has memory limitations to remember the previous output of the data. Consequently, to overcome this problem, we will use another variant of RNN, which is called LSTM RNN. This tends to outperform conventional RNNs.

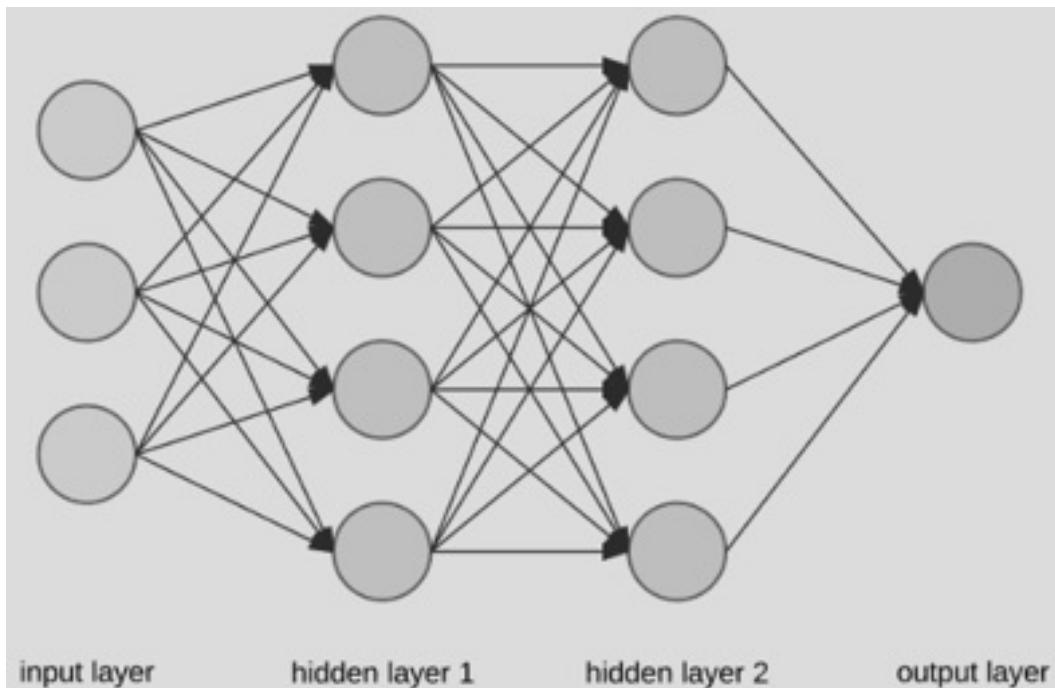
How does the RNN work?

Before looking at the way in which the RNN works, we will first discuss the normal feedforward neural networks that we must be familiar with in order to understand the RNN. However, it is also important to understand what sequential data is.

Sequential data is a kind of ordered data in which there is a sequence, and the previous sequence is related to the next sequence of data. The best example of sequential data is financial data, DNA sequences, or music notations. The most popular type of sequential data is time series data, which is a series of data with respect to time.

Feedforward neural networks

The following diagram shows a feedforward neural network:



In the feedforward neural network, the data moves in only one direction. The information moves from the input layer, passes through all the hidden layers, and finally reaches the output layer. The information moves forward so that it never

touches the nodes again.

These feedforward neural networks have no memory of the information that they received in the previous steps, and so they do not perform well in predicting the next outcome because these neural networks consider only the current input, and have no information regarding the order of the time that it has come in. So, they do not remember anything from the past except their training information.

RNNs

In RNNs, the information or the output cycles through a circular loop, which means that the output of the previous timestamp that the RNN has learned is fed in as the input again. Consequently, the RNN makes decisions based on the current input and the previous input that it received.

An RNN usually has a short-term memory. When it is combined with the LSTM, they usually develop a capacity in their memory. We will discuss LSTM in the upcoming subtopics.

The RNN usually adds the immediate computed past result to the present input. Consequently, the RNN has two inputs: the present and the recent computed past. This is really important, since many sequences of data such as time series data contain very crucial or important information. This is the reason that the RNN (which is a state of the art algorithm) can do things that the other algorithms can't.

A normal feedforward neural network – like all other deep learning techniques – assigns a matrix of weight to its inputs, and then passes through the activation function and gives the output, but with respect to the RNN, they apply weights to the current and previous input. Additionally, they also update the weights for both the inputs through different types of techniques, such as gradient descent and backpropagation, over time.

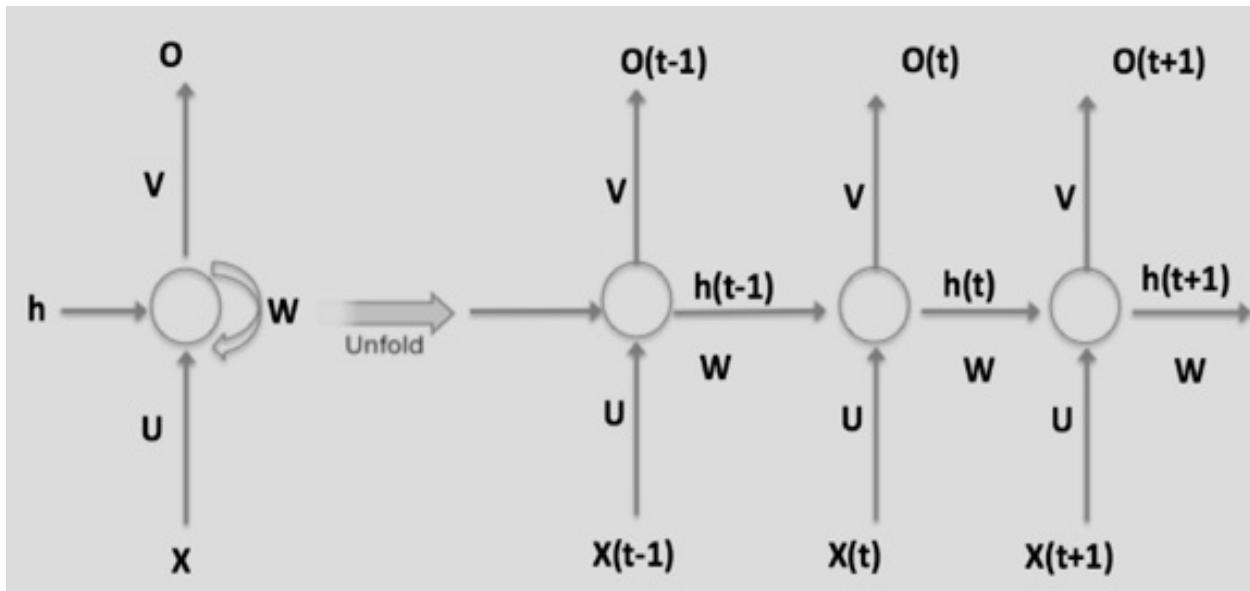
The feedforward neural network usually maps every input to one output, while RNNs have different kinds of mapping mechanisms such as one-to-one, one- to-many, many-to-many (mostly used for translating languages), and many-to-one (classifying a voice).

Backpropagation through time

Backpropagation through time, also known as BPTT, is a technique for performing a backpropagation on an unfolded RNN. Unrolling deals with some of the conceptual and visualization techniques that help us understand what the internal working of the network is. In most deep learning frameworks, whenever we implement an RNN, these frameworks usually take care of the backpropagation. However, we need to understand how this works. This usually helps us debug some of the complex problems that occur during the development process.

When we fold the RNN, we can view the RNN as a sequence of neural networks that are usually trained in a sequential manner.

The following is a diagram representing an unfolded RNN:



On the left-hand side before the equals sign, the representation is an RNN. On the right-hand side, the RNN is unfolded, and we can see that there is no loop or cycle since the information is getting passed from one specific time step to the next one. So, we define an RNN as a sequence of many neural networks.

In the backpropagation through time, it is mandatory to understand the

conceptualization of unrolling, since the error of any given time step usually depends on the previous time step.

In the backpropagation through time, the error is back propagated from the last time step to the first time step in the unfolded RNN. This allows us to compute the error in each and every time step, which, in turn, allows us to update the weights. Usually, in the RNN, the backpropagation through time is an expensive process if our number of time steps is a greater value, or if we have a higher number of time steps.

Problems with standard RNNs

There are two major problems with a standard RNN. These are covered in the following sections.

Vanishing gradient problem in RNN

This is a problem that causes a major difficulty when training an RNN. More specifically, this involves weights in the initial layers of the networks. As we do backpropagation with time, which is moving backward in the unfolded RNN based on the number of time steps, when we calculate the gradient loss or error with respect to weights, the gradients usually become smaller, and they keep getting smaller as we continue moving backward in the RNN. This means that the neurons that are present in the initial layer learn very slowly compared to all the neurons that are present in the final layers. As a result of this, the initial layers in the networks are also trained very slowly.

Why earlier layers of the RNN are important for us

The initial layers of the RNN are the basic building blocks, or layers, for the complete RNN. Consequently, for most of the sequential data, it is always good that the initial layers should be able to distinguish the patterns from the dataset. This is also very important for the financial data that will initially help us to understand the pattern.

What harm does it do to our model?

Usually, due to the vanishing gradient problem, the training time of the model increases while the accuracy of the model decreases.

Exploding gradients in RNN

Exploding gradients is another type of problem that usually occurs in an RNN while, in the gradient, errors are accumulated. This results in a larger update of the weight in the neural network during training.

What are exploding gradients?

Usually, while training an RNN, we will get an error gradient as we perform back propagation over time. In training, these error gradients are usually accumulated with different values that are actually used for updating the weights. Due to the accumulation, it so happens that there is a large update in the weights, which, in turn, creates a very unstable network. This leads to underfitting the RNN so that performance is degraded.

How to fix exploding and vanishing gradients

There are many ways to address or fix the issues of the exploding and vanishing gradients. These include the following:

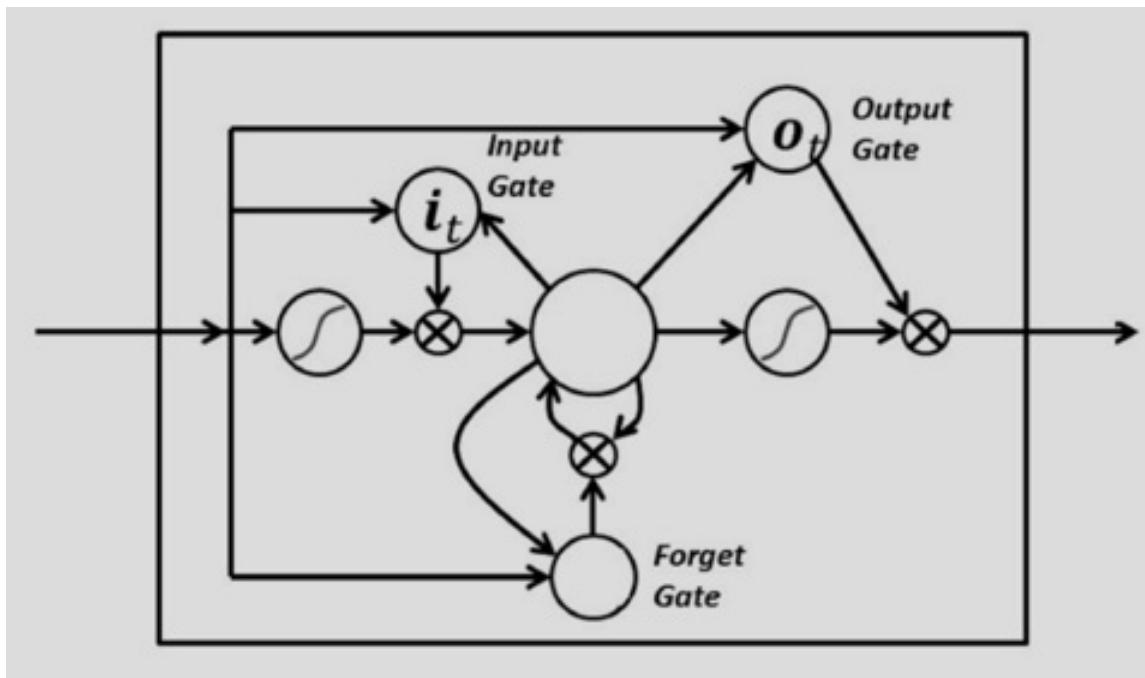
- Redesigning the RNN so that it has fewer layers based on time stamps
- Use of LSTM along with the RNN

LSTM

LSTM is an added extension in the RNN that makes it possible to increase the RNN's memory. Consequently, LSTM can be very useful when we are working with financial data, such as time series data, since the time series data is data with sequences, along with very long time lags in-between.

LSTMs are usually integrated with the RNN in each and every layer, which is also called the LSTM network. These LSTMs enable the RNNs to remember the input and the output for a long time due to the memory. This memory is similar to the memory of any computer, and these LSTMs can write, delete, and read information to and from its memory.

The following diagram shows the architecture of the LSTM:



The LSTM usually consists of three gates and a cell: the input (i_t), output (o_t), and forget gates. The input gate is used to determine whether or not we need to let the input data in, the forget gate is useful to delete the information if it is not required, and the output gate is responsible for handling output data.

The cell is responsible for remembering data over particular intervals of time. Consequently, all three of these gates help us in regulating the information flow in and out of the cell.

All of these gates in the LSTM RNN perform the operation of the sigmoid that is transforming the values in a range from zero to one. The issue of the vanishing gradient is solved using the LSTM because it is responsible for keeping the gradient values steep enough, which means that the training time is relatively shorter and the accuracy is quite high.

Use case to predict stock prices using LSTM

In this section, we will predict the future stock prices for a specific company stock data, and then compare them with the test data (which is our future data) to see if we are getting good accuracy.

For this, we will be considering Google stock prices – we will try to predict the future stock price and then compare the prediction. In this use case, we will be using LSTM RNN using the Keras open source wrapper with the TensorFlow library in the backend.

So, to begin with, we will import the libraries, as shown here:

```
In[1]:  
import numpy as np  
import matplotlib.pyplot as plt  
import pandas as pd
```

We will be using two CSV files, where one CSV file is the training dataset named `Google_Stock_Train.csv` and the other one is the test dataset named `Google_Stock_Test.csv`.

Data preprocessing of the Google stock data

Let's go ahead and import the training dataset, which is the `Google_Stock_Train.csv` file, using the `pandas` library, as shown in the following code snippet:

```
In[2]:  
# Importing the training set  
dataset_train = pd.read_csv('Google_Stock_Price_Train.csv')
```

We can see the top five records by using the `head()` function:

```
In[3]:  
# Check the first 5 records  
dataset_train.head()
```

The output is as follows:

	Date	Open	High	Low	Close	Volume
0	1/3/2012	325.25	332.83	324.97	663.59	7,380,500
1	1/4/2012	331.27	333.87	329.08	666.45	5,749,400
2	1/5/2012	329.83	330.75	326.89	657.21	6,590,300
3	1/6/2012	328.34	328.77	323.68	648.24	5,405,900
4	1/9/2012	322.04	322.29	309.46	620.76	11,688,800

We can use the following code to see the last five records:

```
In[4]:  
# Check the last 5 records  
dataset_train.tail()
```

The output is as follows:

	Date	Open	High	Low	Close	Volume
1253	12/23/2016	790.90	792.74	787.28	789.91	623,400
1254	12/27/2016	790.68	797.86	787.66	791.55	789,100
1255	12/28/2016	793.70	794.23	783.20	785.05	1,153,800
1256	12/29/2016	783.33	785.93	778.92	782.79	744,300
1257	12/30/2016	782.75	782.78	770.41	771.82	1,770,000

Consequently, the training dataset of the Google stock prices are from January 3, 2012 to December 30, 2016. From all present columns, we will consider the `open` column and base the future prediction on that. We can also consider any column and do the prediction.

Let's pick up the column for which we need to make the prediction. We can use the `iloc` operation that present in pandas, and we will also convert the data into arrays by using a `values` operation, as shown here:

```
In[5]:
# Pick up the Open Column
training_set = dataset_train.iloc[:, 1:2].values
```

The next step is to perform feature scaling. This is a process in which we will try to transform the dataset that is measured in various units, ranges, and magnitudes. Usually, we can perform two different types of feature scaling:

- **MinMax scaling:** This scaling helps us to transform the data between zero and one, which is also called uniform distribution. The MinMax scaling formula is as follows:

$$z = \frac{x - \min(x)}{\max(x) - \min(x)}$$

- **Standard scaling:** This scaling mechanism is used for transforming our dataset according to the standard normal distribution.

Before implementing this project, we should also know about another special

library, called **scikit learn**, also known **sklearn**. Sklearn is an open source, free machine learning library available for Python programming. These libraries have various features and sublibraries for data preprocessing, feature engineering, and machine learning algorithms. To learn more about this, you can refer to <https://scikit-learn.org/>.

For the current example, we will be implementing MinMax scaling. Initially, we will import the `MinMaxScaler` library from the `sklearn` open source library. The code is as follows:

```
| In[6]:  
| # Feature Scaling  
| from sklearn.preprocessing import MinMaxScaler
```

After importing the library, we will transform our complete training dataset using the MinMax scaler, which will transform between `0` and `1`. The code is as follows:

```
| In[7]:  
| sc = MinMaxScaler(feature_range = (0, 1))  
| training_set_scaled = sc.fit_transform(training_set)
```

In the preceding code, we first need to initialize an object using the MinMax scaler and we need to provide the range, which is from `0` to `1` in the `feature_range` parameter.

After creating the object variable of the MinMax scaler, which has the name `sc`, we need to apply the `fit_transform()` method to apply the MinMax scaler on all of these datasets, and assign a variable named `training_set_scaled`, which contains our scaled data.

After applying the MinMax scaler functionality, we can see the dataset by using the following code:

```
| In[9]:  
| training_set_scaled
```

The output is as follows:

```
array([[ 0.08581368],
       [ 0.09701243],
       [ 0.09433366],
       ...,
       [ 0.95725128],
       [ 0.93796041],
       [ 0.93688146]])
```

Consequently, we have scaled down the open column values using the MinMax scaler. In the next step, we will divide our dataset into independent and dependent features.

This step is the most important while we are doing data preprocessing, because this is where we have to create a dataset with some time step data, which is our historical data, and one piece of output data. We will consider the time steps as being equal to 100 time steps. Here, 100 time steps is equal to the previous 100 days stock prices.

So, first let me explain what the previous paragraph means. The 100 time steps means that for each time t , the RNN is going to look at the 100 stock prices before time t , which is the stock price between 100 days before time $t-100$ and t . Based on the trends it is capturing during the previous 100 time steps, historical data will try to predict the next output. Consequently, 100 time steps, represent the historical data from the previous 100 dates in the dataset from which our RNN is going to learn and understand some correlation and trends. Based on the 100 time steps. the model is going to predict the next output, which is the stock price at $t+1$. 100 is the timestamp that we are experimenting with, but we can try different time step values too. However, we should select a high value for the time steps. Suppose we choose one time step; the RNN will lead to overfitting. I have even tried 20 timestamps or 30 timestamps, but the model did not give an accurate result for this.

As we know, there are 20 financial days in a month, so 100 time steps corresponds to five months. This means that each day, our RNN is going to look at the previous five months and try to predict the stock price of the next day.

We will restructure our training data so that we have 100 time steps and one output, which will be the stock price at time $t+1$.

Let's look at the coding part to see how we can restructure the input to the data structure. Initially, we will create two variables that indicate the independent and dependent variables, as follows:

```
| In[10]:  
| X_train = []  
| y_train = []
```

The independent variable is defined as `x_train`, and the dependent variable is defined as `y_train`. The following code will help us to create a data structure with 100 time steps and one output. The 100 time steps or the input data will be stored in `x_train`, and the next day output will be stored in `y_train`:

```
| In[11]:  
| for i in range(100, 1258):  
|     X_train.append(training_set_scaled[i-100:i, 0])  
|     y_train.append(training_set_scaled[i, 0])  
| X_train, y_train = np.array(X_train), np.array(y_train)
```

The following code helps us see the shape of the `x_train` dataset:

```
| In[12]:  
| X_train.shape
```

The output is as follows:

```
| Out[12]:  
| (1158, 100)
```

The preceding shape indicates that the independent feature has 1,158 rows and 100 columns. The 100 columns indicate the 100 time steps, or the 100 previous days stock price.

The following code helps us to see the shape of the `y_train` data:

```
| In[13]:  
| y_train.shape
```

The output is as follows:

```
| Out[13]:  
| (1158, )
```

The `y_train` data has the output of the next day with respect to the independent features.

The last step of data preprocessing includes reshaping the input data into a three-dimensional array. The reason we do this is because the structure of the RNN accepts only three-dimensional data. The following code helps us to reshape the data from a two-dimensional shape into a three-dimensional shape:

```
| In[14]:  
| X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
```

The preceding code helps us to reshape the `x_train` data from two-dimensions into three-dimensions with shape (*number of records, number of time steps, 1*).

The following code now helps us to see the shape of the `x_train` data:

```
| In[15]:  
| X_train.shape
```

The output is as follows:

```
| Out[15]:  
| (1158, 100, 1)
```

Building the LSTM RNN

In this section, we will be building the LSTM RNN using the Keras wrapper and the backend library that will be used is TensorFlow. Here, we will create a LSTM RNN with four hidden layers and an output layer.

To begin with, we need to import some of the Keras libraries. The code is as follows:

```
| In[16]:  
| # Importing the Keras libraries and packages  
| from keras.models import Sequential  
| from keras.layers import Dense  
| from keras.layers import LSTM  
| from keras.layers import Dropout
```

The output is as follows:

```
| Using TensorFlow backend.
```

In the preceding code, we imported the `Sequential` library. The `Sequential` library helps us to create a feedforward neural network. The `Dense` library helps us to create neurons in the hidden layers. The `LSTM` library helps us to add the LSTM extensions in the RNN. `Dropout` is a library that will help us to deactivate some of the neurons in the RNN.

First, we need to initialize the RNN. The code for this is as follows:

```
| In[17]:  
| # Initializing the RNN  
| regressor = Sequential()
```

In the following steps, we need to add LSTM layers and dropout regularization.

The LSTM layer code syntax is as follows:

The screenshot shows a Jupyter Notebook cell with the following code:

```
LSTM()  
Init signature:  
LSTM(  
    ['units', "activation='tanh'", "recurrent_activation='hard_sigmoid'", 'use_bias=True'  
, "kernel_initializer='glorot_uniform'", "recurrent_initializer='orthogonal'", "bias_init
```

The following are the most common parameters for constructing the neural network:

- `units`: The number of neurons in the hidden layer
- `activation`: The activation function to be used, such as ReLu, sigmoid, or tanh
- `input_shape`: This is the input shape to be provided to the LSTM RNN

The following code helps us to add a LSTM layer to the neural network:

```
In[18]:  
# Adding the first LSTM layer and some Dropout regularisation  
regressor.add(LSTM(units = 50, return_sequences = True, input_shape = (X_train.shape[1],  
regressor.add(Dropout(0.2))
```

In the preceding regressor model, we are adding the LSTM layer. In this LSTM layer, we have added 50 neurons, the activation function is ReLu, and we have provided the input shape of the input data. We have also added a dropout value of 20%.

Similarly, we will add one more layer of LSTM RNN and add a dropout value of 20% again. The code for this is as follows:

```
In[19]:  
# Adding a second LSTM layer and some Dropout regularisation  
regressor.add(LSTM(units = 50, return_sequences = True))  
regressor.add(Dropout(0.2))
```

In the second layer, we do not have to provide the `input_shape`, since the input has to be provided to the first layer only. `return_sequences` is also set to `True`. `return_sequences`, if set to `True`, will help us to access the hidden state output for every layer in which it is set.

Similarly, we will be adding two more layers of LSTM in the RNN. The code for this is as follows:

```
| In[20]:  
| # Adding a third LSTM layer and some Dropout regularisation  
| regressor.add(LSTM(units = 50, return_sequences = True))  
| regressor.add(Dropout(0.2))
```

The fourth layer is added using the following code:

```
| In[21]:  
| # Adding a fourth LSTM layer and some Dropout regularisation  
| regressor.add(LSTM(units = 50))  
| regressor.add(Dropout(0.2))
```

Finally, we add the output layer. The code for this is as follows:

```
| In[22]:  
| # Adding the output layer  
| regressor.add(Dense(units = 1))
```

Now, we can see the summary of the model by using the following code:

```
| In[23]:  
| regressor.summary()
```

The output is as follows:

Layer (type)	Output Shape	Param #
<hr/>		
lstm_1 (LSTM)	(None, 100, 50)	10400
dropout_1 (Dropout)	(None, 100, 50)	0
lstm_2 (LSTM)	(None, 100, 50)	20200
dropout_2 (Dropout)	(None, 100, 50)	0
lstm_3 (LSTM)	(None, 100, 50)	20200
dropout_3 (Dropout)	(None, 100, 50)	0
lstm_4 (LSTM)	(None, 50)	20200
dropout_4 (Dropout)	(None, 50)	0
dense_1 (Dense)	(None, 1)	51
<hr/>		
Total params:	71,051	
Trainable params:	71,051	
Non-trainable params:	0	

From the preceding summary of the model, we can see a four-layer LSTM RNN. The output layer has one neuron.

The total parameter implies all the weights and biases that are initialized for the RNN.

Compiling the LSTM RNN

In this section, we will select the optimizer process, along with the training process and the number of epochs to be used with the batch size. The batch size indicates how much input data we will be passing to the RNN at a time during training.

The following code helps in selecting the optimizer and the loss or cost function to be used while training the LSTM RNN:

```
| In[26]:  
| # Compiling the RNN  
| regressor.compile(optimizer = 'adam', loss = 'mean_squared_error')
```

Here, we are selecting `optimizer` as `adam` and `loss` as `mean_squared_error`.

The Adam optimizer's functionality is given here:

- **Adam optimizer:** The Adam optimizer is an optimization algorithm that can be used instead of gradient descent or stochastic gradient descent for updating the weights in the neural network.

There are many advantages to implementing the Adam optimizer:

- It is very straightforward and easy to implement
 - It is computationally fast and efficient
 - It requires less memory
 - Hyperparameters require less tuning
- **Mean squared error:** The mean squared error is the loss or the cost function. As we train the LSTM RNN, we need to reduce the loss unless we meet the global minima point, which is found out by the optimizer.

Mathematically, this is given by the following equation:

$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

Here, y_i is the actual value, and the other \hat{y}_i parameter is the predicted value.

The final step is to compile the RNN and start training the LSTM RNN by providing the inputs that fit the RNN to the training set:

```
In[27]:  
# Fitting the RNN to the Training set  
regressor.fit(X_train, y_train, epochs = 100, batch_size = 32)
```

Here, we are running for 100 epochs, and it will take some time to train the network. The RNN will be trained on the `x_train` data, which is our input, and the `y_train` data, which is our output.

The output of the training is as follows:

Epoch 1/100
1158/1158 [=====] - 16s 14ms/step - loss: 0.0530
Epoch 2/100
1158/1158 [=====] - 10s 8ms/step - loss: 0.0060
Epoch 3/100
1158/1158 [=====] - 10s 9ms/step - loss: 0.0061
Epoch 4/100
1158/1158 [=====] - 10s 9ms/step - loss: 0.0052
Epoch 5/100
1158/1158 [=====] - 10s 9ms/step - loss: 0.0051
Epoch 6/100
1158/1158 [=====] - 10s 9ms/step - loss: 0.0052
Epoch 7/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0046
Epoch 8/100
1158/1158 [=====] - 8s 7ms/step - loss: 0.0047
Epoch 9/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0044
Epoch 10/100
1158/1158 [=====] - 10s 9ms/step - loss: 0.0046
Epoch 11/100
1158/1158 [=====] - 10s 9ms/step - loss: 0.0040
Epoch 12/100
1158/1158 [=====] - 11s 9ms/step - loss: 0.0043
Epoch 13/100
1158/1158 [=====] - 11s 9ms/step - loss: 0.0037
Epoch 14/100
1158/1158 [=====] - 11s 9ms/step - loss: 0.0043
Epoch 15/100
1158/1158 [=====] - 11s 9ms/step - loss: 0.0042
Epoch 16/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0043
Epoch 17/100
1158/1158 [=====] - 11s 9ms/step - loss: 0.0046
Epoch 18/100
1158/1158 [=====] - 10s 9ms/step - loss: 0.0037
Epoch 19/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0041
Epoch 20/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0040
Epoch 21/100

The continued output is as follows:

```
Epoch 86/100
1158/1158 [=====] - 10s 8ms/step - loss: 0.0015
Epoch 87/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0016
Epoch 88/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0015
Epoch 89/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0016
Epoch 90/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0016
Epoch 91/100
1158/1158 [=====] - 10s 8ms/step - loss: 0.0017
Epoch 92/100
1158/1158 [=====] - 10s 8ms/step - loss: 0.0014
Epoch 93/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0016
Epoch 94/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0015
Epoch 95/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0014
Epoch 96/100
1158/1158 [=====] - 9s 7ms/step - loss: 0.0016
Epoch 97/100

1158/1158 [=====] - 9s 8ms/step - loss: 0.0015
Epoch 98/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0015
Epoch 99/100
1158/1158 [=====] - 8s 7ms/step - loss: 0.0015
Epoch 100/100
1158/1158 [=====] - 9s 8ms/step - loss: 0.0015

<keras.callbacks.History at 0x1efae01e6d8>
```

After 100 epochs, the loss or the cost function is minimized to 0.0015, which is a

very small value.

Making predictions and visualizing data

To make the prediction for the future stock prices, we will require the test data. For this, we have the `Google_Stock_Price_Test.csv` file in the GitHub repository.

In the first step for making the predictions, we will read the `Google_Stock_Price_Test.csv` file that contains the test data. The code for this is as follows:

```
| In[28]:  
| # Part 3 - Making the predictions and visualising the results  
|  
| # Getting the real stock price of 2017  
| dataset_test = pd.read_csv('Google_Stock_Price_Test.csv')
```

The top five records of the `dataset_test` can be seen by using the following code:

```
| In[29]:  
| dataset_test.head()
```

The output is as follows:

	Date	Open	High	Low	Close	Volume
0	1/3/2017	778.81	789.63	775.80	786.14	1,657,300
1	1/4/2017	788.36	791.34	783.16	786.90	1,073,000
2	1/5/2017	786.08	794.48	785.02	794.02	1,335,200
3	1/6/2017	795.26	807.90	792.20	806.15	1,640,200
4	1/9/2017	806.40	809.97	802.83	806.65	1,272,400

The next step is to consider the `open` column, which has the real test data, so that we can compare this data with the predicted data and convert it into arrays. We

will follow all the data preprocessing steps that we have already discussed:

```
| In[30]:  
| real_stock_price = dataset_test.iloc[:, 1:2].values
```

As discussed in the data preprocessing steps, we need to create the same data structures that we created for the training dataset. Here, we need to consider 100 time steps, which act as the input to the LSTM RNN:

```
| In[36]:  
| # Getting the predicted stock price of 2017  
| dataset_total = pd.concat((dataset_train['Open'], dataset_test['Open']), axis = 0)  
| inputs = dataset_total[len(dataset_total) - len(dataset_test) - 100: ].values  
| inputs = inputs.reshape(-1,1)  
| inputs = sc.transform(inputs)  
| X_test = []
```

The transform function is also applied on the test data based on the MinMax scalar. The following code helps us to see the shape of the input data:

```
| In[37]:  
| inputs.shape
```

The output is as follows:

```
| Out[37]:  
| (142, 1)
```

In the next step, we will create the data structures with 100 inputs or features where we are performing the same data preprocessing for creating the test data:

```
| In[38]:  
| for i in range(100, 142):  
|     X_test.append(inputs[i-100:i, 0])
```

Then, we will convert the `X_test` data into arrays by using the following code:

```
| In[39]:  
| X_test = np.array(X_test)
```

After converting it into an array, we need to convert the test data into three-dimensions, since the input that's required by the LSTM is in that format:

```
| In[40]:  
| X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
```

In the next step, we will predict the stock price for the test data. The code is as follows:

```
In[41]:  
predicted_stock_price = regressor.predict(X_test)
```

We can see that the predicted stock price is as follows:

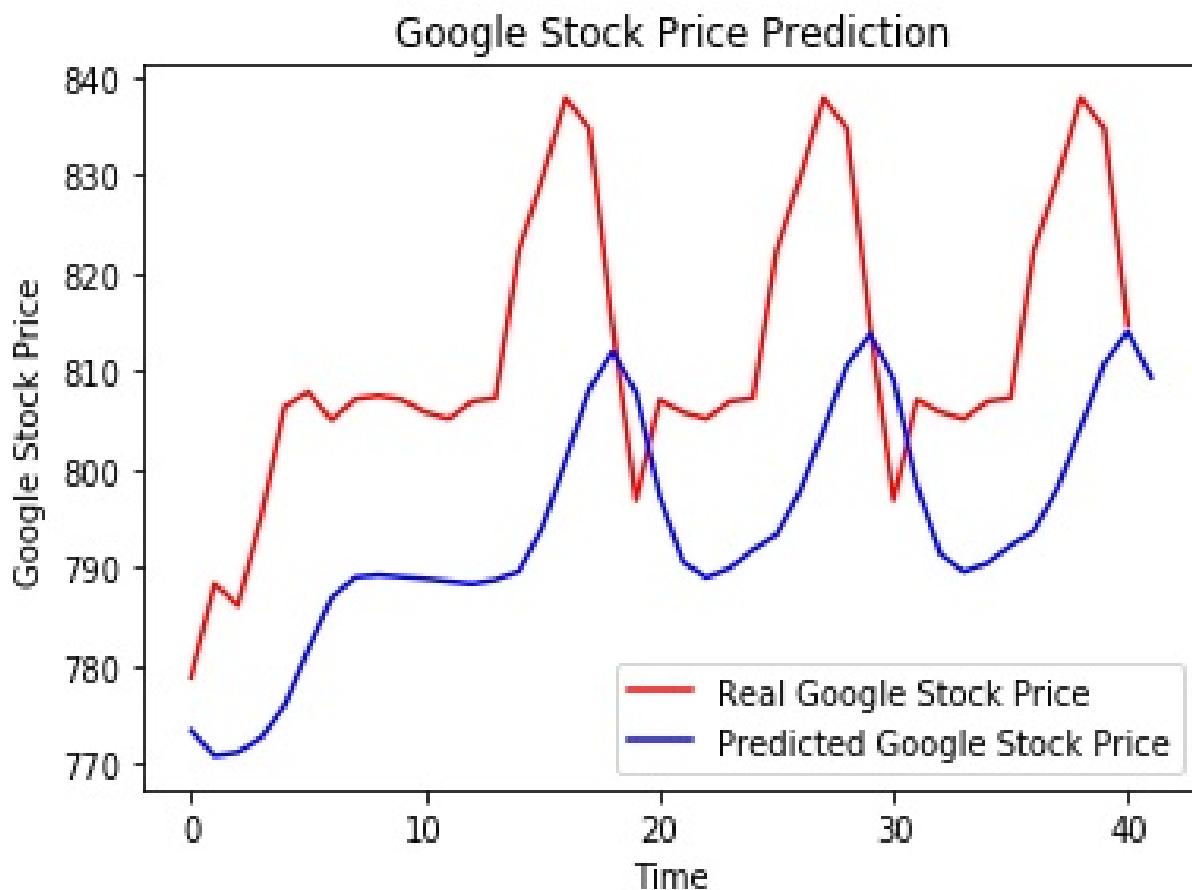
```
In[42]:  
array([[ 773.42578125],  
       [ 770.80810547],  
       [ 771.17224121],  
       [ 772.69415283],  
       [ 775.96777344],  
       [ 781.67315674],  
       [ 786.90692139],  
       [ 788.98205566],  
       [ 789.23120117],  
       [ 789.01574707],  
       [ 788.83050537],  
       [ 788.56585693],  
       [ 788.30200195],  
       [ 788.71746826],  
       [ 789.59545898],  
       [ 794.08117676],  
       [ 800.89807129],  
       [ 808.12402344],  
       [ 811.980896 ],  
       [ 807.83276367],  
       [ 797.33496094],  
       [ 790.59143066],  
       [ 788.92419434],  
       [ 789.91717529],  
       [ 791.81243896],  
       [ 793.37329102],  
       [ 797.74523926],  
       [ 803.99761963],  
       [ 810.53765869],  
       [ 813.78057861],  
       [ 809.16418457],  
       [ 798.33984375],  
       [ 791.37194824],  
       [ 789.54937744],  
       [ 790.43395996],  
       [ 792.25140381],  
       [ 793.7532959 ],  
       [ 798.07678223],  
       [ 804.28723145],  
       [ 810.79046631],  
       [ 814.00152588],  
       [ 809.35919189]], dtype=float32)
```

Finally, we will visualize the predicted values and the actual stock price by using the matplotlib library. The code is as follows:

```
In[43]:  
# Visualising the results  
plt.plot(real_stock_price, color = 'red', label = 'Real Google Stock Price')  
plt.plot(predicted_stock_price, color = 'blue', label = 'Predicted Google Stock Price')  
plt.title('Google Stock Price Prediction')  
plt.xlabel('Time')  
plt.ylabel('Google Stock Price')  
plt.legend()
```

```
| plt.show()
```

In the preceding code, we are plotting the predicted value and the actual values, and then we can compare the predicted results. The output is as follows:



The preceding output indicates the real Google stock price and the predicted Google stock price. From the prediction, we can see that this is quite accurate, and minimal errors.

Predicting wine sales using the ARIMA model

Since we have already discussed the ARIMA model in [Chapter 3, Time Series Analysis and Forecasting](#), in this section, we will be solving another use case, which is predicting the sale of wine. The dataset file we have is of the wine sales dataset.

Initially, we begin by importing all the libraries, as follows:

```
| In[1]:  
| import numpy as np  
| import pandas as pd  
| import matplotlib.pyplot as plt  
| %matplotlib inline
```

After importing the libraries, we will be reading the dataset using pandas. The dataset name is `perrin-freres-monthly-champagne-.csv`. The following is code for reading dataset:

```
| In[2]:  
| df=pd.read_csv('perrin-freres-monthly-champagne-.csv')
```

The following code helps us see the top five records:

```
| In[3]:  
| df.head()
```

The output is as follows:

Month Perrin Freres monthly champagne sales millions ?64-?72

0	1964-01	2815.0
1	1964-02	2672.0
2	1964-03	2755.0
3	1964-04	2721.0
4	1964-05	2946.0

The following code helps us see the last five records:

```
| In[4]:  
df.tail()
```

The output is as shown here:

	Month	Perrin Freres monthly champagne sales millions ?64-?72
102	1972-07	4298.0
103	1972-08	1413.0
104	1972-09	5877.0
105	NaN	NaN
106	Perrin Freres monthly champagne sales millions...	NaN

Consequently, we can see that we have the dataset from January 1964 to September 1972. The first thing we will do is carry out some data preprocessing. We will remove the last row, as this row does not have the correct data. For this, we will apply the drop function that we discussed in [chapter 3, Time Series Analysis and Forecasting](#). The code for this is as follows:

```
| In[5]:  
df.drop(106, axis=0, inplace=True)  
df.drop(105, axis=0, inplace=True)
```

We have set the `inplace` value as `True`, as we want these changes to happen permanently. We will check the last five records again to see if the row has been deleted or not. The code is as follows:

```
| In[6]:  
| df.tail()
```

The output is as follows:

	Month	Perrin Freres monthly champagne sales millions ?64-?72
100	1972-05	4618.0
101	1972-06	5312.0
102	1972-07	4298.0
103	1972-08	1413.0
104	1972-09	5877.0

We will also change the column name to interpret it in a simpler way. The code is as follows:

```
| In[9]:  
| df.columns=['Month', 'Sales per month' ]
```

Let's look at the top five records to see whether or not the column names have been changed:

```
| In[10]:  
| df.head()
```

The output is as follows:

	Month	Sales per month
0	1964-01	2815.0
1	1964-02	2672.0
2	1964-03	2755.0
3	1964-04	2721.0
4	1964-05	2946.0

The general process for the ARIMA model that's used for forecasting is as follows:

1. The first step is to visualize the time series data to discover the trends and find out whether or not the data is seasonal.
2. As we already know, to apply the ARIMA model, we need to use stationary data. The second step, therefore, is to convert the non-stationary data into stationary data using the Dicky-Fuller Test.
3. We then select the p and q values for $ARIMA(p,i,q)$ using **Auto Correlation Function (ACF)** and the **Partial Auto Correlation Function (PACF)**.
4. The next step is to construct the ARIMA model.
5. Finally, we use the model for the prediction.

First, we need to convert the `Month` column into a datetime object using pandas. After that, we will set this `Month` column as the index of the dataframe, as follows:

```
| In[11]:  
| df['Month']=pd.to_datetime(df['Month'])
```

Then, we set the `Month` column as the dataset index:

```
| In[12]:  
| df.set_index('Month',inplace=True)
```

We can see the top five records, as follows:

```
| In[13]:  
| df.head()
```

The output is as follows:

Sales per month	
Month	Sales
1964-01-01	2815.0
1964-02-01	2672.0
1964-03-01	2755.0
1964-04-01	2721.0
1964-05-01	2946.0

The next step is to find out whether or not the dataset is stationary. To do this, we will define a method to check whether the time series data is stationary or not using the Dickey-Fuller library, which is called `adfuller`. This is present in the `statsmodels` library. Based on the p value that's returned, we will decide whether the data is stationary or not. First, we will import the `adfuller` library or class and create a method, as demonstrated in the following code snippet:

```
In[15]:  
from statsmodels.tsa.stattools import adfuller  
# Store in a function for later use!  
def adf_check(time_series):  
  
    result = adfuller(time_series)  
    print('Augmented Dickey-Fuller Test:')  
    labels = ['ADF Test Statistic','p-value','#Lags Used','Number of Observations Used']  
  
    for value,label in zip(result,labels):  
        print(label+ ' : '+str(value) )  
        if result[1] <= 0.05:  
            print("strong evidence against the null hypothesis, reject the null hypothesis")  
        else:  
            print("weak evidence against null hypothesis, time series has a unit root, i")
```

Once we have created the method, we can pass the dataframe with sales per month to the method, and see whether or not the data is stationary, as follows:

```
In[16]:  
adf_check(df['Sales per month'])
```

The output is as follows:

```
Augmented Dickey-Fuller Test:  
ADF Test Statistic : -1.83359305633  
weak evidence against null hypothesis, time series has a unit root, indicating it is non  
p-value : 0.36391577166  
weak evidence against null hypothesis, time series has a unit root, indicating it is non  
#Lags Used : 11  
weak evidence against null hypothesis, time series has a unit root, indicating it is non  
Number of Observations Used : 93  
weak evidence against null hypothesis, time series has a unit root, indicating it is non
```

The p value is not less than 0.05, so this dataset is not a stationary dataset.

The next step is to convert the data into stationary data, which is done by differencing, which we discussed in [Chapter 3, Time Series Analysis and Forecasting](#). The code is as follows:

```
In[17]:  
df['Sales per Month First Difference'] = df['Sales per month'] - df['Sales per month'].s
```

We will pass the Sales per Month First Difference dataset to the adfuller check again to see whether the dataset is stationary or not. The code is as follows:

```
In[18]  
adf_check(df['Sales per Month First Difference'].dropna())
```

The output is as follows:

```
Out[18]:  
Augmented Dickey-Fuller Test:  
ADF Test Statistic : -7.18989644805  
strong evidence against the null hypothesis, reject the null hypothesis. Data has no uni  
p-value : 2.51962044739e-10  
strong evidence against the null hypothesis, reject the null hypothesis. Data has no uni  
#Lags Used : 11  
strong evidence against the null hypothesis, reject the null hypothesis. Data has no uni  
Number of Observations Used : 92  
strong evidence against the null hypothesis, reject the null hypothesis. Data has no uni
```

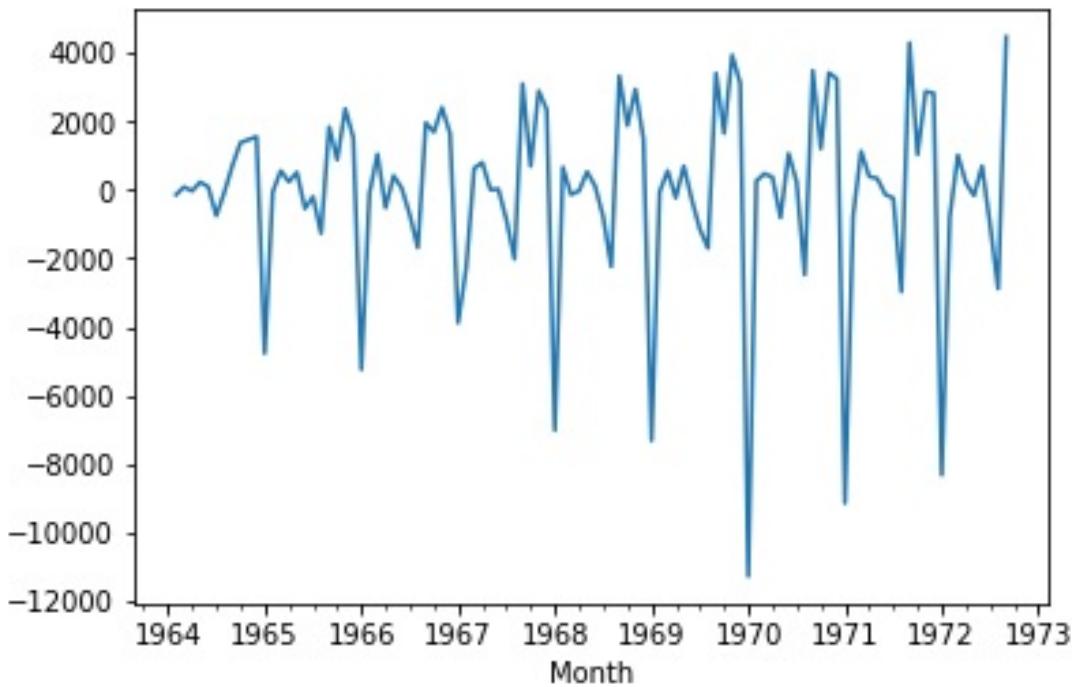
Now, we can see that the value of p is less than 0.05. At this point, we can consider the dataset to be stationary.

We can also plot and see whether the dataset is stationary visually by viewing the stationary graph. The code for this is as follows:

```
In[19]:  
df['Sales per Month First Difference'].plot()
```

The output is as follows:

```
<matplotlib.axes._subplots.AxesSubplot at 0x26ba40613c8>
```



We will now go ahead and apply the `sarimax` model, which is usually applied for seasonal data. The code for this is as follows:

```
In[20]:  
model=sm.tsa.statespace.SARIMAX(df['Sales per month'],order=(1, 1, 1),seasonal_order=(1,  
results=model.fit()
```

Note that we need to set the values of p , q , and d by using autocorrelation and partial autocorrelation, which we discussed in [Chapter 3, Time Series Analysis and Forecasting](#).

We will now complete an exercise in which we will find the value of p .

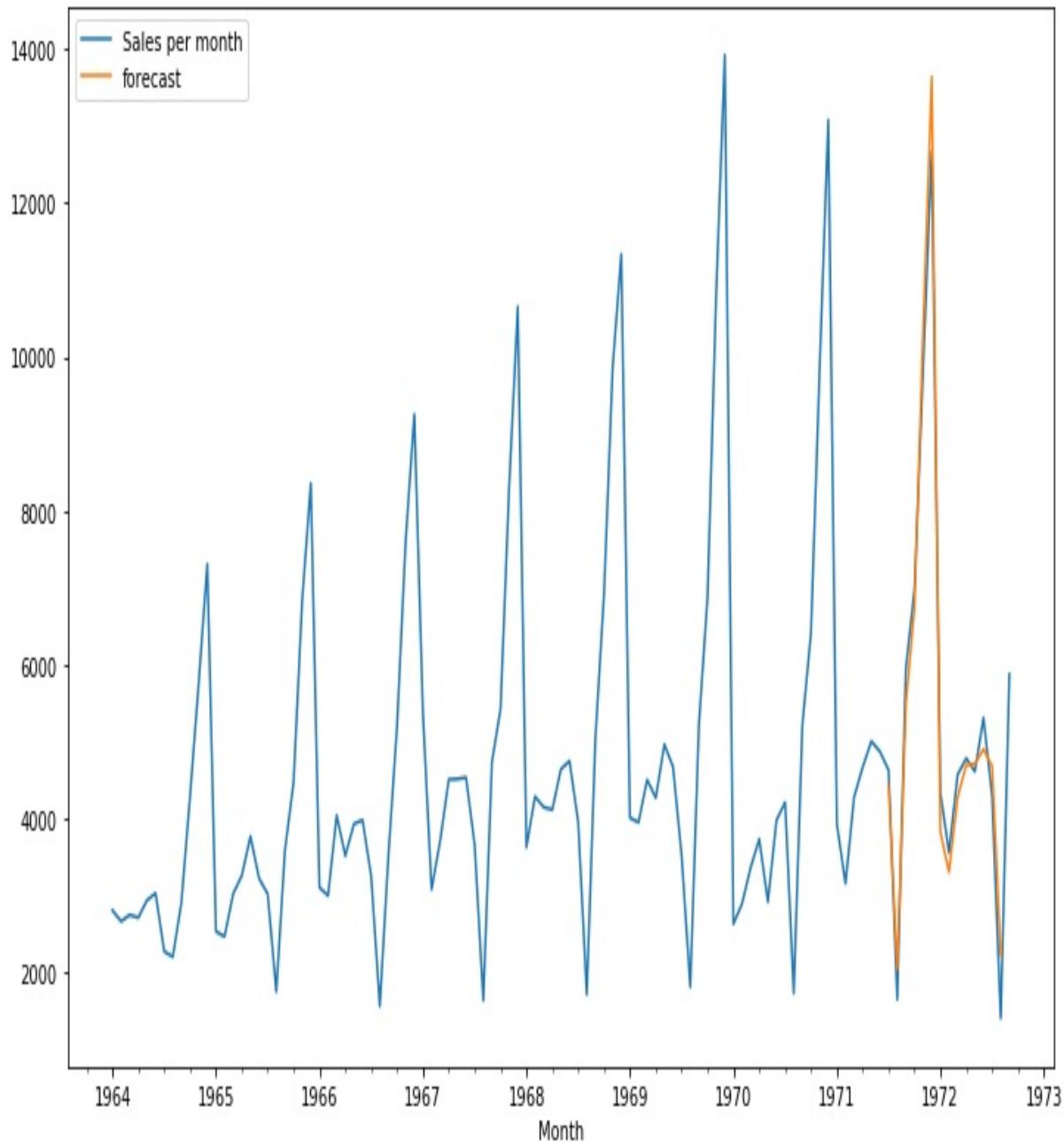
Here, the value of p that we get is 1, the value of q is 1, and the d value, which differs, is also 1.

Now, we can do the prediction by using the following code:

```
In[21]:  
df['forecast']=results.predict(start=90,end=103,dynamic=True)  
df[['Sales per month','forecast']].plot(figsize=(12,8))
```

The output is as follows:

```
<matplotlib.axes._subplots.AxesSubplot at 0x26ba5c09c88>
```



We can also create some future dates by using the `DateOffset` feature from pandas. The following code helps us to create datasets for an additional two years, which will be our future dates:

```
| In[22]:  
| from pandas.tseries.offsets import DateOffset  
| future_dates=[df.index[-1]+ DateOffset(months=x)for x in range(0,24)]
```

In the next step, we will convert this data into dataframes. The code for this is shown here:

```
| In[23]:  
| future_datest_df=pd.DataFrame(index=future_dates[1:],columns=df.columns)
```

We can then check the first five records using the following code:

```
| In[24]:  
| future_datest_df.head()
```

The output for this is as follows:

	Sales per month	Sales per Month First Difference	forecast
1972-10-01	NaN	NaN	NaN
1972-11-01	NaN	NaN	NaN
1972-12-01	NaN	NaN	NaN
1973-01-01	NaN	NaN	NaN
1973-02-01	NaN	NaN	NaN

For the new added dates, we do not have the data from the sales, and we will be predicting the future sales data using our model.

In the next step, we will be concatenating the future dataset with the original dataset. The code for this is as follows:

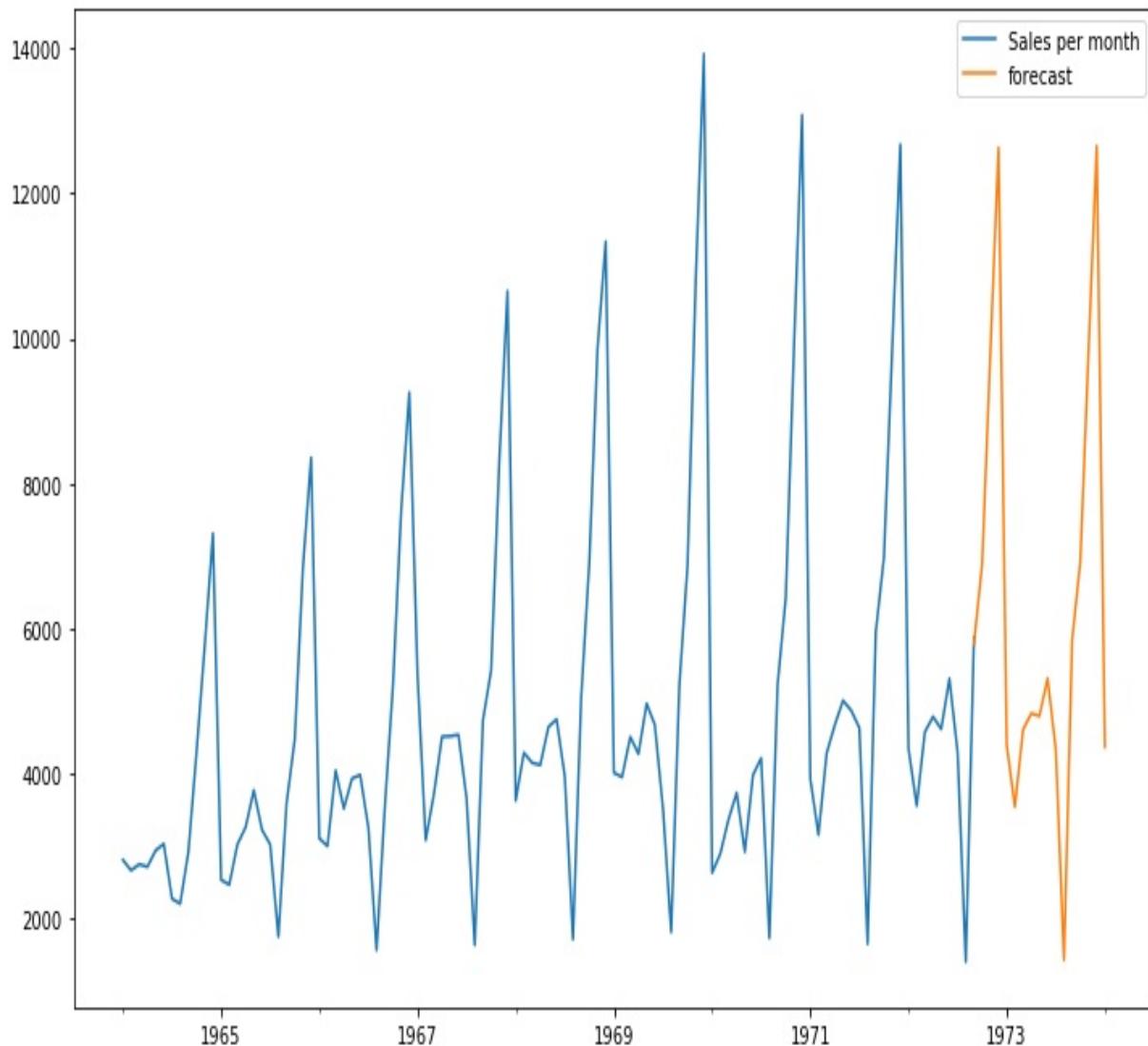
```
| In[25]:  
| future_df=pd.concat([df,future_datest_df])
```

Finally, we carry out the prediction for a future date we created by using the following code:

```
| In[26]:  
| future_df['forecast'] = results.predict(start = 104, end = 120, dynamic= True)  
| future_df[['Sales per month', 'forecast']].plot(figsize=(12, 8))
```

The output for this is as follows:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1e436306c88>
```



The preceding output shows the predictions of the future dataset that was initialized for the years 1973 and 1974.

This chapter covered two use cases: the stock prediction and forecasting using LSTM RNN, and the wine sales use case prediction. We also forecasted for the upcoming two years using ARIMA.

Summary

In this chapter, we have explored various concepts of deep learning techniques in which we discussed the RNN and the way in which the LSTM RNN works. We also implemented the stock prediction use case, where we discussed all the steps to follow using LSTM RNN. We also discussed how we can increase the accuracy of the LSTM RNN by considering large amounts of data.

In addition, we also implemented an additional use case in which we needed to predict the sales data of wine using an ARIMA model. We already discussed the ARIMA model in [Chapter 3, *Time Series Analysis and Forecasting*](#), and, using those concepts, we implemented a use case for time series data.

In the next chapter, we will discuss many other use cases for the concepts that we have learned about in this book.

What Is Next?

We produce huge amounts of data, around 2.5 quintillion bytes, every day. A huge chunk of this data is financial data, from transactions, banking, investments, and trading. Financial data can be quite complex, ranging from time-series transaction data to high frequency trading and algorithmic trading data that's produced at short intervals in a single day.

In this chapter, we will take a look at different use cases in which we can apply all of the different techniques to do with Python, machine learning, and deep learning that we learned about in this book.

In this chapter, we will discuss the following topics:

- A summary of different financial applications
- Additional research paper links for further reference

A summary of different financial applications

The financial sector generates petabytes of data in transactions, transfers, exchange, and trading. This makes the financial sector an ideal candidate for deep learning, since this huge amount of data can be used for creating a robust model that can be used in various financial applications:

- Automating risk management.
- Real-time analytics. This includes the following:
 - Fraud detection
 - Algorithm trading
- Managing customer data.
- Financial process automation.
- Financial security analysis.
- Credit score analysis and underwriting decisions.
- Automated algorithmic trading.
- Financial recommendations and predictions.
- Financial mergers and acquisitions.

In the following sections, we'll discuss these financial applications a little further.

Automating risk management

Risk management is one of the most important areas for any financial company involved with different kinds of decision-making work, such as security or strategy. There are many ways through which different kind of risks can appear, due to different competitors in the market, regulating company policies because of changes in government policies, and monetizing risks. Since, in finance, we have huge amounts of data, such as customer data or other financial data, by using machine learning models and deep learning models, we can make better risk management decisions.

Real-time analytics

Real-time analytics data helps to apply dynamic predictions and sentiment analysis on data. This can be applied to a huge amount of data very easily. There are two main applications in which real-time analytics can be applied in finance:

- **Fraud detection:** It is always very important for any financial institution to provide security for its customers. Usually, there are various ways in which fraudsters use loopholes to find traps that might be bad for a country or a company. We see people who apply for credit cards and do not pay their bills, loans that are converted into NPAs, and various other cases. Machine learning experts are able to create algorithms that are able to detect and find anomalies or frauds. Fraud detection can be used in the banking domain extensively for this purpose.
- **Algorithm trading:** This area has the highest impact on real-time data because each and every second is always at stake. Based on the recent research that's done in financial use cases, which involves both structured and unstructured data, many financial companies and institutions have made useful and beneficial decisions from real-time data. These days, financial companies are hiring machine learning and deep learning experts and are using real-time data to create trading algorithms and predictive modeling to forecast market opportunities.

Managing customer data

As we know, data is increasing exponentially day by day due to social networking sites, mobile interactions, and excessive use of digital technologies. This data consists of both structured and unstructured data. Usually, companies apply machine learning and deep learning techniques to understand or extract intelligence and useful information from the data. Sentiment analysis is one key application that is used to understand the sentiments related to stock prices.

Financial process automation

Deep learning applications can help in many common financial activities, such as the following:

- Chat bots for end user interaction
- Automated form filling
- Financial training activities for employees

Leading financial companies use natural language processing and pattern recognition to extract data from legal financial documents. Many firms also employ AI-driven chat bots in their applications.

Financial security analysis

Machine learning can also be used for detecting fraudulent financial transactions. Machine learning models can be trained on various aspects of data, such as the customer's location, their behavior, and their previous transactions, to detect whether a real-time transaction is valid or not with high precision.

Whenever a suspicious or ambiguous transaction is detected by the model, it can communicate with the user and even ask for further identification to validate the transaction. These models can also block transactions if they have a high chance of being fraudulent.

Deep learning models can be trained on numerous parameters of network security and financial monitoring data, which can help to create near real-time inferences on fraudulent transactions and cyber attack threats.

Many financial online payment gateway companies rely heavily on machine learning and deep learning techniques to increase their security.

Credit score analysis and underwriting decisions

Deep learning models that are trained on a large amount of customer data that's made up of numerous parameters, ranging from background to financial status, can be used to predict the credit score of a new customer effectively. Banking and insurance companies can use archived historical customer data such as existing loans, payment history, credit card uses, and mortgages to build a model.

Automated algorithmic trading

Deep learning models help to make better trading decisions. Trading models can be trained on all of the current and historical trade data, along with the current news, which helps to predict patterns and real-time stock prices. These recommendations are used to sell, hold, or buy stock.

Financial recommendations and predictions

Deep learning can help in portfolio management. Algorithms can be used to allocate and manage the customer's assets effectively. These models can provide an indication of the best way to manage and increase savings based on the current financial assets and target goals. They provide the best way to achieve financial goals by minimizing risks.

Many online banking and insurance providers provide the best investment and insurance policies based on their current customer's financial status and future financial goals. Many portals use machine learning and deep learning techniques to get the best possible policy that's suited to a particular person.

Financial data is the key for creating financial models that are used for helping in various financial use cases. Many financial companies make huge profits from the business of selling real-time transaction data. This helps to build well-trained models for predicting trends in the financial market.

Financial mergers and acquisitions

The logical reasoning behind the process of merging is that two separate companies are of more value together than as separate entities. This consolidation of two companies is a critical corporate strategy for companies to preserve their competitive advantages.

To facilitate merging and acquisitions, many investment broker firms employ the use of data analysis and modeling. This can predict the future valuations of the companies if two firms merge, what the per share valuations of the two firms should be, and what the stock prices of the merged entities will be as well. The financial variables and other machine learning approaches can also identify the potential acquisition targets.

Deep learning can also help in predicting the success of a start up venture and the investment roadmap for early stage ventures. This is commonly defined as a two-way strategy that generates a huge amount of revenue to a company's founders, investors, and first employees. A company can either have an **Initial Public Offering (IPO)** by going to a public stock market, or it might be merged or acquired by (M&A) another company. In this case, those who have previously invested money receive immediate cash in return for their shares. This process also helps to prepare exit strategies. Deep learning allows investors to explore their possibilities by creating a predictive model that is able to predict whether a start-up will be successful.

Additional research paper links for further reference

In this section, I've listed some additional research paper links that you can use to find out more about machine learning and deep learning in the financial domain:

- **Deep learning in finance:** <https://journalofbigdata.springeropen.com/articles/10.1186/s40537-017-0111-6>
- **Machine learning applications in financial markets:** <https://www.cse.iitb.ac.in/~saketh/research/ppBTP2010.pdf>
- **Deep learning for financial sentiment analysis in financial news providers:** <https://ieeexplore.ieee.org/abstract/document/7752381/>
- **Deep learning for determining mortgage risks:** <https://arxiv.org/abs/1607.02470>
- **Credit risk assessments:** <https://link.springer.com/article/10.1023/A:1008699112516>
- **Universal features of price formation in financial markets: perspectives from deep learning:** https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3141294

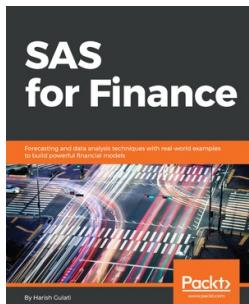
Summary

We have now reached the end of this book. We started by covering various important concepts to do with Python, including statsmodels and deep learning techniques. We looked at various basic libraries, such as NumPy, pandas, and matplotlib, in Python, and then explored time series data and learned about various inbuilt functions and techniques that can be used to handle it. We also looked at the ARIMA model, which was useful for forecasting time series financial data, such as sales data. We then moved on to looking at how to measure investment risks, including portfolio allocation and optimization using Python and the statsmodels library. After that, we discussed regression analysis in finance and looked at a few machine learning algorithms, such as simple linear regression, multiple linear regression, and decision trees. Finally, we explored how we can use deep learning techniques in finance. We used recurrent neural networks and attempted to predict future stock prices.

We covered all of these topics by looking at various examples. We applied various statistical concepts such as the ARIMA model, Monte Carlo simulation, the Black Scholes model, and deep learning techniques using LSTM RNN. You should now be able to use this knowledge in a wide variety of scenarios related to finance. Happy learning!

Other Books You May Enjoy

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

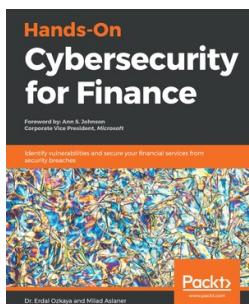


SAS for Finance

Harish Gulati

ISBN: 9781788624565

- Understand time series data and its relevance in the financial industry
- Build a time series forecasting model in SAS using advanced modeling theories
- Develop models in SAS and infer using regression and Markov chains
- Forecast inflation by building an econometric model in SAS for your financial planning
- Manage customer loyalty by creating a survival model in SAS using various groupings
- Understand similarity analysis and clustering in SAS using time series data



Hands-On Cybersecurity for Finance

Dr. Erdal Ozkaya, Milad Aslaner

ISBN: 9781788836296

- Understand the cyber threats faced by organizations
- Discover how to identify attackers
- Perform vulnerability assessment, software testing, and pentesting
- Defend your financial cyberspace using mitigation techniques and remediation plans
- Implement encryption and decryption
- Understand how Artificial Intelligence (AI) affects cybersecurity

Leave a review - let other readers know what you think

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