

MUHAMMAD ASIF



PYTHON FOR GEEKS

Build production-ready applications using advanced
Python concepts and industry best practices



BIRMINGHAM—MUMBAI

Python for Geeks

Copyright © 2021 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.

Group Product Manager: Richa Tripathi

Publishing Product Manager: Sathyanarayanan Ellapulli

Senior Editor: Rohit Singh

Content Development Editor: Vaishali Ramkumar

Technical Editor: Karan Solanki

Copy Editor: Safis Editing

Project Coordinator: Deeksha Thakkar

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Prashant Ghare

First published: August 2021

Production reference: 1090921

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80107-011-9

www.packt.com

To my wife, Saima Arooj, without whose loving support it would not have been possible to complete this book. To my daughters, Sana Asif and Sara Asif, and my son, Zain Asif, who were my inspiration throughout this journey. To the memory of my father and mother for their sacrifices and for exemplifying the power of determination for me. And to my siblings, for all the support and encouragement they have provided in terms of my education and career. –

Muhammad Asif

Contributors

About the author

Muhammad Asif is a principal solution architect with a wide range of multi-disciplinary experience in web development, network and cloud automation, virtualization, and machine learning. With a strong multi-domain background, he has led many large-scale projects to successful deployment. Although moving to more leadership roles in recent years, Muhammad has always enjoyed solving real-world problems by using appropriate technology and by writing the code himself. He earned a Ph.D. in computer systems from Carleton University, Ottawa, Canada in 2012 and currently works for Nokia as a solution lead.

I wish to thank those people who have been close to me and supported me, especially Imran Ahmad who contributed as a co-author of chapter 1.

About the reviewers

Harshit Jain is a data scientist with 5 years of programming experience, helping companies build and apply advanced machine learning algorithms to enhance business efficiency. He has a broad understanding of data science, ranging from classical machine learning to deep learning and computer vision. He is very skilled at applying data science techniques to e-commerce firms. During his spare time, he mentors aspiring data scientists to sharpen their skills. His talks and articles, including *Learning the Basics of Data Science* and *How to Check the Impact on Marketing Activities – Market Mix Modeling*, show that data science is not just his profession but a passion. Reviewing this book is his first, but certainly not the last, attempt to consolidate his valuable Python knowledge to help you in your learning.

Sourabh Bhavsar is a senior full stack developer, agile, and cloud practitioner with over 7 years of experience in the software industry. He has completed a postgraduate course in artificial intelligence and machine learning from the University of Texas at Austin, and also holds an MBA (in marketing) and a bachelor's degree in engineering (IT) from the University of Pune, India. He currently works at PayPal as a lead technical member where he is responsible for designing and developing microservice-based solutions and implementing various types of workflow and orchestration engines. Sourabh believes in continuous learning and enjoys exploring new web technologies. When not coding, he likes to play Tabla and read about astrology.

OceanofPDF.com

Table of Contents

[Preface](#)

Section 1: Python, beyond the Basics

Chapter 1: Optimal Python Development Life Cycle

Python culture and community

Different phases of a Python project

Strategizing the development process

Iterating through the phases

Aiming for MVP first

Strategizing development for specialized domains

Effectively documenting Python code

Python comments

Docstring

Functional or class-level documentation

Developing an effective naming scheme

Methods

Variables

Constant

Classes

Packages

Modules

Import conventions

Arguments

Useful tools

Exploring choices for source control

What does not belong to the source control repository?

Understanding strategies for deploying the code

Batch development

Python development environments

IDLE

Sublime Text

[PyCharm](#)

[Visual Studio Code](#)

[PyDev](#)

[Spyder](#)

[Summary](#)

[Questions](#)

[Further reading](#)

[Answers](#)

Chapter 2: Using Modularization to Handle Complex Projects

Technical requirements

Introduction to modules and packages

Importing modules

Using the import statement

Using the __import__ statement

Using the importlib.import_module statement

Absolute versus relative import

Loading and initializing a module

Standard modules

Writing reusable modules

Building packages

Naming

Package initialization file

Building a package

Accessing packages from any location

Sharing a package

Building a package as per the PyPA guidelines

Installing from the local source code using pip

Publishing a package to Test PyPI

Installing the package from PyPI

Summary

Questions

Further reading

Answers

Chapter 3: Advanced Object-Oriented Python Programming

Technical requirements

Introducing classes and objects

Distinguishing between class attributes and instance attributes

Using constructors and destructors with classes

Distinguishing between class methods and instance methods

Special methods

Understanding OOP principles

Encapsulation of data

Encompassing data and actions

Hiding information

Protecting the data

Using traditional getters and setters

Using property decorators

Extending classes with inheritance

Simple inheritance

Multiple inheritance

Polymorphism

Method overloading

Method overriding

Abstraction

Using composition as an alternative design approach

Introducing duck typing in Python

Learning when not to use OOP in Python

Summary

Questions

Further reading

Answers

Section 2: Advanced Programming Concepts

Chapter 4: Python Libraries for Advanced Programming

[Technical requirements](#)

[Introducing Python data containers](#)

[Strings](#)

[Lists](#)

[Tuples](#)

[Dictionaries](#)

[Sets](#)

[Using iterators and generators for data processing](#)

[Iterators](#)

[Generators](#)

[Handling files in Python](#)

[File operations](#)

[Using a context manager](#)

[Operating on multiple files](#)

[Handling errors and exceptions](#)

[Working with exceptions in Python](#)

[Raising exceptions](#)

[Defining custom exceptions](#)

[Using the Python logging module](#)

[Introducing core logging components](#)

[Working with the logging module](#)

[What to log and what not to log](#)

[Summary](#)

[Questions](#)

[Further reading](#)

[Answers](#)

Chapter 5: Testing and Automation with Python

Technical requirements

Understanding various levels of testing

Unit testing

Integration testing

System testing

Acceptance testing

Working with Python test frameworks

Working with the unittest framework

Working with the pytest framework

Executing TDD

Red

Green

Refactor

Introducing automated CI

Summary

Questions

Further reading

Answers

Chapter 6: Advanced Tips and Tricks in Python

[Technical requirements](#)

[Learning advanced tricks for using functions](#)

[Introducing the counter, itertools, and zip functions for iterative tasks](#)

[Using filters, mappers, and reducers for data transformations](#)

[Learning how to build lambda functions](#)

[Embedding a function within another function](#)

[Modifying function behavior using decorators](#)

[Understanding advanced concepts with data structures](#)

[Embedding a dictionary inside a dictionary](#)

[Using comprehension](#)

[Introducing advanced tricks with pandas DataFrame](#)

[Learning DataFrame operations](#)

[Learning advanced tricks for a DataFrame object](#)

[Summary](#)

[Questions](#)

[Further reading](#)

[Answers](#)

Section 3: Scaling beyond a Single Thread

Chapter 7: Multiprocessing, Multithreading, and Asynchronous Programming

Technical requirements

Understanding multithreading in Python and its limitations

What is a Python blind spot?

Learning the key components of multithreaded programming in Python

Case study – a multithreaded application to download files from Google Drive

Going beyond a single CPU – implementing multiprocessing

Creating multiple processes

Sharing data between processes

Exchanging objects between processes

Synchronization between processes

Case study – a multiprocessor application to download files from Google Drive

Using asynchronous programming for responsive systems

Understanding the asyncio module

Distributing tasks using queues

Case study – asyncio application to download files from Google Drive

Summary

Questions

Further reading

Answers

Chapter 8: Scaling out Python Using Clusters

[Technical requirements](#)

[Learning about the cluster options for parallel processing](#)

[Hadoop MapReduce](#)

[Apache Spark](#)

[Introducing RDDs](#)

[Learning RDD operations](#)

[Creating RDD objects](#)

[Using PySpark for parallel data processing](#)

[Creating SparkSession and SparkContext programs](#)

[Exploring PySpark for RDD operations](#)

[Learning about PySpark DataFrames](#)

[Introducing PySpark SQL](#)

[Case studies of using Apache Spark and PySpark](#)

[Case study 1 – Pi \(\$\pi\$ \) calculator on Apache Spark](#)

[Case study 2 – Word cloud using PySpark](#)

[Summary](#)

[Questions](#)

[Further reading](#)

[Answers](#)

Chapter 9: Python Programming for the Cloud

[Technical requirements](#)

[Learning about the cloud options for Python applications](#)

[Introducing Python development environments for the cloud](#)

[Introducing cloud runtime options for Python](#)

[Building Python web services for cloud deployment](#)

[Using Google Cloud SDK](#)

[Using the GCP web console](#)

[Using Google Cloud Platform for data processing](#)

[Learning the fundamentals of Apache Beam](#)

[Introducing Apache Beam pipelines](#)

[Building pipelines for Cloud Dataflow](#)

[Summary](#)

[Questions](#)

[Further reading](#)

[Answers](#)

Section 4: Using Python for Web, Cloud, and Network Use Cases

Chapter 10: Using Python for Web Development and REST API

Technical requirements

Learning requirements for web development

Web frameworks

User interface

Web server/application server

Database

Security

API

Documentation

Introducing the Flask framework

Building a basic application with routing

Handling requests with different HTTP method types

Rendering static and dynamic contents

Extracting parameters from an HTTP request

Interacting with database systems

Handling errors and exceptions in web applications

Building a REST API

Using Flask for a REST API

Developing a REST API for database access

Case study– Building a web application using the REST API

Summary

Questions

Further reading

Answers

Chapter 11: Using Python for Microservices Development

[Technical requirements](#)

[Introducing microservices](#)

[Learning best practices for microservices](#)

[Building microservices-based applications](#)

[Learning microservice development options in Python](#)

[Introducing deployment options for microservices](#)

[Developing a sample microservices-based application](#)

[Summary](#)

[Questions](#)

[Further reading](#)

[Answers](#)

Chapter 12: Building Serverless Functions using Python

Technical requirements

Introducing serverless functions

Benefits

Use cases

Understanding the deployment options for serverless functions

Learning how to build serverless functions

Building an HTTP-based Cloud Function using the GCP Console

Case study – building a notification app for cloud storage events

Summary

Questions

Further reading

Answers

Chapter 13: Python and Machine Learning

[Technical requirements](#)

[Introducing machine learning](#)

[Using Python for machine learning](#)

[Introducing machine learning libraries in Python](#)

[Best practices of training data with Python](#)

[Building and evaluating a machine learning model](#)

[Learning about an ML model building process](#)

[Building a sample ML model](#)

[Evaluating a model using cross-validation and fine tuning hyperparameters](#)

[Saving an ML model to a file](#)

[Deploying and predicting an ML model on GCP Cloud](#)

[Summary](#)

[Questions](#)

[Further reading](#)

[Answers](#)

Chapter 14: Using Python for Network Automation

Technical requirements

Introducing network automation

Merits and challenges of network automation

Use cases

Interacting with network devices

Protocols for interacting with network devices

Interacting with network devices using SSH-based Python libraries

Interacting with network devices using NETCONF

Integrating with network management systems

Using location services endpoints

Getting an authentication token

Getting network devices and an interface inventory

Updating the network device port

Integrating with event-driven systems

Creating subscriptions for Apache Kafka

Processing events from Apache Kafka

Renewing and deleting a subscription

Summary

Questions

Further reading

Answers

Other Books You May Enjoy

OceanofPDF.com

Preface

Python is a multipurpose language that can be used for solving any medium to complex problems in several fields. *Python for Geeks* will teach you how to advance in your career with the help of expert tips and tricks.

You'll start by exploring the different ways of using Python optimally, both from a design and implementation point of view. Next, you'll understand the lifecycle of a large-scale Python project. As you advance, you'll focus on different ways of creating an elegant design by modularizing a Python project and learn best practices and design patterns for using Python. You'll also discover how to scale out Python beyond a single thread and how to implement multiprocessing and multithreading in Python. In addition to this, you'll understand how you can not only use Python to deploy on a single machine but also using clusters in a private environment as well as in public cloud computing environments. You'll then explore data processing techniques, focus on reusable, scalable data pipelines, and learn how to use these advanced techniques for network automation, serverless functions, and machine learning. Finally, you'll focus on strategizing web development design using the techniques and best practices covered in the book.

By the end of this Python book, you'll be able to do some serious Python programming for large-scale complex projects.

Who this book is for

This book is for intermediate-level Python developers in any field who are looking to build their skills to develop and manage large-scale complex projects. Developers who want to create reusable modules and Python libraries and cloud developers building applications for cloud deployment will also find this book useful. Prior experience with Python will help you get the most out of this book.

What this book covers

[Chapter 1](#), *Optimal Python Development Life Cycle*, helps you to understand the lifecycle of a typical Python project and its phases, with a discussion of best practices for writing Python code.

[Chapter 2](#), *Using Modularization to Handle Complex Projects*, focuses on understanding the concepts of modules and packages in Python.

[Chapter 3](#), *Advanced Object-Oriented Python Programming*, discusses how the advanced concepts of object-oriented programming can be implemented using Python.

[Chapter 4](#), *Python Libraries for Advanced Programming*, explores advanced concepts such as iterators, generators, error and exception handling, file handling, and logging in Python.

[Chapter 5](#), *Testing and Automation with Python*, introduces not only different types of test automation such as unit testing, integration testing, and system testing but also discusses how to implement unit tests using popular test frameworks.

[Chapter 6](#), *Advanced Tips and Tricks in Python*, discusses advanced features of Python for data transformation, building decorators, and also how to use data structures including pandas DataFrames for analytics applications.

[Chapter 7](#), *Multiprocessing, Multithreading, and Asynchronous Programming*, helps you to learn about different options for building multi-threaded or multi-processed applications using built-in libraries in Python.

[Chapter 8](#), *Scaling Out Python using Clusters*, explores how to work with Apache Spark and how we can write Python applications for large data processing applications that can be executed using an Apache Spark cluster.

[Chapter 9](#), *Python Programming for the Cloud*, discusses how to develop and deploy applications to a cloud platform and how to use Apache Beam in general and for Google Cloud Platform in particular.

[Chapter 10](#), *Using Python for Web Development and REST API*, focuses on using the Flask framework to develop web applications, interact with databases, and build REST API or web services.

[Chapter 11](#), *Using Python for Microservices Development*, introduces microservices and how to use the Django framework to build a sample microservice and integrate it with a Flask-based microservice.

[*Chapter 12*](#), *Building Serverless Functions using Python*, addresses the role of serverless functions in cloud computing and how to build them using Python.

[*Chapter 13*](#), *Python and Machine Learning*, helps you to understand how to use Python to build, train, and evaluate machine learning models and how to deploy them in the cloud.

[*Chapter 14*](#), *Using Python for Network Automation*, discusses the use of Python libraries in fetching data from a network device and **network management systems (NMSes)** and for pushing configurational data to devices or NMSes.

To get the most out of this book

Prior knowledge of Python is a must to get real benefits from this book. You will need Python version 3.7 or later installed on your system. All code examples have been tested with Python 3.7 and Python 3.8 and expected to work with any future 3.x release. A Google Cloud Platform account (a free trial will work fine) will be helpful to deploy some code examples in the cloud.

Software/hardware covered in the book	Operating system requirements
Python release 3.7 or above	Windows
Apache Spark release 3.1.1	macOS
Cisco IOS XR (network device) release 7.12	Linux
Nokia Network Services Platform (NSP) release 21.6	
Google Cloud Platform Cloud SDK release 343.0.0	

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section).

Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at

<https://github.com/PacktPublishing/Python-for-Geeks>. If there's an update to the code, it will be updated in the 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 and diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781801070119_ColorImages.pdf.

Conventions used

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

Code in text: 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: "Mount the downloaded **WebStorm-10*.dmg** disk image file as another disk in your system."

A block of code is set as follows:

```
resource = {  
    "api_key": "AIzaSyDYKmm85kebxddKrGns4z0",  
    "id": "0B8TxHW2Ci6dbckVwTRtTl3RUU",  
    "fields": "files(name, id, webContentLink)",  
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
#casestudy1.py: Pi calculator  
  
from operator import add  
  
from random import random  
  
from pyspark.sql import SparkSession  
  
spark = SparkSession.builder.  
  
master("spark://192.168.64.2:7077") \  
    .appName("Pi claculator app") \  
    .getOrCreate()  
  
partitions = 2  
  
n = 10000000 * partitions  
  
def func(_):  
    x = random() * 2 - 1  
    y = random() * 2 - 1  
    return 1 if x ** 2 + y ** 2 <= 1 else 0  
  
count = spark.sparkContext.parallelize(range(1, n + 1),  
    partitions).map(func).reduce(add)
```

```
print("Pi is roughly %f" % (4.0 * count / n))
```

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

```
Pi is roughly 3.141479
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "As mentioned earlier, Cloud Shell comes with an editor tool that can be started by using the **Open editor** button."

TIPS OR IMPORTANT NOTES

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, email us at customercare@packtpub.com and mention the book title in the subject of your message.

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.packtpub.com/support/errata and fill in the form.

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.

Share Your Thoughts

Once you've read *Python for Geeks*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Section 1: Python, beyond the Basics

We start our journey by exploring different ways of using Python optimally, from both the design and the implementation points of view. We provide a deeper understanding of the life cycle of a large-scale Python project and its phases. Once we have that understanding, we investigate different ways of creating an elegant design by modularizing a Python project. Wherever necessary, we look under the hood to understand the internal workings of Python. This is followed by a deep dive into object-oriented programming in Python.

This section contains the following chapters:

- [*Chapter 1, Optimal Python Development Life Cycle*](#)
- [*Chapter 2, Using Modularization to Handle Complex Projects*](#)
- [*Chapter 3, Advanced Object-Oriented Python Programming*](#)

Chapter 1: Optimal Python Development Life Cycle

Keeping in mind your prior experience with Python, we have skipped the introductory details of the Python language in this chapter. First, we will have a short discussion of the broader open source Python community and its specific culture. That introduction is important, as this culture is reflected in code being written and shared by the Python community. Then, we will present the different phases of a typical Python project. Next, we will look at different ways of strategizing the development of a typical Python project.

Moving on, we will explore different ways of documenting the Python code. Later, we will look into various options of developing an effective naming scheme that can greatly help improve the maintenance of the code. We will also look into various options for using source control for Python projects, including situations where developers are mainly using Jupyter notebooks for development. Finally, we explore the best practices to deploy the code for use, once it is developed and tested.

We will cover the following topics in this chapter:

- Python culture and community
- Different phases of a Python project
- Strategizing the development process
- Effectively documenting Python code
- Developing an effective naming scheme
- Exploring choices for source control
- Understanding strategies for deploying the code
- Python development environments

This chapter will help you understand the life cycle of a typical Python project and its phases so that you can fully utilize the power of Python.

Python culture and community

Python is an interpreted high-level language that was originally developed by Guido van Rossum in 1991. The Python community is special in the sense that it pays close attention to how the code is written. For that, since the early days of Python, the Python community has created and maintained a particular flavor in its design philosophy. Today, Python is used in a wide variety of industries, ranging from education to medicine. But regardless of the industry in which it is used, the particular culture of the vibrant Python community is usually seen to be part and parcel of Python projects.

In particular, the Python community wants us to write simple code and avoid complexity wherever possible. In fact, there is an adjective, *Pythonic*, which means there are multiple ways to accomplish a certain task but there is a preferred way as per the Python community conventions and as per the founding philosophy of the language. Python nerds try their best to create artifacts that are as Pythonic as possible. Obviously, *unpythonic* code means that we are not good coders in the eyes of these nerds. In this book, we will try to go as Pythonic as possible as we can in our code and design. And there is something official about being Pythonic as well. Tim Peters has concisely written the philosophy of Python in a short document, *The Zen of Python*. We know that Python is said to be one of the easiest languages to read, and *The Zen of Python* wants to keep it that way. It expects Python to be explicit through good documentation and as clean and clear as possible. We can read *The Zen of Python* ourselves, as explained next.

In order to read *The Zen of Python*, open up a Python console and run the **import this** command, as shown in the following screenshot:

```
[1] import this
this

↳ The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
<module 'this' from '/usr/lib/python3.6>this.py'>
```



Figure 1.1 – The Zen of Python

The Zen of Python seems to be a cryptic text discovered in an old Egyptian tomb. Although it is deliberately written in this casual cryptic way, there is a deeper meaning to each line of text. Actually, look closer—it can be used as a guideline to code in Python. We will refer to different lines from *The Zen of Python* throughout the book. Let's first look into some excerpts from it, as follows:

- **Beautiful is better than ugly:** It is important to write code that is well-written, readable, and self-explanatory. Not only should it work—it should be beautifully written. While coding, we should avoid using shortcuts in favor of a style that is self-explanatory.
- **Simple is better than complex:** We should not unnecessarily complicate things. Whenever facing a choice, we should prefer the simpler solution. Nerdy, unnecessary, and complicated ways of writing code are discouraged. Even when it adds some more lines to the source code, simpler remains better than the complex alternative.
- **There should be one-- and preferably only one --obvious way to do it:** In broader terms, for a given problem there should be one possible best solution. We should strive to discover this. As we iterate through the design to improve it, regardless of our approach, our solution is expected to evolve and converge toward that preferable solution.
- **Now is better than never:** Instead of waiting for perfection, let's start solving the given problem using the information, assumptions, skills, tools, and infrastructure we have. Through the process of iteration, we will keep improving the solution. Let's keep things moving instead of idling. Do not slack while waiting for the perfect time. Chances are that the perfect time will never come.
- **Explicit is better than implicit:** The code should be as self-explanatory as possible. This should be reflected in the choice of variable names, the class, and the function design, as well as in the overall **end-to-end (E2E)** architecture. It is better to err on the side of caution. Always make it more explicit whenever facing a choice.
- **Flat is better than nested:** A nested structure is concise but also creates confusion. Prefer a flat structure wherever possible.

Different phases of a Python project

Before we discuss the optimal development life cycle, let's start by identifying the different phases of a Python project. Each phase can be thought of as a group of activities that are similar in nature, as illustrated in the following diagram:

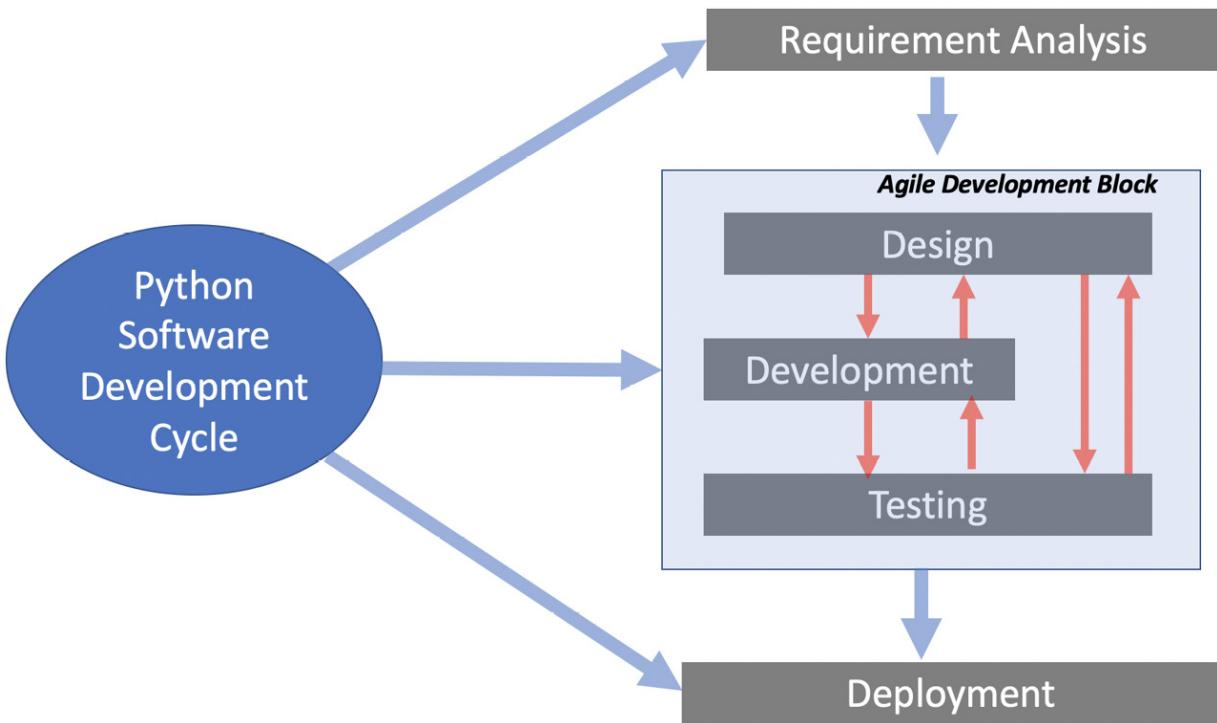


Figure 1.2 – Various phases of a Python project

The various phases of a typical Python project are outlined here:

- **Requirement analysis:** This phase is about collecting the requirements from all key stakeholders and then analyzing them to understand *what* needs to be done and later think about the *how* part of it. The stakeholders can be our actual users of the software or business owners. It is important to collect the requirements in as much detail as possible. Wherever possible, requirements should be fully laid out, understood, and discussed with the end user and stakeholders before starting the design and development.

An important point is to ensure that the requirement-analysis phase should be kept out of the iterative loop of the design, development, and testing phases. Requirement analysis should be fully conducted and complete before moving on to the next phases. The requirements should include both **functional requirements (FRs)** and **non-functional requirements (NFRs)**. FRs should be grouped into modules. Within each module, the requirements should be numbered in an effort to map them as closely as possible with the code modules.

- **Design:** Design is our technical response to the requirements as laid out in the requirement phase. In the design phase, we figure out the *how* part of the equation. It is a creative process where we use our experience and skills to come up with the right set and structure of modules and the interactions between them in the most efficient and optimal way.

Note that coming up with the right design is an important part of a Python project. Any missteps in the design phase will be much more expensive to correct than missteps in later phases. By some measure, it takes 20 times more effort to change the design and implement the design changes in the subsequent phases (for example, coding phase), as compared to a similar degree of changes if they happen in the coding phase—for example, the inability to correctly identify

classes or figure out the right data and compute the dimension of the project will have a major impact as compared to a mistake when implementing a function. Also, because coming up with the right design is a conceptual process, mistakes may not be obvious and cannot be caught by testing. On the other hand, errors in the coding will be caught by a well-thought-out exception-handling system.

In the design phase, we perform the following activities:

- a) We design the structure of the code and identify the modules within the code.
- b) We decide the fundamental approach and decide whether we should be using functional programming, OOP, or a hybrid approach.
- c) We also identify the classes and functions and choose the names of these higher-level components.

We also produce higher-level documentation.

- **Coding:** This is the phase where we will implement the design using Python. We start by implementing the higher-level abstractions, components, and modules identified by the design first, followed by the detailed coding. We will keep a discussion about the coding phase to a minimum in this section as we will discuss it extensively throughout the book.
- **Testing:** Testing is the process of verifying our code.
- **Deployment:** Once thoroughly tested, we need to hand over the solution to the end user. The end user should not see the details of our design, coding, or testing. Deployment is the process of providing a solution to the end user that can be used to solve the problem as detailed in the requirements. For example, if we are working to develop a **machine learning (ML)** project to predict rainfall in Ottawa, the deployment is about figuring out how to provide a usable solution to the end user.

Having understood what the different phases of a project are, we will move on to see how we can strategize the overall process.

Strategizing the development process

Strategizing the development process is about planning each of the phases and looking into the process flow from one phase to another. To strategize the development process, we need to first answer the following questions:

1. Are we looking for a minimal design approach and going straight to the coding phase with little design?
2. Do we want **test-driven development (TDD)**, whereby we first create tests using the requirements and then code them?
3. Do we want to create a **minimum viable product (MVP)** first and iteratively evolve the solution?
4. What is the strategy for validating NFRs such as security and performance?
5. Are we looking for a single-node development, or do we want to develop and deploy on the cluster or in the cloud?
6. What are the volume, velocity, and variety of our **input and output (I/O)** data? Is it a **Hadoop distributed file system (HDFS)** or **Simple Storage Service (S3)** file-based structure, or a **Structured Query Language (SQL)** or NoSQL database? Is the data

on-premises or in the cloud?

7. Are we working on specialized use cases such as ML with specific requirements for creating data pipelines, testing models, and deploying and maintaining them?

Based on the answers to these questions, we can strategize the steps for our development process. In more recent times, it is always preferred to use iterative development processes in one form or another. The concept of MVP as a starting goal is also popular. We will discuss these in the next subsections, along with the domains' specific development needs.

Iterating through the phases

Modern software development philosophy is based on short iterative cycles of design, development, and testing. The traditional waterfall model that was used in code development is long dead.

Selecting the right granularity, emphasis, and frequency of these phases depends on the nature of the project and our choice of code development strategy. If we want to choose a code development strategy with minimum design and want to go straight to coding, then the design phase is thin. But even starting the code straight away will require some thought in terms of the design of modules that will eventually be implemented.

No matter what strategy we choose, there is an inherent iterative relationship between the design, development, and testing phases. We initially start with the design phase, implement it in the coding phase, and then validate it by testing it. Once we have flagged the deficiencies, we need to go back to the drawing board by revisiting the design phase.

Aiming for MVP first

Sometimes, we select a small subset of the most important requirements to first implement the MVP with the aim of iteratively improving it. In an iterative process, we design, code, and test, until we create a final product that can be deployed and used.

Now, let's talk about how we will implement the solution of some specialized domains in Python.

Strategizing development for specialized domains

Python is currently being used for a wide variety of scenarios. Let's look into the following five important use cases to see how we can strategize the development process for each of them according to their specific needs:

- ML

- Cloud computing and cluster computing
- Systems programming
- Networking programming
- Serverless computing

We will discuss each of them in the following sections.

ML

Over the years, Python has become the most common language used for implementing ML algorithms. ML projects need to have a well-structured environment. Python has an extensive collection of high-quality libraries that are available for use for ML.

For a typical ML project, there is a **Cross-Industry Standard Process for Data Mining (CRISP-DM)** life cycle that specifies various phases of an ML project. A CRISP-DM life cycle looks like this:

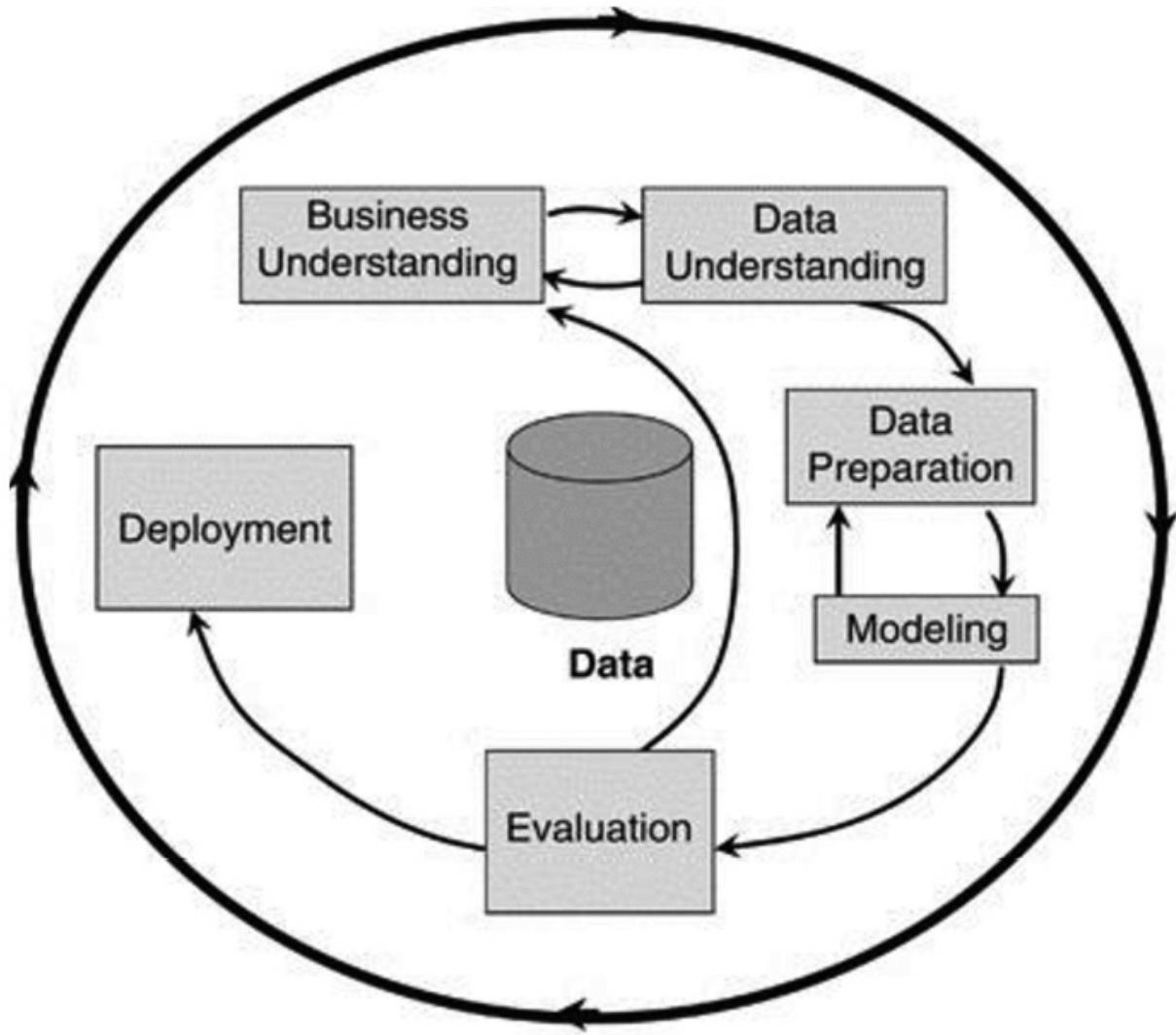


Figure 1.3 – A CRISP-DM life cycle

For ML projects, designing and implementing data pipelines is estimated to be almost 70% of the development effort. While designing data processing pipelines, we should keep in mind that the pipelines will ideally have these characteristics:

- They should be scalable.
- They should be reusable as far as possible.
- They should process both streaming and batch data by conforming to **Apache Beam** standards.
- They should mostly be a concatenation of fit and transform functions, as we will discuss in [Chapter 6, Advanced Tips and Tricks in Python](#).

Also, an important part of the testing phase for ML projects is the model evaluation. We need to figure out which of the performance metrics is the best one to quantify the performance of the model according to the requirement of the problem, nature of the data, and type of algorithm being

implemented. Are we looking at accuracy, precision, recall, F1 score, or a combination of these performance metrics? Model evaluation is an important part of the testing process and needs to be conducted in addition to the standard testing done in other software projects.

Cloud computing and cluster computing

Cloud computing and cluster computing add additional complexity to the underlying infrastructure. Cloud service providers offer services that need specialized libraries. The architecture of Python, which starts with bare-minimum core packages and the ability to import any further package, makes it well suited for cloud computing. The platform independence offered by a Python environment is critical for cloud and cluster computing. **Python** is the language of choice for **Amazon Web Services (AWS)**, Windows Azure, and Google Cloud Platform (GCP).

Cloud computing and cluster computing projects have separate development, testing, and production environments. It is important to keep the development and production environments in sync.

When using **infrastructure-as-a-service (IaaS)**, Docker containers can help a lot, and it is recommended to use them. Once we are using the Docker container, it does not matter where we are running the code as the code will have exactly the same environment and dependencies.

Systems programming

Python has interfaces to operating system services. Its core libraries have **Portable Operating System Interface (POSIX)** bindings that allow developers to create so-called shell tools, which can be used for system administration and various utilities. Shell tools written in Python are compatible across various platforms. The same tool can be used in Linux, Windows, and macOS without any change, making them quite powerful and maintainable.

For example, a shell tool that copies a complete directory developed and tested in Linux can run unchanged in Windows. Python's support for systems programming includes the following:

- Defining environment variables
- Support for files, sockets, pipes, processes, and multiple threads
- Ability to specify a **regular expression (regex)** for pattern matching
- Ability to provide command-line arguments
- Support for standard stream interfaces, shell-command launchers, and filename expansion
- Ability to zip file utilities
- Ability to parse **Extensible Markup Language (XML)** and **JavaScript Object Notation (JSON)** files

When using Python for system development, the deployment phase is minimal and may be as simple as packaging the code as an executable file. It is important to mention that Python is not intended to be used for the development of system-level drivers or operating system libraries.

Network programming

In the digital transformation era where **Information Technology (IT)** systems are moving quickly toward automation, networks are considered the main bottleneck in full-stack automation. The reason for this is the proprietary network operating systems from different vendors and a lack of openness, but the prerequisites of digital transformation are changing this trend and a lot of work is in progress to make the network programmable and consumable as a service (**network-as-a-service**, or **NaaS**). The real question is: *Can we use Python for network programming?* The answer is a big **YES**. In fact, it is one of the most popular languages in use for network automation.

Python support for network programming includes the following:

- Socket programming including **Transmission Control Protocol (TCP)** and **User Datagram Protocol (UDP)** sockets
- Support for client and server communication
- Support for port listening and processing data
- Executing commands on a remote **Secure Shell (SSH)** system
- Uploading and downloading files using **Secure Copy Protocol (SCP)/File Transfer Protocol (FTP)**
- Support for the library for **Simple Network Management Protocol (SNMP)**
- Support for the **REpresentational State Transfer (RESTCONF)** and **Network Configuration (NETCONF)** protocols for retrieving and updating configuration

Serverless computing

Serverless computing is a cloud-based application execution model in which the **cloud service providers (CSPs)** provide the computer resources and application servers to allow developers to deploy and execute the applications without any hassle of managing the computing resources and servers themselves. All of the major public cloud vendors (Microsoft Azure Serverless Functions, AWS Lambda, and **Google Cloud Platform**, or **GCP**) support serverless computing for Python.

We need to understand that there are still servers in a serverless environment, but those servers are managed by CSPs. As an application developer, we are not responsible for installing and maintaining the servers as well as having no direct responsibility for the scalability and performance of the servers.

There are popular serverless libraries and frameworks available for Python. These are described next:

- **Serverless:** The Serverless Framework is an open source framework for serverless functions or AWS Lambda services and is written using Node.js. Serverless is the first framework developed for building applications on AWS Lambda.
- **Chalice:** This is a Python serverless microframework developed by AWS. This is a default choice for developers who want to quickly spin up and deploy their Python applications using AWS Lambda Services, as this enables you to quickly spin up and deploy a working serverless application that scales up and down on its own as required, using AWS Lambda. Another key feature of Chalice is that it provides a utility to simulate your application locally before pushing it to the cloud.

- **Zappa:** This is more of a deployment tool built into Python and makes the deployment of your **Web Server Gateway Interface (WSGI)** application easy.

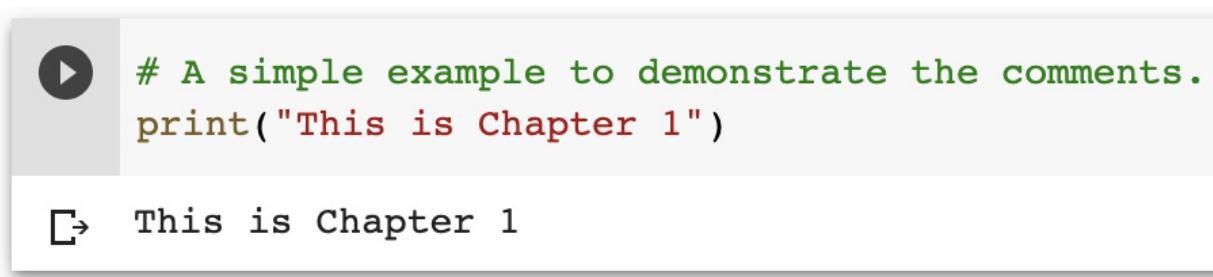
Now, let's look into effective ways of developing Python code.

Effectively documenting Python code

Finding an effective way to document code is always important. The challenge is to develop a comprehensive yet simple way to develop Python code. Let's first look into Python comments and then docstrings.

Python comments

In contrast with a docstring, Python comments are not visible to the runtime compiler. They are used as a note to explain the code. Comments start with a # sign in Python, as shown in the following screenshot:



The screenshot shows a code editor window. On the left, there is a play button icon. The code area contains the following text:
A simple example to demonstrate the comments.
print("This is Chapter 1")

On the right, the output of the code is displayed in a terminal-like interface:
This is Chapter 1

Figure 1.4 – An example of a comment in Python

Docstring

The main workhorse for documenting the code is the multiline comments block called a **docstring**. One of the features of the Python language is that DocStrings are associated with an object and are available for inspection. The guidelines for DocStrings are described in **Python Enhancement Proposal (PEP) 257**. According to the guidelines, their purpose is to provide an overview to the readers. They should have a good balance between being concise yet elaborative. DocStrings use a triple-double-quote string format: ("""").

Here are some general guidelines when creating a docstring:

- A docstring should be placed right after the function or the class definition.
- A docstring should be given a one-line summary followed by a more detailed description.

- Blank spaces should be strategically used to organize the comments but they should not be overused. You can use blank lines to organize code, but don't use them excessively.

In the following sections, let's take a look at more detailed concepts of docStrings.

Docstring styles

A Python docstring has the following slightly different styles:

- Google
- NumPy/SciPy
- Epytext
- Restructured

Docstring types

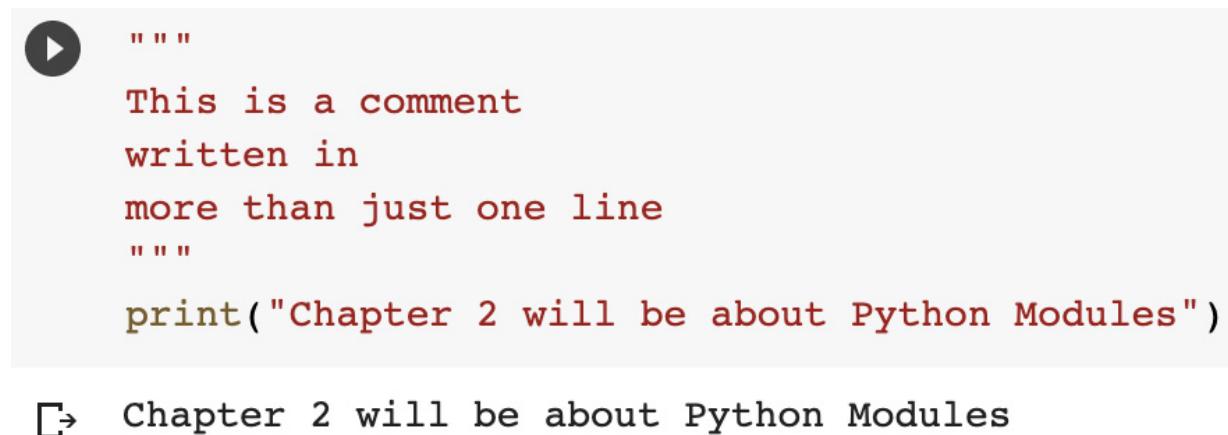
While developing the code, various types of documentation need to be produced, including the following:

- Line-by-line commentary
- Functional or class-level documentation
- Algorithmic details

Let's discuss them, one by one.

Line-by-line commentary

One simple use of a docstring is to use it to create multiline comments, as shown here:



```
    """
    This is a comment
    written in
    more than just one line
    """

    print("Chapter 2 will be about Python Modules")
```

→ Chapter 2 will be about Python Modules

Figure 1.5 – An example of a line-by-line commentary-type docstring

Functional or class-level documentation

A powerful use of a docstring is for functional or class-level documentation. If we place the docstring just after the definition of a function or a class, Python associates the docstring with the function or a class. This is placed in the `__doc__` attribute of that particular function or class. We can print that out at runtime by either using the `__doc__` attribute or by using the `help` function, as shown in the following example:

```
[5] def double(n):
    '''Takes in a number n, returns the double of its value'''
    return n**2

[6] print(double.__doc__)

⇒ Takes in a number n, returns the double of its value

[8] help(double)

⇒ Help on function double in module __main__:

double(n)
    Takes in a number n, returns the double of its value
```

Figure 1.6 – An example of the help function

When using a docstring for documenting classes, the recommended structure is as follows:

- A summary: usually a single line
- First blank line
- Any further explanation regarding the docstring
- Second blank line

An example of using a docstring on the class level is shown here:



```

class ComplexNumber:
    """
    This is a class for mathematical operations on complex numbers.

    Attributes:
        real (int): The real part of complex number.
        imag (int): The imaginary part of complex number.
    """

    def __init__(self, real, imag):
        """
        The constructor for ComplexNumber class.

        Parameters:
            real (int): The real part of complex number.
            imag (int): The imaginary part of complex number.
        """

    def add(self, num):
        """
        The function to add two Complex Numbers.

        Parameters:
            num (ComplexNumber): The complex number to be added.

        Returns:
            ComplexNumber: A complex number which contains the sum.
        """

        re = self.real + num.real
        im = self.imag + num.imag

        return ComplexNumber(re, im)

```

Figure 1.7 – An example of a class-level docstring

Algorithmic details

More and more often, Python projects use descriptive or predictive analytics and other complex logic. The details of the algorithm that is used need to be clearly specified with all the assumptions that were made. If an algorithm is implemented as a function, then the best place to write the summary of the logic of the algorithm is before the signature of the function.

Developing an effective naming scheme

If developing and implementing the right logic in code is science, then making it pretty and readable is an art. Python developers are famous for paying special attention to the naming scheme and bringing *The Zen of Python* into it. Python is one of the few languages that have comprehensive guidelines on the naming scheme written by Guido van Rossum. They are written in a *PEP 8* document that has a complete section on naming conventions, which is followed by many code bases. *PEP 8* has naming and style guidelines that are suggested. You can read more about it at <https://www.Python.org/dev/peps/pep-0008/>.

The naming scheme suggested in *PEP 8* can be summarized as follows:

- In general, all module names should be **all_lower_case**.
- All class names and exception names should be **CamelCase**.
- All global and local variables should be **all_lower_case**.
- All functions and method names should be **all_lower_case**.
- All constants should be **ALL_UPPER_CASE**.

Some guidelines about the structure of the code from *PEP 8* are given here:

- Indentation is important in Python. Do not use *Tab* for indentation. Instead, use four spaces.
- Limit nesting to four levels.
- Remember to limit the number of lines to 79 characters. Use the \ symbol to break long lines.
- To make code readable, insert two blank lines to separate functions.
- Insert a single black line between various logical sections.

Remember that *PEP* guidelines are just suggestions that may be customized by different teams. Any customized naming scheme should still use *PEP 8* as the basic guideline.

Now, let's look in more detail at the naming scheme in the context of various Python language structures.

Methods

Method names should use lowercase. The name should consist of a single word or more than one word separated by underscores. You can see an example of this here:

```
calculate_sum
```

To make the code readable, the method should preferably be a verb, related to the processing that the method is supposed to perform.

If a method is non-public, it should have a leading underscore. Here's an example of this:

```
_my_calculate_sum
```

Dunder or magic methods are methods that have a leading and trailing underscore. Examples of Dunder or magic methods are shown here:

- `__init__`
- `__add__`

It is never a good idea to use two leading and trailing underscores to name a method, and the use of these by developers is discouraged. Such a naming scheme is designed for Python methods.

Variables

Use a lowercase word or words separated by an underscore to represent variables. The variables should be nouns that correspond to the entity they are representing.

Examples of variables are given here:

- `x`
- `my_var`

The names of private variables should start with an underscore. An example is `_my_secret_variable`.

Boolean variables

Starting a Boolean variable with `is` or `has` makes it more readable. You can see a couple of examples of this here:

```
class Patient:  
    is_admitted = False  
    has_heartbeat = False
```

Collection variables

As collections are buckets of variables, it is a good idea to name them in a plural format, as illustrated here:

```
class Patient:  
    admitted_patients = ['John', 'Peter']
```

Dictionary variables

The name of the dictionary is recommended to be as explicit as possible. For example, if we have a dictionary of people mapped to the cities they are living in, then a dictionary can be created as follows:

```
persons_cities = {'Imran': 'Ottawa', 'Steven': 'Los Angeles'}
```

Constant

Python does not have immutable variables. For example, in C++, we can specify a **const** keyword to specify that the variable is immutable and is a constant. Python relies on naming conventions to specify constants. If the code tries to treat a constant as a regular variable, Python will not give an error.

For constants, the recommendation is to use uppercase words or words separated by an underscore. An example of a constant is given here:

CONVERSION_FACTOR

Classes

Classes should follow the CamelCase style—in other words, they should start with a capital letter. If we need to use more than one word, the words should not be separated by an underscore, but each word that is appended should have an initial capital letter. Classes should use a noun and should be named in a way to best represent the entity the class corresponds to. One way of making the code readable is to use classes with suffixes that have something to do with their type or nature, such as the following:

- **HadoopEngine**
- **ParquetType**
- **TextboxWidget**

Here are some points to keep in mind:

- There are exception classes that handle errors. Their names should always have **Error** as the trailing word. Here's an example of this:

FileNotFoundException

- Some of Python's built-in classes do not follow this naming guideline.
- To make it more readable, for base or abstract classes, a **Base** or **Abstract** prefix can be used. An example could be this:

AbstractCar

BaseClass

Packages

The use of an underscore is not encouraged while naming a package. The name should be short and all lowercase. If more than one word needs to be used, the additional word or words should also be lowercase. Here's an example of this:

mypackage

Modules

When naming a module, short and to-the-point names should be used. They need to be lowercase, and more than one word will be joined by underscores. Here's an example:

`main_module.py`

Import conventions

Over the years, the Python community has developed a convention for aliases that are used for commonly used packages. You can see an example of this here:

```
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels as sm
import matplotlib.pyplot as plt
```

Arguments

Arguments are recommended to have a naming convention similar to variables, because arguments of a function are, in fact, temporary variables.

Useful tools

There are a couple of tools that can be used to test how closely your code conforms to *PEP 8* guidelines. Let's look into them, one by one.

Pylint

Pylint can be installed by running the following command:

```
$ pip install pylint
```

Pylint is a source code analyzer that checks the naming convention of the code with respect to *PEP 89*. Then, it prints a report. It can be customized to be used for other naming conventions.

PEP 8

PEP 8 can be installed by running the following command:

```
pip: $ pip install pep8
```

pep8 checks the code with respect to *PEP 8*.

So far, we have learned about the various naming conventions in Python. Next, we will explore different choices for using source control for Python.

Exploring choices for source control

First, we will see a brief history of source control systems to provide a context. Modern source control systems are quite powerful. The evolution of the source control systems went through the following stages:

- **Stage 1:** The source code was initially started by local source control systems that were stored on a hard drive. This local code collection was called a local repository.
- **Stage 2:** But using source control locally was not suitable for larger teams. This solution eventually evolved into a central server-based repository that was shared by the members of the team working on a particular project. It solved the problem of code sharing among team members, but it also created an additional challenge of locking the files for the multiuser environment.
- **Stage 3:** Modern version control repositories such as Git evolved this model further. All members of a team now have a full copy of the repository that is stored. The members of the team now work offline on the code. They need to connect to the repository only when there is a need to share the code.

What does not belong to the source control repository?

Let's look into what should not be checked into the source control repository.

Firstly, anything other than the source code file shouldn't be checked in. The computer-generated files should not be checked into source control. For example, let's assume that we have a Python source file named **main.py**. If we compile it, the generated code does not belong to the repository. The compiled code is a derived file and should not be checked into source control. There are three reasons for this, outlined as follows:

- The derived file can be generated by any member of the team once we have the source code.
- In many cases, the compiled code is much larger than the source code, and adding it to the repository will make it slow and sluggish. Also, remember that if there are 16 members in the team, then all of them unnecessarily get a copy of that generated file, which will unnecessarily slow down the whole system.
- Source control systems are designed to store the delta or the changes you have made to the source files since your last commit. Files other than the source code files are usually binary files. The source control system is most likely unable to have a **diff** tool for that, and it will need to store the whole file each time it is committed. It will have a negative effect on the performance of the source control framework.

Secondly, anything that is confidential does not belong to the source control. This includes API keys and passwords.

For the source repository, GitHub is the preferred choice of the Python community. Much of the source control of the famous Python packages also resides on GitHub. If the Python code is to be

utilized across teams, then the right protocol and procedures need to be developed and maintained.

Understanding strategies for deploying the code

For projects where the development team is not the end user, it is important to come up with a strategy to deploy the code for the end user. For relatively large-scale projects, when there is a well-defined **DEV** and **PROD** environment, deploying the code and strategizing it becomes important.

Python is the language of choice for cloud and cluster computing environments as well.

Issues related to deploying the code are listed as follows:

- Exactly the same transformations need to happen in **DEV**, **TEST**, and **PROD** environments.
- As the code keeps getting updated in the **DEV** environment, how will the changes be synced to the **PROD** environment?
- What type of testing do you plan to do in the **DEV** and **PROD** environments?

Let's look into two main strategies for deploying the code.

Batch development

This is the traditional development process. We develop the code, compile it, and then test it. This process is repeated iteratively until all the requirements are met. Then, the developed code is deployed.

Employing continuous integration and continuous delivery

Continuous integration/continuous delivery (CI/CD) in the context of Python refers to continuous integration and deployment instead of conducting it as a batch process. It helps to create a **development-operations (DevOps)** environment by bridging the gap between development and operations.

CI refers to continuously integrating, building, and testing various modules of the code as they are being updated. For a team, this means that the code developed individually by each team member is integrated, built, and tested, typically many times a day. Once they are tested, the repository in the source control is updated.

An advantage of CI is that problems or bugs are fixed right in the beginning. A typical bug fixed on the day it was created takes much less time to resolve right away instead of resolving it days, weeks, or months later when it has already trickled down to other modules and those affected may have created multilevel dependencies.

Unlike Java or C++, Python is an interpreted language, which means the built code is executable on any target machine with an interpreter. In comparison, the compiled code is typically built for one type of target machine and may be developed by different members of the team. Once we have figured out which steps need to be followed each time a change is made, we can automate it.

As Python code is dependent on external packages, keeping track of their names and versions is part of automating the build process. A good practice is to list all these packages in a file named **requirements.txt**. The name can be anything, but the Python community typically tends to call it **requirements.txt**.

To install the packages, we will execute the following command:

```
$ pip install -r requirements.txt
```

To create a **requirements** file that represents the packages used in our code, we can use the following command:

```
$ pip freeze > requirements.txt
```

The goal of integration is to catch errors and defects early, but it has the potential to make the development process unstable. There will be times when a member of the team has introduced a major bug, thus *breaking the code*, if other team members may have to wait until that bug is resolved. Robust self-testing by team members and choosing the right frequency for integration will help to resolve the issue. For robust testing, running testing each time a change is made should be implemented. This testing process should be eventually completely automated. In the case of errors, the build should fail and the team member responsible for the defective module should be notified. The team member can choose to first provide a quick fix before taking time to resolve and fully test the problem to make sure other team members are not blocked.

Once the code is built and tested, we can choose to update the deployed code as well. That will implement the **CD** part. If we choose to have a complete CI/CD process, it means that each time a change is made, it is built and tested and the changes are reflected in the deployed code. If managed properly, the end user will benefit from having a constantly evolving solution. In some use cases, each CI/CD cycle may be an iterative move from MVP to a full solution. In other use cases, we are trying to capture and formulate a fast-changing real-world problem, discarding obsolete assumptions, and incorporating new information. An example is the pattern analysis of the COVID-19 situation, which is changing by the hour. Also, new information is coming at a rapid pace, and any use case related to it may benefit from CI/CD, whereby developers are constantly updating their solutions based on new emerging facts and information.

Next, we will discuss commonly used development environments for Python.

Python development environments

Text editors are a tempting choice for editing Python code. But for any medium-to-large-sized project, we have to seriously consider Python **integrated development environments (IDEs)**, which are very helpful for writing, debugging, and troubleshooting the code using the version control and facilitating ease of deployments. There are many IDEs available, mostly free, on the market. In this section, we will review a few of them. Note that we will not try to rank them in any order but will emphasize the value each of them brings, and it is up to the reader to make the best choice based on their past experience, project requirements, and the complexity of their projects.

IDLE

Integrated Development and Learning Environment (IDLE) is a default editor that comes with Python and is available for all main platforms (Windows, macOS, and Linux). It is free and is a decent IDE for beginners for learning purposes. It is not recommended for advanced programming.

Sublime Text

Sublime Text is another popular code editor and can be used for multiple languages. It is free for evaluation purposes only. It is also available for all main platforms (Windows, macOS, and Linux). It comes with basic Python support but with its powerful extensions framework, we can customize it to make a full development environment that needs extra skills and time. Integration with a version control system such as Git or **Subversion (SVN)** is possible with plugins but may not expose full version control features.

Atom is another popular editor that is also in the same category as Sublime Text. It is free.

PyCharm

PyCharm is one of the best Python IDE editors available for Python programming and it is available for Windows, macOS, and Linux. It is a complete IDE tailored for Python programming, which helps programmers with code completion, debugging, refactoring, smart search, access to popular database servers, integration with version control systems, and many more features. The IDE provides a plugin platform for developers to extend the base functionalities as needed. PyCharm is available in the following formats:

- Community version, which is free and comes for pure Python development

- Professional version, which is not free and comes with support for web development such as **HyperText Markup Language (HTML)**, JavaScript, and SQL

Visual Studio Code

Visual Studio Code (VS Code) is an open source environment developed by Microsoft. For Windows, VS Code is the best Python IDE. It does not come with a Python development environment by default. The Python extensions for VS Code can make it a Python development environment.

It is lightweight and full of powerful features. It is free and is also available for macOS and Linux. It comes with powerful features such as code completion, debugging, refactoring, searching, accessing database servers, version control system integration, and much more.

PyDev

If you are using or have used Eclipse, you may like to consider PyDev, which is a third-party editor for Eclipse. It is in the category of one of the best Python IDEs and can also be used for Jython and IronPython. It is free. As PyDev is just a plugin on top of Eclipse, it is available for all major platforms, such as Eclipse. This IDE comes with all the bells and whistles of Eclipse, and on top of that, it streamlines integration with Django, unit testing, and **Google App Engine (GAE)**.

Spyder

If you are planning to use Python for data science and ML, you may want to consider **Spyder** as your IDE. Spyder is written in Python. This IDE offers tools for full editing, debugging, interactive execution, deep inspection, and advanced visualization capabilities. Additionally, it supports integration with Matplotlib, SciPy, NumPy, Pandas, Cython, IPython, and SymPy to make it a default IDE for data scientists.

Based on the review of different IDEs in this section, we can recommend PyCharm and PyDev for professional application developers. But if you are more into data science and ML, Spyder is surely worth exploring.

Summary

In this chapter, we laid down the groundwork for the advanced Python concepts discussed in the later chapters of this book. We started by presenting the flavor, guidance, and ambience of a Python

project. We started the technical discussion by first identifying different phases of the Python project and then exploring different ways of optimizing it based on the use cases we are working on. For a terse language such as Python, good-quality documentation goes a long way to make the code readable and explicit.

We also looked into various ways of documenting the Python code. Next, we investigated the recommended ways of creating documentation in Python. We also studied the naming schemes that can help us in making code more readable. Next, we looked into the different ways we can use source control. We also figured out what are the different ways of deploying Python code. Finally, we reviewed a few development environments for Python to help you choose a development environment based on the background they have and the type of project you are going to work on.

The topics we covered in this chapter are beneficial for anyone who is starting a new project involving Python. These discussions help to make the strategy and design decision of a new project promptly and efficiently. In the next chapter, we will investigate how we can modularize the code of a Python project.

Questions

1. What is *The Zen of Python*?
2. In Python, what sort of documentation is available at runtime?
3. What is a CRISP-DM life cycle?

Further reading

- *Modern Python Cookbook – Second Edition*, by Steven F. Lott
- *Python Programming Blueprints*, by Daniel Furtado
- *Secret Recipes of the Python Ninja*, by Cody Jackson

Answers

1. A collection of 19 guidelines written by Tim Peters that apply to the design of Python projects.
2. As opposed to regular comments, docstrings are available at runtime to the compiler.
3. **CRISP-DM** stands for **Cross-Industry Standard Process for Data Mining**. It applies to a Python project life cycle in the ML domain and identifies different phases of a project.

Chapter 2: Using Modularization to Handle Complex Projects

When you start programming in Python, it is very tempting to put all your program code in a single file. There is no problem in defining functions and classes in the same file where your main program is. This option is attractive to beginners because of the ease of execution of the program and to avoid managing code in multiple files. But a single-file program approach is not scalable for medium- to large-size projects. It becomes challenging to keep track of all the various functions and classes that you define.

To overcome the situation, modular programming is the way to go for medium to large projects. Modularity is a key tool to reduce the complexity of a project. Modularization also facilitates efficient programming, easy debugging and management, collaboration, and reusability. In this chapter, we will discuss how to build and consume modules and packages in Python.

We will cover the following topics in this chapter:

- Introduction to modules and packages
- Importing modules
- Loading and initializing a module
- Writing reusable modules
- Building packages
- Accessing packages from any location
- Sharing a package

This chapter will help you understand the concepts of modules and packages in Python.

Technical requirements

The following are the technical requirements for this chapter:

- You need to have Python 3.7 or later installed on your computer.
- You need to register an account with Test PyPI and create an API token under your account.

Sample code for this chapter can be found at <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter02>.

Introduction to modules and packages

Modules in Python are Python files with a **.py** extension. In reality, they are a way to organize functions, classes, and variables using one or more Python files such that they are easy to manage, reuse across the different modules, and extend as the programs become complex.

A Python package is the next level of modular programming. A package is like a folder for organizing multiple modules or sub-packages, which is fundamental for sharing the modules for reusability.

Python source files that use only the standard libraries are easy to share and easy to distribute using email, GitHub, and shared drives, with the only caveat being that there should be Python version compatibility. But this sharing approach will not scale for projects that have a decent number of files and have dependencies on third-party libraries and may be developed for a specific version of Python. To rescue the situation, building and sharing packages is a must for efficient sharing and reusability of Python programs.

Next, we will discuss how to import modules and the different types of import techniques supported in Python.

Importing modules

Python code in one module can get access to the Python code in another module by a process called importing modules.

To elaborate on the different module and package concepts, we will build two modules and one main script that will use those two modules. These two modules will be updated or reused throughout this chapter.

To create a new module, we will create a **.py** file with the name of the module. We will create a **mycalculator.py** file with two functions: **add** and **subtract**. The **add** function computes the sum of the two numbers provided to the function as arguments and returns the computed value. The **subtract** function computes the difference between the two numbers provided to the function as arguments and returns the computed value.

A code snippet of **mycalculator.py** is shown next:

```
# mycalculator.py with add and subtract functions

def add(x, y):
    """This function adds two numbers"""
    return x + y

def subtract(x, y):
    """This function subtracts two numbers"""
```

```
    return x - y
```

Note that the name of the module is the name of the file.

We will create a second module by adding a new file with the name **myrandom.py**. This module has two functions: **random_1d** and **random_2d**. The **random_1d** function is for generating a random number between 1 and 9 and the **random_2d** function is for generating a random number between 10 and 99. Note that this module is also using the **random** library, which is a built-in module from Python.

The code snippet of **myrandom.py** is shown next:

```
# myrandom.py with default and custom random functions

import random

def random_1d():
    """This function generates a random number between 0 \
    and 9"""

    return random.randint (0,9)

def random_2d():
    """This function generates a random number between 10 \
    and 99"""

    return random.randint (10,99)
```

To consume these two modules, we also created the main Python script (**calcmain1.py**), which imports the two modules and uses them to achieve these two calculator functions. The **import** statement is the most common way to import built-in or custom modules.

A code snippet of **calcmain1.py** is shown next:

```
# calcmain1.py with a main function

import mycalculator

import myrandom

def my_main( ):
    """ This is a main function which generates two random\
        numbers and then apply
    calculator functions on them """

    x = myrandom.random_2d( )

    y = myrandom.random_1d( )

    sum = mycalculator.add(x, y)

    diff = mycalculator.subtract(x, y)

    print("x = {}, y = {}".format(x, y))
```

```

print("sum is {}".format(sum))
print("diff is {}".format(diff))

""" This is executed only if the special variable '__name__' is set as main"""
if __name__ == "__main__":
    my_main()

```

In this main script (another module), we import the two modules using the **import** statement. We defined the main function (**my_main**), which will be executed only if this script or the **calcmain1** module is executed as the main program. The details of executing the main function from the main program will be covered later in the *Setting special variables* section. In the **my_main** function, we are generating two random numbers using the **myrandom** module and then calculating the sum and difference of the two random numbers using the **mycalculator** module. In the end, we are sending the results to the console using the **print** statement.

IMPORTANT NOTE

A module is loaded only once. If a module is imported by another module or by the main Python script, the module will be initialized by executing the code in the module. If another module in your program imports the same module again, it will not be loaded twice but only once. This means if there are any local variables inside the module, they will act as a Singleton (initialized only once).

There are other options available to import a module, such as **importlib.import_module()** and the built-in **__import__()** function. Let's discuss how **import** and other alternative options works.

Using the **import** statement

As mentioned already, the **import** statement is a common way to import a module. The next code snippet is an example of using an **import** statement:

```
import math
```

The **import** statement is responsible for two operations: first, it searches for the module given after the **import** keyword, and then it binds the results of that search to a variable name (which is the same as the module name) in the local scope of the execution. In the next two subsections, we will discuss how **import** works and also how to import specific elements from a module or a package.

Learning how import works

Next, we need to understand how the **import** statement works. First, we need to remind ourselves that all global variables and functions are added to the global namespace by the Python interpreter at the start of an execution. To illustrate the concept, we can write a small Python program to spit out of the contents of the **globals** namespace, as shown next:

```
# globalmain.py with globals() function
```

```

def print_globals():
    print (globals())
def hello():
    print ("Hello")
if __name__ == "__main__":
    print_globals()

```

This program has two functions: **print_globals** and **hello**. The **print_globals** function will spit out the contents of the global namespace. The **hello** function will not be executed and is added here to show its reference in the console output of the global namespace. The console output after executing this Python code will be similar to the following:

```

{
    "__name__": "__main__",
    "__doc__": "None",
    "__package__": "None",
    "__loader__": "<_frozen_importlib_external.\\" +
        "SourceFileLoader object at 0x101670208>",
    "__spec__": "None",
    "__annotations__": {
    },
    "__builtins__": "<module 'builtins' (built-in)>",
    "__file__": "/ PythonForGeeks/source_code/chapter2/\" +
        "modules/globalmain.py",
    "__cached__": "None",
    "print_globals": "<function print_globals at \
        0x1016c4378>",
    "hello": "<function hello at 0x1016c4400>"
}

```

The key points to be noticed in this console output are as follows:

- The **__name__** variable is set to the **__main__** value. This will be discussed in more detail in the *Loading and initializing a module* section.
- The **__file__** variable is set to the file path of the main module here.
- A reference to each function is added at the end.

If we add `print(globals())` to our `calcmain1.py` script, the console output after adding this statement will be similar to the following:

```
{  
    "__name__": "__main__",  
    "__doc__": "None",  
    "__package__": "None",  
    "__loader__": "<_frozen_importlib_external.\\"  
                 SourceFileLoader object at 0x100de1208>",  
    "__spec__": "None",  
    "__annotations__": {},  
    "__builtins__": "<module 'builtins' (built-in)>",  
    "__file__": "/PythonForGeeks/source_code/chapter2/module1/      main.py",  
    "__cached__": "None",  
    "mycalculator": "<module 'mycalculator' from \\"  
                  '/PythonForGeeks/source_code/chapter2/modules/\\"  
                  mycalculator.py'>",  
    "myrandom": "<module 'myrandom' from  
'/PythonForGeeks/source_      code/chapter2/modules/myrandom.py'>",  
    "my_main": "<function my_main at 0x100e351e0>"  
}
```

An important point to note is that there are two additional variables (**mycalculator** and **myrandom**) added to the global namespace corresponding to each **import** statement used to import these modules. Every time we import a library, a variable with the same name is created, which holds a reference to the module just like a variable for the global functions (**my_main** in this case).

We will see, in other approaches of importing modules, that we can explicitly define some of these variables for each module. The **import** statement does this automatically for us.

Specific import

We can also import something specific (variable or function or class) from a module instead of importing the whole module. This is achieved using the **from** statement, such as the following:

```
from math import pi
```

Another best practice is to use a different name for an imported module for convenience or sometimes when the same names are being used for different resources in two different libraries. To illustrate this idea, we will be updating our `calcmain1.py` file (the updated program is `calcmain2.py`)

from the earlier example by using the **calc** and **rand** aliases for the **mycalculator** and **myrandom** modules, respectively. This change will make use of the modules in the main script much simpler, as shown next:

```
# calcmain2.py with alias for modules
import mycalculator as calc
import myrandom as rand

def my_main():
    """ This is a main function which generates two random\
        numbers and then apply calculator functions on them """
    x = rand.random_2d()
    y = rand.random_1d()
    sum = calc.add(x,y)
    diff = calc.subtract(x,y)
    print("x = {}, y = {}".format(x,y))
    print("sum is {}".format(sum))
    print("diff is {}".format(diff))

    """ This is executed only if the special variable '__name__' is set as main"""
if __name__ == "__main__":
    my_main()
```

As a next step, we will combine the two concepts discussed earlier in the next iteration of the **calcmain1.py** program (the updated program is **calcmain3.py**). In this update, we will use the **from** statement with the module names and then import the individual functions from each module. In the case of the **add** and **subtract** functions, we used the **as** statement to define a different local definition of the module resource for illustration purposes.

A code snippet of **calcmain3.py** is as follows:

```
# calcmain3.py with from and alias combined
from mycalculator import add as my_add
from mycalculator import subtract as my_subtract
from myrandom import random_2d, random_1d

def my_main():
    """ This is a main function which generates two random
        numbers and then apply calculator functions on them """
    x = random_2d()
```

```

y = random_1d()
sum = my_add(x,y)
diff = my_subtract(x,y)
print("x = {}, y = {}".format(x,y))
print("sum is {}".format(sum))
print("diff is {}".format(diff))
print(globals())
"""
This is executed only if the special variable '__name__' is set as main"""
if __name__ == "__main__":
    my_main()

```

As we used the **print (globals())** statement with this program, the console output of this program will show that the variables corresponding to each function are created as per our alias. The sample console output is as follows:

```
{
    "__name__": "__main__",
    "__doc__": "None",
    "__package__": "None",
    "__loader__": "<_frozen_importlib_external.\\" +
        "SourceFileLoader object at 0x1095f1208>",
    "__spec__": "None",
    "__annotations__": {},
    "__builtins__": "<module 'builtins' (built-in)>",
    "__file__": "/PythonForGeeks/source_code/chapter2/module1/main_2.py",
    "__cached__": "None",
    "my_add": "<function add at 0x109645400>",
    "my_subtract": "<function subtract at 0x109645598>",
    "random_2d": "<function random_2d at 0x10967a840>",
    "random_1d": "<function random_1d at 0x1096456a8>",
    "my_main": "<function my_main at 0x109645378>"
}
```

Note that the variables in bold correspond to the changes we made in the **import** statements in the **calcmain3.py** file.

Using the **__import__** statement

The `__import__` statement is a low-level function in Python that takes a string as input and triggers the actual import operation. Low-level functions are part of the core Python language and are typically meant to be used for library development or for accessing operating system resources, and are not commonly used for application development. We can use this keyword to import the **random** library in our **myrandom.py** module as follows:

```
#import random  
random = __import__('random')
```

The rest of the code in **myrandom.py** can be used as it is without any change.

We illustrated a simple case of using the `__import__` method for academic reasons and we will skip the advanced details for those of you who are interested in exploring as further reading. The reason for this is that the `__import__` method is not recommended to be used for user applications; it is designed more for interpreters.

The **importlib.import_module** statement is the one to be used other than the regular import for advanced functionality.

Using the `importlib.import_module` statement

We can import any module using the **importlib** library. The **importlib** library offers a variety of functions, including `__import__`, related to importing modules in a more flexible way. Here is a simple example of how to import a **random** module in our **myrandom.py** module using **importlib**:

```
import importlib  
random = importlib.import_module('random')
```

The rest of the code in **myrandom.py** can be used as it is without any change.

The **importlib** module is best known for importing modules dynamically and is very useful in cases where the name of the module is not known in advance and we need to import the modules at runtime. This is a common requirement for the development of plugins and extensions.

Commonly used functions available in the **importlib** module are as follows:

- **__import__**: This is the implementation of the `__import__` function, as already discussed.
- **import_module**: This is used to import a module and is most commonly used to load a module dynamically. In this method, you can specify whether you want to import a module using an absolute or relative path. The **import_module** function is a wrapper around **importlib.__import__**. Note that the former function brings back the package or module (for example, **packageA.module1**), which is specified with the function, while the latter function always returns the top-level package or module (for example, **packageA**).
- **importlib.util.find_spec**: This is a replaced method for the **find_loader** method, which is deprecated since Python release 3.4. This method can be used to validate whether the module exists and it is valid.

- **invalidate_caches**: This method can be used to invalidate the internal caches of finders stored at `sys.meta_path`. The internal cache is useful to load the module faster without triggering the finder methods again. But if we are dynamically importing a module, especially if it is created after the interpreter began execution, it is a best practice to call the `invalidate_caches` method. This function will clear all modules or libraries from the cache to make sure the requested module is loaded from the system path by the `import` system.
- **reload**: As the name suggests, this function is used to reload a previously imported module. We need to provide the module object as an input parameter for this function. This means the `import` function has to be done successfully. This function is very helpful when module source code is expected to be edited or changed and you want to load the new version without restarting the program.

Absolute versus relative import

We have fairly a good idea of how to use `import` statements. Now it is time to understand **absolute** and **relative** imports, especially when we are importing custom or project-specific modules. To illustrate the two concepts, let's take an example of a project with different packages, sub-packages, and modules, as shown next:

```
project
  └── pkg1
    |   ├── module1.py
    |   └── module2.py (contains a function called func1 ())
  └── pkg2
    ├── __init__.py
    ├── module3.py
    └── sub_pkg1
      └── module6.py (contains a function called func2 ())
  └── pkg3
    ├── module4.py
    ├── module5.py
    └── sub_pkg2
      └── module7.py
```

Using this project structure, we will discuss how to use absolute and relative imports.

Absolute import

We can use absolute paths starting from the top-level package and drilling down to the sub-package and module level. A few examples of importing different modules are shown here:

```
from pkg1 import module1
from pkg1.module2 import func1
```

```
from pkg2 import module3  
from pkg2.sub_pkg1.module6 import func2  
from pkg3 import module4, module5  
from pkg3.sub_pkg2 import module7
```

For absolute import statements, we must give a detailed path for each package or file, from the top-level package folder, which is similar to a file path.

Absolute imports are recommended because they are easy to read and easy to follow the exact location of imported resources. Absolute imports are least impacted by project sharing and changes in the current location of **import** statements. In fact, PEP 8 explicitly recommends the use of absolute imports.

Sometimes, however, absolute imports are quite long statements depending on the size of the project folder structure, which is not convenient to maintain.

Relative import

A relative import specifies the resource to be imported relative to the current location, which is mainly the current location of the Python code file where the **import** statement is used.

For the project examples discussed earlier, here are a few scenarios of relative import. The equivalent relative import statements are as follows:

- Scenario 1: Importing **func1** inside **module1.py**:

```
from .module2 import func1
```

We used one dot (.) only because **module2.py** is in the same folder as **module1.py**.

- Scenario 2: Importing **module4** inside **module1.py**:

```
from ..pkg3 import module4
```

In this case, we used two dots (..) because **module4.py** is in the sibling folder of **module1.py**.

- Scenario 3: Importing **Func2** inside **module1.py**:

```
from ..pkg2.sub_pkg_1.module2 import Func2
```

For this scenario, we used two dots (..) because the target module (**module2.py**) is inside a folder that is in the sibling folder of **module1.py**. We used one dot to access the **sub_pkg_1** package and another dot to access **module2**.

One advantage of relative imports is that they are simple and can significantly reduce long **import** statements. But relative **import** statements can be messy and difficult to maintain when projects are shared across teams and organizations. Relative imports are not easy to read and manage.

Loading and initializing a module

Whenever the Python interpreter interacts with an **import** or equivalent statement, it does three operations, which are described in the next sections.

Loading a module

The Python interpreter searches for the specified module on a **sys.path** (to be discussed in the *Accessing packages from any location* section) and loads the source code. This has been explained in the *Learning how import works* section.

Setting special variables

In this step, the Python interpreter defines a few special variables, such as **__name__**, which basically defines the namespace that a Python module is running in. The **__name__** variable is one of the most important variables.

In the case of our example of the **calcmain1.py**, **mycalculator.py**, and **myrandom.py** modules, the **__name__** variable will be set for each module as follows:

Module Name	__name__ =
main.py	__main__
myrandom.py	myrandom
mycalculator.py	mycalculator

Table 2.1 – The **__name__** attribute value for different modules

There are two cases of setting the **__name__** variable, which are described next.

Case A – module as the main program

If you are running your module as the main program, the **__name__** variable will be set to the **__main__** value regardless of whatever the name of the Python file or module is. For example, when **calcmain1.py** is executed, the interpreter will assign the hardcoded **__main__** string to the **__name__** variable. If we run **myrandom.py** or **mycalculator.py** as the main program, the **__name__** variable will automatically get the value of **__main__**.

Therefore, we added an **if __name__ == '__main__'** line to all main scripts to check whether this is the main execution program.

Case B – module is imported by another module

In this case, your module is not the main program, but it is imported by another module. In our example, **myrandom** and **mycalculator** are imported in **calcmain1.py**. As soon as the Python

interpreter finds the **myrandom.py** and **mycalculator.py** files, it will assign the **myrandom** and **mycalculator** names from the **import** statement to the **__name__** variable for each module. This assignment is done prior to executing the code inside these modules. This is reflected in *Table 2.1*.

Some of the other noticeable special variables are as follows:

- **__file__**: This variable contains the path to the module that is currently being imported.
- **__doc__**: This variable will output the docstring that is added in a class or a method. As discussed in [Chapter 1, Optimal Python Development Life Cycle](#), a docstring is a comment line added right after the class or method definition.
- **__package__**: This is used to indicate whether the module is a package or not. Its value can be a package name, an empty string, or **None**.
- **__dict__**: This will return all attributes of a class instance as a dictionary.
- **dir**: This is actually a method that returns every associated method or attribute as a list.
- **Locals** and **globals**: These are also used as methods that display the local and global variables as dictionary entries.

Executing the code

After the special variables are set, the Python interpreter executes the code in the file line by line. It is important to know that functions (and the code under the classes) are not executed unless they are not called by other lines of code. Here is a quick analysis of the three modules from the execution point of view when **calcmain1.py** is run:

- **mycalculator.py**: After setting the special variables, there is no code to be executed in this module at the initialization time.
- **myrandom.py**: After setting the special variables and the **import** statement, there is no further code to be executed in this module at initialization time.
- **calcmain1.py**: After setting the special variables and executing the **import** statements, it executes the following **if** statement: **if __name__ == "__main__":**. This will return **true** because we launched the **calcmain1.py** file. Inside the **if** statement, the **my_main()** function will be called, which in fact then calls methods from the **myrandom.py** and **mycalculator.py** modules.

We can add an **if __name__ == "__main__"** statement to any module regardless of whether it is the main program or not. The advantage of using this approach is that the module can be used both as a module or as a main program. There is also another application of using this approach, which is to add unit tests within the module.

Standard modules

Python comes with a library of over 200 standard modules. The exact number varies from one distribution to the other. These modules can be imported into your program. The list of these modules is very extensive but only a few commonly used modules are mentioned here as an example of standard modules:

- **math**: This module provides mathematical functions for arithmetic operations.

- **random**: This module is helpful to generate pseudo-random numbers using different types of distributions.
- **statistics**: This module offers statistics functions such as **mean**, **median**, and **variance**.
- **base64**: This module provides functions to encode and decode data.
- **calendar**: This module offers functions related to the calendar, which is helpful for calendar-based computations.
- **collections**: This module contains specialized container data types other than the general-purpose built-in containers (such as **dict**, **list**, or **set**). These specialized data types include **deque**, **Counter**, and **ChainMap**.
- **csv**: This module helps in reading from and writing to comma-based delimited files.
- **datetime**: This module offers general-purpose data and time functions.
- **decimal**: This module is specific for decimal-based arithmetic operations.
- **logging**: This module is used to facilitate logging into your application.
- **os** and **os.path**: These modules are used to access operating system-related functions.
- **socket**: This module provides low-level functions for socket-based network communication.
- **sys**: This module provides access to a Python interpreter for low-level variables and functions.
- **time**: This module offers time-related functions such as converting to different time units.

Writing reusable modules

For a module to be declared reusable, it has to have the following characteristics:

- Independent functionality
- General-purpose functionality
- Conventional coding style
- Well-defined documentation

If a module or package does not have these characteristics, it would be very hard, if not impossible, to reuse it in other programs. We will discuss each characteristic one by one.

Independent functionality

The functions in a module should offer functionality independent of other modules and independent of any local or global variables. The more independent the functions are, the more reusable the module is. If it has to use other modules, it has to be minimal.

In our example of **mycalculator.py**, the two functions are completely independent and can be reused by other programs:

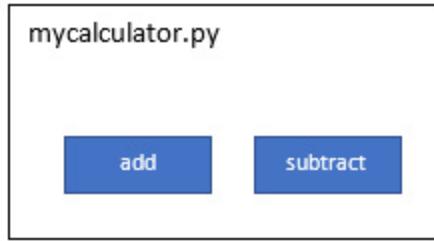


Figure 2.1 – The mycalculator module with add and subtract functions

In the case of `myrandom.py`, we are using the `random` system library to provide the functionality of generating random numbers. This is still a very reusable module because the `random` library is one of the built-in modules in Python:

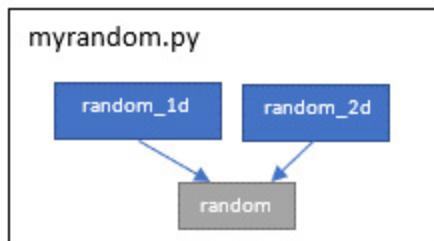


Figure 2.2 – The myrandom module with function dependency on the random library

In cases where we have to use third-party libraries in our modules, we can get into problems when sharing our modules with others if the target environment does not have the third-party libraries already installed.

To elaborate this problem further, we'll introduce a new module, `mypandas.py`, which will leverage the basic functionality of the famous `pandas` library. For simplicity, we added only one function to it, which is to print the DataFrame as per the dictionary that is provided as an input variable to the function.

The code snippet of `mypandas.py` is as follows:

```
#mypandas.py

import pandas

def print_dataframe(dict):
    """
    This function output a dictionary as a data frame """
    brics = pandas.DataFrame(dict)
    print(brics)
```

Our `mypandas.py` module will be using the `pandas` library to create a `dataframe` object from the dictionary. This dependency is shown in the next block diagram as well:

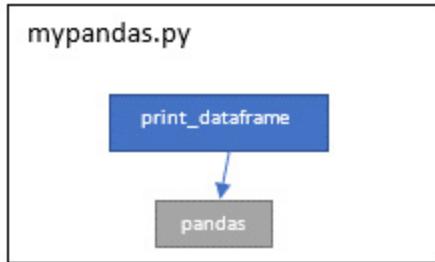


Figure 2.3 – The mypandas module with dependency on a third-party pandas library

Note that the **pandas** library is not a built-in or system library. When we try to share this module with others without defining a clear dependency on a third-party library (**pandas** in this case), the program that will try to use this module will give the following error message:

```
ImportError: No module named pandas'
```

This is why it is important that the module is as independent as possible. If we have to use third-party libraries, we need to define clear dependencies and use an appropriate packaging approach. This will be discussed in the *Sharing a package* section.

Generalization functionality

An ideal reusable module should focus on solving a general problem rather than a very specific problem. For example, we have a requirement of converting inches to centimeters. We can easily write a function that converts inches into centimeters by applying a conversion formula. What about writing a function that converts any value in the imperial system to a value in the metric system? We can have one function for different conversions that may handle inches to centimeters, feet to meters, or miles to kilometers, or separate functions for each type of these conversions. What about the reverse functions (centimeters to inches)? This may not be required now but may be required later on or by someone who is reusing this module. This generalization will make the module functionality not only comprehensive but also more reusable without extending it.

To illustrate the generalization concept, we will revise the design of the **myrandom** module to make it more general and thus more reusable. In the current design, we define separate functions for one-digit and two-digit numbers. What if we need to generate a three-digit random number or to generate a random number between 20 and 30? To generalize the requirement, we introduce a new function, **get_random**, in the same module, which takes user input for lower and upper limits of the random numbers. This newly added function is a generalization of the two random functions we already defined. With this new function in the module, the two existing functions can be removed, or they can stay in the module for convenience of use. Note that the newly added function is also offered by the **random** library out of the box; the reason for providing the function in our module is purely for

illustration of the generalized function (**get_random** in this case) versus the specific functions (**random_1d** and **random_2d** in this case).

The updated version of the **myrandom.py** module (**myrandomv2.py**) is as follows:

```
# myrandomv2.py with default and custom random functions

import random

def random_1d():

    """This function get a random number between 0 and 9"""

    return random.randint(0,9)

def random_2d():

    """This function get a random number between 10 and 99"""

    return random.randint(10,99)

def get_random(lower, upper):

    """This function get a random number between lower and\

        upper"""

    return random.randint(lower,upper)
```

Conventional coding style

This primarily focuses on how we write function names, variable names, and module names. Python has a coding system and naming conventions, which were discussed in the previous chapter of this book. It is important to follow the coding and naming conventions, especially when building reusable modules and packages. Otherwise, we will be discussing such modules as bad examples of reusable modules.

To illustrate this point, we will show the following code snippet with function and parameter names using camel case:

```
def addNumbers(numParam1, numParam2)

    #function code is omitted

Def featureCount(moduleName)

    #function code is omitted
```

If you are coming from a Java background, this code style will seem fine. But it is considered bad practice in Python. The use of the non-Pythonic style of coding makes the reusability of such modules very difficult.

Here is another snippet of a module with appropriate coding style for function names:

```
def add_numbers(num_param1, num_param2)

    #function code is omitted
```

```
Def feature_count(module_name)
    #function code is omitted
```

Another example of a good reusable coding style is illustrated in the next screenshot, which is taken from the PyCharm IDE for the **pandas** library:

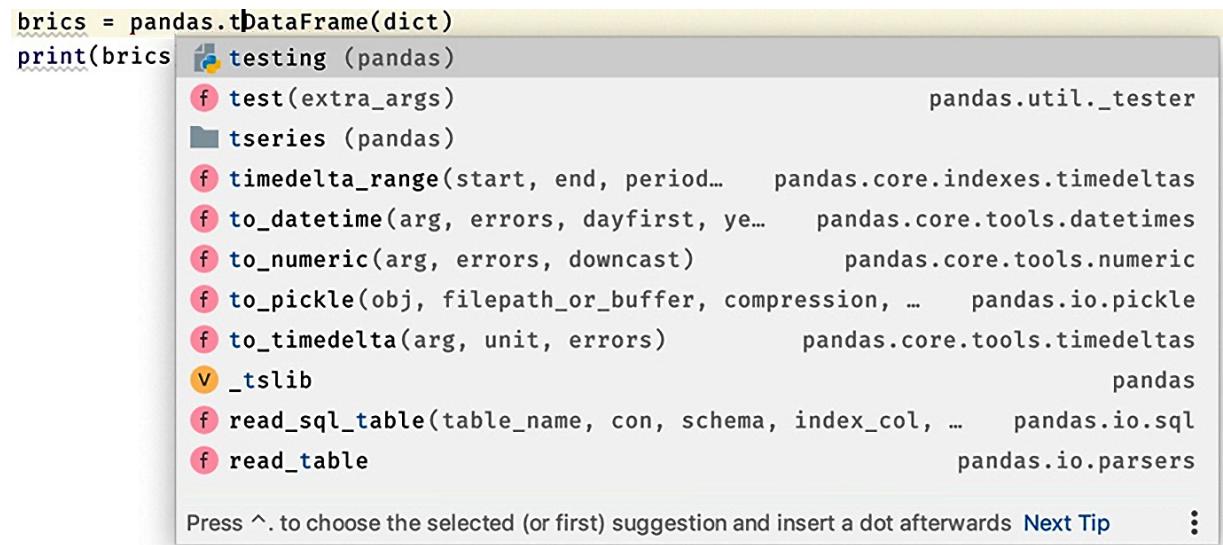


Figure 2.4 – The pandas library view in the PyCharm IDE

The functions and the variable names are easy to follow even without reading any documentation. Following a standard coding style makes the reusability more convenient.

Well-defined documentation

Well-defined and clear documentation is as important as writing a generalized and independent module with the Python coding guidelines. Without clear documentation, the module will not increase the interest of developers to reuse with convenience. But as programmers, we put more focus on the code than the documentation. Writing a few lines of documentation can make 100 lines of our code more usable and maintainable.

We will provide a couple of good examples of documentation from a module point of view by using our **mycalculator.py** module example:

```
"""mycalculator.py

This module provides functions for add and subtract of two numbers"""

def add(x, y):
    """ This function adds two numbers.

    usage: add (3, 4) """
    return x + y

def subtract(x, y):
```

```
""" This function subtracts two numbers  
usage: subtract (17, 8)  
return x - y
```

In Python, it is important to remember the following:

- We can use three quote characters to mark a string that goes across more than one line of the Python source file.
- Triple-quoted strings are used at the start of a module, and then this string is used as the documentation for the module as a whole.
- If any function starts with a triple-quoted string, then this string is used as documentation for that function.

As a general conclusion, we can make as many modules as we want by writing hundreds of lines of code, but it takes more than writing code to make a reusable module, including generalization, coding style, and most importantly, documentation.

Building packages

There are a number of techniques and tools available for creating and distributing packages. The truth is that Python does not have a great history of standardizing the packaging process. There have been multiple projects started in the first decade of the 21st century to streamline this process but not with a lot of success. In the last decade, we have had some success, thanks to the initiatives of the **Python Packaging Authority (PyPA)**.

In this section, we will be covering techniques of building packages, accessing the packages in our program, and publishing and sharing the packages as per the guidelines provided by PyPA.

We will start with package names, followed by the use of an initialization file, and then jump into building a sample package.

Naming

Package names should follow the same rule for naming as for modules, which is lowercase with no underscores. Packages act like structured modules.

Package initialization file

A package can have an optional source file named `__init__.py` (or simply an `init` file). The presence of the `init` file (even a blank one) is recommended to mark folders as packages. Since Python release 3.3 or later, the use of an `init` file is optional (PEP 420: Implicit Namespace Packages). There can be multiple purposes of using this `init` file and there is always a debate about what can go inside an `init` file versus what cannot go in. A few uses of the `init` file are discussed here:

- **Empty `__init__.py`:** This will force developers to use explicit imports and manage the namespaces as they like. As expected, developers have to import separate modules, which can be tedious for a large package.
- **Full import in `__init__.py`:** In this case, developers can import the package and then refer to the modules directly in their code using the package name or its alias name. This provides more convenience but at the expense of maintaining the list of all imports in the `__init__` file.
- **Limited import:** This is another approach in which the module developers can import only key functions in the `init` file from different modules and manage them under the package namespace. This provides the additional benefit of providing a wrapper around the underlying module's functionality. If by any chance we have to refactor the underlying modules, we have an option to keep the namespace the same, especially for API consumers. The only drawback of this approach is that it requires extra effort to manage and maintain such `init` files.

Sometimes, developers add code to the `init` file that is executed when a module is imported from a package. An example of such code is to create a session for remote systems such as a database or remote SSH server.

Building a package

Now we will discuss how to build a package with one sample package example. We will build a **masifutil** package using the following modules and a sub-package:

- The `mycalculator.py` module: We already built this module for the *Importing modules* section.
- The `myrandom.py` module: This module was also built for the *Importing modules* section.
- The **advcalc** sub-package: This will be a sub-package and will contain one module in it (`advcalculator.py`). We will define an `init` file for this sub-package but it will be empty.

The `advcalculator.py` module has additional functions for calculating the square root and log using base 10 and base 2. The source code for this module is shown next:

```
# advcalculator.py with sqrt, log and ln functions

import math

def sqrt(x):
    """This function takes square root of a number"""
    return math.sqrt(x)

def log(x):
    """This function returns log of base 10"""
    return math.log(x,10)

def ln(x):
    """This function returns log of base 2"""
    return math.log(x,2)
```

The file structure of the **masifutil** package with `init` files will look like this:

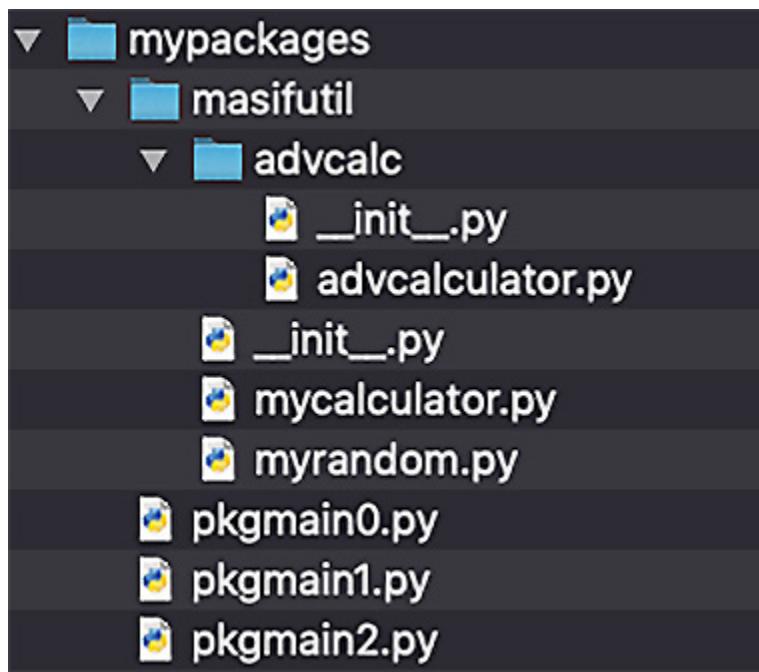


Figure 2.5 – Folder structure of the masifutil package with modules and sub-packages

In the next step, we will build a new main script (**pkgmain1.py**) to consume the modules from the package or **masifutil** subfolder. In this script, we will import the modules from the main package and sub-package using the folder structure, and then use the module functions to compute two random numbers, the sum and difference of the two numbers, and the square root and logarithmic values of the first random numbers. The source code for **pkgmain1.py** is as follows:

```
# pkgmain0.py with direct import
import masifutil.mycalculator as calc
import masifutil.myrandom as rand
import masifutil.advcalc.advcalculator as acalc
def my_main():
    """ This is a main function which generates two random\
    numbers and then apply calculator functions on them """
    x = rand.random_2d()
    y = rand.random_1d()
    sum = calc.add(x,y)
    diff = calc.subtract(x,y)
    sroot = acalc.sqrt(x)
    log10x = acalc.log(x)
    log2x = acalc.ln(x)
```

```

print("x = {}, y = {}".format(x, y))

print("sum is {}".format(sum))

print("diff is {}".format(diff))

print("square root is {}".format(sroot))

print("log base of 10 is {}".format(log10x))

print("log base of 2 is {}".format(log2x))

""" This is executed only if the special variable '__name__' is set as main"""

if __name__ == "__main__":
    my_main()

```

Here, we will be using the package name and module name to import the modules, which is cumbersome especially when we need to import the sub-packages. We can also use the following statements with the same results:

```

# mypkgmain1.py with from statements

from masifutil import mycalculator as calc

from masifutil import myrandom as rand

from masifutil.advcalc import advcalculator as acalc

#rest of the code is the same as in mypkgmain1.py

```

As mentioned earlier, the use of the empty `__init__.py` file is optional. But we have added it in this case for illustration purposes.

Next, we will explore how to add some `import` statements to the `init` file. Let's start with importing the modules inside the `init` file. In this top-level `init` file, we will import all functions as shown next:

```

#__init__ file for package 'masifutil'

from .mycalculator import add, subtract

from .myrandom import random_1d, random_2d

from .advcalc.advcalculator import sqrt, log, ln

```

Note the use of `.` before the module name. This is required for Python for the strict use of relative imports.

As a result of these three lines inside the `init` file, the new main script will become simple and the sample code is shown next:

```

# pkgmain2.py with main function

import masifutil

def my_main():

    """ This is a main function which generates two random\

```

```

numbers and then apply calculator functions on them """
x = masifutil.random_2d()
y = masifutil.random_1d()
sum = masifutil.add(x,y)
diff = masifutil.subtract(x,y)
sroot = masifutil.sqrt(x)
log10x = masifutil.log(x)
log2x = masifutil.ln(x)
print("x = {}, y = {}".format(x, y))
print("sum is {}".format(sum))
print("diff is {}".format(diff))
print("square root is {}".format(sroot))
print("log base of 10 is {}".format(log10x))
print("log base of 2 is {}".format(log2x))

""" This is executed only if the special variable '__name__' is set as main"""

if __name__ == "__main__":
    my_main()

```

The functions of the two main modules and the sub-package module are available at the main package level and the developers do not need to know the underlying hierarchy and structure of the modules within the package. This is the convenience we discussed earlier of using **import** statements inside the **init** file.

We build the package by keeping the package source code in the same folder where the main program or script resides. This works only to share the modules within a project. Next, we will discuss how to access the package from other projects and from any program from anywhere.

Accessing packages from any location

The package we built in the previous subsection is accessible only if the program calling the modules is at the same level as the package location. This requirement is not practical for code reusability and code sharing.

In this section, we will discuss a few techniques to make packages available and usable from any program on any location in our system.

Appending sys.path

This is a useful option for setting `sys.path` dynamically. Note that `sys.path` is a list of directories on which a Python interpreter searches every time it executes an `import` statement in a source program. By using this approach, we are appending (adding) paths of directories or folders containing our packages to `sys.path`.

For the `masifutil` package, we will build a new program, `pkgmain3.py`, which is a copy of `pkgmain2.py` (to be updated later) but is kept outside the folder where our `masifutil` package is residing. `pkgmain3.py` can be in any folder other than the `mypackages` folder. Here is the folder structure with a new main script (`pkgmain3.py`) and the `masifutil` package for reference:

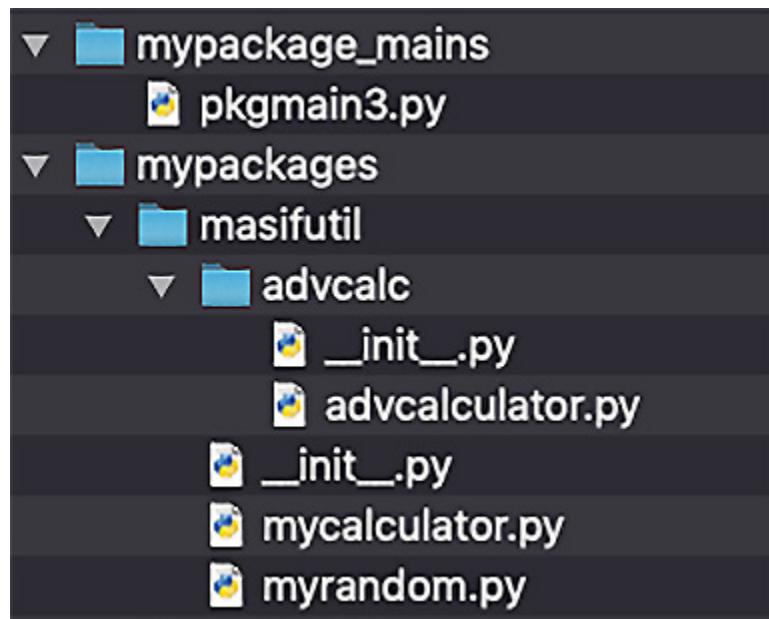


Figure 2.6 – Folder structure of the masifutil package and a new main script, `pkgmain3.py`

When we execute the `pkgmain3.py` program, it returns an error: `ModuleNotFoundError: No module named 'masifutil'`. This is expected as the path of the `masifutil` package is not added to `sys.path`. To add the package folder to `sys.path`, we will update the main program; let's name it `pkgmain4.py`, with additional statements for appending `sys.path`, which is shown next:

```
# pkgmain4.py with sys.path append code

import sys

sys.path.append('/Users/muasif/Google
Drive/PythonForGeeks/source_code/chapter2/mypackages')

import masifutil

def my_main():

    """ This is a main function which generates two random\
    numbers and then apply calculator functions on them """

```

```

x = masifutil.random_2d()
y = masifutil.random_1d()
sum = masifutil.add(x,y)
diff = masifutil.subtract(x,y)
sroot = masifutil.sqrt(x)
log10x = masifutil.log(x)
log2x = masifutil.ln(x)
print("x = {}, y = {}".format(x, y))
print("sum is {}".format(sum))
print("diff is {}".format(diff))
print("square root is {}".format(sroot))
print("log base of 10 is {}".format(log10x))
print("log base of 2 is {}".format(log2x))

""" This is executed only if the special variable '__name__' is set as main"""
if __name__ == "__main__":
    my_main()

```

After adding the additional lines of appending **sys.path**, we executed the main script without any error and with the expected console output. This is because our **masifutil** package is now available on a path where the Python interpreter can load it when we are importing it in our main script.

Alternative to appending **sys.path**, we can also use the **site.addsitedir** function from the site module. The only advantage of using this approach is that this function also looks for **.pth** files within the included folders, which is helpful for adding additional folders such as sub-packages. A snippet of a sample main script (**pktpamin5.py**) with the **addsitedir** function is shown next:

```

# pkgmain5.py
import site
site.addsitedir('/Users/muasif/Google
Drive/PythonForGeeks/source_code/chapter2/mypackages')
import masifutil
#rest of the code is the same as in pkymain4.py

```

Note that the directories we append or add using this approach are available only during the program execution. To set **sys.path** permanently (at the session or system level), the approaches that we will discuss next are more helpful.

Using the PYTHONPATH environment variable

This is a convenient way to add our package folder to **sys.path**, which the Python interpreter will use to search for the package and modules if not present in the built-in library. Depending on the operating system we are using, we can define this variable as follows.

In Windows, the environment variable can be defined using either of the following options:

- **The command line:** Set **PYTHONPATH = "C:\pythonpath1;C:\pythonpath2"**. This is good for one active session.
- **The graphical user interface:** Go to **My Computer | Properties | Advanced System Settings | Environment Variables**. This is a permanent setting.

In Linux and macOS, it can be set using **export PYTHONPATH= '/some/path/'**. If set using Bash or an equivalent terminal, the environment variable will be effective for the terminal session only. To set it permanently, it is recommended to add the environment variable at the end of a profile file, such as **~/bash_profile**.

If we execute the **pkgmain3.py** program without setting **PYTHONPATH**, it returns an error:

ModuleNotFoundError: No module named 'masifutil'. This is again expected as the path of the **masifutil** package is not added to **PYTHONPATH**.

In the next step, we will add the folder path containing **masifutil** to the **PYTHONPATH** variable and rerun the **pkgmain3** program. This time, it works without any error and with the expected console output.

Using the .pth file under the Python site package

This is another convenient way of adding packages to **sys.path**. This is achieved by defining a **.pth** file under the Python site packages. The file can hold all the folders we want to add to **sys.path**.

For illustration purposes, we created a **my.pth** file under **venv/lib/Python3.7/site-packages**. As we can see in *Figure 2.7*, we added a folder that contains our **masifutil** package. With this simple **.pth** file, our main script **pkymain3.py** program works fine without any error and with expected console output:

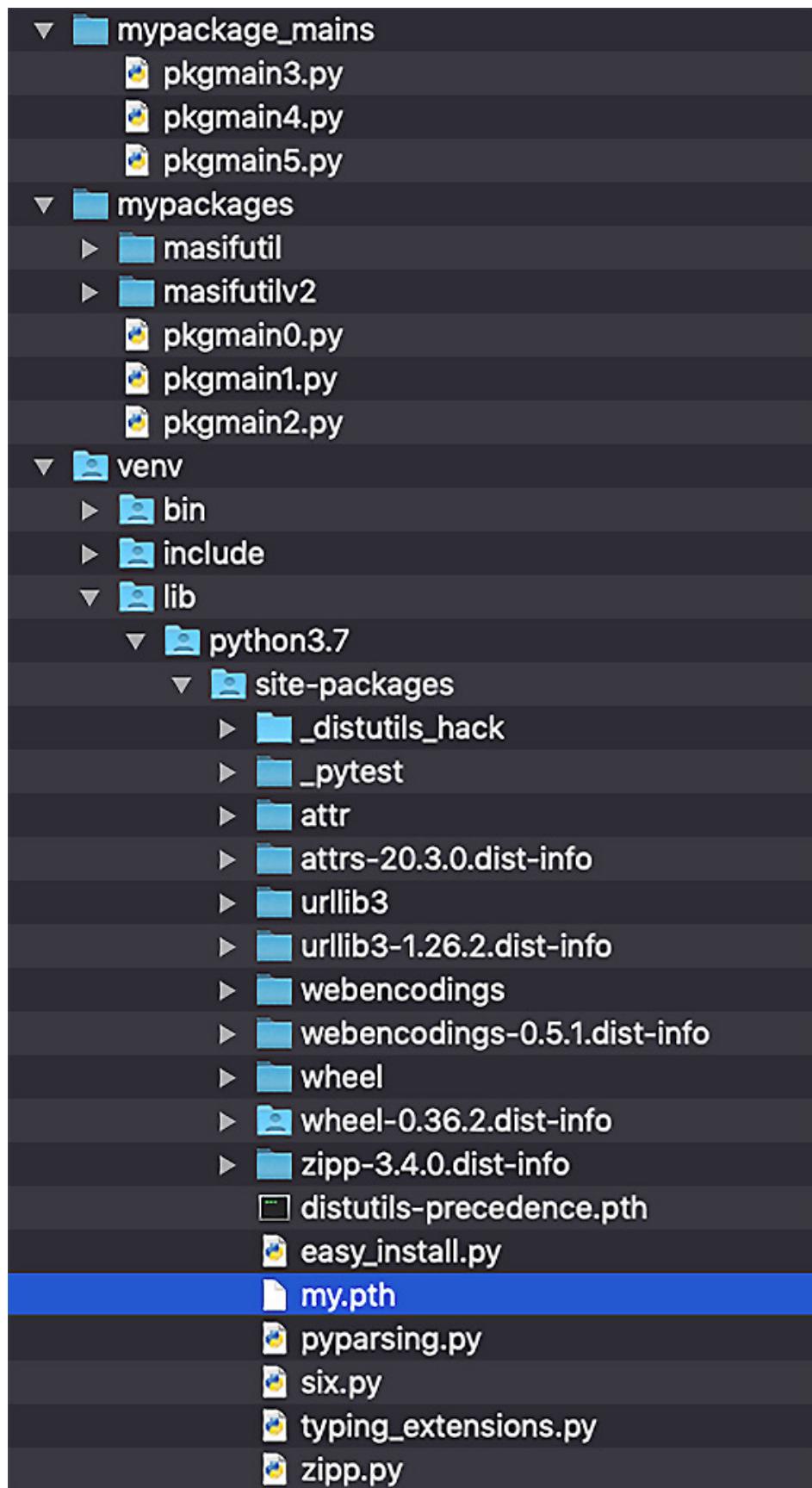


Figure 2.7 – A view of a virtual environment with the my.pth file

The approaches we discussed to access custom packages are effective to reuse the packages and modules on the same system with any program. In the next section, we will explore how to share packages with other developers and communities.

Sharing a package

To distribute Python packages and projects across communities, there are many tools available. We will focus only on the tools that are recommended as per the guidelines provided by PyPA.

In this section, we will be covering installing and distributing packaging techniques. A few tools that we will use or are at least worth mentioning in this section as a reference are as follows:

- **distutils**: This comes with Python with base functionality. It is not easy to extend for complex and custom package distribution.
- **setuptools**: This is a third-party tool and an extension of distutils and is recommended for building packages.
- **wheel**: This is for the Python packaging format and it makes installations faster and easier as compared to its predecessors.
- **pip**: pip is a package manager for Python packages and modules, and it comes as part of Python if you are installing Python version 3.4 or later. It is easy to use pip to install a new module by using a command such as **pip install <module name>**.
- **The Python Package Index (PyPI)**: This is a repository of software for the Python programming language. PyPI is used to find and install software developed and shared by the Python community.
- **Twine**: This is a utility for publishing Python packages to PyPI.

In the next subsections, we will update the **masifutil** package to include additional components as per the guidelines provided by PyPA. This will be followed by installing the updated **masifutil** package system-wide using pip. In the end, we will publish the updated **masifutil** package to **Test PyPI** and install it from Test PyPI.

Building a package as per the PyPA guidelines

PyPA recommends using a sample project for building reusable packages and it is available at <https://github.com/pypa/sampleproject>. A snippet of the sample project from the GitHub location is as shown:

File/Folder	Description	Date
.github/workflows	Don't fail when releasing an existing version (#126)	14 months ago
data	add data_files example	8 years ago
src/sample	Removed blank lines	14 months ago
tests	26 added simple module and test module	14 months ago
.gitignore	have the tests we're including actually run and pass	6 years ago
.travis.yml	Remove Python 3.5 and add 3.9 (#132)	7 months ago
LICENSE.txt	Add a LICENSE.txt file	6 years ago
MANIFEST.in	Remove Python 3.5 and add 3.9 (#132)	7 months ago
README.md	119 updated python logo (removed some other code from my own fo...)	14 months ago
pyproject.toml	Implement PEP 518 and opt into PEP 517 builds	2 years ago
setup.cfg	Drop support for EOL Python 2	15 months ago
setup.py	Remove Python 3.5 and add 3.9 (#132)	7 months ago
tox.ini	Remove Python 3.5 and add 3.9 (#132)	7 months ago

Figure 2.8 – A view of the sample project on GitHub by PyPA

We will introduce key files and folders, which are important to understand before we use them for updating our **masifutil** package:

- **setup.py**: This is the most important file, which has to exist at the root of the project or package. It is a script for building and installing the package. This file contains a global **setup()** function. The setup file also provides a command-line interface for running various commands.
- **setup.cfg**: This is an **ini** file that can be used by **setup.py** to define defaults.
- **setup() args**: The key arguments that can be passed to the setup function are as follows:
 - a) Name
 - b) Version
 - c) Description
 - d) URL
 - e) Author
 - f) License

- **README.rst/README.md**: This file (either reStructured or Markdown format) can contain information about the package or project.
- **license.txt**: The **license.txt** file should be included with every package with details of the terms and conditions of distribution. The license file is important, especially in countries where it is illegal to distribute packages without the appropriate license.
- **MANIFEST.in**: This file can be used to specify a list of additional files to include in the package. This list of files doesn't include the source code files (which are automatically included).
- **<package>**: This is the top-level package containing all the modules and packages inside it. It is not mandatory to use, but it is a recommended approach.
- **data**: This is a place to add data files if needed.
- **tests**: This is a placeholder to add unit tests for the modules.

As a next step, we will update our previous **masifutil** package as per the PyPA guidelines. Here is the new folder and file structure of the updated **masifutilv2** package:

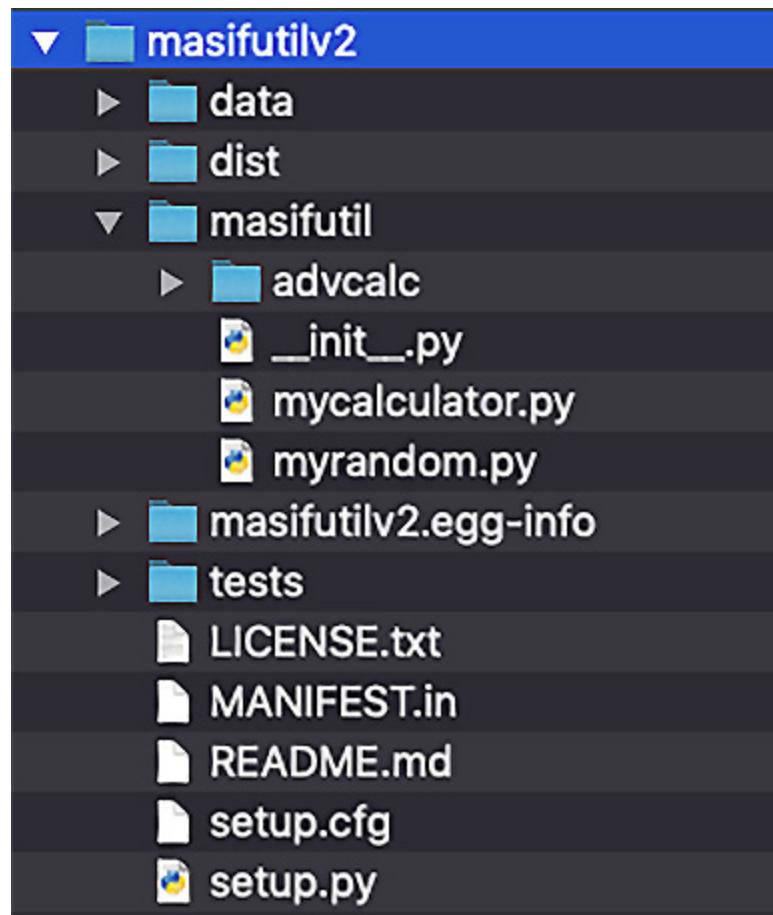


Figure 2.9 – A view of the updated masifutilv2 file structure

We have added **data** and **tests** directories, but they are actually empty for now. We will evaluate the unit tests in a later chapter to complete this topic.

The contents of most of the additional files are covered in the sample project and thus will not be discussed here, except the **setup.py** file.

We updated **setup.py** with basic arguments as per our package project. The details of the rest of the arguments are available in the sample **setup.py** file provided with the sample project by PyPA. Here is a snippet of our **setup.py** file:

```
from setuptools import setup

setup(
    name='masifutilv2',
    version='0.1.0',
    author='Muhammad Asif',
    author_email='ma@example.com',
    packages=['masifutil', 'masifutil/advcalc'],
    python_requires='>=3.5, <4',
    url='http://pypi.python.org/pypi/PackageName/',
    license='LICENSE.txt',
    description='A sample package for illustration purposes',
    long_description=open('README.md').read(),
    install_requires=[
    ],
)
```

With this **setup.py** file, we are ready to share our **masifutilv2** package locally as well as remotely, which we will discuss in the next subsections.

Installing from the local source code using pip

Once we have updated the package with new files, we are ready to install it using the pip utility. The simplest way to install it is by executing the following command with the path to the **masifutilv2** folder:

```
> pip install <path to masifutilv2>
```

The following is the console output of the command when run without installing the wheel package:

```
Processing ./masifutilv2
Using legacy 'setup.py install' for masifutilv2, since package 'wheel' is not installed.
Installing collected packages: masifutilv2
```

```
Running setup.py install for masifutilv2 ... done
```

```
Successfully installed masifutilv2-0.1.0
```

The pip utility installed the package successfully but using the egg format since the **wheel** package was not installed. Here is a view of our virtual environment after the installation:

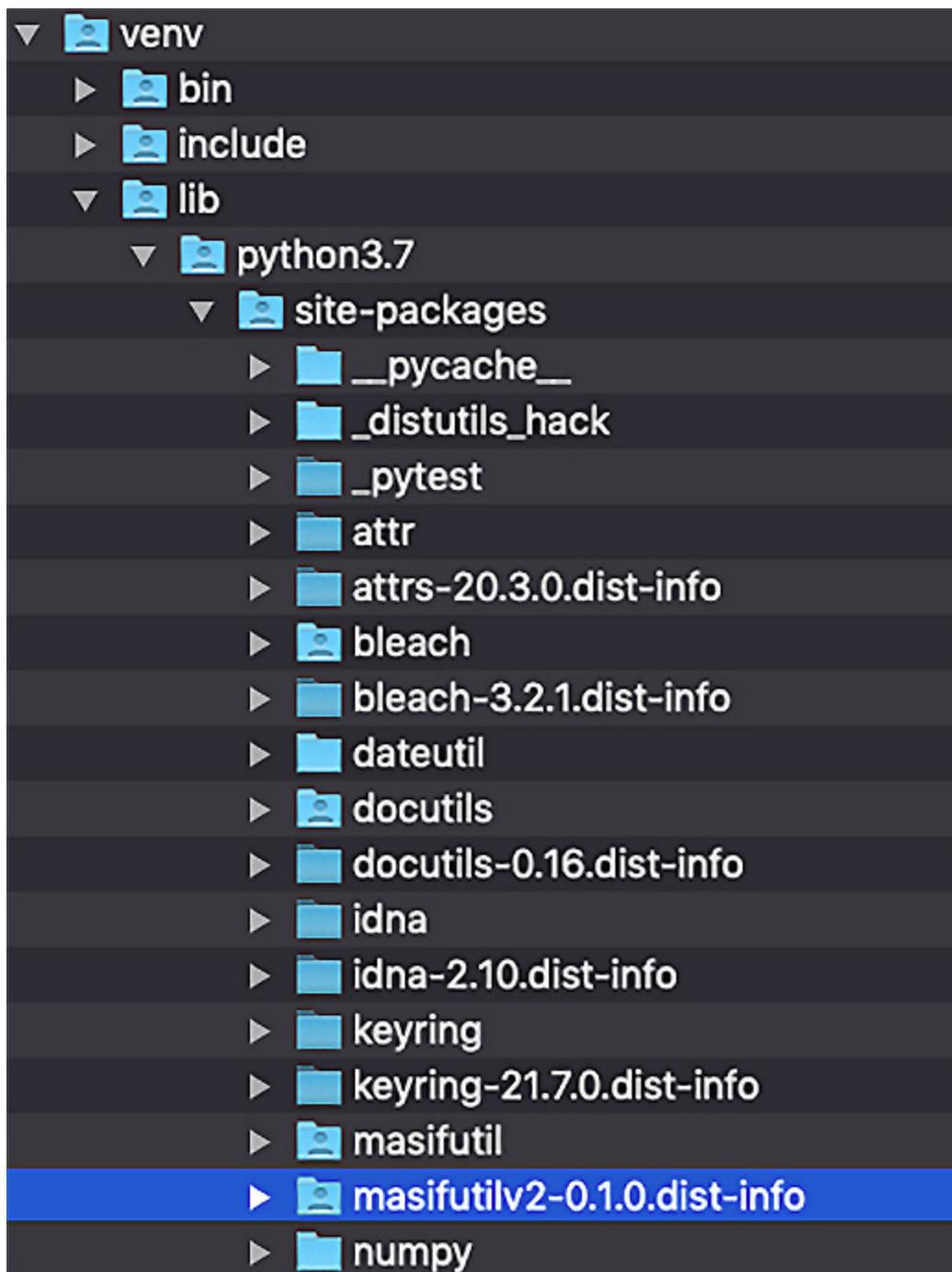


Figure 2.10 – A view of the virtual environment after installing masifutilv2 using pip

After installing the package under the virtual environment, we tested it with our **pkgmain3.py** program, which worked as expected.

TIP

*To uninstall the package, we can use **pip uninstall masifutilv2**.*

As a next step, we will install the **wheel** package and then reinstall the same package again. Here is the installation command:

```
> pip install <path to masifutilv2>
```

The console output will be similar to the following:

```
Processing ./masifutilv2
Building wheels for collected packages: masifutilv2
  Building wheel for masifutilv2 (setup.py) ... done
    Created wheel for masifutilv2: filename=masi_futilv2-0.1.0-py3-none-any.whl size=3497
sha256=038712975b7d7eb1f3fefafa799da9e294b34 e79caea24abb444dd81f4cc44b36e
  Stored in folder: /private/var/folders/xp/g88fvmgs0k90w0rc_qq4xkzxpsx11v/T/pip-ephem-
wheel-cache-l2eyp_wq/wheels/de/14/12/71b4d696301fd1052adf287191fdd054cc17ef6c9b59066277
Successfully built masifutilv2
Installing collected packages: masifutilv2
Successfully installed masifutilv2-0.1.0
```

The package is installed successfully using **wheel** this time and we can see it appears in our virtual environment as follows:

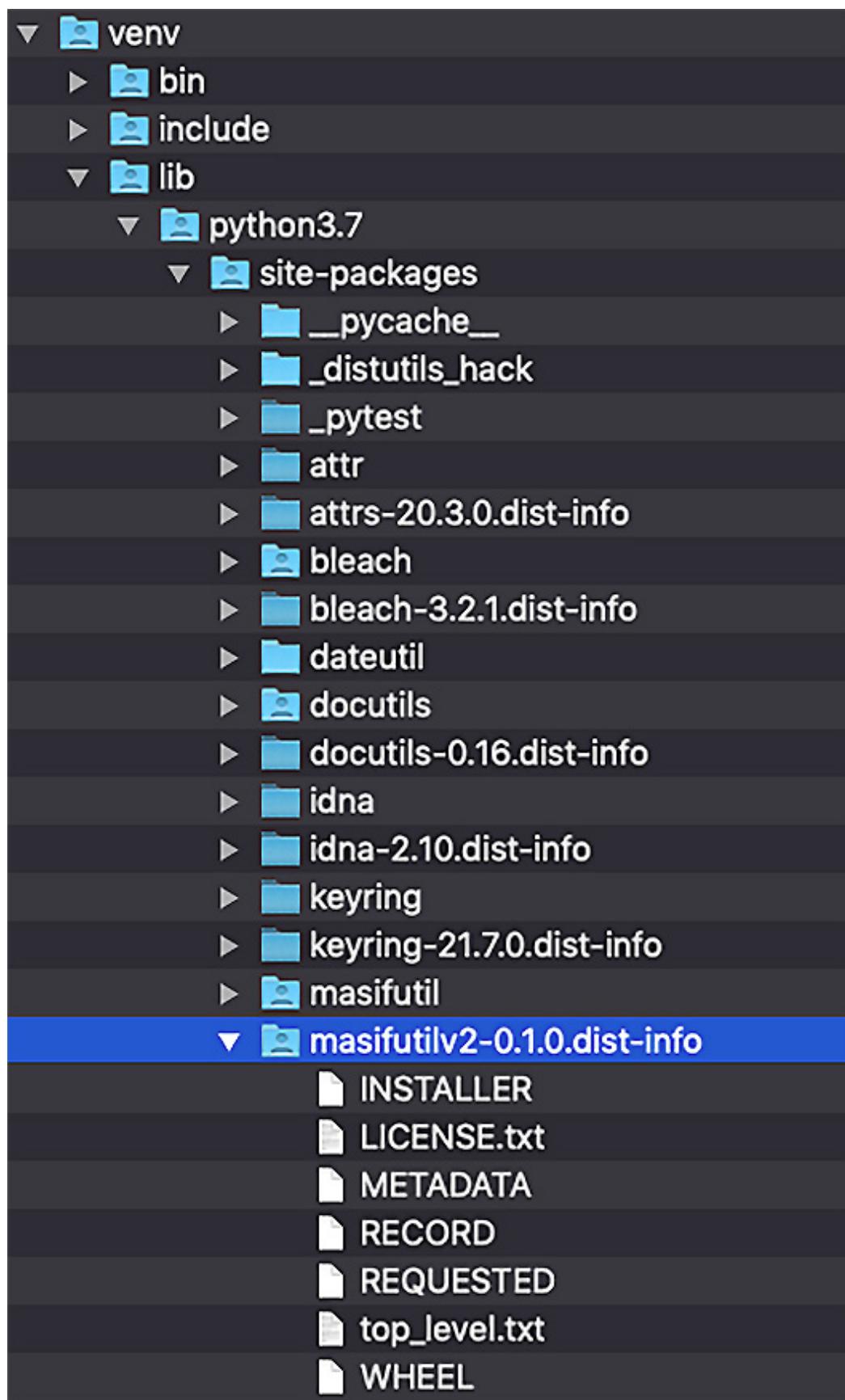


Figure 2.11 – A view of the virtual environment after installing masifutilv2 with wheel and using pip

In this section, we have installed a package using the pip utility from the local source code. In the next section, we will publish the package to a centralized repository (Test PyPI).

Publishing a package to Test PyPI

As a next step, we will add our sample package to the PyPI repository. Before executing any command for publishing our package, we will need to create an account on Test PyPI. Note that Test PyPI is a separate instance of the package index specifically for testing. In addition to the account with Test PyPI, we also need to add an **API token** to the account. We will leave the details of creating an account and adding an API token to the account for you by following the instructions available on the Test PyPI website (<https://test.pypi.org/>).

To push the package to Test PyPI, we will need the Twine utility. We assume Twine is installed using the pip utility. To upload the **masifutilv2** package, we will execute the following steps:

1. Create a distribution using the following command. This **sdist** utility will create a TAR ZIP file under a **dist** folder:

```
> python setup.py sdist
```

2. Upload the distribution file to Test PyPI. When prompted for a username and password, provide **_token_** as the username and the API token as the password:

```
> twine upload --repository testpypi dist/masifutilv2-0.1.0.tar.gz
```

This command will push the package TAR ZIP file to the Test PyPI repository and the console output will be similar to the following:

```
Uploading distributions to https://test.pypi.org/legacy/
Enter your username: _token_
Enter your password:
Uploading masifutilv2-0.1.0.tar.gz
100%|██████████| 5.15k/5.15k [00:02<00:00, 2.21kB/s]
```

We can view the uploaded file at <https://test.pypi.org/project/masifutilv2/0.1.0/> after a successful upload.

Installing the package from PyPI

Installing the package from Test PyPI is the same as installing from a regular repository, except that we need to provide the repository URL by using the **index-url** arguments. The command and the

console output will be similar to the following:

```
> pip install --index-url https://test.pypi.org/simple/ --no-deps masifutilv2
```

This command will present console output similar to the following:

```
Looking in indexes: https://test.pypi.org/simple/
Collecting masifutilv2
  Downloading https://test-
files.pythonhosted.org/packages/b7/e9/7afe390b4ec1e5842e8e62a6084505cbc6b9
f6adf0e37ac695cd23156844/masifutilv2-0.1.0.tar.gz (2.3 kB)

Building wheels for collected packages: masifutilv2
  Building wheel for masifutilv2 (setup.py) ... done
  Created wheel for masifutilv2: filename=masifutilv2- 0.1.0-py3-none-any.whl size=3497
sha256=a3db8f04b118e16ae291bad9642483874   f5c9f447dbe57c0961b5f8fb99501

  Stored in folder:
/Users/muasif/Library/Caches/pip/wheels/1c/47/29/95b9edfe28f02a605757c1
f1735660a6f79807ece430f5b836

Successfully built masifutilv2
Installing collected packages: masifutilv2
Successfully installed masifutilv2-0.1.0
```

As we can see in the console output, pip is searching for the module in Test PyPI. Once it finds the package with the name **masifutilv2**, it starts downloading and then installing it in the virtual environment.

In short, we have observed that once we create a package using the recommended format and style, then publishing and accessing the package is just a matter of using Python utilities and following the standard steps.

Summary

In this chapter, we introduced the concept of modules and packages in Python. We discussed how to build reusable modules and how they can be imported by other modules and programs. We also covered the loading and initializing of modules when included (by an import process) by other programs. In the later part of this chapter, we discussed building simple and advanced packages. We also provided a lot of code examples to access the packages, as well as installing and publishing the package for efficient reusability.

After going through this chapter, you have learned how to build modules and packages and how to share and publish the packages (and modules). These skills are important if you are working on a

project as a team in an organization or you are building Python libraries for a larger community.

In the next chapter, we will discuss the next level of modularization using object-oriented programming in Python. This will encompass encapsulation, inheritance, polymorphism, and abstraction, which are key tools to build and manage complex projects in the real world.

Questions

1. What is the difference between a module and a package?
2. What are absolute and relative imports in Python?
3. What is PyPA?
4. What is Test PyPI and why do we need it?
5. Is an **init** file a requirement to build a package?

Further reading

- *Modular Programming with Python* by Erik Westra
- *Expert Python Programming* by Michał Jaworski and Tarek Ziadé
- Python Packaging User Guide (<https://packaging.python.org/>)
- PEP 420: Implicit Namespace Packages (<https://www.python.org/dev/peps/pep-0420/>)

Answers

1. A module is meant to organize functions, variables, and classes into separate Python code files. A Python package is like a folder to organize multiple modules or sub-packages.
2. Absolute import requires the use of the absolute path of a package starting from the top level, whereas relative import is based on the relative path of the package as per the current location of the program in which the **import** statement is to be used.
3. The **Python Packaging Authority (PyPA)** is a working group that maintains a core set of software projects used in Python packaging.
4. Test PyPI is a repository of software for the Python programming language for testing purposes.
5. The **init** file is optional since Python version 3.3.

Chapter 3: Advanced Object-Oriented Python Programming

Python can be used as a declarative modular programming language such as C, as well as being used for imperative programming or full **object-oriented programming (OOP)** with programming languages such as Java. **Declarative programming** is a paradigm in which we focus on what we want to implement, while **imperative programming** is where we describe the exact steps of how to implement what we want. Python is suitable for both types of programming paradigms. OOP is a form of imperative programming in which we bundle the properties and behaviors of real-world objects into programs. Moreover, OOP also addresses the relations between different types of real-world objects.

In this chapter, we will explore how the advanced concepts of OOP can be implemented using Python. We are assuming that you are familiar with general concepts such as classes, objects, and instances and have basic knowledge of inheritance between objects.

We will cover the following topics in this chapter:

- Introducing classes and objects
- Understanding OOP principles
- Using composition as an alternative design approach
- Introducing duck typing in Python
- Learning when not to use OOP in Python

Technical requirements

These are the technical requirements for this chapter:

- You need to have Python 3.7 or later installed on your computer.
- The sample code for this chapter can be found at <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter03>.

Introducing classes and objects

A class is a blueprint for how something should be defined. It doesn't actually contain any data—it is a template that is used to create instances as per the specifications defined in a template or a blueprint.

An object of a class is an instance that is built from a class, and that is why it is also called an instance of a class. For the rest of this chapter and this book, we will refer to *object* and *instance* synonymously. Objects in OOP are occasionally represented by physical objects such as tables,

chairs, or books. On most occasions, the objects in a software program represent abstracted entities that may not be physical, such as accounts, names, addresses, and payments.

To refresh ourselves with basic concepts of classes and objects, we will define these terminologies with code examples.

Distinguishing between class attributes and instance attributes

Class attributes are defined as part of the class definition, and their values are meant to be the same across all instances created from that class. The class attributes can be accessed using the class name or instance name, although it is recommended to use a class name to access these attributes (for reading or updating). The state or data of an object is provided by **instance attributes**.

Defining a class in Python is simply done by using the **class** keyword. As discussed in [Chapter 1](#), *Optimal Python Development Life Cycle*, the name of the class should be CamelCase. The following code snippet creates a **Car** class:

```
#carexample1.py  
class Car:  
    pass
```

This class has no attributes and methods. It is an empty class, and you may think this class is useless until we add more components to it. Not exactly! In Python, you can add attributes on the fly without defining them in the class. The following snippet is a valid example of code in which we add attributes to a class instance at runtime:

```
#carexample1.py  
class Car:  
    pass  
if __name__ == "__main__":  
    car = Car ()  
    car.color = "blue"  
    car.miles = 1000  
    print (car.color)  
    print (car.miles)
```

In this extended example, we created an instance (**car**) of our **Car** class and then added two attributes to this instance: **color** and **miles**. Note that the attributes added using this approach are instance attributes.

Next, we will add class attributes and instance attributes using a constructor method (`__init__`), which is loaded at the time of object creation. A code snippet with two instance attributes (`color` and `miles`) and the `init` method is shown next:

```
#carexample2.py

class Car:

    c_mileage_units = "Mi"

    def __init__(self, color, miles):
        self.i_color = color
        self.i_mileage = miles

    if __name__ == "__main__":
        car1 = Car ("blue", 1000)
        print (car1.i_color)
        print (car1.i_mileage)
        print (car1.c_mileage_units)
        print (Car.c_mileage_units)
```

In this program, we did the following:

1. We created a `Car` class with a `c_mileage_units` class attribute and two instance variables, `i_color` and `i_mileage`.
2. We created an instance (`car`) of the `Car` class.
3. We printed out the instance attributes using the `car` instance variable.
4. We printed out the class attribute using the `car` instance variable as well as the `Car` class name. The console output is the same for both cases.

IMPORTANT NOTE

`self` is a reference to the instance that is being created. Use of `self` is common in Python to access the instance attributes and methods within the instance method, including the `init` method. `self` is not a keyword, and it is not mandatory to use the word `self`. It can be anything such as `this` or `blah`, except that it has to be the first parameter to the instance methods, but the convention of using `self` as the argument name is too strong.

We can update the class attributes using an instance variable or class name, but the outcome can be different. When we update a class attribute using the class name, it is updated for all the instances of that class. But if we update a class attribute using an instance variable, it will be updated only for that particular instance. This is demonstrated in the following code snippet, which is using the `Car` class:

```
#carexample3.py

#class definition of Class Car is same as in carexample2.py

if __name__ == "__main__":
    car1 = Car ("blue", 1000)
```

```

car2 = Car("red", 2000)

print("using car1: " + car1.c_mileage_units)
print("using car2: " + car2.c_mileage_units)
print("using Class: " + Car.c_mileage_units)

car1.c_mileage_units = "km"

print("using car1: " + car1.c_mileage_units)
print("using car2: " + car2.c_mileage_units)
print("using Class: " + Car.c_mileage_units)

Car.c_mileage_units = "NP"

print("using car1: " + car1.c_mileage_units)
print("using car2: " + car2.c_mileage_units)
print("using Class: " + Car.c_mileage_units)

```

The console output of this program can be analyzed as follows:

1. The first set of **print** statements will output the default value of the class attribute, which is **Mi**.
2. After executing the **car1.c_mileage_units = "km"** statement, the value of the class attribute will be the same (**Mi**) for the **car2** instance and the class-level attribute.
3. After executing the **Car.c_mileage_units = "NP"** statement, the value of the class attribute for **car2** and the class level will change to **NP**, but it will stay the same (**km**) for **car1** as it was explicitly set by us.

IMPORTANT NOTE

Attribute names start with c and i to indicate that they are class and instance variables, respectively, and not regular local or global variables. The name of non-public instance attributes must start with a single or double underscore to make them protected or private. This will be discussed later in the chapter.

Using constructors and destructors with classes

As with any other OOP language, Python also has constructors and destructors, but the naming convention is different. The purpose of having constructors in a class is to initialize or assign values to the class- or instance-level attributes (mainly instance attributes) whenever an instance of a class is being created. In Python, the **__init__** method is known as the constructor and is always executed when a new instance is created. There are three types of constructors supported in Python, listed as follows:

- **Default constructor:** When we do not include any constructor (the **__init__** method) in a class or forget to declare it, then that class will use a default constructor that is empty. The constructor does nothing other than initialize the instance of a class.
- **Non-parameterized constructor:** This type of constructor does not take any arguments except a reference to the instance being created. The following code sample shows a non-parameterized constructor for a **Name**: class:

```
class Name:
```

```

#non-parameterized constructor

def __init__(self):
    print("A new instance of Name class is \
created")

```

Since no arguments are passed with this constructor, we have limited functionality to add to it. For example, in our sample code, we sent a message to the console that a new instance has been created for the **Name** class

- **Parameterized constructor:** A parametrized constructor can take one or more arguments, and the state of the instance can be set as per the input arguments provided through the constructor method. The **Name** class will be updated with a parameterized constructor, as follows:

```

class Name:

    #parameterized constructor

    def __init__(self, first, last):
        self.i_first = first
        self.i_last = last

```

Destructors are the opposite of constructors—they are executed when an instance is deleted or destroyed. In Python, destructors are hardly used because Python has a garbage collector that handles the deletion of the instances that are no longer referenced by any other instance or program. If we need to add logic inside a destructor method, we can implement it by using a special `__del__` method. It is automatically called when all references of an instance are deleted. Here is the syntax of how to define a destructor method in Python:

```

def __del__(self):
    print("Object is deleted.")

```

Distinguishing between class methods and instance methods

In Python, we can define three types of methods in a class, which are described next:

- **Instance methods:** They are associated with an instance and need an instance to be created first before executing them. They accept the `self` attribute as a reference to the instance (`self`) and can read and update the state of the instance. `__init__`, which is a constructor method, is an example of an instance method.
- **Class methods:** These methods are declared with the `@classmethod` decorator. These methods don't need a class instance for execution. For this method, the class reference (`cls` is used as a convention) will be automatically sent as the first argument.
- **Static methods:** These methods are declared with the `@staticmethod` decorator. They don't have access to `cls` or `self` objects. Static methods are like utility functions that take certain arguments and provide the output based on the arguments' values—for example, if we need to evaluate certain input data or parse data for processing, we can write static methods to achieve these goals. Static methods work like regular functions that we define in modules but are available in the context of the class's namespace.

To illustrate how these methods can be defined and then used in Python, we created a simple program, which is shown next:

```
#methodsexample1.py

class Car:

    c_mileage_units = "Mi"

    def __init__(self, color, miles):
        self.i_color = color
        self.i_mileage = miles

    def print_color(self):
        print(f"Color of the car is {self.i_color}")

    @classmethod
    def print_units(cls):
        print(f"mileage unit are {cls.c_mileage_unit}")
        print(f"class name is {cls.__name__}")

    @staticmethod
    def print_hello():
        print("Hello from a static method")

if __name__ == "__main__":
    car = Car ("blue", 1000)
    car.print_color()
    car.print_units()
    car.print_hello()
    Car.print_color(car);
    Car.print_units();
    Car.print_hello()
```

In this program, we did the following:

1. We created a **Car** class with a class attribute (**c_mileage_units**), a class method (**print_units**), a static method (**print_hello**), instance attributes (**i_color** and **i_mileage**), an instance method (**print_color**), and a constructor method (**__init__**).
2. We created an instance of the **Car** class using its constructor as **car**.
3. Using the instance variable (**car** in this example), we called the instance method, the class method, and the static method.
4. Using the class name (**Car** in this example), we again triggered the instance method, the class method, and the static method. Note that we can trigger the instance method using the class name, but we need to pass the instance variable as a first argument (this also explains why we need the **self** argument for each instance method).

The console output of this program is shown next for reference:

```
Color of the car is blue
mileage unit are Mi
class name is Car
Hello from a static method
Color of the car is blue
mileage unit are Mi
class name is Car
Hello from a static method
```

Special methods

When we define a class in Python and try to print one of its instances using a **print** statement, we will get a string containing the class name and the reference of the object instance, which is the object's memory address. There is no default implementation of the **to string** functionality available with an instance or object. The code snippet showing this behavior is presented here:

```
#carexample4.py

class Car:

    def __init__(self, color, miles):
        self.i_color = color
        self.i_mileage = miles

    if __name__ == "__main__":
        car = Car ("blue", 1000)
        print (car)
```

We will get console output similar to the following, which is not what is expected from a **print** statement:

```
<__main__.Car object at 0x100caaee80>
```

To get something meaningful from a **print** statement, we need to implement a special **__str__** method that will return a string with information about the instance and that can be customized as needed. Here is a code snippet showing the **carexample4.py** file with the **__str__** method:

```
#carexample4.py

class Car:

    c_mileage_units = "Mi"

    def __init__(self, color, miles):
        self.i_color = color
```

```

    self.i_mileage = miles

def __str__(self):
    return f"car with color {self.i_color} and \
        mileage {self.i_mileage}"

if __name__ == "__main__":
    car = Car ("blue", 1000)
    print (car)

```

And the console output of the `print` statement is shown here:

```
car with color blue and mileage 1000
```

With a proper `__str__` implementation, we can use a `print` statement without implementing special functions such as `to_string()`. It is the Pythonic way to control the string conversion. Another popular method used for similar reasons is `__repr__`, which is used by a Python interpreter for inspecting an object. The `__repr__` method is more for debugging purposes.

These methods (and a few more) are called special methods or **dunders**, as they always start and end with double underscores. Normal methods should not use this convention. These methods are also known as magic **methods** in some literature, but it is not the official terminology. There are several dozen special methods available for implementation with a class. A comprehensive list of special methods is available with the official Python 3 documentation at <https://docs.python.org/3/reference/datamodel.html#specialnames>.

We reviewed the classes and the objects with code examples in this section. In the next section, we will study different object-oriented principles available in Python.

Understanding OOP principles

OOP is a way of bundling properties and behavior into a single entity, which we call objects. To make this bundling more efficient and modular, there are several principles available in Python, outlined as follows:

- Encapsulation of data
- Inheritance
- Polymorphism
- Abstraction

In the next subsections, we will study each of these principles in detail.

Encapsulation of data

Encapsulation is a fundamental concept in OOP and is also sometimes referred to as abstraction. But in reality, the encapsulation is more than the abstraction. In OOP, bundling of data and the actions associated with the data into a single unit is known as encapsulation. Encapsulation is actually more than just bundling data and the associated actions. We can enumerate three main objectives of encapsulation here, as follows:

- Encompass data and associated actions in a single unit.
- Hide the internal structure and implementation details of the object.
- Restrict access to certain components (attributes or methods) of the object.

Encapsulation simplifies the use of the objects without knowing internal details on how it is implemented, and it also helps to control updates to the state of the object.

In the next subsections, we will discuss these objectives in detail.

Encompassing data and actions

To encompass data and actions in one init, we define attributes and methods in a class. A class in Python can have the following types of elements:

- Constructor and destructor
- Class methods and attributes
- Instance methods and attributes
- **Nested** classes

We have discussed these class elements already in the previous section, except nested or **inner** classes. We already provided the Python code examples to illustrate the implementation of constructors and destructors. We have used instance attributes to encapsulate data in our instances or objects. We have also discussed the class methods, static methods, and class attributes with code examples in the previous section.

To complete the topic, we will discuss the following Python code snippet with a nested class. Let's take an example of our **Car** class and an **Engine** inner class within it. Every car needs an engine, so it makes sense to make it a nested or inner class:

```
#carwithinnerexample1.py

class Car:

    """outer class"""

    c_mileage_units = "Mi"
```

```

def __init__(self, color, miles, eng_size):
    self.i_color = color
    self.i_mileage = miles
    self.i_engine = self.Engine(eng_size)

def __str__(self):
    return f"car with color {self.i_color}, mileage \
{self.i_mileage} and engine of {self.i_engine}"

class Engine:
    """inner class"""

    def __init__(self, size):
        self.i_size = size

    def __str__(self):
        return self.i_size

if __name__ == "__main__":
    car = Car ("blue", 1000, "2.5L")
    print(car)
    print(car.i_engine.i_size)

```

In this example, we defined an **Engine** inner class inside our regular **Car** class. The **Engine** class has only one attribute—**i_size**, the constructor method (**__init__**), and the **__str__** method. For the **Car** class, we updated the following as compared to our previous examples:

- The **__init__** method includes a new attribute for engine size, and a new line has been added to create a new instance of **Engine** associated with the **Car** instance.
- The **__str__** method of the **Car** class includes the **i_size** inner class attributes in it.

The main program is using a **print** statement on the **Car** instance and also has a line to print the value of the **i_size** attribute of the **Engine** class. The console output of this program will be similar to what is shown here:

```

car with color blue, mileage 1000 and engine of 2.5L
2.5L

```

The console output of the main program shows that we have access to the inner class from within the class implementation and we can access the inner class attributes from outside.

In the next subsection, we will discuss how we can hide some of the attributes and methods to not be accessible or visible from outside the class.

Hiding information

We have seen in our previous code examples that we have access to all class-level as well as instance-level attributes without any restrictions. Such an approach led us to a flat design, and the class will simply become a wrapper around the variables and methods. A better object-oriented design approach is to hide some of the instance attributes and make only the necessary attributes visible to the outside world. To discuss how this is achieved in Python, we introduce two terms: **private** and **protected**.

Private variables and methods

A private **variable** or attribute can be defined by using a double *underscore* as a prefix before a variable name. In Python, there is no keyword such as *private*, as we have in other programming languages. Both class and instance variables can be marked as private.

A private **method** can also be defined by using a double *underscore* before a method name. A private method can only be called within the class and is not available outside the class.

Whenever we define an attribute or a method as private, the Python interpreter doesn't allow access for such an attribute or a method outside of the class definition. The restriction also applies to subclasses; therefore, only the code within a class can access such attributes and methods.

Protected variables and methods

A **protected** variable or a method can be marked by adding a *single underscore* before the attribute name or the method name. A protected variable or method *should* be accessed or used by the code written within the class definition and within subclasses—for example, if we want to convert the **i_color** attribute from a public to a protected attribute, we just need to change its name to **_i_color**. The Python interpreter does not enforce this usage of the protected elements within a class or subclass. It is more to honor the naming convention and use or access the attribute or methods as per the definition of the protected variables and methods.

By using private and protected variables and methods, we can hide some of the details of the implementation of an object. This is helpful, enabling us to have a tight and clean source code inside a large-sized class without exposing everything to the outside world. Another reason for hiding attributes is to control the way they can be accessed or updated. This is a topic for the next subsection. To conclude this section, we will discuss an updated version of our **Car** class with private and protected variables and a private method, which is shown next:

```
#carexample5.py  
class Car:  
    c_mileage_units = "Mi"
```

```

__max_speed = 200

def __init__(self, color, miles, model):
    self.i_color = color
    self.i_mileage = miles
    self.__no_doors = 4
    self._model = model

def __str__(self):
    return f"car with color {self.i_color}, mileage {self.i_mileage}, model {self._model} and doors {self.__doors()}""

def __doors(self):
    return self.__no_doors

if __name__ == "__main__":
    car = Car ("blue", 1000, "Camry")
    print (car)

```

In this updated **Car** class, we have updated or added the following as per the previous example:

- A private **__max_speed** class variable with a default value
- A private **__no_doors** instance variable with a default value inside the **__init__** constructor method
- A **_model** protected instance variable, added for illustration purposes only
- A **__doors()** private instance method to get the number of doors
- The **__str__** method is updated to get the door by using the **__doors()** private method

The console output of this program works as expected, but if we try to access any of the private methods or private variables from the main program, it is not available, and the Python interpreter will throw an error. This is as per the design, as the intended purpose of these private variables and private methods is to be only available within a class.

IMPORTANT NOTE

Python does not really make the variables and methods private, but it pretends to make them private. Python actually mangles the variable names with the class name so that they are not easily visible outside the class that contains them.

For the **Car** class example, we can access the private variables and private methods. Python provides access to these attributes and methods outside of the class definition with a different attribute name that is composed of a leading underscore, followed by the class name, and then a private attribute name. In the same way, we can access the private methods as well.

The following lines of codes are valid but not encouraged and are against the definition of private and protected:

```
print (Car._Car__max_speed)
print (car._Car__doors())
print (car._model)
```

As we can see, **_Car** is appended before the actual private variable name. This is done to minimize the conflicts with variables in inner classes as well.

Protecting the data

We have seen in our previous code examples that we can access the instance attributes without any restrictions. We also implemented instance methods and we have no restriction on the use of these. We emulate to define them as private or protected, which works to hide the data and actions from the outside world.

But in real-world problems, we need to provide access to the variables in a way that is controllable and easy to maintain. This is achieved in many object-oriented languages through **access modifiers** such as getters and setters, which are defined next:

- **Getters:** These are methods used to access the private attributes from a class or its instance
- **Setters:** These are methods used to set the private attributes of a class or its instance.

Getters and setters methods can also be used to implement additional logic of accessing or setting the attributes, and it is convenient to maintain such an additional logic in one place. There are two ways to implement the getters and setters methods: a *traditional way* and a *decorative way*.

Using traditional getters and setters

Traditionally, we write the instance methods with a **get** and **set** prefix, followed by the underscore and the variable name. We can transform our **Car** class to use the getter and setter methods for instance attributes, as follows:

```
#carexample6.py
class Car:
    __mileage_units = "Mi"
    def __init__(self, col, mil):
        self.__color = col
        self.__mileage = mil
    def __str__(self):
        return f"car with color {self.get_color()} and \\"
```

```

    mileage {self.get_mileage()}"}

def get_color(self):
    return self.__color

def get_mileage(self):
    return self.__mileage

def set_mileage (self, new_mil):
    self.__mileage = new_mil

if __name__ == "__main__":
    car = Car ("blue", 1000)
    print (car)
    print (car.get_color())
    print(car.get_mileage())
    car.set_mileage(2000)
    print (car.get_color())
    print(car.get_mileage())

```

In this updated **Car** class, we added the following:

- **color** and **mileage** instance attributes were added as private variables.
- Getter methods for **color** and **mileage** instance attributes.
- A setter method only for the **mileage** attribute because **color** usually doesn't change once it is set at the time of object creation.
- In the main program, we get data for the newly created instance of the class using getter methods. Next, we updated the mileage using a setter method, and then we got data again for the **color** and **mileage** attributes.

The console output of each statement in this example is trivial and as per expectations. As mentioned, we did not define a setter for each attribute, but only for those attributes where it makes sense and the design demands. Using getters and setters is a best practice in OOP, but they are not very popular in Python. The culture of Python developers (also known as the Pythonic way) is still to access attributes directly.

Using property decorators

Using a **decorator** to define getters and setters is a modern approach that helps to achieve the Python way of programming.

If you are into using decorators, then we have a **@property** decorator in Python to make the code simpler and cleaner. The **Car** class with traditional getters and setters is updated with decorators, and here is a code snippet showing this:

```

carexample7.py

class Car:

    __mileage_units = "Mi"

    def __init__(self, col, mil):
        self.__color = col
        self.__mileage = mil

    def __str__(self):
        return f"car with color {self.color} and mileage \
{self.mileage}"

@property
def color(self):
    return self.__color

@property
def mileage(self):
    return self.__mileage

@mileage.setter
def mileage (self, new_mil):
    self.__mileage = new_mil

if __name__ == "__main__":
    car = Car ("blue", 1000)
    print (car)
    print (car.color)
    print(car.mileage)
    car.mileage = 2000
    print (car.color)
    print(car.mileage)

```

In this updated class definition, we updated or added the following:

- Instance attributes as private variables
- Getter methods for **color** and **mileage** by using the name of the attribute as the method name and using **@property**
- Setter methods for **mileage** using the **@mileage.setter** decorator, giving the method the same name as the name of the attribute

In the main script, we access the color and the mileage attributes by using the instance name followed by a dot and the attribute name (the Pythonic way). This makes the code syntax concise and readable. The use of decorators also makes the name of the methods simpler.

In conclusion, we discussed all aspects of encapsulation in Python, using classes for the bundling of data and actions, hiding unnecessary information from the outside world of a class, and how to protect data in a class using getters, setters, and property features of Python. In the next section, we will discuss how inheritance is implemented in Python.

Extending classes with inheritance

The concept of inheritance in OOP is similar to the concept of inheritance in the real world, where children inherit some of the characteristics from their parents on top of their own characteristics.

Similarly, a class can inherit elements from another class. These elements include attributes and methods. The class from which we inherit another class is commonly known as a parent class, a **superclass**, or a **base** class. The class we inherit from another class is called a **derived class**, a **child class**, or a **subclass**. The following screenshot shows a simple relationship between a parent class and a child class:



Figure 3.1 – Parent-and-child class relationship

In Python, when a class inherits from another class, it typically inherits all the elements that compose the parent class, but this can be controlled by using naming conventions (such as double underscore) and access modifiers.

Inheritance can be of two types: **simple** or **multiple**. We will discuss these in the next sections.

Simple inheritance

In simple or basic inheritance, a class is derived from a single parent. This is a commonly used inheritance form in OOP and is closer to the family tree of human beings. The syntax of a parent class and a child class using simple inheritance is shown next:

```
class BaseClass:  
    <attributes and methods of the base class >  
class ChildClass (BaseClass):
```

```
<attributes and methods of the child class >
```

For this simple inheritance, we will modify our example of the **Car** class so that it is derived from a **Vehicle** parent class. We will also add a **Truck** child class to elaborate on the concept of inheritance. Here is the code with modifications:

```
#inheritance1.py

class Vehicle:

    def __init__(self, color):
        self.i_color = color

    def print_vehicle_info(self):
        print(f"This is vehicle and I know my color is \
{self.i_color}")

class Car (Vehicle):

    def __init__(self, color, seats):
        self.i_color = color
        self.i_seats = seats

    def print_me(self):
        print( f"Car with color {self.i_color} and no of \
seats {self.i_seats}")

class Truck (Vehicle):

    def __init__(self, color, capacity):
        self.i_color = color
        self.i_capacity = capacity

    def print_me(self):
        print( f"Truck with color {self.i_color} and \
loading capacity {self.i_capacity} tons")

if __name__ == "__main__":
    car = Car ("blue", 5)
    car.print_vehicle_info()
    car.print_me()
    truck = Truck("white", 1000)
    truck.print_vehicle_info()
    truck.print_me()
```

In this example, we created a **Vehicle** parent class with one **i_color** attribute and one **print_vehicle_info** method. Both the elements are a candidate for inheritance. Next, we created two child classes, **Car** and **Truck**. Each child class has one additional attribute (**i_seats** and **i_capacity**) and one additional method (**print_me**). In the **print_me** methods in each child class, we access the parent class instance attribute as well as child class instance attributes.

This design was intentional, to elaborate the idea of inheriting some elements from the parent class and adding some elements of its own in a child class. The two child classes are used in this example to demonstrate the role of inheritance toward reusability.

In our main program, we created **Car** and **Truck** instances and tried to access the parent method as well as the instance method. The console output of this program is as expected and is shown next:

```
This is vehicle and I know my color is blue  
Car with color blue and no of seats 5  
This is vehicle and I know my color is white  
Truck with color white and loading capacity 1000 tons
```

Multiple inheritance

In multiple inheritance, a child class can be derived from multiple parents. The concept of multiple inheritance is applicable in advanced object-oriented designs where the objects have relationships with multiple objects, but we must be careful when inheriting from multiple classes, especially if those classes are inherited from a common superclass. This can lead us to problems such as the diamond problem. The diamond problem is a situation when we create an **X** class by inheriting from two classes, **Y** and **Z**, and the **Y** and **Z** classes are inherited from a common class, **A**. The **X** class will have ambiguity about the common code of the **A** class, which it inherits from classes **Y** and **Z**. Multiple inheritance is not encouraged because of the possible issues it can bring with it.

To illustrate the concept, we will modify our **Vehicle** and **Car** classes and we will add an **Engine** class as one of the parents. The complete code with multiple inheritance of classes is shown in the following snippet:

```
#inheritance2.py  
  
class Vehicle:  
  
    def __init__(self, color):  
        self.i_color = color  
  
    def print_vehicle_info(self):  
        print( f"This is vehicle and I know my color is \\"
```

```

        {self.i_color}")

class Engine:
    def __init__(self, size):
        self.i_size = size
    def print_engine_info(self):
        print(f"This is Engine and I know my size is \
        {self.i_size}")

class Car (Vehicle, Engine):
    def __init__(self, color, size, seat):
        self.i_color = color
        self.i_size = size
        self.i_seat = seat
    def print_car_info(self):
        print(f"This car of color {self.i_color} with \
        seats {self.i_seat} with engine of size \
        {self.i_size}")
if __name__ == "__main__":
    car = Car ("blue", "2.5L", 5 )
    car.print_vehicle_info()
    car.print_engine_info()
    car.print_car_info()

```

In this multiple inheritance example, we created two parent classes as a parent: **Vehicle** and **Engine**. The **Vehicle** parent class is the same as in the previous example. The **Engine** class has one attribute (**i_size**) and one method (**print_engine_info**). The **Car** class is derived from both **Vehicle** and **Engine** and adds one additional attribute (**i_seats**) and one additional method (**print_car_info**). In the instance method, we can access instance attributes of both parent classes.

In the main program, we created an instance of the **Car** class. With this instance, we can access the instance methods of parent classes as well as child classes. The console output of the main program is shown here and is as expected:

```

This is vehicle and I know my color is blue
Car with color blue and no of seats 5
This is vehicle and I know my color is white
Truck with color white and loading capacity 1000 tons

```

In this section, we introduced inheritance and its types as simple and multiple. Next, we will study the concept of polymorphism in Python.

Polymorphism

In its literal meaning, a process of having multiple forms is called polymorphism. In OOP, **polymorphism** is the ability of an instance to behave in multiple ways and a way to use the same method with the same name and the same arguments, to behave differently in accordance with the class it belongs to.

Polymorphism can be implemented in two ways: **method overloading** and **method overriding**. We will discuss each in the next subsections.

Method overloading

Method overloading is a way to achieve polymorphism by having multiple methods with the same name, but with a different type or number of arguments. There is no clean way to implement method overloading in Python. Two methods cannot have the same name in Python. In Python, everything is an object, including classes and methods. When we write methods for a class, they are in fact attributes of a class from the namespace perspective and thus cannot have the same name. If we write two methods with the same name, there will be no syntax error, and the second one will simply replace the first one.

Inside a class, a method can be overloaded by setting the default value to the arguments. This is not the perfect way of implementing method overloading, but it works. Here is an example of method overloading inside a class in Python:

```
#methodoverloading1.py

class Car:

    def __init__(self, color, seats):
        self.i_color = color
        self.i_seat = seats

    def print_me(self, i='basic'):
        if(i == 'basic'):
            print(f"This car is of color {self.i_color}")
        else:
            print(f"This car is of color {self.i_color} \
                  with seats {self.i_seat}")
```

```
if __name__ == "__main__":
    car = Car("blue", 5 )
    car.print_me()
    car.print_me('blah')
    car.print_me('detail')
```

In this example, we add a **print_me** method with an argument that has a default value. The default value will be used when no parameter will be passed. When no parameter is passed to the **print_me** method, the console output will only provide the color of the **Car** instance. When an argument is passed to this method (regardless of the value), we have a different behavior of this method, which is providing both the color and the number of seats of the **Car** instance. Here is the console output of this program for reference:

```
This car is of color blue
This car is of color blue with seats 5
This car is of color blue with seats 5
```

IMPORTANT NOTE

*There are third-party libraries (for example, **overload**) available that can be used to implement method overloading in a cleaner way.*

Method overriding

Having the same method name in a child class as in a parent class is known as method overriding. The implementation of a method in a parent class and a child class is expected to be different. When we call an overriding method on an instance of a child class, the Python interpreter looks for the method in the child class definition, which is the overridden method. The interpreter executes the child class-level method. If the interpreter does not find a method at a child instance level, it looks for it in a parent class. If we have to specifically execute a method in a parent class that is overridden in a child class using the child class instance, we can use the **super()** method to access the parent class-level method. This is a more popular polymorphism concept in Python as it goes hand in hand with inheritance and is one of the powerful ways of implementing inheritance.

To illustrate how to implement method overriding, we will update the **inheritance1.py** snippet by renaming the **print_vehicle_info** method name as **print_me**. As we know, **print_me** methods are already in the two child classes with different implementations. Here is the updated code with the changes highlighted:

```
#methodoverriding1.py
```

```

class Vehicle:

    def __init__(self, color):
        self.i_color = color

    def print_me(self):
        print(f"This is vehicle and I know my color is \
              {self.i_color}")

class Car (Vehicle):

    def __init__(self, color, seats):
        self.i_color = color
        self.i_seats = seats

    def print_me(self):
        print( f"Car with color {self.i_color} and no of \
              seats {self.i_seats}")

class Truck (Vehicle):

    def __init__(self, color, capacity):
        self.i_color = color
        self.i_capacity = capacity

    def print_me(self):
        print( f"Truck with color {self.i_color} and \
              loading capacity {self.i_capacity} tons")

if __name__ == "__main__":
    vehicle = Vehicle("red")
    vehicle.print_me()
    car = Car ("blue", 5)
    car.print_me()
    truck = Truck("white", 1000)
    truck.print_me()

```

In this example, we override the **print_me** method in the child classes. When we create three different instances of **Vehicle**, **Car**, and **Truck** classes and execute the same method, we get different behavior. Here is the console output as a reference:

```

This is vehicle and I know my color is red
Car with color blue and no of seats 5
Truck with color white and loading capacity 1000 tons

```

Method overriding has many practical applications in real-world problems—for example, we can inherit the built-in **list** class and can override its methods to add our functionality. Introducing a custom *sorting* approach is an example of method overriding for a **list** object. We will cover a few examples of method overriding in the next chapters.

Abstraction

Abstraction is another powerful feature of OOP and is mainly related to hide the details of the implementation and show only the essential or high-level features of an object. A real-world example is a car that we derive with the main features available to us as a driver, without knowing the real details of how the feature works and which other objects are involved to provide these features.

Abstraction is a concept that is related to encapsulation and inheritance together, and that is why we have kept this topic till the end to understand encapsulation and inheritance first. Another reason for having this as a separate topic is to emphasize the use of abstract classes in Python.

Abstract classes in Python

An abstract class acts like a blueprint for other classes. An abstract class allows you to create a set of abstract methods (empty) that are to be implemented by a child class. In simple terms, a class that contains one or more abstract methods is called an abstract **class**. On the other hand, an abstract **method** is one that only has a declaration but no implementation.

There can be methods in an abstract class that are already implemented and that can be leveraged by a child class (*as is*) using inheritance. The concept of abstract classes is useful to implement common interfaces such as **application programming interfaces (APIs)** and also to define a common code base in one place that can be reused by child classes.

TIP

Abstract classes cannot be instantiated.

An abstract class can be implemented using a Python built-in module called **Abstract Base Classes (ABC)** from the **abc** package. The **abc** package also includes the **Abstractmethod** module, which utilizes decorators to declare the abstract methods. A simple Python example with the use of the **ABC** module and the **abstractmethod** decorator is shown next:

```
#abstraction1.py
from abc import ABC, abstractmethod
class Vehicle(ABC):
    def hello(self):
        print(f"Hello from abstract class")
```

```

@abstractmethod

def print_me(self):
    pass

class Car (Vehicle):
    def __init__(self, color, seats):
        self.i_color = color
        self.i_seats = seats

    """It is must to implemented this method"""

    def print_me(self):
        print( f"Car with color {self.i_color} and no of \
              seats {self.i_seats}")

if __name__ == "__main__":
    # vehicle = Vehicle()      #not possible
    # vehicle.hello()

    car = Car ("blue", 5)
    car.print_me()
    car.hello()

```

In this example, we did the following:

- We made the **Vehicle** class abstract by inheriting it from the **ABC** class and also by declaring one of the methods (**print_me**) as an abstract method. We used the **@abstractmethod** decorator to declare an abstract method.
- Next, we updated our famous **Car** class by implementing the **print_me** method in it and keeping the rest of the code the same as in the previous example.
- In the main part of the program, we attempted to create an instance of the **Vehicle** class (code commented in the illustration). We created an instance of the **Car** class and executed the **print_me** and **hello** methods.

When we attempt to create an instance of the **Vehicle** class, it gives us an error like this:

```
Can't instantiate abstract class Vehicle with abstract methods print_me
```

Also, if we try to not implement the **print_me** method in the **Car** child class, we get an error. For an instance of the **Car** class, we get the expected console output from the **print_me** and **hello** methods.

Using composition as an alternative design approach

Composition is another popular concept in OOP that is again somewhat relevant to encapsulation. In simple words, composition means to include one or more objects inside an object to form a real-

world object. A class that includes other class objects is called a **composite** class, and the classes whose objects are included in a composite class are known as **component** classes. In the following screenshot, we show an example of a composite class that has three component class objects, **A**, **B**, and **C**:

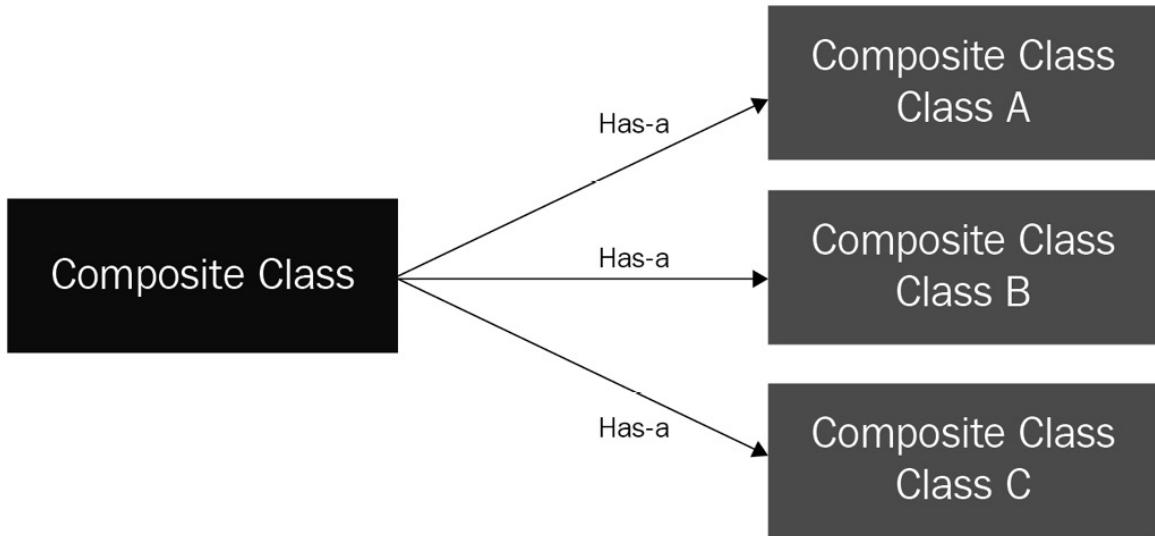


Figure 3.2 – Relationship between a composite class and its component classes

Composition is considered an alternative approach to inheritance. Both design approaches are meant to establish a relationship between objects. In the case of inheritance, the objects are tightly coupled because any changes in parent classes can break the code in child classes. On the other hand, the objects are loosely coupled in the case of composition, which facilitates changes in one class without breaking our code in another class. Because of the flexibility, the composition approach is quite popular, but this does not mean it is the right choice for every problem. How, then, can we determine which one to use for which problem? There is a rule of thumb for this. When we have an *is a* relationship between objects, inheritance is the right choice—for example, a car *is a* vehicle, and a cat *is an* animal. In the case of inheritance, a child class is an extension of a parent class, with additional functionality and the ability to reuse parent class functionality. If the relation between objects is that one object *has* another object, then it is better to use composition—for example, a car *has* a battery.

We will take our previous example of the **Car** class and the **Engine** class. In the example code for multiple inheritance, we implemented the **Car** class as a child of the **Engine** class, which is not really a good use case of inheritance. It's time to use composition by implementing the **Car** class with the **Engine** object inside the **Car** class. We can have another class for **Seat** and we can include it inside the **Car** class as well.

We will illustrate this concept further in the following example, in which we build a **Car** class by including **Engine** and **Seat** classes in it:

```
#composition1.py

class Seat:
    def __init__(self, type):
        self.i_type = type
    def __str__(self):
        return f"Seat type: {self.i_type}"

class Engine:
    def __init__(self, size):
        self.i_size = size
    def __str__(self):
        return f"Engine: {self.i_size}"

class Car:
    def __init__(self, color, eng_size, seat_type):
        self.i_color = color
        self.engine = Engine(eng_size)
        self.seat = Seat(seat_type)
    def print_me(self):
        print(f"This car of color {self.i_color} with \
{self.engine} and {self.seat}")

if __name__ == "__main__":
    car = Car ("blue", "2.5L", "leather" )
    car.print_me()
    print(car.engine)
    print(car.seat)
    print(car.i_color)
    print(car.engine.i_size)
    print(car.seat.i_type)
```

We can analyze this example code as follows:

1. We defined **Engine** and **Seat** classes with one attribute in each class: **i_size** for the **Engine** class and **i_type** for the **Seat** class.
2. Later, we defined a **Car** class by adding the **i_color** attribute, an **Engine** instance, and a **Seat** instance in it. The **Engine** and **Seat** instances were created at the time of creating a **Car** instance.
3. In this main program, we created an instance of **Car** and performed the following actions:

- a) `car.print_me`: This accesses the `print_me` method on the **Car** instance.
- b) `print(car.engine)`: This executes the `__str__` method of the **Engine** class.
- c) `print(car.seat)`: This executes the `__str__` method of the **Seat** class.
- d) `print(car.i_color)`: This accesses the `i_color` attribute of the **Car** instance.
- e) `print(car.engine.i_size)`: This accesses the `i_size` attribute of the **Engine** instance inside the **Car** instance.
- f) `print(car.seat.i_type)`: This accesses the `i_type` attribute of the **Seat** instance inside the **Car** instance

The console output of this program is shown here:

```
This car of color blue with Engine: 2.5L and Seat type: leather  
Engine: 2.5L  
Seat type: leather  
blue  
2.5L  
leather
```

Next, we will discuss duck typing, which is an alternative to polymorphism.

Introducing duck typing in Python

Duck typing, sometimes referred to as **dynamic typing**, is mostly adopted in programming languages that support dynamic typing, such as Python and JavaScript. The name *duck typing* is borrowed based on the following quote:

"If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

This means that if a bird is behaving like a duck, it will likely be a duck. The point of mentioning this quote is that it is possible to identify an object by its behavior, which is the core principle of duck typing in Python.

In duck typing, the type of class of an object is less important than the method (behavior) it defines. Using duck typing, the types of the object are not checked, but the method that is expected is executed.

To illustrate this concept, we take a simple example with three classes, **Car**, **Cycle**, and **Horse**, and we try to implement a `start` method in each of them. In the **Horse** class, instead of naming the

method **start**, we call it **push**. Here is a code snippet with all three classes and the main program at the end:

```
#ducttype1.py

class Car:
    def start(self):
        print ("start engine by ignition /battery")

class Cycle:
    def start(self):
        print ("start by pushing paddles")

class Horse:
    def push(self):
        print ("start by pulling/releasing the reins")

if __name__ == "__main__":
    for obj in Car(), Cycle(), Horse():
        obj.start()
```

In the main program, we try to iterate the instances of these classes dynamically and call the **start** method. As expected, the **obj.start()** line failed for the **Horse** object because the class does not have any such method. As we can see in this example, we can put different class or instance types in one statement and execute the methods across them.

If we change the method named **push** to **start** inside the **Horse** class, the main program will execute without any error. Duck typing has many use cases, where it simplifies the solutions. Use of the **len** method in many objects and the use of iterators are a couple of many examples. We will explore iterators in detail in the next chapter.

So far, we have reviewed different object-oriented concepts and principles and their benefits. In the next section, we will also discuss briefly when it is not very beneficial to use OOP.

Learning when not to use OOP in Python

Python has the flexibility to develop programs using either OOP languages such as Java or using declarative programming such as C. OOP is always appealing to developers because it provides powerful tools such as encapsulation, abstraction, inheritance, and polymorphism, but these tools may not fit every scenario and use case. These tools are more beneficial when used to build a large and complex application, especially one that involves **user interfaces (UIs)** and user interactions.

If your program is more like a script that has to execute certain tasks and there is no need to keep the state of objects, using OOP is overkill. Data science applications and intensive data processing are examples where it is less important to use OOP but more important to define how to execute tasks in a certain order to achieve goals. A real-world example is writing client programs for executing data-intensive jobs on a cluster of nodes, such as Apache Spark for parallel processing. We will cover these types of applications in later chapters. Here are a few more scenarios where using OOP is not necessary:

- Reading a file, applying logic, and writing back to a new file is a type of program that is easier to implement using functions in a module rather than using OOP.
- Configuring devices using Python is very popular and it is another candidate to be done using regular functions.
- Parsing and transforming data from one format to another format is also a use case that can be programmed by using declarative programming rather than OOP.
- Porting an old code base to a new one with OOP is not a good idea. We need to remember that the old code may not be built using OOP design patterns and we may end up with non-OOP functions wrapped in classes and objects that are hard to maintain and extend.

In short, it is important to analyze the problem statement and requirements first before choosing whether to use OOP or not. It also depends on which third-party libraries you will be using with your program. If you are required to extend classes from third-party libraries, you will have to go along with OOP in that case.

Summary

In this chapter, we learned the concept of classes and objects in Python and we also discussed how to build classes and use them to create objects and instances. Later, we deep-dived into the four pillars of OOP: encapsulation, inheritance, polymorphism, and abstraction. We also worked through simple and clear code examples to make it easier for readers to grasp the concepts of OOP. These four pillars are fundamental to using OOP in Python.

In the later sections, we also covered duck typing, which is important for clarifying its non-dependency on classes, before ending the chapter by reviewing when it is not significantly beneficial to use OOP.

By going through this chapter, you not only refreshed your knowledge of the main concepts of OOP but also learned how to apply the concepts using Python syntax. We will review a few Python libraries for advanced programming in the next chapter.

Questions

1. What are a class and an object?
2. What are dunders?
3. Does Python support inheriting a class from multiple classes?
4. Can we create an instance of an abstract class?
5. The type of a class is important in duck typing: true or false?

Further reading

- *Modular Programming with Python*, by Erik Westra
- *Python 3 Object-Oriented Programming*, by Dusty Phillips
- *Learning Object-Oriented Programming*, by Gaston C. Hillar
- *Python for Everyone – Third edition*, by Cay Horstmann and Rance Necaise

Answers

1. A class is a blueprint or a template to tell the Python interpreter how something needs to be defined. An object is an instance that is built from a class based on what is defined in that class.
2. Dunders are special methods that always start and end with double underscores. There are a few dozen special methods available to be implemented with every class.
3. Yes—Python supports inheriting a class from multiple classes.
4. No—we can't create an instance of an abstract class.
5. False. It is the methods that are more important than the class.

OceanofPDF.com

Section 2: Advanced Programming Concepts

We continue our journey by learning the advanced concepts of the Python language in this section. This includes a refresher of some concepts for you with an introduction to advanced subjects such as iterators, generators, errors, and exception handling. This will help you to move to the next level of programming in Python. In addition to writing Python programs, we also explore how to write and automate unit tests and integration tests using test frameworks such as unittest and pytest. In the last part of this section, we discuss some advanced function concepts for data transformation and building decorators in Python, and how to use data structures including pandas DataFrames for analytics applications.

This section contains the following chapters:

- [*Chapter 4, Python Libraries for Advanced Programming*](#)
- [*Chapter 5, Testing and Automation with Python*](#)
- [*Chapter 6, Advanced Tips and Tricks in Python*](#)

Chapter 4: Python Libraries for Advanced Programming

In previous chapters, we have discussed different approaches to building modular and reusable programs in Python. In this chapter, we will investigate a few advanced concepts of the Python programming language such as iterators, generators, logging, and error handling. These concepts are important to write efficient and reusable code. For this chapter, we assume that you are familiar with the Python language syntax and know how to write control and loop structures.

In this chapter, we will learn how loops work in Python, how files are handled and what is the best practice to open and access files, and how to handle erroneous situations, which may be expected or unexpected. We will also investigate the logging support in Python and different ways of configuring the logging system. This chapter will also help you learn how to use the advanced libraries in Python for building complex projects.

We will cover the following topics in this chapter:

- Introducing Python data containers
- Using iterators and generators for data processing
- Handling files in Python
- Handling errors and exceptions
- Using the Python **logging** module

By the end of this chapter, you will have learned how to build iterators and generators, how to handle errors and exceptions in your program, and how to implement logging for your Python project in an efficient way.

Technical requirements

The technical requirement for this chapter is that you need to have installed Python 3.7 or later on your computer. Sample code for this chapter can be found at
<https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter04>.

Let's begin by refreshing our knowledge about the data containers available in Python, which will be helpful for the follow-up topics in this chapter.

Introducing Python data containers

Python supports several data types, both numeric as well as collections. Defining numeric data types such as integers and floating-point numbers is based on assigning a value to a variable. The value we

assign to a variable determines the type of the numeric data type. Note that a specific constructor (for example, **int()** and **float()**) can also be used to create a variable of a specific data type. Container data types can also be defined either by assigning values in an appropriate format or by using a specific constructor for each collection data type. We will study five different container data types in this section: **strings**, **lists**, **tuples**, **dictionaries**, and **sets**.

Strings

Strings are not directly a container data type. But it is important to discuss the string data type because of its wide use in Python programming and also the fact that the string data type is implemented using an **immutable sequence** of Unicode code points. The fact that it uses a sequence (a collection type) makes it a candidate to be discussed in this section.

String objects are immutable objects in Python. With immutability, string objects provide a safe solution for concurrent programs where multiple functions may access the same string object and will get the same result back. This safety is not possible with mutable objects. Being immutable objects, string objects are popular to use as keys for the dictionary data type or as data elements for the set data type. The drawback of immutability is that a new instance needs to be created even if a small change is to be made to an existing string instance.

MUTABLE VERSUS IMMUTABLE OBJECTS

A mutable object can be changed after its creation, but it is not possible to change an immutable object.

String literals can be enclosed by using matching single quotes (for example, `'blah'`), double quotes (for example, `"blah blah"`), or triple single or double quotes (for example, `"""none"""` or `'''none'''`). It is also worth mentioning that string objects are handled differently in Python 3 versus Python 2. In Python 3, string objects can hold only text sequences in the form of Unicode data points, but in Python 2 they can hold text as well as byte data. In Python 3, byte data is handled by the **bytes** data type.

Separating text from bytes in Python 3 makes it clean and efficient but at the cost of data portability. The Unicode text in strings cannot be saved to disk or sent to a remote location on the network without converting it into a binary format. This conversion requires encoding the string data into a byte sequence, which can be achieved in one of the following ways:

- **Using the `str.encode(encoding, errors)` method:** This method is available on the string object and it can take two arguments. A user can provide the type of codec to be used (**UTF-8** being the default) and how to handle the errors.
- **Converting to the bytes datatype:** A string object can be converted to the **Bytes** data type by passing the string instance to the bytes constructor along with the encoding scheme and the error handling scheme.

The details of methods and the attributes available with any string object can be found in the official Python documentation as per the Python release.

Lists

The list is one of the basic collection types in Python, which is used to store multiple objects using a single variable. Lists are dynamic and *mutable*, which means the objects in a list can be changed and the list can grow or shrink.

List objects in Python are not implemented using any linked list concept but using a variable-length array. The array contains references to objects it is storing. The pointer of this array and its length are stored in the list head structure, which is kept up to date as objects are added or deleted from a list. The behavior of such an array is made to appear like a list but in reality, it is not a real list. That is why some of the operations on a Python list are not optimized. For example, inserting a new object into a list and deleting objects from a list will have a complexity of n .

To rescue the situation, Python provides a **deque** data type in the **collections** built-in module. The **deque** data type provides the functionality of stacks and queues and is a good alternative option for cases when a linked list-like behavior is demanded by a problem statement.

Lists can be created empty or with an initial value using *square brackets*. Next, we present a code snippet that demonstrates how to create an empty or non-empty list object using only the square brackets or using the list object constructor:

```
e1 = []          #an empty list
e2 = list()       #an empty list via constructor
g1 = ['a', 'b']   #a list with 2 elements
g2 = list(['a', 'b']) #a list with 2 elements using a \
                     #constructor
g3 = list(g1)     #a list created from a list
```

The details of the operations available with a list object, such as **add**, **insert**, **append**, and **delete** can be reviewed in the official Python documentation. We will introduce tuples in the next section.

Tuples

A tuple is an immutable list, which means it cannot be modified after creation. Tuples are usually used for a small number of entries and when the position and sequence of the entries in a collection is important. To preserve the sequence of entries, tuples are designed as immutable, and this is where

tuples differentiate themselves from lists. Operations on a tuple are typically faster than a regular list datatype. In cases when the values in a collection are required to be constant in a particular order, using tuples is the preferred option because of their superior performance.

Tuples are normally initialized with values because they are immutable. A simple tuple can be created using parenthesis. A few ways to create tuple instances are shown in the next code snippet:

```
w = ()                      #an empty tuple  
x = (2, 3)                  #tuple with two elements  
y = ("Hello World")         #not a tuple, Comma is required \  
                           for single entry tuple  
z = ("Hello World",)        #A comma will make it a tuple
```

In this code snippet, we created an empty tuple (**w**), a tuple with numbers (**x**), and a tuple with the text **Hello World**, which is **z**. The variable **y** is not a tuple since, for a 1-tuple (a single-object tuple), we need a trailing comma to indicate that it is a tuple.

After introducing lists and tuples, we will briefly introduce dictionaries.

Dictionaries

Dictionaries are one of the most used and versatile data types in Python. A dictionary is a collection that is used to store data values in the *key:value* format. Dictionaries are mutable and unordered data types. In other programming languages, they are referred to as *associative arrays* or *hashtables*.

A dictionary can be created using *curly brackets* with a list of *key:value* pairs. The key is separated from its value by a colon ':' and the *key:value* pairs are separated by a comma ','. A code snippet for a dictionary definition follows:

```
mydict = {  
    "brand": "BMW",  
    "model": "330i",  
    "color": "Blue"  
}
```

Duplicate keys are not allowed in a dictionary. A key must be an immutable object type such as a string, tuple, or number. The values in a dictionary can be of any data type, which even includes lists, sets, custom objects, and even another dictionary itself.

When dealing with dictionaries, three objects or lists are important:

- **Keys:** A list of keys in a dictionary is used to iterate through dictionary items. A list of keys can be obtained using the **keys()** method:

```
dict_object.keys()
```

- **Values:** Values are the objects stored against different keys. A list of value objects can be obtained using the **values()** method:

```
dict_object.values()
```

- **Items:** Items are the key-value pairs that are stored in a dictionary. A list of items can be obtained using the **items()** method:

```
dict_object.items()
```

Next, we will discuss sets, which are also key data structures in Python.

Sets

A set is a *unique* collection of objects. A set is a mutable and unordered collection. There is no duplication of objects allowed in a set. Python uses a hashtable data structure to implement uniqueness in a set, which is the same approach used to ensure the uniqueness of keys in a dictionary. The behavior of sets in Python is very similar to sets in mathematics. This data type finds its application in situations where the order of objects is not important, but their uniqueness is. This helps to test whether a certain collection contains a certain object or not.

TIP

*If the behavior of a set is required as an immutable data type, Python has a variant implementation of sets called **frozenset**.*

Creating a new set object is possible using *curly brackets* or using the set constructor (**set()**). The next code snippet shows a few examples of creating a set:

```
s1 = set()          # empty set
s2 = {}            # an empty set using curly
s3 = set(['a', 'b']) # a set created from a list with           # const.
s3 = {1,2}          # a set created using curly bracket
s4 = {1, 2, 1}      # a set will be created with only 1 and 2           #
objects. Duplicate object will be ignored
```

Accessing set objects is not possible using an indexing approach. We need to pop one object from the set like a list or we can iterate on a set to get objects one by one. Like mathematical sets, sets in Python also support operations such as *union*, *intersection*, and *difference*.

In this section, we reviewed the key concepts of strings and collection data types in Python 3, which are important to understand the upcoming topic – iterators and generators.

Using iterators and generators for data processing

Iteration is one of the key tools used for data processing and data transformation. Iterations are especially useful when dealing with large datasets and when bringing the whole dataset into memory is not possible or efficient. Iterators provide a way to bring the data into memory one item at a time.

Iterators can be created by defining them with a separate class and implementing special methods such as `__iter__` and `__next__`. But there is also a new way to create iterators using the `yield` operation, known as generators. In the next subsections, we will study both iterators and generators.

Iterators

Iterators are the objects that are used to iterate on other objects. An object on which an iterator can iterate is called **iterable**. In theory, the two objects are different, but it is possible to implement an iterator within the **iterable** object class. This is not recommended but is technically possible and we will discuss with an example why this approach is not a good design approach. In the next code snippet, we provide a few examples of using the `for` loop for iteration purposes in Python:

```
#iterator1.py

#example 1: iterating on a list
for x in [1,2,3]:
    print(x)

#example 2: iterating on a string
for x in "Python for Geeks":
    print(x, end="")

print('')

#example 3: iterating on a dictionary
week_days = {1:'Mon', 2:'Tue',
            3:'Wed', 4:'Thu',
            5:'Fri', 6:'Sat', 7:'Sun'}

for k in week_days:
    print(k, week_days[k])

#example 4: iterating on a file
for row in open('abc.txt'):
    print(row, end="")
```

In these code examples, we used different `for` loops to iterate on a list, a string, a dictionary, and a file. All these data types are iterable and thus we will be using a simple syntax with the `for` loop to

get through the items in these collections or sequences. Next, we will study what ingredients make an object iterable, which is also referred to as the **Iterator Protocol**.

IMPORTANT NOTE

Every collection in Python is iterable by default.

In Python, an iterator object must implement two special methods: `__iter__` and `__next__`. To iterate on an object, the object has to implement at least the `__iter__` method. Once the object implements the `__iter__` method, we can call the object iterable. These methods are described next:

- `__iter__`: This method returns the iterator object. This method is called at the start of a loop to get the iterator object.
- `__next__`: This method is called at each iteration of the loop and it returns the next item in the iterable object.

To explain how to build a custom object that is iterable, we will implement the **Week** class, which stores the numbers and names of all weekdays in a dictionary. This class will not be iterable by default. To make it iterable, we will add `__iter__`. To keep the example simple, we will also add the `__next__` method in the same class. Here is the code snippet with the **Week** class and the main program, which iterates to get the names of weekdays:

```
#iterator2.py

class Week:

    def __init__(self):
        self.days = {1:'Monday', 2: "Tuesday",
                    3:"Wednesday", 4: "Thursday",
                    5:"Friday", 6:"Saturday", 7:"Sunday"}
        self._index = 1

    def __iter__(self):
        self._index = 1
        return self

    def __next__(self):
        if self._index < 1 | self._index > 7 :
            raise StopIteration
        else:
            ret_value = self.days[self._index]
            self._index +=1
        return ret_value

if(__name__ == "__main__"):
    wk = Week()
```

```
for day in wk:  
    print(day)
```

We shared this code example just to demonstrate how the `__iter__` and `__next__` methods can be implemented in the same object class. This style of implementing an iterator is commonly found on the internet, but it is not a recommended approach and is considered a bad design. The reason is that when we use it in the `for` loop, we get back the main object as an iterator as we implemented `__iter__` and `__next__` in the same class. This can give unpredictable results. We can demonstrate this by executing the following code snippet for the same class, `Week`:

```
#iterator3.py  
  
class Week:  
  
    #class definition is the same as shown in the previous \  
    code example  
  
    if(__name__ == "__main__"):  
        wk = Week()  
        iter1 = iter(wk)  
        iter2 = iter(wk)  
        print(iter1.__next__())  
        print(iter2.__next__())  
        print(next(iter1))  
        print(next(iter2))
```

In this new main program, we are iterating on the same object using two different iterators. The results of this main program are not as expected. This is due to a common `_index` attribute shared by the two iterators. Here is a console output as a reference:

```
Monday  
Tuesday  
Wednesday  
Thursday
```

Note that in this new main program we deliberately did not use a `for` loop. We created two iterator objects for the same object of the `Week` class using the `iter` function. The `iter` function is a Python standard function that calls the `__iter__` method. To get the next item in the iterable object, we directly used the `__next__` method as well as the `next` function. The `next` function is also a general function, like the `iter` function. This approach of using an iterable as an iterator is also not considered thread-safe.

The best approach is always to use a separate iterator class and always create a new instance of an iterator through the `__iter__` method. Each iterator instance has to manage its own internal state. A revised version of the same code example of the `Week` class is shown next with a separate iterator class:

```
#iterator4.py

class Week:
    def __init__(self):
        self.days = {1: 'Monday', 2: "Tuesday",
                    3: "Wednesday", 4: "Thursday",
                    5: "Friday", 6: "Saturday", 7: "Sunday"}

    def __iter__(self):
        return WeekIterator(self.days)

class WeekIterator:
    def __init__(self, dayss):
        self.days_ref = dayss
        self._index = 1

    def __next__(self):
        if self._index < 1 | self._index > 8:
            raise StopIteration
        else:
            ret_value = self.days_ref[self._index]
            self._index +=1
        return ret_value

if(__name__ == "__main__"):
    wk = Week()
    iter1 = iter(wk)
    iter2 = iter(wk)
    print(iter1.__next__())
    print(iter2.__next__())
    print(next(iter1))
    print(next(iter2))
```

In this revised code example, we have a separate iterator class with the `__next__` method and it has its own `_index` attribute for managing the iterator state. The iterator instance will have a reference to the container object (dictionary). The console output of the revised example gives the results as expected: each iterator is iterating separately on the same instance of the `Week` class. The console output is shown next as a reference:

Monday

Monday

Tuesday

Tuesday

In short, to create an iterator, we need to implement the `__iter__` and `__next__` methods, manage internal state, and raise a `StopIteration` exception when there are no values available. Next, we will study generators, which will simplify the way we return iterators.

Generators

A generator is a simple way of returning an iterator instance that can be used for iteration, which is achieved by implementing only a generator function. A generator function is similar to a normal function but with a `yield` statement instead of a `return` statement in it. The `return` statement is still allowed in a generator function but will not be used to return the next item in an iterable object.

By definition, a function will be a generator function if it has at least one `yield` statement in it. The main difference when using the `yield` statement is that it pauses the function and saves its internal state, and when the function is called next time, it starts from the line it yielded the last time. This design pattern makes the iterator functionality simple and efficient.

Internally, methods such as `__iter__` and `__next__` are implemented automatically and the `StopIteration` exception is also raised automatically. The local attributes and their values are preserved between the successive calls and there is no additional logic to be implemented by the developer. The Python interpreter provides all this functionality whenever it identifies a generator function (a function with a `yield` statement in it).

To understand how the generator works, we will start with a simple generator example that is used to generate a sequence of the first three letters of the alphabet:

```
#generators1.py
def my_gen():
    yield 'A'
    yield 'B'
```

```

yield 'C'

if(__name__ == "__main__"):

    iter1 = my_gen()

    print(iter1.__next__())
    print(next(iter1))
    print(iter1.__next__())

```

In this code example, we implemented a simple generator function using three **yield** statements without a **return** statement. In the main part of the program, we did the following:

1. We called the generator function, which returns us an iterator instance. At this stage, no line inside the **my_gen()** generator function is executed.
2. Using the iterator instance, we called the **__next__** method, which starts the execution of the **my_gen()** function, pauses after executing the first **yield** statement, and returns **A**.
3. Next, we call the **next()** function on the iterator instance. The result is the same as we get with the **__next__** method. But this time, the **my_gen()** function starts the execution from the next line from where it paused the last time because of the **yield** statement. The next line is another **yield** statement, which results in another pause after returning the letter **B**.
4. The next **__next__** method will result in the execution of the next **yield** statement, which will return the letter **C**.

Next, we will revisit the **Week** class and its iterator implementation and will use a generator instead of an iterator class. The sample code is presented next:

```

#generator2.py

class Week:

    def __init__(self):
        self.days = {1:'Monday', 2: "Tuesday",
                    3:"Wednesday", 4: "Thursday",
                    5:"Friday", 6:"Saturday", 7:"Sunday"}

    def week_gen(self):
        for x in self.days:
            yield self.days[x]

if(__name__ == "__main__"):

    wk = Week()
    iter1 = wk.week_gen()
    iter2 = iter(wk.week_gen())
    print(iter1.__next__())
    print(iter2.__next__())
    print(next(iter1))

```

```
print(next(iter2))
```

In comparison to **iterator4.py**, the implementation of the **Week** class with a generator is way simpler and cleaner and we can achieve the same results. This is the power of generators and that is why they are very popular in Python. Before concluding this topic, it is important to highlight a few other key features of generators:

- **Generator expressions:** Generator expressions can be used to create simple generators (also known as **anonymous functions**) on the fly without writing a special method. The syntax is similar to list comprehension except we use parentheses instead of square brackets. The next code example (an extension of the example we introduced for list comprehension) shows how a generator expression can be used to create a generator, its usage, and also a comparison with list comprehension:

```
#generator3.py
```

```
L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]  
f1 = [x+1 for x in L]  
g1 = (x+1 for x in L)  
print(g1.__next__())  
print(g1.__next__())
```

- **Infinite streams:** Generators can also be used to implement an infinite stream of data. It is always a challenge to bring an infinite stream into memory, which is solved easily with generators as they return only one data item at a time.
- **Pipelining generators:** When working with complex problems, multiple generators can be used as a pipeline to achieve any goals. The concept of pipelining multiple generators can be explained with an example. We take on a problem, which is to take the sum of the squares of prime numbers. This problem can be solved with traditional **for** loops, but we will try to solve it using two generators: the **prime_gen** generator for generating prime numbers and the **x2_gen** generator for taking the square of the prime numbers fed to this generator by the **prime_gen** generator. We feed the two generators pipelined into the **sum** function to get the desired result. Here is the code snippet for this problem solution:

```
#generator4.py
```

```
def prime_gen(num):  
    for cand in range(2, num+1):  
        for i in range(2, cand):  
            if (cand % i) == 0:  
                break  
        else:  
            yield cand  
  
def x2_gen(list2):  
    for num in list2:  
        yield num*num  
  
print(sum(x2_gen(prime_gen(5))))
```

Generators operate on an on-demand basis, which makes them not only memory efficient but also provides a way to generate values when they are needed. This helps to avoid unnecessary data generation, which may not be used at all. Generators are well suited to be used for a large amount of data processing, for piping the data from one function to others, and to simulate concurrency as well.

In the next section, we will investigate how to handle files in Python.

Handling files in Python

Reading data from a file or writing data to a file is one of the fundamental operations supported by any programming language. Python provides extensive support for handling file operations, which are mostly available in its standard library. In this section, we will discuss core file operations such as opening a file, closing a file, reading from a file, writing to a file, file management with context managers, and opening multiple files with one handle using the Python standard library. We will start our discussion with file operations in the next subsection.

File operations

File operations typically start with opening a file and then reading or updating the contents in that file. The core file operations are as follows:

Opening and closing a file

To apply any read or update operation to a file, we need a pointer or reference to the file. A file reference can be obtained by opening a file using the built-in **open** function. This function returns a reference to the **file** object, which is also known as a **file handle** in some literature. The minimum requirement with the **open** function is the name of the file with an absolute or relative path. One optional parameter is the access mode to indicate in which mode a file is to be opened. The access mode can be **read**, **write**, **append**, or others. A full list of access mode options is as follows:

- **r**: This option is for opening a file in read-only mode. This is a default option if the access mode option is not provided:
`f = open ('abc.txt')`
- **a**: This option is for opening a file to append a new line at the end of the file:
`f = open ('abc.txt', 'a')`
- **w**: This option is for opening a file for writing. If a file does not exist, it will create a new file. If the file exists, this option will override it and any existing contents in that file will be destroyed:
`f = open ('abc.txt', 'w')`
- **x**: This option is for opening a file for exclusive writing. If the file already exists, it will throw an error:
`f = open ('abc.txt', 'x')`

- **t**: This option is for opening a file in text mode. This is the default option.
 - **b**: This option is for opening a file in binary mode.
 - **+**: This option is for opening a file for reading and writing:
- ```
f = open ('abc.txt', 'r+')
```

The mode options can be combined to get multiple options. In addition to the filename and the access mode options, we can also pass the encoding type, especially for text files. Here is an example of opening a file with **utf-8**:

```
f = open("abc.txt", mode='r', encoding='utf-8')
```

When we complete our operations with a file, it is a must to close the file to free up the resources for other processes to use the file. A file can be closed by using the **close** method on the file instance or the file handle. Here is a code snippet showing the use of the **close** method:

```
file = open("abc.txt", 'r+w')
#operations on file
file.close()
```

Once a file is closed, the resources associated with the file instance and locks (if any) will be released by the operating system, which is a best practice in any programming language.

## Reading and writing files

A file can be read by opening the file in access mode **r** and then using one of the read methods. Next, we summarize different methods available for read operations:

- **read(n)**: This method reads **n** characters from a file.
- **readline()**: This method returns one line from a file.
- **readlines()**: This method returns all lines from a file as a list.

Similarly, we can append or write to a file once it is opened in an appropriate access mode. The methods that are relevant to appending a file are as follows:

- **write (x)**: This method writes a string or a sequence of bytes to a file and returns the number of characters added to the file.
- **writelines (lines)**: This method writes a list of lines to a file.

In the next code example, we will create a new file, add a few text lines to it, and then read the text data using the read operations discussed previously:

```
#writereadfile.py: write to a file and then read from it
f1 = open("myfile.txt",'w')
f1.write("This is a sample file\n")
lines =["This is a test data\n", "in two lines\n"]
```

```

f1.writelines(lines)

f1.close()

f2 = open("myfile.txt", 'r')

print(f2.read(4))

print(f2.readline())

print(f2.readline())

f2.seek(0)

for line in f2.readlines():

 print(line)

f2.close()

```

In this code example, we write three lines to a file first. In the read operations, first, we read four characters, followed by reading two lines using the **readline** method. In the end, we move the pointer back to the top of the file using the **seek** method and access all lines in the file using the **readlines** method.

In the next section, we will see how the use of a context manager makes file handling convenient.

## Using a context manager

Correct and fair usage of resources is critical in any programming language. A file handler and a database connection are a couple of many examples where it is a common practice to not release the resources on time after working with objects. If the resources are not released at all, then it will end up in a situation called **memory leakage** and may impact the system performance, and ultimately may result in the system crashing.

To solve this memory leakage and timely resource release problem, Python came up with the concept of context managers. A context manager is designed to reserve and release resources precisely as per the design. When a context manager is used with the **with** keyword, a statement after the **with** keyword is expected to return an object that must implement the **context management protocol**. This protocol requires two special methods to be implemented by the returned object. These special methods are as follows:

- **\_\_enter\_\_()**: This method is called with the **with** keyword and is used to reserve the resources required as per the statement after the **with** keyword.
- **\_\_exit\_\_()**: This method is called after the execution of the **with** block and is used to release the resources that are reserved in the **\_\_enter\_\_()** method.

For example, when a file is opened using the context manager **with** statement (block), there is no need to close the file. The file **open** statement will return the file handler object, which has already implemented the context management protocol and the file will be closed automatically as soon the execution of the **with** block is completed. A revised version of the code example for writing and reading a file using the context manager is as follows:

```
#contextmgr1.py

with open("myfile.txt", 'w') as f1:
 f1.write("This is a sample file\n")
 lines = ["This is a test data\n", "in two lines\n"]
 f1.writelines(lines)

with open("myfile.txt", 'r') as f2:
 for line in f2.readlines():
 print(line)
```

The code with the context manager is simple and easy to read. The use of a context manager is a recommended approach for opening and working with files.

## Operating on multiple files

Python supports opening and operating on multiple files at the same time. We can open these files in different modes and operate on them. There is no limit on the number of files. We can open two files in read mode by using the following sample code and access the files in any order:

```
1.txt
This is a sample file 1
This is a test data 1

2.txt
This is a sample file 2
This is a test data 2

#multifilesread1.py

with open("1.txt") as file1, open("2.txt") as file2:
 print(file2.readline())
 print(file1.readline())
```

We can also read from one file and write to another file using this multifile operating option. Sample code to transfer contents from one file to another file is as follows:

```
#multifilesread2.py
```

```
with open("1.txt",'r') as file1, open("3.txt",'w') as file2:
 for line in file1.readlines():
 file2.write(line)
```

Python also has a more elegant solution to operate on multiple files using the **fileinput** module. This module's input function can take a list of multiple files and then treat all such files as a single input. Sample code with two input files, **1.txt** and **2.txt**, and using the **fileinput** module is presented next:

```
#multifilesread1.py
import fileinput

with fileinput.input(files = ("1.txt",'2.txt')) as f:
 for line in f:
 print(f.filename())
 print(line)
```

With this approach, we get one file handle that operates on multiple files sequentially. Next, we will discuss error and exception handling in Python.

## Handling errors and exceptions

There are many types of errors possible in Python. The most common one is related to the syntax of the program and is typically known as a **syntax error**. On many occasions, errors are reported during the execution of a program. Such errors are called **runtime errors**. The runtime errors that can be handled within our program are called **exceptions**. This section will focus on how to handle runtime errors or exceptions. Before jumping on to error handling, we will briefly introduce the most common runtime errors as follows:

- **IndexError:** This error occurs when a program tries to access an item at an invalid index (location in the memory).
- **ModuleNotFoundError:** This error will be thrown when a specified module is not found at the system path.
- **ZeroDivisionError:** This error is thrown when a program tries to divide a number by zero.
- **KeyError:** This error occurs when a program tries to fetch a value from a dictionary using an invalid key.
- **StopIteration:** This error is thrown when the `__next__` method does not find any further items in a container.
- **TypeError:** This error occurs when a program tries to apply an operation on an object of an inappropriate type.

A complete list of errors is available in the official documentation of Python. In the following subsections, we will discuss how to handle errors, sometimes also called exceptions, using appropriate constructs in Python.

## Working with exceptions in Python

When runtime errors arise, the program can terminate abruptly and can cause damage to system resources such as corrupting files and database tables. This is why error or exception handling is one of the key ingredients of writing robust programs in any language. The idea is to anticipate that runtime errors can occur and if such an error occurs, what the behavior of our program would be as a response to that particular error.

Like many other languages, Python uses the **try** and **except** keywords. The two keywords are followed by separate blocks of code to be executed. The **try** block is a regular set of statements for which we anticipate that an error may occur. The **except** block will be executed only if there is an error in a **try** block. Next is the syntax of writing Python code with **try** and **except** blocks:

```
try:
 #a series of statements

except:
 #statements to be executed if there is an error in \
 try block
```

If we anticipate a particular error type or multiple error types, we can define an **except** block with the error name and can add as many **except** blocks as we need. Such named **except** blocks are executed only if the named exception is raised in the **try** block. With the **except** block statement, we can also add an **as** statement to store the exception object as a variable that is raised during the **try** block. The **try** block in the next code example has many possible runtime errors and that is why it has multiple **except** blocks:

```
#exception1.py

try:
 print (x)
 x = 5
 y = 0
 z = x /y
 print('x'+ y)

except NameError as e:
 print(e)

except ZeroDivisionError:
 print("Division by 0 is not allowed")

except Exception as e:
 print("An error occurred")
 print(e)
```

To illustrate a better use of an **except** block(s), we added multiple except blocks that are explained next:

- **The NameError block:** This block will be executed when a statement in the **try** block tries to access an undefined variable. In our code example, this block will be executed when the interpreter tries to execute the **print(x)** statement. Additionally, we named the exception object as **e** and used it with the **print** statement to get the official error detail associated with this error type.
- **The ZeroDivisionError block:** This block will be executed when we try to execute **z = x/y** and **y = 0**. For this block to be executed, we need to fix the **NameError** block first.
- **The default except block:** This is a catch-all **except** block, which means if no match is found with the previous two **except** blocks, this block will be executed. The last statement **print('x'+ y)** will also raise an error of type **TypeError** and will be handled by this block. Since we are not receiving any one particular type of exception in this block, we can use the **Exception** keyword to store the exception object in a variable.

Note that as soon an error occurs in any statement in the **try** block, the rest of the statements are ignored, and the control goes to one of the **except** blocks. In our code example, we need to fix the **NameError** error first to see the next level of exception and so on. We added three different types of errors in our example to demonstrate how to define multiple **except** blocks for the same **try** block. The order of the **except** blocks is important because more specific **except** blocks with error names have to be defined first and an **except** block without specifying an error name has to always be at the end.

The following figure shows all the exception handling blocks:

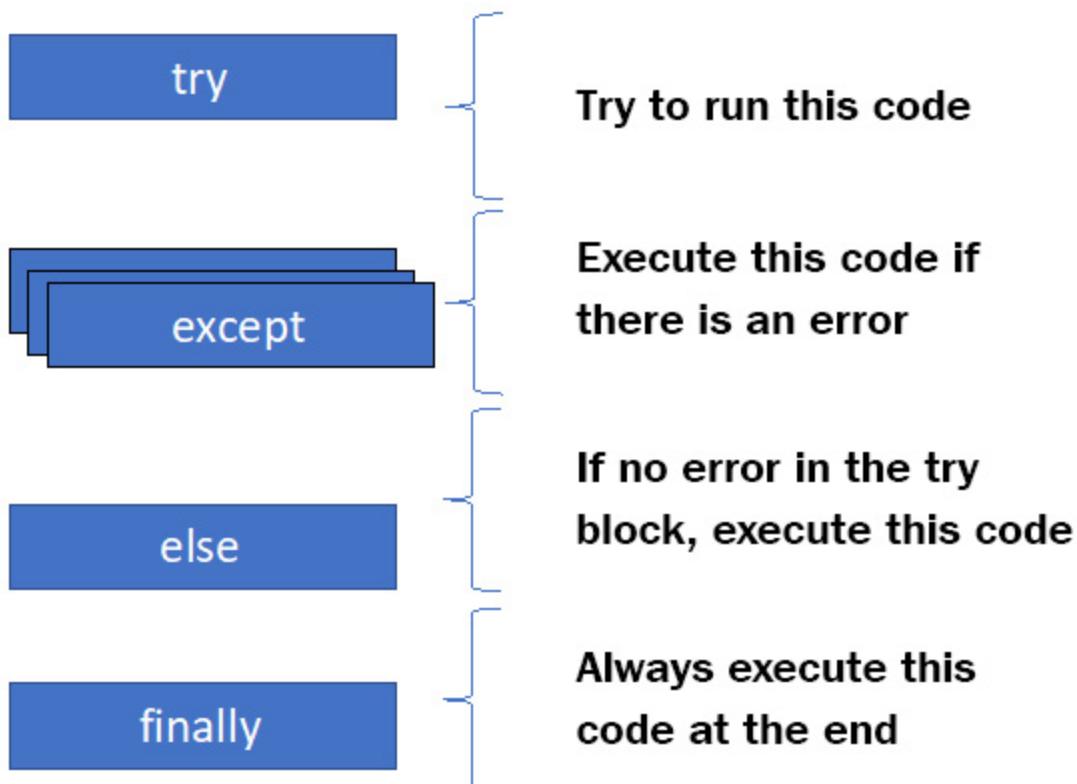


Figure 4.1 – Different exception handling blocks in Python

As shown in the preceding diagram, in addition to **try** and **except** blocks, Python also supports **else** and **finally** blocks to enhance the error handling functionality. The **else** block is executed if no errors were raised during the **try** block. The code in this block will be executed as normal and no exception will be thrown if any error occurs within this block. Nested **try** and **except** blocks can be added within the **else** block if needed. Note that this block is optional.

The **finally** block is executed regardless of whether there is an error in the **try** block or not. The code inside the **finally** block is executed without any exception handling mechanism. This block is mainly used to free up the resources by closing the connections or opened files. Although it is an optional block, it is highly recommended to implement this block.

Next, we will look at the use of these blocks with a code example. In this example, we will open a new file for writing in the **try** block. If an error occurs in opening the file, an exception will be thrown, and we will send the error details to the console using the **print** statement in the **except** block. If no error occurs, we will execute the code in the **else** block that is writing some text to the file. In both cases (error or no error), we will close the file in the **finally** block. The complete sample code is as follows:

```
#exception2.py
try:
 f = open("abc.txt", "w")
except Exception as e:
 print("Error:" + e)
else:
 f.write("Hello World")
 f.write("End")
finally:
 f.close()
```

We have covered extensively how to handle an exception in Python. Next, we will discuss how to raise an exception from Python code.

## Raising exceptions

Exceptions or errors are raised by the Python interpreter at runtime when an error occurs. We can also raise errors or exceptions ourselves if a condition occurs that may give us bad output or crash the

program if we proceed further. Raising an error or exception will provide a graceful exit of the program.

An exception (object) can be thrown to the caller by using the **raise** keyword. An exception can be of one of the following types:

- A built-in exception
- A custom exception
- A generic **Exception** object

In the next code example, we will be calling a simple function to calculate a square root and will implement it to throw an exception if the input parameter is not a valid positive number:

```
#exception3.py
import math

def sqrt(num):
 if not isinstance(num, (int, float)) :
 raise TypeError("only numbers are allowed")
 if num < 0:
 raise Exception ("Negative number not supported")
 return math.sqrt(num)

if __name__ == "__main__":
 try:
 print(sqrt(9))
 print(sqrt('a'))
 print (sqrt(-9))
 except Exception as e:
 print(e)
```

In this code example, we raised a built-in exception by creating a new instance of the **TypeError** class when the number passed to the **sqrt** function is not a number. We also raised a generic exception when the number passed is lower than **0**. In both cases, we passed our custom text to its constructor. In the next section, we will study how to define our own custom exception and then throw it to the caller.

## Defining custom exceptions

In Python, we can define our own custom exceptions by creating a new class that has to be derived from the built-in **Exception** class or its subclass. To illustrate the concept, we will revise our previous example by defining two custom exception classes to replace the built-in **TypeError** and the **Exception** error types. The new custom exception classes will be derived from the **TypeError** and the **Exception** classes. Here is sample code for reference with custom exceptions:

```
#exception4.py
import math
class NumTypeError(TypeError):
 pass
class NegativeNumError(Exception):
 def __init__(self):
 super().__init__("Negative number not supported")
def sqrt(num):
 if not isinstance(num, (int, float)) :
 raise NumTypeError("only numbers are allowed")
 if num < 0:
 raise NegativeNumError
 return math.sqrt(num)
if __name__ == "__main__":
 try:
 print(sqrt(9))
 print(sqrt('a'))
 print(sqrt(-9))
 except NumTypeError as e:
 print(e)
 except NegativeNumError as e:
 print(e)
```

In this code example, the **NumTypeError** class is derived from the **TypeError** class and we have not added anything in this class. The **NegativeNumError** class is inherited from the **Exception** class and we override its constructor and add a custom message for this exception as part of the constructor. When we raise these custom exceptions in the **sqrt()** function, we do not pass any text with the **NegativeNumError** exception class. When we used the main program, we get the message with the **print (e)** statement as we have set it as part of the class definition.

In this section, we covered how to handle built-in error types using **try** and **except** blocks, how to define custom exceptions, and how to raise an exception declaratively. In the next section, we will cover logging in Python.

## Using the Python logging module

Logging is a fundamental requirement for any reasonably sized application. Logging not only helps in debugging and troubleshooting but also provides insight into details of an application's internal issues. A few advantages of logging are as follows:

- Debugging code, especially to diagnose why and when an application failed or crashed
- Diagnosing unusual application behavior
- Providing auditing data for regulatory or legal compliance matters
- Identifying users' behaviors and malicious attempts to access unauthorized resources

Before discussing any practical examples of logging, we will first discuss the key components of the logging system in Python.

## Introducing core logging components

The following components are fundamental to set up logging for an application in Python:

- Logger
- Logging levels
- Logging formatter
- Logging handler

A high-level architecture of the Python logging system can be summarized here:

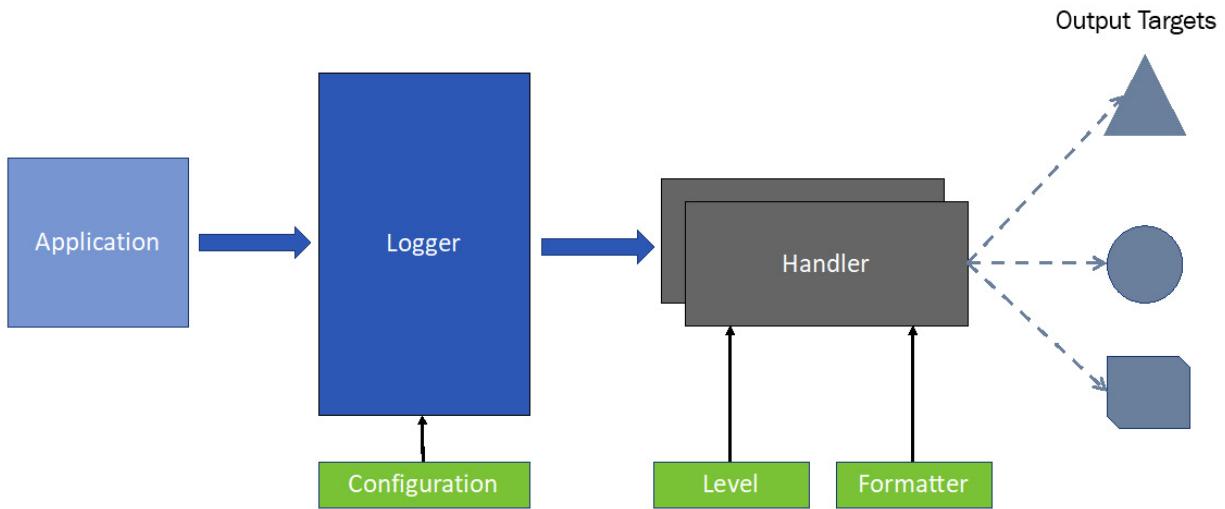


Figure 4.2 – Logging components in Python

Each of these components is discussed in detail in the following subsections.

### The logger

The logger is the entry point to the Python logging system. It is the interface to the application programmer. The **Logger** class available in Python provides several methods to log messages with different priorities. We will study the **Logger** class methods with code examples later in this section.

An application interacts with the **Logger** instance, which is set up using logging configuration such as the logging level. On receiving logging events, the **Logger** instance selects one or more appropriate logging handlers and delegates the events to the handlers. Each handler is typically designed for a specific output target. A handler sends the messages after applying a filter and formatting to the intended output target.

### Logging levels

All events and messages for a logging system are not of the same priority. For example, messages about errors are more urgent than warning messages. Logging levels are a way to set different priorities for different logging events. There are six levels defined in Python. Each level is associated with an integer value that indicates the severity. These levels are **NOTSET**, **DEBUG**, **INFO**, **WARNING**, **ERROR**, and **CRITICAL**. These are summarized here:

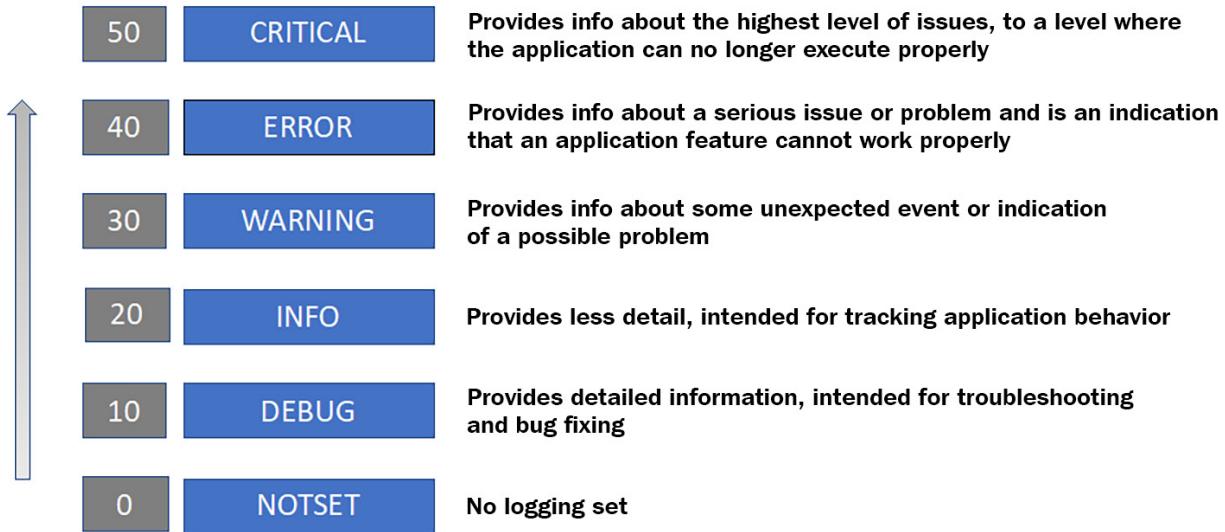


Figure 4.3 – Logging levels in Python

## The logging formatter

The logging formatter component helps to improve the formatting of messages, which is important for consistency and for human and machine readability. The logging formatter also adds extra context to messages such as time, module name, line number, threads, and process, which is extremely useful for debugging purposes. An example formatter expression is as follows:

```
"%(asctime)s - %(name)s - %(levelname)s - %(funcName)s:%(lineno)d - %(message)s"
```

When such a formatter expression is used, the log message **hello Geeks** of level **INFO** will be displayed similar to the console output that follows:

```
2021-06-10 19:20:10,864 - a.b.c - INFO - <module name>:10 - hello Geeks
```

## The logging handler

The role of a logging handler is to write log data to an appropriate destination, which can be a console, a file, or even an email. There are many types of built-in logging handlers available in Python. A few popular handlers are introduced here:

- **StreamHandler** for displaying the logs on a console
- **FileHandler** for writing the logs to a file
- **SMTPHandler** for sending the logs to an email
- **SocketHandler** for sending the logs to a network socket
- **SyslogHandler** for sending the logs to a local or remote *Unix* syslog server
- **HTTPHandler** for sending the logs to a web server using either **GET** or **POST** methods

The logging handler uses the logging formatter to add more context info to the logs and the logging level to filter the logging data.

# Working with the logging module

In this section, we will discuss how to use the **logging** module with code examples. We will start with basic logging options and will take them to an advanced level in a gradual manner.

## Using the default logger

Without creating an instance of any logger class, there is already a default logger available in Python. The default logger, also known as the **root logger**, can be used by importing the **logging** module and using its methods to dispatch logging events. The next code snippet shows the use of the root logger for capturing log events:

```
#logging1.py
import logging
logging.debug("This is a debug message")
logging.warning("This is a warning message")
logging.info("This is an info message")
```

The **debug**, **warning**, and **info** methods are used to dispatch logging events to the logger as per their severity. The default log level for this logger is set to **WARNING** and the default output is set to **stderr**, which means all the messages will go to the console or terminal only. This setting will block **DEBUG** and **INFO** messages to be displayed on the console output, which will be as follows:

```
WARNING:root:This is a warning message
```

The level of the root logger can be changed by adding the following line after the **import** statement:

```
logging.basicConfig(level=logging.DEBUG)
```

After changing the logging level to **DEBUG**, the console output will now show all the log messages:

```
DEBUG:root:This is a debug message
WARNING:root:This is a warning message
INFO:root:This is an info message
```

Although we discussed the default or root logger in this subsection, it is not recommended to use it other than for basic logging purposes. As a best practice, we should create a new logger with a name, which we will discuss in the next code examples.

## Using a named logger

We can create a separate logger with its own name and possibly with its own log level, handlers, and formatters. The next code snippet is an example of creating a logger with a custom name and also using a different logging level than the root logger:

```
#logging2.py
```

```
import logging

logger1 = logging.getLogger("my_logger")

logging.basicConfig()

logger1.setLevel(logging.INFO)

logger1.warning("This is a warning message")

logger1.info("This is a info message")

logger1.debug("This is a debug message")

logging.info("This is an info message")
```

When we create a logger instance using the **getLogger** method with a string name or using the module name (by using the `__name__` global variable), then only one instance is managed for one name. This means if we try to use the **getLogger** method with the same name in any part of the application, the Python interpreter will check whether there is already an instance created for this name. If there is already one created, it will return the same instance.

After creating a logger instance, we need to make a call to the root logger (**basicConfig()**) to provide a handler and formatter to our logger. Without any handler configuration, we will get an internal handler as the last resort, which will only output messages without any formatting and the logging level will be **WARNING** regardless of the logging level we set for our logger. The console output of this code snippet is shown next, and it is as expected:

```
WARNING:my_logger:This is a warning message
INFO:my_logger:This is a info message
```

It is also important to note the following:

- We set the logging level for our logger to **INFO** and we were able to log **warning** and **info** messages but not the debug message.
- When we used the root logger (by using the **logging** instance), we were not able to send out the **info** message. This was because the root logger was still using the default logging level, which is **WARNING**.

## Using a logger with a built-in handler and custom formatter

We can create a logger object using a built-in handler but with a custom formatter. In this case, the handler object can use a custom formatter object and the handler object can be added to the logger object as its handler before we start using the logger for any log events. Here is a code snippet to illustrate how to create a handler and a formatter programmatically and then add the handler to the logger:

```
#logging3.py

import logging

logger = logging.getLogger('my_logger')
```

```

my_handler = logging.StreamHandler()
my_formatter = logging.Formatter('%(asctime)s - '\
 '%(name)s - %(levelname)s - %(message)s')
my_handler.setFormatter(my_formatter)
logger.addHandler(my_handler)
logger.setLevel(logging.INFO)
logger.warning("This is a warning message")
logger.info("This is an info message")
logger.debug("This is a debug message")

```

We can create a logger with the same settings by using the **basicConfig** method as well with appropriate arguments. The next code snippet is a revised version of **logging3.py** with the **basicConfig** settings:

```

#logging3A.py
import logging
logger = logging.getLogger('my_logger')
logging.basicConfig(handlers=[logging.StreamHandler()],
 format="%(asctime)s - %(name)s - "
 "%(levelname)s - %(message)s",
 level=logging.INFO)
logger.warning("This is a warning message")
logger.info("This is an info message")
logger.debug("This is a debug message")

```

Up till now, we have covered cases where we used built-in classes and objects to set up our loggers. Next, we will set up a logger with custom handlers and formatters.

## Using a logger with a file handler

The logging handler sends the log messages to their final destination. By default, every logger is set up to send log messages to the console or terminal associated with the running program. But this can be changed by configuring a logger with a new handler with a different destination. A file handler can be created by using one of the two approaches we already discussed in the previous subsection. In this section, we will use a third approach to create a file handler automatically with the **basicConfig** method by providing the filename as an attribute to this method. This is shown in the next code snippet:

```

#logging4.py
import logging

```

```

logging.basicConfig(filename='logs/logging4.log'
 , level=logging.DEBUG)

logger = logging.getLogger('my_logger')
logger.setLevel(logging.INFO)
logger.warning("This is a warning message")
logger.info("This is a info message")
logger.debug("This is a debug message")

```

This will generate log messages to the file we specified with the **basicConfig** method and as per the logging level, which is set to **INFO**.

### **Using a logger with multiple handlers programmatically**

Creating a logger with multiple handlers is pretty straightforward and can be achieved either by using the **basicConfig** method or by attaching handlers manually to a logger. For illustration purposes, we will revise our code example **logging3.py** to do the following:

1. We will create two handlers (one for the console output and one for the file output) that are instances of the **streamHandler** and **fileHandler** classes.
2. We will create two separate formatters, one for each handler. We will not include the time information for the formatter of the console handler.
3. We will set separate logging levels for the two handlers. It is important to understand that the logging level at the handler level cannot override the root level handler.

Here is the complete code example:

```

#logging5.py

import logging

logger = logging.getLogger('my_logger')

logger.setLevel(logging.DEBUG)

console_handler = logging.StreamHandler()

file_handler = logging.FileHandler("logs/logging5.log")

#setting logging levels at the handler level
console_handler.setLevel(logging.DEBUG)

file_handler.setLevel(logging.INFO)

#createing separate formatter for two handlers

console_formatter = logging.Formatter(
 '%(name)s - %(levelname)s - %(message)s')

file_formatter = logging.Formatter('%(asctime)s - '
 '%(name)s - %(levelname)s - %(message)s')

```

```

#adding formatters to the handler
console_handler.setFormatter(console_formatter)
file_handler.setFormatter(file_formatter)

#adding handlers to the logger
logger.addHandler(console_handler)
logger.addHandler(file_handler)

logger.error("This is an error message")
logger.warning("This is a warning message")
logger.info("This is an info message")
logger.debug("This is a debug message")

```

Although we set different logging levels for the two handlers, which are **INFO** and **DEBUG**, they will be effective only if the logging level of the logger is at a lower value (the default is **WARNING**). This is why we have to set the logging level for our logger to **DEBUG** at the beginning of the program. The logging level at the handler level can be **DEBUG** or any higher level. This is a very important point to consider whenever designing a logging strategy for your application.

In the code example shared in this section, we basically configured the logger programmatically. In the next section, we will work on how to configure a logger through a configuration file.

## **Configuring a logger with multiple handlers using a configuration file**

Setting up a logger programmatically is appealing but not practical for production environments. In production environments, we have to set up the logger configuration differently as compared to the development setup and sometimes we have to enhance the logging level to troubleshoot problems that we encounter only in live environments. This is why we have the option of providing the logger configuration through a file that is easy to change as per the target environment. The configuration file for a logger can be written using **JSON (JavaScript Object Notation)** or **YAML (Yet Another Markup Language)** or as a list of *key:value* pairs in a **.conf** file. For illustration purposes, we will demonstrate the logger configuration using a YAML file, which is exactly the same as we achieved programmatically in the previous section. The complete YAML file and the Python code is as follows:

The following is the YAML config file:

```

version: 1
formatters:
 console_formatter:
 format: '%(name)s - %(levelname)s - %(message)s'
 file_formatter:

```

```

format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'

handlers:

console_handler:

 class: logging.StreamHandler
 level: DEBUG
 formatter: console_formatter
 stream: ext://sys.stdout

file_handler:

 class: logging.FileHandler
 level: INFO
 formatter: file_formatter
 filename: logs/logging6.log

loggers:

my_logger:

 level: DEBUG
 handlers: [console_handler, file_handler]
 propagate: no

root:

 level: ERROR
 handlers: [console_handler]

```

The following is the Python program using the YAML file to configure the logger:

```

#logging6.py

import logging
import logging.config
import yaml

with open('logging6.conf.yaml', 'r') as f:
 config = yaml.safe_load(f.read())
 logging.config.dictConfig(config)

logger = logging.getLogger('my_logger')
logger.error("This is an error message")
logger.warning("This is a warning message")
logger.info("This is a info message")
logger.debug("This is a debug message")

```

To load config from a file, we used the **dictConfig** method instead of the **basicConfig** method. The outcome of the YAML-based logger configuration is exactly the same as we achieved with Python statements. There are other additional configuration options available for a full-featured logger.

In this section, we presented different scenarios of configuring one or more logger instances for an application. Next, we will discuss what type of events to log and what not to log.

## What to log and what not to log

There is always a debate about what information we should log and what not to log. As a best practice, the following information is important for logging:

- An application should log all errors and exceptions and the most appropriate way is to log these events at the source module.
- Exceptions that are handled with an alternative flow of code can be logged as warnings.
- For debugging purposes, entry and exit to a function is useful information for logging.
- It is also useful to log decision points in the code because it can be helpful for troubleshooting.
- The activities and actions of users, especially related to the access of certain resources and functions in the application, are important to log for security and auditing purposes.

When logging messages, the context information is also important, which includes the time, logger name, module name, function name, line number, logging level, and so on. This information is critical for identifying the route cause analysis.

A follow-up discussion on this topic is what not to capture for logging. We should not log any sensitive information such as user ID, email address, passwords, and any private and sensitive data. We should also avoid logging any personal and business record data such as health records, government-issued document details, and organization details.

## Summary

In this chapter, we discussed a variety of topics that require the use of advanced Python modules and libraries. We started by refreshing our knowledge about data containers in Python. Next, we learned how to use and build iterators for iterable objects. We also covered generators, which are more efficient and easier to build and use than iterators. We discussed how to open and read from files and how to write to files, followed by the use of a context manager with files. In the next topic, we discussed how to handle errors and exceptions in Python, how to raise exceptions through programming, and how to define custom exceptions. Exception handling is fundamental to any decent Python application. In the last section, we covered how to configure the logging framework in Python using different options for handlers and formatters.

After going through this chapter, you now know how to build your own iterators and design generator functions to iterate on any iterable object, and how to handle files, errors, and exceptions in Python. You have also learned how to set up loggers with one or more handlers to manage the logging of an application using different logging levels. The skills you have learned in this chapter are key to building any open source or commercial applications.

In the next chapter, we will switch our focus to how to build and automate unit tests and integration tests.

## Questions

1. What is the difference between a list and a tuple?
2. Which Python statement will always be used when working with a context manager?
3. What is the use of the **else** statement with the **try-except** block?
4. Generators are better to use than iterators. Why?
5. What is the use of multiple handlers for logging?

## Further reading

- *Fluent Python* by Luciano Ramalho
- *Advanced Guide to Python 3 Programming* by John Hunt
- *The Python 3 Standard Library by Example* by Doug Hellmann
- *Python 3.7.10 documentation* (<https://docs.python.org/3.7/>)
- To learn more about additional options available for configuring a logger, you can refer to the official Python documentation at <https://docs.python.org/3/library/logging.config.html>

## Answers

1. A list is a mutable object whereas a tuple is immutable. This means we can update a list after creating it. This is not true for tuples.
2. The **with** statement is used with a context manager.
3. The **else** block is executed only when the code in the **try** block is executed without any error. A follow-up action can be coded in the **else** block once the core functionality is executed without any problem in the **try** block.
4. Generators are efficient in memory and also easy to program as compared to iterators. A generator function automatically provides an **iterator** instance and the **next** function implementation out of the box.
5. The use of multiple handlers is common because one handler usually focuses on one type of destination. If we need to send log events to multiple destinations and perhaps with different priority levels, we will need multiple handlers. Also, if we need to log messages to multiple files with different logging levels, we can create different file handlers to coordinate with multiple files.



## *Chapter 5: Testing and Automation with Python*

Software testing is the process of validating an application or a program as per user requirements or desired specifications and evaluating the software for scalability and optimization goals. Validating software as a real user takes a long time and is not an efficient use of human resources. Moreover, testing is not performed only one or two times, but it is a continuous process as a part of software development. To rescue the situation, test automation is recommended for all sorts of testing. **Test automation** is a set of programs written to validate an application's behavior using different scenarios as input to these programs. For professional software development environments, it is a must that automation tests get executed every time the source code is updated (also called a **commit operation**) into a central repository.

In this chapter, we will study different approaches to automated testing, followed by looking at different types of testing frameworks and libraries that are available for Python applications. Then, we will focus on unit testing and will look into different ways of implementing unit testing in Python. Next, we will study the usefulness of **test-driven development (TDD)** and the right way to implement it. Finally, we will focus on automated **continuous integration (CI)** and will look into the challenges of implementing it robustly and efficiently. This chapter will help you understand the concepts of automated testing in Python at various levels.

We will cover the following topics in this chapter:

- Understanding various levels of testing
- Working with Python test frameworks
- Executing TDD
- Introducing automated CI

At the end of this chapter, you will not only understand different types of test automation but will also be able to write unit tests using one of the two popular test frameworks.

## **Technical requirements**

These are the technical requirements for this chapter:

- You need to have installed Python 3.7 or later on your computer.
- You need to register an account with Test PyPI and create an **application programming interface (API)** token under your account.

The sample code for this chapter can be found at <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter05>.

## Understanding various levels of testing

Testing is performed at various levels based on the application type, its complexity level, and the role of the team that is working on the application. The different levels of testing include the following:

- Unit testing
- Integration testing
- System testing
- Acceptance testing

These different levels of testing are applied in the order shown here:

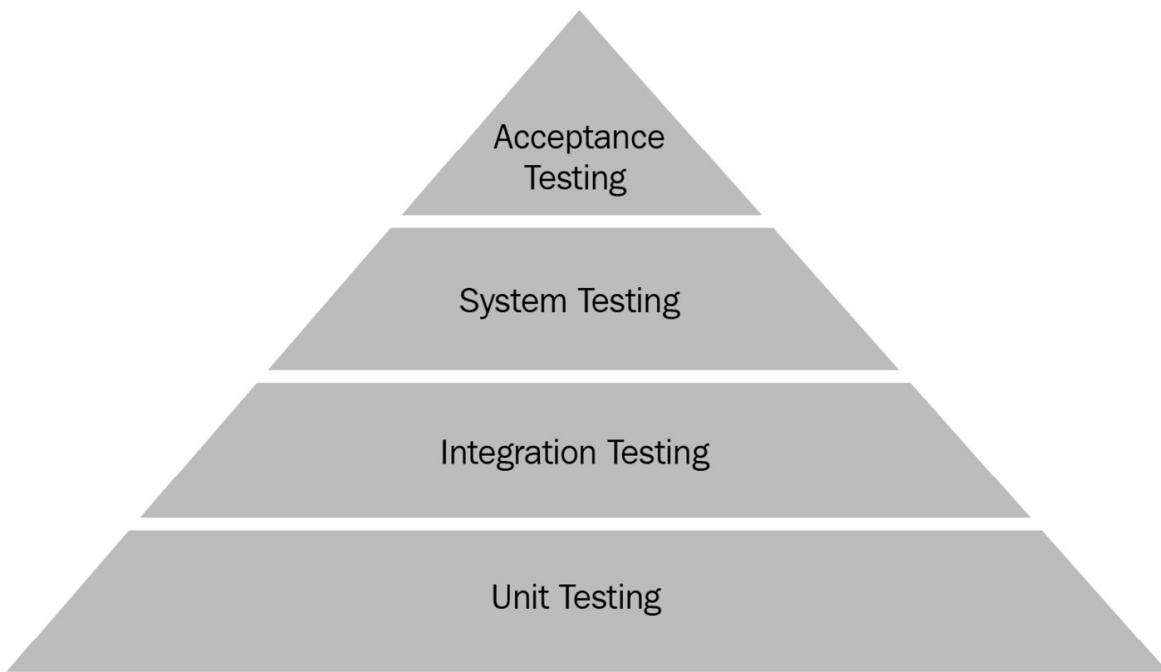


Figure 5.1 – Different levels of testing during software development

These testing levels are described in the next subsections.

## Unit testing

Unit testing is a type of testing that is focused on the smallest possible unit level. A unit corresponds to a unit of code that can be a function in a module or a method in a class, or it can be a module in an application. A unit test executes a single unit of code in isolation and validates that the code is working as expected. Unit testing is a technique used by developers to identify bugs at the early stages of code development and fix them as part of the first iteration of the development process. In Python, unit testing mainly targets a particular class or module without involving dependencies.

Unit tests are developed by application developers and can be performed at any time. Unit testing is a kind of **white-box testing**. The libraries and tools available for unit testing in Python are **pyunit** (**unittest**), **pytest**, **doctest**, **nose**, and a few others.

## Integration testing

Integration testing is about testing individual units of a program collectively in the form of a group. The idea behind this type of testing is to test the combination of different functions or modules of an application together to validate the interfaces between the components and the data exchange between them.

Integration testing is typically done by testers and not by developers. This type of testing starts after the unit testing process, and the focus of this testing is to identify the integration problem when different modules or functions are used together. In some cases, integration testing requires external resources or data that may not be possible to provide in a development environment. This limitation can be managed by using mock testing, which provides replacement mock objects for external or internal dependencies. The mock objects simulate the behavior of the real dependencies. Examples of mock testing can be sending an email or making a payment using a credit card.

Integration testing is a kind of **black-box testing**. The libraries and the tools used for integration testing are pretty much the same as for unit testing, with a difference that the boundaries of tests are pushed further out to include multiple units in a single test.

## System testing

The boundaries of system testing are further pushed out to the system level, which may be a full-blown module or an application. This type of testing validates the application functionality from an **end-to-end (E2E)** perspective.

System tests are also developed by testers but after completing the integration testing process. We can say that integration testing is a prerequisite for system testing; otherwise, a lot of effort will be repeated while performing system testing. System testing can identify potential problems but does not pinpoint the location of the problem. The exact root cause of the problem is typically identified by integration testing or even by adding more unit tests.

System testing is also a type of black-box testing and can leverage the same libraries that are available for integration testing.

## Acceptance testing

Acceptance testing is end-user testing before accepting the software for day-to-day use. Acceptance testing is not commonly a candidate for automation testing, but it is worth using automation for acceptance testing in situations where application users have to interact with the product using an API. This testing is also called **user acceptance testing (UAT)**. This type of testing can be easily mixed up with system testing but it is different in that it ensures the usability of the application from a real user's point of view. There are also further two types of acceptance testing: **factory acceptance testing (FAT)** and **operational acceptance testing (OAT)**. The former is more popular from a hardware point of view, and the latter is performed by the operation teams, who are responsible for using the product in production environments.

Additionally, we also hear about **alpha** and **beta** testing. These are also user-level testing approaches and are not meant for test automation. Alpha testing is performed by developers and internal staff to emulate actual user behavior. Beta testing is performed by customers or actual users for early feedback before declaring **general availability (GA)** of the software.

We also use the term **regression testing** in software development. This is basically the execution of tests every time we make a change in the source code or any internal or external dependency changes. This practice ensures that our product is performing in the same way as it was before making a change. Since regression testing is repeated many times, automating the tests is a must for this type of testing.

In the next section, we will investigate how to build test cases using the test frameworks in Python.

## Working with Python test frameworks

Python comes with standard as well as third-party libraries for test automation. The most popular frameworks are listed here:

- **pytest**
- **unittest**
- **doctest**
- **nose**

These frameworks can be used for unit testing as well as for integration and system testing. In this section, we will evaluate two of these frameworks: **unittest**, which is part of the Python standard library, and **pytest**, which is available as an external library. The focus of this evaluation will be on building test cases (mainly unit tests) using these two frameworks, although the integration and system tests can also be built using the same libraries and design patterns.

Before we start writing any test cases, it is important to understand what a test case is. In the context of this chapter and book, we can define a test case as a way of validating the outcomes of a particular behavior of a programming code as per the expected results. The development of a test case can be broken down into the following four stages:

1. **Arrange:** This is a stage where we prepare the environment for our test cases. This does not include any action or validation step. In the test automation community, this stage is more commonly known as preparing **test fixtures**.
2. **Act:** This is the action stage that triggers the system we want to test. This action stage results in a change in the system behavior, and the changed state of the system is something we want to evaluate for validation purposes. Note that we do not validate anything at this stage.
3. **Assert:** At this stage, we evaluate the results of the *act* stage and validate the results against the expected outcome. Based on this validation, the test automation tools mark the test case as failed or passed. In most of the tools, this validation is achieved using built-in *assert* functions or statements.
4. **Cleanup:** At this stage, the environment is cleaned up to make sure the other tests are not impacted by the status changes caused by the *act* stage.

The core stages of a test case are *act* and *assert*. The *arrange* and *cleanup* stages are optional but highly recommended. These two stages mainly provide software test fixtures. A test fixture is a type of equipment or device or software that provides an environment to test a device or a machine or software consistently. The term *test fixture* is used in the same context for unit testing and integration testing.

The test frameworks or libraries provide helper methods or statements to facilitate the implementation of these stages conveniently. In the next sections, we will evaluate the **unittest** and the **pytest** frameworks for the following topics:

- How to build base-level test cases for act and assert stages
- How to build test cases with test fixtures
- How to build test cases for exception and error validation
- How to run test cases in bulk
- How to include and exclude test cases in execution

These topics not only cover the development of a variety of test cases but also include different ways to execute them. We will start our evaluation with the **unittest** framework.

## Working with the unittest framework

Before starting to discuss practical examples with the **unittest** framework or library, it is important to introduce a few terms and traditional method names related to unit testing and, in particular, to the **unittest** library. This terminology is used more or less by all test frameworks and is outlined here:

- **Test case:** A test or test case or test method is a set of code instructions that are based on a comparison of the current condition versus the post-execution conditions after executing a unit of application code.
- **Test suite:** A test suite is a collection of test cases that may have common pre-conditions, initialization steps, and perhaps the same cleanup steps. This foments reusability of test automation code and reduced execution time.
- **Test runner:** This is a Python application that executes the tests (unit tests), validates all the assertions defined in the code, and gives the results back to us as a success or a failure.
- **Setup:** This is a special method in a test suite that will be executed before each test case.
- **setupClass:** This is a special method in a test suite that will be executed only once at the start of the execution of tests in a test suite.
- **teardown:** This is another special method in a test suite that is executed after completion of every test regardless of whether the test passes or fails.
- **tearDownClass:** This is another special method in a test suite that is executed only once when all the tests in a suite are completed.

To write test cases using the **unittest** library, we are required to implement the test cases as instance methods of a class that must be inherited from the **TestCase** base class. The **TestCase** class comes with several methods to facilitate writing as well as executing the test cases. These methods are grouped into three categories, which are discussed next:

- **Execution-related methods:** The methods included in this category are **setUp**, **tearDown**, **setupClass**, **tearDownClass**, **run**, **skipTest**, **skipTestIf**, **subTest**, and **debug**. These tests are used by the test runner to execute a piece of code before or after a test case or running a set of test cases, running a test, skipping a test, or running any block of code as a sub-test. In our test case implementation class, we can override these methods. The exact details of these methods are available as part of the Python documentation at <https://docs.python.org/3/library/unittest.html>.
- **Validation methods (assert methods):** These methods are used to implement test cases to check for success or failure conditions and report success or failures for a test case automatically. These methods' name typically starts with an *assert* prefix. The list of assert methods is very long. We provide commonly used assert methods here as examples:

| Method name         | Evaluating condition              |
|---------------------|-----------------------------------|
| assertEquals (x, y) | Check if x is equal to y          |
| assertTrue (x)      | Check if x is Boolean true        |
| assertFalse (x)     | Check if x is Boolean false       |
| assertNotEqual (x)  | Check if x is not equal to y      |
| assertIn (a, c)     | Check if a is in collection c     |
| assertNotIn (a, c)  | Check if a is not in collection c |
| assertIs (x, y)     | Check if x is y                   |
| assertIsNot (x, y)  | Check if x is not y               |
| assertIsNone (x)    | Check if x is none                |

Figure 5.2 – A few examples of assert methods of the TestCase class

- **Additional information-related methods and attributes:** These methods and attributes provide additional information related to test cases that are to be executed or already executed. Some of the key methods and attributes in this category are summarized next:
  - a) **failureException:** This attribute provides an exception raised by a test method. This exception can be used as a superclass to define a custom failure exception with additional information.
  - b) **longMessage:** This attribute determines what to do with a custom message that is passed as an argument with an **assert** method. If the value of this attribute is set to **True**, the message is

appended to the standard failure message. If this attribute is set to **false**, a custom message replaces the standard message.

c) **countTestCases()**: This method returns the number of tests attached to a test object.

d) **shortDescription()**: This method returns a description of a test method if there is any description added, using a docstring.

We have reviewed the main methods of the **TestCase** class in this section. In the next section, we will explore how to use **unittest** to build unit tests for a sample module or an application.

## Building test cases using the base TestCase class

The **unittest** library is a standard Python testing framework that is highly inspired by the **JUnit** framework, a popular testing framework in the Java community. Unit tests are written in separate Python files and it is recommended to make the files part of the main project. As we discussed in [Chapter 2, Using Modularization to Handle Complex Projects](#), in the *Building a package* section, the **Python Packaging Authority (PyPA)** guidelines recommend having a separate folder for tests when building packages for a project or a library. In our code examples for this section, we will follow a similar structure to the one shown here:

```
Project-name
| -- src
| -- __init__.py
| -- myadd/myadd.py
| -- tests
| -- __init__.py
| -- tests_myadd/test_myadd1.py
| -- tests_myadd/test_myadd2.py
| -- README.md
```

In our first code example, we will build a test suite for the **add** function in the **myadd.py** module, as follows:

```
myadd.py with add two numbers
def add(x, y):
 """This function adds two numbers"""
 return x + y
```

It is important to understand that there can be more than one test case for the same piece of code (an **add** function, in our case). For the **add** function, we implemented four test cases by varying the

values of input parameters. Next is a code sample with four test cases for the **add** function, as follows:

```
#test_myadd1.py test suite for myadd function

import unittest

from myunittest.src.myadd.myadd import add

class MyAddTestSuite(unittest.TestCase):

 def test_add1(self):
 """ test case to validate two positive numbers"""
 self.assertEqual(15, add(10 , 5), "should be 15")

 def test_add2(self):
 """ test case to validate positive and negative \
 numbers"""
 self.assertEqual(5, add(10 , -5), "should be 5")

 def test_add3(self):
 """ test case to validate positive and negative \
 numbers"""
 self.assertEqual(-5, add(-10 , 5), "should be -5")

 def test_add4(self):
 """ test case to validate two negative numbers"""
 self.assertEqual(-15, add(-10 , -5), "should be -15")

if __name__ == '__main__':
 unittest.main()
```

All the key points of the preceding test suite are discussed next, as follows:

- To implement unit tests using the **unittest** framework, we need to import a standard library with the same name, **unittest**.
- We need to import the module or modules we want to test in our test suite. In this case, we imported the **add** function from the **myadd.py** module using the relative import approach (see the *Importing modules* section of [Chapter 2, Using Modularization to Handle Complex Projects](#), for details)
- We will implement a test suite class that is inherited from the **unittest.TestCase** base class. The test cases are implemented in the subclass, which is the **MyAddTestSuite** class in this case. The **unittest.TestCase** class constructor can take a method name as an input that can be used to run the test cases. By default, there is a **runTest** method already implemented that is used by the test runner to execute the tests. In a majority of the cases, we do not need to provide our own method or re-implement the **runTest** method.
- To implement a test case, we need to write a method that starts with the **test** prefix and is followed by an underscore. This helps the test runner to look for the test cases to be executed. Using this naming convention, we added four methods to our test suite.
- In each test-case method, we used a special **assertEqual** method, which is available from the base class. This method represents the assert stage of a test case and is used to decide if our test will be declared as passed or failed. The first parameter of this

method is the expected results of the unit test, the second parameter is the value that we get after executing the code under test, and the third parameter (optional) is the message to be provided in the report in case the test is failed.

- At the end of the test suite, we added the **unittest.main** method to trigger the test runner to run the **runTest** method, which makes it easy to execute the tests without using the commands at the console. This **main** method (a **TestProgram** class under the hood) will first discover all the tests to be executed and then execute the tests.

## **IMPORTANT NOTE**

*Unit tests can be run using a command such as **Python -m unittest <test suite or module>**, but the code examples we provide in this chapter will assume that we are running the test cases using the PyCharm integrated development environment (IDE).*

Next, we will build the next level of test cases using the test fixtures.

## **Building test cases with test fixtures**

We have discussed **setUp** and **tearDown** methods that are run automatically by test runners before and after executing a test case. These methods (along with the **setUpClass** and **tearDownClass** methods) provide the test fixtures and are useful to implement the unit tests efficiently.

First, we will revise the implementation of our **add** function. In the new implementation, we will make this unit of code a part of the **MyAdd** class. We are also handling the situation by throwing a **TypeError** exception in case the input arguments are invalid. Next is the complete code snippet with the new **add** method:

```
myadd2.py is a class with add two numbers method

class MyAdd:

 def add(self, x, y):
 """This function adds two numbers"""

 if (not isinstance(x, (int, float))) | \
 (not isinstance(y, (int, float))) :
 raise TypeError("only numbers are allowed")

 return x + y
```

In the previous section, we built test cases using only the act stage and the assert stage. In this section, we will revise the previous code example by adding **setUp** and **tearDown** methods. Next is the test suite for this **myAdd** class, as follows:

```
#test_myadd2.py test suite for myadd2 class method

import unittest

from myunittest.src.myadd.myadd2 import MyAdd

class MyAddTestSuite(unittest.TestCase):

 def setUp(self):
 self.myadd = MyAdd()
```

```

def tearDown(self):
 del (self.myadd)
def test_add1(self):
 """ test case to validate two positive numbers"""
 self.assertEqual(15, self.myadd.add(10 , 5), \
 "should be 15")
def test_add2(self):
 """ test case to validate positive and negative numbers"""
 self.assertEqual(5, self.myadd.add(10 , -5), \
 "should be 5")
#test_add3 and test_add4 are skipped as they are very \
same as test_add1 and test_add2

```

In this test suite, we added or changed the following:

- We added a **setUp** method in which we created a new instance of the **MyAdd** class and saved its reference as an instance attribute. This means we will be creating a new instance of the **MyAdd** class *before* we execute any test case. This may not be ideal for this test suite, as a better approach could be to use the **setUpClass** method and create a single instance of the **MyAdd** class for the whole test suite, but we have implemented it this way for illustration purposes.
- We also added a **tearDown** method. To demonstrate how to implement it, we simply called the destructor (using the **del** function) on the **MyAdd** instance that we created in the **setUp** method. As with the **setUp** method, the **tearDown** method is executed *after* each test case. If we intend to use the **setUpClass** method, there is an equivalent method for teardown, which is **tearDownClass**.

In the next section, we will present code examples that will build test cases to handle a **TypeError** exception.

## Building test cases with error handling

In the previous code examples, we only compared the test-case results with the expected results. We did not consider any exception handling such as what would be the behavior of our program if the wrong types of arguments were passed as input to our **add** function. The unit tests have to cover these aspects of the programming as well.

In the next code example, we will build test cases to handle errors or exceptions which are expected from a unit of code. For this example, we will use the same **add** function, which throws a **TypeError** exception if the argument is not a number. The test cases will be built by passing non-numeric arguments to the **add** function. The next code snippet shows the test cases:

```

#test_myadd3.py test suite for myadd2 class method to validate errors
import unittest
from myunittest.src.myadd.myadd2 import MyAdd
class MyAddTestSuite(unittest.TestCase):

```

```

def setUp(self):
 self.myadd = MyAdd()
def test_typeerror1(self):
 """ test case to check if we can handle non \
 number input"""
 self.assertRaises(TypeError, self.myadd.add, \
 'a' , -5)
def test_typeerror2(self):
 """ test case to check if we can handle non \
 number input"""
 self.assertRaises(TypeError, self.myadd.add, \
 'a' , 'b')

```

In the preceding code snippet, we added two additional test cases to the **test\_add3.py** module. These test cases use the **assertRaises** method to validate if a particular type of exception is thrown or not. In our test cases, we used a single letter (**a**) or two letters (**a** and **b**) as arguments for the two test cases. In both cases, we are expecting the intended exception (**TypeError**) to be thrown. It is important to note the arguments of the **assertRaises** method. This method expects only the method or function name as a second argument. The parameters of the method or function have to be passed separately as arguments of the **assertRaises** function.

So far, we have executed multiple test cases under a single test suite. In the next section, we will discuss how we can run multiple test suites simultaneously, using the command line and also programmatically.

## **Executing multiple test suites**

As we built test cases for each unit of code, the number of test cases (unit test cases) grows very quickly. The idea of using test suites is to bring modularity into the test-case development. Test suites also make it easier to maintain and extend the test cases as we add more functionality to an application. The next aspect that comes to our mind is how to execute multiple test suites through a master script or a workflow. CI tools such as Jenkins provides such functionality out of the box. Test frameworks such as **unittest**, **nose**, or **pytest** also provide similar features.

In this section, we will build a simple calculator application (a **MyCalc** class) with **add**, **subtract**, **multiply**, and **divide** methods in it. Later, we will add one test suite for each method in this class. This way, we will add four test suites for this calculator application. A directory structure is important in implementing the test suites and test cases. For this application, we will use the following directory structure:

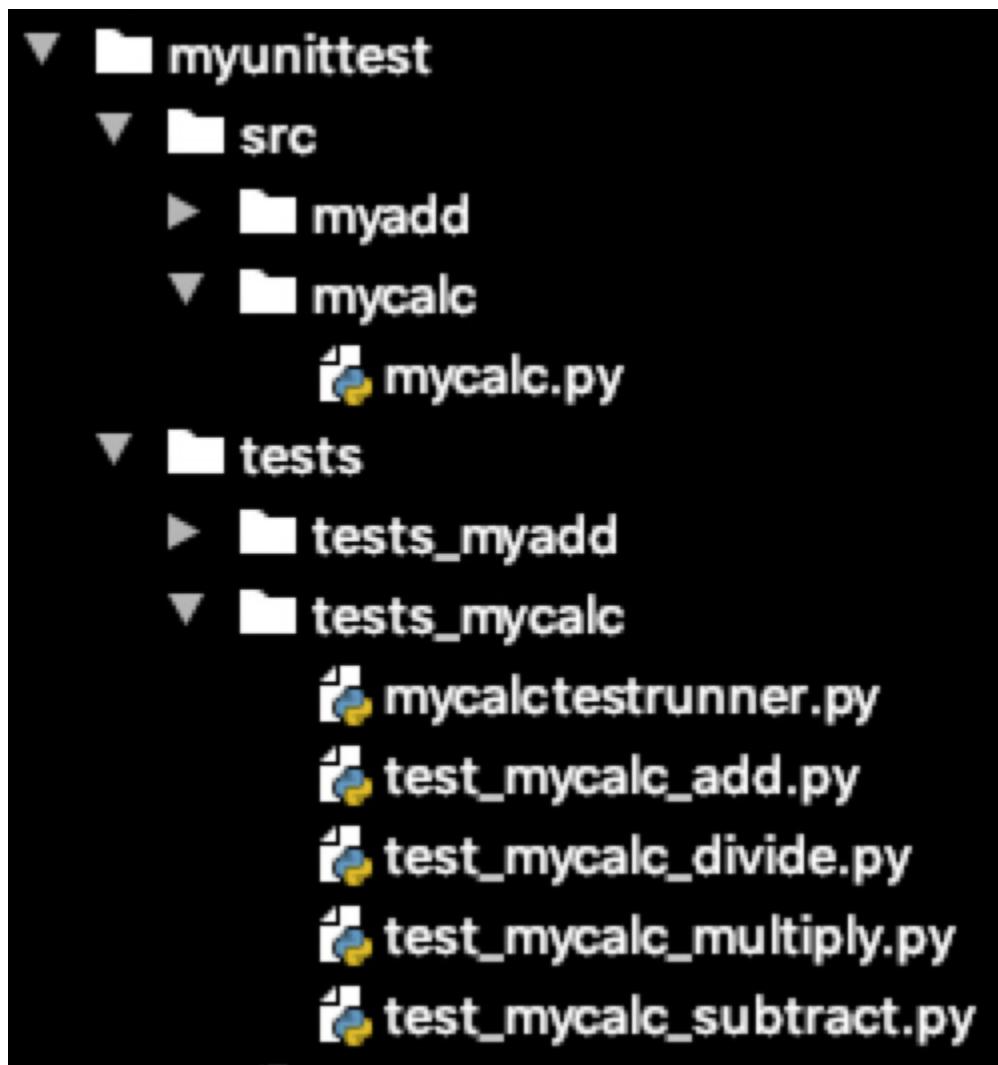


Figure 5.3 – Directory structure for the mycalc application and test suites associated with this application

The Python code is written in the **mycalc.py** module and the test suite files (**test\_mycalc\*.py**) are shown next. Note that we show only one test case in each test suite in the code examples shown next. In reality, there will be multiple test cases in each test suite. We will start with the calculator functions in the **mycalc.py** file, as follows:

```
mycalc.py with add, subtract, multiply and divide functions

class MyCalc:

 def add(self, x, y):
 """This function adds two numbers"""
 return x + y

 def subtract(self, x, y):
 """This function subtracts two numbers"""


```

```

 return x - y

 def multiply(self, x, y):
 """This function subtracts two numbers"""
 return x * y

 def divide(self, x, y):
 """This function divides two numbers"""
 return x / y

```

Next, we have a test suite to test the **add** function in the **test\_mycalc\_add.py** file, as illustrated in the following code snippet:

```

test_mycalc_add.py test suite for add class method

import unittest

from myunittest.src.mycalc.mycalc import MyCalc

class MyCalcAddTestSuite(unittest.TestCase):

 def setUp(self):
 self.calc = MyCalc()

 def test_add(self):
 """ test case to validate two positive numbers"""
 self.assertEqual(15, self.calc.add(10, 5), \
 "should be 15")

```

Next, we have a test suite to test the **subtract** function in the **test\_mycalc\_subtract.py** file, as illustrated in the following code snippet:

```

#test_mycalc_subtract.py test suite for subtract class method

import unittest

from myunittest.src.mycalc.mycalc import MyCalc

class MyCalcSubtractTestSuite(unittest.TestCase):

 def setUp(self):
 self.calc = MyCalc()

 def test_subtract(self):
 """ test case to validate two positive numbers"""
 self.assertEqual(5, self.calc.subtract(10,5), \
 "should be 5")

```

Next, we have a test suite to test the **multiply** function in the **test\_mycalc\_multiply.py** file, as illustrated in the following code snippet:

```
#test_mycalc_multiply.py test suite for multiply class method

import unittest

from myunittest.src.mycalc.mycalc import MyCalc

class MyCalcMultiplyTestSuite(unittest.TestCase):

 def setUp(self):
 self.calc = MyCalc()

 def test_multiply(self):
 """ test case to validate two positive numbers"""
 self.assertEqual(50, self.calc.multiply(10, 5), "should be 50")
```

Next, we have a test suite to test the **divide** function in the **test\_mycalc\_divide.py** file, as illustrated in the following code snippet:

```
#test_mycalc_divide.py test suite for divide class method

import unittest

from myunittest.src.mycalc.mycalc import MyCalc

class MyCalcDivideTestSuite(unittest.TestCase):

 def setUp(self):
 self.calc = MyCalc()

 def test_divide(self):
 """ test case to validate two positive numbers"""
 self.assertEqual(2, self.calc.divide(10, 5), \
 "should be 2")
```

We have the sample application code and all four test suites' code. The next aspect is how to execute all the test suites in one go. One easy way to do this is by using the **command-line interface (CLI)** with the **discover** keyword. In our example case, we will run the following command from the top of the project to discover and execute all test cases in all the four test suites that are available in the **tests\_mycalc** directory:

```
python -m unittest discover myunittest/tests/tests_mycalc
```

This command will be executed recursively, which means it can discover the test cases in sub-directories as well. The other (optional) parameters can be used to select a set of test cases for execution, and these are described as follows:

- **-v**: To make the output verbose.
- **-s**: Start directory for the discovery of test cases.
- **-p**: Pattern to use for searching the test files. The default is **test\*.py**, but it can be changed by this parameter.

- **-t**: This is a top-level directory of the project. If not specified, the start directory is the top-level directory

Although the command-line option of running multiple test suites is simple and powerful, we sometimes need to control the way we run selected tests from different test suites that may be in different locations. This is where loading and executing the test cases through the Python code is handy. The next code snippet is an example of how to load the test suites from a class name, find the test cases in each of the suites, and then run them using the **unittest** test runner:

```
import unittest

from test_mycalc_add import MyCalcAddTestSuite
from test_mycalc_subtract import MyCalcSubtractTestSuite
from test_mycalc_multiply import MyCalcMultiplyTestSuite
from test_mycalc_divide import MyCalcDivideTestSuite

def run_mytests():

 test_classes = [MyCalcAddTestSuite, \
 MyCalcSubtractTestSuite, \
 MyCalcMultiplyTestSuite, MyCalcDivideTestSuite]

 loader = unittest.TestLoader()
 test_suites = []
 for t_class in test_classes:
 suite = loader.loadTestsFromTestCase(t_class)
 test_suites.append(suite)

 final_suite = unittest.TestSuite(test_suites)
 runner = unittest.TextTestRunner()
 results = runner.run(final_suite)

if __name__ == '__main__':
 run_mytests()
```

In this section, we have covered building test cases using the **unittest** library. In the next section, we will work with the **pytest** library.

## Working with the pytest framework

The test cases written using the **unittest** library are easier to read and manage, especially if you are coming from a background of using JUnit or other similar frameworks. But for large-scale Python applications, the **pytest** library stands out as one of the most popular frameworks, mainly because of

its ease of use in implementation and its ability to extend for complex testing requirements. In the case of the **pytest** library, there is no requirement to extend the unit test class from any base class; in fact, we can write the test cases without even implementing any class.

**pytest** is an open source framework. The **pytest** test framework can auto-discover tests, just as with the **unittest** framework, if the filename has a **test** prefix, and this discovery format is configurable. The **pytest** framework includes the same level of functionality as it is provided by the **unittest** framework for writing unit tests. In this section, we will focus on discussing the features that are different or additional in the **pytest** framework.

### **Building test cases without a base class**

To demonstrate how to write unit test cases using the **pytest** library, we will revise our **myadd2.py** module by implementing the **add** function without a class. This new **add** function will add two numbers and throw an exception if the *numbers* are not passed as arguments. The test-case code using the **pytest** framework is shown in the following snippet:

```
myadd3.py is a class with add two numbers method
def add(self, x, y):
 """This function adds two numbers"""
 if (not isinstance(x, (int, float))) | \
 (not isinstance(y, (int, float))):
 raise TypeError("only numbers are allowed")
 return x + y
```

And the test cases' module is shown next, as follows:

```
#test_myadd3.py test suite for myadd function
import pytest
from mypytest.src.myadd3 import add
def test_add1():
 """ test case to validate two positive numbers"""
 assert add(10, 5) == 15
def test_add2():
 """ test case to validate two positive numbers"""
 assert add(10, -5) == 5, "should be 5"
```

We only showed two test cases for the **test\_myadd3.py** module as the other test cases will be similar to the first two test cases. These additional test cases are available with this chapter's source code

under the GitHub directory. A couple of key differences in the test case implementation are outlined here:

- There is no requirement to implement test cases under a class, and we can implement test cases as class methods without inheriting them from any base class. This is a key difference in comparison to the **unittest** library.
- The **assert** statements are available as a keyword for validation of any condition to declare whether a test passed or failed. Separating **assert** keywords from the conditional statement makes assertions in test cases very flexible and customizable.

It is also important to mention that the console output and the reporting is more powerful with the **pytest** framework. As an example, the console output of executing test cases using the **test\_myadd3.py** module is shown here:

```
test_myadd3.py::test_add1 PASSED [25%]
test_myadd3.py::test_add2 PASSED [50%]
test_myadd3.py::test_add3 PASSED [75%]
test_myadd3.py::test_add4 PASSED [100%]
=====
===== 4 passed in 0.03s =====
```

Next, we will investigate how to validate expected errors using the **pytest** library.

## Building test cases with error handling

Writing test cases to validate the throwing of an expected exception or error is different in the **pytest** framework as compared to writing such test cases in the **unittest** framework. The **pytest** framework utilizes the context manager for exception validation. In our **test\_myadd3.py** test module, we already added two test cases for exception validation. An extract of the code in the **test\_myadd3.py** module with the two test cases is shown next, as follows:

```
def test_typeerror1():
 """ test case to check if we can handle non number \
 input"""
 with pytest.raises(TypeError):
 add('a', 5)

def test_typeerror2():
 """ test case to check if we can handle non number \
 input"""
 with pytest.raises(TypeError, match="only numbers are \
 allowed"):
 add('a', 'b')
```

To validate the exception, we are using the **raises** function of the **pytest** library to indicate what sort of exception is expected by running a certain unit of code (**add('a', 5)** in our first test case). In the

second test case, we used a **match** argument to validate the message that is set when an exception is thrown.

Next, we will discuss how to use markers with the **pytest** framework.

## Building test cases with pytest markers

The **pytest** framework is equipped with markers that allow us to attach metadata or define different categories for our test cases. This metadata can be used for many purposes, such as including or excluding certain test cases. The markers are implemented using the **@pytest.mark** decorator.

The **pytest** framework provides a few built-in markers, with the most popular ones being described next:

- **skip**: The test runner will skip a test case unconditionally when this marker is used.
- **skipif**: This marker is used to skip a test based on a conditional expression that is passed as an argument to this marker.
- **xfail**: This marker is used to ignore an expected failure in a test case. It is used with a certain condition.
- **parametrize**: This marker is used to perform multiple calls to the test case with different values as arguments.

To demonstrate the use of the first three markers, we rewrite our **test\_add3.py** module by adding markers with the test-case functions. The revised test-case module (**test\_add4.py**) is shown here:

```
@pytest.mark.skip
def test_add1():
 """ test case to validate two positive numbers"""
 assert add(10, 5) == 15
@pytest.mark.skipif(sys.version_info > (3,6), \
reason=" skipped for release > than Python 3.6")
def test_add2():
 """ test case to validate two positive numbers"""
 assert add(10, -5) == 5, "should be 5"
@pytest.mark.xfail(sys.platform == "win32", \
reason="ignore exception for windows")
def test_add3():
 """ test case to validate two positive numbers"""
 assert add(-10, 5) == -5
 raise Exception()
```

We used the **skip** marker unconditionally for the first test case. This will ignore the test case. For the second test case, we used the **skipif** marker with a condition of a Python version greater than 3.6. For the last test case, we deliberately raised an exception, and we used the **xfail** marker to ignore this type

of exception if the system platform is Windows. This type of marker is helpful for ignoring errors in test cases if they are expected for a certain condition, such as the operating system in this case.

The console output from the execution of the test cases is shown here:

```
test_myadd4.py::test_add1 SKIPPED (unconditional skip) [33%]
Skipped: unconditional skip

test_myadd4.py::test_add2 SKIPPED (skipped for release > than
Python...) [66%]
Skipped: skipped for release > than Python 3.6

test_myadd4.py::test_add3 XFAIL (ignore exception for
mac) [100%]
@ pytest.mark.xfail(sys.platform == "win32",
 reason="ignore exception for mac")
=====
2 skipped, 1 xfailed in 0.06s =====
```

Next, we will discuss the use of the **parametrize** marker with the **pytest** library.

## Building test cases with parametrization

In all previous code examples, we built test-case functions or methods without passing any parameters to them. But for many test scenarios, we need to run the same test case by varying the input data. In a classical approach, we run multiple test cases that are different only in terms of the input data we used for them. Our previous example of **test\_myadd3.py** shows how to implement test cases using this classical approach. A recommended approach for such type of testing is to use **data-driven testing (DDT)**. DDT is a form of testing in which the test data is provided through a table, a dictionary, or a spreadsheet to a single test case. This type of testing is also called **table-driven testing** or **parametrized testing**. The data provided through a table or a dictionary is used to execute the tests using a common implementation of the test source code. DDT is beneficial in scenarios when we have to test functionality by using a permutation of input parameters. Instead of writing test cases for each permutation of input parameters, we can provide the permutations in a table or a dictionary format and use it as input to our single test case. Frameworks such as **pytest** will execute our test case as many times as the number of permutations is in the table or the dictionary. A real-world example of DDT is to validate the behavior of a login feature of an application by using a variety of users with valid and invalid credentials.

In the **pytest** framework, DDT can be implemented using parametrization with the **pytest** marker. By using the **parametrize** marker, we can define which input argument we need to pass and also the test dataset we need to use. The **pytest** framework will automatically execute the test-case function multiple times as per the number of entries in the test data provided with the **parametrize** marker.

To illustrate how to use the **parametrize** marker for DDT, we will revise our **myadd4.py** module for the test cases of the **add** function. In the revised code, we will have only one test-case function but different test data to be used for the input parameters, as illustrated in the following snippet:

```
test_myadd5.py test suite using parameterize marker
import sys
import pytest
from mypytest.src.myadd3 import add
@pytest.mark.parametrize("x,y,ans",
[(10,5,15),(10,-5,5),
(-10,5,-5),(-10,-5,-15)],
ids=["pos-pos", "pos-neg",
"neg-pos", "neg-neg"])

def test_add(x, y, ans):
 """ test case to validate two positive numbers"""
 assert add(x, y) == ans
```

For the **parametrize** marker, we used three parameters, which are described as follows:

- **Test-case arguments:** We provide a list of arguments to be passed to our test function in the same order as defined with the test-case function definition. Also, the test data we need to provide in the next argument will follow the same order.
- **Data:** The test data to be passed will be a list of different sets of input arguments. The number of entries in the test data will determine how many times the test case will be executed.
- **ids:** This is an optional parameter that is mainly attaching a friendly tag to different test datasets we provided in the previous argument. These **identifier (ID)** tags will be used in the output report to identify different executions of the same test case.

The console output for this test-case execution is shown next:

```
test_myadd5.py::test_add[pos-pos] PASSED [25%]
test_myadd5.py::test_add[pos-neg] PASSED [50%]
test_myadd5.py::test_add[neg-pos] PASSED [75%]
test_myadd5.py::test_add[neg-neg] PASSED [100%]
===== 4 passed in 0.04s =====
```

This console output shows us how many times the test case is executed and with which test data. The test cases built using the **pytest** markers are concise and easy to implement. This saves a lot of time and enables us to write more test cases (by varying data only) in a short time.

Next, we will discuss another important feature of the **pytest** library: fixtures.

## Building test cases with pytest fixtures

In the **pytest** framework, the test fixtures are implemented using Python decorators (**@pytest.fixture**). The implementation of test fixtures in the **pytest** framework is very powerful as compared to the other frameworks for the following key reasons:

- Fixtures in the **pytest** framework provide high scalability. We can define a generic setup or fixtures (methods) that can be reused across functions, classes, modules, and packages.
- Fixture implementation of the **pytest** framework is modular in nature. We can use one or more fixtures with a test case. A fixture can use one or many other fixtures as well, just as we use functions to call other functions.
- Each test case in a test suite will have the flexibility to use the same or a different set of fixtures.
- We can create fixtures in the **pytest** framework with a scope set for them. The default scope is **function**, which means the fixture will be executed before every function (test case). Other scope options are **module**, **class**, **package**, or **session**. These are defined briefly next:
  - a) **Function**: The fixture is destroyed after executing a test case.
  - b) **Module**: The fixture is destroyed after executing the last test case in a module.
  - c) **Class**: The fixture is destroyed after executing the last test case in a class.
  - d) **Package**: The fixture is destroyed after executing the last test case in a package.
  - e) **Session**: The fixture is destroyed after executing the last test case in a test session.

The **pytest** framework has a few useful built-in fixtures that can be used out of the box, such as **capfd** to capture output to the file descriptors, **capsys** to capture output to **stdout** and **stderr**, **request** to provide information on the requesting test function, and **testdir** to provide a temporary test directory for test executions.

Fixtures in the **pytest** framework can be used to reset or tear down at the end of a test case as well. We will discuss this later on in this section.

In the next code example, we will build test cases for our **MyCalc** class using custom fixtures. The sample code for **MyCalc** is already shared in the *Executing multiple test suites* section. The implementation of a test fixture and test cases is shown here:

```
test_mycalc1.py test calc functions using test fixture

import sys

import pytest

from mypytest.src.myadd3 import add

from mypytest.src.mycalc import MyCalc

@pytest.fixture(scope="module")

def my_calc():

 return MyCalc()
```

```

@pytest.fixture

def test_data():
 return {'x':10, 'y':5}

def test_add(my_calc, test_data):
 """ test case to add two numbers"""
 assert my_calc.add(test_data.get('x'), \
 test_data.get('y')) == 15

def test_subtract(my_calc, test_data):
 """ test case to subtract two numbers"""
 assert my_calc.subtract(test_data.get('x'), \
 test_data.get('y'))== 5

```

In this test-suite example, these are the key points of discussion:

- We created two fixtures: **my\_calc** and **test\_data**. The **my\_calc** fixture is set with a scope set to **module** because we want it to be executed only once to provide an instance of the **MyCalc** class. The **test\_data** fixture is using the default scope (**function**), which means it will be executed before every method.
- For the test cases (**test\_add** and **test\_subtract**), we used the fixtures as input arguments. The name of the argument has to match the fixture function name. The **pytest** framework automatically looks for a fixture with the name used as an argument for a test case.

The code example we discussed is using a fixture as the setup function. A question we may want to ask is: *How we can achieve teardown functionality with the pytest fixtures?* There are two approaches available for implementing the teardown functionality, and these are discussed next.

### **Using yield instead of a return statement**

With this approach, we write some code mainly for setup purposes, use a **yield** statement instead of **return**, and then write code for teardown purposes after the **yield** statement. If we have a test suite or module with many fixtures used in it, the **pytest** test runner will execute each fixture (as per the evaluated order of execution) till the **yield** statement is encountered. As soon as the test-case execution is completed, the **pytest** test runner triggers the execution of all fixtures that are yielded and executes the code that is written after the **yield** statement. The use of a yield-based approach is clean in the sense that the code is easy to follow and maintain. Therefore, it is a recommended approach.

### **Adding a finalizer method using the request fixture**

With this approach, we have to consider three steps to write a teardown method, outlined as follows:

- We have to use a **request** object in our fixtures. The **request** object can be provided using the built-in fixture with the same name.
- We will define a **teardown** method, separately or as a part of the fixture implementation.
- We will provide the **teardown** method as a callable method to the request object using the **addfinalizer** method.

To illustrate both approaches with code examples, we will modify our previous implementation of the fixtures. In the revised code, we will implement the **my\_calc** fixture using a **yield** approach and the **data\_set** fixture using an **addfinalizer** approach. Here is the revised code example:

```
test_mycalc2.py test calc functions using test fixture
<import statements>
@pytest.fixture(scope="module")
def my_calc():
 my_calc = MyCalc()
 yield my_calc
 del my_calc
@pytest.fixture
def data_set(request):
 dict = {'x':10, 'y':5}
 def delete_dict(obj):
 del obj
 request.addfinalizer(lambda: delete_dict(dict))
 return dict
<rest of the test cases>
```

Note that there is no real need for teardown functionality for these example fixtures, but we added them for illustration purposes.

### *TIP*

*Using nose and doctest for test automation is similar to using the unittest and pytest frameworks.*

In the next section, we will discuss a TDD approach to software development.

## Executing TDD

TDD is a well-known practice in software engineering. This is a software development approach in which test cases are written first before writing any code for a required feature in an application. Here are the three simple rules of TDD:

- Do not write any functional code unless you write a unit test that is failing.

- Do not write any additional code in the same test more than you need to make the test fail.
- Do not write any functional code more than what is needed to pass a failing test.

These TDD rules also drive us to follow a famous three-phase approach of software development called **Red**, **Green**, **Refactor**. The phases are repeated continuously for TDD. These three phases are shown in *Figure 5.4* and are described next.

## Red

In this phase, the first step is to write a test without having any code to test. The test will obviously fail in this case. We will not try to write a complete test case but only write enough code to fail the test.

## Green

In this phase, the first step is to write the code until an already written test passes. Again, we will only write enough code to pass the test. We will run all tests to make sure previously written tests also pass.

## Refactor

In this phase, we should consider improving the quality of the code, which means making the code easy to read and use optimization—for example, any hardcoded values have to be removed. Running the tests after each refactoring cycle is also recommended. The outcome of the refactor phase is clean code. We can repeat the cycle by adding more test scenarios and adding code to make the new test pass, and this cycle must be repeated until a feature is developed.

It is important to understand that TDD is neither a testing nor a design approach. It is an approach to developing software according to specifications that are defined by writing test cases first.

The following diagram shows the three phases of TDD:

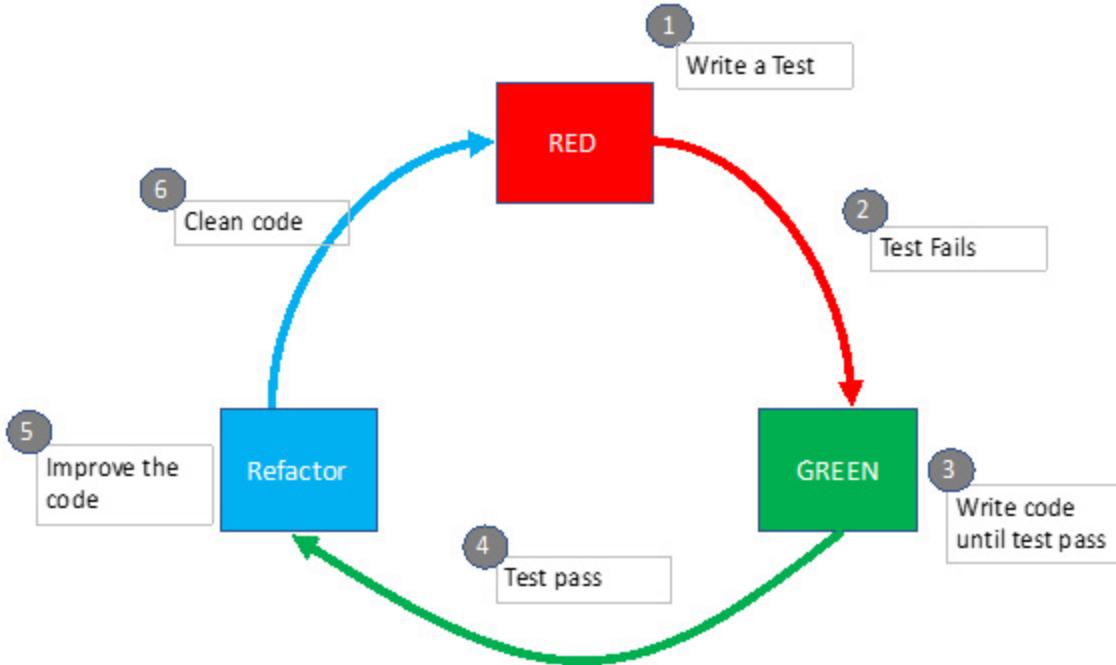


Figure 5.4 – TDD, also known as Red, Green, Refactor

In the next section, we will introduce the role of test automation in the CI process.

## Introducing automated CI

CI is a process that combines the benefits of both automated testing and version control systems to achieve a fully automated integration environment. With a CI development approach, we integrate our code into a shared repository frequently. Every time we add our code to a repository, the following two processes are expected to kick in:

- An automated build process starts to validate that the newly added code is not breaking anything from a compilation or syntax point of view.
- An automated test execution starts to verify that the existing, as well as new functionality is as per the test cases defined.

The different steps and phases of the CI process are depicted in the following diagram. Although we have shown the build phase in this flowchart, it is not a required phase for Python-based projects as we can execute integration tests without compiled code:

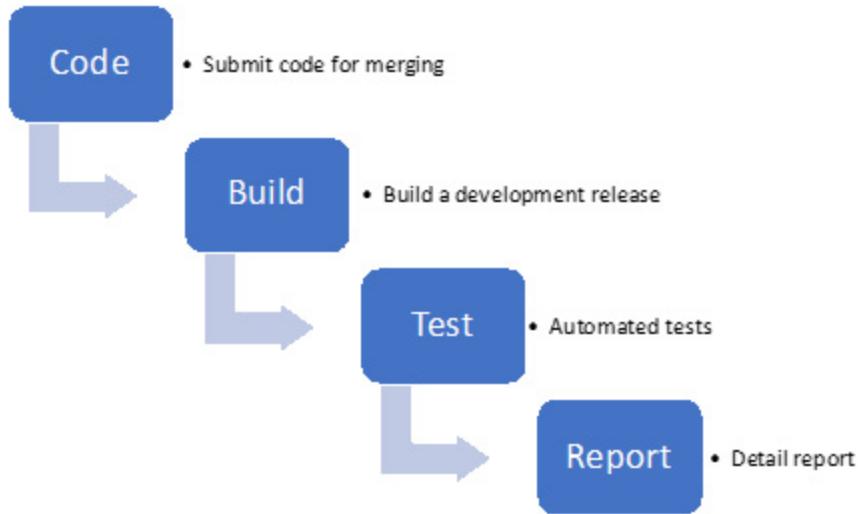


Figure 5.5 – Phases of CI testing

To build a CI system, we need to have a stable distributed version control and a tool that can be used to implement workflow for testing the whole application through a series of test suites. There are several commercial and open source software tools available that provide CI and **continuous delivery (CD)** functionality. These tools are designed for easy integration with a source control system and with a test automation framework. A few popular tools available for CI are *Jenkins*, *Bamboo*, *Buildbot*, *GitLab CI*, *CircleCI*, and *Buddy*. Details of these tools appear in the *Further reading* section, for those of you who are interested to learn more.

The obvious benefits of this automated CI are to detect errors quickly and fix them more conveniently right at the beginning. It is important to understand that CI is not about bug fixing, but it definitely helps to identify bugs easily and get them fixed promptly.

## Summary

In this chapter, we introduced different levels of testing for software applications. We also evaluated two test frameworks (**unittest** and **pytest**) that are available for Python test automation. We learned how to build basic- and advanced-level test cases using these two frameworks. Later in the chapter, we introduced the TDD approach and its clear benefits for software development. Finally, we touched base on the topic of CI, which is a key step in delivering software using **agile** and **development-operations (devops)** models.

This chapter is useful for anyone who wants to start writing unit tests for their Python application. The code examples provided provide a good starting point for us to write test cases using any test framework.

In the next chapter, we will explore different tricks and tips for developing applications in Python.

## Questions

1. Is unit testing a form of white-box or black-box testing?
2. When should we use mock objects?
3. Which methods are used to implement test fixtures with the **unittest** framework?
4. How is TDD different from CI?
5. When should we use DDT?

## Further reading

- *Learning Python Testing*, by Daniel Arbuckle
- *Test-Driven Development with Python*, by Harry J.W. Percival
- *Expert Python Programming*, by Michał Jaworski and Tarek Ziadé
- *unittest* framework details are available with the Python documentation at <https://docs.python.org/3/library/unittest.html>.

## Answers

1. White-box testing
2. Mock objects help simulate the behavior of external or internal dependencies. By using mock objects, we can focus on writing tests for validating functional behavior.
3. **setUp**, **tearDown**, **setUpClass**, **tearDownClass**
4. TDD is an approach to developing software by writing the test cases first. CI is a process in which all the tests are executed every time we build a new release. There is no direct relationship between TDD and CI.
5. DDT is used when we have to do functional testing with several permutations of input parameters. For example, if we are required to test an API endpoint with a different combination of arguments, we can leverage DDT.

## *Chapter 6: Advanced Tips and Tricks in Python*

In this chapter, we will introduce some advanced tips and tricks that can be used as powerful programming techniques when writing code in Python. These include the advanced use of Python functions, such as nested functions, lambda functions, and building decorators with functions. Additionally, we will cover data transformations with the filter, mapper, and reducer functions. This will be followed by some tricks that can be used with data structures, such as the use of nested dictionaries and comprehension with different collection types. Finally, we will investigate the advanced functionality of the pandas library for DataFrame objects. These advanced tips and tricks will not only demonstrate Python's power in achieving advanced features with less code, but it will also help you code faster and more efficiently.

In this chapter, we will cover the following topics:

- Learning advanced tricks for using functions
- Understanding advanced concepts with data structures
- Introducing advanced tricks with pandas DataFrame

By the end of this chapter, you will have gained an understanding of how to use Python functions for advanced features such as data transformations and building decorators. Additionally, you will learn how to use data structures including pandas DataFrame for analytics-based applications.

## **Technical requirements**

The technical requirements for this chapter are as follows:

- You need to have Python 3.7 or later installed on your computer.
- You need to register an account with TestPyPI and create an API token under your account.

The sample code for this chapter can be found at <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter06>.

We will start our discussion with the advanced concepts for using functions in Python.

## **Learning advanced tricks for using functions**

The use of functions in Python and other programming languages is key for reusability and modularization. However, with new advances to modern programming languages, the role of functions has been extended beyond reusability, which includes writing simple, short, and concise code without using complex loops and conditional statements.

We will start with the use of the **counter**, **zip**, and **itertools** functions, which we will discuss next.

## Introducing the counter, itertools, and zip functions for iterative tasks

For any data processing tasks, developers extensively use iterators. We have covered iterators, in detail, in [Chapter 4, Python Libraries for Advanced Programming](#). In this section, we will learn about the next level of utility functions to help you conveniently work with iterators and iterables. These include the **counter** module, the **zip** function, and the **itertools** module. We will discuss each of these in the following subsections.

### Counter

**Counter** is a type of container that keeps track of the count of each element that is present in a container. The count of elements in a container is useful for finding the data frequency, which is a prerequisite for many data analysis applications. To illustrate the concept and use of the **Counter** class, we will present a simple code example, as follows:

```
#counter.py
from collections import Counter
#applying counter on a string object
print(Counter("people"))
#applying counter on a list object
my_counter = Counter([1,2,1,2,3,4,1,3])
print(my_counter.most_common(1))
print(list(my_counter.elements()))
#applying counter on a dict object
print(Counter({'A': 2, 'B': 2, 'C': 2, 'C': 3}))
```

In the preceding code example, we created multiple **Counter** instances using a **String** object, a list object, and a dictionary object. The **Counter** class has methods such as **most\_common** and **elements**. We used the **most\_common** method with a value of **1**, which gives us the element that appears the most in the **my-counter** container. Additionally, we used the **elements** method to return the original list from the **Counter** instance. The console output of this program should be as follows:

```
Counter({'p': 2, 'e': 2, 'o': 1, 'l': 1})
[(1, 3)]
[1, 1, 1, 2, 2, 3, 3, 4]
Counter({'C': 4, 'A': 2, 'B': 2})
```

It is important to note that in the case of the dictionary object, we deliberately used a repeated key, but in the **Counter** instance, we get only one key-value pair, which is the last one in the dictionary. Additionally, the elements in the **Counter** instance are ordered based on the values for each element. Note that the **Counter** class converts the dictionary object into a hashtable object.

## zip

The **zip** function is used to create an aggregated iterator based on two or more individual iterators. The **zip** function is useful when we are required to iterate on multiple iterations in parallel. For example, we can use the **zip** function when implementing mathematical algorithms that involve interpolation or pattern recognition. This is also helpful in digital signal processing where we combine multiple signals (data sources) into a single signal. Here is a simple code example that uses a **zip** function:

```
#zip.py

num_list = [1, 2, 3, 4, 5]
lett_list = ['alpha', 'bravo', 'charlie']
zipped_iter = zip(num_list, lett_list)
print(next(zipped_iter))
print(next(zipped_iter))
print(list(zipped_iter))
```

In the preceding code example, we combined the two lists for iteration purposes by using the **zip** function. Note that one list is larger than the other in terms of the number of elements. The console output of this program should be as follows:

```
(1, 'alpha')
(2, 'bravo')
[(3, 'charlie'), (4, 'delta')]
```

As expected, we get the first two tuples using the **next** function, which is a combination of corresponding elements from each list. In the end, we used the **list** constructor to iterate over the rest of the tuples from the **zip** iterator. This gives us a list of the remaining tuples in a list format.

## itertools

Python offers a module, called **itertools**, that provides useful functions to work with iterators. When working with a large set of data, the use of iterators is a must, and that is where the utility functions provided by the **itertools** module prove to be very helpful. There are many functions available with the **itertools** module. We will briefly introduce a few key functions here:

- **count**: This function is used to create an iterator for counting numbers. We can provide a starting number (default = 0) and, optionally, set a size of the counting step for the increment. The following code example will return an iterator that provides

counting numbers, such as 10, 12, and 14:

```
#itertools_count.py
import itertools
iter = itertools.count(10, 2)
print(next(iter))
print(next(iter))
```

- **cycle:** This function allows you to cycle through an iterator endlessly. The following code snippet illustrates how you can use this function for a list of alphabet letters:

```
letters = ['A', 'B', 'C']
for letter in itertools.cycle(letters):
 print(letter)
```

- **Repeat:** This function provides us with an iterator that returns an object over and over again unless there is a **times** argument set with it. The following code snippet will repeat the **Python** string object five times:

```
for x in itertools.repeat('Python', times=5):
 print(x)
```

- **accumulate:** This function will return an iterator that provides us with an accumulated sum or other accumulated results based on an aggregator function that was passed to this **accumulate** function as an argument. It is easier to understand the use of this function with a code example, as follows:

```
#itertools_accumulate.py
import itertools, operator
list1 = [1, 3, 5]
res = itertools.accumulate(list1)
print("default:")
for x in res:
 print(x)
res = itertools.accumulate(list1, operator.mul)
print("Multiply: ")
for x in res:
 print(x)
```

In this code example, first, we used the **accumulate** function without providing an aggregator function for any accumulated results. By default, the **accumulate** function will add two numbers (**1** and **3**) from the original list. This process is repeated for all numbers, and the results are stored inside an iterable (in our case, this is **res**). In the second part of this code example, we provided the **mul** (multiplication) function from the **operator** module, and this time, the accumulated results are based on the multiplication of two numbers.

- **chain**: This function combines two or more iterables and returns a combined iterable. Take a look at the following example code showing two iterables (lists) along with the **chain** function:

```
list1 = ['A', 'B', 'C']
list2 = ['W', 'X', 'Y', 'Z']
chained_iter = itertools.chain(list1, list2)
for x in chained_iter:
 print(x)
```

Note that this function will combine the iterables in a serial manner. This means that items in **list1** will be accessible first, followed by the items in **list2**.

- **compress**: This function can be used to filter elements from one iterable based on another iterable. In the example code snippet, we have selected alphabet letters from a list based on a **selector** iterable:

```
letters = ['A', 'B', 'C']
selector = [True, 0, 1]
for x in itertools.compress(letters, selector):
 print (x)
```

For the **selector** iterable, we can use **True/False** or **1/0**. The output of this program will be the letters **A** and **C**.

- **groupby**: This function identifies the keys for each item in an iterable object and groups the items based on the identified keys. This function requires another function (known as **key\_func**) that identifies a key in each element of an iterable object. The following example code explains the use of this function along with how to implement a **key\_func** function:

```
#itertools_groupby.py
import itertools
mylist = [("A", 100), ("A", 200), ("B", 30), \
 ("B", 10)]
def get_key(group):
 return group[0]
for key, grp in itertools.groupby(mylist, get_key):
 print(key + "-->", list(grp))
```

- **tee**: This is another useful function that can be used to duplicate iterators from a single iterator. Here is an example code that duplicates two iterators from a single list iterable:

```
letters = ['A', 'B', 'C']
iter1, iter2 = itertools.tee(letters)
for x in iter1:
 print(x)
for x in iter2:
```

```
print(x)
```

Next, we will discuss another category of functions that is extensively used for data transformation.

## Using filters, mappers, and reducers for data transformations

**map**, **filter**, and **reduce** are three functions available in Python that are used to simplify and write concise code. These three functions are applied to iterables in a single shot without using iterative statements. The **map** and **filter** functions are available as built-in functions, while the **reduce** function requires you to import the **functools** module. These functions are extensively used by data scientists for data processing. The **map** function and the **filter** function are used to transform or filter data, whereas the **reduce** function is used in data analysis to get meaningful results from a large dataset.

In the following subsections, we will evaluate each function with its application and code examples.

### **map**

The **map** function in Python is defined using the following syntax:

```
map(func, iter, ...)
```

The **func** argument is the name of the function that will be applied to each item of the **iter** object. The three dots indicate that it is possible to pass multiple iterable objects. However, it is important to understand that the number of arguments of the function (**func**) must match the number of iterable objects. The output of the **map** function is a **map** object, which is a generator object. The return value can be converted into a list by passing the **map** object to the **list** constructor.

#### *IMPORTANT NOTE*

*In Python 2, the **map** function returns a list. This behavior has been changed in Python 3.*

Before discussing the use of a **map** function, first, we will implement a simple transformation function that converts a list of numbers into their square values. The code example is provided next:

```
#map1.py to get square of each item in a list
mylist = [1, 2, 3, 4, 5]
new_list = []
for item in mylist:
 square = item*item
 new_list.append(square)
print(new_list)
```

Here, the code example uses a **for** loop structure to iterate through a list, calculates the square of each entry in the list, and then adds it to a new list. This style of writing code is common, but it is definitely not a Pythonic way to write code. The console output of this program is as follows:

```
[1, 4, 9, 16, 25]
```

With the use of the **map** function, this code can be simplified and shortened, as follows:

```
map2.py to get square of each item in a list
def square(num):
 return num * num
mylist = [1, 2, 3, 4, 5]
new_list = list(map(square, mylist))
print(new_list)
```

By using the **map** function, we provided the name of the function (in this example, it is **square**) and the reference of the list (in this example, it is **mylist**). The **map** object that is returned by the **map** function is converted into a list object by using the **list** constructor. The console output of this code example is the same as the previous code example.

In the following code example, we will provide two lists as input for the **map** function:

```
map3.py to get product of each item in two lists
def product(num1, num2):
 return num1 * num2
mylist1 = [1, 2, 3, 4, 5]
mylist2 = [6, 7, 8, 9]
new_list = list(map(product, mylist1, mylist2))
print(new_list)
```

This time, the goal of the **map** function that has been implemented is to use the **product** function. The **product** function takes each item from two lists and multiplies the corresponding item in each list before returning it to the **map** function.

The console output of this code example is as follows:

```
[6, 14, 24, 36]
```

An analysis of this console output tells us that only the first four items from each list are used by the **map** function. The **map** function automatically stops when it runs out of the items in any of the iterables (in our case, these are the two lists). This means that even if we provide iterables of different sizes, the **map** function will not raise any exception but will work for the number of items that are possible to map across iterables using the function provided. In our code example, we have a smaller

number of items in the **mylist2** list, which is four. That is why we only have four items in the output list (in our case, this is **new\_list**). Next, we will discuss the **filter** function with some code examples.

## **filter**

The **filter** function also operates on iterables but only on one iterable object. As its name suggests, it provides a filtering functionality on the iterable object. The filtering criteria are provided through the function definition. The syntax of a **filter** function is as follows:

```
filter (func, iter)
```

The **func** function provides the filtering criteria, and it has to return **True** or **False**. Since only one iterable is allowed alongside the **filter** function, only one argument is allowed for the **func** function. The following code example uses a **filter** function to select the items whose values are even numbers. To implement the selection criteria, the function **is\_even** is implemented to evaluate whether a number provided to it is an even number or not. The sample code is as follows:

```
filter1.py to get even numbers from a list

def is_even(num):
 return (num % 2 == 0)

mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9]
new_list = list(filter(is_even, mylist))
print(new_list)
```

The console output of the preceding code example is as follows:

```
[2, 4, 6, 8]
```

Next, we will discuss the **reduce** function.

## **reduce**

The **reduce** function is used to apply a cumulative processing function on each element of a sequence, which is passed to it as an argument. This cumulative processing function is not for transformation or filtration purposes. As its name suggests, the cumulative processing function is used to get a single result at the end based on all of the elements in a sequence. The syntax of using the **reduce** function is as follows:

```
reduce (func, iter[,initial])
```

The **func** function is a function that is used to apply cumulative processing on each element of the iterable. Additionally, **initial** is an optional value that can be passed to the **func** function to be used as an initial value for cumulative processing. It is important to understand that there will always be two arguments to the **func** function for the **reduce** function case: the first argument will either be the

initial value (if provided) or the first element of the sequence, and the second argument will be the next element from the sequence.

In the following code example, we will use a simple list of the first five numbers. We will implement a custom method to add the two numbers and then use the **reduce** method to sum all of the elements in the list. The code example is shown next:

```
reduce1.py to get sum of numbers from a list
from functools import reduce
def seq_sum(num1, num2):
 return num1+num2
mylist = [1, 2, 3, 4, 5]
result = reduce(seq_sum, mylist)
print(result)
```

The output of this program is **15**, which is a numerical sum of all the elements of the list (in our example, this is called **mylist**). If we provide the initial value to the **reduce** function, the result will be appended as per the initial value. For example, the output of the same program with the following statement will be **25**:

```
result = reduce(seq_sum, mylist, 10)
```

As mentioned previously, the result or return value of the **reduce** function is a single value, which is as per the **func** function. In this example, it will be an integer.

In this section, we discussed the **map**, **filter**, and **reduce** functions that are available within Python. These functions are used extensively by data scientists for data transformation and data refinement. One problem of using functions such as **map** and **filter** is that they return an object of the **map** or **filter** type, and we have to convert the results explicitly into a **list** data type for further processing. The comprehensions and generators do not have such limitations but provide similar functionality, and they are relatively easier to use. That is why they are getting more traction than the **map**, **filter**, and **reduce** functions. We will discuss comprehension and generators in the *Understanding advanced concepts with data structures* section. Next, we will investigate the use of lambda functions.

## Learning how to build lambda functions

Lambda functions are anonymous functions that are based on a single-line expression. Just as the **def** keyword is used to define regular functions, the **lambda** keyword is used to define anonymous functions. Lambda functions are restricted to a single line. This means they cannot use multiple

statements, and they cannot use a return statement. The return value is automatically returned after the evaluation of the single-line expression.

The lambda functions can be used anywhere a regular function is used. The easiest and most convenient usage of lambda functions is with the **map**, **reduce**, and **filter** functions. Lambda functions are helpful when you wish to make the code more concise.

To illustrate a lambda function, we will reuse the map and filter code examples that we discussed earlier. In these code examples, we will replace **func** with a lambda function, as highlighted in the following code snippet:

```
lambda1.py to get square of each item in a list
mylist = [1, 2, 3, 4, 5]
new_list = list(map(lambda x: x*x, mylist))
print(new_list)

lambda2.py to get even numbers from a list
mylist = [1, 2, 3, 4, 5, 6, 7, 8, 9]
new_list = list(filter(lambda x: x % 2 == 0, mylist))
print(new_list)

lambda3.py to get product of corresponding item in the\
two lists
mylist1 = [1, 2, 3, 4, 5]
mylist2 = [6, 7, 8, 9]
new_list = list(map(lambda x,y: x*y, mylist1, mylist2))
print(new_list)
```

Although the code has become more concise, we should be careful about using lambda functions. These functions are not reusable, and they are not easy to maintain. We need to rethink this before introducing a lambda function into our program. Any changes or additional functionality will not be easy to add. A rule of thumb is to only use lambda functions for simple expressions when writing a separate function would be an overhead.

## Embedding a function within another function

When we add a function within an existing function, it is called an **inner function** or a **nested function**. The advantage of having inner functions is that they have direct access to the variables that are either defined or available in the scope of an outer function. Creating an inner function is the same as defining a regular function with the **def** keyword and with the appropriate indentation. The

inner functions cannot be executed or called by the outside program. However, if the outer function returns a reference of the inner function, it can be used by the caller to execute the inner function. We will take a look at examples of returning inner function references for many use cases in the following subsections.

Inner functions have many advantages and applications. We will describe a few of them next.

## Encapsulation

A common use case of an inner function is being able to hide its functionality from the outside world. The inner function is only available within the outer function scope and is not visible to the global scope. The following code example shows one outer function that is hiding an inner function:

```
#inner1.py

def outer_hello():

 print ("Hello from outer function")

 def inner_hello():

 print("Hello from inner function")

 inner_hello()

 outer_hello()
```

From the outside of the outer function, we can only call the outer function. The inner function can only be called from the body of the outer function.

## Helper functions

In some cases, we can find ourselves in a situation where the code within a function code is reusable. We can turn such reusable code into a separate function; otherwise, if the code is reusable only within the scope of a function, then it is a case of building an inner function. This type of inner function is also called the helper function. The following code snippet illustrates this concept:

```
def outer_fn(x, y):

 def get_prefix(s):
 return s[:2]

 x2 = get_prefix(x)
 y2 = get_prefix(y)

 #process x2 and y2 further
```

In the preceding sample code, we defined an inner function, called `get_prefix` (a helper function), inside an outer function to filter the first two letters of an argument value. Since we have to repeat this filtering process for all arguments, we added a helper function for reusability within the scope of this function as it is specific to this function.

## The closure and factory functions

This is a type of use case in which the inner functions shine. A **closure** is an inner function along with its enclosing environment. A closure is a dynamically created function that can be returned by another function. The real magic of a closure is that the returned function has full access to the variables and namespaces where it was created. This is true even when the enclosing function (in this context, it is the outer function) has finished executing.

The closure concept can be illustrated by a code example. The following code example shows a use case where we have implemented a closure factory to create a function to calculate the power of the base value, and the base value is retained by the closure:

```
inner2.py

def power_calc_factory(base):
 def power_calc(exponent):
 return base**exponent
 return power_calc

power_calc_2 = power_gen_factory(2)
power_calc_3 = power_gen_factory(3)

print(power_calc_2(2))
print(power_calc_2(3))
print(power_calc_3(2))
print(power_calc_3(4))
```

In the preceding code example, the outer function (that is, **power\_calc\_factory**) acts as a closure factory function because it creates a new closure every time it is called, and then it returns the closure to the caller. Additionally, **power\_calc** is an inner function that takes one variable (that is, **base**) from the closure namespace and then takes the second variable (that is, **exponent**), which is passed to it as an argument. Note that the most important statement is **return power\_calc**. This statement returns the inner function as an object with its enclosure.

When we call the **power\_calc\_factory** function for the first time along with the **base** argument, a closure is created with its namespace, including the argument that was passed to it, and the closure is returned to the caller. When we call the same function again, we get a new closure with the inner function object. In this code example, we created 2 closures: one with a **base** value of 2 and the other with a **base** value of 3. When we called the inner function by passing different values for the **exponent** variable, the code inside the inner function (in this case, the **power\_calc** function) will also have access to the **base** value that was already passed to the outer function.

These code examples illustrated the use of outer and inner functions to create functions dynamically. Traditionally, inner functions are used for hiding or encapsulating functionality inside a function. But when they are used along with the outer functions acting as a factory for creating dynamic functions, it becomes the most powerful application of the inner functions. Inner functions are also used to implement decorators. We will discuss this in more detail in the following section.

## Modifying function behavior using decorators

The concept of decorators in Python is based on the **Decorator** design pattern, which is a type of structural design pattern. This pattern allows you to add new behavior to objects without changing anything in the object implementation. This new behavior is added inside the special wrapper objects.

In Python, **decorators** are special high-order functions that enable developers to add new functionality to an existing function (or a method) without adding or changing anything within the function. Typically, these decorators are added before the definition of a function. Decorators are used for implementing many features of an application, but they are particularly popular in data validation, logging, caching, debugging, encryption, and transaction management.

To create a decorator, we have to define a callable entity (that is, a function, a method, or a class) that accepts a function as an argument. The callable entity will return another function object with a decorator-defined behavior. The function that is decorated (we will call it a *decorated function* for the remainder of this section) is passed as an argument to the function that is implementing a decorator (which will be called a *decorator function* for the remainder of this section). The decorator function executes the function passed to it in addition to the additional behavior that was added as part of the decorator function.

A simple example of a decorator is shown in the following code example in which we define a decorator to add a timestamp before and after the execution of a function:

```
decorator1.py
from datetime import datetime
def add_timestamps(myfunc):
 def _add_timestamps():
 print(datetime.now())
 myfunc()
 print(datetime.now())
 return _add_timestamps
@add_timestamps
```

```
def hello_world():
 print("hello world")
hello_world()
```

In this code example, we define a **add\_timestamps** decorator function that takes any function as an argument. In the inner function (**\_add\_timestamps**), we take the current time before and after the execution of the function, which is then passed as an argument. The decorator function returns the inner function object with a closure. The decorators are doing nothing more than using inner functions smartly, as we discussed in the previous section. The use of the @ symbol to decorate a function is equivalent to the following lines of codes:

```
hello = add_timestamps(hello_world)
hello()
```

In this case, we are calling the decorator function explicitly by passing the function name as a parameter. In other words, the decorated function is equal to the inner function, which is defined inside the decorator function. This is exactly how Python interprets and calls the decorator function when it sees a decorator with the @ symbol before the definition of a function.

However, a problem arises when we have to obtain additional details about the invocation of functions, which is important for debugging. When we use the built-in **help** function with the **hello\_world** function, we only receive help for the inner function. The same happens if we use the docstring, which will also work for the inner function but not the decorated function. Additionally, serializing the code is going to be a challenge for decorated functions. There is a simple solution that is available in Python for all of these problems; that solution is to use the **wraps** decorator from the **functools** library. We will revise our previous code example to include the **wraps** decorator. The complete code example is as follows:

```
decorator2.py
from datetime import datetime
from functools import wraps
def add_timestamps(myfunc):
 @wraps(myfunc)
 def _add_timestamps():
 print(datetime.now())
 myfunc()
 print(datetime.now())
 return _add_timestamps
@add_timestamps
```

```

def hello_world():
 print("hello world")
hello_world()
help(hello_world)
print(hello_world)

```

The use of the **wraps** decorators will provide additional details about the executions of the nested functions, and we can view these in the console output if we run the example code that has been provided.

So far, we have looked at a simple example of a decorator to explain this concept. For the remainder of this section, we will learn how to pass arguments with a function to a decorator, how to return value from a decorator, and how to chain multiple decorators. To begin, we will learn how to pass attributes and return a value with decorators.

### **Using a decorated function with a return value and argument**

When our decorated function takes arguments, then decorating such a function requires some additional tricks. One trick is to use **\*args** and **\*\*kwargs** in the inner wrapper function. This will make the inner function accept any arbitrary number of positional and keyword arguments. Here is a simple example of a decorated function with arguments along with the return value:

```

decorator3.py
from functools import wraps
def power(func):
 @wraps(func)
 def inner_calc(*args, **kwargs):
 print("Decorating power func")
 n = func(*args, **kwargs)
 return n
 return inner_calc
@power
def power_base2(n):
 return 2**n
print(power_base2(3))

```

In the preceding example, the inner function of **inner\_calc** takes the generic parameters of **\*args** and **\*\*kwargs**. To return a value from an inner function (in our code example, **inner\_calc**), we can hold the returned value from the function (in our code example, this is either **func** or **power\_base2(n)**)

that is executed inside our inner function and return the final return value from the inner function of `inner_calc`.

## Building a decorator with its own arguments

In the previous examples, we used what we call **standard decorators**. A standard decorator is a function that gets the decorated function name as an argument and returns an inner function that works as a decorator function. However, it is a bit different when we have a decorator with its own arguments. Such decorators are built on top of standard decorators. Put simply, a decorator with arguments is another function that actually returns a standard decorator (not the inner function inside a decorator). This concept of a standard decorator function wrapped within another decorator function can be understood better with a revised version of the `decorator3.py` example. In the revised version, we calculate the power of a base value that is passed as an argument to the decorator. You can view a complete code example using nested decorator functions as follows:

```
decorator4.py

from functools import wraps

def power_calc(base):

 def inner_decorator(func):
 @wraps(func)

 def inner_calc(*args, **kwargs):
 exponent = func(*args, **kwargs)
 return base**exponent

 return inner_calc

 return inner_decorator

@power_calc(base=3)

def power_n(n):
 return n

print(power_n(2))
print(power_n(4))
```

The working of this code example is as follows:

- The `power_calc` decorator function takes one argument `base` and returns the `inner_decorator` function, which is a standard decorator implementation.
- The `inner_decorator` function takes a function as an argument and returns the `inner_calc` function to do the actual calculation.
- The `inner_calc` function calls the decorated function to get the `exponent` attribute (in this case) and then uses the `base` attribute, which is passed to the outer decorator function as an argument. As expected, the closure around the inner function makes the value of the `base` attribute available to the `inner_calc` function.

Next, we will discuss how to use more than one decorator with a function or a method.

## Using multiple decorators

We have learned numerous times that there is a possibility of using more than one decorator with a function. This is possible by chaining the decorators. Chained decorators can either be the same or different. This can be achieved by placing the decorators one after the other before the function definition. When more than one decorator is used with a function, the decorated function is only executed once. To illustrate its implementation and practical use, we have selected an example in which we log a message to a target system using a timestamp. The timestamp is added through a separate decorator, and the target system is also selected based on another decorator. The following code sample shows the definitions of three decorators, that is, **add\_time\_stamp**, **file**, and **console**:

```
decorator5.py (part 1)

from datetime import datetime

from functools import wraps

def add_timestamp(func):

 @wraps(func)

 def inner_func(*args, **kwargs):

 res = "{}:{}\n".format(datetime.now(), func(*args, \
 **kwargs))

 return res

 return inner_func

def file(func):

 @wraps(func)

 def inner_func(*args, **kwargs):

 res = func(*args, **kwargs)

 with open("log.txt", 'a') as file:

 file.write(res)

 return res

 return inner_func

def console(func):

 @wraps(func)

 def inner_func(*args, **kwargs):

 res = func(*args, **kwargs)

 print(res)

 return res
```

```
 return inner_func
```

In the preceding code example, we implemented three decorator functions. They are as follows:

- **file**: This decorator uses a predefined text file and adds the message provided by the decorated function to the file.
- **console**: This decorator outputs the message provided by the decorated function to the console.
- **add\_timestamp**: This decorator adds a timestamp prior to the message provided by the decorated function. The execution of this decorator function has to occur before the file or console decorators, which means this decorator has to be placed last in the chain of decorators.

In the following code snippet, we can use these decorators for different functions inside our main program:

```
#decorator5.py (part 2)

@file

@add_timestamp

def log(msg):
 return msg

@file

@console

@add_timestamp

def log1(msg):
 return msg

@console

@add_timestamp

def log2(msg):
 return msg

log("This is a test message for file only")
log1("This is a test message for both file and console")
log2("This message is for console only")
```

In the preceding code sample, we used the three decorator functions defined earlier in different combinations to exhibit the different behaviors from the same logging function. In the first combination, we output the message to the file only after adding the timestamp. In the second combination, we output the message to both the file and the console. In the final combination, we output the message to the console only. This shows the flexibility that the decorators provide without needing to change the functions. It is worth mentioning that the decorators are very useful in simplifying the code and adding behavior in a concise way, but they have the cost of additional

overheads during execution. The use of decorators should be limited to those scenarios where the benefit is enough to compensate for any overhead costs.

This concludes our discussion regarding advanced function concepts and tricks. In the next section, we will switch gears to some advanced concepts related to data structures.

## Understanding advanced concepts with data structures

Python offers comprehensive support for data structures, including key tools for storing data and accessing data for processing and retrieving. In *Chapter 4, Python Libraries for Advanced Programming*, we discussed the data structure objects that are available in Python. In this section, we will cover a number of advanced concepts such as a dictionary within a dictionary and how to use comprehension with a data structure. We will start by embedding a dictionary inside a dictionary.

### Embedding a dictionary inside a dictionary

A dictionary in a dictionary or a nested dictionary is the process of putting a dictionary inside another dictionary. A nested dictionary is useful in many real-world examples, particularly when you are processing and transforming data from one format into the other.

*Figure 6.1* shows a nested dictionary. The root dictionary has two dictionaries against key **1** and key **3**. The dictionary against key **1** has further dictionaries inside it. The dictionary against key **3** is a regular dictionary with key-value pairs as its entries:

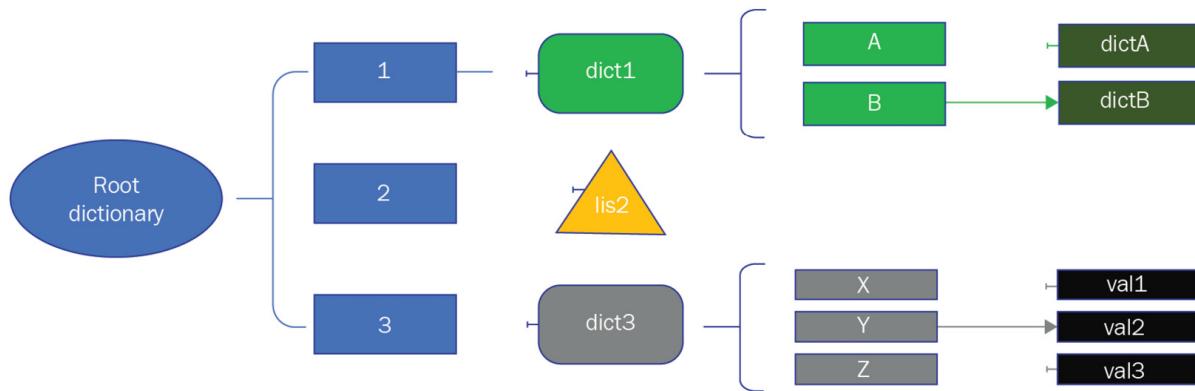


Figure 6.1: An example of a dictionary inside a dictionary

The root dictionary shown in *Figure 6.1* can be written as follows:

```
root_dict = {'1': {'A': {dictA}, 'B':{dictB}},
 '2': [list2],
```

```
'3': {'X': val1, 'Y':val2,'Z': val3}
}
```

Here, we created a root dictionary with a mix of dictionary objects and list objects inside it.

## Creating or defining a nested dictionary

A nested dictionary can be defined or created by placing comma-separated dictionaries within curly brackets. To demonstrate how to create a nested dictionary, we will create a dictionary for students.

Each student entry will have another dictionary with **name** and **age** as its elements, which are mapped to their student number:

```
dictionary1.py

dict1 = {100:{'name':'John', 'age':24},
 101:{'name':'Mike', 'age':22},
 102:{'name':'Jim', 'age':21} }

print(dict1)

print(dict1.get(100))
```

Next, we will learn how to create a dictionary dynamically and how to add or update nested dictionary elements.

## Adding to a nested dictionary

To create a dictionary in a dictionary dynamically or to add elements to an existing nested dictionary, we can use multiple approaches. In the following code example, we will used three different approaches to build a nested dictionary. They are the same as the ones we defined in the **dictionary1.py** module:

- In the first case, we will build an inner dictionary (that is, **student101**) through the direct assignment of key-value pair items and then by assigning it to a key in the root dictionary. This is the preferred approach whenever possible because the code is both easier to read and manage.
- In the second case, we created an empty inner dictionary (that is, **student102**) and assigned the values to the keys through assignment statements. This is also a preferred approach when the values are available to us through other data structures.
- In the third case, we directly initiate an empty directory for the third key of the root dictionary. After the initialization process, we assign the values using double indexing (that is, two keys): the first key is for the root dictionary, and the second key is for the inner dictionary. This approach makes the code concise, but it is not a preferred approach if code readability is important for maintenance reasons.

The complete code example for these three different cases is as follows:

```
dictionary2.py

#defining inner dictionary 1

student100 = {'name': 'John', 'age': 24}

#defining inner dictionary 2
```

```

student101 = {}

student101['name'] = 'Mike'
student101['age'] = '22'

#assigning inner dictionaries 1 and 2 to a root dictionary

dict1 = {}

dict1[100] = student100
dict1[101] = student101

#creating inner dictionary directly inside a root \
dictionary

dict1[102] = {}

dict1[102]['name'] = 'Jim'
dict1[102]['age'] = '21'

print(dict1)

print(dict1.get(102))

```

Next, we will discuss how to access different elements from a nested dictionary.

### **Accessing elements from a nested dictionary**

As we discussed earlier, to add values and dictionaries inside a dictionary, we can use double indexing. Alternatively, we can use the **get** method of the dictionary object. The same approach is applicable to access different elements from an inner dictionary. The following is an example code that illustrates how to access different elements from the inner dictionaries using the **get** method and double indexes:

```

dictionary3.py

dict1 = {100:{'name':'John', 'age':24},
 101:{'name':'Mike', 'age':22},
 102:{'name':'Jim', 'age':21} }

print(dict1.get(100))

print(dict1.get(100).get('name'))

print(dict1[101])

print(dict1[101]['age'])

```

Next, we will examine how to delete an inner dictionary or a key-value pair item from an inner dictionary.

### **Deleting from a nested dictionary**

To delete a dictionary or an element from a dictionary, we can use the generic **del** function, or we can use the **pop** method of the **dictionary** object. In the following example code, we will present both the **del** function and the **pop** method to demonstrate their usage:

```
dictionary4.py

dict1 = {100:{'name':'John', 'age':24},
 101:{'name':'Mike', 'age':22},
 102:{'name':'Jim', 'age':21} }

del (dict1[101]['age'])

print(dict1)

dict1[102].pop('age')

print(dict1)
```

In the next section, we will discuss how comprehension helps to process data from different data structure types.

## Using comprehension

**Comprehension** is a quick way in which to build new sequences such as lists, sets, and dictionaries from existing sequences. Python supports four different types of comprehension, as follows:

- List comprehension
- Dictionary comprehension
- Set comprehension
- Generator comprehension

We will discuss a brief overview, with code examples, for each of these comprehension types in the following subsections.

### List comprehension

**List comprehension** involves creating a dynamic list using a loop and a conditional statement if needed.

A few examples of how to use list comprehension will help us to understand the concept better. In the first example (that is, **list1.py**), we will create a new list from an original list by adding 1 to each element of the original list. Here is the code snippet:

```
#list1.py

list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

list2 = [x+1 for x in list1]
```

```
print(list2)
```

In this case, the new list will be created using the **x+1** expression, where **x** is an element in the original list. This is equivalent to the following traditional code:

```
list2 = []
for x in list1:
 list2.append(x+1)
```

Using list comprehension, we can achieve these three lines of code with only one line of code.

In the second example (that is, **list2.py**), we will create a new list from the original list of numbers from 1 to 10 but only include even numbers. We can do this by simply adding a condition to the previous code example, as follows:

```
#list2.py
list1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
list2 = [x for x in list1 if x % 2 == 0]
print(list2)
```

As you can see, the condition is added to the end of the comprehension expression. Next, we will discuss how to build dictionaries using comprehension.

## Dictionary comprehension

Dictionaries can also be created by using **dictionary comprehension**. Dictionary comprehension, which is similar to list comprehension, is an approach of creating a dictionary from another dictionary in such a way that the items from the source dictionary are selected or transformed conditionally. The following code snippet shows an example of creating a dictionary from existing dictionary elements that are less than or equal to 200 and by dividing each selected value by 2. Note that the values are also converted back into integers as part of the comprehension expression:

```
#dictcomp1.py
dict1 = {'a': 100, 'b': 200, 'c': 300}
dict2 = {x:int(y/2) for (x, y) in dict1.items() if y <=200}
print(dict2)
```

This dictionary comprehension code is equivalent to the following code snippet if done using traditional programming:

```
Dict2 = {}
for x,y in dict1.items():
 if y <= 200:
 dict2[x] = int(y/2)
```

Note that comprehension reduces the code significantly. Next, we will discuss set comprehension.

## Set comprehension

Sets can also be created using **set comprehension**, just like list comprehension. The code syntax for creating sets using set comprehension is similar to list comprehension. The exception is that we will be using curly brackets instead of square brackets. In the following code snippet, you can view an example of creating a set from a list using set comprehension:

```
#setcomp1.py

list1 = [1, 2, 6, 4, 5, 6, 7, 8, 9, 10, 8]

set1 = {x for x in list1 if x % 2 == 0}

print(set1)
```

This set comprehension code is equivalent to the following code snippet with traditional programming:

```
Set1 = set()

for x in list1:
 if x % 2 == 0:
 set1.add(x)
```

As expected, the duplicate entries will be discarded in the set.

This concludes our discussion regarding the types of comprehension that are available in Python for different data structures. Next, we will discuss the filtering options that are available with data structures.

## Introducing advanced tricks with pandas DataFrame

**pandas** is an open source Python library that provides tools for high-performance data manipulation to make data analysis quick and easy. The typical uses of the pandas library are to reshape, sort, slice, aggregate, and merge data.

The pandas library is built on top of the **NumPy** library, which is another Python library that is used for working with arrays. The NumPy library is significantly faster than traditional Python lists because data is stored at one continuous location in memory, which is not the case with traditional lists.

The pandas library deals with three key data structures, as follows:

- **Series:** This is a single-dimensional array-like object that contains an array of data and an array of data labels. The array of data labels is called an **index**. The **index** can be specified automatically using integers from 0 to n-1 if not explicitly specified by a user.

- **DataFrame:** This is a representation of tabular data such as a spreadsheet containing a list of columns. The DataFrame object helps to store and manipulate tabular data in rows and columns. Interestingly, the DataFrame object has an index for both columns and rows.
- **Panel:** This is a three-dimensional container of data.

The DataFrame is the key data structure that is used in data analysis. In the remainder of this section, we will be using the DataFrame object extensively in our code examples. Before we discuss any advanced tricks regarding these pandas DataFrame objects, we will do a quick review of the fundamental operations available for DataFrame objects.

## Learning DataFrame operations

We will start by creating DataFrame objects. There are several ways to create a DataFrame, such as from a dictionary, a CSV file, an Excel sheet, or from a NumPy array. One of the easiest ways is to use the data in a dictionary as input. The following code snippet shows how you can build a DataFrame object based on weekly weather data stored in a dictionary:

```
pandas1.py
import pandas as pd
weekly_data = {'day': ['Monday', 'Tuesday', 'Wednesday', \
'Thursday', 'Friday', 'Saturday', 'Sunday'],
 'temperature':[40, 33, 42, 31, 41, 40, 30],
 'condition':['Sunny', 'Cloudy', 'Sunny', 'Rain' \
, 'Sunny', 'Cloudy', 'Rain']}
df = pd.DataFrame(weekly_data)
print(df)
```

The console output will show the contents of the DataFrame as follows:

	day	temp	condition
0	Monday	40	Sunny
1	Tuesday	33	Cloudy
2	Wednesday	42	Sunny
3	Thursday	31	Rain
4	Friday	41	Sunny
5	Saturday	40	Cloudy
6	Sunday	30	Rain

Figure 6.2 – The contents of the DataFrame

The pandas library is very rich in terms of methods and attributes. However, it is beyond the scope of this section to cover all of them. Instead, we will present a quick summary of the commonly used attributes and methods of DataFrame objects next to refresh our knowledge before using them in the upcoming code examples:

- **index:** This attribute provides a list of indexes (or labels) of the DataFrame object.
- **columns:** This attribute provides a list of columns in the DataFrame object.
- **size:** This returns the size of the DataFrame object in terms of the number of rows multiplied by the number of columns.
- **shape:** This provides us with a tuple representing the dimension of the DataFrame object.
- **axes:** This attribute returns a list that represents the axes of the DataFrame object. Put simply, it includes rows and columns.
- **describe:** This powerful method generates statistics data such as the count, mean, standard deviation, and minimum and maximum values.
- **head:** This method returns  $n$  (default = 5) rows from a DataFrame object similar to the head command on files.
- **tail:** This method returns the last  $n$  (default = 5) rows from a DataFrame object.
- **drop\_duplicates:** This method drops duplicate rows based on all of the columns in a DataFrame.
- **dropna:** This method removes missing values (such as rows or columns) from a DataFrame. By passing appropriate arguments to this method, we can either remove rows or columns. Additionally, we can set whether the rows or columns will be removed based on a single occurrence of a missing value or only when all of the values in a row or column are missing.
- **sort\_values:** This method can be used to sort the rows based on single or multiple columns.

In the following sections, we will review some fundamental operations for DataFrame objects.

## Setting a custom index

The column labels (index) are normally added as per the data provided with a dictionary or according to whatever other input data stream has been used. We can change the index of the DataFrame by

using one of the following options:

- Set one of the data columns as an index, such as **day** in the previously mentioned example, by using a simple statement like this:

```
df_new = df.set_index('day')
```

The DataFrame will start using the **day** column as an index column, and its contents will be as follows:

	temp	condition
day		
Monday	40	Sunny
Tuesday	33	Cloudy
Wednesday	42	Sunny
Thursday	31	Rain
Friday	41	Sunny
Saturday	40	Cloudy
Sunday	30	Rain

Figure 6.3 – The contents of the DataFrame after using the **day** column as an index

- Set the index manually by providing it through a list, such as in the following code snippet:

```
pandas2.py

weekly_data = <same as previous example>

df = pd.DataFrame(weekly_data)

df.index = ['MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT', 'SUN']

print(df)
```

With this code snippet, the DataFrame will start using the index as provided by us through a list object. The contents of the DataFrame will show this change as follows:

		day	temp	condition
MON	Monday	40	Sunny	
TUE	Tuesday	33	Cloudy	
WED	Wednesday	42	Sunny	
THU	Thursday	31	Rain	
FRI	Friday	41	Sunny	
SAT	Saturday	40	Cloudy	
SUN	Sunday	30	Rain	

Figure 6.4 – The contents of the DataFrame after setting custom entries for an index column

Next, we will discuss how to navigate inside a DataFrame using a certain index and column.

### Navigating inside a DataFrame

There are a few dozen ways in which to get a row of data or a particular location from a DataFrame object. The typical methods that are used to navigate inside a DataFrame are the **loc** and **iloc** methods. We will explore a few options of how to navigate through a DataFrame object using the same sample data that we used in the previous example:

```
pandas3.py
import pandas as pd
weekly_data = <same as in pandas1.py example>
df = pd.DataFrame(weekly_data)
df.index = ['MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT', 'SUN']
```

Next, we will discuss a few techniques, with code samples, regarding how to select a row or location in this DataFrame object:

- We can select one or more rows by using index labels with the **loc** method. The index label is provided as a single item or a list. In the following code snippet, we have illustrated two examples of how to select one or more rows:

```
print(df.loc['TUE'])
print(df.loc[['TUE', 'WED']])
```

- We can select a value from a location in a DataFrame object using the row index label and the column label, as follows:

```
print(df.loc['FRI', 'temp'])
```

- We can also select a row by using an index value without providing any labels:

```
#Provide a row with index 2
```

```
print(df.iloc[2])
```

- We can select a value from a location using the row index value and the column index value by treating the DataFrame object like a two-dimensional array. In the next code snippet, we will get a value from a location in which the row index = 2 and the column index = 2:

```
print(df.iloc[2,2])
```

Next, we will discuss how to add a row or a column to a DataFrame object.

## Adding a row or column to a DataFrame

The easiest way to add a row to a DataFrame object is by assigning a list of values to an index location or an index label. For example, we can add a new row with the **TST** label for the previous example (that is, **pandas3.py**) by using the following statement:

```
df.loc['TST'] = ['Test day 1', 50, 'NA']
```

It is important to note that if the row label already exists in the DataFrame object, the same line of code can update the row with new values.

If we are not using the index label but the default index instead, we can use the index number to update an existing row or add a new row by using the following line of code:

```
df.loc[8] = ['Test day 2', 40, 'NA']
```

A complete code example is shown for reference:

```
pandas4.py
import pandas as pd
weekly_data = <same as in pandas1.py example>
df = pd.DataFrame(weekly_data)
df.index = ['MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT', 'SUN']
df.loc['TST1'] = ['Test day 1', 50, 'NA']
df.loc[7] = ['Test day 2', 40, 'NA']
print(df)
```

To add a new column to a DataFrame object, multiple options are available in the pandas library. We will only illustrate three options, as follows:

- **By adding a list of values next to the column label:** This approach will add a column after the existing columns. If we use an existing column label, this approach can also be used to update or replace an existing column.
- **By using the insert method:** This method will take a label and a list of values as arguments. This is particularly useful when you want to insert a column at any location. Note that this method does not allow you to insert a column if there is already an existing column inside the DataFrame object with the same label. This means this method cannot be used to update an existing column.
- **By using the assign method:** This method is useful when you want to add multiple columns in one go. If we use an existing column label, this method can be used to update or replace an existing column.

In the following code example, we will use all three approaches to insert a new column to a DataFrame object:

```
pandas5.py

import pandas as pd

weekly_data = <same as in pandas1.py example>

df = pd.DataFrame(weekly_data)

#Adding a new column and then updating it

df['Humidity1'] = [60, 70, 65, 62, 56, 25, '']

df['Humidity1'] = [60, 70, 65, 62, 56, 251, '']

#Inserting a column at column index of 2 using the insert method

df.insert(2, "Humidity2", [60, 70, 65, 62, 56, 25, ''])

#Adding two columns using the assign method

df1 = df.assign(Humidity3 = [60, 70, 65, 62, 56, 25, ''], Humidity4 = [60, 70,
65, 62, 56, 25, ''])

print(df1)
```

Next, we will evaluate how to delete rows and columns from a DataFrame object.

### **Deleting an index, a row, or a column from a DataFrame**

Removing an index is relatively straightforward, and you can do so by using the **reset\_index** method. However, the **reset\_index** method adds default indexes and keeps the custom index column as a data column. To remove the custom index column completely, we have to use the **drop** argument with the **reset\_index** method. The following code snippet uses the **reset\_index** method:

```
pandas6.py

import pandas as pd

weekly_data = <same as in pandas1.py example>

df = pd.DataFrame(weekly_data)

df.index = ['MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT', 'SAT']

print(df)

print(df.reset_index(drop=True))
```

To delete a duplicate row from a DataFrame object, we can use the **drop\_duplicate** method. To delete a particular row or column, we can use the **drop** method. In the following code example, we will remove any rows with the **SAT** and **SUN** labels and any columns with the **condition** label:

```
#pandas7.py

import pandas as pd
```

```

weekly_data = <same as in pandas1.py example>

df = pd.DataFrame(weekly_data)

df.index = ['MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT', 'SUN']

print(df)

df1= df.drop(index=['SUN', 'SAT'])

df2= df1.drop(columns=['condition'])

print(df2)

```

Next, we will examine how to rename an index or a column.

### **Renaming indexes and columns in a DataFrame**

To rename an index or a column label, we will use the **rename** method. A code example of how to rename an index and a column is as follows:

```

#pandas8.py

import pandas as pd

weekly_data = <same as in pandas1.py example>

df = pd.DataFrame(weekly_data)

df.index = ['MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT', 'SUN']

df1=df.rename(index={'SUN': 'SU', 'SAT': 'SA'})

df2=df1.rename(columns={'condition': 'cond'})

print(df2)

```

It is important to note that the current label and the new label for the index and column are provided as a dictionary. Next, we will discuss some advanced tricks for using DataFrame objects.

## **Learning advanced tricks for a DataFrame object**

In the previous section, we evaluated the fundamental operations that can be performed on a DataFrame object. In this section, we will investigate the next level of operations on a DataFrame object for data evaluation and transformation. These operations are discussed in the following subsections.

### **Replacing data**

One common requirement is to replace numeric data or string data with another set of values. The pandas library is full of options in which to carry out such data replacements. The most popular method for these operations is to use the **at** method. The **at** method provides an easy way to access or update data in any cell in a DataFrame. For bulk replacement operations, you can also use a **replace** method, and we can use this method in many ways. For example, we can use this method to replace a

number with another number or a string with another string, or we can replace anything that matches a regular expression. Additionally, we can use this method to replace any entries provided through a list or a dictionary. In the following code example (that is, **pandastrick1.py**), we will cover most of these replacement options. For this code example, we will use the same DataFrame object that we used in previous code examples. Here is the sample code:

```
pandastrick1.py
import pandas as pd

weekly_data = <same as in pandas1.py example>
df = pd.DataFrame(weekly_data)
```

Next, we will explore several replacement operations on this DataFrame object, one by one:

- Replace any occurrences of the numeric value of **40** with **39** across the DataFrame object using the following statement:  
`df.replace(40, 39, inplace=True)`
- Replace any occurrences of a **Sunny** string with **Sun** across the DataFrame object using the following statement:  
`df.replace("Sunny", "Sun", inplace=True)`
- Replace any occurrences of a string based on a regular expression (the aim is to replace **Cloudy** with **Cloud**) using the following statement:  
`df.replace(to_replace="^Cl.*", value="Cloud", inplace=True, regex=True)`  
#or we can apply on a specific column as well.  
`df["condition"].replace(to_replace="^Cl.*", value="Cloud", inplace=True, regex=True)`

Note that the use of the **to\_replace** and **value** argument labels is optional.

- Replace any occurrences of multiple strings represented by a list with another list of strings using the following statement:  
`df.replace(["Monday", "Tuesday"], ["Mon", "Tue"], inplace=True)`

In this code, we replaced **Monday** and **Tuesday** with **Mon** and **Tue**.

- Replace any occurrences of multiple strings in a DataFrame object using the key-value pairs in a dictionary. You can do this by using the following statement:  
`df.replace({"Wednesday": "Wed", "Thursday": "Thu"}, inplace=True)`

In this case, the keys of the dictionary (that is, **Wednesday** and **Thursday**) will be replaced by their corresponding values (that is, **Wed** and **Thu**).

- Replace any occurrences of a string for a certain column using multiple dictionaries. You can do this by using the column name as a key in the dictionary and a sample statement such as the following:  
`df.replace({"day": "Friday"}, {"day": "Fri"}, inplace=True)`

In this scenario, the first dictionary is used to indicate the column name and the value to be replaced. The second dictionary is used to indicate the same column name but with a value that

will replace the original value. In our case, we will replace all instances of **Friday** in the **day** column with the value of **Fri**.

- Replace any occurrences of multiple strings using a nested dictionary. You can do this by using a code sample such as the following:

```
df.replace({"day":{"Saturday":"Sat", "Sunday":"Sun"},
 "condition":{"Rainy":"Rain"}}, inplace=True)
```

In this scenario, the outer dictionary (with the **day** and **condition** keys in our code sample) is used to identify the columns for this operation and the inner dictionary is used to hold the data to be replaced along with the replacing value. By using this approach, we replaced **Saturday** and **Sunday** with **Sat** and **Sun** inside the **day** column and the **Rainy** string with **Rain** inside the **condition** column.

The complete code with all these sample operations is available within the source code of this chapter as **pandastrick1.py**. Note that we can either trigger the replacement operation across the DataFrame object or we can limit it to a certain column or a row.

### **IMPORTANT NOTE**

*The `inplace=True` argument is used with all `replace` method calls. This argument is used to set the output of the `replace` method within the same DataFrame object. The default option is to return a new DataFrame object without changing the original object. This argument is available with many DataFrame methods for convenience.*

### **Applying a function to the column or row of a DataFrame object**

Sometimes, we want to clean up the data, adjust the data, or transform the data before starting data analysis. There is an easy way in which to apply some type of function on a DataFrame using the **apply**, **applymap**, or **map** methods. The **apply** method is applicable to columns or rows, while the **applymap** method works element by element for the whole DataFrame. In comparison, the **map** method works element by element for a single series. Now, we will discuss a couple of code examples to illustrate the use of the **apply** and **map** methods.

It is common to have data imported into a DataFrame object that might need some cleaning up. For example, it could have trailing or leading whitespaces, new line characters, or any unwanted characters. These can be removed from the data easily by using the **map** method and the lambda function on a column series. The lambda function is used on each element of the column. In our code example, first, we will remove the trailing whitespace, dot, and comma. Then, we will remove the leading whitespace, underscore, and dash for the **condition** column.

After cleaning up the data inside the **condition** column, the next step is to create a new **temp\_F** column from the values of the **temp** column and convert them from Celsius units into Fahrenheit units. Note that we will use another lambda function for this conversion and use the **apply** method.

When we get the result from the **apply** method, we will store it inside a new column label, **temp\_F**, to create a new column. Here is the complete code example:

```
pandastrick2.py

import pandas as pd

weekly_data = {'day': ['Monday', 'Tuesday',
 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'],
 'temp':[40, 33, 42, 31, 41, 40, 30],
 'condition': ['Sunny', '_Cloudy ', 'Sunny', 'Rainy', '--Sunny.',
 'Cloudy.', 'Rainy']}

df = pd.DataFrame(weekly_data)

print(df)

df["condition"] = df["condition"].map(
 lambda x: x.lstrip('_').rstrip('.. '))

df["temp_F"] = df["temp"].apply(lambda x: 9/5*x+32)

print(df)
```

Note that for the preceding code example, we provided the same input data as in previous examples except that we added trailing and leading characters to the **condition** column data.

## Querying rows in a DataFrame object

To query rows based on the values in a certain column, one common approach is to apply a filter using *AND* or *OR* logical operations. However, this quickly becomes a messy approach for simple requirements such as searching a row with a value in between a range of values. The pandas library offers a cleaner tool: the **between** method, which is somewhat similar to the *between* keyword in SQL.

The following code example uses the same **weekly\_data** DataFrame object that we used in the previous example. First, we will show the use of a traditional filter, and then we will show the use of the **between** method to query the rows that have temperature values between 30 and 40 inclusively:

```
pandastrick3.py

import pandas as pd

weekly_data = <same as in pandas1.py example>

df = pd.DataFrame(weekly_data)

print(df[(df.temp >= 30) & (df.temp<=40)])

print(df[df.temp.between(30,40)])
```

We get the same console output for both approaches we used. However, using the **between** method is far more convenient than writing conditional filters.

Querying rows based on text data is also very well supported in the pandas library. This can be achieved by using the **str** accessor on the string-type columns of the DataFrame object. For example, if we want to search rows in our **weekly\_data** DataFrame object based on the condition of a day, such as **Rainy** or **Sunny**, we can either write a traditional filter or we can use the **str** accessor on the column with the **contains** method. The following code example illustrates the use of both options to get the rows with **Rainy** or **Sunny** as data values in the **condition** column:

```
pandastrick4.py
import pandas as pd

weekly_data = <same as in pandas1.py example>

df = pd.DataFrame(weekly_data)

print(df[(df.condition=='Rainy') | (df.condition=='Sunny')])
print(df[df['condition'].str.contains('Rainy|Sunny')])
```

If you run the preceding code, you will find that the console output is the same for both of the approaches we used for searching the data.

## Getting statistics on the DataFrame object data

To get statistical data such as central tendency, standard deviation, and shape, we can use the **describe** method. The output of the **describe** method for numeric columns includes the following:

- **count**
- **mean**
- **standard deviation**
- **min**
- **max**
- **25<sup>th</sup> percentiles, 50<sup>th</sup> percentile, 75<sup>th</sup> percentile**

The default breakdown of percentiles can be changed by using the **percentiles** argument with the desired breakdown.

If the **describe** method is used for non-numeric data, such as strings, we will get *count*, *unique*, *top*, and *freq*. The *top* value is the most common value, whereas *freq* is the most common value frequency. By default, only numeric columns are evaluated by the **describe** method unless we provide the **include** argument with an appropriate value.

In the following code example, we will evaluate the following for the same **weekly\_date** DataFrame object:

- The use of the **describe** method with or without the **include** argument
- The use of the **percentiles** argument with the **describe** method
- The use of the **groupby** method to group data on a column basis and then using the **describe** method on top of it

The complete code example is as follows:

```
pandastrick5.py

import pandas as pd

import numpy as np

pd.set_option('display.max_columns', None)

weekly_data = <same as in pandas1.py example>

df = pd.DataFrame(weekly_data)

print(df.describe())

print(df.describe(include="all"))

print(df.describe(percentiles=np.arange(0, 1, 0.1)))

print(df.groupby('condition').describe(percentiles=np.arange(0, 1, 0.1)))
```

Note that we changed the **max\_columns** options for the pandas library at the beginning in order to display all of the columns that we expected in the console output. Without this, some of the columns will be truncated for the console output of the **groupby** method.

This concludes our discussion of the advanced tricks of working with a DataFrame object. This set of tricks and tips will empower anyone to start using the pandas library for data analysis. For additional advanced concepts, we recommend that you refer to the official documentation of the pandas library.

## Summary

In this chapter, we introduced some advanced tricks that are important when you want to write efficient and concise programs in Python. We started with advanced functions such as the mapper, reducer, and filter functions. We also discussed several advanced concepts of functions, such as inner functions, lambda functions, and decorators. This was followed by a discussion of how to use data structures, including nested dictionaries and comprehensions. Finally, we reviewed the fundamental operations of a DataFrame object, and then we evaluated a few use cases using some advanced operations of the DataFrame object.

This chapter mainly focused on hands-on knowledge and experience of how to use advanced concepts in Python. This is important for anyone who wants to develop Python applications, especially for data analysis. The code examples provided in this chapter are very helpful for you to

begin learning the advanced tricks that are available for functions, data structures, and the pandas library.

In the next chapter, we will explore multiprocessing and multithreading in Python.

## Questions

1. Which of the **map**, **filter**, and **reduce** functions are built-in Python functions?
2. What are standard decorators?
3. Would you prefer a generator comprehension or a list comprehension for a large dataset?
4. What is a DataFrame in the context of the pandas library?
5. What is the purpose of the **inplace** argument in pandas' library methods?

## Further reading

- *Mastering Python Design Patterns*, by Sakis Kasampalis
- *Python for Data Analysis*, by Wes McKinney
- *Hands-On Data Analysis with Pandas, Second Edition*, by Stefanie Molin
- *The official Pandas documentation*, which is available at <https://pandas.pydata.org/docs/>

## Answers

1. The **map** and the **filter** functions are built-in.
2. Standard decorators are the ones without any arguments.
3. The generator comprehension is preferred in this case. It is memory efficient as the values are generated one by one.
4. The DataFrame is a representation of tabular data, such as a spreadsheet, and is a commonly used object for data analysis using the pandas library.
5. When the **inplace** argument in pandas' library methods is set to **True**, the result of the operation is saved to the same DataFrame object on which the operation is applied.

## Section 3: Scaling beyond a Single Thread

In this part of the book, our journey will take a turn toward programming for scalable applications. A typical Python interpreter runs on a single thread running on a single process. For this part of our journey, we discuss how to scale out Python beyond this single thread running on a single process. To do that, we first look into multithreading, multiprocessing, and asynchronous programming on a single machine. Then, we explore how we can go beyond the single machine and run our applications on clusters using Apache Spark. After that, we investigate using cloud computing environments to focus on the application and leave the infrastructure management to cloud providers.

This section contains the following chapters:

- [\*Chapter 7, Multiprocessing, Multithreading, and Asynchronous Programming\*](#)
- [\*Chapter 8, Scaling Out Python using Clusters\*](#)
- [\*Chapter 9, Python Programming for the Cloud\*](#)

[OceanofPDF.com](#)

## *Chapter 7: Multiprocessing, Multithreading, and Asynchronous Programming*

We can write efficient and optimized code for faster execution time, but there is always a limit to the amount of resources available for the processes running our programs. However, we can still improve application execution time by executing certain tasks in parallel on the same machine or across different machines. This chapter will cover parallel processing or concurrency in Python for the applications running on a single machine. We will cover parallel processing using multiple machines in the next chapter. In this chapter, we focus on the built-in support available in Python for the implementation of parallel processing. We will start with the multithreading in Python followed by discussing the multiprocessing. After that, we will discuss how we can design responsive systems using asynchronous programming. For each of the approaches, we will design and discuss a case study of implementing a concurrent application to download files from a Google Drive directory.

We will cover the following topics in this chapter:

- Understanding multithreading in Python and its limitations
- Going beyond a single CPU – implementing multiprocessing
- Using asynchronous programming for responsive systems

After completing this chapter, you will be aware of the different options for building multithreaded or multiprocessing applications using built-in Python libraries. These skills will help you to build not only more efficient applications but also build applications for large-scale users.

## Technical requirements

The following are the technical requirements for this chapter:

- Python 3 (3.7 or later)
- A Google Drive account
- API key enabled for your Google Drive account

Sample code for this chapter can be found at <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter07>.

We will start our discussion with multithreading concepts in Python.

## Understanding multithreading in Python and its limitations

A thread is a basic unit of execution within an operating system process, and it consists of its own program counter, a stack, and a set of registers. An application process can be built using multiple threads that can run simultaneously and share the same memory.

For multithreading in a program, all the threads of a process share common code and other resources, such as data and system files. For each thread, all its related information is stored as a data structure inside the operating system kernel, and this data structure is called the **Thread Control Block (TCB)**. The TCB has the following main components:

- **Program Counter (PC):** This is used to track the execution flow of the program.
- **System Registers (REG):** These registers are used to hold variable data.
- **Stack:** The stack is an array of registers that manages the execution history.

The anatomy of a thread is exhibited in *Figure 7.1*, with three threads. Each thread has its own PC, a stack, and REG, but shares code and data with other threads:

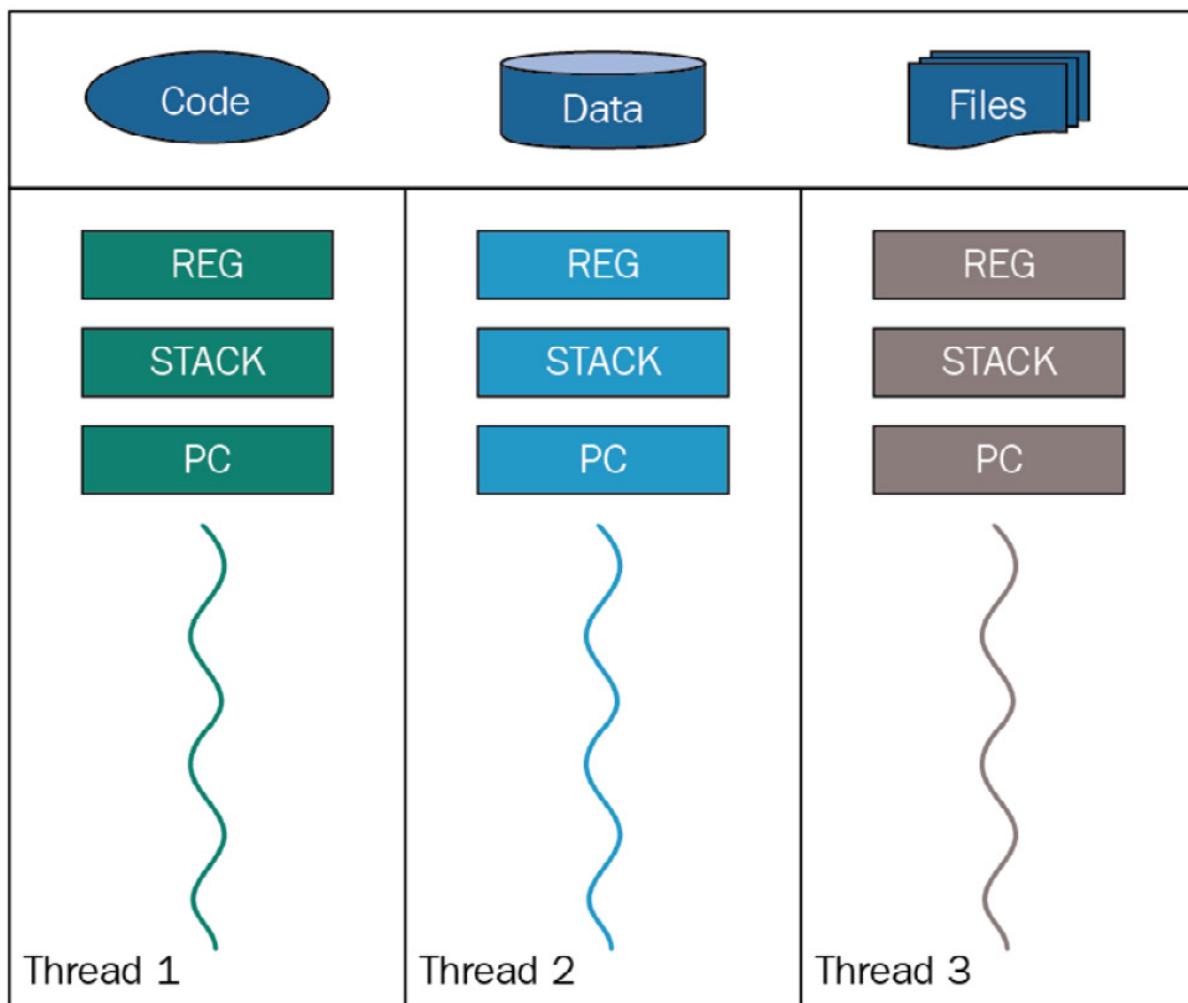


Figure 7.1 – Multiple threads in a process

The TCB also contains a thread identifier, the state of the thread (such as running, waiting, or stopped), and a pointer to the process it belongs to. Multithreading is an operating system concept. It is a feature offered through the system kernel. The operating system facilitates the execution of multiple threads concurrently in the same process context, allowing them to share the process memory. This means the operating system has full control of which thread will be activated, rather than the application. We need to underline this point for a later discussion comparing different concurrency options.

When threads are run on a single-CPU machine, the operating system actually switches the CPU from one thread to the other such that the threads appear to be running concurrently. Is there any advantage to running multiple threads on a single-CPU machine? The answer is yes and no, and it depends on the nature of the application. For applications running using only the local memory, there may not be any advantage; in fact, it is likely to exhibit lower performance due to the overhead of switching threads on a single CPU. But for applications that depend on other resources, the execution can be faster because of the better utilization of the CPU: when one thread is waiting for another resource, another thread can utilize the CPU.

When executing multiple threads on multiprocessors or multiple CPU cores, it is possible to execute them concurrently. Next, we will discuss the limitations of multithreaded programming in Python.

## What is a Python blind spot?

From a programming perspective, multithreading is an approach to running different parts of an application concurrently. Python uses multiple kernel threads that can run the Python user threads. But the Python implementation (*CPython*) allows threads to access the Python objects through one global lock, which is called the **Global Interpreter Lock (GIL)**. In simple words, the GIL is a mutex that allows only one thread to use the Python interpreter at a time and blocks all other threads. This is necessary to protect the reference count that is managed for each object in Python from garbage collection. Without such protection, the reference count can get corrupted if it's updated by multiple threads at the same time. The reason for this limitation is to protect the internal interpreter data structures and third-party C code that is not thread safe.

### *IMPORTANT NOTE*

*This GIL limitation does not exist in Jython and IronPython, which are other implementations of Python.*

This Python limitation may give us the impression that there is no advantage to writing multithreaded programs in Python. This is not true. We still can write code in Python that runs concurrently or in

parallel, and we will see it in our case study. Multithreading can be beneficial in the following cases:

- **I/O bound tasks:** When working with multiple I/O operations, there is always room to improve performance by running tasks using more than one thread. When one thread is waiting for a response from an I/O resource, it will release the GIL and let the other threads work. The original thread will wake up as soon as the response arrives from the I/O resource.
- **Responsive GUI application:** For interactive GUI applications, it is necessary to have a design pattern to display the progress of tasks running in the background (for example, downloading a file) and also to allow a user to work on other GUI features while one or more tasks are running in the background. This is all possible by using separate threads for the actions initiated by a user through the GUI.
- **Multiuser applications:** Threads are also a prerequisite for building multiuser applications. A web server and a file server are examples of such applications. As soon as a new request arrives in the main thread of such an application, a new thread is created to serve the request while the main thread at the back listens for a new request.

Before discussing a case study of a multithreaded application, it is important to introduce the key components of multithreaded programming in Python.

## Learning the key components of multithreaded programming in Python

Multithreading in Python allows us to run different components of a program concurrently. To create multiple threads of an application, we will use the Python **threading** module, and the main components of this module are described next.

We will start by discussing the **threading** module in Python first.

### The **threading** module

The **threading** module comes as a standard module and provides simple and easy-to-use methods for building multiple threads of a program. Under the hood, this module uses the lower level **\_thread** module, which was a popular choice of multithreading in the early version of Python.

To create a new thread, we will create an object of the **Thread** class that can take a function (to be executed) name as the **target** attribute and arguments to be passed to the function as the **args** attribute. A thread can be given a name that can be set at the time it is created using the **name** argument with the constructor.

After creating an object of the **Thread** class, we need to start the thread by using the **start** method. To make the main program or thread wait until the newly created thread object(s) finishes, we need to use the **join** method. The **join** method makes sure that the main thread (a calling thread) waits until the thread on which the **join** method is called completes its execution.

To explain the process of creating, starting, and waiting to finish the execution of a thread, we will create a simple program with three threads. A complete code example of such a program is shown

```

next:

thread1.py to create simple threads with function

from threading import current_thread, Thread as Thread

from time import sleep

def print_hello():

 sleep(2)

 print("{}: Hello".format(current_thread().name))

def print_message(msg):

 sleep(1)

 print("{}: {}".format(current_thread().name, msg))

create threads

t1 = Thread(target=print_hello, name="Th 1")

t2 = Thread(target=print_hello, name="Th 2")

t3 = Thread(target=print_message, args=["Good morning"],

 name="Th 3")

start the threads

t1.start()

t2.start()

t3.start()

wait till all are done

t1.join()

t2.join()

t3.join()

```

In this program, we implemented the following:

- We created two simple functions, **print\_hello** and **print\_message**, that are to be used by the threads. We used the **sleep** function from the **time** module in both functions to make sure that the two functions finish their execution time at different times.
- We created three **Thread** objects. Two of the three objects will execute one function (**print\_hello**) to illustrate the code sharing by the threads, and the third thread object will use the second function (**print\_message**), which takes one argument as well.
- We started all three threads one by one using the **start** method.
- We waited for each thread to finish by using the **join** method.

The **Thread** objects can be stored in a list to simplify the **start** and **join** operations using a **for** loop.

The console output of this program will look like this:

```
Th 3: Good morning
```

Th 2: Hello

Th 1: Hello

Thread 1 and thread 2 have more sleep time than thread 3, so thread 3 will always finish first. Thread 1 and thread 2 can finish in any order depending on who gets hold of the processor first.

### ***IMPORTANT NOTE***

*By default, the **join** method blocks the caller thread indefinitely. But we can use a timeout (in seconds) as an argument to the **join** method. This will make the caller thread block only for the timeout period.*

We will review a few more concepts before discussing a more complex case study.

### **Daemon threads**

In a normal application, our main program implicitly waits until all other threads finish their execution. However, sometimes we need to run some threads in the background so that they run without blocking the main program from terminating itself. These threads are known as **daemon threads**. These threads stay active as long as the main program (with non-daemon threads) is running, and it is fine to terminate the daemon threads once the non-daemon threads exit. The use of daemon threads is popular in situations where it is not an issue if a thread dies in the middle of its execution without losing or corrupting any data.

A thread can be declared a daemon thread by using one of the following two approaches:

- Pass the **daemon** attribute set to **True** with the constructor (**daemon = True**).
- Set the **daemon** attribute to **True** on the thread instance (**thread.daemon = True**).

If a thread is set as a daemon thread, we start the thread and forget about it. The thread will be automatically killed when the program that called it quits.

The next code shows the use of both daemon and non-daemon threads:

```
#thread2.py to create daemon and non-daemon threads
from threading import current_thread, Thread as Thread
from time import sleep

def daeom_func():
 #print(threading.current_thread().isDaemon())
 sleep(3)
 print("{}: Hello from daemon".format(current_thread().name))

def nondaeom_func():
 #print(threading.current_thread().isDaemon())
 sleep(1)
 print("{}: Hello from non-daemon".format(current_thread().name))
```

```

#creating threads

t1 = Thread(target=daeom_func, name="Daemon Thread", daemon=True)
t2 = Thread(target=nondaeom_func, name="Non-Daemon Thread")

start the threads

t1.start()
t2.start()

print("Exiting the main program")

```

In this code example, we created one daemon and one non-daemon thread. The daemon thread (**daeom\_func**) is executing a function that has a sleep time of 3 seconds, whereas the non-daemon thread is executing a function (**nondaeom\_func**) that has a sleep time of 1 second. The sleep time of the two functions is set to make sure the non-daemon thread finishes its execution first. The console output of this program is as follows:

```

Exiting the main program

Non-Daemon Thread: Hello from non-daemon

```

Since we did not use a **join** method in any thread, the main thread exits first, and then the non-daemon thread finishes a bit later with a print message. But there is no print message from the daemon thread. This is because the daemon thread is terminated as soon as the non-daemon thread finishes its execution. If we change the sleep time in the **nondaeom\_func** function to 5, the console output will be as follows:

```

Exiting the main program

Daemon Thread: Hello from daemon

Non-Daemon Thread: Hello from non-daemon

```

By delaying the execution of the non-daemon thread, we make sure the daemon thread finished its execution and does not get terminated abruptly.

### ***IMPORTANT NOTE***

*If we use a **join** on the daemon thread, the main thread will be forced to wait for the daemon thread to finish its execution.*

Next, we will investigate how to synchronize the threads in Python.

### **Synchronizing threads**

**Thread synchronization** is a mechanism to ensure that the two or more threads do not execute a shared block of code at the same time. The block of code that is typically accessing shared data or shared resources is also known as the **critical section**. This concept can be made clearer through the following figure:

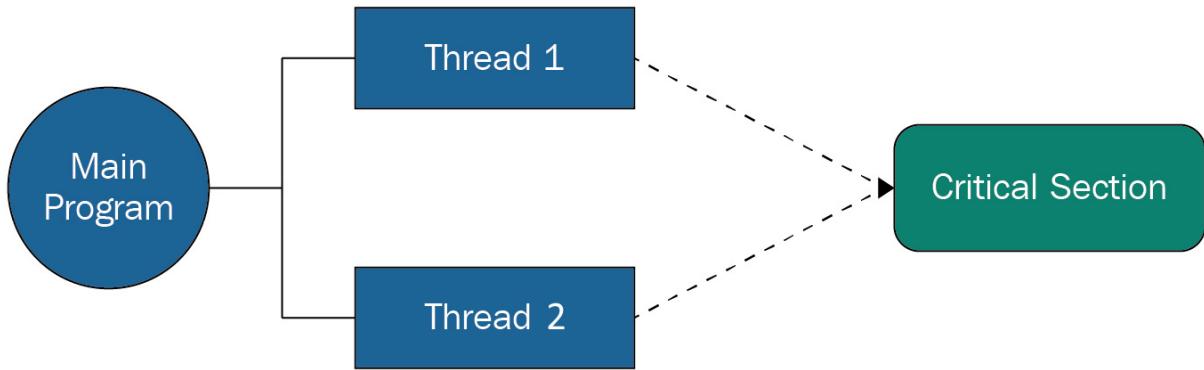


Figure 7.2 – Two threads accessing a critical section of a program

Multiple threads accessing the critical section at the same time may try to access or change the data at the same time, which may result in unpredictable results on the data. This situation is called a **race condition**.

To illustrate the concept of the race condition, we will implement a simple program with two threads, and each thread increments a shared variable 1 million times. We chose a high number for the increment to make sure that we can observe the outcome of the race condition. The race condition may also be observed by using a lower value for the increment cycle on a slower CPU. In this program, we will create two threads that are using the same function (`inc` in this case) as the target. The code for accessing the shared variable and incrementing it by 1 occurs in the critical section, and the two threads are accessing it without any protection. The complete code example is as follows:

```

thread3a.py when no thread synchronization used
from threading import Thread as Thread

def inc():
 global x
 for _ in range(1000000):
 x+=1

#global variabale
x = 0

creating threads
t1 = Thread(target=inc, name="Th 1")
t2 = Thread(target=inc, name="Th 2")

start the threads
t1.start()
t2.start()

#wait for the threads

```

```

t1.join()
t2.join()
print("final value of x :", x)

```

The expected value of **x** at the end of the execution is **2,000,000**, which will not be observed in the console output. Every time we execute this program, we will get a different value of **x** that's a lot lower than 2,000,000. This is because of the race condition between the two threads. Let's look at a scenario where threads **Th 1** and **Th 2** are running the critical section (**x+=1**) at the same time. Both threads will ask for the current value of **x**. If we assume the current value of **x** is **100**, both threads will read it as **100** and increment it to a new value of **101**. The two threads will write back to the memory the new value of **101**. This is a one-time increment and, in reality, the two threads should increment the variable independently of each other and the final value of **x** should be **102**. How can we achieve this? This is where thread synchronization comes to the rescue.

Thread synchronization can be achieved by using a **Lock** class from the **threading** module. The lock is implemented using a **semaphore** object provided by the operating system. A semaphore is a synchronization object at the operating system level to control access to the resources and data for multiple processors and threads. The **Lock** class provides two methods, **acquire** and **release**, which are described next:

- The **acquire** method is used to acquire a lock. A lock can be **blocking** (default) or **non-blocking**. In the case of a blocking lock, the requesting thread's execution is blocked until the lock is released by the current acquiring thread. Once the lock is released by the current acquiring thread (**unlocked**), then the lock is provided to the requesting thread to proceed. In the case of a non-blocking acquire request, the thread execution is not blocked. If the lock is available (**unlocked**), then the lock is provided (and **locked**) to the requesting thread to proceed, otherwise the requesting thread gets **False** as a response.
- The **release** method is used to release a lock, which means it resets the lock to an **unlocked** state. If there is any thread blocking and waiting for the lock, it will allow one of the threads to proceed.

The **thread3a.py** code example is revised with the use of a lock around the increment statement on the shared variable **x**. In this revised example, we created a lock at the main thread level and then passed it to the **inc** function to acquire and release a lock around the shared variable. The complete revised code example is as follows:

```

thread3b.py when thread synchronization is used
from threading import Lock, Thread as Thread
def inc_with_lock (lock):
 global x
 for _ in range(1000000):
 lock.acquire()
 x+=1

```

```

lock.release()

x = 0

mylock = Lock()

creating threads

t1 = Thread(target= inc_with_lock, args=(mylock,), name="Th 1")
t2 = Thread(target= inc_with_lock, args=(mylock,), name="Th 2")

start the threads

t1.start()
t2.start()

#wait for the threads

t1.join()
t2.join()

print("final value of x :", x)

```

After using the **Lock** object, the value of **x** is always **2000000**. The **Lock** object made sure that only one thread increments the shared variable at a time. The advantage of thread synchronization is that you can use system resources with enhanced performance and predictable results.

However, locks have to be used carefully because improper use of locks can result in a deadlock situation. Suppose a thread acquires a lock on resource A and is waiting to acquire a lock on resource B. But another thread already holds a lock on resource B and is looking to acquire a lock on resource A. The two threads will wait for each other to release the locks, but it will never happen. To avoid deadlock situations, the multithreading and multiprocessing libraries come with mechanisms such as adding a timeout for a resource to hold a lock, or using a context manager to acquire locks.

## Using a synchronized queue

The **Queue** module in Python implements multi-producer and multi-consumer queues. Queues are very useful in multithread applications when the information has to be exchanged between different threads safely. The beauty of the synchronized queue is that they come with all the required locking mechanisms, and there is no need to use additional locking semantics.

There are three types of queues in the **Queue** module:

- **FIFO**: In the FIFO queue, the task added first is retrieved first.
- **LIFO**: In the LIFO queue, the last task added is retrieved first.
- **Priority queue**: In this queue, the entries are sorted and the entry with the lowest value is retrieved first.

These queues use locks to protect access to the queue entries from competing threads. The use of a queue with a multithreaded program is best illustrated with a code example. In the next example, we

will create a FIFO queue with dummy tasks in it. To process the tasks from the queue, we will implement a custom thread class by inheriting the **Thread** class. This is another way of implementing a thread.

To implement a custom thread class, we need to override the **init** and **run** methods. In the **init** method, it is required to call the **init** method of the superclass (the **Thread** class). The **run** method is the execution part of the thread class. The complete code example is as follows:

```
thread5.py with queue and custom Thread class

from queue import Queue

from threading import Thread as Thread

from time import sleep

class MyWorker (Thread):

 def __init__(self, name, q):
 threading.Thread.__init__(self)
 self.name = name
 self.queue = q

 def run(self):
 while True:
 item = self.queue.get()
 sleep(1)
 try:
 print ("{}: {}".format(self.name, item))
 finally:
 self.queue.task_done()

#filling the queue

myqueue = Queue()

for i in range (10):
 myqueue.put("Task {}".format(i+1))

creating threads

for i in range (5):
 worker = MyWorker("Th {}".format(i+1), myqueue)
 worker.daemon = True
 worker.start()

myqueue.join()
```

In this code example, we created five worker threads using the custom thread class (**MyThread**). These five worker threads access the queue to get the task item from it. After getting the task item, the threads sleep for 1 second and then print the thread name and the task name. For each **get** call for an item of a queue, a subsequent call of **task\_done()** indicates that the processing of the task has been completed.

It is important to note that we used the **join** method on the **myqueue** object and not on the threads. The **join** method on the queue blocks the main thread until all items in the queue have been processed and completed (**task\_done** is called for them). This is a recommended way to block the main thread when a queue object is used to hold the tasks' data for threads.

Next, we will implement an application to download files from Google Drive using the **Thread** class, the **Queue** class, and a couple of third-party libraries.

## Case study – a multithreaded application to download files from Google Drive

We have discussed in the previous section that multithreaded applications in Python stand out well when different threads are working on input and output tasks. That is why we selected to implement an application that downloads files from a shared directory of Google Drive. To implement this application, we will need the following:

- **Google Drive:** A Google Drive account (a free basic account is fine) with one directory marked as shared.
- **API key:** An API key to access Google APIs is required. The API key needs to be enabled to use the Google APIs for Google Drive. The API can be enabled by following the guidelines on the Google Developers site (<https://developers.google.com/drive/api/v3/enable-drive-api>).
- **getfilelistpy:** This is a third-party library that gets a list of files from a Google Drive shared directory. This library can be installed using the **pip** tool.
- **gdown:** This is a third-party library that downloads a file from Google Drive. This library can also be installed through the **pip** tool as well. There are other libraries available that offer the same functionality. We selected the **gdown** library for its ease of use.

To use the **getfilelistpy** module, we need to create a resource data structure. This data structure will include a folder identifier as **id** (this will be Google Drive folder ID in our case), the API security key (**api\_key**) for accessing the Google Drive folder, and a list of file attributes (**fields**) to be fetched when we get a list of files. We build the resource data structure as follows:

```
resource = {
 "api_key": "AIzaSyDYKmm85kebxddKrGns4z0",
 "id": "0B8TxHW2Ci6dbckVwTRtTl3RUU",
 "fields": "files(name, id, webContentLink)",
```

```
}
```

'''API key and id used in the examples are not original, so should be replaced as per your account and shared directory id'''

We limit the file attributes to the **file id**, **name**, and its **web link** (URL) only. Next, we need to add each file item into a queue as a task for threads. The queue will be used by multiple worker threads to download the files in parallel.

To make the application more flexible in terms of the number of workers we can use, we build a pool of worker threads. The size of the pool is controlled by a global variable that is set at the beginning of the program. We created worker threads as per the size of the thread pool. Each worker thread in the pool has access to the queue, which has a list of files. Like the previous code example, each worker thread will take one file item from the queue at a time, download the file, and mark the file item as complete using the **task\_done** method. An example code for defining a resource data structure and for defining a class for the worker thread is as follows:

```
#threads_casestudy.py
from queue import Queue
from threading import Thread
import time
from getfilelistpy import getfilelist
import gdown
THREAD_POOL_SIZE = 1
resource = {
 "api_key": "AIzaSyDYKmm85kea2bxddKrGns4z0",
 "id": "0B8TxHW2Ci6dbckVweTRtTl3RUU ",
 "fields": "files(name,id,webContentLink)",
}
class DownlaodWorker(Thread):
 def __init__(self, name, queue):
 Thread.__init__(self)
 self.name = name
 self.queue = queue
 def run(self):
 while True:
 # Get the file id and name from the queue
 item1 = self.queue.get()
```

```

try:
 gdown.download(item1['webContentLink'],
 './files/{}'.format(item1['name']),
 quiet=False)
finally:
 self.queue.task_done()

```

We get the files' metadata from a Google Drive directory using the resource data structure as follows:

```

def get_files(resource):
 #global files_list
 res = getfilelist.GetFileList(resource)
 files_list = res['fileList'][0]
 return files_list

```

In the **main** function, we create a **Queue** object to insert file metadata into the queue. The **Queue** object is handed over to a pool of worker threads for downloading the files. The worker threads will download the files, as discussed earlier. We use the **time** class to measure the time it takes to complete the download of all the files from the Google Drive directory. The code for the **main** function is as follows:

```

def main():
 start_time = time.monotonic()
 files = get_files(resource)
 #add files info into the queue
 queue = Queue()
 for item in files['files']:
 queue.put(item)
 for i in range (THREAD_POOL_SIZE):
 worker = DownlaodWorker("Thread {}".format(i+1),
 queue)
 worker.daemon = True
 worker.start()
 queue.join()
 end_time = time.monotonic()
 print('Time taken to download: {} seconds'.
 format(end_time - start_time))

```

```
main()
```

For this application, we have 10 files in the Google Drive directory, varying in size from 500 KB to 3 MB. We ran the application with 1, 5, and 10 worker threads. The total time taken to download the 10 files with 1 thread was approximately 20 seconds. This is almost equivalent to writing a code without any threads. In fact, we have written a code to download the same files without any threads and made it available with this book's source code as an example. The time it took to download 10 files with a non-threaded application was approximately 19 seconds.

When we changed the number of worker threads to 5, the time taken to download the 10 files reduced significantly to approximately 6 seconds on our MacBook machine (Intel Core i5 with 16 GB RAM). If you run the same program on your computer, the time may be different, but there will definitely be an improvement if we increase the number of worker threads. With 10 threads, we observed the execution time to be around 4 seconds. This observation shows that there is an improvement in the execution time for I/O bound tasks by using multithreading regardless of the GIL limitation it has.

This concludes our discussion of how to implement threads in Python and how to benefit from different locking mechanisms using the **Lock** class and the **Queue** class. Next, we will discuss multiprocessing programming in Python.

## Going beyond a single CPU – implementing multiprocessing

We have seen the complexity of multithreaded programming and its limitations. The question is whether the complexity of multithreading is worth the effort. It may be worth it for I/O-related tasks but not for general application use cases, especially when an alternative approach exists. The alternative approach is to use multiprocessing because separate Python processes are not constrained by the GIL and execution can happen in parallel. This is especially beneficial when applications run on multicore processors and involve intensive CPU-demanding tasks. In reality, the use of multiprocessing is the only option in Python's built-in libraries to utilize multiple processor cores.

**Graphics Processing Units (GPUs)** provide a greater number of cores than regular CPUs and are considered more suitable for data processing tasks, especially when executing them in parallel. The only caveat is that in order to execute a data processing program on a GPU, we have to transfer the data from the main memory to the GPU's memory. This additional step of data transfer will be compensated when we are processing a large dataset. But there will be little or no benefit if our dataset is small. Using GPUs for big data processing, especially for training machine learning models, is becoming a popular option. NVIDIA has introduced a GPU for parallel processing called CUDA, which is well supported through external libraries in Python.

Each process has a data structure called the **Process Control Block (PCB)** at the operating system level. Like the TCB, the PCB has a **Process ID (PID)** for process identification, stores the state of the process (such as running or waiting), and has a program counter, CPU registers, CPU scheduling information, and many more attributes.

In the case of multiple processes for CPUs, there is no sharing of memory natively. This means there is a lower chance of data corruption. If the two processes have to share the data, they need to use some interprocess communication mechanism. Python supports interprocess communication through its primitives. In the next subsections, we will first discuss the fundamentals of creating processes in Python and then discuss how to achieve interprocess communication.

## Creating multiple processes

For multiprocessing programming, Python provides a **multiprocessing** package that is very similar to the multithreading package. The **multiprocessing** package includes two approaches to implement multiprocessing, which are using the **Process** object and the **Pool** object. We will discuss each of these approaches one by one.

### Using the Process object

The processes can be spawned by creating a **Process** object and then using its **start** method similar to the **start** method for starting a **Thread** object. In fact, the **Process** object offers the same API as the **Thread** object. A simple code example for creating multiple child processes is as follows:

```
process1.py to create simple processes with function
import os
from multiprocessing import Process, current_process as cp
from time import sleep

def print_hello():
 sleep(2)
 print("{}-{}: Hello".format(os.getpid(), cp().name))

def print_message(msg):
 sleep(1)
 print("{}-{}: {}".format(os.getpid(), cp().name, msg))

def main():
 processes = []
 # creating process
 processes.append(Process(target=print_hello, name="Process 1"))

 processes.append(Process(target=print_message, name="Process 2"))

 processes.append(Process(target=print_hello, name="Process 3"))

 processes.append(Process(target=print_message, name="Process 4"))

 processes.append(Process(target=print_hello, name="Process 5"))

 processes.append(Process(target=print_message, name="Process 6"))

 processes.append(Process(target=print_hello, name="Process 7"))

 processes.append(Process(target=print_message, name="Process 8"))

 processes.append(Process(target=print_hello, name="Process 9"))

 processes.append(Process(target=print_message, name="Process 10"))

 for process in processes:
 process.start()

 for process in processes:
 process.join()
```

```

processes.append(Process(target=print_hello, name="Process 2"))

processes.append(Process(target=print_message, args=["Good morning"], name="Process 3"))

start the process

for p in processes:

 p.start()

wait till all are done

for p in processes:

 p.join()

print("Exiting the main process")

if __name__ == '__main__':
 main()

```

As already mentioned, the methods used for the **Process** object are pretty much the same as those used for the **Thread** object. The explanation of this example is the same as for the example code in the multithreading code examples.

## Using the Pool object

The **Pool** object offers a convenient way (using its **map** method) of creating processes, assigning functions to each new process, and distributing input parameters across the processes. We selected the code example with a pool size of 3 but provided input parameters for five processes. The reason for setting the pool size to 3 is to make sure a maximum of three child processes are active at a time, regardless of how many parameters we pass with the **map** method of the **Pool** object. The additional parameters will be handed over to the same child processes as soon they finish their current execution. Here is a code example with a pool size of 3:

```

process2.py to create processes using a pool

import os

from multiprocessing import Process, Pool, current_process as cp
from time import sleep

def print_message(msg):
 sleep(1)
 print("{}-{}: {}".format(os.getpid(), cp().name, msg))

def main():
 # creating process from a pool
 with Pool(3) as proc:
 proc.map(print_message, ["Orange", "Apple", "Banana",

```

```

 "Grapes", "Pears"])

print("Exiting the main process")

if __name__ == '__main__':
 main()

```

The magic of distributing input parameters to a function that is tied to a set of pool processes is done by the **map** method. The **map** method waits until all functions complete their execution, and that is why there is no need to use a **join** method if the processes are created using the **Pool** object.

A few differences between using the **Process** object versus the **Pool** object are shown in the following table:

Using the Pool object	Using the Process object
Only active processes stay in memory	All created processes stay in memory
Works better for large datasets and for repetitive tasks	Works better for small datasets
Processes block on I/O operation until I/O resource is granted	Processes are not blocked on the I/O operation

Table 7.1 – Comparison of using the Pool object and the Process object

Next, we will discuss how to exchange data between processes.

## Sharing data between processes

There are two approaches in the multiprocessing package to share data between processes. These are **shared memory** and **server process**. They are described next.

### Using shared ctype objects (shared memory)

In this case, a *shared memory* block is created, and the processes have access to this shared memory block. The shared memory is created as soon we initiate one of the **ctype** datatypes available in the **multiprocessing** package. The datatypes are **Array** and **Value**. The **Array** datatype is a **ctype** array and the **Value** datatype is a generic **ctype** object, both of which are allocated from the shared memory. To create a **ctype** array, we will use a statement like the following:

```
mylist = multiprocessing.Array('i', 5)
```

This will create an array of the **integer** datatype with a size of 5. **i** is one of the typecodes, and it stands for integer. We can use the **d** typecode for float datatypes. We can also initialize the array by providing the sequence as a second argument (instead of the size) as follows:

```
mylist = multiprocessing.Array('i', [1,2,3,4,5])
```

To create a **Value ctype** object, we will use a statement similar to the following:

```
obj = multiprocessing.Value('i')
```

This will create an object of the **integer** datatype because the typecode is set to **i**. The value of this object can be accessed or set by using the **value** attribute.

Both these **ctype** objects have **Lock** as an optional argument, which is set to **True** by default. This argument when set to **True** is used to create a new recursive lock object that provides synchronized access to the value of the objects. If it is set to **False**, there will be no protection and it will not be a safe process. If your process is only accessing the shared memory for reading purposes, it is fine to set the **Lock** to **False**. We leave this **Lock** argument as the default (**True**) in our next code examples.

To illustrate the use of these **ctype** objects from the shared memory, we will create a default list with three numeric values, a **ctype** array of size 3 to hold the incremented values of the original array, and a **ctype** object to hold the sum of the incremented array. These objects will be created by a parent process in shared memory and will be accessed and updated by a child process from the shared memory. This interaction of the parent and the child processes with the shared memory is shown in the following figure:

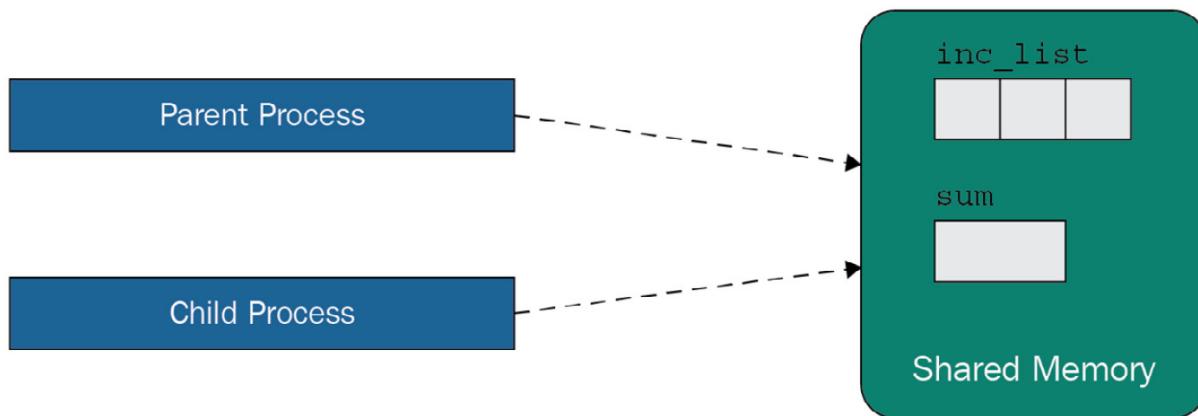


Figure 7.3 – Use of shared memory by a parent and a child process

A complete code example of using the shared memory is shown next:

```
process3.py to use shared memory ctype objects

import multiprocessing

from multiprocessing import Process, Pool, current_process as cp

def inc_sum_list(list, inc_list, sum):
 sum.value = 0
 for index, num in enumerate(list):
```

```

inc_list[index] = num + 1
sum.value = sum.value + inc_list[index]

def main():
 mylist = [2, 5, 7]
 inc_list = multiprocessing.Array('i', 3)
 sum = multiprocessing.Value('i')
 p = Process(target=inc_sum_list,
 args=(mylist, inc_list, sum))
 p.start()
 p.join()
 print("incremented list: ", list(inc_list))
 print("sum of inc list: ", sum.value)
 print("Exiting the main process")
if __name__ == '__main__':
 main()

```

The shared datatypes (**inc\_list** and **sum** in this case) are accessed by both the parent process and the child process. It is important to mention that using the shared memory is not a recommended option because it requires synchronization and locking mechanisms (similar to what we did for multithreading) when the same shared memory objects are accessed by multiple processes and the **Lock** argument is set to **False**.

The next approach of sharing data between processes is using the server process.

### **Using the server process**

In this case, a server process is started as soon as a Python program starts. This new process is used to create and manage the new child processes requested by a parent process. This server process can hold Python objects that other processes can access using proxies.

To implement the server process and share the objects between the processes, the **multiprocessing** package provides a **Manager** object. The **Manager** object supports different data types such as the following:

- Lists
- Dictionaries
- Locks
- Rlocks
- Queues
- Values

- Arrays

The code example we selected for illustrating the server process creates a **dictionary** object using the **Manager** object, then passes the dictionary object to different child processes to insert more data and to print out the dictionary contents. We will create three child processes for our example: two for inserting data into the dictionary object and one for getting the dictionary contents as the console output. The interaction between the parent process, the server process, and the three child processes is shown in *Figure 7.4*. The parent process creates the server process as soon as a new process request is executed using the *Manager* context. The child processes are created and managed by the server process. The shared data is available within the server process and is accessible by all processes, including the parent process:

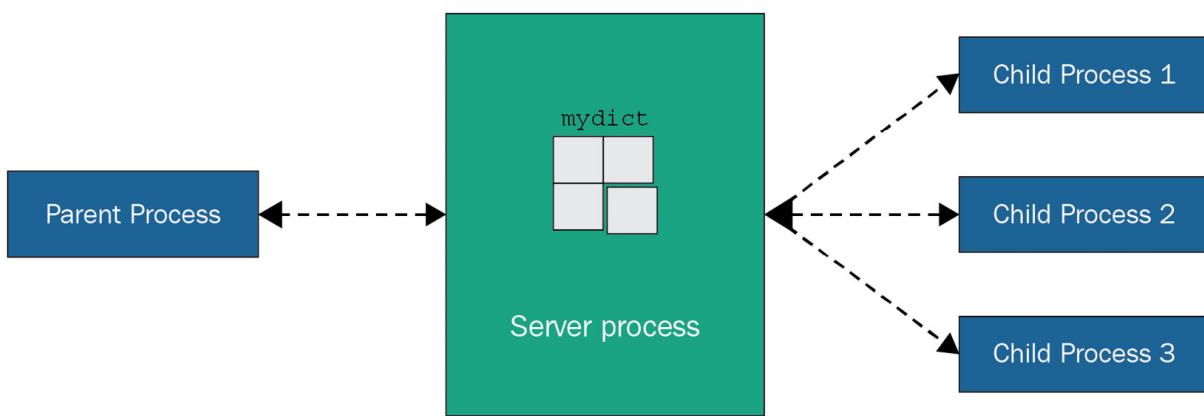


Figure 7.4 – Use of server process for sharing data between processes

The complete code example is shown next:

```

process4.py to use shared memory using the server process
import multiprocessing
from multiprocessing import Process, Manager
def insert_data(dict1, code, subject):
 dict1[code] = subject
def output(dict1):
 print("Dictionary data: ", dict1)
def main():
 with multiprocessing.Manager() as mgr:
 # create a dictionary in the server process
 mydict = mgr.dict({100: "Maths", 200: "Science"})
 p1 = Process(target=insert_data, args=(mydict, 300, "English"))
 p2 = Process(target=insert_data, args=(mydict, 400, "French"))

```

```

p3 = Process(target=output, args=(mydict,))

p1.start()
p2.start()

p1.join()
p2.join()

p3.start()

p3.join()

print("Exiting the main process")

if __name__ == '__main__':
 main()

```

The server process approach offers more flexibility than the shared memory approach because it supports a large variety of object types. However, this comes at the cost of slower performance compared to the shared memory approach.

In the next section, we will explore the options of direct communication between the processes.

## Exchanging objects between processes

In the previous section, we studied how to share data between the processes through an external memory block or a new process. In this section, we will investigate exchanging of data between processes using Python objects. The **multiprocessing** module provides two options for this purpose. These are using the **Queue** object and the **Pipe** object.

### Using the Queue object

The **Queue** object is available from the **multiprocessing** package and is nearly the same as the synchronized queue object (**queue.Queue**) that we used for multithreading. This **Queue** object is process-safe and does not require any additional protection. A code example to illustrate the use of the multiprocessing **Queue** object for data exchange is shown next:

```

process5.py to use queue to exchange data

import multiprocessing

from multiprocessing import Process, Queue

def copy_data (list, myqueue):
 for num in list:
 myqueue.put(num)

def output(myqueue):
 while not myqueue.empty():

```

```

 print(myqueue.get())

def main():
 mylist = [2, 5, 7]
 myqueue = Queue()
 p1 = Process(target=copy_data, args=(mylist, myqueue))
 p2 = Process(target=output, args=(myqueue,))
 p1.start()
 p1.join()
 p2.start()
 p2.join()
 print("Queue is empty: ", myqueue.empty())
 print("Exiting the main process")
if __name__ == '__main__':
 main()

```

In this code example, we created a standard **list** object and a multiprocessing **Queue** object. The **list** and **Queue** objects are passed to a new process, which is attached to a function called **copy\_data**. This function will copy the data from the **list** object to the **Queue** object. A new process is initiated to print the contents of the **Queue** object. Note that the data in the **Queue** object is set by the previous process and the data will be available to the new process. This is a convenient way to exchange data without adding the complexity of shared memory or the server process.

## Using the Pipe object

The **Pipe** object is like a pipe between two processes for exchanging data. This is why this object is especially useful when two-way communication is required. When we create a **Pipe** object, it provides two connection objects, which are the two ends of the **Pipe** object. Each connection object provides a **send** and a **recv** method to send and receive data.

To illustrate the concept and use of the **Pipe** object, we will create two functions that will be attached to two separate processes:

- The first function is for sending the message through a **Pipe** object connection. We will send a few data messages and finish the communication with a **BYE** message.
- The second function is to receive the message using the other connection object of the **Pipe** object. This function will run in an infinite loop until it receives a **BYE** message.

The two functions (or processes) are provided with the two connection objects of a pipe. The complete code is as follows:

```
process6.py to use Pipe to exchange data
```

```

from multiprocessing import Process, Pipe

def mysender (s_conn):
 s_conn.send({100, "Maths"})
 s_conn.send({200, "Science"})
 s_conn.send("BYE")

def myreceiver(r_conn):
 while True:
 msg = r_conn.recv()
 if msg == "BYE":
 break
 print("Received message : ", msg)

def main():
 sender_conn, receiver_conn= Pipe()
 p1 = Process(target=mysender, args=(sender_conn,))
 p2 = Process(target=myreceiver, args=(receiver_conn,))
 p1.start()
 p2.start()
 p1.join()
 p2.join()
 print("Exiting the main process")

if __name__ == '__main__':
 main()

```

It is important to mention that the data in a **Pipe** object can easily be corrupted if the two processes try to read from or write to it using the same connection object at the same time. That is why multiprocessing queues are the preferred option: because they provide proper synchronization between the processes.

## Synchronization between processes

Synchronization between processes makes sure that two or more processes do not access the same resources or program code at the same time, which is also called the **critical section**. Such a situation can lead to a race condition, which can corrupt the data. The chance of a race condition occurring among different processes is not very high, but it is still possible if they are using shared memory or accessing the same resources. These situations can be avoided either by using the appropriate objects

with built-in synchronization or by using the **Lock** object, similar to what we used in the case of multithreading.

We illustrated the use of **queues** and **ctype** datatypes with **Lock** set to **True**, which is process safe. In the next code example, we will illustrate the use of the **Lock** object to make sure one process gets access to the console output at a time. We created the processes using the **Pool** object and to pass the same **Lock** object to all processes, we used the **Lock** from the **Manager** object and not the one from the multiprocessing package. We also used the **partial** function to tie the **Lock** object to each process, along with a list to be distributed to each process function. Here is the complete code example:

```
process7.py to show synchronization and locking
from functools import partial
from multiprocessing import Pool, Manager
def printme (lock, msg):
 lock.acquire()
 try:
 print(msg)
 finally:
 lock.release()
def main():
 with Pool(3) as proc:
 lock = Manager().Lock()
 func = partial(printme,lock)
 proc.map(func, ["Orange", "Apple", "Banana",
 "Grapes", "Pears"])
 print("Exiting the main process")
if __name__ == '__main__':
 main()
```

If we do not use the **Lock** object, the output from the different processes can be mixed up.

## Case study – a multiprocessor application to download files from Google Drive

In this section, we will implement the same case study as we did in the *Case study – a multithreaded application to download files from Google Drive* section, but using processors instead. The

prerequisites and goals are the same as described for the case study of the multithreaded application.

For this application, we used the same code that we built for the multithreaded application except that we used processes instead of threads. Another difference is that we used the **JoinableQueue** object from the **multiprocessing** module to achieve the same functionality as we were getting from the regular **Queue** object. Code for defining a resource data structure and for a function to download files from Google Drive is as follows:

```
#processes_casestudy.py

import time

from multiprocessing import Process, JoinableQueue
from getfilelistpy import getfilelist
import gdown

PROCESSES_POOL_SIZE = 5

resource = {
 "api_key": "AIzaSyDYKmm85keqnk4bF1Da2bxddKrGns4z0",
 "id": "0B8TxHW2Ci6dbckVwetTlV3RUU",
 "fields": "files(name,id,webContentLink)",
}

def mydownloader(queue):
 while True:
 # Get the file id and name from the queue
 item1 = queue.get()
 try:
 gdown.download(item1['webContentLink'],
 './files/{}'.format(item1['name']),
 quiet=False)
 finally:
 queue.task_done()
```

We get the files' metadata, such as the name and HTTP link, from a Google Drive directory using the resource data structure as follows:

```
def get_files(resource):
 res = getfilelist.GetFileList(resource)
 files_list = res['fileList'][0]
 return files_list
```

In our **main** function, we create a **JoinableQueue** object and insert the files' metadata into the queue. The queue will be handed over to a pool of processes to download the files. The processes will download the files. We used the **time** class to measure the time it takes to download all the files from the Google Drive directory. The code for the **main** function is as follows:

```
def main():
 files = get_files(resource)
 #add files info into the queue
 myqueue = JoinableQueue()
 for item in files['files']:
 myqueue.put(item)
 processes = []
 for id in range(PROCESSES_POOL_SIZE):
 p = Process(target=mydownloader, args=(myqueue,))
 p.daemon = True
 p.start()
 start_time = time.monotonic()
 myqueue.join()
 total_exec_time = time.monotonic() - start_time
 print(f'Time taken to download: {total_exec_time:.2f} seconds')
if __name__ == '__main__':
 main()
```

We ran this application by varying the different number of processes, such as **3**, **5**, **7**, and **10**. We found that the time it took to download the same files (as for the case study of multithreading) is slightly better than with the multithreaded application. The execution time will vary from machine to machine, but on our machine (MacBook Pro: Intel Core i5 with 16 GB RAM), it took around 5 seconds with 5 processes and 3 seconds with 10 processes running in parallel. This improvement of 1 second over the multithreaded application is in line with the expected results as multiprocessing provides true concurrency.

## Using asynchronous programming for responsive systems

With multiprocessing and multithreaded programming, we were mostly dealing with synchronous programming, where we request something and wait for the response to be received before we move to the next block of code. If any context switching is applied, it is provided by the operating system. Asynchronous programming in Python is different mainly in the following two aspects:

- The tasks are to be created for asynchronous execution. This means the parent caller does not have to wait for the response from another process. The process will respond to the caller once it finishes the execution.
- The operating system is no longer managing the context switching between the processes and the threads. The asynchronous program will be given only a single thread in a process, but we can do multiple things with it. In this style of execution, every process or task voluntarily releases control whenever it is idle or waiting for another resource to make sure that the other tasks get a turn. This concept is called **cooperative multitasking**.

Cooperative multitasking is an effective tool for achieving concurrency at the application level. In cooperative multitasking, we do not build processes or threads, but tasks, which comprises **tasklets**, or **coroutines**, and or **green threads**. These tasks are coordinated by a single function known as an **event loop**. The event loop registers the tasks and handles the flow of control between the tasks. The beauty of the event loop in Python is that it is implemented using generators, and these generators can execute a function and pause it at a specific point (using **yield**) while keeping the stack of objects under control before it is resumed.

For a system based on cooperative multitasking, there is always a question of when to release the control back to a scheduler or to an event loop. The most commonly used logic is to use the I/O operation as the event to release the control because there is always a waiting time involved whenever we are doing an I/O operation.

But hold on, is it not the same logic we used for multithreading? We found that multithreading improves application performance when dealing with I/O operations. But there is a difference here. In the case of multithreading, the operating system is managing the context switching between the threads, and it can preempt any running thread for any reason and give control to another thread. But in asynchronous programming or cooperative multitasking, the tasks or coroutines are not visible to the operating systems and cannot be preempted. The coroutines in fact cannot be preempted by the main event loop. But this does not mean that the operating system cannot preempt the whole Python process. The main Python process is still competing for resources with other applications and processes at the operating system level.

In the next section, we will discuss a few building blocks of asynchronous programming in Python, which is provided by the **asyncio** module, and we will conclude with a comprehensive case study.

## Understanding the **asyncio** module

The **asyncio** module is available in Python 3.5 or later to write concurrent programs using the **async/await** syntax. But it is recommended to use Python 3.7 or later to build any serious **asyncio** application. The library is rich with features and supports creating and running Python coroutines, performing network I/O operations, distributing tasks to queues, and synchronizing concurrent code.

We will start with how to write and execute coroutines and tasks.

## Coroutines and tasks

Coroutines are the functions that are to be executed asynchronously. A simple example of sending a string to the console output using a coroutine is as follows:

```
#asyncio1.py to build a basic coroutine

import asyncio
import time

async def say(delay, msg):
 await asyncio.sleep(delay)
 print(msg)

print("Started at ", time.strftime("%X"))

asyncio.run(say(1, "Good"))
asyncio.run(say(2, "Morning"))

print("Stopped at ", time.strftime("%X"))
```

In this code example, it is important to note the following:

- The coroutine takes **delay** and **msg** arguments. The **delay** argument is used to add a delay before sending the **msg** string to the console output.
- We used the **asyncio.sleep** function instead of the traditional **time.sleep** function. If the **time.sleep** function is used, control will not be given back to the event loop. That is why it is important to use the compatible **asyncio.sleep** function.
- The coroutine is executed twice with two different values of the **delay** argument by using the **run** method. The **run** method will not execute the coroutines concurrently.

The console output of this program will be as follows. This shows that the coroutines are executed one after the other as the total delay added is 3 seconds:

```
Started at 15:59:55
Good
Morning
Stopped at 15:59:58
```

To run the coroutines in parallel, we need to use the **create\_task** function from the **asyncio** module. This function creates a task that can be used to schedule coroutines to run concurrently.

The next code example is a revised version of **asyncio1.py**, in which we wrapped the coroutine (**say** in our case) into a task using the **create\_task** function. In this revised version, we created two tasks that are wrapping the **say** coroutine. We waited for the two tasks to be completed using the **await** keyword:

```
#asyncio2.py to build and run coroutines in parallel

import asyncio
import time

async def say(delay, msg):
 await asyncio.sleep(delay)
 print(msg)

async def main():
 task1 = asyncio.create_task(say(1, 'Good'))
 task2 = asyncio.create_task(say(1, 'Morning'))

 print("Started at ", time.strftime("%X"))

 await task1
 await task2

 print("Stopped at ", time.strftime("%X"))

asyncio.run(main())
```

The console output of this program is as follows:

```
Started at 16:04:40
```

```
Good
```

```
Morning
```

```
Stopped at 16:04:41
```

This console output shows that the two tasks were completed in 1 second, which is proof that the tasks are executed in parallel.

## Using awaitable objects

An object is awaitable if we can apply the **await** statement to it. The majority of **asyncio** functions and modules inside it are designed to work with awaitable objects. But most Python objects and third-party libraries are not built for asynchronous programming. It is important to select compatible libraries that provide awaitable objects to use when building asynchronous applications.

Awaitable objects are split mainly into three types: coroutines, tasks, and **Futures**. We already discussed coroutines and tasks. A **Future** is a low-level object that is like a callback mechanism used to process the result coming from the **async/await**. The **Future** objects are typically not exposed for user-level programming.

## Running tasks concurrently

If we have to run multiple tasks in parallel, we can use the **await** keyword as we did in the previous example. But there is a better way of doing this by using the **gather** function. This function will run

the awaitable objects in the sequence provided. If any of the awaitable objects is a coroutine, it will be scheduled as a task. We will see the use of the **gather** function in the next section with a code example.

## Distributing tasks using queues

The **Queue** object in the **asyncio** package is similar to the **Queue** module but it is not thread safe. The **asyncio** module provides a variety of queue implementations, such as FIFO queues, priority queues, and LIFO queues. The queues in the **asyncio** module can be used to distribute the workloads to the tasks.

To illustrate the use of a queue with tasks, we will write a small program that will simulate the execution time of a real function by sleeping for a random amount of time. The random sleeping time is calculated for 10 such executions and added to a **Queue** object as working items by the main process. The **Queue** object is passed to a pool of three tasks. Each task in the pool executes the assigned coroutine, which consumes the execution time as per the queue entry available to it. The complete code is shown next:

```
#asyncio3.py to distribute work via queue
import asyncio
import random
import time

async def executer(name, queue):
 while True:
 exec_time = await queue.get()
 await asyncio.sleep(exec_time)
 queue.task_done()
 #print(f'{name} has taken {exec_time:.2f} seconds')

async def main():
 myqueue = asyncio.Queue()
 calc_exuection_time = 0
 for _ in range(10):
 sleep_for = random.uniform(0.4, 0.8)
 calc_exuection_time += sleep_for
 myqueue.put_nowait(sleep_for)

 tasks = []
```

```

for id in range(3):
 task = asyncio.create_task(executer(f'Task-{id+1}', myqueue))
 tasks.append(task)

start_time = time.monotonic()
await myqueue.join()

total_exec_time = time.monotonic() - start_time
for task in tasks:
 task.cancel()
await asyncio.gather(*tasks, return_exceptions=True)
print(f"Calculated execution time {calc_exuection_time:0.2f}")
print(f"Actual execution time {total_exec_time:0.2f}")

asyncio.run(main())

```

We used the **put\_no\_wait** function of the **Queue** object because it is a non-blocking operation. The console output of this program is as follows:

```

Calculated execution time 5.58
Actual execution time 2.05

```

This clearly shows that the tasks are executed in parallel, and the execution is three times better than if tasks are run sequentially.

So far, we have covered the fundamental concepts of the **asyncio** package in Python. Before concluding this topic, we will revisit the case study we did for the multithreading section by implementing it using the **asyncio** tasks.

## Case study – asyncio application to download files from Google Drive

We will implement the same case study as we did in the *Case study – a multithreaded application to download files from Google Drive* section, but using the **asyncio** module with **async**, **await**, and **async queue**. The prerequisites for this case study are the same except that we use the **aiohttp** and **aiofiles** library instead of the **gdown** library. The reason is simple: the **gdown** library is not built as an async module. There is no benefit of using it with async programming. This is an important point to consider whenever selecting libraries to be used with async applications.

For this application, we built a coroutine, **mydownloader**, to download a file from Google Drive using the **aiohttp** and **aiofiles** modules. This is shown in the following code, and the code that is different from the previous case studies is highlighted:

```

#asyncio_casestudy.py

import asyncio
import time
import aiofiles, aiohttp
from getfilelistpy import getfilelist
TASK_POOL_SIZE = 5
resource = {
 "api_key": "AIzaSyDYKmm85keqnk4bF1DpYa2dKrGns4z0",
 "id": "0B8TxHW2Ci6dbckVwetTlV3RUU",
 "fields": "files(name, id, webContentLink)",
}
async def mydownloader(name, queue):
 while True:
 # Get the file id and name from the queue
 item = await queue.get()
 try:
 async with aiohttp.ClientSession() as sess:
 async with sess.get(item['webContentLink'])
 as resp:
 if resp.status == 200:
 f = await
aiofiles.open('./files/{}'.format(
 item['name']), mode='wb')
 await f.write(await resp.read())
 await f.close()
 finally:
 print(f"{name}: Download completed for
", item['name'])
 queue.task_done()

```

The process to get the list of files from a shared Google Drive folder is the same as we used in the previous case study for multithreading and multiprocessing. In this case study, we created a pool of tasks (configurable) based on the **mydownloader** coroutine. These tasks are then scheduled to run together, and our parent process waits for all tasks to complete their execution. A code to get a list of files from Google Drive and then download the files using **asyncio** tasks is as follows:

```

def get_files(resource):
 res = getfilelist.GetFileList(resource)
 files_list = res['fileList'][0]
 return files_list

async def main():
 files = get_files(resource)
 #add files info into the queue
 myqueue = asyncio.Queue()
 for item in files['files']:
 myqueue.put_nowait(item)
 tasks = []
 for id in range(TASK_POOL_SIZE):
 task = asyncio.create_task(
 mydownloader(f'Task-{id+1}', myqueue))
 tasks.append(task)
 start_time = time.monotonic()
 await myqueue.join()
 total_exec_time = time.monotonic() - start_time
 for task in tasks:
 task.cancel()
 await asyncio.gather(*tasks, return_exceptions=True)
 print(f'Time taken to download: {total_exec_time:.2f} seconds')
asyncio.run(main())

```

We ran this application by varying the number of tasks, such as 3, 5, 7, and 10. We found that the time it took to download the files with the **asyncio** tasks is lower than when we downloaded the same files using the multithreading approach or the multiprocessing approach. The exact details of the time taken with the multithreading approach and the multiprocessing approach are available in the *Case study – a multithreaded application to download files from Google Drive* and *Case study – a multiprocessor application to download files from Google Drive* sections.

The execution time can vary from machine to machine, but on our machine (MacBook Pro: Intel Core i5 with 16 GB RAM), it took around 4 seconds with 5 tasks and 2 seconds with 10 tasks running in parallel. This is a significant improvement compared to the numbers we observed for the multithreading and multiprocessing case studies. This is in line with expected results, as **asyncio**

provides a better concurrency framework when it comes to I/O-related tasks, but it has to be implemented using the right set of programming objects.

This concludes our discussion of asynchronous programming. This section provided all the core ingredients to build an asynchronous application using the **asyncio** package.

## Summary

In this chapter, we discussed different options of concurrent programming in Python using the standard libraries. We started with multithreading with an introduction to the core concepts of concurrent programming. We introduced the challenges with multithreading, such as the GIL, which allows only one thread at a time to access Python objects. The concepts of locking and synchronization were explored with practical examples of Python code. We also discussed the types of task that multithreaded programming is more effective for using a case study.

We studied how to achieve concurrency using multiple processes in Python. With multiprocessing programming, we learned how to share data between processes using shared memory and the server process, and also how to exchange objects safely between processes using the **Queue** object and the **Pipe** object. In the end, we built the same case study as we did for the multithreading example, but using processes instead. Then, we introduced a completely different approach to achieving concurrency by using asynchronous programming. This was a complete shift in concept, and we started it by looking at the high-level concepts of the **async** and **await** keywords and how to build tasks, or coroutines, using the **asyncio** package. We concluded the chapter with the same case study we examined for multiprocessing and multithreading but using asynchronous programming.

This chapter has provided a lot of hands-on examples of how to implement concurrent applications in Python. This knowledge is important for anyone who wants to build multithreaded or asynchronous applications using the standard libraries available in Python.

In the next chapter, we will explore using third-party libraries to build concurrent applications in Python.

## Questions

1. What coordinates Python threads? Is it a Python interpreter?
2. What is the GIL in Python?
3. When should you use daemon threads?
4. For a system with limited memory, should we use a **Process** object or **Pool** object to create processes?
5. What are Futures in the **asyncio** package?

6. What is an event loop in asynchronous programming?
7. How do you write an asynchronous coroutine or function in Python?

## Further reading

- *Learning Concurrency in Python* by Elliot Forbes
- *Expert Python Programming* by Michal Jaworski and Tarek Ziade
- *Python 3 Object-Oriented Programming, Second Edition* by Dusty Phillips
- *Mastering Concurrency in Python* by Quan Nguyen
- *Python Concurrency with asyncio* by Mathew Fowler

## Answers

1. The threads and processes are coordinated by the operating system kernel.
2. Python's GIL is a locking mechanism used by Python to allow only one thread to execute at a time.
3. Daemon threads are used when it is not an issue for a thread to be terminated once its main thread terminates.
4. The **Pool** object keeps only the active processes in memory, so it is a better choice.
5. Futures are like a callback mechanism that is used to process the result coming from async/await calls.
6. An event loop object keeps track of tasks and handles the flow of control between them.
7. We can write an asynchronous coroutine by starting with **async def**.

[OceanofPDF.com](http://OceanofPDF.com)

## *Chapter 8: Scaling out Python Using Clusters*

In the previous chapter, we discussed parallel processing for a single machine using threads and processes. In this chapter, we will extend our discussion of parallel processing from a single machine to multiple machines in a cluster. A cluster is a group of computing devices that work together to perform compute-intensive tasks such as data processing. In particular, we will study Python's capabilities in the area of data-intensive computing. Data-intensive computing typically uses clusters for processing large volumes of data in parallel. Although there are quite a few frameworks and tools available for data-intensive computing, we will focus on **Apache Spark** as a data processing engine and PySpark as a Python library to build such applications.

If Apache Spark with Python is properly configured and implemented, the performance of your application can increase manyfold and surpass competitor platforms such as **Hadoop MapReduce**. We will also look into how distributed datasets are utilized in a clustered environment. This chapter will help you to understand the use of cluster computing platforms for large-scale data processing and how to implement data processing applications using Python. To illustrate the practical use of Python for applications with cluster computing requirements, we will include two case studies; the first one is to compute the value of Pi ( $\pi$ ) and the second one is to generate a word cloud from a data file.

We will cover the following topics in this chapter:

- Learning about the cluster options for parallel processing
- Introducing **resilient distributed datasets (RDD)**
- Using PySpark for parallel data processing
- Case studies of using Apache Spark and PySpark

By the end of this chapter, you will know how to work with Apache Spark and how you can write Python applications for data processing that can be executed on the worker nodes of an Apache Spark cluster.

## **Technical requirements**

The following are the technical requirements for this chapter:

- Python 3.7 or later installed on your computer
- An Apache Spark single-node cluster
- PySpark installed on top of Python 3.7 or later for driver program development

**NOTE**

*The Python version used with Apache Spark has to match the Python version that is used to run the driver program.*

The sample code for this chapter can be found at <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter08>.

We will start our discussion by looking at the cluster options available for parallel processing in general.

## Learning about the cluster options for parallel processing

When we have a large volume of data to process, it is not efficient and sometimes even not feasible to use a single machine with multiple cores to process the data efficiently. This is especially a challenge when working with real-time streaming data. For such scenarios, we need multiple systems that can process data in a distributed manner and perform these tasks on multiple machines in parallel. Using multiple machines to process compute-intensive tasks in parallel and in a distributed manner is called **cluster computing**. There are several big data distributed frameworks available to coordinate the execution of jobs in a cluster, but Hadoop MapReduce and Apache Spark are the leading contenders in this race. Both frameworks are open source projects from Apache. There are many variants (for example, Databricks) of these two platforms available with add-on features as well as maintenance support, but the fundamentals remain the same.

If we look at the market, the number of Hadoop MapReduce deployments may be higher compared to Apache Spark, but with its increasing popularity, Apache Spark is going to turn the tables eventually. Since Hadoop MapReduce is still very relevant due to its large install base, it is important to discuss what exactly Hadoop MapReduce is and how Apache Spark is becoming a better choice. Let's have a quick overview of the two in the next subsections.

## Hadoop MapReduce

Hadoop is a general-purpose distributed processing framework that offers the execution of large-scale data processing jobs across hundreds or thousands of computing nodes in a Hadoop cluster. The three core components of Hadoop are included in the following figure:

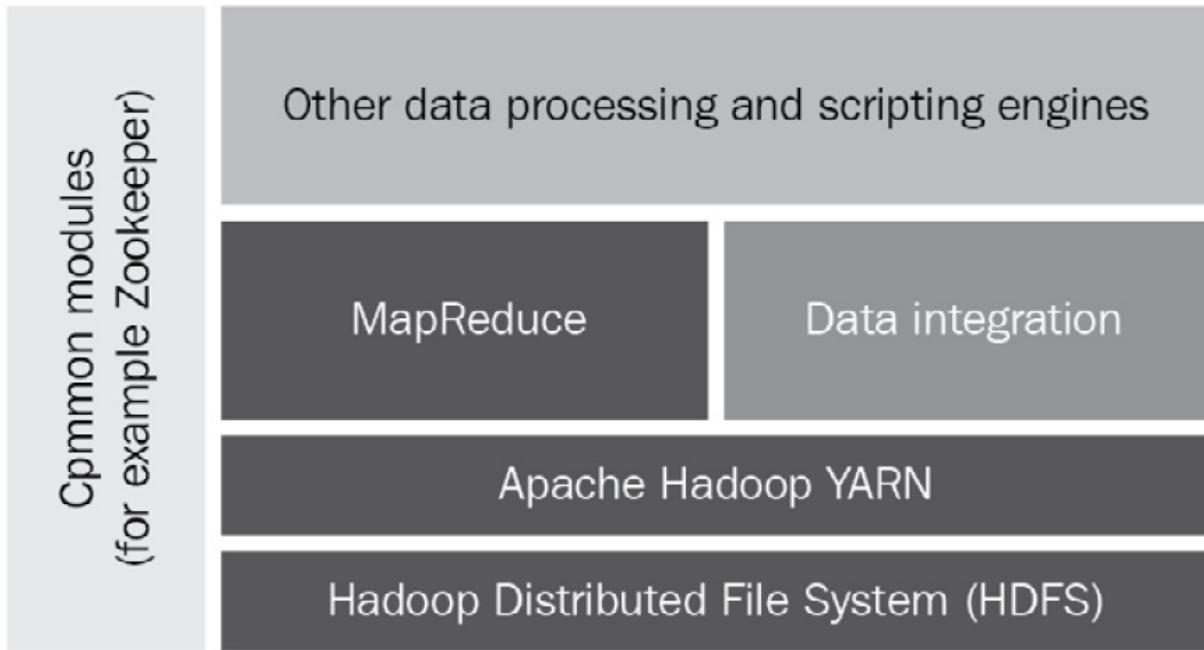


Figure 8.1 – Apache Hadoop MapReduce ecosystem

The three core components are described as follows:

- **Hadoop Distributed File System (HDFS)**: This is a Hadoop-native filesystem used to stores files such that those files can be parallelized across a cluster.
- **Yet Another Resource Negotiator (YARN)**: This is a system that processes data stored in HDFS and schedules the submitted jobs (for data processing) to be run by a processing system. The processing systems can be used for graph processing, stream processing, or batch processing.
- **MapReduce**: This is a programming framework that enables us to process large datasets by distributing the data into several small datasets. The MapReduce framework processes the data using two types of functions: mapper and reducer. The individual roles of the mapper (**map**) and reducer (**reduce**) functions are the same as we discussed in [Chapter 6, Advanced Tips and Tricks in Python](#). The key difference is that we use many **map** and **reduce** functions in parallel to process several datasets at the same time.

After breaking the large dataset into small datasets, we can provide the small datasets as input to many mapper functions for processing on different nodes of a Hadoop cluster. Each mapper function takes one set of data as an input, processes the data based on the goal set by the programmer, and produces the output as key-value pairs. Once the output of all the small datasets is available, one or multiple reducer functions will take the output from the mapper functions and aggregate the results as per the goals of the reducer functions.

To explain it in a bit more detail, we can take an example of counting particular words such as *attack* and *weapon* in a large source of text data. The text data can be divided into small datasets, for example, eight datasets. We can have eight mapper functions that count the two words within the dataset provided to them. Each mapper function provides us with the count of the words *attack* and *weapon* as an output for the dataset provided to it. In the next phase, the outputs of all

the mapper functions are provided to two reducer functions, one for each word. Each reducer function aggregates the count for each word and provides the aggregated results as an output. The operating of the MapReduce framework for this word count example is shown next. Note that the mapper function is typically implemented as **map** and the reducer function as **reduce** in Python programming:

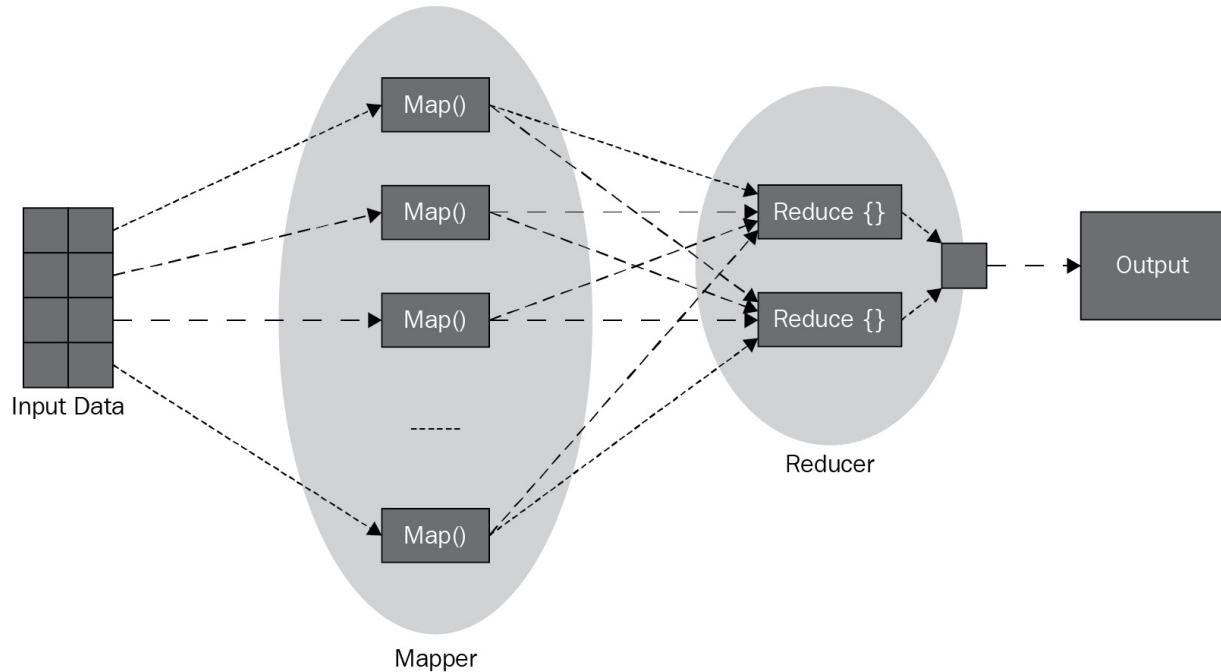


Figure 8.2 – Working of the MapReduce framework

We will skip the next levels of Hadoop components as they are not relevant to our discussion in this chapter. Hadoop is built mainly in Java, but any programming language, such as Python, can be used to write custom mapper and reducer components for the MapReduce module.

Hadoop MapReduce is good for processing a large chunk of data by breaking it into small blocks. The cluster nodes process these blocks separately and then the results are aggregated before being sent to the requester. Hadoop MapReduce processes the data from a filesystem and thus is not very efficient in terms of performance. However, it works very well if the speed of processing is not a critical requirement, for instance, if data processing can be scheduled to occur at night.

## Apache Spark

Apache Spark is an open source cluster computing framework for real-time as well as batch data processing. The main feature of Apache Spark is that it is an in-memory data processing framework,

which makes it efficient in terms of achieving low latency and makes it suitable for many real-world scenarios because of the following additional factors:

- It gets results quickly for mission-critical and time-sensitive applications such as real-time or near real-time scenarios.
- It's good for performing tasks repeatedly or iteratively in an efficient way due to in-memory processing.
- You can utilize out-of-the-box machine learning algorithms.
- You can leverage the support of additional programming languages such as Java, Python, Scala, and R.

In fact, Apache Spark covers a wide range of workloads, including batch data, iterative processing, and streaming data. The beauty of Apache Spark is that it can use Hadoop (via YARN) as a deployment cluster as well, but it has its own cluster manager as well.

At a high level, the main components of Apache Spark are segregated into three layers, as shown in the following figure:

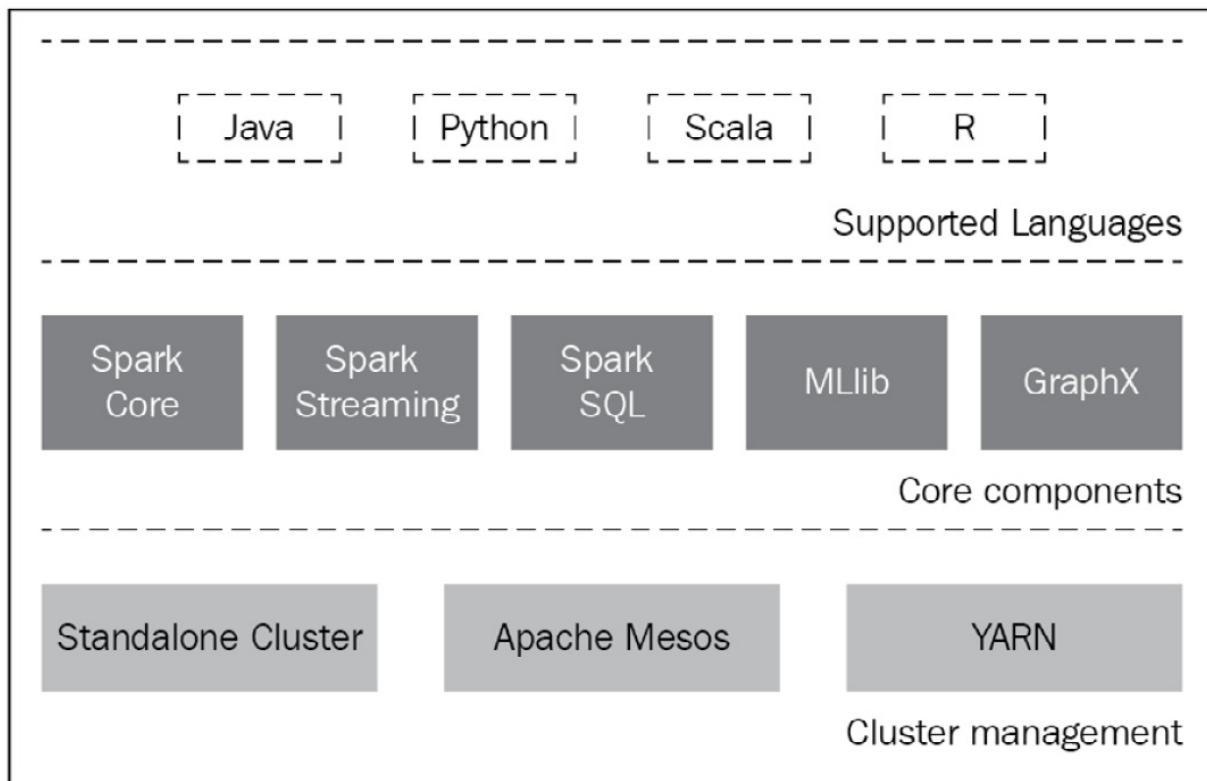


Figure 8.3 – Apache Spark ecosystem

These layers are discussed next.

### Support languages

Scala is a native language of Apache Spark, so it is quite popular for development. Apache Spark also provides high-level APIs for Java, Python, and R. In Apache Spark, multi-language support is

provided by using the **Remote Procedure Call (RPC)** interface. There is an RPC adapter written for each language in Scala that transforms the client requests written in a different language to the native Scala requests. This makes its adoption easier across the development community.

## Core components

A brief overview of each of the core components is discussed next:

- **Spark Core and RDDs:** Spark Core is a core engine of Spark and is responsible for providing abstraction to RDDs, scheduling and distributing jobs to a cluster, interacting with storage systems such as HDFS, Amazon S3, or an RDBMS, and managing memory and fault recoveries. An RDD is a resilient distributed dataset that is an immutable and distributable collection of data. RDDs are partitioned to be executed on the different nodes of a cluster. We will discuss RDDs in more detail in the next section.
- **Spark SQL:** This module is for querying data stored both in RDDs and in external data sources using abstracted interfaces. Using these common interfaces enables the developers to mix the SQL commands with the analytics tools for a given application.
- **Spark Streaming:** This module is used to process real-time data, which is critical to analyze live data streams with low latency.
- **MLlib:** The **Machine Learning Library (MLlib)** is used to apply machine learning algorithms in Apache Spark.
- **GraphX:** This module provides an API for graph-based parallel computing. This module comes with a variety of graph algorithms. Note that a graph is a mathematical concept based on vertices and edges that represents how a set of objects are related or dependent on each other. The objects are represented by vertices and their relationships by the edges.

## Cluster management

Apache Spark supports a few cluster managers, such as Standalone, Mesos, YARN, and Kubernetes. The key function of a cluster manager is to schedule and execute the jobs on cluster nodes and manage the resources on cluster nodes. But to interact with one or more cluster managers, there is a special object used in the main or driver program called **SparkSession**. Prior to release 2.0, the **SparkContext** object was considered as an entry point, but its API is now wrapped as part of the **SparkSession** object. Conceptually, the following figure shows the interaction of a **cluster manager**, **SparkSession (SparkContext)**, and the **worker nodes** in a cluster:

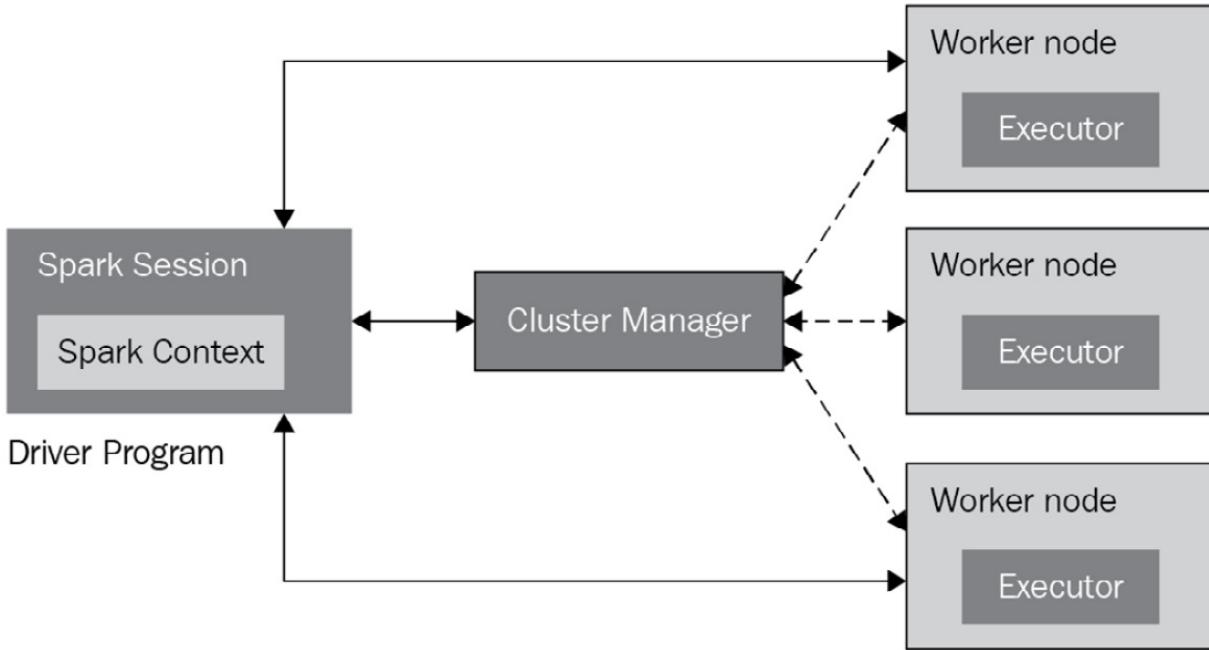


Figure 8.4 – Apache Spark ecosystem

The **SparkSession** object can connect to different types of cluster managers. Once connected, the executors are acquired on the cluster nodes through the cluster managers. The executors are the Spark processes that run the jobs and store the computational job results. The cluster manager on a master node is responsible for sending the application code to the executor processes on the worker nodes. Once the application code and data (if applicable) are moved to the worker nodes, the **SparkSession** object in a driver program interacts directly with executor processes for the execution of tasks.

As per Apache Spark release 3.1, the following cluster managers are supported:

- **Standalone:** This is a simple cluster manager that is included as part of the Spark Core engine. The Standalone cluster is based on master and worker (or slave) processes. A master process is basically a cluster manager, and the worker processes host the executors. Although the masters and the workers can be hosted on a single machine, this is not the real deployment scenario of a Spark Standalone cluster. It is recommended to distribute workers to different machines for the best outcome. The Standalone cluster is easy to set up and provides most of the features required from a cluster.
- **Apache Mesos:** This is another general-purpose cluster manager that can also run Hadoop MapReduce. For large-scale cluster environments, Apache Mesos is the preferred option. The idea of this cluster manager is that it aggregates the physical resources into a single virtual resource that acts as a cluster and provides a node-level abstraction. It is a distributed cluster manager by design.
- **Hadoop YARN:** This cluster manager is specific to Hadoop. This is also a distributed framework by nature.
- **Kubernetes:** This is more in the experimental phase. The purpose of this cluster manager is to automate the deployment and scaling of the containerized applications. The latest release of Apache Spark includes the Kubernetes scheduler.

Before concluding this section, it is worth mentioning another framework, **Dask**, which is an open source library written in Python for parallel computing. The Dask framework works directly with

distributed hardware platforms such as Hadoop. The Dask framework utilizes industry-proven libraries and Python projects such as NumPy, pandas, and scikit-learn. Dask is a small and lightweight framework compared to Apache Spark and can handle small to medium-sized clusters. In comparison, Apache Spark supports multiple languages and is the most appropriate choice for large-scale clusters.

After introducing the cluster options for parallel computing, we will discuss in the next section the core data structure of Apache Spark, which is the RDD.

## Introducing RDDs

The RDD is the core data structure in Apache Spark. This data structure is not only a distributed collection of objects but is also partitioned in such a way that each dataset can be processed and computed on different nodes of a cluster. This makes the RDD a core element of distributed data processing. Moreover, an RDD object is resilient in the sense that it is fault-tolerant and the framework can rebuild the data in the case of a failure. When we create an RDD object, the master node replicates the RDD object to multiple executors or worker nodes. If any executor process or worker node fails, the master node detects the failure and enables an executor process on another node to take over the execution. The new executor node will already have a copy of the RDD object, and it can start the execution immediately. Any data processed by the original executor node before failing will be lost data that will be computed again by the new executor node.

In the next subsections, we will learn about two key RDD operations and how to create RDD objects from different data sources.

## Learning RDD operations

An RDD is an immutable object, which means once it is created, it cannot be altered. But two types of operations can be performed on the data of an RDD. These are **transformations** and **actions**. These operations are described next.

### Transformations

These operations are applied on an RDD object and result in creating a new RDD object. This type of operation takes an RDD as input and produces one or more RDDs as an output. We also need to remember that these transformations are lazy in nature. This means they will only be executed when an action is triggered on them, which is another type of operation. To explain the concept of lazy evaluation, we can assume that we are transforming numeric data in an RDD by subtracting 1 from each element and then adding arithmetically (the action) all elements to the output RDD from the

transformation step. Because of the lazy evaluation, the transformation operation will not happen until we call the action operation (the addition, in this case).

There are several built-in transformation functions available with Apache Spark. The commonly used transformation functions are as follows:

- **map**: The **map** function iterates every element or line of an RDD object and applies the defined **map** function for each element.
- **filter**: This function will filter the data from the original RDD and provide a new RDD with the filtered results.
- **union**: This function is applied to two RDDs if they are of the same type and results in producing another RDD that is a union of the input RDDs.

## Actions

Actions are computational operations applied on an RDD and the results of such operations are to be returned to the driver program (for example, **SparkSession**). There are several built-in action functions available with Apache Spark. The commonly used action functions are as follows:

- **count**: The **count** action returns the number of elements in an RDD.
- **collect**: This action returns the entire RDD to the driver program.
- **reduce**: This action will reduce the elements from an RDD. A simple example is an addition operation on an RDD dataset.

For a complete list of transformation and action functions, we suggest you check the official documentation of Apache Spark. Next, we will study how to create RDDs.

## Creating RDD objects

There are three main approaches to create RDD objects, which are described next.

### Parallelizing a collection

This is one of the more simple approaches used in Apache Spark to create RDDs. In this approach, a collection is created or loaded into a program and then passed to the **parallelize** method of the **SparkContext** object. This approach is not used beyond development and testing. This is because it requires an entire dataset to be available on one machine, which is not convenient for a large amount of data.

### External datasets

Apache Spark supports distributed datasets from a local filesystem, HDFS, HBase, or even Amazon S3. In this approach of creating RDDs, the data is loaded directly from an external data source. There are convenient methods available with the **SparkContext** object that can be used to load all sorts of data into RDDs. For example, the **textFile** method can be used to load text data from local or remote resources using an appropriate URL (for example, **file://**, **hdfs://**, or **s3n://**).

## From existing RDDs

As discussed previously, RDDs can be created using transformation operations. This is one of the differentiators of Apache Spark from Hadoop MapReduce. The input RDD is not changed as it is an immutable object, but new RDDs can be created from existing RDDs. We have already seen some examples of how to create RDDs from existing RDDs using the **map** and **filter** functions.

This concludes our introduction of RDDs. In the next section, we will provide further details with Python code examples using the PySpark library.

## Using PySpark for parallel data processing

As discussed previously, Apache Spark is written in Scala language, which means there is no native support for Python. There is a large community of data scientists and analytics experts who prefer to use Python for data processing because of the rich set of libraries available with Python. Hence, it is not convenient to switch to using another programming language only for distributed data processing. Thus, integrating Python with Apache Spark is not only beneficial for the data science community but also opens the doors for many others who would like to adopt Apache Spark without learning or switching to a new programming language.

The Apache Spark community has built a Python library, **PySpark**, to facilitate working with Apache Spark using Python. To make the Python code work with Apache Spark, which is built on Scala (and Java), a Java library, **Py4J**, has been developed. This Py4J library is bundled with PySpark and allows the Python code to interact with JVM objects. This is the reason that when we install PySpark, we need to have JVM installed on our system first.

PySpark offers almost the same features and advantages as Apache Spark. These include in-memory computation, the ability to parallelize workloads, the use of the lazy evaluation design pattern, and support for multiple cluster managers such as Spark, YARN, and Mesos.

Installing PySpark (and Apache Spark) is beyond the scope of this chapter. The focus of this chapter is to discuss the use of PySpark to utilize the power of Apache Spark and not how to install Apache Spark and PySpark. But it is worth mentioning some installation options and dependencies.

There are many installation guides available online for each version of Apache Spark/PySpark and the various target platforms (for example Linux, macOS, and Windows). PySpark is included in the official release of Apache Spark, which can now be downloaded from the Apache Spark website (<https://spark.apache.org/>). PySpark is also available via the **pip** utility from PyPI, which can be used for a local setup or to connect to a remote cluster. Another option when installing PySpark is using **Anaconda**, which is another popular package and environment management system. If we are

installing PySpark along with Apache Spark, we need the following to be available or installed on the target machine:

- JVM
- Scala
- Apache Spark

For the code examples that will be discussed later, we have installed Apache Spark version 3.1.1 on macOS with PySpark included. PySpark comes with the **PySpark shell**, which is a CLI to the PySpark API. When the PySpark shell is started, it initializes the **SparkSession** and **SparkContext** objects automatically, which can be used to interact with the core Apache Spark engine. The following figure shows the initialization of the PySpark shell:

```
Welcome to
 //_/_/_/_/_/_/_/_/_/_/_
 / \ / \ . \ \ , / / / / \ \ \ \ \ \ \ \ \
 /_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/__/
 version 3.1.1

Using Python version 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018 23:26:24)
Spark context Web UI available at http://192.168.1.110:4040
Spark context available as 'sc' (master = local[*], app id = local-1628712798632).
SparkSession available as 'spark'.
>>>
```

Figure 8.5 – PySpark shell

From the initialization steps of the PySpark shell, we can observe the following:

- The **SparkContext** object is already created, and its instance is available in the shell as **sc**.
- The **SparkSession** object is also created, and its instance is available as **spark**. Now, **SparkSession** is an entry point to the PySpark framework to dynamically create RDD and DataFrame objects. The SparkSession object can also be created programmatically, and we will discuss this later with a code example.
- Apache Spark comes with a web UI and a web server to host the web UI, and it is initiated at **http://192.168.1.110:4040** for our local machine installation. Note that the IP address mentioned in this URL is a private address that is specific to our machine. Port **4040** is selected as the default port by Apache Spark. If this port is in use, Apache Spark will try to host on the next available port, such as **4041** or **4042**.

In the next subsections, we will learn how to create **SparkSession** objects, explore PySpark for RDD operations, and learn how to use PySpark DataFrames and PySpark SQL. We will start with creating a Spark session using Python.

## Creating SparkSession and SparkContext programs

Prior to Spark release 2.0, **SparkContext** was used as an entry point to PySpark. Since Spark release 2.0, **SparkSession** has been introduced as an entry point to the PySpark underlying framework to work with RDDs and DataFrames. **SparkSession** also includes all the APIs available in **SparkContext**, **SQLContext**, **StreamingContext**, and **HiveContext**. Now, **SparkSession** can also be created using the **SparkSession** class by using its **builder** method, which is illustrated in the next code example:

```
import pyspark
from pyspark.sql import SparkSession
spark1 = SparkSession.builder.master("local[2]")
 .appName('New App').getOrCreate()
```

When we run this code in the PySpark shell, which already has a default **SparkSession** object created as **spark**, it will return the same session as an output of this **builder** method. The following console output shows the location of the two **SparkSession** objects (**spark** and **spark1**), which confirms that they are pointing to the same **SparkSession** object:

```
>>> spark
<pyspark.sql.session.SparkSession object at 0x1091019e8>
>>> spark1
<pyspark.sql.session.SparkSession object at 0x1091019e8>
```

A few key concepts to understand regarding the **builder** method are as follows:

- **getOrCreate**: This method is the reason that we will get an already created session in the case of the PySpark shell. This method will create a new session if no session already exists; otherwise, it returns an already existing session.
- **master**: If we want to create a session connected to a cluster, we will provide the master name, which can be instance name of the Spark, or YARN, or Mesos cluster manager. If we are using a locally deployed Apache Spark option, we can use **local[n]**, where **n** is an integer greater than zero. The **n** will determine the number of partitions to be created for the RDD and DataFrame. For a local setup, **n** can be the number of CPU cores on the system. If we set it to **local[\*]**, which is a common practice, this will create as many worker threads as there are logical cores on the system.

If a new **SparkSession** object needs to be created, we can use the **newSession** method, which is available at the instance level of an existing **SparkSession** object. A code example of creating a new **SparkSession** object is shown next:

```
import pyspark
from pyspark.sql import SparkSession
spark2 = spark.newSession()
```

The console output for the **spark2** object confirms that this is a different session than the previously created **SparkSession** objects:

```
>>> spark2
<pyspark.sql.session.SparkSession object at 0x10910df98>
```

The **SparkContext** object can also be created programmatically. The easiest way to get a **SparkContext** object from a **SparkSession** instance is by using the **sparkContext** attribute. There is also a **SparkConext** class in the PySpark library that can also be used to create a **SparkContext** object directly, which was a common approach prior to Spark release 2.0.

#### *NOTE*

*We can have multiple **SparkSession** objects but only one **SparkContext** object per JVM.*

The **SparkSession** class offers a few more useful methods and attributes that are summarized next:

- **getActiveSession**: This method returns an active **SparkSession** under the current Spark thread.
- **createDataFrame**: This method creates a DataFrame object from an RDD, a list of objects, or a pandas DataFrame object.
- **conf**: This attribute returns the configuration interface for a Spark session.
- **catalog**: This attribute provides an interface to create, update, or query associated databases, functions, and tables.

A complete list of methods and attributes can be explored using the PySpark documentation for the **SparkSession** class at <https://spark.apache.org/docs/latest/api/python/reference/api/>.

## Exploring PySpark for RDD operations

In the *Introducing RDDs* section, we covered some of the key functions and operations of RDDs. In this section, we will extend the discussion in the context of PySpark with code examples.

### **Creating RDDs from a Python collection and from an external file**

We discussed a few ways to create RDDs in the previous section. In the following code examples, we will discuss how to create RDDs from an in-memory Python collection and from an external file resource. These two approaches are described next:

- To create an RDD from a Python data collection, we have a **parallelize** method available under the **sparkContext** instance. This method distributes the collection to form an RDD object. The method takes a collection as a parameter. An optional second parameter is available with the **parallelize** method to set the number of partitions to be created. By default, this method creates the partitions according to the number of cores available on the local machine or the number of cores set at the time of creating the **SparkSession** object.
- To create an RDD from an external file, we will use the **textFile** method available under the **sparkContext** instance. The **textFile** method can load a file as an RDD from HDFS or from a local filesystem (to be available on all cluster nodes). For local system-based deployment, an absolute and/or relative path can be provided. It is possible to set the minimum number of partitions to be created for the RDD using this method.

Some quick sample code (**rddcreate.py**) is shown next to illustrate the exact syntax of the PySpark statements to be used for the creation of a new RDD:

```

data = [5, 4, 6, 3, 2, 8, 9, 2, 8, 7,
 8, 4, 4, 8, 2, 7, 8, 9, 6, 9]
rdd1 = spark.sparkContext.parallelize(data)
print(rdd1.getNumPartitions())
rdd2 = spark.sparkContext.textFile('sample.txt')
print(rdd2.getNumPartitions())

```

Note that the **sample.txt** file has random text data, and its contents are not relevant for this code example.

## RDD transformation operations with PySpark

There are several built-in transformation operations available with PySpark. To illustrate how to implement a transformation operation such as **map** using PySpark, we will take a text file as an input and use the **map** function available with RDDs to transform it to another RDD. The sample code (**rddtransform1.py**) is shown next:

```

rdd1 = spark.sparkContext.textFile('sample.txt')
rdd2 = rdd1.map(lambda lines: lines.lower())
rdd3 = rdd1.map(lambda lines: lines.upper())
print(rdd2.collect())
print(rdd3.collect())

```

In this sample code, we applied two lambda functions with the **map** operation to convert the text in the RDD to lowercase and uppercase. In the end, we used the **collect** operation to get the contents of the RDD objects.

Another popular transformation operation is **filter**, which can be used to filter out some entries of data. Some example code (**rddtransform2.py**) is shown next that is developed to filter all the even numbers from an RDD:

```

data = [5, 4, 6, 3, 2, 8, 9, 2, 8, 7,
 8, 4, 4, 8, 2, 7, 8, 9, 6, 9]
rdd1 = spark.sparkContext.parallelize(data)
rdd2 = rdd1.filter(lambda x: x % 2 !=0)
print(rdd2.collect())

```

When you execute this code, it will provide console output with 3, 7, 7, and 9 as collection entries. Next, we will explore a few action examples with PySpark.

## RDD action operations with PySpark

To illustrate the implementation of action operations, we will use an RDD created from the Python collection and then apply a few built-in action operations that come with the PySpark library. The sample code (**rddaction1.py**) is shown next:

```
data = [5, 4, 6, 3, 2, 8, 9, 2, 8, 7,
 8, 4, 4, 8, 2, 7, 8, 9, 6, 9]

rdd1 = spark.sparkContext.parallelize(data)

print("RDD contents with partitions:" + str(rdd1.glom().collect()))

print("Count by values: " + str(rdd1.countByValue()))

print("reduce function: " + str(rdd1.glom().collect()))

print("Sum of RDD contents:" + str(rdd1.sum()))

print("top: " + str(rdd1.top(5)))

print("count: " + str(rdd1.count()))

print("max: " + str(rdd1.max()))

print("min" + str(rdd1.min()))

time.sleep(60)
```

Some of the action operations used in this code example are self-explanatory and trivial (**count**, **max**, **min**, **count**, and **sum**). The rest of the action operations (non-trivial) are explained next:

- **glom**: This results in an RDD that is created by coalescing all data entries with each partition into a list.
- **collect**: This method returns all the elements of an RDD as a list.
- **reduce**: This is a generic function to apply to the RDD to reduce the number of elements in it. In our case, we used a lambda function to combine two elements into one, and so on. This results in adding all the elements in the RDD.
- **top(x)**: This action returns the top  $x$  elements in the array if the elements in the array are ordered.

We have covered how to create RDDs using PySpark and how to implement transformation and action operations on an RDD. In the next section, we will cover the PySpark DataFrame, which is another popular data structure used mainly for analytics.

## Learning about PySpark DataFrames

The **PySpark DataFrame** is a tabular data structure consisting of rows and columns, like the tables we have in a relational database and like the pandas DataFrame, which we introduced in [Chapter 6, Advanced Tips and Tricks in Python](#). In comparison to pandas DataFrames, the key difference is that PySpark DataFrame objects are distributed in the cluster, which means data is stored across different nodes in a cluster. The use of a DataFrame is mainly to process a large collection of structured or unstructured data, which may reach into the petabytes, in a distributed manner. Like RDDs, PySpark

DataFrames are immutable and based on lazy evaluation, which means evaluation will be delayed until it needs to be done.

We can store numeric as well as string data types in a DataFrame. The columns in a PySpark DataFrame cannot be empty; they must have the same data type and must be of the same length. Rows in a DataFrame can have data of different data types. Row names in a DataFrame are required to be unique.

In the next subsections, we will learn how to create a DataFrame and cover some key operations on DataFrames using PySpark.

## Creating a DataFrame object

A PySpark DataFrame can be created using one of the following sources of data:

- Python collections such as lists, tuples, and dictionaries.
- Files (CSV, XML, JSON, Parquet, and so on).
- RDDs, by using the **toDF** method or the **createDataFrame** method of PySpark.
- Apache Kafka streaming messages can be converted to PySpark DataFrames by using the **readStream** method of the **SparkSession** object.
- Database (for example, Hive and HBase) tables can be queried using traditional SQL commands and the output will be transformed into a PySpark DataFrame.

We will start creating a DataFrame from a Python collection, which is the simplest approach, but it is more helpful for illustration purposes. The next bit of sample code shows us how to create a PySpark DataFrame from a collection of employees data:

```
data = [('James','','Bylsma','HR','M',40000),
 ('Kamal','Rahim','','HR','M',41000),
 ('Robert','','Zaine','Finance','M',35000),
 ('Sophia','Anne','Richer','Finance','F',47000),
 ('John','Will','Brown','Engineering','F',65000)
]
columns = ["firstname","middlename","lastname",
 "department","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
print(df.printSchema())
print(df.show())
```

In this code example, we first created the row data as a list of employees and then created a schema with column names. When the schema is only a list of column names, the data type of each column is

determined by the data, and each column is marked as nullable by default. A more advanced API (**StructType** or **StructField**) can be used to define the DataFrame schema manually, which includes setting the data type and marking a column as nullable or not nullable. The console output of this sample code is shown next, which shows the schema first and then the DataFrame contents as a table:

```
root
|-- firstname: string (nullable = true)
|-- middlename: string (nullable = true)
|-- lastname: string (nullable = true)
|-- department: string (nullable = true)
|-- gender: string (nullable = true)
|-- salary: long (nullable = true)

+-----+-----+-----+-----+-----+
|firstname|middlename|lastname| department|gender|salary|
+-----+-----+-----+-----+-----+
| James| | Bylsma| HR| M| 40000|
| Kamal| Rahim| | HR| M| 41000|
| Robert| | Zaine| Finance| M| 35000|
| Sophia| Anne| Richer| Finance| F| 47000|
| John| Will| Brown|Engineering| F| 65000|
+-----+-----+-----+-----+-----+
```

In the next code example, we will create a DataFrame from a CSV file. The CSV file will have the same entries as we used in the previous code example. In this sample code (**dfcreate2.py**), we also defined the schema manually by using the **StructType** and **StructField** objects:

```
schemas = StructType([
 StructField("firstname", StringType(), True),
 StructField("middlename", StringType(), True),
 StructField("lastname", StringType(), True),
 StructField("department", StringType(), True),
 StructField("gender", StringType(), True),
 StructField("salary", IntegerType(), True)
])

df = spark.read.csv('df2.csv', header=True, schema=schemas)
print(df.printSchema())
print(df.show())
```

The console outcome of this code will be the same as shown for the previous code example. The importing of JSON, text, or XML files into a DataFrame is supported by the **read** method using a similar syntax. The support of other data sources, such as RDDs and databases, is left for you to evaluate and implement as an exercise.

## Working on a PySpark DataFrame

Once we have created a DataFrame from some data, regardless of the source of the data, we are ready to analyze it, transform it, and take some actions on it to get meaningful results from it. Most of the operations supported by the PySpark DataFrame are similar to RDDs and pandas DataFrames. For illustration purposes, we will load the same data as in the previous code example into a DataFrame object and then perform the following operations:

1. Select one or more columns from the DataFrame object using the **select** method.
2. Replace the values in a column using a dictionary and the **replace** method. There are more options to replace data in a column available in the PySpark library.
3. Add a new column with values based on an existing column's data.

The complete sample code (**dfoperations.py**) is shown next:

```
data = [('James','','Bylsma','HR','M',40000),
 ('Kamal','Rahim','','HR','M',41000),
 ('Robert','','Zaine','Finance','M',35000),
 ('Sophia','Anne','Richer','Finance','F',47000),
 ('John','Will','Brown','Engineering','F',65000)
]
columns = ["firstname","middlename","lastname",
 "department","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
#show two columns
print(df.select([df.firstname, df.salary]).show())
#replacing values of a column
myDict = {'F':'Female','M':'Male'}
df2 = df.replace(myDict, subset=['gender'])
#adding a new colum Pay Level based on an existing column values
df3 = df2.withColumn("Pay Level",
 when((df2.salary < 40000), lit("10")) \
 .when((df.salary >= 40000) & (df.salary <= 50000), lit("11")) \
 .otherwise(lit("12")) \
```

```

)
print(df3.show())

```

Following is the output for the preceding code example:

firstname	salary					
James	40000					
Kamal	41000					
Robert	35000					
Sophia	47000					
John	65000					
firstname	middlename	lastname	department	gender	salary	Pay Level
James		Bylsma	HR	Male	40000	11
Kamal	Rahim		HR	Male	41000	11
Robert		Zaine	Finance	Male	35000	10
Sophia	Anne	Richer	Finance	Female	47000	11
John	Will	Brown	Engineering	Female	65000	12

Figure 8.6 – Console output of the dfoperations.py program

The first table shows the result of the **select** operation. The next table shows the result of the **replace** operation on the **gender** column and also a new column, **Pay Level**.

There are many built-in operations available to work with PySpark DataFrames, and many of them are the same as we discussed for pandas DataFrames. The details of those operations can be explored by using the Apache Spark official documentation for the software release you have.

There is one legitimate question that anyone would ask at this point, which is, *Why should we use the PySpark DataFrame when we already have pandas DataFrame offering the same types of operations?* The answer is very simple. PySpark offers distributed DataFrames, and the operations on

such DataFrames are meant to be executed on a cluster of nodes in parallel. This makes the PySpark DataFrame's performance significantly better than the pandas DataFrame's.

We have seen so far that, as programmers, we are not actually having to program anything regarding how to delegate distributed RDDs and DataFrames to different executors in a standalone or distributed cluster. Our focus is only on the programming aspect of the data processing. Coordination and communication with a local or remote cluster of nodes is automatically taken care of by **SparkSession** and **SparkContext**. This is the beauty of Apache Spark and PySpark: letting programmers focus on solving the real problems instead of worrying about how workloads will be executed.

## Introducing PySpark SQL

Spark SQL is one of the key modules of Apache Spark; it is used for structured data processing and acts as a distributed SQL query engine. As you can imagine, Spark SQL is highly scalable, being a distributed processing engine. Usually, the data source for Spark SQL is a database, but SQL queries can be applied to temporary views, which can be built from RDDs and DataFrames.

To demonstrate using the PySpark library with Spark SQL, we will use the same DataFrame as in the previous sample code, using employees data to build a **TempView** instance for SQL queries. In our code example, we will do the following:

1. We will create a PySpark DataFrame for the employees data from a Python collection as we did for the previous code example.
2. We will create a **TempView** instance from the PySpark DataFrame using the **createOrReplaceTempView** method.
3. Using the **sql** method of the Spark Session object, we will execute the conventional SQL queries on the **TempView** instance, such as querying all employee records, querying employees with salaries higher than 45,000, querying the count of employees per gender type, and using the **group by** SQL command for the **gender** column.

The complete code example (**sql1.py**) is as follows:

```
data = [('James','','Bylsma','HR','M',40000),
 ('Kamal','Rahim','','HR','M',41000),
 ('Robert','','Zaine','Finance','M',35000),
 ('Sophia','Anne','Richer','Finance','F',47000),
 ('John','Will','Brown','Engineering','F',65000)
]
columns = ["firstname","middlename","lastname",
 "department","gender","salary"]
df = spark.createDataFrame(data=data, schema = columns)
```

```

df.createOrReplaceTempView("EMP_DATA")

df2 = spark.sql("SELECT * FROM EMP_DATA")
print(df2.show())

df3 = spark.sql("SELECT firstname,middlename,lastname, salary FROM EMP_DATA WHERE
SALARY > 45000")
print(df3.show())

df4 = spark.sql(("SELECT gender, count(*) from EMP_DATA group by gender"))
print(df4.show())

```

The console output will show the results of the three SQL queries:

```

+-----+-----+-----+-----+-----+
|firstname|middlename|lastname| department|gender|salary|
+-----+-----+-----+-----+-----+
| James| | Bylsma| HR| M| 40000|
| Kamal| Rahim| | HR| M| 41000|
| Robert| | Zaine| Finance| M| 35000|
| Sophia| Anne| Richer| Finance| F| 47000|
| John| Will| Brown|Engineering| F| 65000|
+-----+-----+-----+-----+-----+
+-----+-----+-----+
|firstname|middlename|lastname|salary|
+-----+-----+-----+
| Sophia| Anne| Richer| 47000|
| John| Will| Brown| 65000|
+-----+-----+-----+
+-----+
|gender|count(1)|
+-----+
| F| 2|
| M| 3|
+-----+

```

Spark SQL is a big topic within Apache Spark. We provided only an introduction to Spark SQL to show the power of using SQL commands on top of Spark data structures without knowing the source of the data. This concludes our discussion of the use of PySpark for data processing and data

analysis. In the next section, we will discuss a couple of case studies to build some real-world applications.

## Case studies of using Apache Spark and PySpark

In previous sections, we covered the fundamental concepts and architecture of Apache Spark and PySpark. In this section, we will discuss two case studies for implementing two interesting and popular applications for Apache Spark.

### Case study 1 – Pi ( $\pi$ ) calculator on Apache Spark

We will calculate Pi ( $\pi$ ) using the Apache Spark cluster that is running on our local machine. Pi is the area of a circle when its radius is 1. Before discussing the algorithm and the driver program for this application, it is important to introduce the Apache Spark setup used for this case study.

#### Setting up the Apache Spark cluster

In all previous code examples, we used PySpark locally installed on our machine without a cluster. For this case study, we will set up an Apache Spark cluster by using multiple virtual machines. There are many virtualization software tools available, such as **VirtualBox**, and any of these software tools will work for building this kind of setup.

We used Ubuntu **Multipass** (<https://multipass.run/>) to build the virtual machines on top of macOS. Multipass works on Linux and on Windows as well. Multipass is a lightweight virtualization manager and is designed specifically for developers to create virtual machines with a single command. Multipass has very few commands, which makes it easier to use. If you decide to use Multipass, we recommend that you use the official documentation for installation and configuration. In our virtual machines setup, we have the following virtual machines created using Multipass:

Name	State	IPv4	Image
vm1	Running	192.168.64.2	Ubuntu 20.04 LTS
vm2	Running	192.168.64.3	Ubuntu 20.04 LTS
vm3	Running	192.168.64.4	Ubuntu 20.04 LTS

Figure 8.7 – Virtual machines created for our Apache Spark cluster

We installed *Apache Spark 3.1.1* on each virtual machine by using the **apt-get** utility. We started Apache Spark as the master on **vm1** and then started Apache Spark as the worker on **vm2** and **vm3** by providing the master Spark URI, which is **Spark://192.168.64.2.7077** in our case. The complete Spark cluster setup will look as shown here:

Node Name	Role	Web UI
192.168.64.2	Master (Spark://192.168.64.2:7077)	http://192.168.64.2:8080/
192.168.64.3	Worker	http://192.168.64.3:8081/
192.168.64.4	Worker	http://192.168.64.4:8081/

Figure 8.8 – Details of the Apache Spark cluster nodes

The web UI for the master Spark node will look as shown here:

The screenshot shows the Apache Spark 3.1.1 Master Web UI. At the top, it displays "Spark Master at spark://192.168.64.2:7077". Below this, various cluster statistics are listed: URL (spark://192.168.64.2:7077), Alive Workers (2), Cores in use (2 Total, 0 Used), Memory in use (2.0 GiB Total, 0.0 B Used), Resources in use, Applications (0 Running, 0 Completed), Drivers (0 Running, 0 Completed), and Status (ALIVE). A section titled "Workers (2)" lists two workers: worker-20210529145544-192.168.64.3-43027 and worker-20210529150201-192.168.64.4-34453, both marked as ALIVE with 1 core and 1024.0 MiB memory. Below the worker list are sections for "Running Applications (0)" and "Completed Applications (0)".

Worker Id	Address	State	Cores	Memory	Resources
worker-20210529145544-192.168.64.3-43027	192.168.64.3:43027	ALIVE	1 (0 Used)	1024.0 MiB (0.0 B Used)	
worker-20210529150201-192.168.64.4-34453	192.168.64.4:34453	ALIVE	1 (0 Used)	1024.0 MiB (0.0 B Used)	

Figure 8.9 – Web UI for the master node in the Apache Spark cluster

A summary of the web UI for the master node is given here:

- The web UI provides the node name with the Spark URL. In our case, we used the IP address as the host name, which is why we have an IP address in the URL.
- There are the details of the worker nodes, of which there are two in our case. Each worker node uses 1 CPU core and 1 GB of memory.
- The web UI also provides details of the running and completed applications.

The web UI for the worker nodes will look as follows:



# Spark Worker at 192.168.64.3:43027

**ID:** worker-20210529145544-192.168.64.3-43027

**Master URL:** spark://192.168.64.2:7077

**Cores:** 1 (0 Used)

**Memory:** 1024.0 MiB (0.0 B Used)

**Resources:**

[Back to Master](#)

## ▼ Running Executors (0)

ExecutorID	State	Cores	Memory	Resources	Job Details	Logs
------------	-------	-------	--------	-----------	-------------	------

## ► Finished Executors (13)

Figure 8.10 – Web UI for the worker nodes in the Apache Spark cluster

A summary of the web UI for the worker nodes is given here:

- The web UI provides the worker IDs as well as the node names and the ports where the workers are listening for requests.
- The master node URL is also provided in the web UI.
- Details of the CPU core and memory allocated to the worker nodes are also available.
- The web UI provides details of jobs in progress (**Running Executors**) and jobs that are already finished.

## Writing a driver program for Pi calculation

To calculate Pi, we are using a commonly used algorithm (the **Monte Carlo** algorithm) that assumes a square having an area equal to 4 that is circumscribing a unit circle (circle with a radius value equal to 1). The idea is to generate a huge amount of random numbers in the domain of a square with sides having a length of 2. We can assume there is a circle inside the square with the same diameter value as the length of the side of the square. This means that the circle will be inscribed inside the square. The value of Pi is estimated by calculating the ratio of the number of points that lie inside the circle to the total number of generated points.

The complete sample code for the driver program is shown next. In this program, we decided to use two partitions as we have two workers available to us. We used 10,000,000 points for each worker. Another important thing to note is that we used the Spark master node URL as a master attribute when creating the Apache Spark session:

```
#casestudy1.py: Pi calculator
```

```
from operator import add
from random import random
from pyspark.sql import SparkSession
spark = SparkSession.builder.master
 ("spark://192.168.64.2:7077") \
 .appName("Pi calculator app") \
 .getOrCreate()
partitions = 2
n = 10000000 * partitions
def func(_):
 x = random() * 2 - 1
 y = random() * 2 - 1
 return 1 if x ** 2 + y ** 2 <= 1 else 0
count = spark.sparkContext.parallelize(range(1, n + 1),
 partitions).map(func).reduce(add)
print("Pi is roughly %f" % (4.0 * count / n))
```

The console output is as follows:

```
Pi is roughly 3.141479
```

The Spark web UI will provide the status of the application when running and even after it completes its execution. In the following screenshot, we can see that two workers were engaged to complete the job:

## APACHE Spark 3.1.1 Spark Master at spark://192.168.64.2:7077

**URL:** spark://192.168.64.2:7077  
**Alive Workers:** 2  
**Cores in use:** 2 Total, 2 Used  
**Memory in use:** 2.0 GiB Total, 2.0 GiB Used  
**Resources in use:**  
**Applications:** 1 [Running](#), 0 [Completed](#)  
**Drivers:** 0 Running, 0 Completed  
**Status:** ALIVE

### ▼ Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20210529145544-192.168.64.3-43027	192.168.64.3:43027	ALIVE	1 (1 Used)	1024.0 MiB (1024.0 MiB Used)	
worker-20210529150201-192.168.64.4-34453	192.168.64.4:34453	ALIVE	1 (1 Used)	1024.0 MiB (1024.0 MiB Used)	

### ▼ Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20210529191451-0000 (kill)	Pi claculator app	2	1024.0 MiB		2021/05/29 19:14:51	muasif	RUNNING	9 s

Figure 8.11 – Pi calculator status in the Spark web UI

We can click on the application name to go to the next level of detail for the application, as shown in *Figure 8.12*. This screenshot shows which workers are involved in completing the tasks and what resources are being used (if things are still running):

## APACHE Spark 3.1.1 Application: Pi claculator app

**ID:** app-20210529191451-0000  
**Name:** Pi claculator app  
**User:** muasif  
**Cores:** Unlimited (2 granted)  
**Executor Limit:** Unlimited (2 granted)  
**Executor Memory:** 1024.0 MiB  
**Executor Resources:**  
**Submit Date:** 2021/05/29 19:14:51  
**State:** FINISHED

### ▼ Executor Summary (2)

ExecutorID	Worker	Cores	Memory	Resources	State	Logs
------------	--------	-------	--------	-----------	-------	------

### ▼ Removed Executors (2)

ExecutorID	Worker	Cores	Memory	Resources	State	Logs
1	worker-20210529150201-192.168.64.4-34453	1	1024		KILLED	stdout stderr
0	worker-20210529145544-192.168.64.3-43027	1	1024		KILLED	stdout stderr

Figure 8.12 – Pi calculator application executor-level details

In this case study, we covered how we can set up an Apache Spark cluster for testing and experimentation purposes and how we can build a driver program in Python using the PySpark library to connect to Apache Spark and submit our jobs to be processed on two different cluster nodes.

In the next case study, we will build a word cloud using the PySpark library.

## Case study 2 – Word cloud using PySpark

A **word cloud** is a visual representation of the frequency of words that appear in some text data. Put simply, if a specific word appears more frequently in a text, it appears bigger and bolder in the word cloud. These are also known as **tag clouds** or **text clouds** and are very useful tools to identify what parts of some textual data are more important. A practical use case of this tool is the analysis of content on social media, which has many applications for marketing, business analytics, and security.

For illustration purposes, we have built a simple word cloud application that reads a text file from the local filesystem. The text file is imported into an RDD object that is then processed to count the number of times each word occurred. We process the data further to filter out the words that are repeated fewer than two times and also filter out words that are of a length that's less than four letters. The word frequency data is fed to the **WordCloud** library object. To display the word cloud, we used the **matplotlib** library. The complete sample code is shown next:

```
#casestudy2.py: word count application
import matplotlib.pyplot as plt
from pyspark.sql import SparkSession
from wordcloud import WordCloud
spark = SparkSession.builder.master("local[*]")\
 .appName("word cloud app")\
 .getOrCreate()
wc_threshold = 1
wl_threshold = 3
textRDD = spark.sparkContext.textFile('wordcloud.txt',3)
flatRDD = textRDD.flatMap(lambda x: x.split(' '))
wCRDD = flatRDD.map(lambda word: (word, 1))\
 .reduceByKey(lambda v1, v2: v1 + v2)
filter out words with fewer than threshold occurrences
filteredRDD = wCRDD.filter(lambda pair: pair[1] >= wc_threshold)
```

```
filteredRDD2 = filteredRDD.filter(lambda pair: len(pair[0]) > wl_threshold)
word_freq = dict(filteredRDD2.collect())
Create the wordcloud object
wordcloud = WordCloud(width=480, height=480, margin=0).\
 generate_from_frequencies(word_freq)
Display the generated cloud image
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.margins(x=0, y=0)
plt.show()
```

The output of this program is plotted as a window application and the output will look as shown here, based on the sample text (**wordcloud.txt**) provided to the application:



## Summary

In this chapter, we explored how to execute data-intensive jobs on a cluster of machines to achieve parallel processing. Parallel processing is important for large-scale data, which is also known as big data. We started by evaluating the different cluster options available for data processing. We provided a comparative analysis of Hadoop MapReduce and Apache Spark, which are the two main competing platforms for clusters. The analysis showed that Apache Spark has more flexibility in terms of supported languages and cluster management systems, and it outperforms Hadoop MapReduce for real-time data processing because of its in-memory data processing model.

Once we had established that Apache Spark is the most appropriate choice for a variety of data processing applications, we started looking into its fundamental data structure, which is the RDD. We discussed how to create RDDs from different sources of data and introduced two types of operations, transformations and actions.

In the core part of this chapter, we explored using PySpark to create and manage RDDs using Python. This included several code examples of transformation and action operations. We also introduced PySpark DataFrames for the next level of data processing in a distributed manner. We concluded the topic by introducing PySpark SQL with a few code examples.

Finally, we looked at two case studies using Apache Spark and PySpark. These case studies included the calculation of Pi and the building of a word cloud from text data. We also covered in the case studies how we can set up a Standalone Apache Spark instance on a local machine for testing purposes.

This chapter gave you a lot of experience in setting up Apache Spark locally as well as setting up Apache Spark clusters using virtualization. There are plenty of code examples provided in this chapter for you to enhance your practical skills. This is important for anyone who wants to process their big data problems using clusters for efficiency and scale.

In the next chapter, we will explore options for leveraging frameworks such as Apache Beam and extend our discussion of using public clouds for data processing.

## Questions

1. How is Apache Spark different from Hadoop MapReduce?
2. How are transformations different from actions in Apache Spark?
3. What is lazy evaluation in Apache Spark?
4. What is **SparkSession**?
5. How is the PySpark DataFrame different from the pandas DataFrame?

## Further reading

- *Spark in Action, Second Edition* by Jean-Georges Perrin
- *Learning PySpark* by Tomasz Drabas, Denny Lee
- *PySpark Recipes* by Raju Kumar Mishra
- *Apache Spark documentation* for the release you are using (<https://spark.apache.org/docs/rel#>)
- *Multipass documentation* available at <https://multipass.run/docs>

## Answers

1. Apache Spark is an in-memory data processing engine, whereas Hadoop MapReduce has to read from and write to the filesystem.
2. Transformation is applied to convert or translate data from one form to another, and the results stay within the cluster. Actions are the functions applied to data to get the results that are returned to the driver program.
3. Lazy evaluation is applied mainly for transformation operations, which means transformation operations are not executed until an action is triggered on a data object.
4. **SparkSession** is an entry point to the Spark application to connect to one or more cluster managers and to work with executors for task execution.
5. The PySpark DataFrame is distributed and is meant to be available on multiple nodes of an Apache Spark cluster for parallel processing.

[OceanofPDF.com](https://OceanofPDF.com)

## *Chapter 9: Python Programming for the Cloud*

Cloud computing is a broad term that is used for a wide variety of use cases. These use cases include an offering of physical or virtual compute platforms, software development platforms, big data processing platforms, storage, network functions, software services, and many more. In this chapter, we will explore Python for cloud computing from two correlated aspects. First, we will investigate the options of using Python for building applications for cloud runtimes. Then, we will extend our discussion of data-intensive processing, which we started in *Chapter 8, Scaling Out Python using Clusters*, from clusters to cloud environments. The focus of the discussion in this chapter will largely center on the three public cloud platforms; that is, **Google Cloud Platform (GCP)**, **Amazon Web Services (AWS)**, and **Microsoft Azure**.

We will cover the following topics in this chapter:

- Learning about the cloud options for Python applications
- Building Python web services for cloud deployment
- Using Google Cloud Platform for data processing

By the end of this chapter, you will know how to develop and deploy applications to a cloud platform and how to use Apache Beam in general and for Google Cloud Platform.

## Technical requirements

The following are the technical requirements for this chapter:

- You need to have Python 3.7 or later installed on your computer.
- You will need a service account for Google Cloud Platform. A free account will work fine.
- You will need the Google Cloud SDK installed on your computer.
- You will need Apache Beam installed on your computer.

The sample code for this chapter can be found at <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter09>.

We will start our discussion by looking at the cloud options that are available for developing applications for cloud deployments.

## Learning about the cloud options for Python applications

Cloud computing is the ultimate frontier for programmers these days. In this section, we will investigate how we can develop Python applications using a cloud development environment or using

a specific **Software Development Kit (SDK)** for cloud deployment, and then how we can execute the Python code in a cloud environment. We will also investigate options regarding data-intensive processing, such as Apache Spark on the cloud. We will start with the cloud-based development environments.

## Introducing Python development environments for the cloud

When it comes to setting up a Python development environment for one of the three main public clouds, two types of models are available:

- Cloud-native **Integrated Development Environment (IDE)**
- Locally installed IDE with an integration option for the cloud

We will discuss these two modes next.

### Cloud-native IDE

There are several cloud-native development environments available in general that are not specifically attached to the three public cloud providers. These include **PythonAnyWhere**, **Repl.it**, **Trinket**, and **Codeanywhere**. Most of these cloud environments offer a free license in addition to a paid one. These public cloud platforms offer a mixture of tools for development environments, as explained here:

- **AWS:** This offers a sophisticated cloud IDE in the form of **AWS Cloud9**, which can be accessed through a web browser. This cloud IDE has a rich set of features for developers and the option of supporting several programming languages, including Python. It is important to understand that AWS Cloud9 is offered as an application hosted on an Amazon EC2 instance (a virtual machine). There is no direct fee for using AWS Cloud9, but there will be a fee for using the underlying Amazon EC2 instance and storage space, which is very nominal for limited use. The AWS platform also offers tools for building and testing the code for **continuous integration (CI)** and **continuous delivery (CD)** goals.

**AWS CodeBuild** is another service that's available that compiles our source code, runs tests, and builds software packages for deployment. It is a build server similar to Bamboo. **AWS CodeStar** is commonly used with AWS Cloud9 and offers a projects-based platform to help develop, build, and deploy software. AWS CodeStar offers predefined project templates to define an entire continuous delivery toolchain until the code is released.

- **Microsoft Azure:** This comes with the **Visual Studio** IDE, which is available online (cloud-based) if you are part of the Azure DevOps platform. Online access to the Visual Studio IDE is based on a paid subscription. Visual Studio IDE is well-known for its rich features and capabilities for offering an environment for team-level collaboration. Microsoft Azure offers **Azure Pipelines** for building, testing, and deploying your code to any platform such as Azure, AWS, and GCP. Azure Pipelines support many languages, such as Node.js, Python, Java, PHP, Ruby, C/C++, and .NET, and even mobile development toolkits.
- **Google:** Google offers **Cloud Code** to write, test, and deploy the code that can be written either through your browser (such as via ASW Cloud9) or using a local IDE of your choice. Cloud Code comes with plugins for the most popular IDEs, such as IntelliJ IDE, Visual Studio Code, and JetBrains PyCharm. Google Cloud Code is available free of charge and is targeted at container

runtime environments. Like AWS CodeBuild and Azure Pipelines, Google offers an equivalent service that is also called **Cloud Build** for the continuous building, testing, and deploying of software in multiple environments, such as virtual machines and containers. Google also offers Google **Colaboratory** or **Google Colab** that offers Jupyter Notebooks remotely. The Google Colab option is popular among data scientists.

Google Cloud also offers **Tekton** and the **Jenkins** service for building CI/CD development and delivery models.

In addition to all these dedicated tools and services, these cloud platforms offer online as well as locally installed shell environments. These shell environments are also a quick way to manage code in a limited capacity.

Next, we will discuss the local IDE options for using Python for the cloud.

### **Local IDE for cloud development**

The cloud-native development environment is a great tool for having native integration options in the rest of your cloud ecosystem. This makes instantiating on-demand resources and then deploying them convenient, and doesn't require any authentication tokens. But this comes with some caveats. First, although the tools are mostly free, the underlying resources that they are using are not. The second caveat is that the offline availability of these cloud-native tools is not seamless. Developers like to write code without any online ties so that they can do this anywhere, such as on a train or in a park.

Due to these caveats, developers like to use local editors or IDEs for developing and testing the software before using additional tools to deploy on one of the cloud platforms. Microsoft Azure IDEs such as Visual Studio and Visual Studio Code are available for local machines. AWS and Google platform offer their own SDKs (shell-like environments) and plugins to be integrated with your IDE of choice. We will explore these models of development later in this chapter.

Next, we will discuss the runtime environments that are available on the public clouds.

## **Introducing cloud runtime options for Python**

The simplest way to get a Python runtime environment is to get a Linux virtual machine or a container with Python installed. Once we have a virtual machine or a container, we can also install the Python version of our choice. For data-intensive workloads, the Apache Spark cluster can be set up on the compute nodes of the cloud. But this requires us to own all platform-related tasks and maintenance in case anything goes wrong. Almost all public cloud platforms offer more elegant solutions to simplify developers' and IT administrators' lives. These cloud providers offer one or more pre-built runtime environments based on the application types. We will discuss a few of the runtime environments that are available from the three public cloud providers – Amazon AWS, GCP, and Microsoft Azure.

## **WHAT IS A RUNTIME ENVIRONMENT?**

A *runtime environment* is an execution platform that runs Python code.

## **Runtime options offered by Amazon AWS**

Amazon AWS offers the following runtime options:

- **AWS Beanstalk:** This **Platform-as-a-Service (PaaS)** offering can be used to deploy web applications that have been developed using Java, .NET, PHP, Node.js, Python, and many more. This service also offers the option of using Apache, Nginx, Passenger, or IIS as a web server. This service provides flexibility in managing the underlining infrastructure, which is sometimes required for deploying complex applications.
- **AWS App Runner:** This service can be used to run containerized web applications and microservices with an API. This service is fully managed, which means you have no administrative responsibilities, as well as no access to the underlying infrastructure.
- **AWS Lambda:** This is a serverless compute runtime that allows you to run your code without the worry of managing any underlying servers. This server supports multiple languages, including Python. Although Lambda code can be executed directly from an application, this is well-suited for cases when we must run a certain piece of code in case an event is triggered by the other AWS services.
- **AWS Batch:** This option is used to run computing jobs in large volumes in the form of batches. This is a cloud option from Amazon that's an alternative to the Apache Spark and Hadoop MapReduce cluster options.
- **AWS Kinesis:** This service is also for data processing, but for real-time streaming data.

## **Runtime options offered by GCP**

The following are the runtime options that are available from GCP:

- **App Engine:** This is a PaaS option from GCP that can be used to develop and host web applications at scale. The applications are deployed as containers on App Engine, but your source code is packed into a container by the deployment tool. This complexity is hidden from developers.
- **CloudRun:** This option is used to host any code that has been built as a container. The container applications must have HTTP endpoints to be deployed on CloudRun. In comparison to App Engine, packaging applications to a container is the developer's responsibility.
- **Cloud Function:** This is an event-driven, serverless, and single-purpose solution for hosting lightweight Python code. The hosted code is typically triggered by listening to events on other GCP services or via direct HTTP requests. This is comparable to the AWS Lambda service.
- **Dataflow:** This is another serverless option but mainly for data processing with minimal latency. This simplifies a data scientist's life by taking away the complexity of the underlying processing platform and offering a data pipeline model based on Apache Beam.
- **Dataproc:** This service offers a computer platform based on Apache Spark, Apache Flink, Presto, and many more tools. This platform is suitable for those who have data processing jobs with dependencies on a Spark or Hadoop ecosystem. This service requires that we manually provision clusters.

## **Runtime options offered by Microsoft Azure**

Microsoft Azure offers the following runtime environments:

- **App Service:** This service is used to build and deploy web apps at scale. This web application can be deployed as a container or run on Windows or Linux.

- **Azure Functions:** This is a serverless event-driven runtime environment that's used to execute code based on a certain event or direct request. This is comparable to AWS Lambda and GCP CloudRun.
- **Batch:** As its name suggests, this service is used to run cloud-scale jobs that require hundreds or thousands of virtual machines.
- **Azure Databricks:** Microsoft has partnered with Databricks to offer this Apache Spark-based platform for large-scale data processing.
- **Azure Data Factory:** This is a serverless option from Azure that you can use to process streaming data and transform the data into meaningful outcomes.

As we have seen, the three main cloud providers offer a variety of execution environments based on the applications and workloads that are available. The following use cases can be deployed on cloud platforms:

- Developing web services and web applications
- Data processing using a cloud runtime
- Microservice-based applications (containers) using Python
- Serverless functions or applications for the cloud

We will address the first two use cases in the upcoming sections of this chapter. The remaining use cases will be discussed in the upcoming chapters as they require more extensive discussion. In the next section, we will start building a web service using Python and explore how to deploy it on the GCP App Engine runtime environment.

## Building Python web services for cloud deployment

Building an application for cloud deployment is slightly different than doing so for a local deployment. There are three key requirements we must consider while developing and deploying an application to any cloud. These requirements are as follows:

- **Web interface:** For most cloud deployments, applications that have a **graphical user interface (GUI)** or **application programming interface (API)** are the main candidates. Command-line interface-based applications will not get their usability from a cloud environment unless they are deployed in a dedicated virtual machine instance, and we can execute them on a VM instance using SSH or Telnet. This is why we selected a web interface-based application for our discussion.
- **Environment setup:** All public cloud platforms support multiple languages, as well as different versions of a single language. For example, GCP App Engine supports Python versions 3.7, 3.8, and 3.9 as of June 2021. Sometimes, cloud services allow you to bring your own version for deployment as well. For web applications, it is also important to set an entry point for accessing the code and project-level settings. These are typically defined in a single file (a YAML file, in the case of the GCP App Engine application).
- **Dependency management:** The main challenge regarding the portability of any application is its dependency on third-party libraries. For GCP App Engine applications, we document all dependencies in a text file (**requirements.txt**) manually or using the **PIP freeze** command. There are other elegant ways available to solve this problem as well. One such way is to package all third-party libraries with applications into a single file for cloud deployment, such as the Java web archive file (**.war** file). Another approach is to bundle all the dependencies containing application code and the target execution platform into a container and

deploy the container directly on a container hosting platform. We will explore container-based deployment options in [Chapter 11](#), *Using Python for Microservices Development*.

There are at least three options for deploying a Python web service application on GCP App Engine, which are as follows:

- Using the Google Cloud SDK via the CLI interface
- Using the GCP web console (portal) along with Cloud Shell (CLI interface)
- Using a third-party IDE such as PyCharm

We will discuss the first option in detail and provide a summary of our experience with the other two options.

### ***IMPORTANT NOTE***

*To deploy a Python application in AWS and Azure, the procedural steps are the same in principle, but the details are different, depending on the SDK and API support available from each cloud provider.*

## **Using Google Cloud SDK**

In this section, we will discuss how to use Google Cloud SDK (mainly the CLI interface) to create and deploy a sample application. This sample application will be deployed on the **Google App Engine (GAE)** platform. GAE is a PaaS platform and is best suited for deploying web applications using a wide variety of programming languages, including Python.

To use the Google Cloud SDK for Python application deployment, we must have the following prerequisites on our local machine:

- Install and initialize the Cloud SDK. Once installed, you can access it via the CLI interface and check its version with the following command. Note that almost all Cloud SDK commands start with **gcloud**:  
**gcloud --version**
- Install the Cloud SDK components to add the App Engine extension for Python 3. This can be done by using the following command:  
**gcloud components install app-engine-python**
- The GCP CloudBuild API must be enabled for the GCP cloud project.
- Cloud billing must be enabled for the GCP cloud project, even if you are using a trial account, by associating your GCP billing account with the project.
- The GCP user privileges to set up a new App Engine application and to enable API services should be done at the *Owner* level.

Next, we will describe how to set up a GCP cloud project, create a sample web service application, and deploy it to the GAE.

### **Setting up a GCP cloud project**

The concept of a GCP cloud project is the same as we see in most development IDEs. A GCP cloud project consists of a set of project-level settings that manage how our code interacts with GCP services and tracks the resources in use by the project. A GCP project must be associated with a billing account. This is a prerequisite, in terms of billing, for tracking how many GCP services and resources are consumed on a per-project basis.

Next, we will explain how to set up a project using Cloud SDK:

1. Log in to the Cloud SDK using the following command. This will take you to the web browser so that you can sign in, in case you have not already done so:

```
gcloud init
```

2. Create a new project called **time-wsproj**. The project's name should be short and use only letters and numbers. The use of - is allowed for better readability:

```
gcloud projects create time-wsproj
```

3. Switch your default scope of the Cloud SDK to the newly created project, if you haven't done so already, by using the following command:

```
gcloud config set project time-wsproj
```

This will enable Cloud SDK to use this project as a default project for any command we push through the Cloud SDK CLI.

4. Create an App Engine instance under a default project or for any project by using the **project** attribute with one of the following commands:

```
gcloud app create #for default project
```

```
gcloud app create --project=time-wsproj #for specific project
```

Note that this command will reserve cloud resources (mainly compute and storage) and will prompt you to select a region and zone to host the resources. You can select the region and zone that's closest to you and also more appropriate from your audience's point of view.

5. Enable the Cloud Build API service for the current project. As we've discussed already, the Google Cloud Build service is used to build the application before it's deployed to a Google runtime such as App Engine. The Cloud Build API service is easier to enable through the GCP web console as it only takes a few clicks. To enable it using Cloud SDK, first, we need to know the exact name of the service. We can get the list of available GCP services by using the **gcloud services list** command.

This command will give you a long list of GCP services so that you can look for a service related to Cloud Build. You can also use **format**, attributed with any command, to beautify the Cloud SDK's output. To make this even more convenient, you can use the Linux **grep** utility (if you are using Linux or macOS) with this command to filter the results and then enable the service using the **enable** command:

```
gcloud services list --available | grep cloudbuild
#output will be like: NAME: cloudbuild.googleapis.com
```

```
#Cloud SDK command to enable this service
gcloud services enable cloudbuild.googleapis.com
```

6. To enable the Cloud Billing API service for our project, first, we need to associate a billing account with our project. Support for billing accounts in Cloud SDK hasn't been achieved with **General Availability (GA)** yet, as per Cloud SDK release 343.0.0. Attaching a billing account to a project can be done through the GCP web console. But there is also a beta version of the Cloud SDK commands available so that you can achieve the same. As a first step, we need to know the billing account ID that will be used. The billing accounts associated with the logged-in user can be retrieved by using the **beta** command presented here:

```
gcloud beta billing accounts list

#output will include following

#billingAccounts/0140E8-51G144-2AB62E

#enable billing on the current project using
gcloud beta billing projects link time-wsproj --billing-account 0140E8-51G144-2AB62E
```

Note that if you are using the **beta** commands for the first time, you will be prompted to install the beta component. You should go ahead and install it. If you are already using a Cloud SDK version with a billing component included for GA, you can skip using the beta keyword or use the appropriate commands, as per the Cloud SDK release documentation.

7. Enable the Cloud Billing API service for the current project by following the same steps we followed for enabling the Cloud Build API. First, we must find the name of the API service and then enable it using the following set of Cloud SDK commands:

```
gcloud services list --available | grep cloudbilling

#output will be: NAME: cloudbilling.googleapis.com

#command to enable this service
gcloud services enable cloudbilling.googleapis.com
```

The steps you must follow to set up a cloud project are straightforward for an experienced cloud user and will not take more than a few minutes. Once the project has been set up, we can get the project configuration details by running the following command:

```
gcloud projects describe time-wsproj
```

The output of this command will provide the project life cycle's status, the project's name, the project's ID, and the project's number. The following is some example output:

```
createTime: '2021-06-05T12:03:31.039Z'
lifecycleState: ACTIVE
name: time-wsproj
projectId: time-wsproj
projectNumber: '539807460484'
```

Now that the project has been set up, we can start developing our Python web application. We will do this in the next section.

## Building a Python application

For cloud deployments, we can build a Python application using an IDE or system editor and then emulate the App Engine runtime locally using the Cloud SDK and the *app-engine-python component*, which we have installed as a prerequisite. As an example, we will build a web service-based application that will provide us with the date and time through a REST API. The application can be triggered via an API client or using a web browser. We did not enable any authentication to keep the deployment simple.

To build the Python application, we will set up a Python virtual environment using the Python **venv** package. A virtual environment, created using the **venv** package, will be used to wrap the Python interpreter, core, and third-party libraries and scripts to keep them separate from the system Python environment and other Python virtual environments. Creating and managing a virtual environment in Python using the **venv** package has been supported in Python since v3.3. There are other tools available for creating virtual environments, such as **virtualenv** and **pipenv**. PyPA recommends using **venv** for creating a virtual environment, so we selected it for most of the examples presented in this book.

As a first step, we will create a web application project directory named **time-wsproj** that contains the following files:

- **app.yaml**
- **main.py**
- **requirements.txt**

We used the same name for the directory that we used to create the cloud project just for convenience, but this is not a requirement. Let's look at these files in more detail.

### YAML file

This file contains the deployment and runtime settings for an App Engine application, such as runtime version number. For Python 3, the **app.yaml** file must have at least a runtime parameter (**runtime: python38**). Each service in the web application can have its own YAML file. For the sake of simplicity, we will use only one YAML file. In our case, this YAML file will only contain the runtime attribute. We added a few more attributes to the sample YAML file for illustration purposes:

```
runtime: python38
```

## main.py Python file

We selected the **Flask** library to build our sample application. **Flask** is a well-known library for web development, mainly because of the powerful features it offers, along with its ease of use. We will cover Flask in the next chapter in detail.

This **main.py** Python module is the entry point of our application. The complete code of the application is presented here:

```
from flask import Flask
from datetime import date, datetime
If 'entrypoint' is not defined in app.yaml, App Engine will look #for an app variable.
This is the case in our YAML file
app = Flask(__name__)
@app.route('/')
def welcome():
 return 'Welcome Python Geek! Use appropriate URI for date and time'
@app.route('/date')
def today():
 today = date.today()
 return "{date:" + today.strftime("%B %d, %Y") + '}'"
@app.route('/time')
def time():
 now = datetime.now()
 return "{time:" + now.strftime("%H:%M:%S") + '}'"
if __name__ == '__main__':
 # For local testing
 app.run(host='127.0.0.1', port=8080, debug=True)
```

This module provides the following key features:

- There is a default endpoint called **app** that's defined in this module. The **app** variable is used to redirect the requests that are sent to this module.
- Using Flask's annotation, we have defined handlers for three URLs:
  - a) The root / URL will trigger a function named **welcome**. The **welcome** function returns a greeting message as a string.
  - b) The **/date** URL will trigger the **today** function, which will return today's date in JSON format.

c) The **/time** URL will execute the **time** function, which will return the current time in JSON format.

- At the end of the module, we added a **\_\_main\_\_** function to initiate a local web server that comes with Flask for testing purposes.

## Requirements file

This file contains a list of project dependencies for third-party libraries. The contents of this file will be used by App Engine to make the required libraries available to our application. In our case, we will need the Flask library to build our sample web application. The contents of this file for our project are as follows:

```
Flask==2.0.1
```

Once we have created the project directory and made these files, we must create a virtual environment inside or outside the project directory and activate it using the source command:

```
python -m venv myenv
source myenv/bin/activate
```

After activating the virtual environment, we must install the necessary dependencies, as per the **requirements.txt** file. We will use the **pip** utility from the same directory where the **requirements.txt** file resides:

```
pip install -r requirements.txt
```

Once the Flask library and its dependencies have been installed, the directory structure will look like this in our PyCharm IDE:

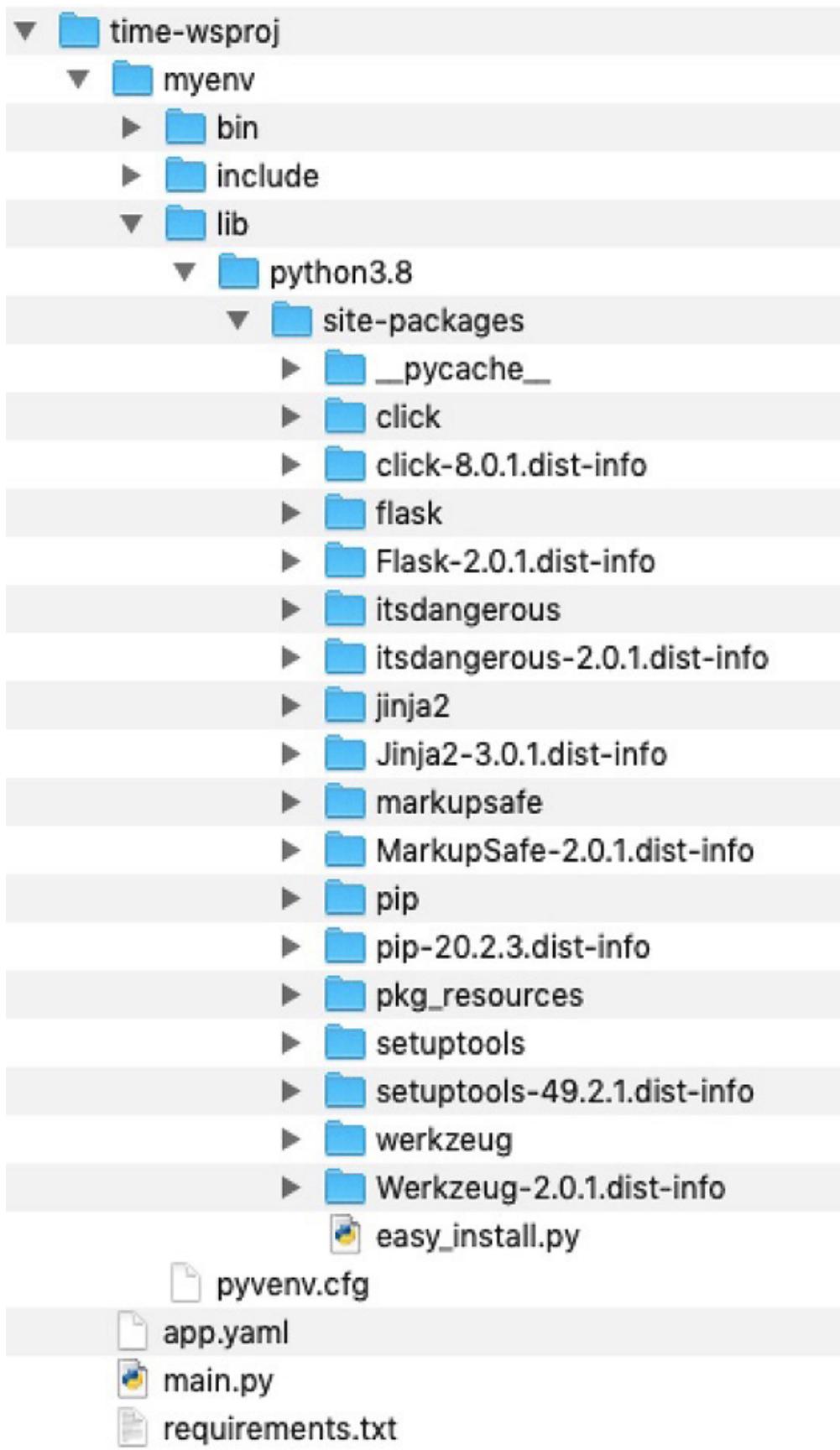


Figure 9.1 – Directory structure for a sample web application

Once the project file and dependencies have been set up, we will start the web server locally using the following command:

```
python main.py
```

The server will start with the following debug messages, which makes it clear that this server option is only for testing purposes and not for production environments:

```
* Serving Flask app 'main' (lazy loading)
* Environment: production
 WARNING: This is a development server. Do not use it in a production deployment.
 Use a production WSGI server instead.

* Debug mode: on
* Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 668-656-035
```

Our web service application can be accessed using the following URIs:

- <http://localhost:8080/>
- <http://localhost:8080/date>
- <http://localhost:8080/time>

The response from the web servers for these URIs is shown here:

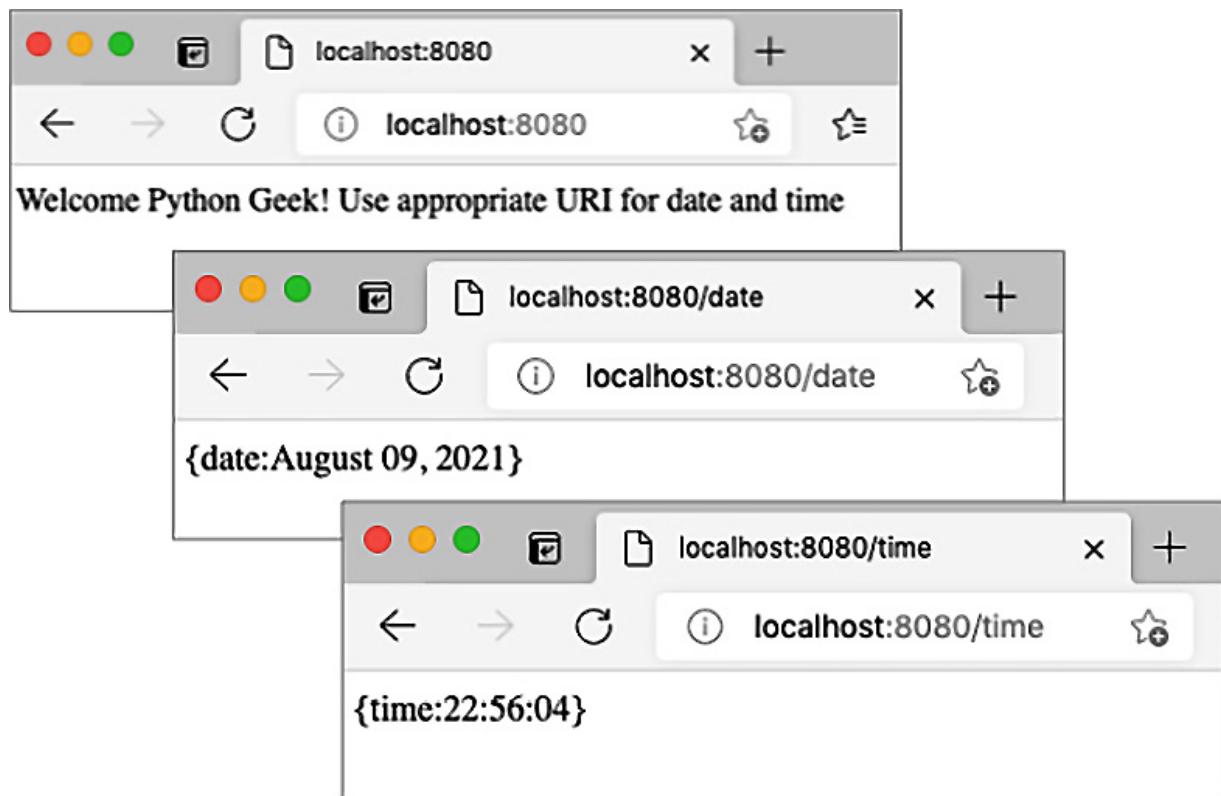


Figure 9.2 – Response in the web browser from our sample web service application

The web server will be stopped before we move on to the next phase – that is, deploying this application to Google App Engine.

## Deploying to Google App Engine

To deploy our web service application to GAE, we must use the following command from the project directory:

```
gcloud app deploy
```

Cloud SDK will read the **app.yaml** file, which provides input for creating an App Engine instance for this application. During the deployment, a container image is created using the Cloud Build service; then, this container image is uploaded to GCP storage for deployment. Once it has been successfully deployed, we can access the web service using the following command:

```
gcloud app browse
```

This command will open the application using the default browser on your machine. The URL of the hosted application will vary, depending on the region and zone that was selected during app creation.

It is important to understand that every time we execute the **deploy** command, it will create a new version of our application in App Engine, which means more resources will be consumed. We can check the versions that have been installed for a web application using the following command:

```
gcloud app versions list
```

The older versions of the application still can be in a serving state with slightly different URLs assigned to them. The older versions can be stopped, started, or deleted using the **gcloud app versions** Cloud SDK command and the version ID. An application can be stopped or started using the **stop** or **start** commands, as shown here:

```
gcloud app versions stop <version id>
gcloud app versions start <version id>
gcloud app versions delete <version id>
```

The version ID is available when we run the **gcloud app versions list** command. This concludes our discussion on building and deploying a Python web application to Google Cloud. Next, we will summarize how we can leverage the GCP console to deploy the same application.

## Using the GCP web console

The GCP console provides an easy-to-use web portal for accessing and managing GCP projects, as well as an online version of Google **Cloud Shell**. The console also offers customizable dashboards, visibility to cloud resources used by projects, billing details, activity logging, and many more features. When it comes to developing and deploying a web application using the GCP console, we have some features we can use thanks to the web UI, but most of the steps will require the use of Cloud Shell. This is a Cloud SDK that's available online through any browser.

Cloud Shell is more than Cloud SDK in several ways:

- It offers access to the **gcloud** CLI, as well as the **kubectl** CLI. **kubectl** is used for managing resources on the GCP Kubernetes engine.
- With Cloud Shell, we can develop, debug, build, and deploy our applications using **Cloud Shell Editor**.
- Cloud Shell also offers an online development server for testing an application before deploying it to App Engine.
- Cloud Shell comes with tools to upload and download files between the Cloud Shell platform and your machine.
- Cloud Shell comes with the ability to preview the web application on port 8080 or a port of your choice.

The Cloud Shell commands that are required to set up a new project, build the application, and deploy to App Engine are the same ones we discussed for Cloud SDK. That is why we will leave this for you to explore by following the same steps that we described in the previous section. Note that the project can be set up using the GCP console. The Cloud Shell interface can be enabled using the Cloud Shell icon on the top menu bar, on the right-hand side. Once Cloud Shell has been enabled, a command-line interface will appear at the bottom of the console's web page. This is shown in the following screenshot:

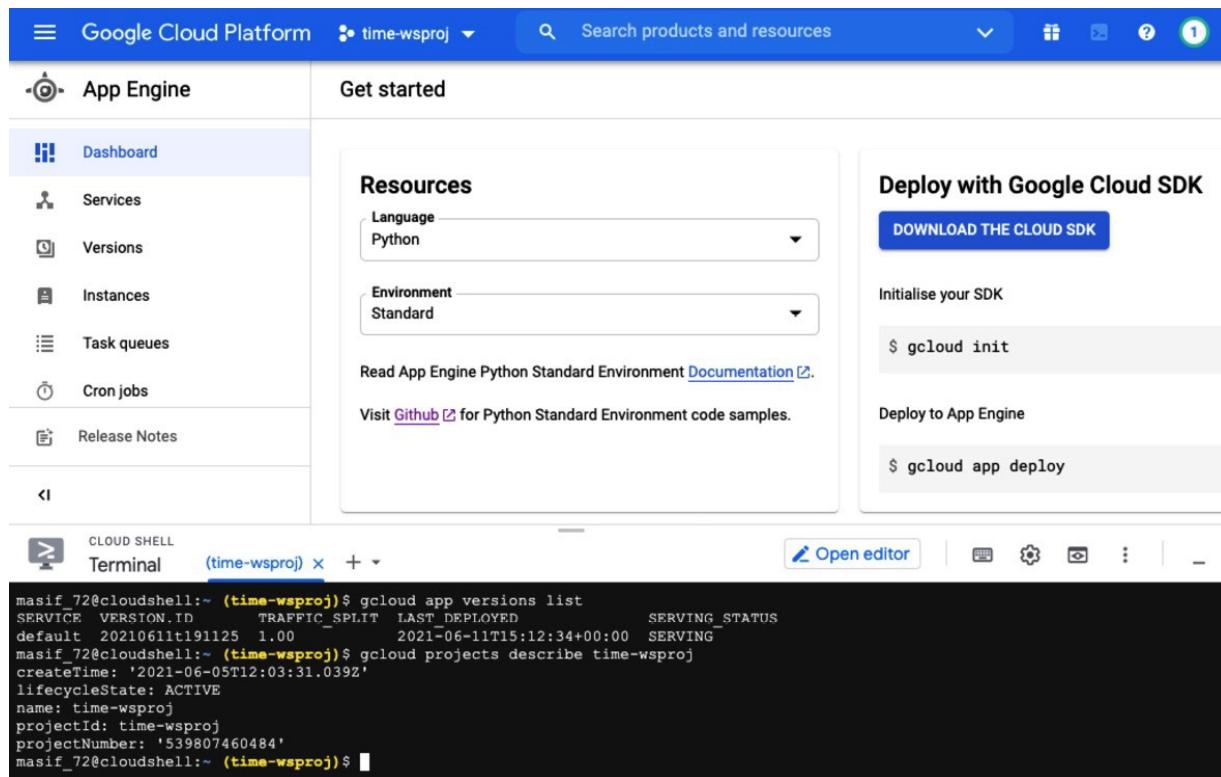


Figure 9.3 – GCP console with Cloud Shell

As we mentioned earlier, Cloud Shell comes with an editor tool that can be started by using the **Open editor** button. The following screenshot shows the Python file opened inside **Cloud Shell Editor**:

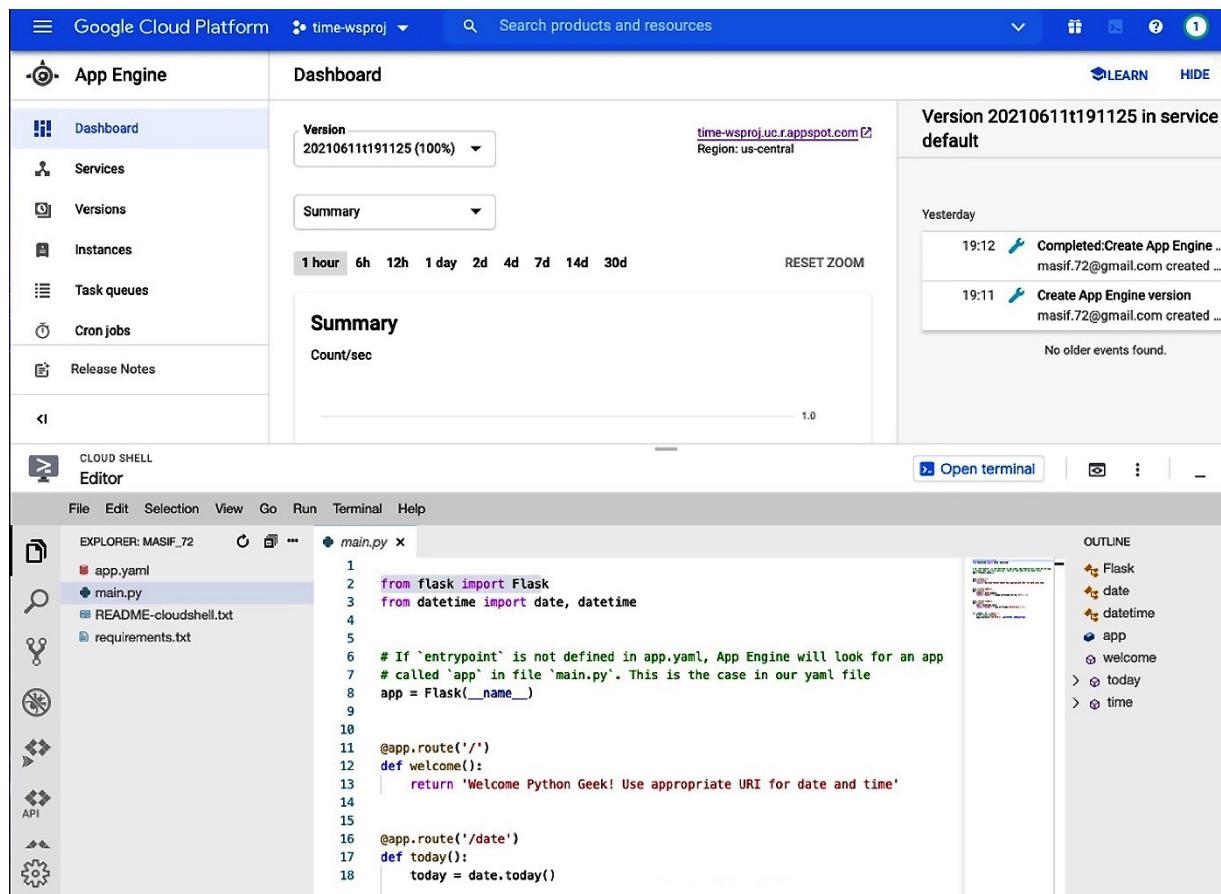


Figure 9.4 – GCP console with Cloud Shell Editor

Another option when it comes to building and deploying web applications is using third-party IDEs with Google App Engine plugins. Based on our experience, the plugins that are available for commonly used IDEs such as PyCharm and Eclipse are mostly built for Python 2 and legacy web application libraries. Directly integrating IDEs with GCP requires more work and evolution. At the time of writing, the best option is to use Cloud SDK or Cloud Shell directly in conjunction with the editor or IDE of your choice for application development.

In this section, we covered developing web applications using Python and deploying them to the GCP App Engine platform. Amazon offers the AWS Beanstalk service for web application deployment. The steps for deploying a web application in AWS Beanstalk are nearly the same as for GCP App Engine, except that AWS Beanstalk does not need projects to be set up as a prerequisite. Therefore, we can deploy applications faster in AWS Beanstalk.

To deploy our web service application in AWS Beanstalk, we must provide the following information, either using the AWS console or using the AWS CLI:

- Application name
- Platform (Python version 3.7 or 3.8, in our case)

- Source code version
- Source code, along with a **requirements.txt** file

We recommend using the AWS CLI for web applications that have a dependency on third-party libraries. We can upload our source code as a ZIP file or as a web archive (**WAR** file) from our local machine or copy them from an **Amazon S3** location.

The exact steps of deploying a web application on AWS Beanstalk are available at <https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create-deploy-python-flask.html>. Azure offers App Service for building and deploying web applications. You can find the steps for creating and deploying a web application on Azure at <https://docs.microsoft.com/en-us/azure/app-service/quickstart-python>.

Next, we will explore building driver programs for data processing using cloud platforms.

## Using Google Cloud Platform for data processing

Google Cloud Platform offers Cloud Dataflow as a data processing service to serve both batch and real-time data streaming applications. This service is meant for data scientists and analytics application developers so that they can set up a processing **pipeline** for data analysis and data processing. Cloud Dataflow uses Apache Beam under the hood. **Apache Beam** originated from Google, but it is now an open source project under Apache. This project offers a programming model for building data processing using pipelines. Such pipelines can be created using Apache Beam and then executed using the Cloud Dataflow service.

The Google Cloud Dataflow service is similar to **Amazon Kinesis**, Apache Storm, **Apache Spark**, and Facebook Flux. Before we discuss how to use Google Dataflow with Python, we will introduce Apache Beam and its pipeline concepts.

## Learning the fundamentals of Apache Beam

In the current era, data is like a cash cow for many organizations. A lot of data is generated by applications, by devices, and by human interaction with systems. Before consuming the data, it is important to process it. The steps that are defined for data processing are typically called pipelines in Apache Beam nomenclature. In other words, a data pipeline is a series of actions that are performed on raw data that originates from different sources, and then moves that data to a destination for consumption by analytic or business applications.

Apache Beam is used to break the problem into small bundles of data that can be processed in parallel. One of the main use cases of Apache Beam is **Extract, Transform, and Load (ETL)** applications. These three ETL steps are core for a pipeline whenever we have to move the data from a raw form to a refined form for data consumption.

The core concepts and components of Apache Beam are as follows:

- **Pipeline:** A pipeline is a scheme for transforming the data from one form into the other as part of data processing.
- **PCollection:** A Pcollection, or Parallel Collection, is analogous to RDD in Apache Spark. It is a distributed dataset that contains an immutable and unordered bag of elements. The size of the dataset can be fixed or bounded, similar to batch processing, where we know how many jobs to process in one batch. The size can also be flexible or unbounded based on the continuously updating and streaming data source.
- **PTransforms:** These are the operations that are defined in a pipeline to transform the data. These operations are operated on PCollection objects.
- **SDK:** A language-specific software development kit that's available for Java, Python, and Go to build pipelines and submit them to a runner for execution.
- **Runner:** This is an execution platform for Apache Beam pipelines. Runner software has to implement a single method called **run (Pipeline)** that is asynchronous by default. A few runners that are available are Apache Flink, Apache Spark, and Google Cloud Dataflow.
- **User-Defined Functions (ParDo/DoFn):** Apache Beam offers several types of **user-defined functions (UDFs)**. The most commonly used function is **DoFn**, which operates on a per-element basis. The provided **DoFn** implementation is wrapped inside an **ParDo** object that is designed for parallel execution.

A simple pipeline looks as follows:

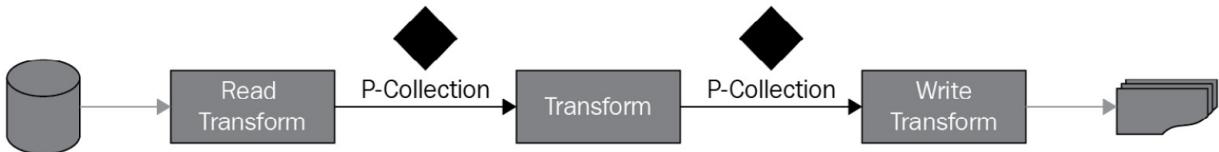


Figure 9.5 – Flow of a pipeline with three PTransform operations

To design a pipeline, we must typically consider three elements:

1. First, we need to understand the source of the data. Is it stored in a file or in a database, or is it coming as a stream? Based on this, we will determine what type of Read Transform operation we have to implement. As part of the read operation or a separate operation, we also need to understand the data format or structure.
2. The next step is to define and design what to do with this data. This is our main transform operation(s). We can have multiple transform operations one after the other in a serial way or in parallel on the same data. The Apache Beam SDK provides several pre-built transforms that can be used. It also allows us to write our own transforms using ParDo/DoFn functions.
3. Last, we need to know what the output of our pipeline will be and where to store the output results. This is shown as a Write Transform in the preceding diagram.

In this section, we discussed a simple pipeline structure to explain different concepts related to Apache Beam and pipelines. In practice, the pipeline can be relatively complex. A pipeline can have

multiple input data sources and multiple output sinks. The PTransforms operations may result in multiple PCollection objects, which requires PTransform operations to run in parallel.

In the next few sections, we will learn how to create a new pipeline and execute a pipeline using an Apache Beam runner or Cloud Dataflow runner.

## Introducing Apache Beam pipelines

In this section, we will discuss how to create Apache Beam pipelines. As we've discussed already, a pipeline is a set of actions or operations that are orchestrated to achieve certain data processing goals. The pipeline requires an input data source that can contain in-memory data, local or remote files, or streaming data. The pseudocode for a typical pipeline will look as follows:

```
[Final PCollection] = ([Initial Input PCollection] | [First PTransform] | [Second PTransform] | [Third PTransform])
```

The initial PCollection is used as input to the First PTransform operation. The output PCollection of First PTransform will be used as input to Second PTransform, and so on. The final output of the PCollection of the last PTransform will be captured as a Final PCollection object and be used to export the results to the target destination.

To illustrate this concept, we will build a few example pipelines of different complexity levels. These examples are designed to show the roles of different Apache components and libraries that are used in building and executing a pipeline. In the end, we will build a pipeline for a famous *word count* application that is also referenced in the Apache Beam and GCP Dataflow documentation. It is important to highlight that we must install the **apache-beam** Python library using the **pip** utility for all the code examples in this section.

### Example 1 – creating a pipeline with in-memory string data

In this example, we will create an input PCollection from an in-memory collection of strings, apply a couple of transform operations, and then print the results to the output console. The following is the complete example code:

```
#pipeline1.py: Separate strings from a PCollection
import apache_beam as beam
with beam.Pipeline() as pipeline:
 subjects = (
 pipeline
 | 'Subjects' >> beam.Create([
 'English Maths Science French Arts',]))
```

```
| 'Split subjects' >> beam.FlatMap(str.split)
| beam.Map(print))
```

For this example, it is important to highlight a few points:

- We used | to write different PTransform operations in a pipeline. This is an overloaded operator that is more like applying a PTransform to a PCollection to produce another PCollection.
- We used the >> operator to name each PTransform operation for logging and tracking purposes. The string between | and >> is used for displaying and logging purposes.
- We used three transform operations; all are part of the Apache Beam library:
  - a) The first transform operation is used to create a PCollection object, which is a string containing five subject names.
  - b) The second transform operation is used to split the string data into a new PCollection using a built-in **String** object method (**split**).
  - c) The third transform operation is used to print each entry in the PCollection to the console output.

The console's output will show a list of subject names, with one name in one line.

## **Example 2 – creating and processing a pipeline with in-memory tuple data**

In this code example, we will create a PCollection of tuples. Each tuple will have a subject name and a grade associated with it. The core PTransform operation of this pipeline is to separate the subject and its grade from the data. The sample code is as follows:

```
#pipeline2.py: Separate subjects with grade from a PCollection
import apache_beam as beam
def my_format(sub, marks):
 yield '{}\t{}'.format(sub,marks)
with beam.Pipeline() as pipeline:
 plants = (
 pipeline
 | 'Subjects' >> beam.Create([
 ('English', 'A'),
 ('Maths', 'B+'),
 ('Science', 'A-'),
 ('French', 'A'),
 ('Arts', 'A+')])
```

```

])
| 'Format subjects with marks' >> beam.FlatMapTuple(my_
format)
| beam.Map(print))

```

In comparison to the first example, we used the **FlatMapTuple** transform operation with a custom function to format the tuple data. The console output will show each subject's name, along with its grade, in a separate line.

### **Example 3 – creating a pipeline with data from a text file**

In the first two examples, we focused on building a simple pipeline to parse string data from a large string and to split tuples from a PCollection of tuples. In practice, we are working on a large volume of data that is either loaded from a file or storage system or coming from a streaming source. In this example, we will read data from a local text file to build our initial PCollection object and also output the final results to an output file. The complete code example is as follows:

```

#pipeline3.py: Read data from a file and give results back to another file
import apache_beam as beam

from apache_beam.io import WriteToText, ReadFromText
with beam.Pipeline() as pipeline:
 lines = pipeline | ReadFromText('sample1.txt')
 subjects = (
 lines
 | 'Subjects' >> beam.FlatMap(str.split))
 subjects | WriteToText(file_path_prefix='subjects',
 file_name_suffix='.txt',
 shard_name_template='')

```

In this code example, we applied a PTransform operation to read the text data from a file before applying any data processing-related PTransforms. Finally, we applied a PTransform operation to write the data to an output file. We used two new functions in this code example called

**ReadFromText** and **WriteToText**, as explained here:

- **ReadFromText:** This function is part of the Apache Beam I/O module and is used to read data from text files into a PCollection of strings. The file path or file pattern can be provided as an input argument to read from a local path. We can also use **gs://** to access any file in GCS storage locations.
- **WriteToText:** This function is used to write PCollection data to a text file. This requires the **file\_path\_prefix** argument at a minimum. We can also provide the **file\_path\_suffix** argument to set the file extension. **shard\_name\_template** is set to empty to create the file with a name using the prefix and suffix arguments. Apache Beam supports a shard name template for defining the filename based on a template.

When this pipeline is executed locally, it will create a file named **subjects.txt** with subject names captured in it, as per the PTransform operation.

## **Example 4 – creating a pipeline for an Apache Beam runner with arguments**

So far, we have learned how to create a simple pipeline, how to build a PCollection object from a text file, and how to write the results back to a file. In addition to these core steps, we need to perform a few more steps, to make sure our driver program is ready to submit the job to a GCP Dataflow runner or any other cloud runner. These additional steps are as follows:

- In the previous example, we provided the names of the input file and the output file pattern that are set in the driver program. In practice, we should expect these parameters to be provided through command-line arguments. We will use the **argparse** library to parse and manage command-line arguments.
- We will add extended arguments such as setting a runner. This argument will be used to set the target runner of the pipeline using DirectRunner or a GCP Dataflow runner. Note that DirectRunner is a pipeline runtime for your local machine. It makes sure that those pipelines follow the Apache Beam model as closely as possible.
- We will also implement and use the **ParDo** function, which will utilize our custom-built function for parsing strings from text data. We can achieve this using **String** functions, but it has been added here to illustrate how to use **ParDo** and **DoFn** with PTransform.

Here are the steps:

1. First, we will build the argument parser and define the arguments we expect from the command line. We will set the default values for those arguments and set additional helping text to be shown with the **help** switch on the command line. The **dest** attribute is important because it is used to identify any argument to be used in programming statements. We will also define the **ParDo** function, which will be used to execute the pipeline. Some sample code is presented here:

```
#pipeline4.py(part 1): Using argument for a pipeline

import re, argparse, apache_beam as beam

from apache_beam.io import WriteToText, ReadFromText

from apache_beam.options.pipeline_options import PipelineOptions

class WordParsingDoFn(beam.DoFn):

 def process(self, element):
 return re.findall(r'[\w\']+', element, re.UNICODE)

def run(argv=None):
 parser = argparse.ArgumentParser()
 parser.add_argument(
 '--input',
 dest='input',
 default='sample1.txt',
 help='Input file to process.')
```

```

parser.add_argument(
 '--output',
 dest='output',
 default='subjects',
 help='Output file to write results to.')

parser.add_argument(
 '--extension',
 dest='ext',
 default='.txt',
 help='Output file extension to use.')

known_args, pipeline_args = parser.parse_known_args(argv)

```

2. Now, we will set **DirectRunner** as our pipeline runtime and name the job to be executed. The sample code for this step is as follows:

```

#pipeline4.py(part 2): under the run method

pipeline_args.extend([
 '--runner=DirectRunner',
 '--job_name=demo-local-job',
])

pipeline_options = PipelineOptions(pipeline_args)

```

3. Finally, we will create a pipeline using the **pipeline\_options** object that we created in the previous step. The pipeline will be reading data from an input text file, transforming data as per our **ParDo** function, and then saving the results as output:

```

#pipeline4.py(part 3): under the run method

with beam.Pipeline(options=pipeline_options) as pipeline:
 lines = pipeline | ReadFromText(known_args.input)
 subjects = (
 lines
 | 'Subjects' >> beam.ParDo(WordParsingDoFn())
 with_output_types(str))
 subjects | WriteToText(known_args.output, known_args.ext)

```

When we execute this program directly through an IDE using the default values for the argument or initiate it from a command-line interface using the following command, we will get the same output:

```
python pipeline4.py --input sample1.txt --output myoutput --extension .txt
```

The words from the input text file (**sample1.txt**) are parsed and are put as one word in one line in the output file.

Apache Beam is a vast topic, so it is not possible to cover all its features without writing a few chapters on it. However, we have covered the fundamentals by providing code examples that will enable us to start writing simple pipelines that we can deploy on GCP Cloud Dataflow. We will cover this in the next section.

## Building pipelines for Cloud Dataflow

The code examples we've discussed so far focused on building simple pipelines and executing them using DirectRunner. In this section, we will build a driver program to deploy a word count data processing pipeline on Google Cloud Dataflow. This driver program is important for Cloud Dataflow deployments because we will set all cloud-related parameters inside the program. Due to this, there will be no need to use Cloud SDK or Cloud Shell to execute additional commands.

The word count pipeline will be an extended version of our **pipeline4.py** example. The additional components and steps required to deploy the word count pipeline are summarized here:

1. First, we will create a new GCP cloud project using steps that are similar to the ones we followed for our web service application for App Engine deployment. We can use Cloud SDK, Cloud Shell, or the GCP console for this task.
2. We will enable Dataflow Engine API for the new project.
3. Next, we will create a storage bucket for storing the input and output files and to provide temporary and staging directories for Cloud Dataflow. We can achieve this by using the GCP console, Cloud Shell, or Cloud SDK. We can use the following command if we are using Cloud Shell or Cloud SDK to create a new bucket:

```
gsutil mb gs://<bucket name>
```

4. You may need to associate a service account with the newly created bucket if it is not under the same project as the dataflow pipeline job is and select the *storage object admin* role for access control.

5. We must install Apache Beam with the necessary **gcp** libraries. This can be achieved by using the **pip** utility, as follows:

```
pip install apache-beam[gcp]
```

6. We must create a key for authentication for the GCP service account used for the GCP cloud project. This is not required if we will be running the driver program from a GCP platform such as Cloud Shell. The service account key must be downloaded on your local machine. To make the key available to the Apache Beam SDK, we need to set the path of the key file (a JSON file) to an environment variable called **GOOGLE\_APPLICATION\_CREDENTIALS**.

Before discussing how to execute a pipeline on Cloud Dataflow, we will take a quick look at the sample word count driver program for this exercise. In this driver program, we will define command-line arguments, very similar to the ones we did in the previous code example (**pipeline4.py**), except we will do the following:

- Instead of setting the **GOOGLE\_APPLICATION\_CREDENTIALS** environment variable through the operating system, we will set it using our driver program for ease of execution for testing purposes.
- We will upload the **sample.txt** file to Google storage, which is the **gs://muasif/input** directory in our case. We will use the path to this Google storage as the default value of the **input** argument.

The complete sample code is as follows:

```
wordcount.py(part 1): count words in a text file

import argparse, os, re, apache_beam as beam

from apache_beam.io import ReadFromText, WriteToText

from apache_beam.options.pipeline_options import PipelineOptions

from apache_beam.options.pipeline_options import SetupOptions

def run(argv=None, save_main_session=True):

 os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "some folder/ key.json"

 parser = argparse.ArgumentParser()

 parser.add_argument(
 '--input',
 dest='input',
 default='gs://muasif/input/sample.txt',
 help='Input file to process.')

 parser.add_argument(
 '--output',
 dest='output',
 default='gs://muasif/input/result',
 help='Output file to write results to.')

 known_args, pipeline_args = parser.parse_known_args(argv)
```

In the next step, we will set up extended arguments for the pipeline options to execute our pipeline on the Cloud Dataflow runtime. These arguments are as follows:

- Runtime platform (runner) for pipeline execution (DataflowRunner, in this case)
- GCP Cloud Project ID
- GCP region
- Google storage bucket paths for storing input, output, and temporary files
- Job name for tracking purposes

Based on these arguments, we will create a pipeline options object to be used for pipeline execution. The sample code for these tasks is as follows:

```

wordcount.py (part 2): under the run method

pipeline_args.extend([
 '--runner=DataflowRunner',
 '--project=word-count-316612',
 '--region=us-central1',
 '--staging_location=gs://muasif/staging',
 '--temp_location=gs://muasif/temp',
 '--job_name=my-wordcount-job',
])
pipeline_options = PipelineOptions(pipeline_args)
pipeline_options.view_as(SetupOptions).\
 save_main_session = save_main_session

```

Finally, we will implement a pipeline with the pipeline options that have already been defined and add our PTransform operations. For this code example, we added an extra PTransform operation to build a pair of each word with **1**. In the follow-up PTransform operation, we grouped the pairs and applied the **sum** operation to count their frequency. This gives us the count of each word in the input text file:

```

wordcount.py (part 3): under the run method

with beam.Pipeline(options=pipeline_options) as p:
 lines = p | ReadFromText(known_args.input)

 # Count the occurrences of each word.
 counts = (
 lines
 | 'Split words' >> (
 beam.FlatMap(
 lambda x: re.findall(r'[A-Za-z]+', x)).
 with_output_types(str))
 | 'Pair with 1' >> beam.Map(lambda x: (x, 1))
 | 'Group & Sum' >> beam.CombinePerKey(sum))

 def format_result(word_count):
 (word, count) = word_count
 return '%s: %s' % (word, count)

 output = counts | 'Format' >> beam.Map(format_result)
 output | WriteToText(known_args.output)

```

We set default values for each argument within the driver program. This means that we can execute the program directly with the **python wordcount.py** command or we can use the following command to pass the arguments through the CLI:

```
python wordcount.py \
 --project word-count-316612 \
 --region us-central1 \
 --input gs://muasif/input/sample.txt \
 --output gs://muasif/output/results \
 --runner DataflowRunner \
 --temp_location gs://muasif/temp \
 --staging_location gs://muasif/staging
```

The output file will contain the results, along with the count of each word in the file. GCP Cloud Dataflow provides additional tools for monitoring the progress of submitted jobs and for understanding the resource utilization to perform the job. The following screenshot of the GCP console shows a list of jobs that have been submitted to Cloud Dataflow. The summary view shows their statuses and a few key metrics:

Jobs								CREATE JOB FROM TEMPLATE	CREATE JOB FROM SQL	ENABLE SORTING	REFRESH	LEARN
	Running	Filter	Filter jobs									
Name	Type	End time	Elapsed time	Start time	Status	SDK version	ID					Region
my-wordcount-job	Batch	14 Jun 2021, 21:39:43	5 min 14 sec	14 Jun 2021, 21:34:29	Succeeded	2.30.0	2021-06-14_10_34_27-17633507493411316047					us-central1
my-wordcount-job	Batch	14 Jun 2021, 21:30:54	5 min 3 sec	14 Jun 2021, 21:25:51	Succeeded	2.30.0	2021-06-14_10_25_49-9276325152354335464					us-central1
my-wordcount-job	Batch	14 Jun 2021, 13:56:34	4 min 43 sec	14 Jun 2021, 13:51:51	Failed	2.30.0	2021-06-14_02_51_50-7355109158749375121					us-central1

Figure 9.6 – Cloud Dataflow jobs summary

We can navigate to the detailed job view (by clicking any job name), as shown in the following screenshot. This view shows the job and environment details on the right-hand side and a progress summary of the different PTransforms we defined for our pipeline. When the job is running, the status of each PTransform operation is updated in real time, as shown here:

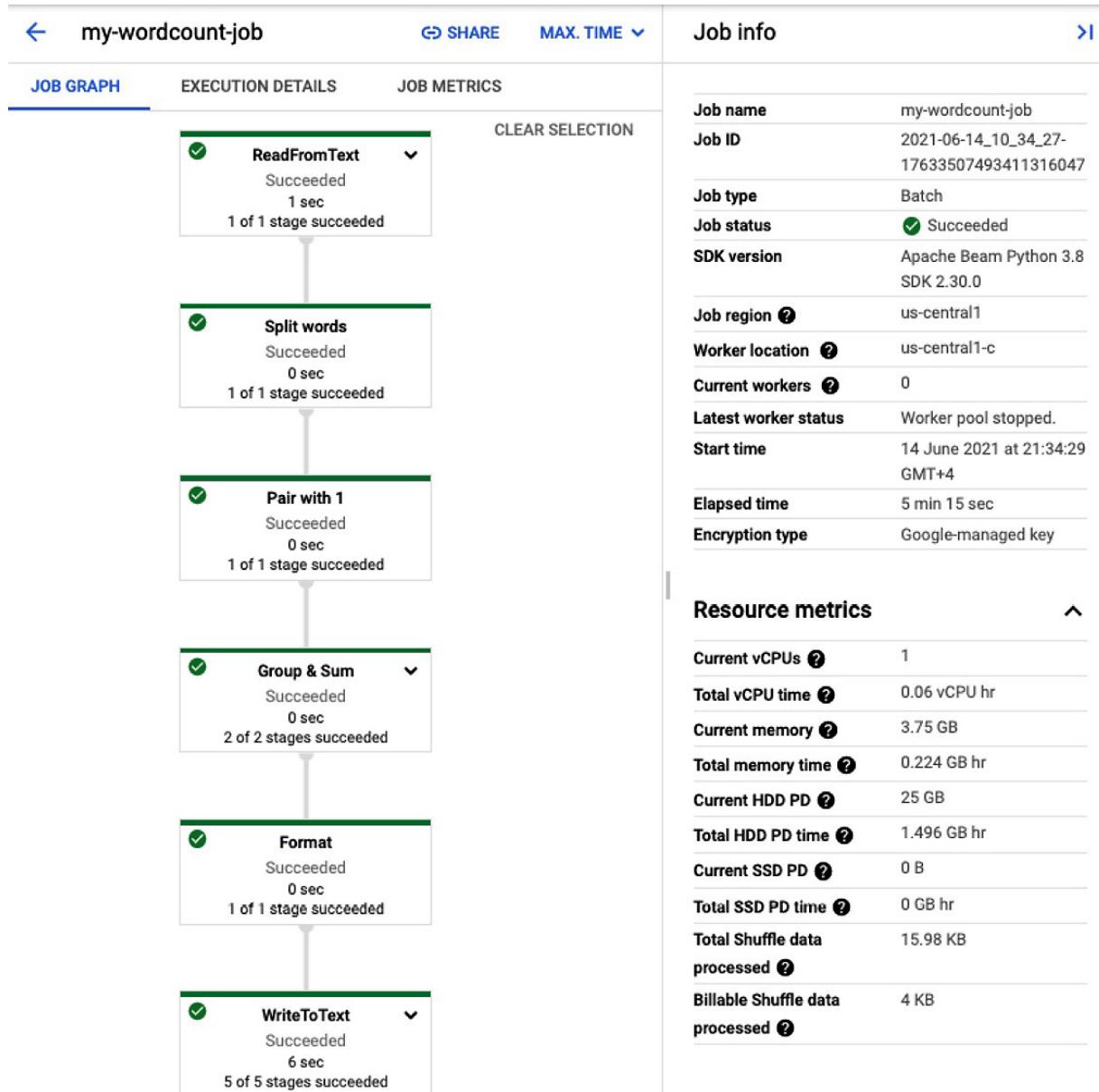


Figure 9.7 – Cloud Dataflow job detail view with a flowchart and metrics

A very important point to notice is that the different PTransform operations are named according to the strings we used with the `>>` operator. This is helpful for visualizing the operations conveniently. This concludes our discussion on building and deploying a pipeline for Google Dataflow. In comparison to Apache Spark, Apache Beam provides more flexibility for parallel and distributed data processing. With the availability of cloud data processing options, we can focus entirely on modeling the pipelines and leave the job of executing pipelines to cloud providers.

As we mentioned earlier, Amazon offers a similar service (AWS Kinesis) for deploying and executing pipelines. AWS Kinesis is more focused on data streams for real-time data. Like AWS

Beanstalk, AWS Kinesis does not require that we set up a project as a prerequisite. The user guides for data processing using AWS Kinesis are available at <https://docs.aws.amazon.com/kinesis/>.

## Summary

In this chapter, we discussed the role of Python for developing applications for cloud deployment in general, as well as the use of Apache Beam with Python for deploying data processing pipelines on Google Cloud Dataflow. We started this chapter by comparing three main public cloud providers in terms of what they offer for developing, building, and deploying different types of applications. We also compared the options that are available from each cloud provider for runtime environments. We learned that each cloud provider offers a variety of runtime engines based on the application or program. For example, we have separate runtime engines for classic web applications, container-based applications, and serverless functions. To explore the effectiveness of Python for cloud-native web applications, we built a sample application and learned how to deploy such an application on Google App Engine by using Cloud SDK. In the last section, we extended our discussion of the data process, which we started in the previous chapter. We introduced a new modeling approach for data processing (pipelines) using Apache Beam. Once we learned how to build pipelines with a few code examples, we extended our discussion to how to build pipelines for Cloud Dataflow deployment.

This chapter provided a comparative analysis of public cloud service offerings. This was followed by hands-on knowledge of building web applications and data processing applications for the cloud. The code examples included in this chapter will enable you to start creating cloud projects and writing code for Apache Beam. This knowledge is important for anyone who wants to solve their big data problems using cloud-based data processing services.

In the next chapter, we will explore the power of Python for developing web applications using the Flask and Django frameworks.

## Questions

1. How is AWS Beanstalk different from AWS App Runner?
2. What is the GCP Cloud Function service?
3. What services from GCP are available for data processing?
4. What is an Apache Beam pipeline?
5. What is the role of PCollection in a data processing pipeline?

## Further reading

- *Flask Web Development*, by Miguel Grinberg.
- *Advanced Guide to Python 3 Programming*, by John Hunt.
- *Apache Beam: A Complete Guide*, by Gerardus Blokdyk.
- *Google Cloud Platform for Developers*, by Ted Hunter, Steven Porter.
- *Google Cloud Dataflow documentation* is available at <https://cloud.google.com/dataflow/docs>.
- *AWS Elastic Beanstalk documentation* is available at <https://docs.aws.amazon.com/elastic-beanstalk>.
- *Azure App Service documentation* is available at <https://docs.microsoft.com/en-us/azure/app-service/>.
- *AWS Kinesis documentation* is available at <https://docs.aws.amazon.com/kinesis/>.

## Answers

1. AWS Beanstalk is a general-purpose PaaS offering for deploying web applications, whereas AWS App Runner is a fully managed service for deploying container-based web applications.
2. GCP Cloud Function is a serverless, event-driven service for executing a program. The specified event can be triggered from another GCP service or through an HTTP request.
3. Cloud Dataflow and Cloud Dataproc are two popular services for data processing offered by GCP.
4. An Apache Beam pipeline is a set of actions that have been defined to load the data, transform the data from one form into another, and write the data to a destination.
5. PCollection is like an RDD in Apache Spark that holds data elements. In pipeline data processing, a typical PTransform operation takes one or more PCollection objects as input and produces the results as one or more PCollection objects.

[OceanofPDF.com](http://OceanofPDF.com)

## Section 4: Using Python for Web, Cloud, and Network Use Cases

We will not finish our journey until we apply what we have learned so far to build modern solutions, especially for the cloud. This is a core part of our journey, where we challenge our learning progress by solving real-world problems. First, we look into how to build web applications and REST APIs using Python frameworks such as Flask. Next, we deep dive into implementing serverless applications for the cloud using the microservices architecture and serverless functions. We cover both local and cloud deployment options for our code exercises and case studies. Later, we explore how to use Python to build machine learning models and deploy them in the cloud. We conclude our journey by investigating the role of Python for network automation with real-world network management-related use cases.

This section contains the following chapters:

- [Chapter 10](#), *Using Python for Web Development and REST API*
- [Chapter 11](#), *Using Python for Microservices Development*
- [Chapter 12](#), *Building Serverless Functions using Python*
- [Chapter 13](#), *Python and Machine Learning*
- [Chapter 14](#), *Using Python for Network Automation*

[OceanofPDF.com](#)

## *Chapter 10: Using Python for Web Development and REST API*

A web application is a type of application that is hosted and run by a **web server** within an intranet or on the internet and accessed through a web browser on a client device. The use of web browsers as a client interface makes it convenient for users to access the applications from anywhere without installing any additional software on a local machine. This ease of access has contributed to the success and popularity of web applications for more than two decades. The use of web applications varies, from delivering static and dynamic content such as Wikipedia and newspapers, e-commerce, online games, social networking, training, multimedia content, surveys, and blogs, to complex **Enterprise Resource Planning (ERP)** applications.

Web applications are multi-tier by nature, typically three-tier applications. The three tiers are a UI, business logic, and database access. Therefore, developing web applications involves interacting with web servers for a UI, an **application server** for business logic, and database systems for persisting data. In the mobile applications era, the UI may be a mobile app that requires access to the business logic tier via a REST API. The availability of a REST API or any sort of web services interface has become a fundamental requirement for web applications. This chapter discusses how to use Python for building multi-tier web applications. There are several frameworks available in Python to develop web applications, but we selected **Flask** for our discussion in this chapter because of being feature-rich yet lightweight. Web applications are also termed *web apps* to differentiate them from mobile apps that are targeted at small devices.

We will cover the following topics in this chapter:

- Learning requirements for web development
- Introducing the Flask framework
- Interacting with databases using Python
- Building a REST API using Python
- Case study: Building a web application using the REST API

By the end of this chapter, you will be able to use the Flask framework to develop web applications, interact with databases, and build a REST API or web services.

## **Technical requirements**

The following are the technical requirements for this chapter:

- You need to have Python 3.7 or later installed on your computer.

- Python Flask library 2.x with its extensions installed on top of Python 3.7 or later.

Sample code for this chapter can be found at <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter10>.

We will start our discussion with key requirements of developing web applications and a REST API.

## Learning requirements for web development

Developing a web application includes building UI, routing user requests or user actions to application endpoints, translating the user input data, writing business logic for user requests, interacting with the data layer to read or write data, and serving the results back to users. All these development components may require different platforms and sometimes even use different programming languages for implementation. In this section, we will understand the components and tools required for web development, starting with web application frameworks or web frameworks.

## Web frameworks

Developing web applications from scratch is time-consuming and tedious. To make it convenient for web developers, web application frameworks were introduced in the very early days of web development. Web frameworks provide a set of libraries, directory structures, reusable components, and deployment tools. Web frameworks typically follow an architecture that enables developers to build complex applications in less time and in an optimized way.

There are several web frameworks available for Python: **Flask** and **Django** are the most popular. Both frameworks are free and open source. Flask is a lightweight framework and comes with standard functionalities that are required to build a web application, but it also allows additional libraries or extensions to use as they are needed. On the other hand, Django is a full stack framework that comes with all features out of the box without requiring additional libraries. Both approaches have merits and demerits, but in the end, we can develop any web application with any of these frameworks.

Flask is considered a better choice if you want to have full control of your application with a choice of using external libraries as they fit. Flask is also a good fit when project requirements are changing very frequently. Django is suitable if you want to have all tools and libraries available to you out of the box and you want to focus only on implementing the business logic. Django is a good choice for large-scale projects, but can be overkill for simple projects. The learning curve for Django is steep and requires prior experience in web development. If you are staring at web development with

Python for the first time, Flask is the way to go. Once you learn Flask, it is easy to adopt the Django framework for the next level of web projects.

When building web applications or any UI applications, we often come across the term **Model View Controller (MVC)** design pattern. This is an architectural design pattern that divides an application into three layers:

- **Model:** The model layer represents the data that is typically stored in a database.
- **View:** This layer is the UI with which users interact.
- **Controller:** The controller layer is designed to provide a logic to handle user interactions with the application through UI. For example, a user may want to create a new object or update an existing object. The logic of which UI (view) to present to the user for the create or update request with or without a model (data) is all implemented in the controller layer.

Flask does not provide direct support for MVC design patterns, but it can be implemented through programming. Django provides close enough implementation for MVC, but not fully. The controller in the Django framework is managed by Django itself and is not available for writing our own code in it. Django and many other web frameworks for Python follow the **Model View Template (MVT)** design pattern, which is like MVC except the template layer. The template layer in the MVT provides specially formatted templates to produce the expected UI with the capability to insert dynamic contents within HTML.

## User interface

A **UI** is the presentation layer of the application, and sometimes it is included as part of the web frameworks. But we are discussing it separately here to highlight key technologies and choices available for this layer. First and foremost, the user is interacting through a browser in **HyperText Markup Language (HTML)** and **Cascading Style Sheets (CSS)**. We can build our interfaces by writing HTML and CSS directly, but it is tedious and not feasible for delivering dynamic content in a timely manner. There are a few technologies available to make our lives easier when building UI:

- **UI framework:** These are mainly HTML and CSS libraries to provide different classes (styles) for building UI. We still need to write or generate core HTML parts of UI, but without worrying about how to beautify our web pages. A popular example of a UI framework is **bootstrap**, which is built on top of CSS. It was introduced by Twitter for internal use, but was made open source later for anyone to use. **ReactJS** is another popular option, but this is more a library than a framework and was introduced by Facebook.
- **Template engine:** A template engine is another popular mechanism to produce web content dynamically. A template is more like a definition of the desired output that holds static data as well as placeholders for dynamic contents. The placeholders are tokenized strings that are replaced by values at runtime. The output can be any format, such as HTML, XML, JSON, or PDF. **Jinja2** is one of the most popular templating engines used with Python, and is also included with the Flask framework. Django comes with its own templating engine.

- **Client-side scripting:** A client-side script is a program that is downloaded from the web server and is executed by the client web browser. JavaScript is the most popular client-side scripting language. There are many JavaScript libraries available to make web development easier.

We can use more than one technology to develop web interfaces. In a typical web project, all three are used in different capacities.

## Web server/application server

A web server is software that listens to client requests through HTTP and delivers content (such as web pages, scripts, images) as per the request type. The web server's fundamental job is to serve only static resources and is not capable of code execution.

The application server is more specific to a programming language. The main job of an application server is to provide access to the implementation of a business logic that is written using a programming language such as Python. For many production environments, the web server and application server are bundled as one software for ease of deployment. Flask comes with its own built-in web server, named **Werkzeug**, for the development phase, but this is not recommended for use in production. For production, we must use other options such as **Gunicorn**, **uWSGI**, and **GCP runtime engines**.

## Database

This is not a mandatory component, but it is almost essential for any interactive web application. Python offers several libraries to access commonly used database systems, such as **MySQL**, **MariaDB**, **PostgreSQL**, and **Oracle**. Python also comes equipped with a lightweight database server, **SQLite**.

## Security

Security is fundamental for web applications, mainly because the target audience is typically internet users, and data privacy is the utmost requirement in such environments. **Secure Sockets Layer (SSL)** and the more recently introduced **Transport Layer Security (TLS)** are the minimum acceptable security standards to secure the transmission of data between clients and the server. The transport-level security requirements are typically handled at the web server or sometimes proxy server level. User-level security is the next fundamental requirement, with minimum username and password requirements. User security is application-level security and developers are mainly responsible for designing and implementing it.

## API

The business logic layer in web applications can be consumed by additional clients. For example, a mobile app can use the same business logic for a limited or same set of features. For **Business to Business (B2B)** applications, a remote application can submit requests to the business logic layer directly. This is all possible if we expose standard interfaces such as a REST API for our business logic layer. In the current era, accessing the business logic layer through an API is a best practice to make the API ready from day one.

## Documentation

Documentation is as important as writing programming code. This is especially true for APIs. When we say that we have an API for our application, the first question from API consumers is whether you can share API documentation with us. The best way to have API documentation is to use built-in tools that come or possibly integrate with our web framework. **Swagger** is a popular tool that is used to generate documentation automatically from the comments that are added at the time of coding.

Now that we have discussed key requirements of web development, we will deep dive into how to develop a web application using Flask in the next section.

## Introducing the Flask framework

Flask is a micro web development framework for Python. The term *micro* indicates that the core of Flask is lightweight, but with the flexibility of being extensible. A simple example is interacting with a database system. Django comes with libraries required to interact with the most common databases. On the other hand, Flask allows the use of an extension as per the database type or as per the integration approach to achieve the same goal. Another philosophy of Flask is to use *convention over configuration*, which means that if we follow standard conventions of web development, we have to do less configuration. This makes Flask the best choice for beginners to learn web development with Python. We selected Flask for our web development, not only because of its ease-of-use capability, but also because it allows us to introduce different concepts in a stepwise approach.

In this section, we will learn the following aspects of web applications using Flask:

- Building a basic web application with routing
- Handling requests with different HTTP method types
- Rendering static and dynamic contents using Jinja2
- Extracting arguments from an HTTP request

- Interacting with database systems
- Handling errors and exceptions

Before we start working with code examples to be used in the next sections, Flask 2.x needs to be installed in our virtual environment. We will start with a basic web application using Flask.

## Building a basic application with routing

We have already used Flask to build a sample application for GCP App Engine deployment in the last chapter, *Python Programming for the Cloud*. We will refresh our knowledge of using Flask to develop a simple web application. We will start with a code example to understand how a web application is built and how routing works in it. The complete code example is as follows:

```
#app1.py: routing in a Flask application

from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
 return 'Hello World!'

@app.route('/greeting')
def greeting():
 return 'Greetings from Flask web app!'

if __name__ == '__main__':
 app.run()
```

Let's analyze this code example step by step:

1. **Initialization:** A Flask application must create an application instance (**app** in our case) as a first step. The web server will pass all requests from clients to this application instance using a protocol known as a **Web Server Gateway Interface (WSGI)**. The application instance is created by using the statement **app = Flask(\_\_name\_\_)**.

It is important to pass the module name as an argument to the **Flask** constructor. The Flask uses this argument to learn the location of the application, which becomes an input to determine the location of other files such as static resources, templates, and images. Using **\_\_name\_\_** is the convention (over configuration) to pass to the **Flask** constructor, and Flask takes care of the rest.

2. **Route:** Once a request arrives at the Flask **app** instance, now it is the responsibility of the instance to execute a certain piece of code to handle the request. This piece of code, which is typically a Python function, is called **handler**. The good news is that each request is typically (not all the time) associated with a single URL, which makes it possible to define a mapping between a URL and a Python function. This URL to Python function mapping is called a route. In our code example, we selected a simple approach to define this mapping by using the **route** decorator. For example, the **/hello** URL is mapped to the **hello** function, and

the `/greeting` URL is mapped to the `greeting` function. If we prefer to define all routes in one place, we can use `add_url_rule` with the app instance for all route definitions.

3. **Handler function:** The handler function after processing the request has to send a response back to the client. A response can be a simple string with or without HTML, or it can be a complex web page that can be static or dynamic based on a template. In our code example, we returned a simple string for illustration purposes.
4. **Web server:** Flask applications come with a development server that can be started with the `app.run()` method, or by using the `flask run` command in a shell. When we start this web server, it looks for an `app.py` or `wsgi.py` module by default, and it will be loaded automatically with the server if we use the `app.py` name for our module file (again, convention over configuration). But if we are using a different name for our module (it is our case), we must set an environment variable, `FLASK_APP = <module name>`, which will be used by the web server to load the module.

If you have created a Flask project using an IDE such as **PyCharm Pro**, the environment variable will be set for you as part of project settings. In case you are using a command-line shell, you can set the environment variable using the following command as per your operating system:

```
export FLASK_APP= app1.py. #for macOS and Linux
set FLASK_APP = app1.py. #for MS windows
```

When the server starts, it listens for client requests at the URL `http://localhost:5000/` and it is accessible only on your local machine by default. If we want to start the server using a different hostname and port, we can use the following command (or the equivalent Python statement):

```
Flask run --host <ip_address> --port <port_num>
```

5. **Web client:** We can test our application through a browser by entering the URL in the address bar or by using a `curl` utility for simple HTTP requests. For our example, we can test our application using the following `curl` commands:

```
curl -X GET http://localhost:5000/
curl -X GET http://localhost:5000/greeting
```

Now that we have finished our discussion on the fundamentals of a Flask application, we will start exploring topics related to consuming the request and sending a dynamic response back to the clients.

## Handling requests with different HTTP method types

HTTP is based on a **request-response** model between a client and a server. A client (for example, a web browser) can send different verbs or, more appropriately, call methods to identify the type of request for the server. These methods include **GET**, **POST**, **PUT**, **DELETE**, **HEAD**, **PATCH**, and **OPTIONS**. **GET** and **POST** are the most frequently used HTTP methods, and so we will cover only these to illustrate our web development concepts.

But before discussing these two methods, it is also important to understand the two key components of HTTP, which are **HTTP request** and **HTTP response**. An HTTP request is divided into three parts:

- **Request line:** This line includes the HTTP method to be used, the URI of the request, and the HTTP protocol (version) to be used:

```
GET /home HTTP/1.1
```

- **Header fields:** Headers are metadata that provide the information pertaining to the request. Each header entry is provided as a key-value pair, separated by colons (:)
- **Body** (optional): This is a placeholder where we can add additional data. For a web application, we can send form data with **POST** requests inside the body of the HTTP request. For a REST API, we can send data for **PUT** or **POST** requests within the body.

When we send an HTTP request to a web server, we will get an HTTP response as a result. The HTTP response will have similar parts to the HTTP request:

- **Status line:** This line indicates whether the response is successful or comes with an error. An error code is highlighted in the status line:

```
HTTP/1.1 200 OK
```

Status code **200** or something in the range of **200-299** indicates success. Error codes are in the range of **400-499** for client-side errors, and in the range of **500-599** for server-side errors.

- **Header:** Header fields are similar to the HTTP request header fields.
- **Body** (optional): Although optional, this is the key part of an HTTP response. This can include HTML pages for web applications or data in any other format.

**GET** is used to send a request for a certain resource identified in the URL with the option to add a **query string** as part of the URL. The ? is added in the URL to keep the query string separate from the base URL. For example, if we search for the word *Python* on Google, we will see a URL as follows in the browser:

<https://www.google.com/search?q=Python>

In this URL, **q=Python** is a query string. A query string is used to carry data in the form of key-value pairs. This approach of accessing resources is popular because of its simplicity but does have its limitations. The data in a query string is visible in the URL, which means we cannot send sensitive information such as a username and password as a query string. The length of the query string cannot be more than 255 characters. However, the **GET** method is in use for searching websites such as Google and YAHOO for reasons of simplicity. In the case of the **POST** method, the data is sent via the HTTP request body, which eliminates the limitations of the **GET** method. The data does not appear as part of the URL and there is no limit in terms of the data that we can send to the HTTP server. There is also no limit in terms of the data types supported using the **POST** method.

Flask provides a few convenient ways to identify whether a request is sent using **GET** or **POST** or is using any other method. In our next code example, we illustrate two approaches; the first approach uses the **route** decorator, with an exact list of the method types expected, and the second approach is

using a decorator specific for the HTTP method type, such as the **get** decorator and the **post** decorator. The use of both approaches is illustrated in the next code example, followed by a detailed analysis:

```
#app2.py: map request with method type

from flask import Flask, request

app = Flask(__name__)

@app.route('/submit', methods=['GET'])
def req_with_get():

 return "Received a get request"

@app.post('/submit')

def req_with_post():

 return "Received a post request"

@app.route('/submit2', methods = ['GET', 'POST'])
def both_get_post():

 if request.method == 'POST':
 return "Received a post request 2"
 else:
 return "Received a get request 2"
```

Let's discuss the three route definitions and corresponding functions in our sample code one by one:

- In the first route definition (`@app.route('/submit', methods=['GET'])`), we used the **route** decorator for mapping a URL, with requests of the **GET** type to a Python function. With this decorator setting, our Python function will handle requests with the **GET** method just for the `/submit` URL.
- In the second route definition (`@app.post('/submit')`), we used the **post** decorator and only specify the request URL with it. This is a simplified version of mapping a request with the **POST** method to a Python function. This new setting is equivalent to the first route definition, but with the **POST** method type in a simplified form. We can achieve the same for a **GET** method by using the **get** decorator.
- In the third route definition (`@app.route('/submit2', methods = ['GET', 'POST'])`), we mapped a single URL with requests using both the **POST** and **GET** methods to a single Python function. This is a convenient approach when we are expecting to handle any request method by using a single handler (Python function). Inside the Python function, we used the **method** attribute of the request object to identify whether the request is of the **GET** or **POST** type. Note that the request object is made available to our Flask app by the web server once we import the **request** package into our program. This approach gives flexibility for clients to submit requests using any one of the two methods using the same URL and, as a developer, we mapped them to a single Python function.

We can test this code example more conveniently through the **curl** utility because it will not be easy to submit a **POST** request without defining an HTML form. The following **curl** commands can be used to send HTTP requests to our web application:

```
curl -X GET http://localhost:5000/submit
curl -X POST http://localhost:5000/submit
curl -X GET http://localhost:5000/submit2
curl -X POST http://localhost:5000/submit2
```

Next, we will discuss how to render a response from static pages and templates.

## Rendering static and dynamic contents

The static contents are important for a web application as they include CSS and JavaScript files. The static files can be served directly by the web server. Flask can also make this happen if we create a directory called **static** in our project and redirect the client to the static file location.

Dynamic contents can be created using Python, but it is tedious and requires quite an effort to maintain such a code in Python. The recommended approach is to use a template engine such as **Jinja2**. Flask comes with a Jinja2 library, so there is no additional library to install, nor do we need to add any additional configuration to set up Jinja2. A sample code with two functions, one handling a request for static contents and the other for dynamic contents, is shown next:

```
#app3.py: rendering static and dynamic contents

from flask import Flask, render_template, url_for, redirect

app = Flask(__name__)

@app.route('/hello')

def hello():

 hello_url = url_for('static', filename='app3_s.html')

 return redirect(hello_url)

@app.route('/greeting')

def greeting():

 msg = "Hello from Python"

 return render_template('app3_d.html', greeting=msg)
```

For a better understanding of this sample code, we will highlight the key points:

- We import additional modules from Flask such as **url\_for**, **redirect**, and **render\_template**.
- For the **/hello** route, we build a URL using the **url\_for** function with the **static** directory and the name of the HTML file as arguments. We send the response, which is an instruction to the browser, to redirect the client to the URL of a static file location. The redirect instructions are indicated to the web browser by using a status code in the range of **300-399**, which is automatically set by Flask when we used the **redirect** function.
- For the **/greeting** route, we render a Jinja template, **app3\_d.html**, using the **render\_template** function. We also passed a greeting message string as a value to a variable for the template. The greeting variable will be available to the Jinja template, as

shown in the following template excerpt from the **app3\_d.html** file:

```
<!DOCTYPE html>
<body>
{%
 if greeting %}
 <h1> {{greeting}}!</h1>
{%
 endif %}
</body>
</html>
```

This is the simplest Ninja template with an **if** statement enclosed by **<% %>**, and the Python variable is included by using the two curly brackets **{}** format. We will not go into the details regarding Ninja2 templates, but we highly recommended that you get familiar with Ninja2 templates through their online documentation (<https://jinja.palletsprojects.com/>).

This sample web application can be accessed using a web browser and using the **curl** utility. In the next section, we will discuss how to extract parameters from different types of requests.

## Extracting parameters from an HTTP request

Web applications are different from websites because they are interactive with users and this is not possible without exchanging data between a client and a server. In this section, we will discuss how to extract data from a request. Depending on the type of HTTP method used, we will adopt a different approach. We will cover the following three types of requests as follows:

- Parameters as part of the request URL
- Parameters as query string with a **GET** request
- Parameters as HTML form data with a **POST** request

A sample code with three different routes to cover the three aforementioned request types is as follows. We are rendering a Ninja template (**app4.html**), which is the same as we used for the **app3\_d.html** file, except the variable name is **name** instead of **greeting**:

```
#app4.py: extracting parameters from different requests
from flask import Flask, request, render_template
app = Flask(__name__)
@app.route('/hello')
@app.route('/hello/<fname> <lname>')
def hello_user(fname=None, lastname=None):
 return render_template('app4.html', name=f"{fname}{lastname}")
```

```

@app.get('/submit')

def process_get_request_data():
 fname = request.args['fname']
 lname = request.args.get('lname', '')
 return render_template('app4.html', name=f'{fname}{lname}')

@app.post('/submit')

def process_post_request_data():
 fname = request.form['fname']
 lname = request.form.get('lname', '')
 return render_template('app4.html', name=f'{fname}{lname}')

```

Next, we will discuss the parameter extraction approach for each case:

- For the first set of routes (**app.route**), we defined a route in a way that any text after `/hello/` is considered parameters with the request. We can set zero or one or two parameters, and our Python function is able to handle any combination and returns the name (which can be empty) to the template as a response. This approach is good for simple cases of parameter passing to the server program. This is a popular choice in REST API development to access a single resource instance.
- For the second route (**app.get**), we are extracting query string parameters from the `args` dictionary object. We can fetch the parameter value either by using its name as a dictionary key or by using the **GET** method with the second argument as a default value. We used an empty string as a default value with the **GET** method. We showed both options, but we recommend using the **GET** method if you want to set a default value in case no parameter exists in the request.
- For the third route (**app.post**), parameters come as form data as part of the body of the HTTP request and we will be using form dictionary object to extract these parameters. Again, we used the parameter name as a dictionary key and also used the **GET** method for illustration purposes.
- To test these scenarios, we recommend using the **curl** utility, especially for **POST** requests. We tested the application with the following commands:

```

curl -X GET http://localhost:5000/hello
curl -X GET http://localhost:5000/hello/jo%20so
curl -X GET 'http://localhost:5000/submit?fname=jo&lname=so'
curl -d "fname=jo&lname=so" -X POST http://localhost:5000/submit

```

In the next section, we will discuss how to interact with a database in Python.

## Interacting with database systems

A full stack web application requires the persistence of structured data, so knowledge and experience of working with a database are prerequisites for web development. Python and Flask can integrate with most SQL or no-SQL database systems. Python itself comes with a lightweight SQLite database with the module name **sqlite3**. We will use SQLite because it does not require setting up a separate

database server and works very well for small-scale applications. For production environments, we must use other database systems, such as MySQL or MariaDB, or PostgreSQL. To access and interact with a database system, we will use one of the Flask extensions, **Flask-SQLAlchemy**. The **Flask-SQLAlchemy** extension is based on the **SQLAlchemy** library of Python and makes the library available to our web application. The **SQLAlchemy** library provides an **Object Relational Mapper (ORM)**, which means it maps our database tables to Python objects. The use of an ORM library not only expedites the development cycle, but also provides the flexibility to switch the underlying database system without changing our code. Therefore, we will recommend using **SQLAlchemy** or a similar library to work with database systems.

To interact with any database system from our application, we need to create our Flask application instance as usual. The next step is to configure the application instance with the URL for our database location (a file in the case of SQLite3). Once the application instance is created, we will create a **SQLAlchemy** instance by passing it to the application instance. When using a database such as SQLite, we only have to initialize the database the first time. It can be initiated from a Python program, but we will not favor this approach so as to avoid the database reset every time we start our application. It is recommended to initialize the database one time only from a command line using the **SQLAlchemy** instance. We will discuss the exact steps of initializing the database after our code example.

To illustrate the use of the **SQLAlchemy** library with our web application, we will create a simple application to *add*, *list*, and *delete* student objects from a database table. Here is a sample code of the application to initialize the Flask application and the database instance (a **SQLAlchemy** instance) and also to create a **Model** object of the **Student** class:

```
#app5.py (part1): interacting with db for create, delete and list objects
from flask import Flask, request, render_template, redirect
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///student.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)
class Student(db.Model):
 id = db.Column(db.Integer, primary_key=True)
 name = db.Column(db.String(80), nullable=False)
 grade = db.Column(db.String(20), nullable=True)
```

```

def __repr__(self):
 return '<Student %r>' % self.name

```

Once we have created the **SQLAlchemy** instance, **db**, we can work with database objects. The beauty of an ORM library such as **SQLAlchemy** is that we can define a database schema, known as a **model** in ORM terminology, within Python as Python classes. For our code example, we created a class, **Student**, which is inherited from a base class, **db.Model**. In this model class, we defined the **id**, **name**, and **grade** attributes that will correspond to three columns in a database table, *Student*, in the **SQLite3** database instance. For each attribute, we defined its data type with the maximum length, whether it has a primary key, and whether it is nullable. These additional attribute definitions are important to configure the database tables in an optimized way.

In the following code snippet, we will illustrate a Python function, **list\_students**, for getting a list of student objects from the database. This function is mapped to the **/list** URL of our sample web application and it returns all **Student** objects from the database table by using the **all** method on the **query** instance (an attribute of the **db** instance). Note that the **query** instance and its methods are available from the base class, **db.Model**:

```

#app5.py (part 2)

@app.get('/list')
def list_students():
 student_list = Student.query.all()
 return render_template('app5.html', students=student_list)

```

In the next code snippet, we will write a function (**add\_student**) to add students to the database table. This function is mapped to the **/add** URL and expects the student's name and their grade passed as request parameters using the **GET** method. To add a new object to the database, we will create a new instance of the **Student** class with the requisite attribute values and then use the **db.Session** instance to add to the ORM layer by using the **add** function. The **add** function will not add the instance to the database by itself. We will use the **commit** method to push it to the database table. Once a new student is added to our database table, we redirect control to the **/list** URL. The reason we used a redirect to this URL is that we want to return the latest list of students after adding a new one and to reuse the **list\_students** function, which we already implemented. The complete code for the **add\_student** function is as follows:

```

#app5.py(part 3)

@app.get('/add')
def add_student():
 fname = request.args['fname']

```

```

lname = request.args.get('lname', '')
grade = request.args.get('grade', '')
student = Student(name=f"{fname} {lname}", grade=grade)
db.session.add(student)
db.session.commit()
return redirect("/list")

```

In the last part of this code example, we will write a Python function (**delete\_student**) to delete a student from the database table. This function is mapped to the `/delete<int:id>` URL. Note that we are expecting the client to send the student ID (which we sent with a list of students using the **list** request). To delete a student, first, we query for the exact student instance by using the student ID. This is achieved by using the **filter\_by** method on the **query** instance. Once we have the exact **Student** instance, we use the **delete** method of the **db.Session** instance and then commit the changes. As with the **add\_student** function, we redirected the client to the `/list` URL to return an up-to-date list of students to our Jinja template:

```

#app5.py (part 4)

@app.get('/delete/<int:id>')
def del_student(id):
 todelete = Student.query.filter_by(id=id).first()
 db.session.delete(todelete)
 db.session.commit()
 return redirect("/list")

```

To show a list of students in a browser, we created a simple Jinja template (**app5.html**). The **app5.html** template file will provide a list of students in a table format. It is important to note that we use a Jinja **for** loop to build the HTML table rows dynamically, as shown in the following Jinja template:

```

<!DOCTYPE html>
<body>
<h2>Students</h2>
{% if students|length > 0 %}
<table>
<thead>
<tr>
<th scope="col">SNo</th>
<th scope="col">name</th>

```

```

<th scope="col">grade</th>
</tr>
</thead>
<tbody>
 {% for student in students %}
 <tr>
 <th scope="row">{{student.id}}</th>
 <td>{{student.name}}</td>
 <td>{{student.grade}}</td>
 </tr>
 {% endfor %}
</tbody>
</table>
{% endif %}
</body>
</html>

```

Before starting this application, we should initialize the database schema as a one-time step. This can be done by using a Python program, but we have to make sure the code is executed only once or only when the database is not already initialized. A recommended approach is to do this step manually using the Python shell. In the Python shell, we can import the **db** instance from our application module and then use the **db.create\_all** method to initialize the database as per the model classes defined in our program. Here are the sample commands to be used for our application for database initialization:

```

>>> from app5 import db
>>> db.create_all()

```

These commands will create a **student.db** file in the same directory where we have our program. To reset the database, we can either delete the **student.db** file and rerun the initialization commands, or we can use the **db.drop\_all** method in the Python shell.

We can test the application using the **curl** utility or through a browser using the following URLs:

- <http://localhost:5000/list>
- <http://localhost:5000/add?fname=John&Lee=asif&grade=9>
- <http://localhost:5000/delete/<id>>

Next, we will discuss how to handle errors in a Flask-based web application.

## Handling errors and exceptions in web applications

In all our code examples, we did not pay attention to how to deal with situations when a user enters an incorrect URL in their browser or sends a wrong set of arguments to our application. This was not a design intention, but the aim was to focus on the key components of web applications first. The beauty of web frameworks is that they typically support error handling by default. If any error occurs, an appropriate status code is returned automatically. The error codes are well defined as part of the HTTP protocol. For example, the error codes from **400** to **499** indicate errors with client requests, and the error codes from **500** to **599** indicate problems with the server while executing the request. A few commonly observed errors are summarized next:

Error Code	Name	Description
400	Bad Request	This error indicates either the URL or the request contents are incorrect.
401	Unauthorized	This error appears when the username or password is not provided or not correctly provided with the request.
403	Forbidden	This error shows that a user is trying to access a resource that is not allowed for that user.
404	Not Found	This error is returned when we are trying to access a resource that is not available. This is usually for the wrong URL.
500	Internal Server Error	This error indicates that the request is correct but that something happened on the server side.

Table 10.1 – A commonly observed HTTP errors

A complete list of HTTP status codes and errors is available at <https://httpstatuses.com/>.

The Flask framework also comes with an error handling framework. While handling client requests, if our program breaks, a **500 Internal Server Error** is returned by default. If a client requests a URL that is not mapped to any Python function, Flask will return a **404 Not Found** error to the client. These different error types are implemented as subclasses of the **HTTPException** class, which is part of the Flask library.

If we want to handle these errors or exceptions with a custom behavior or custom message, we can register our handler with the Flask application. Note that an error handler is a function in Flask that is only triggered when an error occurs, and we can associate a specific error or generic exception with our handlers. We build a sample code to illustrate the concept at a high level. First, we will illustrate a simple web application with two functions (**hello** and **greeting**) for handling two URLs, as shown in the following sample code:

```
#app6.py(part 1): error and exception handling

import json

from flask import Flask, render_template, abort

from werkzeug.exceptions import HTTPException

app = Flask(__name__)

@app.route('/')
def hello():

 return 'Hello World!'

@app.route('/greeting')
def greeting():

 x = 10/0

 return 'Greetings from Flask web app!'
```

To handle the errors, we will register our handler against the app instance using the **errorHandler** decorator. For our sample code (shown next), we registered a **page\_not\_found** handler against error code **404** for the Flask application. For error code **500**, we registered an **internal\_error** function as an error handler. In the end, we registered **generic\_handler** for the **HTTPException** class. This generic handler will catch the error or exception other than **404** and **500**. A sample code with all three handlers is shown next:

```
#app6.py(part 2)

@app.errorhandler(404)
def page_not_found(error):

 return render_template('error404.html'), 404

@app.errorhandler(500)
def internal_error(error):

 return render_template('error500.html'), 500

@app.errorhandler(HTTPException)
def generic_handler(error):

 error_detail = json.dumps({
 "code": error.code,
 "name": error.name,
 "description": error.description,
 })

 return render_template('error.html',
 err_msg=error_detail), error.code
```

For illustration purposes, we also wrote basic Jinja templates with custom messages; **error404.html**, **error500.html**, and **error.html**. The **error404.html** and **error500.html** templates are using the message that is hardcoded in the template. However, the **error.html** template expects the custom message coming from the web server. To test these sample applications, we will request the following through a browser or the **curl** utility:

- **GET http://localhost:5000/**: We will expect a normal response in this case.
- **GET http://localhost:5000/hello**: We will expect a **404** error as there is no Python function mapped to this URL and the Flask app will render an **error404.html** template.
- **GET http://localhost:5000/greeting**: We will expect a **500** error because we try to divide a number by zero to raise the **ZeroDivisionError** error. Since this is a server-side error (**500**), it will trigger our **internal\_error** handler, which renders a generic **error500.html** template.
- **POST http://localhost:5000/**: To emulate the role of a generic handler, we will send a request that triggers an error code other than **404** and **500**. This is easily possible by sending a **POST** request for a URL that is expecting a **GET** request and the server will raise a **405** error in this case (for an unsupported HTTP method). We have no error handler specific for error code **405** in our application, but we have registered a generic handler with **HTTPException**. This generic handler will handle this error and render a generic **error.html** template.

This concludes our discussion of using the Flask framework for web application development. Next, we will explore building a REST API using Flask extensions.

## Building a REST API

**REST**, or **ReST**, is an acronym for **Representational State Transfer**, which is an architecture for client machines to request information about the resources that exist on a remote machine. **API** stands for **Application Programming Interface**, which is a set of rules and protocols for interacting with application software running on different machines. The interaction of different software entities is not a new requirement. In the last few decades, there have been many technologies proposed and invented to make software-level interactions seamless and convenient. A few noticeable technologies include **Remote Procedure Call (RPC)**, **Remote Method Invocation (RMI)**, CORBA, and SOAP web services. These technologies have limitations in terms of being tied to a certain programming language (for example, RMI) or tied to a proprietary transport mechanism, or using only a certain type of data format. These limitations have been almost entirely eliminated by the RESTful API, which is commonly known as the REST API.

The flexibility and simplicity of the HTTP protocol make it a favorable candidate to use as a transport mechanism for a REST API. Another advantage of using HTTP is that it allows several data formats for data exchange (such as text, XML, and JSON) and is not constrained to one format, such as how XML is the only format for SOAP-based APIs. The REST API is not tied to any one specific

language, which makes it a de facto choice for building APIs for web interactions. An architectural view of a REST API call from a REST client to a REST server using HTTP is shown next:

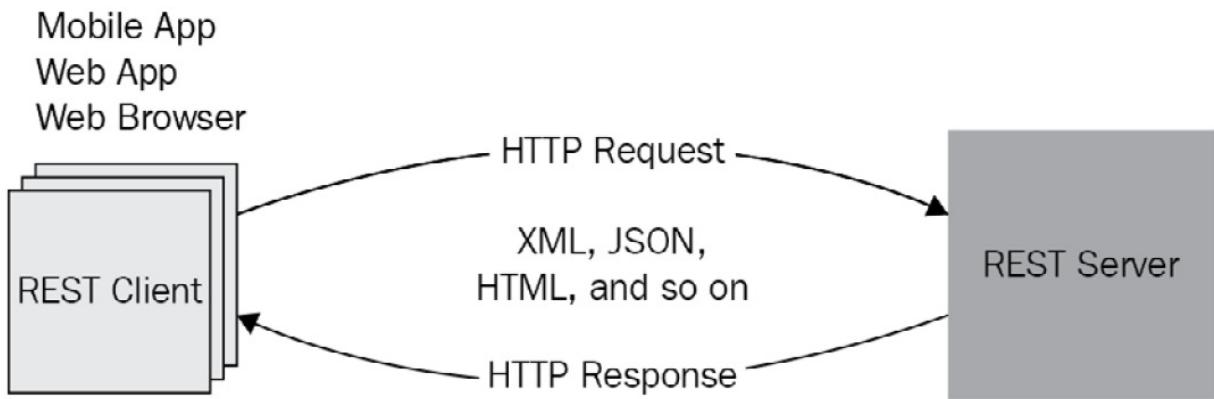


Figure 10.1 – REST API interaction between clients and a server

A REST API relies on HTTP requests and uses its native methods such as **GET**, **PUT**, **POST**, and **DELETE**. The use of the HTTP method simplifies the implementation of client-side and server-side software from an API design perspective. A REST API is developed keeping in mind the CRUD operations concept. **CRUD** stands for **Create**, **Read**, **Update**, and **Delete**. This is where HTTP methods find themselves aligned one on one with the CRUD operations, for example, **GET** for **Read**, **POST** for **Create**, **PUT** for **Update**, and **DELETE** for **Delete** operation.

When building a REST API with HTTP methods, we must be careful in choosing the correct method based on its idempotency capabilities. An operation is considered idempotent in mathematics if the operation is giving the same results even if it is repeated multiple times. From the rest API design perspective, the **POST** method is not idempotent, which means we have to make sure that the API clients are not initiating a **POST** request multiple times for the same set of data. The **GET**, **PUT**, and **DELETE** methods are idempotent, although it is quite possible to get a **404** error code if we try to delete the same resource the second time. However, this is acceptable behavior from an idempotence point of view.

## Using Flask for a REST API

A REST API in Python can be built using different libraries and frameworks. The most popular frameworks for building a REST API are Django, Flask (using the **Flask-RESTful** extension), and **FastAPI**. Each of these frameworks has merits and demerits. Django is a suitable choice for building a REST API if the web application is also being built using Django. However, using Django only for API development would be overkill. The Flask-RESTful extension works seamlessly with a Flask web application. Both Django and Flask have strong community support, which is sometimes an

important factor when selecting a library or a framework. FastAPI is considered the best in performance and is a good choice if the goal is only to build a REST API for your application. However, community support for FastAPI is not at the same level as we have for Django and Flask.

We selected a Flask RESTful extension for REST API development for the continuation of our discussion that we had started for the web application development. Note that we can build a simple web API using just Flask, and we have done this in the previous chapter when we developed a sample web service-based application for Google Cloud deployment. In this section, we will focus on using the REST architectural style in building the API. This means that we will use the HTTP method to perform an operation on a resource that will be represented by a Python object.

## **IMPORTANT**

*Flask-RESTful support is unique in the way that it provides a convenient way to set the response code and response header as a part of return statements.*

To use Flask and the Flask-RESTful extension, we will be required to install the Flask-RESTful extension. We can install it in our virtual environment by using the following **pip** command:

```
pip install Flask-RESTful
```

Before discussing how to implement a REST API, it is beneficial to get familiar with a few terms and concepts related to API.

## **Resource**

A resource is a key element for a REST API, and this is powered by the Flask-RESTful extension. A resource object is defined by extending our class from the base **Resource** class, which is available from the Flask-RESTful extension library. The base **Resource** class offers several magic functions to assist API development and automatically associates HTTP methods with Python methods defined in our resource object.

## **API endpoint**

An API endpoint is a point of entry to establish communication between client software and server software. In simple terms, an API endpoint is alternative terminology for a URL of a server or service where a program is listening for API requests. With the Flask-RESTful extension, we define an API endpoint by associating a certain URL (or URLs) with a resource object. In Flask implementation, we implement a resource object by extending from the base **Resource** class.

## **Routing**

The concept of routing for API is similar to web application routing in Flask, with the only difference being that in the case of an API, we need to map a **Resource** object to one or more endpoint URLs.

## **Argument parsing**

The parsing of request arguments for an API is possible by using the query string or HTML form-encoded data. However, this is not a preferred approach because both the query string or HTML forms are not meant or designed to be used with an API. The recommended approach is to extract the arguments directly from an HTTP request. To facilitate this, the Flask-RESTful extension offers a special class, **reqparse**. This **reqparse** class is similar to **argparse**, which is a popular choice for parsing command-line arguments.

Next, we will learn about building a REST API for accessing data from a database system.

## Developing a REST API for database access

To illustrate the use of Flask and the Flask-RESTful extension for building a REST API, we will revise our web application (**app5.py**) and will offer access to the **Student** object (a **Resource** object) using the REST architectural style. We are expecting the arguments sent for the **PUT** and **POST** methods inside the request body and the API will send back the response in JSON format. The revised code of **app5.py** with a REST API interface is shown next:

```
#api_app.py: REST API application for student resource
from flask_sqlalchemy import SQLAlchemy
from flask import Flask
from flask_restful import Resource, Api, reqparse
app = Flask(__name__)
api = Api(app)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///student.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)
```

In the preceding code snippet, we started with the initialization of the Flask application and the database instance. As a next step, we created an API instance using the Flask instance. We achieved this by means of the **api = Api(app)** statement. This API instance is the key to develop the rest of the API application and we will use it.

Next, we need to configure the **reqparse** instance by registering the argument we are expecting to parse from the HTTP request. In our code example, we registered two arguments of the string type, **name** and **grade**, as shown in the following code snippet:

```
parser = reqparse.RequestParser()
parser.add_argument('name', type=str)
parser.add_argument('grade', type=str)
```

The next step is to create a **Student** model object, which is mostly the same as we did in **app5.py**, except that we will add a **serialize** method to convert our object into JSON format. This is an important step in serializing the JSON response before sending it back to the API clients. There are other solutions available to achieve the same, but we selected this option for simplicity reasons. The precise sample code for the creation of a **Student** object is as follows:

```
class Student(db.Model):

 id = db.Column(db.Integer, primary_key=True)
 name = db.Column(db.String(80), nullable=False)
 grade = db.Column(db.String(20), nullable=True)

 def serialize(self):
 return {
 'id': self.id,
 'name': self.name,
 'grade': self.grade
 }
```

Next, we created two **Resource** classes to access student database objects. These are **StudentDao** and **StudentListDao**. These are described next:

- **StudentDao** offers methods such as **get** and **delete** on the individual resource instance, and these methods are mapped to the **GET** and **DELETE** methods of the HTTP protocol.
- **StudentListDao** offers methods such as **get** and **post**. The **GET** method is added to provide a list of all resources of the **Student** type using the **GET** HTTP method, and the **POST** method is included to add a new resource object using the **POST** HTTP method. This is a typical design pattern used to implement CRUD functionality for a web resource.

As regards methods implemented for **StudentDao** and **StudentListDao** classes, we returned the status code and the object itself in a single statement. This is a convenience offered by the Flask-RESTful extension.

A sample code for the **StudentDao** class is shown next:

```
class StudentDao(Resource):

 def get(self, student_id):
 student = Student.query.filter_by(id=student_id).\
 first_or_404(description='Record with id={} is not
available'.format(student_id))

 return student.serialize()

 def delete(self, student_id):
 student = Student.query.filter_by(id=student_id).\
 first_or_404(description='Record with id={} is not
available'.format(student_id))

 db.session.delete(student)
 db.session.commit()
```

```

 first_or_404(description='Record with id={} is not
available'.format(student_id))

 db.session.delete(student)

 db.session.commit()

 return '', 204

```

A sample code for the **StudentListDao** class is as follows:

```

class StudentListDao(Resource):

 def get(self):
 students = Student.query.all()
 return [Student.serialize(student) for student in
 students]

 def post(self):
 args = parser.parse_args()
 name = args['name']
 grade = args['grade']
 student = Student(name=name, grade=grade)
 db.session.add(student)
 db.session.commit()
 return student, 200

```

For our **post** method of the **StudentListDao** class, we used the **reqparse** parser to extract the name and the grade arguments from the request. The rest of the implementation in the **POST** method is the same as we did for the **app5.py** example.

In the next two lines of our sample API application, we mapped URLs to our **Resource** objects. All requests coming for **/students/<student\_id>** will be redirected to the **StudentDao** resource class. Any request coming for **/students** will be redirected to the **StudentListDao** resource class:

```

api.add_resource(StudentDao, '/students/<student_id>')

api.add_resource(StudentListDao, '/students')

```

Note that we skipped the **PUT** method implementation from the **StudentDao** class, but it is available with the source code provided for this chapter for completeness. For this code example, we did not add error and exception handling to keep the code concise for our discussion, but it is highly recommended to have this included in our final implementation.

In this section, we have covered the base concepts and implementation principles for developing REST APIs that are adequate for anyone to start building a REST API. In the next section, we will extend our knowledge to build a complete web application based on a REST API.

## Case study– Building a web application using the REST API

In this chapter, we learned how to build a simple web application using Flask and how to add a REST API to a business logic layer using a Flask extension. In the real world, web applications are typically three tiers: *web layer*, *business logic layer*, and *data access layer*. With the popularity of mobile apps, the architecture has evolved to have a REST API as a building block for the business layer. This affords the freedom of building web apps and mobile apps using the same business logic layer. Moreover, the same API can be available for B2B interactions with other vendors. This type of architecture is captured in *Figure 10.2*:

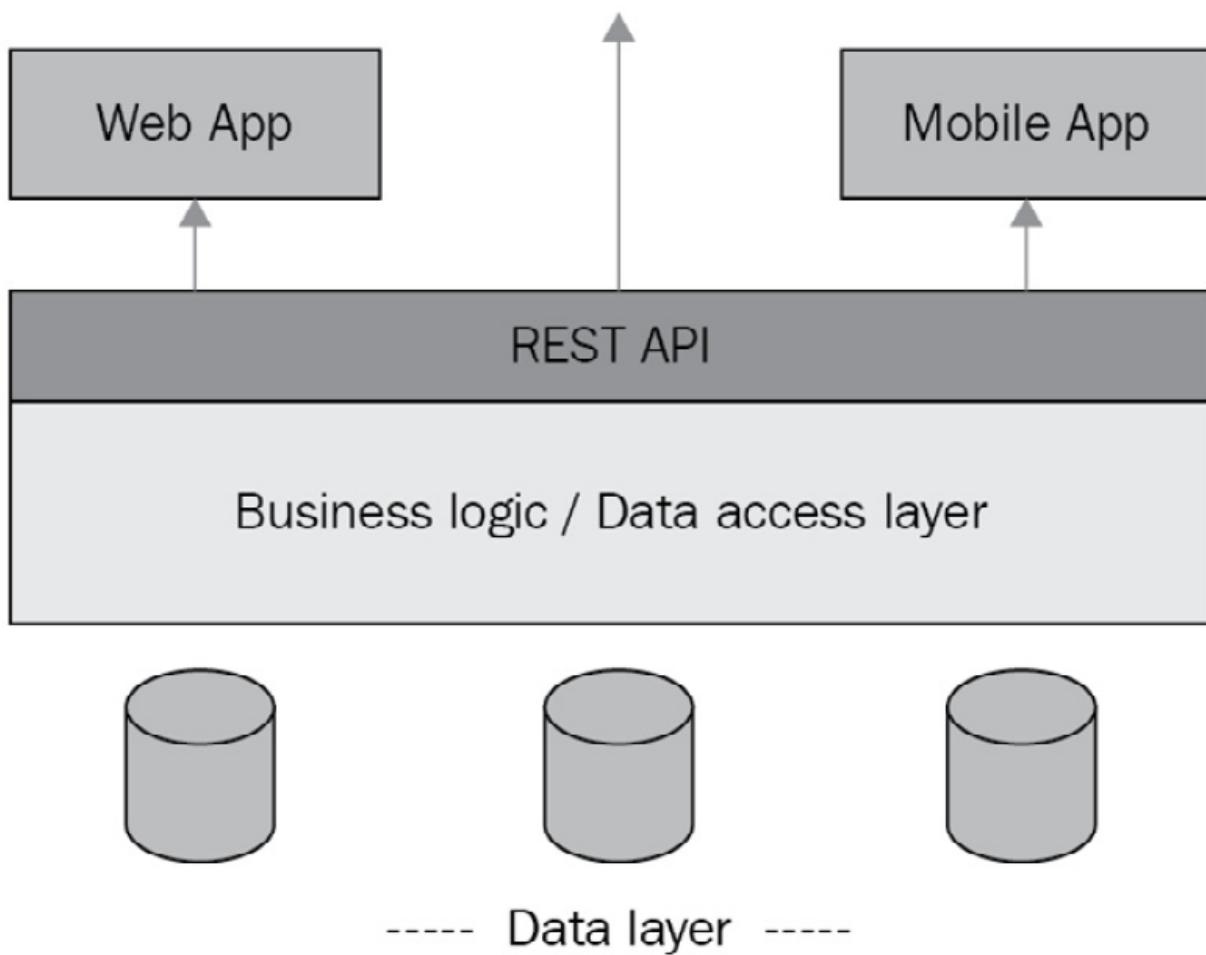


Figure 10.2 – Web/mobile application architecture

In our case study, we will develop a web application on top of the REST API application that was developed for the **Student** model object in the previous code example. At a high level, we will have the components in our application as shown here:

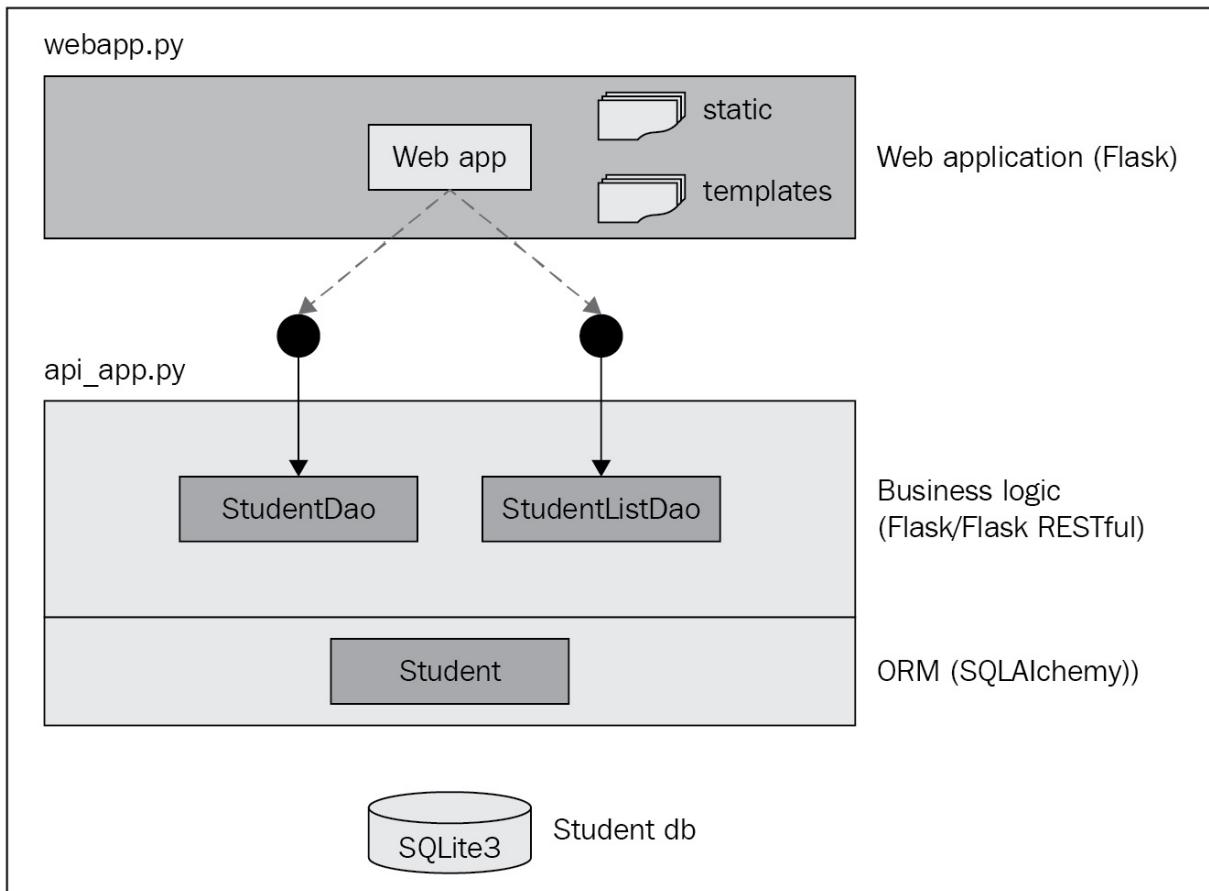


Figure 10.3 – Sample web application with a REST API as a backend engine

We already developed a business logic layer and data access layer (ORM) and exposed the functionality through two API endpoints. This is discussed in the *Using Flask for a REST API* section. We will develop a web application part for web access and consume the API offered by the business logic.

The **webapp.py** web application will be based on Flask. The **webapp.py** application (referred to as *webapp* moving forward) will be independent of the **api\_app.py** application (referred to as *apiapp* moving forward) in the sense that the two applications will be running separately as two Flask instances, ideally on two separate machines. But if we are running the two Flask instances on the same machine for testing purposes, we must use different ports and use the local machine IP address as a host. Flask uses the *127.0.0.1* address as a host to run its built-in web server, which may not be allowed for running two instances. The two applications will talk to each other only via a REST API. Additionally, we will develop a few Jinja templates to submit requests for create, update, and delete operations. We will reuse the **api\_py.py** application code as is, but we will develop the **webapp.py** application with features such as listing students, adding a new student, deleting a student, and updating a student's data. We will add Python functions for each feature one by one:

1. We will begin with the initialization of the Flask instance as we have done for previous code examples. The sample code is as follows:

```
#webapp.py: interacting with business latyer via REST API

for create, delete and list objects

from flask import Flask, render_template, redirect, request
import requests, json

app = Flask(__name__)
```

2. Next, we will add a **list** function to handle requests with the / URL as follows:

```
@app.get('/')
def list():

 response = requests.get('http://localhost:8080 /students')
 data = json.loads(response.text)

 return render_template('main.html', students=data)
```

In all our Python functions, we are using the **requests** library to send a REST API request to the **apiapp** application that is hosted on the same machine in our test environment.

3. Next, we will implement an **add** function to process the request for adding a new student to the database. Only the request with the **POST** method type is mapped to this Python function. The sample code is as follows:

```
@app.post('/')
def add():

 fname = request.form['fname']
 lname = request.form['lname']
 grade = request.form['grade']

 payload = {'name': f'{fname} {lname}', 'grade': grade}
 response = requests.post('http://localhost:8080 /students', data=payload)
 return redirect("/")
```

Note that for the post API call to our **apiapp** application, we built the **payload** object and passed it as a **data** attribute to the **POST** method of the **requests** module.

4. Next, we will add a **DELETE** function to handle a request for deleting an existing student. The request type mapped to this method is expected to provide the student ID as part of the URL.

```
@app.get('/delete/<int:id>')
def delete(id):

 response = requests.delete('http://localhost:8080 /students/'+str(id))
 return redirect("/")
```

5. Next, we will add two functions to handle the update feature. One function (**update**) is used to update the data of a student in the same manner as the **post** function does. But before triggering the **update** function, our webapp application will offer a form to the

user with the current data of a **student** object. The second function (**load\_student\_for\_update**) will get a **student** object and send it to a Ninja template for users to edit. The code for both functions is as follows:

```
@app.post('/update/<int:id>')
def update(id):
 fname = request.form['fname']
 lname = request.form['lname']
 grade = request.form['grade']
 payload = {'name' : f'{fname} {lname}', 'grade':grade}
 response = requests.put('http://localhost:8080 /students/' + str(id), data =
payload)
 return redirect("/")
@app.get('/update/<int:id>')
def load_student_for_update(id):
 response = requests.get('http://localhost:8080 /students/'+str(id))
 student = json.loads(response.text)
 fname = student['name'].split()[0]
 lname = student['name'].split()[1]
 return render_template('update.html', fname=fname, lname=lname, student=
student)
```

The code inside these functions is no different from what we already discussed in relation to previous examples. Therefore, we will not go into the details of every line of the code, but we will highlight the key points of this web application and its interaction with a REST API application:

- For our web application, we are using two Ninja templates (**main.html** and **update.html**). We are also using a template (we called it **base.html**) that is common to both templates. The **base.html** template is mainly built using the bootstrap UI framework. We will not discuss the details of the Ninja templates and the bootstrap, but we will encourage you to get familiar with both using the references provided at the end of this chapter. The sample Ninja templates with bootstrap code are available with the source code of this chapter.
- The root / URL of our web application will launch the main page (**main.html**), which allows us to add a new student and also provides a list of existing students. The following screenshot shows the screenshot of the main page, which will be rendered with our **main.html** template:

## Students



### Add a Student

First name

Last name

Grade

Submit

## Students

No	Name	Grade	Actions
1	John Lee	10	<button>Update</button> <button>Delete</button>
2	Brian Miles	7	<button>Update</button> <button>Delete</button>

Figure 10.4 – Main page of the webapp application

- If we add the first and last name of a student with a grade and click the **Submit** button, this will trigger a **POST** request with data of these three input fields. Our webapp application will delegate this request to the **add** function. The **add** function will use the corresponding REST API of the **apiapp** application to add a new student and the **add** function will render the main page again with an updated list of students (including new students).

- On the main webapp page (**main.html**), we added two buttons (**Update** and **Delete**) with each student record. On clicking the **Delete** button, the browser will trigger a **GET** request with the `/delete/<id>` URL. This request will be delegated to the **delete** function. The **delete** function will use the REST API of the **apiapp** application to delete the student from the **SQLite3** database and will render the main page again with an updated list of students.
- Upon clicking the **Update** button, the browser will trigger a **GET** request with the `/update/<id>` URL. This request will be delegated to the **load\_student\_for\_update** function. This function will first load the student data using the REST API of the **apiapp** application, set the data in the response, and render the **update.html** template. The **update.html** template will show the user an HTML form filled with student data to allow editing. The form we developed for the update scenario is shown here:

The image shows a web-based form titled "Update Student". At the top left is the word "Students" and a menu icon. The main title "Update Student" is centered above three input fields: "First name" containing "John", "Last name" containing "Lee", and "Grade" containing "10". Below these fields is a large, prominent black button with the word "Update" in white.

Figure 10.5 – A sample form to update a student

After the changes, if a user submits the form by clicking the **Update** button, the browser will trigger a **POST** request with the `/update/<id>` URL. We have registered the **update** function for this request. The **update** function will extract data from the request and pass it to the REST API of the **apiapp** application. Once the student info is updated, we will render the **main.html** page again with an updated list of students.

In this chapter, we have omitted the details of pure web technologies such as HTML, Jinja, CSS, and UI frameworks in general. The beauty of web frameworks is that they allow any web technologies to be used for customer interfaces, especially if we are building our applications using a REST API.

This concludes our discussion of building web applications and developing a REST API using Flask and its extensions. Web development is not limited to one language or one framework. The core principles and architecture remain the same across web frameworks and languages. The web development principles you learned here will help you to understand any other web framework for Python or any other language.

## Summary

In this chapter, we discussed how to use Python and web frameworks such as Flask for developing web applications and a REST API. We started the chapter by analyzing the requirements for web development, which include a web framework, a UI framework, a web server, a database system, API support, security, and documentation. Later, we introduced how to use the Flask framework for building web applications with several code examples. We discussed different request types with different HTTP methods and how to parse the request data with relevant code examples. We also learned about the use of Flask to interact with a database system using an ORM library such as SQLAlchemy. In the latter part of the chapter, we introduced the role of a web API for web applications, mobile applications, and business-to-business applications. We investigated a Flask extension to develop a REST API with a detailed analysis by using a sample API application. In the last section, we discussed a case study of developing a *student* web application. The web application is built using two independent applications, both running as Flask applications. One application offers a REST API for the business logic layer on top of a database system. The other application provides a web interface to users and consumes the REST API interface of the first application to provide access to the student resource objects.

This chapter provides extensive hands-on knowledge of building web applications and a REST API using Flask. The code examples included in this chapter will enable you to start creating web applications and write a REST API. This knowledge is critical for anyone who is seeking a career in web development and working in building a REST API.

In the next chapter, we will explore how to use Python to develop microservices, a new paradigm of software development.

## Questions

1. What is the purpose of TLS?
2. When is Flask a superior choice to the Django framework?
3. What are the commonly used HTTP methods?
4. What is CRUD and how is it related to a REST API?
5. Does a REST API only use JSON as a data format?

## Further reading

- *Flask Web Development*, by Miguel Grinberg
- *Advanced Guide to Python 3 Programming*, by John Hunt
- *REST APIs with Flask and Python*, by Jason Myers & Rick Copeland
- *Essential SQLAlchemy, 2nd Edition*, by Jose Salvatierra
- *Bootstrap 4 Quick Start*, by Jacob Lett
- *Jinja online documentation*, available at <https://jinja.palletsprojects.com/>

## Answers

1. The main purpose of TLS is to provide encryption of data that is exchanged between the two systems on the internet.
2. Flask is a better choice for small- to medium-sized applications and especially when the project requirements are expected to change frequently.
3. **GET** and **POST**.
4. **CRUD** stands for **Create**, **Read**, **Update**, and **Delete** operations in software development. From an API perspective, each CRUD operation is mapped to one of the HTTP methods (**GET**, **POST**, **PUT**, and **DELETE**).
5. A REST API can support any string-based format, such as JSON, XML, or HTML. Data format support is more related to HTTP's ability to carry the data as part of the HTTP body element.

## *Chapter 11: Using Python for Microservices Development*

Monolithic applications that are built as a single-tiered software have been a popular option for developing applications for many years. However, it is not efficient to deploy monolithic applications in cloud platforms in terms of reserving and utilizing resources. This is even true for the deployment of large-scale monolithic applications on physical machines. The maintenance and development costs of such applications are always high. Multi-tier applications have solved this problem to some extent for web applications by breaking the applications down into several tiers.

To meet dynamic resource demands and to reduce development/maintenance costs, the real savior is a **microservices architecture**. This new architecture encourages applications to be built on loosely coupled services and deployed on dynamically scalable platforms such as containers. Organizations such as Amazon, Netflix, and Facebook have already moved from a monolithic model to microservices-based architecture. Without this change, these organizations couldn't have served a large number of clients.

We will cover the following topics in this chapter:

- Introducing microservices
- Learning best practices for microservices
- Building microservices-based applications

After completing this chapter, you will learn about microservices and will be able to build applications based on microservices.

## Technical requirements

The following are the technical requirements for this chapter:

- You need to have Python 3.7 or later installed on your computer.
- Python Flask library with RESTful extensions installed on top of a Python 3.7 or later release.
- Python Django with Django Rest Framework library on top of a Python 3.7 or later release.
- You need to have a Docker registry account and installed Docker Engine and Docker Compose on your machine.
- To deploy a microservice in GCP Cloud Run, you will need a GCP account (a free trial will work fine).

Sample code for this chapter can be found at <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter11>.

We will start our discussion with the introduction of microservices.

## Introducing microservices

A microservice is an independent software entity that must have the following characteristics:

- **Loosely coupled** with other services and without any dependency on other software components
- **Easy to develop** and **Maintain** by a small team without depending on other teams
- **Independently installable** as a separate entity, preferably in a container
- Offers **easy-to-consume** interfaces using synchronous protocols such as REST APIs, or asynchronous protocols such as **Kafka** or **RabbitMQ**

The keywords in terms of software being called a microservice are independently deployable, loosely coupled, and easily maintainable. Each microservice can have its own database servers to avoid sharing resources with other microservices. This will ensure the elimination of dependencies between the microservices.

The microservices architecture style is a software development paradigm for developing applications using purely microservices. This architecture even includes the main interface entity of the application, such as a web application. An example of a microservices-based application is illustrated next:

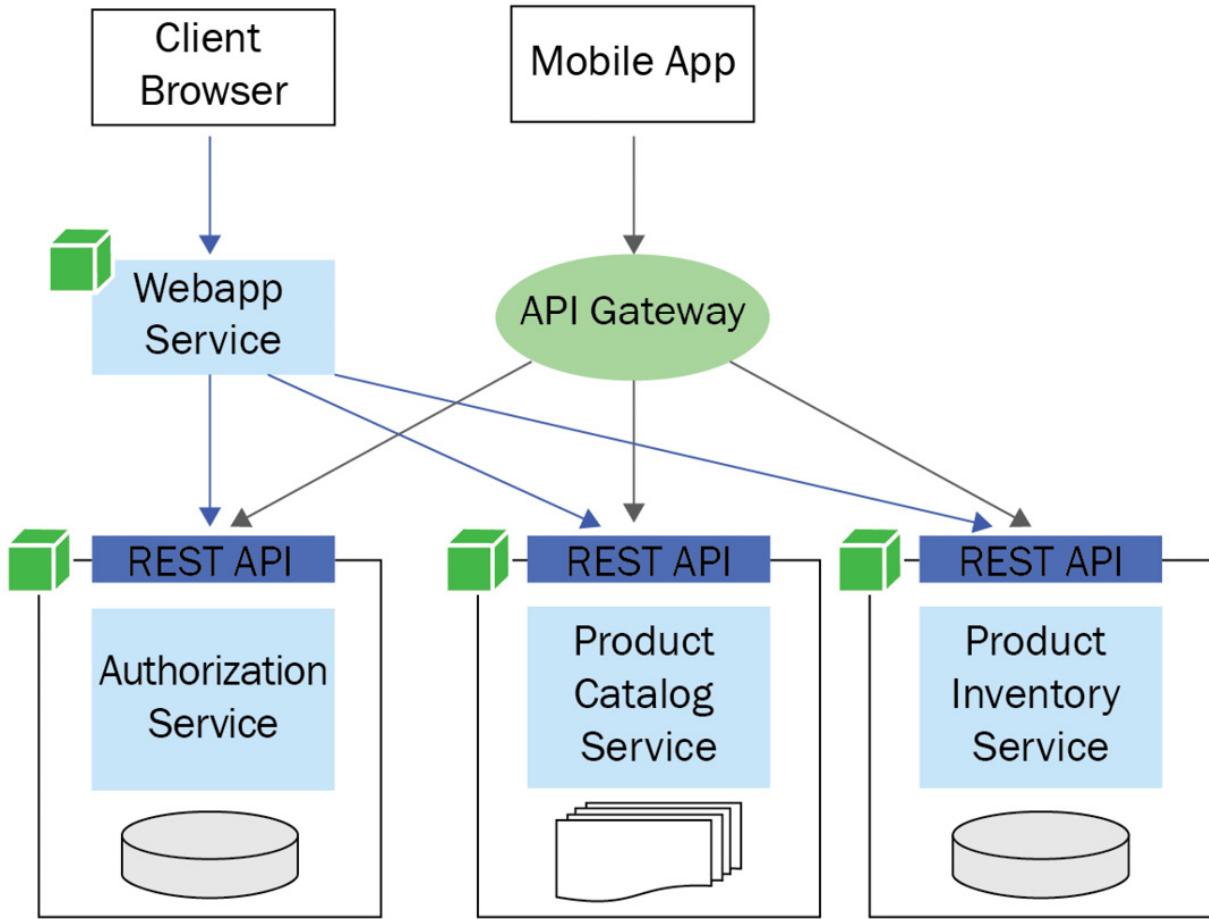


Figure 11.1 – A sample microservices architecture style application

In this example application, we have individual microservices such as **Authorization Service**, **Product Catalog Service**, and **Product Inventory Service**. We built a web application, also as a microservice, that uses the three individual microservices through the REST API. For mobile clients, a mobile app can be built using the same individual microservices through an API gateway. We can see an immediate advantage of microservices architecture, which is reusability. A few other advantages of using microservices are as follows:

- We are flexible when it comes to selecting any technology and any programming language that suits any individual microservice requirements. We can even reuse legacy code written in any language if we can expose it using an API interface.
- We can develop, test, and maintain individual microservices by small independent teams. It is crucial to have independent and autonomous small teams for the development of large-scale applications.
- One of the challenges with monolithic applications is the managing of conflicting library versions, which we are forced to include because of different features bundled in a single application. With microservices, the chances of conflict as regards the versions of these libraries are minimized.
- We can deploy and patch the individual microservices independently. This enables us to use **Continuous Integration/Continuous Delivery (CI/CD)** for complex applications. This is also important when we need to apply a patch or upgrade one feature of an

application. For monolithic applications, we will redeploy the entire application, which means there are chances to break other parts of the application. With microservices, only one or two services will be redeployed without risks of breaking anything else in other microservices.

- We can isolate faults and failures on a microservice level instead of the application level. If there is a fault or failure with one service, we can debug it, fix it, and patch it or stop it for maintenance without impacting the rest of the application's functionality. In the case of monolithic applications, a problem in one component can bring down the entire application.

Despite several advantages, there are some disadvantages associated with using the microservices architecture style:

- First is the increased complexity of creating microservices-based applications. The complexity mainly arises from the fact that each microservice has to expose an API and the consumer service or program has to interact with microservices using an API. Security on a per microservice basis is another contributor to the complexity.
- The second disadvantage is the increased resource requirements as compared to monolithic applications. Every microservice requires additional memory to be hosted independently in a container or a virtual machine, even if it is a **Java Virtual Machine (JVM)**.
- The third disadvantage is that additional efforts are required to debug and troubleshoot a problem across different microservices that may be deployed in separate containers or systems.

Next, we will study the best practice of building microservices.

## Learning best practices for microservices

When starting a new application, the first and foremost question we should be asking ourselves is whether the microservices architecture is a good fit. This starts with an analysis of the application requirements and the ability to divide the requirements into separate and individual components. If you see that your components frequently depend on one another, this is an indicator that the segregation of components may require reworking, or that this application may not be a fit for the microservices architecture.

It is important to make this decision of whether to use microservices right in the early phase of an application. There is a school of thought that says that it is better to start building an application using monolithic architecture to avoid the additional costs of microservices in the beginning. However, this is not an advisable approach. Once we have built a monolithic application, it is difficult to transform it into microservices architecture, especially if the application is already in production. Companies such as Amazon and Netflix have done it, but they did it as part of their evolution of technology and certainly, they have the human and technology resources available to undertake this transformation.

Once we have made our decision to build our next application using microservices, the following best practices will guide you in making design and deployment decisions:

- **Independent and loosely coupled:** These requirements are part of the microservices definition. Each microservice should be built independently from the other microservices and is built as loosely coupled as possible.

- **Domain-Driven Design (DDD):** The purpose of microservices architecture is not to have as many small microservices as possible. We need to always remember that each microservice has its overhead costs. We should build as many microservices as the business or domain requires. We recommend considering DDD, which was introduced by *Eric Evans* in 2004.

If we try to apply DDD to microservices, it suggests having a strategic design first to define different contexts by combining related business domains and their subdomains. The strategic design can be followed by a tactical design that focuses on breaking down the core domains into fine-grained building blocks and entities. This breakdown will provide clear guidelines to map the requirements to possible microservices.

- **Communication interfaces:** We should use well-defined microservice interfaces, preferably a REST API or an event-driven API, for communication. Microservices should avoid calling each other directly.
- **Use an API gateway:** Microservices and their consumer applications should interact with individual microservices using an API gateway. The API gateway can take care of security aspects, such as authentication and load balancing, out of the box. Moreover, with a new version of a microservice, we can use the API gateway to redirect the client requests to the newer version without impacting the client-side software.
- **Limit technology stack:** Although microservice architecture allows the use of any programming language and framework on a per-service basis, it is not advisable to develop microservices using different technologies in the absence of any business or reusability reasons. A diverse technology stack may be appealing for academic reasons, but it will bring operational complexity in maintaining and troubleshooting the application.
- **Deployment model:** It is not mandatory to deploy a microservice in a container, but it is recommended. Containers bring a lot of built-in features, such as automated deployment, cross-platform support, and interoperability. Additionally, by using containers, we can assign the resources to the service as per its requirements and ensure a fair distribution of resources across different microservices.
- **Version control:** We should be using a separate versioning control system for each microservice.
- **Team organization:** Microservices architecture provides an opportunity to have dedicated teams on a per-microservice basis. We should be keeping this principle in mind when organizing teams for a large-scale project. The size of a team should be based on the *two-pizza philosophy*, which states that we should set up a team with as many engineers who can be fed by two large pizzas. A team can own one or more microservices based on their complexity.
- **Centralized logging/monitoring:** As discussed previously, troubleshooting problems in a microservice architecture-style application can be time-consuming, especially if the microservices are running in containers. We should be using open source or professional tools to monitor and troubleshoot the microservices to reduce such operational costs. A few examples of such tools are **Splunk**, **Grafana**, **Elk**, and **App Dynamics**.

Now that we have covered the introduction as well as best practices of microservices, next, we will deep dive into learning to build an application using microservices.

## Building microservices-based applications

Before going into the implementation details of microservices, it is important to analyze several microservice frameworks and deployment options. We will start with a microservices framework available in Python.

# Learning microservice development options in Python

In Python, we have a plethora of frameworks and libraries available for microservice development. We cannot enumerate all the available options, but it is worth highlighting the most popular and those that have some different feature sets. These options are described next:

- **Flask:** This is a lightweight framework that can be used to build **Web Service Gateway Interface (WSGI)**-based microservices. Note that WSGI is based on a request-response synchronous design pattern. We already used Flask and its RESTful extension to build a REST API application in [Chapter 10, Using Python for Web Development and REST API](#). Since Flask is a popular web and API development framework, it is an easy adoption choice for many developers who are already using Flask.
- **Django:** Django is another popular web framework with a large community of developers. With its **Django Rest Framework (DRF)**, we can build microservices with REST API interfaces. Django offers both WSGI and **Asynchronous Service Gateway Interface (ASGI)**-based microservices. ASGI is considered a successor to the WSGI interface. ASGI is an excellent option if you are interested in developing your application based on **Asyncio**, a topic we discussed in detail in [Chapter 7, Multiprocessing, Multithreading, and Asynchronous Programming](#).
- **Falcon:** This is also a popular web framework after Flask and Django. It does not come with a built-in web server, but it is well optimized for microservices. Like Django, it supports both ASGI and WSGI.
- **Nameko:** This framework is specifically designed for microservice development in Python, and is not a web application framework. Nameko comes with built-in support for **Remote Procedure Call (RPC)**, asynchronous events, and WebSocket-based RPCs. If your application requires any of these communication interfaces, you should consider Nameko.
- **Bottle:** This is a super lightweight WSGI-based microservices framework. The whole framework is based on a single file and leverages only a Python standard library for its operations.
- **Tornado:** It is based on non-blocking network I/O. Tornado can handle high traffic with extremely low overheads. This is also a suitable choice for long-polling and WebSocket-based connections.

For the development of our sample microservices, we can use any of the frameworks mentioned earlier. But we will use Flask and Django for two reasons. First, these two are the most popular for developing web applications and microservices. Second, we will reuse an example API application that we developed in the previous chapter. A new microservice will be developed using Django and it will illustrate how to use Django for web and API development.

Next, we will discuss microservice deployment options.

## Introducing deployment options for microservices

Once we write microservices, the next question is how to deploy them as an isolated and independent entity. For the sake of discussion, we will assume that microservices are built with HTTP/REST interfaces. We can deploy all microservices on the same web server as different web apps or host one web server for one microservice. One microservice in a separate web server can be deployed on a single machine (physical or virtual) or on separate machines or even on separate containers. We have summarized all these different deployment models in the following diagram:

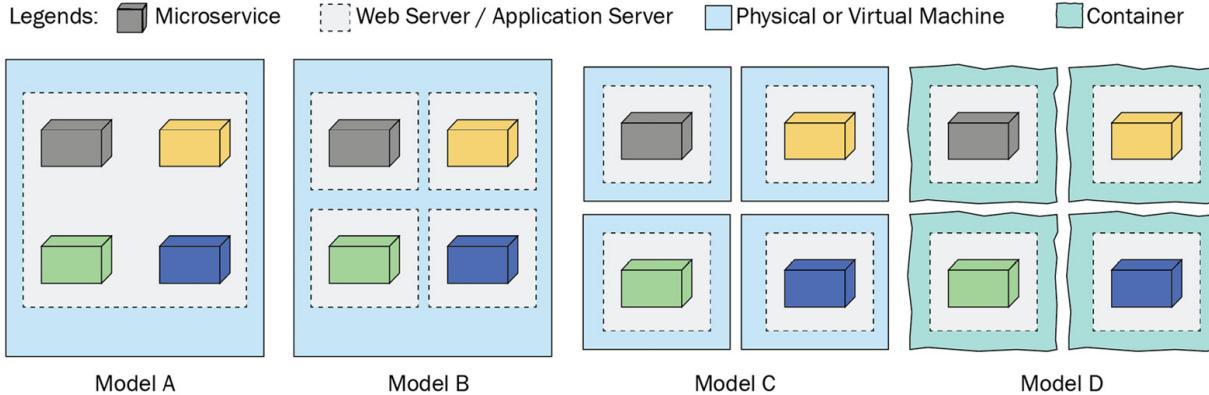


Figure 11.2 – Microservices deployment models

The four deployment models shown in *Figure 11.2* are described as follows:

- **Model A:** In this model, we are deploying four different microservices on the same web server. There is a good chance that the microservices in this case are sharing libraries, being on a single web server. This may result in library conflicts and is not a recommended model for deploying microservices.
- **Model B:** For this model, we deployed four microservices on a single machine, but using one microservice per web server to make them independent. This model is fine for development environments, but may not be suitable on a production scale.
- **Model C:** This model is using four virtual machines to host four different microservices. Each machine host only one microservice with a web server. This model is suitable for production if it is not possible to use containers. The main caveat of this model is the additional costs because of resource overheads that each virtual machine will bring with it.
- **Model D:** In this model, we deploy each microservice as a container on a single machine or across multiple machines. This is not only cost-effective but also provides an easy way to be compliant with microservice specifications. This is the recommended model whenever it is feasible to use.

We analyzed different deployment models to understand which option is more appropriate than others. For the development of our sample microservices-based application, we will use a mix of both a container-based microservice and a microservice hosted only on a web server. This mixed model illustrates that we can use any option technically, although the container-based deployment is a recommended one. Later, we will take one of our microservices to the cloud to demonstrate the portability of microservices.

After discussing the development and deployment options for microservices, it is time to start building an application using two microservices in the next section.

## Developing a sample microservices-based application

For the sample application, we will develop two microservices and a web application using Flask and Django frameworks. Our sample application will be an extension of the **Student** web application that

was developed as a case study in the previous chapter. The application architecture will appear as shown here:

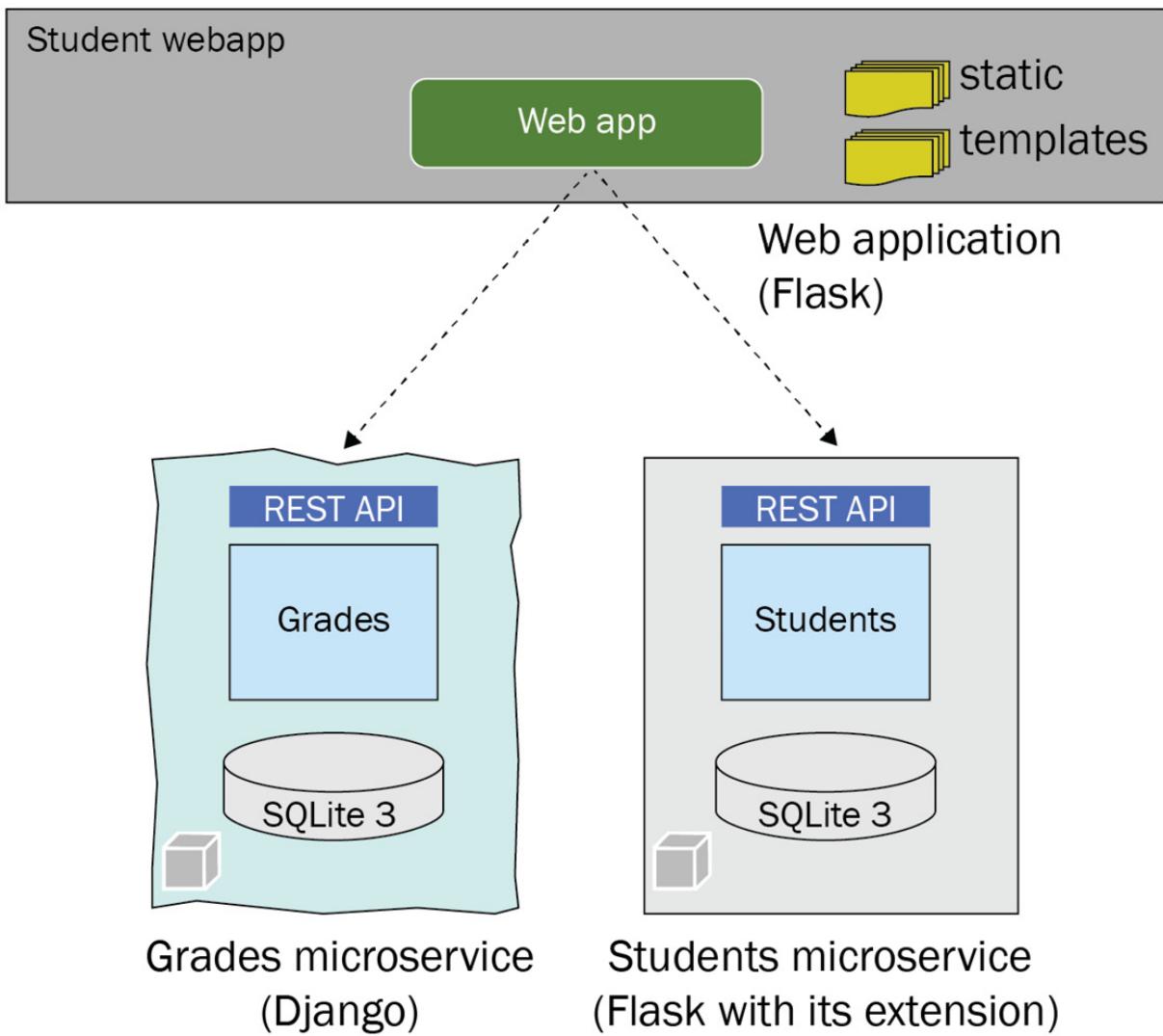


Figure 11.3 – C microservices-based architecture of a sample application

To develop this sample application, we will develop the following components:

- Build a new **Grades** microservice using the Django framework and deploy it with a **Docker Engine**. Docker Engine is an open source software to containerize our application. The Grades microservice will provide additional information about each grade, such as the building name and class teacher name, and these attributes are not stored with the **Student** model under the Students microservice.
- Reuse the **apiapp** application from the previous chapter. It is named as a **Students** microservice for this sample application. There will be no change in the code of this application/module.
- We will update the **webapp** application from the previous chapter's case study to consume the **Grades** microservice and add additional **Grade** attributes with each **Student** object. This will require minor updates to the Jinja templates as well.

We will start by building the **Grades** microservice with Django.

## Creating a Grades microservice

To develop a microservice using Django, we will use **Django Rest Framework (DRF)**. Django uses various components from its web framework to build a REST API and microservices. Hence, this development exercise will also give you a high-level idea about developing web applications using Django.

Since we started with Flask and are already familiar with the core web concepts of web development, it will be a convenient transition for us to start using Django. Let's now understand the steps involved:

1. First things first, we will create a project directory or create a new project in our favorite IDE with a virtual environment. If you are not using an IDE, you can create and activate a virtual environment under your project directory using the following commands:

```
python -m venv myenv
source myenv/bin/activate
```

For any web application, it is vital to create a virtual environment for each application. Using a global environment for library dependencies can result in such errors that are hard to troubleshoot.

2. To build a Django application, we will require at least two libraries that can be installed using the following **pip** commands:

```
pip install django
pip install django-rest-framework
```

3. Once we have installed Django, we can use the **django-admin** command-line utility to create a Django project. The command shown next will create a Django **grades** project for our microservice:

```
django-admin startproject grades
```

This command will create an **admin** web app under the **grades** directory and add a **manage.py** file to our project. The **admin** web app includes built-in web server launch scripts, a settings file, and a URL routing file. **manage.py** is also a command-line utility, like **django-admin**, and offers similar features but in the context of a Django project. The file structure of the project directory will look as follows when we create a new Django project:

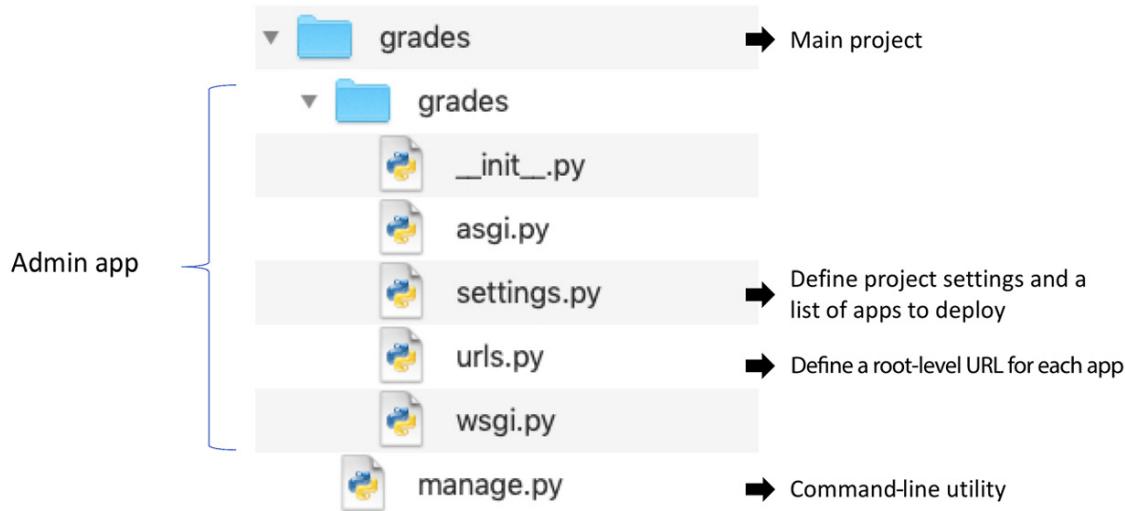


Figure 11.4 – File structure of a new Django project

As shown in *Figure 11.4*, the **settings.py** file contains a project-level setting including a list of apps to deploy with the web server. The **urls.py** file contains routing information for different deployed applications. Currently, only the **admin** app is included in this file. **asgi.py** and **wsgi.py** are available to launch the ASGI or WSGI web server, and the option of which one to use is set in the **settings.py** file.

4. The next step is to create a new Django application (our **Grades** microservice) by using the following command under the main **grades** project directory:

```
python3 manage.py startapp grades_svc
```

This command will create a new application (with web components) in a separate directory with the same name as we gave this command, **grades\_svc**. This will also create a default **SQLite3** database instance. The option of using the default **SQLite3** database is available in the **settings.py** file, but it can be changed if we decide to use any other database.

5. In addition to the files created automatically in the **grades\_svc** directory, we will add two more files – **urls.py** and **serializers.py**. A complete project directory structure with two additional files is shown in *Figure 11.5*. The roles of different files relevant to our project are also elaborated in this diagram:

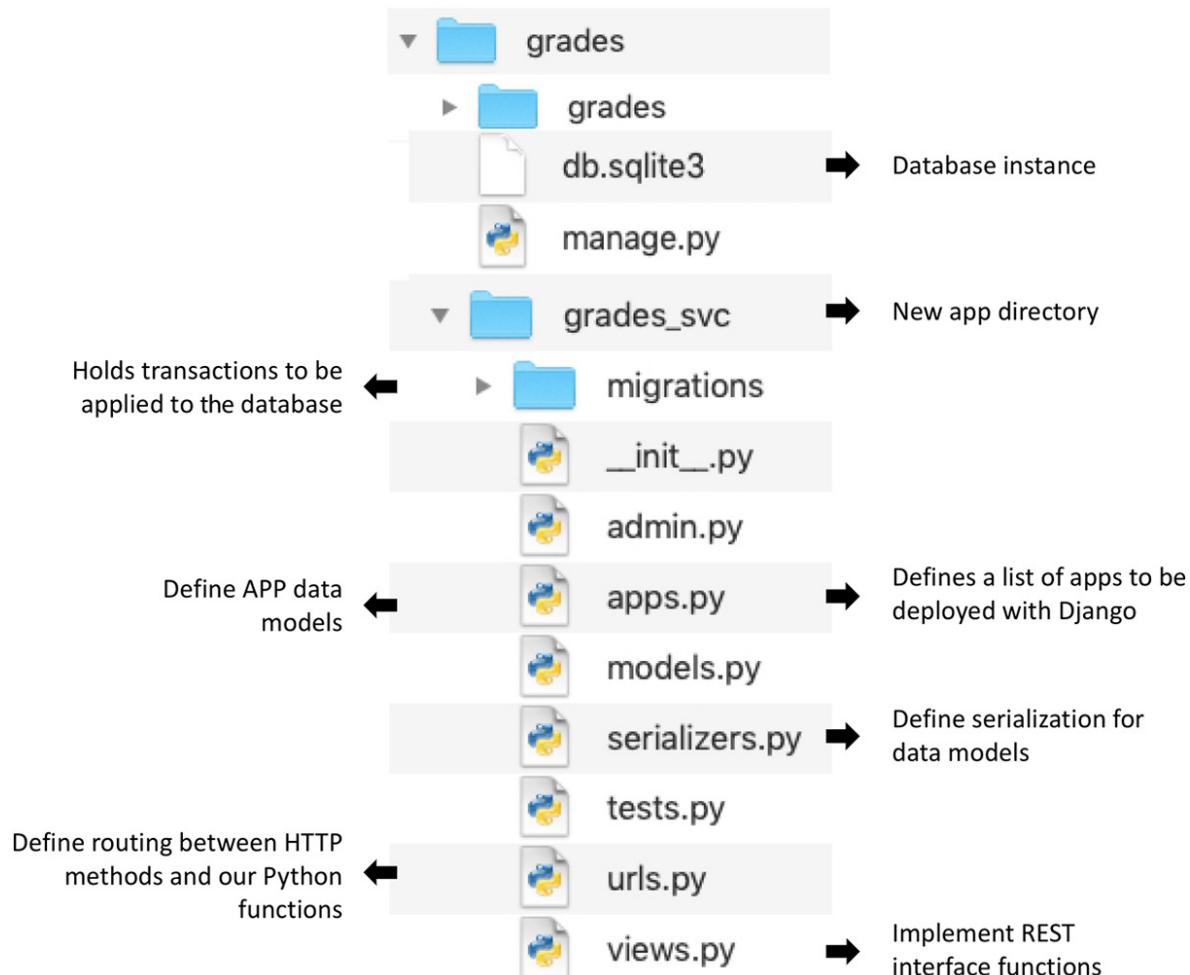


Figure 11.5 – Full directory structure with the grades\_svc app

6. Next, we will add the necessary code for our microservice in these files one by one. We will start by defining our **Grade** model class by extending the **Model** class from the Django database **models** package. The complete code of the **models.py** file is as follows:

```
from django.db import models

class Grade(models.Model):

 grade_id = models.CharField(max_length=20)
 building = models.CharField(max_length=200)
 teacher = models.CharField(max_length=200)

 def __str__(self):
 return self.grade_id
```

7. To make our model visible in the **admin** app dashboard, we need to register our model **Grade** class in the **admin.py** file as follows:

```
from django.contrib import admin
from .models import Grade
```

```
admin.site.register(Grade)
```

8. Next, we will implement a method to retrieve a list of **Grade** objects from the database. We will add a **GradeViewSet** class by extending **ViewSet** in the **views.py** file as follows:

```
from rest_framework import viewsets, status
from rest_framework.response import Response
from .models import Grade
from .serializers import GradeSerializer
class GradeViewSet(viewsets.ViewSet):
 def list(self, request):
 grades_list = Grade.objects.all()
 serializer = GradeSerializer(grades_list, many=True)
 return Response(serializer.data)
 def create(self, request):
 pass:
 def retrieve(self, request, id=None):
 pass:
```

Note that we also added methods for adding a new **Grade** object and for getting a **Grade** object according to its ID in actual implementation for the completeness of our microservice. We are showing only the **list** method because this is the only method relevant for our sample application. It is also important to highlight that the view objects should be implemented as classes and we should avoid putting application logic in the view objects.

Once we implement our core methods under the **grades\_svc** application, we will add our application to the Django project for deployment and add routes at the application as well as at the API level:

1. First, we will add our **grades\_svc** app and also **rest-framework** to the list of **INSTALLED\_APPS** in the **settings.py** file as follows:

```
INSTALLED_APPS = [
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 'grades_svc',
 'rest_framework',
```

]

A common mistake made by developers is to keep adding new components to a single settings file, which is hard to maintain for a large project. The best practice is to split the file into multiple files and load them in the main settings file.

2. This will also ensure that our application is visible in the **admin** app. The next step is to add URL configuration at the **admin** app level and then at the application level. First, we will add the URL for our application in the **urls.py** file under the **admin** app as follows:

```
urlpatterns = [
 path('admin/', admin.site.urls),
 path('', include('grades_svc.urls')),
]
```

In the **urls.py** file of the admin app, we are redirecting every request to our microservice, except the one that comes with the **admin/** URL.

3. The next step is to set routes in our application based on different HTTP methods. This requires us to add the **urls.py** file to our **grades\_svc** directory with the following route definitions:

```
from django.urls import path
from .views import GradeViewSet
urlpatterns = [
 path(grades '/', GradeViewSet.as_view({
 'get': 'list', #relevant for our sample
 'post': 'create'
 })),
 path('grades/<str:id>', GradeViewSet.as_view({
 'get': 'retrieve'
 }))
]
```

In this file, we are attaching **GET** and **POST** methods of HTTP requests with the **grades/** URL to the **list** and **create** methods of the **GradeViewSet** class that we implemented in the **views.py** file earlier. Similarly, we attached the **GET** request with the **grades/<str:id>** URL to the **retrieve** method of the **GradeViewSet** class. By using this file, we can add additional URL mapping to the Python functions/methods.

This concludes our implementation of our **Grades** microservice. The next step is to run this service under the Django web server for validation. But before running the service, we will make sure that the model objects are transferred to the database. This is equivalent to initializing the database in the

case of Flask. In the case of Django, we run the following two commands to prepare the changes and then execute them:

```
python3 manage.py makemigrations
python3 manage.py migrate
```

Often, developers miss this important step and get errors when trying to start the application. So, ensure that all the changes are executed before we start the web server by using the following command:

```
python3 manage.py runserver
```

This will start a web server on a default port, **8000**, on our local host machine. Note that the default settings, including the database and web server with host and port attributes, can be changed in the **settings.py** file. Additionally, we will recommend setting up a user account for the **admin** app by using the following command:

```
python3 manage.py createsuperuser
```

This command will prompt you to select a username, email address, and password for the admin account. Once our microservice is performing the functions as expected, it is time to bundle it in a container and run it as a container application. This is explained in the next section.

## Containerizing a microservice

Containerization is a type of operating system virtualization in which applications are run in their separate user space, but sharing the same operating system. This separate user space is called a **container**. Docker is the most popular platform for creating, managing, and running applications as containers. Docker still holds more than an 80% market share, but there are other container runtimes such as **CoreOS rkt**, **Mesos**, **Ixc**, and **containerd**. Before using Docker to containerize our microservice, we will quickly review the main components of the Docker platform:

- **Docker Engine:** This is the core Docker application for building, packaging, and running container-based applications.
- **Docker image:** A Docker image is a file that is used to run the application in a container environment. The applications developed using Docker Engine are stored as Docker images, which are a collection of application code, libraries, resource files, and any other dependencies that are required for application execution.
- **Docker Hub:** This is an online repository of Docker images for sharing within your team and with the community as well. **Docker Registry** is another term used in the same context. Docker Hub is an official name of the Docker registry that manages Docker image repositories.
- **Docker Compose:** This is a tool for building and running container applications using a YAML-based file instead of using the CLI commands of Docker Engine. Docker Compose provides an easy way to deploy and run multiple containers with configuration attributes and dependencies. Therefore, we will recommend using Docker Compose or similar technology to build and run your containers.

To use Docker Engine and Docker Compose, you need to have an account with the Docker registry. Also, you must download and install Docker Engine and Docker Compose on your machine before starting the following steps:

1. As a first step, we will create a list of our project dependencies by using the **pip freeze** command file as follows:

```
pip freeze -> requirements.txt
```

This command will create a list of dependencies and export them to the **requirements.txt** file.

This file will be used by Docker Engine to download these libraries inside the container on top of a Python interpreter. The contents of this file in our project are as follows:

```
asgiref==3.4.1
Django==3.2.5
django-rest-framework==0.1.0
djangorestframework==3.12.4
pytz==2021.1
sqlparse==0.4.1
```

2. In the next step, we will build **Dockerfile**. This file will also be used by Docker Engine to create a new image of a container. In our case, we will add the following lines to this file:

```
FROM python:3.8-slim
ENV PYTHONUNBUFFERED 1
WORKDIR /app
COPY requirements.txt /app/requirements.txt
RUN pip install -r requirements.txt
COPY . /app
CMD python manage.py runserver 0.0.0.0:8000
```

The first line in this file is setting the base image for this container, and we set it to **Python:3.8-slim**, which is already available in the Docker repository. The second line in the file is setting an environment variable for better logging. The rest of the lines are self-explanatory as they are mostly Unix commands.

3. As a next step, we will create a Docker Compose file (**docker-compose.yml**) as follows:

```
version: '3.7'
services:
 gradesms:
 build:
 context: .
 dockerfile: Dockerfile
```

```

ports:
 - 8000:8000
volumes:
 - .:/app

```

This is a YAML file, and we define containers as services in it. Since we have only one container, we defined the **gradesms** service. Note that **build** is pointing to **Dockerfile** we just created and assuming it is in the same directory as this **docker-compose.yml** file. The container port **8000** is mapped to the web server port **8000**. This is an important step in allowing traffic from the container to your application inside the container.

4. As the last step, we mount the current directory (.) to the /**app** directory inside the container. This will allow the changes made on our system to be reflected in the container and vice versa. This step is important if you are creating containers during the development cycle.

5. We can start our container by using the following Docker Compose command:

**docker-compose up**

For the first time, it will build a new container image and will require internet access to download the base container image from the Docker registry. After creating a container image, it will automatically start the container.

Details of how Docker Engine and Docker Compose work are beyond the scope of this book, but we recommend that you become familiar with container technology such as Docker through their online documentation (<https://docs.docker.com/>).

## Reusing our Students API app

We will be reusing our **Students** API app, which we developed in the previous chapter. This app will be started with its built-in server and we will name it the **Students** microservice for our sample application. There will be no change in this application.

## Updating our Students web application

The **webapp** application, which we developed for the case study in the previous chapter, is only using **apiapp** via a REST API. In a revised version of this web application, we will use the **Grades** microservice and the **Students** microservice to fetch the list of **Grade** objects and the list of **Student** objects. The **list** function in our web application will combine the two object lists to provide additional info to web clients. The updated **list** function in the **webapp.py** file will be as follows:

```

STUDENTS_MS = "http://localhost:8080/students"
GRADES_MS = "http://localhost:8000/grades"
@app.get('/')
def list():

```

```

student_svc_resp = requests.get(STUDENTS_MS)
students = json.loads(student_svc_resp.text)
grades_svc_resp = requests.get(GRADES_MS)
grades_list = json.loads(grades_svc_resp.text)
grades_dict = {cls_item['grade']:
 cls_item for cls_item in grades_list}

for student in students:
 student['building'] = grades_dict[student['grade']]['building']
 student['teacher'] = grades_dict[student['grade']]['teacher']

return render_template('main.html', students=students)

```

In this revised code, we created a **grades** dictionary using a dictionary comprehension from the list of **Grades** objects. This dictionary will be used to insert the grade attributes inside the **Student** objects before sending them to the Jinja template for rendering. In our main Jinja template (**main.html**), we added two additional columns, **Building** and **Teacher**, to the **Students** table, as shown here:

The screenshot shows a web application interface. At the top, there is a header bar with the title "Students" and a menu icon. Below the header, there is a modal window titled "Add a Student". Inside the modal, there are three input fields labeled "First name", "Last Name", and "Grade", each with a corresponding text input box. Below these fields is a "Submit" button. To the right of the modal, there is a table titled "Students". The table has a header row with columns: "No", "Name", "Grade", "Building", "Teacher", and "Actions". There are two data rows: Row 1 contains "1", "John Lee", "10", "Building #10", "Miss Hannah", and "Update" and "Delete" buttons; Row 2 contains "2", "Brian Miles", "7", "NA", "Miss Fey", and "Update" and "Delete" buttons.

No	Name	Grade	Building	Teacher	Actions
1	John Lee	10	Building #10	Miss Hannah	<button>Update</button> <button>Delete</button>
2	Brian Miles	7	NA	Miss Fey	<button>Update</button> <button>Delete</button>

Figure 11.6 – Updated main page with Building and Teacher data

In this section, we covered creating a microservice, deploying it as a Docker container as well as a web app on a web server, and combining the results of the two microservices for a web application.

## Deploying the Students microservice to GCP Cloud Run

So far, we have used the **Students** microservice as a web application with a REST API hosted in a flask development server. It is now the time to containerize it and deploy it to the **Google Cloud Platform (GCP)**. GCP has a runtime engine (**Cloud Run**) for deploying containers and running them as a service (microservice). Here are the steps involved:

1. To package the application code of our **Students** microservice in a container, we will first identify a list of dependencies and export them to a **requirements.txt** file. We will run the following command from the virtual environment of the **Students** microservice project:

```
pip freeze -> requirements.txt
```

2. The next step is to build a Dockerfile in the root directory of the project, like the one we prepared for our **Grades** microservice.

The contents of the Dockerfile are as follows:

```
FROM python:3.8-slim

ENV PYTHONUNBUFFERED True

WORKDIR /app

COPY . .

#Install production dependencies.

RUN pip install -r requirements.txt

RUN pip install Flask gunicorn

Run the web service on container startup. we will use

gunicorn and bind our api_app as the main application

CMD exec gunicorn --bind:$PORT --workers 1 --threads 8 api_app:app
```

To deploy our application on GCP Cloud Run, Dockerfile will suffice. But first, we need to build the container image using the GCP Cloud SDK. This will require us to create a GCP project using either the Cloud SDK or GCP Console. We explained the steps of creating a GCP project and associating a billing account with it in the previous chapters. We assume you have created a project with the name **students-run** on GCP.

3. Once the project is ready, we can use the following command to build a container image of our **Students** API application:

```
gcloud builds submit --tag gcr.io/students-run/students
```

Note that **gcr** stands for Google Container Registry.

4. To create an image, we have to provide the tag attribute in the following format:

```
<hostname>/<Project ID>/<Image name>
```

5. In our case, the hostname is **gcr.io**, which is based in the United States. We can use a locally created image as well, but we must first set the **tag** attribute as per the aforementioned format and then push it to the Google registry. This can be achieved with the following Docker commands:

```
docker tag SOURCE_IMAGE <hostname>/<Project ID>/<Image name>:tagid
```

```
docker push <hostname>/<Project ID>/<Image name>
```

```
#or if we want to push a specific tag
docker push <hostname>/<Project ID>/<Image name>:tag
```

As we can see, the **gcloud build** command can achieve two steps in one command.

6. The next step is to run the uploaded image. We can run our container image by using the following Cloud SDK command:

```
gcloud run deploy --image gcr.io/students-run/students
```

The execution of our image can be triggered from the GCP console as well. Once the container is successfully deployed and running, the output of this command (or on a GCP console) will include the URL of our microservice.

To consume this new version of the **Students** microservice from GCP Cloud Run, we will update our web application to switch to use the URL of this newly deployed service in GCP Cloud Run. If we test our web application with a locally deployed **Grades** microservice and the remotely deployed **Students** microservice, we will get the same results as shown earlier in *Figure 11.6* and can perform all operations as we did when the **Students** microservice was deployed locally.

This concludes our discussion on building microservices using different Python frameworks, deploying them locally as well as in the cloud, and consuming them from a web application.

## Summary

In this chapter, we introduced microservices architecture and discussed its merits and demerits. We covered several best practices for building, deploying, and operationalizing microservices. We also analyzed the development options available in Python for building microservices that include Flask, Django, Falcon, Nameko, Bottle, and Tornado. We selected Flask and Django to build sample microservices. To implement a new microservice, we used Django with its REST framework (DRF). This microservice implementation also introduces you to how the Django framework works in general. Later, we provided details of how to containerize a newly created microservice using Docker Engine and Docker Compose. Finally, we converted our **Students** API application to a Docker image and deployed it on GCP Cloud Run. We updated the **Students** web application to consume two microservices deployed in different parts of the world.

The code examples included in this chapter will provide you with hands-on experience in building and deploying microservices for different environments. This knowledge is beneficial for anyone who is looking to build microservices in their next projects. In the next chapter, we will explore how to use Python to develop serverless functions, another new paradigm of software development for the cloud.

## Questions

1. Can we deploy a microservice without a container?
2. Is it appropriate for two microservices to share a single database but with a different schema?
3. What is Docker Compose and how does it help to deploy microservices?
4. Is REST the only format for data exchange for microservices?

## Further reading

- *Hands-On Docker for Microservices with Python*, by Jaime Buelta
- *Python Microservices Development*, by Tarek Ziade
- *Domain-Driven Design: Tackling Complexity in the Heart of Software*, by Eric Evans
- *Google Cloud Run quick-start tutorials* for building and deploying microservices, available at <https://cloud.google.com/run/docs/quickstarts/>

## Answers

1. Yes, but it is recommended to deploy it in a container.
2. Technically, it is feasible, but it is not a best practice. Database failure will bring both microservices down.
3. Docker Compose is a tool for deploying and running container applications using a YAML file. It provides an easy format to define different services (containers) with deployment and runtime attributes.
4. A REST API is the most popular interface for data exchange for microservices, but not the only one. Microservices can also use RPC and events-based protocols for data exchange.

[OceanofPDF.com](http://OceanofPDF.com)

## *Chapter 12: Building Serverless Functions using Python*

Serverless computing is a new model of cloud computing that separates the management of physical or virtual servers and infrastructure-level software, such as database systems, from the application itself. This model allows developers to solely focus on application development and enables someone else to manage the underlying infrastructure resources. Cloud providers are the best option to use to adopt this model. Containers are not only opportune for complex deployments, but they are also a breakthrough technology for the **serverless computing** era. In addition to containers, there is another form of serverless computing, which is known as **Function as a Service (FaaS)**. In this new paradigm, cloud providers offer a platform to develop and run application functions or **serverless functions**, usually in response to an event or as a direct call to those functions. All public cloud providers, such as Amazon, Google, Microsoft, IBM, and Oracle, offer this service. The focus of this chapter will be on understanding and building serverless functions using Python.

In this chapter, we will cover the following topics:

- Introducing serverless functions
- Understanding deployment options for serverless functions
- Learning how to build serverless functions with a case study

After completing this chapter, you should have a clear understanding of the role of serverless functions in cloud computing and how to build them using Python.

## **Technical requirements**

The following is a list of the technical requirements for this chapter:

- You will need to have Python 3.7, or later, installed on your computer.
- To deploy a serverless function in **Google Cloud Platform (GCP)** Cloud Functions, you will need a GCP account (a free trial will work fine).
- You will need an account (that is, a free account) with *SendGrid* for sending emails.

The sample code for this chapter can be found at <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter12>.

Let's begin with an introduction to serverless functions.

## **Introducing serverless functions**

A serverless function is a model that can be used to develop and execute software components or modules without needing to know or worry about an underlying hosting platform. These software modules or components are known as **Lambda functions** or **Cloud functions** in the public cloud providers' product offerings. Amazon was the first vendor that offered such serverless functions on its AWS platform as **AWS Lambda**. It was followed by Google and Microsoft, which offer Google **Cloud Functions** and **Azure Functions**, respectively.

Typically, a serverless function has four components, as follows:

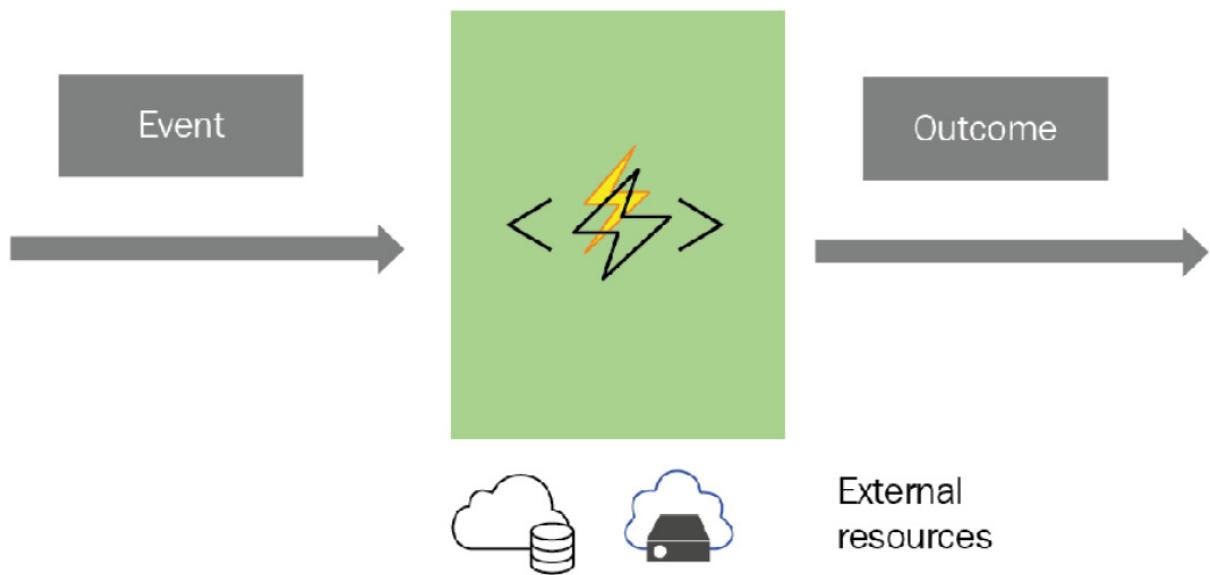


Figure 12.1 – The components of a serverless function

These four components are described next:

- **Functional code:** This is a programming unit that performs certain tasks as per the business or functional goal of the function. For example, we can write a serverless function to process an input stream of data or write a scheduled activity to check certain data resources for monitoring purposes.
- **Events:** Serverless functions are not meant to be used like microservices. Instead, they are meant to be used based on a trigger that can be initiated by an event from a pub/sub system, or they can come as HTTP calls based on an external event in the field such as events from field sensors.
- **Outcome:** When a serverless function is triggered to do a job, there is an output from the function, which can either be a simple response to the caller or trigger the next actions to mitigate the impact of an event. One example of the outcome of a serverless function is to trigger another cloud service such as a database service or send an email to subscribed parties.
- **Resources:** Sometimes, functional code has to use an additional resource to do its job, for example, a database service or cloud storage to access or push files.

## Benefits

Serverless functions bring with them all the benefits of serverless computing, as follows:

- **Ease of development:** Serverless functions take away infrastructure complexities from developers so that they can focus on the functional aspect of the program.
- **Built-in scalability:** Serverless functions are offered with built-in scalability to handle any traffic growth at any time.
- **Cost efficiency:** Serverless functions not only reduce development costs but also offer optimized deployment and an operational mode. Typically, this is a *pay-as-you-use* model that means you will only be charged for the time during which your function is being executed.
- **Technology agnostic:** Serverless functions are technology agnostic. This means that you can build them in many programming languages using a variety of different cloud resources.

Note that there are a few limitations to serverless functions; for instance, we will have less system-level control in building such functions and troubleshooting can be tricky without system-level access.

## Use cases

There are several possible uses of serverless functions. For example, we can use such functions for data processing if we receive an event of a file upload in cloud storage or if we have data available through real-time streaming. In particular, serverless functions can be integrated with the **Internet of Things (IoT)** sensors. Typically, IoT sensors are thousands in number. Serverless functions possess the ability to handle the requests from such a large number of sensors in a scalable manner. A mobile application can use such functions as a backend service to perform certain tasks or process data without jeopardizing the mobile device resources. One practical use of serverless functions in real life is the **Amazon Alexa** product. It is not possible to put every skill or ounce of intelligence inside the Alexa device itself. Instead, it uses Amazon Lambda functions for these skills. Another reason why Alexa uses Amazon Lambda functions is the ability to scale them based on the demand. For instance, some functions might be used more often than others such as weather queries.

In the next section, we will investigate the various deployment options for implementing and executing serverless functions.

## Understanding the deployment options for serverless functions

Using a virtual machine or another runtime resource on public clouds for sporadically accessed applications might not be a commercially attractive solution. In such situations, serverless functions come to the rescue. Here, a cloud provider offers dynamically managed resources for your application and only charges you when your application is executed in response to a certain event. In

other words, a serverless function is a backend computing method that is an on-demand and a pay-as-you-use service that is only offered on public clouds. We will introduce a few options for deploying serverless functions in the public clouds, as follows:

- **AWS Lambda:** This is considered to be one of the first service offerings from any of the public cloud providers. AWS Lambda functions can be written in Python, Node.js, Java, PowerShell, Ruby, Java, C#, and Go. AWS Lambda functions can be executed in response to events, such as file uploads to **Amazon S3**, a notification from **Amazon SNS**, or a direct API call. AWS Lambda functions are stateless.
- **Azure Functions:** Microsoft introduced Azure Functions almost two years after the launch of AWS Lambda functions. These functions can be attached to events within the cloud infrastructure. Microsoft provides support to build and debug these functions using Visual Studio, Visual Studio Code, IntelliJ, and Eclipse. Azure Functions can be written in C#, F#, Node.js, PowerShell, PHP, and Python. Additionally, Microsoft offers **Durable Functions** that allow us to write stateful functions in a serverless environment.
- **Google Cloud Functions:** GCP offers Google Cloud Functions as serverless functions. Google Cloud Functions can be written in Python, Node.js, Go, .NET, Ruby, and PHP. Like its competitors, AWS Lambda and Azure Functions, Google Cloud Functions can be triggered by HTTP requests or by events from the Google Cloud infrastructure. Google allows you to use Cloud Build for the automatic testing and deployment of Cloud Functions.

In addition to the top three public cloud providers, there are a few more offerings from other cloud providers. For example, IBM offers Cloud Functions that are based on the open source **Apache OpenWhisk** project. Oracle offers its serverless computing platform based on the open source **Fn** project. The beauty of using these open source projects is that you can develop and test your code locally. Additionally, these projects allow you to port your code from one cloud to another cloud or even to an on-premises environment deployment without any changes.

It is worth mentioning another framework that is well known in serverless computing, called the **Serverless Framework**. This is not a deployment platform but a software tool that can be used locally to build and package your code for serverless deployment and then be used to deploy the package to one of your favorite public clouds. The serverless framework supports several programming languages such as Python, Java, Node.js, Go, C#, Ruby, and PHP.

In the next section, we will build a couple of serverless functions using Python.

## Learning how to build serverless functions

In this section, we will investigate how to build serverless functions for one of the public cloud providers. Although Amazon AWS pioneered serverless functions in 2014 by offering AWS Lambda functions, we will use the Google Cloud Functions platform for our example functions. The reason for this is that we already introduced GCP in great detail in previous chapters, and you can leverage the same GCP account for the deployment of these example functions. However, we strongly recommend that you use the other platforms, especially if you are planning to use their serverless

functions in the future. The core principles of building and deploying these functions on various cloud platforms are the same.

GCP Cloud Functions offers several ways in which to develop and deploy serverless functions (going forward, we will call them *Cloud Functions* in the context of GCP). We will explore two types of events in our example Cloud Functions, which can be described as follows:

- The first Cloud Function will be built and deployed using the GCP Console from end to end. This Cloud Function will be triggered based on an HTTP call (or event).
- The second Cloud Function will be part of a case study to build an application that listens to an event in the cloud infrastructure and takes an action such as sending an email as a response to this event. The Cloud Function used in this case study will be built and deployed using the Cloud **Software Development Kit (SDK)**.

We will start by building a Cloud Function using the GCP Console.

## Building an HTTP-based Cloud Function using the GCP Console

Let's begin with the Google Cloud Function development process. We will build a very simple Cloud Function that provides today's date and current time for an HTTP trigger. Note that the HTTP trigger is the easiest way in which a Cloud Function can be invoked. First, we will need a GCP project. You can create a new GCP project using the GCP Console for this Cloud Function or an existing GCP project. The steps regarding how to create a GCP project and associate a billing account with it are discussed in [Chapter 9, Python Programming for the Cloud](#). Once you have a GCP project ready, building a new Cloud Function is a three-step process. We will explain these steps in the following subsections.

### Configuring Cloud Function attributes

When we initiate the **Create Function** workflow from the GCP Console, we are prompted to provide the Cloud Function definition, as follows:

Cloud Functions | [Copy function](#)

Configuration —  2 Code

### Basics

Function name \* my-datetime [?](#)

Region asia-southeast1 [?](#)

### Trigger

HTTP

Trigger type HTTP [▼](#)

URL <https://asia-southeast1-students-run.cloudfunctions.net/my-datetime>

#### Authentication

Allow unauthenticated invocations  
Check this if you are creating a public API or website.

Require authentication  
Manage authorised users with Cloud IAM.

Require HTTPS [?](#)

**SAVE** CANCEL

Figure 12.2 – The steps to create a new Cloud Function using the GCP Console (1/2)

A high-level summary of the Cloud Function definition appears as follows:

1. We provide the **Function Name** (in our case, this is **my-datetime**) and select the GCP **Region** to host this function.
2. We select **HTTP** as the **Trigger type** for our function. Selecting a trigger for your function is the most important step. There are also other triggers available such as **Cloud Pub/Sub** and **Cloud Storage**. At the time of writing this book, GCP has added a few more triggers for evaluation purposes.

3. For the sake of simplicity, we will allow unauthenticated access for our function.

After clicking on the **Save** button, we will be prompted to enter **RUNTIME, BUILD AND CONNECTIONS SETTINGS**, as shown in the following screenshot:

**RUNTIME, BUILD AND CONNECTIONS SETTINGS** ^

**RUNTIME**    **BUILD**    **CONNECTIONS**

---

**Memory allocated \*** ▼

128 MiB

**Timeout \*** seconds ?

60

---

**Runtine service account** ?

Runtime service account

App Engine default service account ▼

---

**Auto-scaling** ?

Maximum number of instances 1

---

**Runtine environment variables** ?

**+ ADD VARIABLE**

---

**NEXT**    **CANCEL**

Figure 12.3 – The steps to create a new Cloud Function using the GCP Console (2/2)

We can provide the **RUNTIME, BUILD AND CONNECTIONS SETTINGS** as follows:

1. We can leave the runtime attributes in their default settings, but we will reduce the **Memory allocated** to **128 MiB** for our function. We have associated a default service account as a **Runtime service account** to this function. We will leave **Auto-scaling**

to its default setting, but this can be set to a maximum number of instances for our function.

2. We can add **Runtime environment variables** underneath the **RUNTIME** tab if we have such a requirement to do so. We will not add any environment variables for our Cloud Function.
3. Underneath the **BUILD** tab, there is an option to add **Build environment variables**. We will not add any variable for our Cloud Function.
4. Underneath the **CONNECTIONS** tab, we can leave the default settings as they are and allow all traffic to access our Cloud Function.

After setting the Cloud Function's runtime, build, and connection settings, the next step will be to add the implementation code for this Cloud Function.

## Adding Python code to a Cloud Function

After clicking on the **Next** button, as shown in *Figure 12.3*, the GCP Console will offer us a view to define or add the function implementation details, as shown in the following screenshot:

The screenshot shows the 'Edit function' interface for a Cloud Function. The top navigation bar includes 'Cloud Functions' and 'Edit function'. Below it, a navigation bar has 'Configuration' (with a checkmark) and 'Code' selected (indicated by a blue circle with the number 2). The 'Runtime' dropdown is set to 'Python 3.8'. The 'Entry point' field contains 'today\_datatime'. The 'Source code' section shows 'main.py' selected in the file tree, with 'Inline Editor' chosen. The code editor displays the following Python code:

```
1 from datetime import date, datetime
2
3 def today_datatime(request):
4
5 request_json = request.get_json()
6 if request.args and 'requester' in request.args:
7 requester_name = request.args.get('requester')
8 elif request_json and 'requester' in request_json:
9 requester_name = request_json['requester']
10 else:
11 requester_name = f'anonymous'
12
13 today = date.today()
14 now = datetime.now()
15 resp = "{date:" + today.strftime("%B %d, %Y") + \
16 ", time: " + now.strftime("%H:%M:%S") + "}"
17 return f'Hello ' + requester_name + '! Here is today date and time:\n' + resp
```

At the bottom, there are buttons for 'PREVIOUS', 'DEPLOY' (highlighted in blue), and 'CANCEL'.

Figure 12.4 – The implementation steps of a Cloud Function using the GCP Console

The options that are available for adding our Python code are as follows:

- We can select several runtime options such as Java, PHP, Node.js, or various Python versions. We selected **Python 3.8** as the **Runtime** for our Cloud Function.
- The **Entry point** attribute must be the name of the function in our code. Google Cloud Function will invoke the function in our code based on this **Entry point** attribute.
- The Python source code can be added inline using the **Inline Editor** on the right-hand side; alternatively, it can be uploaded using a ZIP file from your local machine or even from cloud storage. We can also provide the GCP **Cloud Source** repository location for the source code. Here, we selected to implement our function using the **Inline Editor** tool.

- For Python, the GCP Cloud Functions platform automatically creates two files: **main.py** and **requirements.txt**. The **main.py** file will have our code implementation and the **requirements.txt** file should contain our dependencies on third-party libraries.

A sample code, which is shown inside the **Inline Editor** tool first checks if the caller has sent a **requester** attribute in the HTTP request or not. Based on the **requester** attribute value, we will send a welcome message with today's date and time. We implemented a similar code example with two separate web APIs using a Flask web application in *Chapter 9, Python Programming for the Cloud*, to demonstrate the capabilities of GCP App Engine.

Once we are satisfied with our Python code, we will deploy the function on the Google Cloud Functions platform.

## Deploying a Cloud Function

The next step is to deploy this function using the **Deploy** button at the bottom of the screen, as shown in *Figure 12.4*. GCP will start deploying the function immediately, and it can take a few minutes to complete this activity. It is important to understand that Google Cloud Functions are deployed using containers just like microservices on GCP Cloud Run. The key differences are that they can be invoked using different types of events and they use the pay-as-you-use pricing model.

Once our function has been deployed, we can duplicate it, test it, or delete it from the **Cloud Functions** list, as shown in the following screenshot:

	Name ↑	Region	Trigger	Memory allocated	Actions
<input type="checkbox"/>	<input checked="" type="radio"/> handle_storage_delete	us-central1	Bucket: ch12-cfunc-testing	256 MiB	⋮
<input type="checkbox"/>	<input checked="" type="radio"/> handle_storage_upload	us-central1	Bucket: ch12-cfunc-testing	256 MiB	⋮
<input type="checkbox"/>	<input checked="" type="radio"/> my-datetime	us-east1	HTTP	128 MiB	⋮

Copy function  
Test function  
View logs  
Delete

Figure 12.5 – The main view of Google Cloud Functions

Now, we will quickly show you how convenient it is to test and troubleshoot our Cloud Function using the GCP Console. Once we have selected the **Test function** option for our newly deployed

Cloud Function, the GCP Console will offer us a test page, which is similar to the one shown in *Figure 12.6*, underneath the **TESTING** tab. To test our deployed Cloud Function, we can pass the **requester** attribute in JSON format, as follows:

```
{"requester": "John"}
```

After clicking on [...]TEST THE FUNCTION, we can view the results under the **Output** section and the log details under the **Logs** section at the bottom of the screen, as shown in *Figure 12.6*.

Because we are using an HTTP trigger for our Cloud Function, we can also test it by using a web browser or **CURL** utility from anywhere on the internet. However, we have to make sure that our Cloud Function includes **allUsers** as its member with the role of **Cloud Functions Invoker**. This can be set underneath the **PERMISSIONS** tab. However, we do not recommend doing so without setting an authentication mechanism for your Cloud Function:

The screenshot shows the GCP Cloud Functions console for a function named 'my-datetime'. The 'TESTING' tab is active. In the 'Triggering event' section, the JSON input is shown as:

```
1 {"requester": "John"}
2 |
```

The 'TEST THE FUNCTION' button is visible. The 'Output' section shows the response:

```
$ Hello John! Here is today date and time:
{date:July 10, 2021, time: 07:24:52}
```

The 'Logs' section shows two log entries:

- 2021-07-10T07:24:51.717Z my-datetime ke0djhswsq2 Function execution started
- 2021-07-10T07:24:52.033Z my-datetime ke0djhswsq2 Function execution took 316 ms, finished with status code: 200

Figure 12.6 – Testing your Cloud Function using the GCP Console

Building a simple Cloud Function using the GCP Console is a straightforward process. Next, we will explore a case study of a real-world application of Cloud Functions.

## Case study – building a notification app for cloud storage events

In this case study, we will develop a Cloud Function that is triggered for events on a **Google Storage bucket**. On receiving such an event, our Cloud Function will send an email to a predefined list of

email addresses as a notification. The flow of this notification app with a Cloud Function appears as follows:

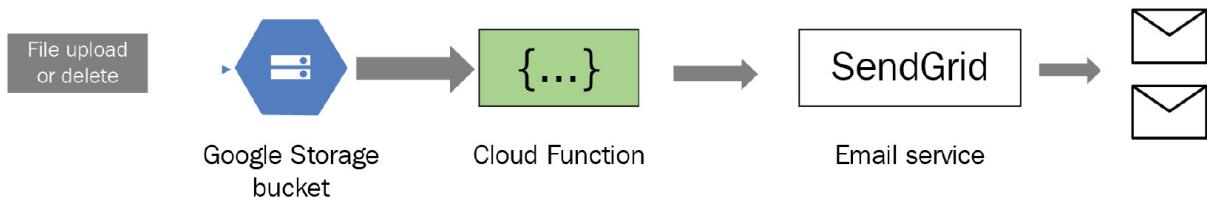


Figure 12.7 – A Cloud Function listening to Google storage bucket events

Note that we can set our Cloud Function to listen to one or more Google Storage events. Google Cloud Functions supports the following Google Storage events:

- **finalize**: This event is created when a new file is added or replaced within a storage bucket.
- **delete**: This event represents the deletion of a file from a storage bucket. This applies to non-versioning buckets. Note that a file is not deleted in reality, but it is archived if the bucket is set to use versioning.
- **archive**: This event is raised when a file is archived. The archive operation is triggered when a file is deleted or overwritten for buckets with versioning.
- **metadata update**: If there is any update in the metadata of a file, this event is created.

After receiving an event from a Google Storage bucket, the Cloud Function will extract the attributes from the context and event objects passed as arguments to our Cloud Function. Then, the cloud function will use a third-party email service (such as *SendGrid* from *Twilio*) to send the notification.

As a prerequisite, you have to create a free account with *SendGrid* (<https://sendgrid.com/>). After creating an account, you will need to create at least one sender user inside your *SendGrid* account. Additionally, you will need to set up a secret API key inside the *SendGrid* account that can be used with the Cloud Function to send emails. Twilio SendGrid offers within the range of 100 emails per day for free, which is good enough for testing purposes.

For this case study, we will write our Python code for the Cloud Function locally and then deploy it to the Google Cloud Functions platform using the Cloud SDK. We will implement this notification application step by step, as follows:

1. We will create a storage bucket to attach to our Cloud Function, and we will upload or delete files from this bucket to simulate the events of our Cloud Function. We can use the following Cloud SDK command to create a new bucket:

```
gsutil mb gs://<bucket name>
gsutil mb gs://muasif-testcloudfn #Example bucket created
```

To keep the generation of these events simple, we will turn off the versioning on our storage bucket by using the following Cloud SDK command:

```
gsutil versioning set off gs://muasif-testcloudfn
```

2. Once the storage bucket is ready, we will create a local project directory and set up a virtual environment using the following commands:

```
python -m venv myenv
source myenv/bin/activate
```

3. Next, we will install the **sendgrid** Python package using the **pip** utility, as follows:

```
pip install sendgrid
```

4. Once our third-party libraries have been installed, we will need to create the **requirements.txt** dependencies file, as follows:

```
pip freeze -r requirements.txt
```

5. Next, we will create a new Python file (**main.py**) with a **handle\_storage\_event** function within it. This function will be the entry point for our Cloud Function. The sample code for this entry point function is as follows:

```
#main.py

from sendgrid import SendGridAPIClient

from sendgrid.helpers.mail import Mail, Email, To, Content

def handle_storage_event(event, context):

 from_email = Email("abc@domain1.com")

 to_emails = To("xyz@domain2.com")

 subject = "Your Storage Bucket Notification"

 content = f"Bucket Impacted:{event['bucket']} \n" + \
 f"File Impacted: {event['name']} \n" + \
 f"Event Time: {event['timeCreated']} \n" + \
 f"Event ID: {context.event_id} \n" + \
 f"Event Type: {context.event_type}"

 mail = Mail(from_email, to_emails, subject, content)

 sg = SendGridAPIClient()

 response = sg.send(mail)

 print(response.status_code) # for logging purpose

 print(response.headers)
```

In the preceding Python code example, our **handle\_storage\_event** function is expected to receive **event** and **context** objects as input arguments. An **event** object is a dictionary that contains the data of the event. We can access the event data from this object using keys such as **bucket** (that is, the bucket name), **name** (that is, the filename), and **timeCreated** (that is, the creation time).

The **context** object provides the context of the event such as **event\_id** and **event\_type**.

Additionally, we use the **sendgrid** library to prepare the email contents and then send the email with the event information to the target email list.

6. Once we have our Python code file (in our case, this is **main.py**) and **requirements.txt** files ready, we can trigger the deployment operation using the following Cloud SDK command:

```
gcloud functions deploy handle_storage_create \
--entry-point handle_storage_event --runtime python38 \
--trigger-resource gs://muasif-testcloudfn/ \
--trigger-event google.storage.object.finalize \
--set-env-vars SENDGRID_API_KEY=<Your SEND-GRID KEY>
```

We should run this command under a GCP project with billing enabled, as discussed in the previous section. We have provided the name for our Cloud Function as **handle\_storage\_create**, and the **entry-point** attribute is set to the **handle\_storage\_event** function in the Python code. We set **trigger-event** to the **finalize** event. By using **set-env-vars**, we set **SENDGRID\_API\_KEY** for the SendGrid service.

The **deploy** command will package the Python code from the current directory, prepare the target platform as per the **requirements.txt** file, and then deploy our Python code to the GCP Cloud Functions platform. In our case, we can create a **.gcloudignore** file to exclude the files and directories so that they can be ignored by the Cloud SDK **deploy** command.

7. Once we deploy our Cloud Function, we can test it by uploading a local file to our storage bucket using the Cloud SDK command, as follows:

```
gsutil cp test1.txt gs://muasif-testcloudfn
```

As soon as the file copy operation (upload) has been completed, the **finalize** event will trigger the execution of our Cloud Function. As a result, we will receive an email with the event details. We can also check the logs of the Cloud Functions by using the following command:

```
gcloud functions logs read --limit 50
```

For this notification app, we attached our Cloud Function to the **Finalize** event only. However, what if want to attach another event type as well, such as a **Delete** event? Well, only one Cloud Function can be attached to one trigger event. But hold on, a Cloud Function is a deployment entity and not the actual program code. This means we do not need to write or duplicate our Python code to handle another type of event. We can create a new Cloud Function using the same Python code but for the **Delete** event, as follows:

```
gcloud functions deploy handle_storage_delete \
--entry-point handle_storage_event --runtime python38 \
--trigger-resource gs://muasif-testcloudfn/ \
--trigger-event google.storage.object.delete \
--set-env-vars SENDGRID_API_KEY=<Your SEND-GRID KEY>
```

If you notice this version of the **deploy** command, the only changes we made were with the *name* of the Cloud Function and the *type* of the trigger event. This **deploy** command will create a new Cloud Function and will work in parallel to an earlier Cloud Function but will be triggered based on a different event (in this case, this is **delete**).

To test the **delete** event with our newly added Cloud Function, we can remove the already uploaded file (or any file) from our storage bucket using the following Cloud SDK command:

```
gsutil rm gs://muasif-testcloudfn/test1.txt
```

We can create more Cloud Functions using the same Python code for other storage events. This concludes our discussion of how to build Cloud Functions for storage events using the Cloud SDK. All the steps discussed using the Cloud SDK can also be implemented using the GCP Console.

## Summary

In this chapter, we introduced serverless computing and FaaS, followed by an analysis of the main ingredients of serverless functions. Next, we discussed the key benefits of serverless functions and their pitfalls. Additionally, we analyzed several deployment options that are available to build and deploy serverless functions, and these options include AWS Lambda, Azure Functions, Google Cloud Functions, Oracle Fn, and IBM Cloud Functions. In the final part of this chapter, we built a simple Google Cloud Function based on an HTTP trigger using the GCP Console. Then, we built a notification app based on Google storage events and a Google Cloud Function using the Cloud SDK. These serverless functions were deployed using the Google Cloud Functions platform.

The code examples included in this chapter should provide you with some experience of how to use both the GCP Console and the Cloud SDK to build and deploy Cloud Functions. This hands-on knowledge is beneficial for anyone who is looking to build a career in serverless computing.

In the next chapter, we will explore how to use Python with machine learning.

## Questions

1. How are serverless functions different from microservices?
2. What is the practical use of serverless functions in real-world examples?
3. What are Durable functions and who offers them?
4. One Cloud Function can be attached to multiple triggers. Is this true or false?

## Further reading

- *Serverless Computing with Google Cloud* by Richard Rose

- *Mastering AWS Lambda* by Yohan Wadia
- *Mastering Azure Serverless Computing* by Lorenzo Barbieri and Massimo Bonanni
- *Google Cloud Functions Quickstart tutorials* for building and deploying Cloud Functions, which is available at <https://cloud.google.com/functions/docs/quickstarts>

## Answers

1. Both are two different offerings of serverless computing. Typically, serverless functions are triggered by an event and are based on the *pay-as-you-use* model. In comparison, microservices are typically consumed through API calls and are not based on the *pay-as-you-use* model.
2. Amazon Alexa uses AWS Lambda functions to provide intelligence and other skills for its users.
3. Durable functions are an extension of Microsoft Azure Functions, which offers stateful functionality in a serverless environment.
4. False. One Cloud Function can only be attached to a single trigger.

[OceanofPDF.com](http://OceanofPDF.com)

## *Chapter 13: Python and Machine Learning*

**Machine learning (ML)** is a branch of **artificial intelligence (AI)** that is based on building models by learning patterns from data and then using those models to make predictions. It is one of the most popular AI techniques for helping humans as well as businesses in many ways. For example, it is being used for medical diagnosis, image processing, speech recognition, predicting threats, data mining, classification, and many more scenarios. We all understand the importance and usefulness of machine learning in our lives. Python, being a concise but powerful language, is being used extensively to implement machine learning models. Python's ability to process and prepare data using libraries such as NumPy, pandas, and PySpark makes it a preferred choice for developers for building and training ML models.

In this chapter, we will discuss using Python for machine learning tasks in an optimized way. This is especially important because training an ML model is a compute-intensive task and optimizing the code is fundamental when using Python for machine learning.

We will cover the following topics in this chapter:

- Introducing machine learning
- Using Python for machine learning
- Testing and evaluating machine learning models
- Deploying machine learning models in the cloud

After completing this chapter, you will understand how to use Python to build, train, and evaluate machine learning models and how to deploy them in the cloud and use them to make predictions.

## **Technical requirements**

The following are the technical requirements for this chapter:

- You need to have Python 3.7 or later installed on your computer.
- You need to install additional libraries for machine learning such as SciPy, NumPy, pandas, and scikit-learn.
- To deploy an ML model on GCP's AI Platform, you will need a GCP account (a free trial will work fine).

The sample code for this chapter can be found at <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter13>.

We will start our discussion with an introduction to machine learning.

## **Introducing machine learning**

In traditional programming, we provide data and some rules as input to our program to get the desired output. Machine learning is a fundamentally different programming approach, in which the data and the expected output are provided as input to produce a set of rules. This is called a **model** in machine learning nomenclature. This concept is illustrated in the following diagram:

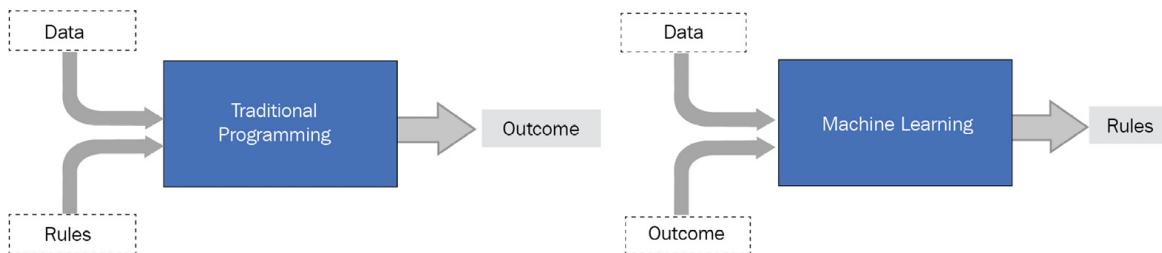


Figure 13.1 – Traditional programming versus machine learning programming

To understand how machine learning works, we need to familiarize ourselves with its core components or elements:

- **Dataset:** Without a good set of data, machine learning is nothing. Good data is the real power of machine learning. It has to be collected from different environments and cover various situations to represent a model close to a real-world process or system. Another requirement for data is that it has to be large, and by large we mean thousands of records. Moreover, the data should be as accurate as possible and have meaningful information in it. Data is used to train the system and also to evaluate its accuracy. We can collect data from many sources but most of the time, it is in a raw format. We can use data processing techniques by utilizing libraries such as pandas, as we discussed in the previous chapters.
- **Feature extraction:** Before using any data to build a model, we need to understand what type of data we have and how it is structured. Once we have understood that, we can select what features of the data can be used by an ML algorithm to build a model. We can also compute additional features based on the original feature set. For example, if we have raw image data in the form of pixels, which itself may not be useful for training a model, we can use the length or breadth of the shape inside an image as features to build rules for our model.
- **Algorithm:** This is a program that is used to build an ML model from the available data. In mathematical terms, a machine learning algorithm tries to learn a target function  $f(X)$  that can map the input data,  $X$ , to an output,  $y$ , like so:

$$y = f(X)$$

There are several algorithms available for different types of problems and situations because there is not a single algorithm that can solve every problem. A few popular algorithms are **linear regression**, **classification and regression trees**, and **support vector classifier (SVC)**. The mathematical details of how these algorithms work are beyond the scope of this book. We recommend checking the additional links provided in the *Further reading* section for details regarding these algorithms.

- **Models:** Often, we hear the term model in machine learning. A model is a mathematical or computational representation of a process that is happening in our day-to-day life. From a machine learning perspective, it is the output of a machine learning

algorithm when we apply it to our dataset. This output (model) can be a set of rules or some specific data structure that can be used to make predictions when used for any real-world data.

- **Training:** This is not a new component or step in machine learning. When we say training a model, this means applying an ML algorithm to a dataset to produce an ML model. The model we get as output is said to be trained on a certain dataset. There are three different ways to train a model:
  - a) **Supervised learning:** This includes providing the desired output, along with our data records. The goal here is to learn how the input (X) can be mapped to the output (Y) using the available data. This approach of learning is used for classification and regression problems. Image classification and predicting house prices (regression) are a couple of real-world examples of supervised learning. In the case of image processing, we can train a model to identify the type of animal in an image, such as a cat or a dog, based on the shape, length, and breadth of the image. To train our image classification model, we will label each image in the training dataset with the animal's name. To predict house pricing, we must provide data about the houses in the location we are looking at, such as the area they're in, the number of rooms and bathrooms, and so on.
  - b) **Unsupervised learning:** In this case, we train a model without knowing the desired output. Unsupervised learning is typically applied to clustering and association use cases. This type of learning is mainly based on observations and finding groups or clusters of data points so that the data points in a group or cluster have similar characteristics. This type of learning approach is extensively used by online retail stores such as Amazon to find different groups of customers (clustering) based on their shopping behavior and offer them items they're interested in. Online stores also try to find an association between different purchases, such as how likely that a person buying item A will want to buy item B as well.
  - c) **Reinforcement learning:** In the case of reinforcement learning, the model is rewarded for making an appropriate decision in a particular situation. In this case, no training data is available at all, but the model has to learn from experience. Autonomous cars are a popular example of reinforcement learning.
- **Testing:** We need to test our model on a dataset that is not used to train the model. A traditional approach is to train our model using two-thirds of the dataset and test the model using the remaining one-third.

In addition to the three learning approaches we discussed, we also have deep learning. This is an advanced type of machine learning based on the approach of how the human brain achieves a certain type of knowledge using neural network algorithms. In this chapter, we will use supervised learning to build our sample models.

In the next section, we will explore the options that are available in Python for machine learning.

## Using Python for machine learning

Python is a popular language in the data scientist community because of its simplicity, cross-platform compatibilities, and rich support for data analysis and data processing through its libraries. One of the key steps in machine learning is preparing data for building the ML models, and Python is a natural winner in doing this. The only challenge in using Python is that it is an interpreted language, so the speed of executing code is slow in comparison to languages such as C. But this is not a major issue as there are libraries available to maximize Python's speed by using multiple cores of **central processing units (CPUs)** or **graphics processing units (GPUs)** in parallel.

In the next subsection, we will introduce a few Python libraries for machine learning.

## Introducing machine learning libraries in Python

Python comes with several machine learning libraries. We already mentioned supporting libraries such as NumPy, SciPy, and pandas, which are fundamental for data refinement, data analysis, and data manipulation. In this section, we will briefly discuss the most popular Python libraries for building machine learning models.

### **scikit-learn**

This library is a popular choice because it has a large variety of built-in ML algorithms and tools to evaluate the performance of those ML algorithms. These algorithms include classification and regression algorithms for supervised learning and clustering and association algorithms for unsupervised learning. scikit-learn is mostly written in Python and relies on the NumPy library for many operations. For beginners, we recommend starting with the scikit-learn library and then moving to the next level of libraries, such as TensorFlow. We will use scikit-learn to illustrate the concepts of building, training, and evaluating the ML models.

scikit-learn also offers **gradient boost algorithms**. These algorithms are based on the mathematical concept of **Gradient**, which is a slope of a function. It measures the change in an error in the ML context. The idea of gradient-based algorithms is to fine-tune the parameters iteratively to find the local minimum of a function (minimizing errors for the ML models). Gradient boost algorithms use the same strategy to improve a model iteratively by taking into account the performance of the previous model, by fine-tuning the parameters for the new model, and by setting the target to accept the new model if it minimizes the errors more than the previous model.

### **XGBoost**

XGBoost, or **eXtreme Gradient Boosting**, is a library of algorithms that relies on gradient boosted decision trees. This library is popular as it is extremely fast and offers the best performance compared to other implementations of gradient boosting algorithms, as well as traditional machine learning

algorithms. scikit-learn also offers gradient boost algorithms, which are fundamentally the same as XGBoost, though XGBoost is significantly fast. The main reason is the maximal utilization of parallelism across different cores of a single machine or in a distributed cluster of nodes. XGBoost can also regularize the decision trees to avoid overfitting the model to the data. XGBoost is not a full framework for machine learning but offers mainly algorithms (models). To use XGBoost, we must use scikit-learn for the rest of the utility functions and tools, such as data analysis and data preparation.

## TensorFlow

TensorFlow is another very popular open source library for machine learning, developed by the Google Brain team for high-performance computation. TensorFlow is particularly useful for training and running deep neural networks and is a popular choice in the area of deep learning.

## Keras

This is an open source API for deep learning for neural networks in Python. Keras is more of a high-level API on top of TensorFlow. For developers, using Keras is more convenient than using TensorFlow directly, so it is recommended to use Keras if you are starting to develop deep learning models with Python. Keras can work with both CPUs and GPUs.

## PyTorch

PyTorch is another open source machine learning library that is a Python implementation of the popular **Torch** library in C.

In the next section, we will briefly discuss the best practices for using Python for machine learning.

## Best practices of training data with Python

We have already highlighted how important the data is when training a machine learning model. In this section, we will highlight a few best practices and recommendations when preparing and using data to train your ML model. These are as follows:

- As we mentioned previously, collecting a large set of data is of key importance (a few thousand data records or at least many hundreds). The bigger the size of the data, the more accurate the ML model will be.
- Clean and refine your data before starting any training. This means that there should not be any missing fields or misleading fields in the data. Python libraries such as pandas are very handy for such tasks.
- Using a dataset without compromising the privacy and security of data is important. You need to make sure you are not using the data of some other organization without the appropriate approval.
- GPUs work well with data-intensive applications. We encourage you to use GPUs to train your algorithms for faster results. Libraries such as XGBoost, TensorFlow, and Keras are well known for using GPUs for training purposes.

- When dealing with a large set of data for training, it is important to utilize the system memory efficiently. We should be loading the data in memory in chunks, or utilizing distributed clusters to process the data. We encourage you to use the generator function as much as you can.
- It is also a good practice to watch your memory usage during data-intensive tasks (for example, while training a model) and free up memory periodically by forcing garbage collection to release unreferenced objects.

Now that we've covered the available Python libraries and the best practices of using Python for machine learning, it is time to start working with real code examples.

## Building and evaluating a machine learning model

Before we start writing a Python program, we will evaluate the process of building a machine learning model.

## Learning about an ML model building process

We discussed the different components of machine learning in the *Introducing machine learning* section. The machine learning process uses those elements as input to train a model. This process follows a procedure with three main phases, and each phase has several steps in it. These phases are shown here:

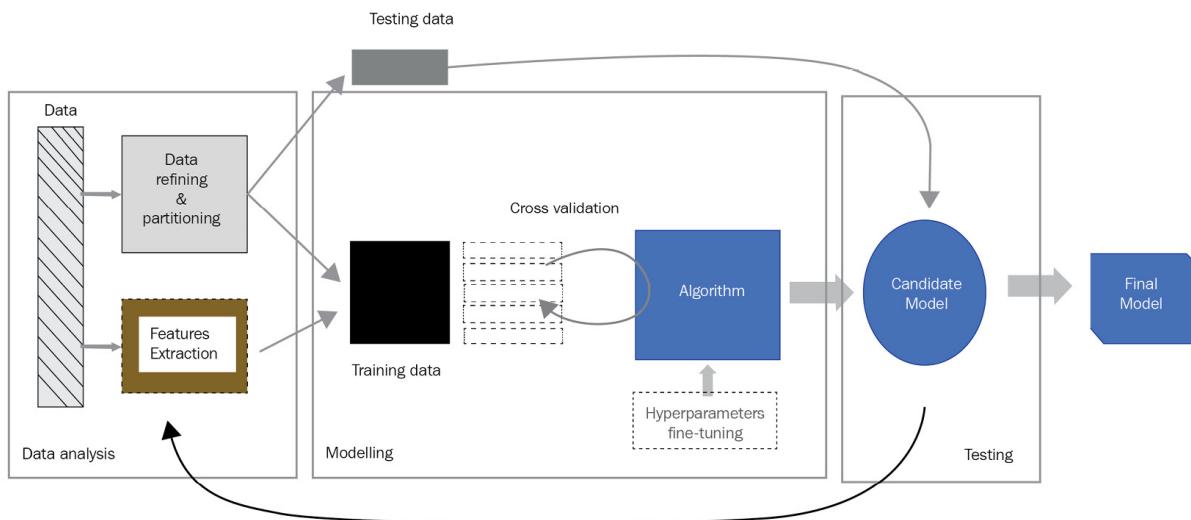


Figure 13.2 – Steps of building an ML model using a classic learning approach

Each phase, along with detailed steps of it, is described here:

- **Data analysis:** In this phase, we collect raw data and transform it into a form that can be analyzed and then used to train and test a model. We may discard some data, such as records with empty values. Through data analysis, we try to select the features (attributes) that can be used to identify patterns in our data. Extracting features is a very important step, and a lot depends on these features when building a successful model. In many cases, we have to fine-tune features after the testing phase to make sure we

have the right set of features for the data. Typically, we partition the data into two sets; one part is used to train the model in the modeling phase, while the other part is used to test the trained model for accuracy in the testing phase. We can skip the testing phase if we are evaluating the model using other approaches, such as **cross-validation**. We recommend having a testing phase in your ML building process and keeping some data (unseen to the model) aside for the testing phase, as shown in the preceding diagram.

- **Modeling:** This phase is about training our model based on the training data and features we extracted in the previous phase. In a traditional ML approach, we can use the training data as-is to train our model. But to ensure our model has better accuracy, we can use the following additional techniques:
  - a) We can partition our training data into slices and use one slice for evaluating of our model and use the remaining slices for training the model. We repeat this for a different combination of training slices and the evaluation slice. This evaluation approach is called cross-validation.
  - b) ML algorithms come with several parameters that can be used to fine-tune the model to best fit the data. Fine-tuning these parameters, also known as **hyperparameters**, is typically done along with cross-validation during the modeling phase.

The feature values in data may use different scales of measurement, which makes it difficult to build rules with a combination of such features. In such cases, we can transform the data (feature values) into a common scale or into a normalized scale (say 0 to 1). This step is called scaling the data, or normalization. All these scaling and evaluation steps (or some of them) can be added to a pipeline (such as an Apache Beam pipeline) and can be executed together to evaluate different combinations for selecting the best model. The output of this phase is a candidate ML model, as shown in the preceding diagram.

- **Testing:** In the testing phase, we use the data we set aside to test the accuracy of the candidate ML model we built in the previous phase. The output of this phase can be used to add or remove some features and fine-tune the model until we get one with acceptable accuracy.

Once we are satisfied with the accuracy of our model, we can implement it to predict based on the data from the real world.

## Building a sample ML model

In this section, we will build a sample ML model using Python, which will identify three types of Iris plants. To build this model, we will use a commonly available dataset containing four features (length and width of sepals and petals) and three types of Iris plants.

For this code exercise, we will use the following components:

- We will use the Iris dataset provided by *UC Irvine Machine Learning Repository* (<http://archive.ics.uci.edu/ml/>). This dataset contains 150 records and three expected patterns to identify. This is a refined dataset that comes with the necessary features already identified.

- We will use various Python libraries, as follows:
  - a) The pandas and the matplotlib libraries, for data analysis
  - b) The scikit-learn library, for training and testing our ML model

First, we will write a Python program for analyzing the Iris dataset.

## Analyzing the Iris dataset

For ease of programming, we downloaded the two files for the Iris dataset (**iris.data** and **iris.names**) from <https://archive.ics.uci.edu/ml/machine-learning-databases/iris/>.

We can directly access the data file from this repository through Python. But in our sample program, we will use a local copy of the files. The scikit-learn library also provides several datasets as part of the library and can be used directly for evaluation purposes. We decided to use the actual files as this will be close to real-world scenarios, where you collect data yourself and then use it in your program.

The Iris data file contains 150 records that are sorted based on the expected output. In the data file, the values of four different features are provided. These four features are described in the **iris.names** file as **sepal-length**, **sepal-width**, **petal-length**, and **petal-width**. The expected output types of Iris plant, as per the data file, are **Iris-setosa**, **Iris-versicolor**, and **Iris-virginica**. We will load the data into a pandas DataFrame and then analyze it for different attributes of interest. Some sample code for analyzing the Iris data is as follows:

```
#iris_data_analysis.py
from pandas import read_csv
from matplotlib import pyplot
data_file = "iris/iris.data"
iris_names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']
df = read_csv(data_file, names=iris_names)
print(df.shape)
print(df.head(20))
print(df.describe())
print(df.groupby('class').size())
box and whisker plots
df.plot(kind='box', subplots=True, layout=(3,3), sharex=False, sharey=False)
pyplot.show()
check the histograms
df.hist()
pyplot.show()
```

In the first part of the data analysis, we checked a few metrics about the data using the pandas library functions, as follows:

- We used the **shape** method to get the dimension of the DataFrame. This should be [150, 5] for the Iris dataset as we have 150 records and five columns (four for features and one for the expected output). This step ensures that all the data is loaded into our DataFrame correctly.
- We checked the actual data using the **head** or **tail** method. This is only to see the data visually, especially if we have not seen what is inside the data file.
- The **describe** method gave us the different statistical KPIs available for the data. The outcome of this method is as follows:

	sepal-length	sepal-width	petal-length	petal-width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

These KPIs can help us select the right algorithm for the dataset.

- The **groupby** method was used to identify the number of records for each **class** (name of the column for the expected output). The output will indicate that there are 50 records for each type of Iris plant:

```
Iris-setosa 50
Iris-versicolor 50
Iris-virginica 50
```

In the second part of the analysis, we tried to use a **box plot** (also known as a **box and whisker** plot) and **histogram** plots. Box plots are a visual way of displaying the KPIs we received by using the **describe** method (minimum value, first quartile, second quartile (median), third quartile, and the maximum value). This plot will tell you if your data is symmetrically distributed or grouped in a certain range, or how much of your data is skewed toward one side of the distribution. For our Iris dataset, we will get the box plots for our four features as follows:

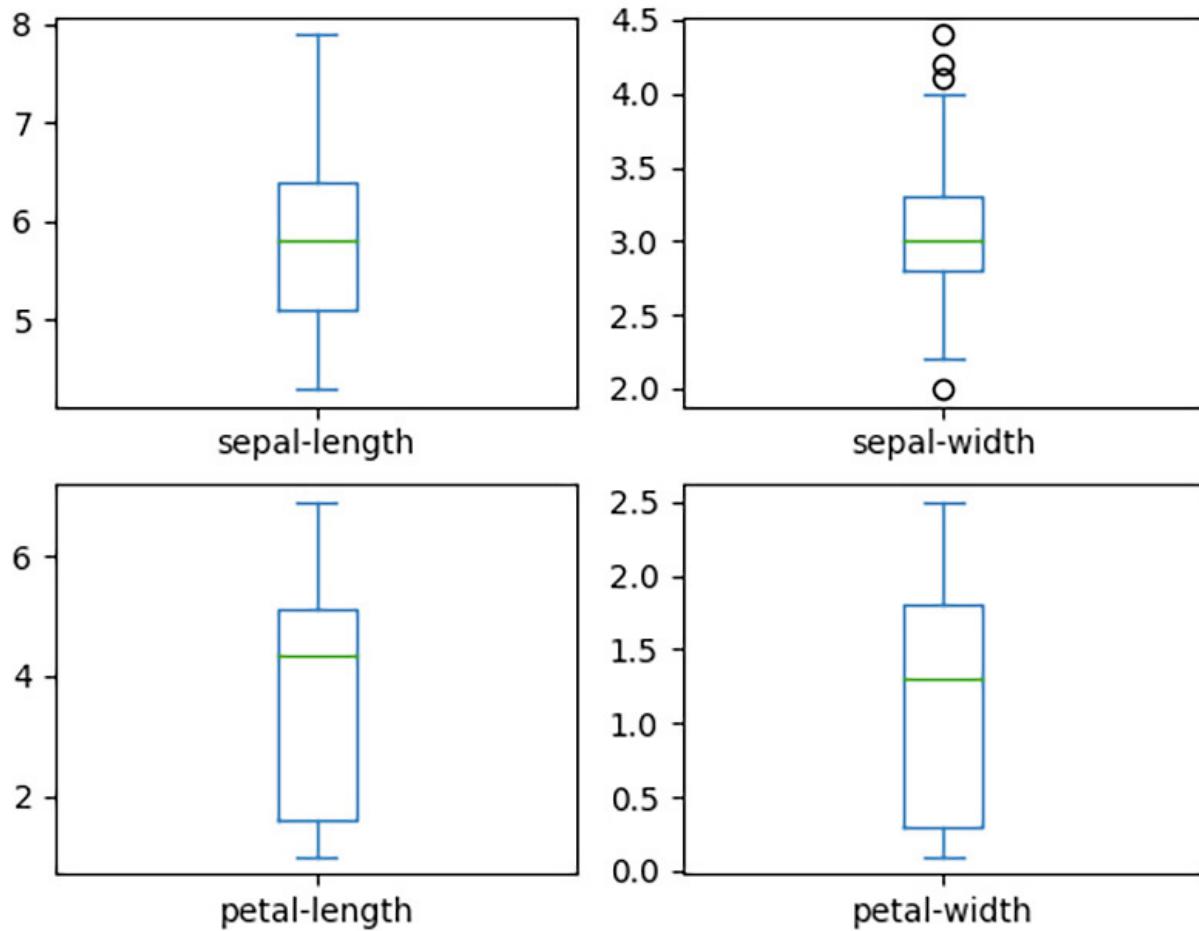


Figure 13.3 – Box and whisker plots of Iris dataset features

From these plots, we can see that the **petal-length** and the **petal-width** data has the most grouping between the first quartile and the third quartile. We can confirm this by analyzing the data distribution by using the histogram plots, as follows:

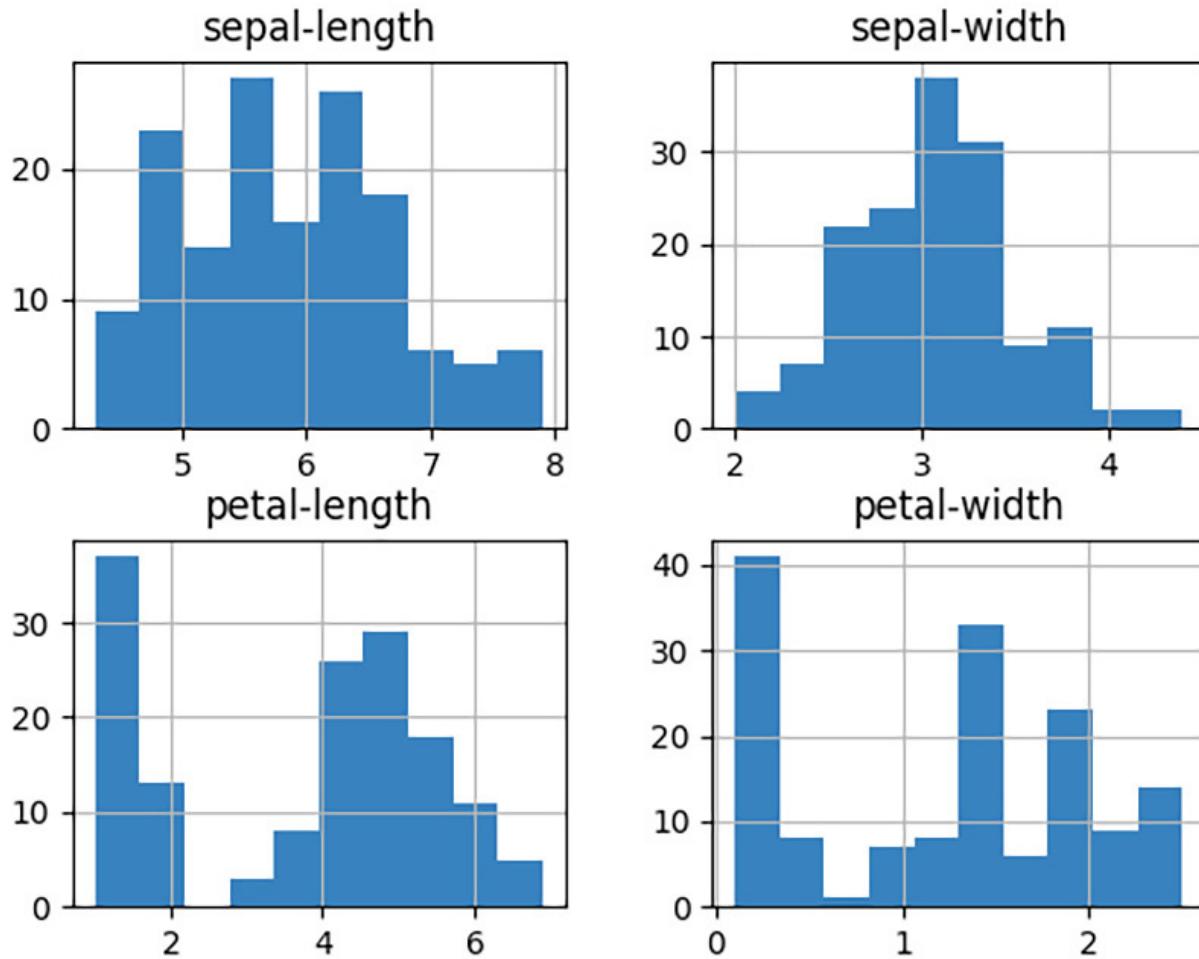


Figure 13.4 – Histogram of Iris dataset features

After analyzing the data and selecting the right type of algorithm (model) to use, we will move on to the next step, which is training our model.

### Training and testing a sample ML model

To train and test an ML algorithm (model), we must follow these steps:

- As a first step, we will split our original dataset into two groups: training data and testing data. This approach of splitting the data is called the **Holdout** method. The scikit-learn library provides the **train\_test\_split** function to make this split convenient:

```
#iris_build_svm_model.py (#1)

Split the dataset

X = df.drop('class', axis = 1)

y = df['class']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20,
random_state=1, shuffle=True)
```

Before calling the **train\_test\_split** function, we split the full dataset into a features dataset (typically called **X**, which should be uppercase in machine learning nomenclature) and the expected output dataset (called **y**, which should be lowercase in machine learning nomenclature). These two datasets (**X** and **y**) are split by the **train\_test\_split** function as per our **test\_size** (20%, in our case). We also allow the data to be shuffled before splitting it. The output of this operation will give us four datasets (**X\_train**, **y\_train**, **X\_test**, and **y\_test**) for training and testing purposes.

2. In the next step, we will create a model and provide the training data (**X\_train** and **y\_train**) to train this model. The choice of ML algorithm is not that important for this exercise. For the Iris dataset, we will use the SVC algorithm with default parameters. Some sample Python code is as follows:

```
#iris_build_svm_model.py (#2)

make predictions

model = SVC(gamma='auto')

model.fit(X_train, y_train)

predictions = model.predict(X_test)
```

To train our model, we used the **fit** method. In the next statement, we made predictions based on the testing data (**X\_test**). These predictions will be used to evaluate the performance of our trained model.

3. Finally, the predictions will be evaluated with the expected results, as per the test data (**y\_test**), using the **accuracy\_score** and **classification\_report** functions of the scikit-learn library, as follows:

```
#iris_build_svm_model.py (#3)

predictions evaluation

print(accuracy_score(y_test, predictions))

print(classification_report(y_test, predictions))
```

The console output of this program is as follows:

```
0.9666666

 Iris-setosa 1.00 1.00 1.00 11
 Iris-versicolor 1.00 0.92 0.96 13
 Iris-virginica 0.86 1.00 0.92 6
 accuracy 0.97 0.97 0.97 30
 macro avg 0.95 0.97 0.96 30
 weighted avg 0.97 0.97 0.97 30
```

The accuracy scope is very high (0.966), which indicates that the model can predict the Iris plant with nearly 96% accuracy for the testing data. The model is doing an excellent job for **Iris-setosa** and

**Iris-versicolor** but only a decent job (86% precise) in the case of **Iris-virginica**. There are several ways to improve the performance of our model, all of which we will discuss in the next section.

## Evaluating a model using cross-validation and fine tuning hyperparameters

For the previous sample model, we kept the training process simple for the sake of learning the core steps of building an ML model. For production deployments, we cannot rely on a dataset that only contains 150 records. Additionally, we must evaluate the model for the best predictions using techniques such as the following:

- **k-fold cross validation:** In the previous model, we shuffled the data before splitting it into training and testing datasets using the Holdout method. Due to this, the model can give us different results every time we train it, thus resulting in an unstable model. It is not trivial to select training data from a small dataset that contains 150 records since in our case, that can truly represent the data of a real-world system or environment. To make our previous model more stable with a small dataset, k-fold cross-validation is the recommended approach. This approach is based on dividing our dataset into  $k$  folds or slices. The idea is to use  $k-1$  slices for training and to use the  $k^{\text{th}}$  slice for evaluating or testing. This process is repeated until we use every slice of data for testing purposes. This is equivalent to repeating the holdout method  $k$  times using the different slices of data for testing.

To elaborate further, we must split our whole dataset or training data set into five slices, say  $k=5$ , for 5-fold cross-validation. In the first iteration, we can use the first slice (20%) for testing and the remaining four slices (80%) for training. In the second iteration, we can use the second slice for testing and the remaining four slices for training, and so on. We can evaluate the model for all five possible training datasets and select the best model in the end. The selection scheme of data for training and testing is shown here:



Figure 13.5 – Cross-validation scheme for five slices of data

The cross-validation accuracy is calculated by taking the average accuracy of each model we build in each iteration, as per the value of  $k$ .

- **Optimizing hyperparameters:** In the previous code example, we used the machine learning algorithm with default parameters. Each machine learning algorithm comes with many hyperparameters that can be fine-tuned to customize the model, as per the dataset. It may be possible for statisticians to set a few parameters manually by analyzing the data distribution, but it is tedious to analyze the impact of a combination of these parameters. There is a need to evaluate our model by using different values of these hyperparameters, which can assist us in selecting the best hyperparameter combination in the end. This technique is called fine-tuning or optimizing the hyperparameters.

Cross-validation and fine-tuning hyperparameters are tedious to implement, even through programming. The good news is that the scikit-learn library comes with tools to achieve these evaluations in a couple of lines of Python code. The scikit-learn library offers two types of tools for this evaluation: **GridSearchCV** and **RandomizedSearchCV**. We will discuss each of these tools next.

## GridSearchCV

The **GridSearchCV** tool evaluates any given model by using the cross-validation approach for all possible combinations of values provided for the hyperparameters. Each combination of values of hyperparameters will be evaluated by using cross-validation on dataset slices.

In the following code example, we will use the **GridSearchCV** class from the scikit-learn library to evaluate the SVC model for a combination of **C** and **gamma** parameters. The **C** parameter is a regularization parameter that manages the tradeoff between having a low training error versus having a low testing error. A higher value of **C** means we can accept a higher number of errors. We will use 0.001, 0.01, 1, 5, 10, and 100 as values for **C**. The **gamma** parameter is used to define the non-linear hyperplanes or non-linear lines for the classification. The higher the value of **gamma**, the model can try to fit more data by adding more curvature or curve to the hyperplane or the line. We will use values such as 0.001, 0.01, 1, 5, 10, and 100 for **gamma** as well. The complete code for

**GridSearchCV** is as follows:

```
#iris_eval_svc_model.py (part 1 of 2)

from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.datasets import load_iris
from sklearn.svm import SVC

iris= load_iris()

X = iris.data
y = iris.target

X_train, X_test, y_train, y_test=train_test_split (X,y,test_size=0.2)

params = {"C": [0.001, 0.01, 1, 5, 10, 100], "gamma": [0.001, 0.01, 0.1, 1, 10, 100]}

model=SVC()
```

```

grid_cv=GridSearchCV(model, params, cv=5)
grid_cv.fit(X_train,y_train)
print(f"GridSearch- best parameter:{grid_cv.best_params_}")
print(f"GridSearch- accuracy: {grid_cv.best_score_}")
print(classification_report(y_test, grid_cv.best_estimator_.predict(X_test)))

```

In this code example, the following points need to be highlighted:

- We loaded the data directly from the scikit-learn library for illustration purposes. You can use the previous code to load the data from a local file as well.
- It is important to define the **params** dictionary for fine-tuning hyperparameters as a first step. We set the values for the **C** and **gamma** parameters in this dictionary.
- We set **cv=5**. This will evaluate each parameter combination by using cross-validation across the five slices.

The output of this program will give us the best combination of **C** and **gamma** and the accuracy of the model with cross-validation. The console output for the best parameters and the accuracy of the best model is as follows:

```

GridSearch- best parameter: {'C': 5, 'gamma': 0.1}
GridSearch- accuracy: 0.9833333333333334

```

By evaluating different combinations of parameters and using cross-validation with **GridSearchCV**, the overall accuracy of the model is improved to 98% from 96%, compared to the results we observed without cross-validation and hyperparameter fine-tuning. The classification report (not shown in the program output) shows that the precision for the three plant types is 100% for our test data. However, this tool is not feasible to use when we have a large number of parameter values with a large dataset.

## **RandomizedSearchCV**

In the case of the **RandomizedSearchCV** tool, we only evaluate a model for randomly selected hyperparameter values instead of all the different combinations. We can provide the parameter values and the number of random iterations to perform as input. The **RandomizedSearchCV** tool will randomly select the parameter combination as per the number of iterations provided. This tool is useful when we are dealing with a large dataset and when many combinations of parameters/values are possible. Evaluating all the possible combinations for a large dataset can be a very long process that requires a lot of computing resources.

The Python code for using **RandomizedSearchCV** is the same as for the **GridSearchCV** tool, except for the following additional lines of codes:

```

#iris_eval_svc_model.py (part 2 of 2)
rand_cv=RandomizedSearchCV(model, params, n_iter = 5, cv=5)

```

```
rand_cv.fit(x_train,y_train)
print(f" RandomizedSearch - best parameter: {rand_cv.best_params_}")
print(f" RandomizedSearch - accuracy: {rand_cv.best_score_}")
```

Since we defined **n\_iter=5**, **RandomizedSearchCV** will select only five combinations of the **C** and **gamma** parameters and evaluate the model accordingly.

When this part of the program is executed, we will get an output similar to the following:

```
RandomizedSearch- best parameter: {'gamma': 10, 'C': 5}
RandomizedSearch- accuracy: 0.9333333333333333
```

Note that you may get a different output because this tool may select different parameter values for the evaluation. If we increase the number of iterations (**n\_iter**) for the **RandomizedSearchCV** object, we will observe more accuracy in the output. If we do not set **n\_iter**, we will run the evaluation for all combinations, which means we'll get the same output that **GridSearchCV** provides.

As we can see, the best parameter combination that's selected by the **GridSearchCV** tool is different than the one selected by the **RandomizedSearchCV** tool. This is expected because we ran the two tools for a different number of iterations.

This concludes our discussion on building a sample ML model using the scikit-learn library. We covered the core steps and concepts that are required in building and evaluating such models. In practice, we also scale the data for normalization. This scaling can be achieved either by using built-in scaler classes in the scikit-learn library, such as **StandardScaler**, or by building our own scaler class. The scaling operation is a data transformation operation and can be combined with the model training task under a single pipeline. scikit-learn supports combining multiple operations or tasks as a pipeline using the **Pipeline** class. The **Pipeline** class can also be used directly with the **RandomizedSearchCV** or **GridSearchCV** tools. You can find out more about how to use scalers and pipelines with the scikit-learn library by reading the online documentation for the scikit-learn library ([https://scikit-learn.org/stable/user\\_guide.html](https://scikit-learn.org/stable/user_guide.html)).

In the next section, we will discuss how to save a model to a file and restore a model from a file.

## Saving an ML model to a file

When we have evaluated a model and selected the best one as per our dataset, the next step is to implement this model for future predictions. This model can be implemented as part of any Python application, such as web applications, Flask, or Django, or it can be used as a microservice or even a cloud function. The real question is how to transfer the model object from one program to the other.

There are a couple of libraries such as **pickle** and **joblib** that can be used to serialize a model into a file. The file can then be used in any application to load the model again in Python and make predictions using the **predict** method of the model object.

To illustrate this concept, we will save the ML model we created in one of the previous code examples (for example, the **model** object in the **iris\_build\_svm\_model.py** program) to a file called **model.pkl**. In the next step, we will load the model from this file using the pickle library and make a prediction using new data to emulate the use of a model in any application. The complete sample code is as follows:

```
#iris_save_load_predict_model.py

#model is creating using the code in #iris_build_svm_model.py

#saving the model to a file

with open("model.pkl", 'wb') as file:

 pickle.dump(model, file)

#loading the model from a file (in another application

with open("model.pkl", 'rb') as file:

 loaded_model = pickle.load(file)

 x_new = [[5.6, 2.6, 3.9, 1.2]]

 y_new = loaded_model.predict(x_new)

 print("X=%s, Predicted=%s" % (x_new[0], y_new[0]))
```

The use of the joblib library is simpler than the pickle library but it may require you to install this library if it has not been installed as a dependency of scikit-learn. The following sample code shows the use of the joblib library to save our best model, as per the evaluation of the **GridSearchCV** tool we did in the previous section, and then load the model from the file:

```
#iris_save_load_predict_gridmodel.py

#grid_cv is created and trained using the code in the

#iris_eval_svm_model.py

joblib.dump(grid_cv.best_estimator_, "model.joblib")

loaded_model = joblib.load("model.joblib")

x_new = [[5.6, 2.5, 3.9, 1.1]]

y_new = loaded_model.predict(x_new)

print("X=%s, Predicted=%s" % (x_new[0], y_new[0]))
```

The code for the joblib library is concise and simple. The prediction part of the sample code is the same as in the previous code sample for the pickle library.

Now that we've learned how the model can be saved in a file, we can take the model to any application for deployment and even to a cloud platform, such as GCP AI Platform. We will discuss how to deploy our ML model on a GCP platform in the next section.

## Deploying and predicting an ML model on GCP Cloud

Public cloud providers are offering several AI platforms for training built-in models, as well as your custom models, for deploying the models for predictions. Google offers the **Vertex AI** platform for ML use cases, whereas Amazon and Azure offer the **Amazon SageMaker** and **Azure ML** services, respectively. We selected Google because we assume you have set up an account with GCP and that you are already familiar with the core concepts of GCP. GCP offers its AI Platform, which is part of the Vertex AI Platform, for training and deploying your ML models at scale. The GCP AI Platform supports libraries such as scikit-learn, TensorFlow, and XGBoost. In this section, we will explore how to deploy our already trained model on GCP and then predict the outcome based on that model.

Google AI Platform offers its prediction server (compute node) either through a global endpoint (**ml.googleapis.com**) or through a regional endpoint (**<region>-ml.googleapis.com**). The global API endpoint is recommended for batch predictions, which are available for TensorFlow on Google AI Platform. The regional endpoint offers additional protection against outages in other regions. We will use a regional endpoint to deploy our sample ML model.

Before we start deploying a model in GCP, we will need to have a GCP project. We can create a new GCP project or use an existing GCP project that we've created for previous exercises. The steps of creating a GCP project and associating a billing account with it were discussed in [Chapter 9, Python Programming for the Cloud](#). Once we have a GCP project ready, we can deploy the **model.joblib** model, which we created in the previous section. The steps for deploying our model are as follows:

1. As a first step, we will create a storage bucket where we will store our model file. We can use the following Cloud SDK command to create a new bucket:

```
gsutil mb gs://<bucket name>
gsutil mb gs://muasif-svc-model #Example bucket created
```

2. Once our bucket is ready, we can upload our model file (**model.joblib**) to this storage bucket using the following Cloud SDK command:

```
gsutil cp model.joblib gs://muasif-svc-model
```

Note that the model's filename should be *model.\**. This means that the filename must be **model** with an extension such as **pkl**, **joblib**, or **bst**, depending on the library we used to package the model.

We can now initiate a workflow to create a model object on the AI Platform by using the following command. Note that the name of the model must include only alphanumeric and underscore characters:

```
gcloud ai-platform models create my_iris_model - region=us-central1
```

3. Now, we can create a version for our model by using the following command:

```
gcloud ai-platform versions create v1 \
--model=my_iris_model \
--origin=gs://muasif-svc-model \
--framework=scikit-learn \
--runtime-version=2.4 \
--python-version=3.7 \
--region=us-central1 \
--machine-type=n1-standard-2
```

The different attributes of this command are as follows:

- a) The **model** attribute will point to the name of the model we created in the previous step.
- b) The **origin** attribute will point to the storage bucket location where the model file is residing. We will only provide the directory's location, not the path to the file.
- c) The **framework** attribute is used to select which ML library to use. GCP offers scikit-learn, TensorFlow, and XGBoost.
- d) **runtime-version** is for the scikit-learn library in our case.
- e) **python-version** is selected as 3.7, which is the highest version offered that's by GCP AI Platform at the time of writing this book.
- f) The **region** attribute is set as per the region that was selected for the model.
- g) The **machine-type** attribute is optional and is used to indicate what type of compute node to use for model deployment. If not provided, the **n1-standard-2** machine type is used.

The **versions create** command may take a few minutes to deploy a new version. Once it is done, we will get an output similar to the following:

```
Using endpoint [https://us-central1- ml.googleapis.com/]
Creating version (this might take a few minutes).....done.
```

4. To check if our model and version have been deployed correctly, we can use the **describe** command under the **versions** context, as shown here:

```
gcloud ai-platform versions describe v1 - model=my_iris_model
```

5. Once our model has been deployed with its version, we can use new data to predict the outcome using the model we deployed on Google AI Platform. For testing, we added a couple of data records, different from the original dataset, in a JSON file (**input.json**), as follows:

```
[5.6, 2.5, 3.9, 1.1]
```

```
[3.2, 1.4, 3.0, 1.8]
```

We can use the following Cloud SDK command to predict the outcome based on the records inside the **input.json** file, as follows:

```
gcloud ai-platform predict --model my_iris_model --version v1 --json-instances input.json
```

The console output will show the predicted class for each record, as well as the following:

```
Using endpoint [https://us-central1-ml.googleapis.com/]
['Iris-versicolor', 'Iris-virginica']
```

To use the deployed model in our application (local or cloud), we can use *Cloud SDK* or *Cloud Shell*, but the recommended approach is to use the Google AI API to make any predictions.

With that, we have covered cloud deployment and the prediction options for our ML model using Google AI Platform. However, you can also take your ML model to other platforms, such as Amazon SageMaker and Azure ML for deployment and prediction. You can find more details about the Amazon SageMaker platform at <https://docs.aws.amazon.com/sagemaker/> and more details about Azure ML at <https://docs.microsoft.com/en-us/azure/machine-learning/>.

## Summary

In this chapter, we introduced machine learning and its main components, such as datasets, algorithms, and models, as well as training and testing a model. This introduction was followed by a discussion of popular machine learning frameworks and libraries available for Python. These include scikit-learn, TensorFlow, PyTorch, and BGBoost. We also discussed the best practices of refining and managing the data for training ML models. To get familiar with the scikit-learn library, we built a sample ML model using the SVC algorithm. We trained the model and evaluated it using techniques such as k-fold cross-validation and fine-tuning hyperparameters. We also learned how to store a trained model in a file and then load that model into any program for prediction purposes. In the end, we demonstrated how we can deploy our ML model and predict results using the Google AI Platform with a few GCP Cloud SDK commands.

The concepts and the hands-on exercises included in this chapter are adequate to help build the foundation for using Python for machine learning projects. This theoretical and hands-on knowledge is beneficial for those who are looking to start using Python for machine learning.

In the next chapter, we will explore how to use Python for network automation.

## Questions

1. What are supervised learning and unsupervised learning?
2. What is k-fold cross-validation and how it is used to evaluate a model?
3. What is **RandomizedSearchCV** and how it is different from **GridSearchCV**?
4. What libraries can we use to save a model in a file?
5. Why are regional endpoints preferred option over global endpoints for Google AI Platform?

## Further reading

- *Machine Learning Algorithms*, by Giuseppe Bonacorso
- *40 Algorithms Every Programmer Should Know*, by Imran Ahmad
- *Mastering Machine Learning with scikit-learn*, by Gavin Hackeling
- *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2*, by Sebastian Raschka and Vahid Mirjalili
- *scikit-learn User Guide*, available at [https://scikit-learn.org/stable/user\\_guide.html](https://scikit-learn.org/stable/user_guide.html)
- *Google AI Platform Guides* for training and deploying ML models are available at <https://cloud.google.com/ai-platform/docs>

## Answers

1. In supervised learning, we provide the desired output with the training data. The desired output is not included as part of the training data for unsupervised learning.
2. Cross-validation is a statistical technique that's used to measure the performance of an ML model. In k-fold cross-validation, we divide the data into  $k$  folds or slices. We train our model using the  $k-1$  slices of the dataset and test the accuracy of the model using the  $k$ th slice. We repeat this process until each  $k$ th slice is used as testing data. The cross-validation accuracy of the model is computed by taking the average of the accuracy of all the models we built through  $k$  iterations.
3. **RandomizedSearchCV** is a tool that's available with scikit-learn for applying cross-validation functionality to an ML model for randomly selected hyperparameters. **GridSearchCV** provides similar functionality to **RandomizedSearchCV**, except that it validates the model for all the combinations of hyperparameter values provided to it as an input.
4. Pickle and Joblib.
5. Regional endpoints offer additional protection against any outages in other regions, and the availability of computing resources is more for regional endpoints than global endpoints.

## *Chapter 14: Using Python for Network Automation*

Traditionally, networks are built and operated by network experts, and this is still a trend in the telecom industry. However, this manual approach of managing and operating a network is slow and sometimes results in costly network outages due to human mistakes. Additionally, to obtain a new service (such as an internet service), customers have to wait for days after placing a request for a new service before it's ready. Based on the experience of smartphones and mobile applications, where you can enable new services and applications with a click of a button, customers expect network service readiness in minutes, if not seconds. This is not possible with the current approach to network management. The traditional approaches are also sometimes a roadblock in introducing new products and services by the telecom service providers.

**Network automation** can improve these situations by offering software for automating the management as well as operational aspects of a network. Network automation helps eliminate human errors in configuring network devices and reduce operational costs significantly by automating repetitive tasks. Network automation helps accelerate service delivery and enables telecom service providers to introduce new services.

Python is a popular choice for network automation. In this chapter, we will discover Python capabilities for network automation. Python provides libraries such as **Paramiko**, **Netmiko**, and **NAPALM** that can be used to interact with network devices. If the network devices are managed by a **Network Management System (NMS)** or a network controller/orchestrator, Python can interact with these platforms using the **REST** or **RESTCONF** protocols. End-to-end network automation is not possible without listening to real-time events happening in the network. These real-time network events or real-time streaming data is typically available through systems such as **Apache Kafka**. We will also explore interaction with an event-driven system using Python.

We will cover the following topics in this chapter:

- Introducing network automation
- Interacting with network devices
- Integrating with network management systems
- Working with event-based systems

After completing this chapter, you will understand how to use Python libraries to fetch data from a network device and to push configurational data to these devices. These are foundational steps for any network automation process.

# Technical requirements

The following are the technical requirements for this chapter:

- You need to have Python 3.7 or later installed on your computer.
- You need to install Paramiko, Netmiko, NAPALM, ncclient, and the requests libraries on top of Python.
- You need to have access to one or more network devices with the SSH protocol.
- You need to have access to a Nokia developer lab to be able to access Nokia's NMS (known as **Network Services Platform (NSP)**).

The sample code for this chapter can be found at <https://github.com/PacktPublishing/Python-for-Geeks/tree/master/Chapter14>.

## *IMPORTANT NOTE*

*In this chapter, you will need access to physical or virtual network devices and network management systems to execute the code examples. This may not be possible for everyone. You can use any network device with similar capabilities. We will focus more on the Python side of the implementation and make it convenient to reuse the code for any other device or management system.*

We will start our discussion by providing an introduction to network automation.

## Introducing network automation

Network automation is the use of technology and software to automate the processes of managing and operationalizing networks. The keyword for network automation is *automating a process*, which means it is not only about deploying and configuring a network but also the steps that must be followed to achieve network automation. For example, sometimes, the automation steps involve gaining approval from different stakeholders before a configuration is pushed to a network.

Automating such an approval step is part of network automation. Therefore, the network automation process can vary from one organization to another based on the internal processes each organization follows. This makes it challenging to build a single platform that can perform automation out of the box for many customers.

There are a significant number of ongoing efforts to provide the necessary platforms from the network device vendors that can help in building customized automation with minimal effort. A few examples of such platforms are Cisco **Network Services Orchestrator (NSO)**, the **Paragon Automation** platform from Juniper Networks, and **NSP** from Nokia.

One of the challenges with these automation platforms is that they are typically vendor locked. This means that the vendors claim that their platform can manage and automate other vendors' network devices as well, but the process to achieve multi-vendor automation is tedious and costly. Therefore, telecom service providers are looking beyond the vendor's platforms for automation. **Python** and

**Ansible** are two popular programming languages that are used for automation in the telecom industry. Before we jump into how Python achieves network automation, let's explore a few merits and challenges of network automation.

## Merits and challenges of network automation

We have highlighted a few merits of network automation already. We can summarize the key merits as follows:

- **Expedite service delivery:** Faster service delivery to new customers enables you to start the service billing early and have more satisfied customers.
- **Reducing operational costs:** The operational costs of a network can be reduced by automating repetitive tasks and monitoring the network through tools and closed-loop automation platforms.
- **Eliminate human errors:** The majority of network outages are because of human errors. Network automation can eliminate this cause by configuring the networks using standard templates. These templates are deeply evaluated and tested before being put into production.
- **Consistent network setup:** When humans are configuring a network, it is impossible to follow consistent templates and naming conventions, which are important for the operations team to manage the network. Network automation brings consistency in setting up the network as we configure the network every time using the same script or template.
- **Network visibility:** With network automation tools and platforms, we can have access to performance monitoring capabilities and can visualize our network from end to end. Proactive network management is possible by detecting traffic spikes and heavy resources utilization before they cause bottlenecks for the network traffic.

Network automation is a must for digital transformation, but there are some costs and challenges to achieve it. These challenges are as follows:

- **Cost:** There is always a cost when it comes to building or customizing the software for network automation. Network automation is a journey and a cost budget must be set for it on an annual basis.
- **Human resistance:** In many organizations, human resources consider network automation as a threat to their jobs, so they resist adopting network automation, especially within operation teams.
- **Organizational structure:** Network automation brings a real **return on investment (ROI)** when it is used across different network layers and network domains such as IT and network domains. The challenge in many organizations is that these domains are owned by different departments and each has their own automation strategies and preferences regarding automation platforms.
- **Selecting an automation platform/tool:** Selecting an automation platform from network equipment vendors such as Cisco or Nokia, or working with third-party automation platforms such as HP or Accenture, is not an easy decision. In many cases, the telecom service providers end up with multiple vendors for building their network automation, and this brings a new set of challenges to make these vendors work together.
- **Maintenance:** Maintaining automation tools and scripts is as essential as building them. This requires either buying essential maintenance contracts from automation vendors or setting an internal team to provide maintenance for such automation platforms.

Next, we look at the use cases.

## Use cases

Several monotonous tasks regarding network management can be automated using Python or other tools. But the real benefits are to automate those tasks that are repetitive, error-prone, or tedious if done manually. From a telecom service provider's point of view, the following are the main applications of network automation:

- We can automate the day-to-day configuration of network devices, such as creating new IP interfaces and network connectivity services. It is time-consuming to do these tasks manually.
- We can configure firewall rules and policies to save time. Creating firewall rule configurations is a tedious activity, and any mistakes can result in wasting time in troubleshooting the communication challenges.
- When we have thousands of devices in a network, upgrading their software is a big challenge and sometimes, it takes 1 to 2 years to achieve this. Network automation can expedite this activity and enforce pre- and post-upgrade checks conveniently for seamless upgrades.
- We can use network automation to onboard new network devices in the network. If the device is to be installed on a customer's premises, we can save a truck roll by automating the device's onboarding process. This onboarding process is also known as **zero touch provisioning (ZTP)**.

Now that we've introduced network automation, let's explore how to interact with network devices using different protocols.

## Interacting with network devices

Python is a popular choice for network automation because it is easy to learn and can be used to integrate with network devices directly, as well as through NMS. In fact, many vendors, such as Nokia and Cisco, support Python runtimes on their network devices. The option of on-device Python runtimes is useful for automating tasks and activities in the context of a single device. In this section, we will focus on the off-device Python runtime option. This option will give us the flexibility to work with multiple devices at a time.

### ***IMPORTANT NOTE***

*For all the code examples provided in this section, we will use a virtual network device from Cisco (IOS XR with release 7.1.2). For integration with the NMS we will use the Nokia NSP system.*

Before working with Python so that we can interact with network devices, we will discuss the protocols that are available for communicating with network devices.

## Protocols for interacting with network devices

When it comes to talking to network devices directly, there are several protocols we can use, such as **Secure Shell Protocol (SSH)**, **Simple Network Management Protocol (SNMP)**, and **Network**

**Configuration (NETCONF).** Some of these protocols work on top of each other. The most commonly used protocols will be described next.

## SSH

SSH is a network protocol for communicating between any two devices or computers securely. All the information between the two entities will be encrypted before it's sent to a transport channel. We typically use an SSH client to connect to a network device using the **ssh** command. The SSH client uses the *username* of the logged-in operating system user with the **ssh** command:

```
ssh <server ip or hostname>
```

To use a different user other than the logged-in user, we can specify the *username*, as follows:

```
ssh username@<server IP or hostname>
```

Once an SSH connection has been established, we can send CLI commands either to retrieve configuration or operational information from a device or to configure the device. **SSH version 2 (SSHv2)** is a popular choice for interacting with devices for network management and even for automation purposes.

We will discuss how to use the SSH protocol with Python libraries such as Paramiko, Netmiko, and NAPALM in the *Interacting with network devices using SSH-based protocols* section. SSH is also a foundational transport protocol for many advanced network management protocols, such as NETCONF.

## SNMP

This protocol has been a de facto standard for network management for 30+ years and it is still used heavily for network management. However, it is being replaced by more advanced and scalable protocols such as NETCONF and gNMI. SNMP can be used both for network configuration and for network monitoring, but it is more popular for network monitoring. In today's world, it is considered a legacy protocol that was introduced in the late 1980s, purely for network management.

The SNMP protocol relies on **Management Information Base (MIB)**, which is a device model. This model was built using a data modeling language called **Structure of Management Information (SMI)**.

## NETCONF

The NETCONF protocol, which was introduced by the **Internet Engineering Task Force (IETF)**, is considered a successor of SNMP. NETCONF is primarily used for configuring network devices and is expected to be supported by all new network devices. NETCONF is based on four layers:

- **Content:** This is a data layer that relies on YANG modeling. Every device offers several YANG models for various modules it offers. These models can be explored at <https://github.com/YangModels/yang>.

- **Operations:** NETCONF operations are actions or instructions that are sent from a NETCONF client to a NETCONF server (also called **NETCONF Agent**). These operations are wrapped in the request and reply messages. Examples of NETCONF operations are **get**, **get-config**, **edit-config**, and **delete-config**.
- **Messages:** These are **Remote Procedure Call (RPC)** messages that are exchanged between the NETCONF clients and NETCONF Agent. NETCONF operations and data that's encoded as XML are wrapped within the RPC messages.
- **Transport:** This layer provides a communication path between a client and a server. NETCONF messages can use NETCONF over SSH or NETCONF over TLS with the SSL certificate option.

The NETCONF protocol is based on XML messages that have been exchanged via the SSH protocol using port **830** as the default port. There are typically two types of configuration databases that are managed by network devices. The first type is called the **running** database, which represents the active configuration on a device, including operation data. This is a mandatory database for each device. The second type is known as the **candidate** database, which represents the candidate configuration before it can be pushed to the running database. When a candidate database exists, configuration changes are not allowed to be made directly to the running database.

We will discuss how to work with NETCONF using Python in the *Interacting with network devices using NETCONF* section.

## RESTCONF

RESTCONF is another *IETF* standard that offers a subset of NETCONF functionality using the RESTful interface. Instead of using NETCONF RPC calls with XML encoding, RESTCONF offers HTTP/HTTPS-based REST calls, with the option of using XML or JSON messages. If network devices offer the RESTCONF interface, we can use HTTP methods (**GET**, **PATCH**, **PUT**, **POST**, and **DELETE**) for network management. When RESTCONF is used for network automation, we must understand that it provides a limited NETCONF functionality over HTTP/HTTPS. NETCONF operations such as commits, rollbacks, and configuration locking are not supported through RESTCONF.

## gRPC/gNMI

gNMI is a gRPC **Network Management Interface (NMI)**. gRPC is a remote procedure call that was developed by Google for low-latency and highly scalable data retrieval. The gRPC protocol was developed originally for mobile clients that wanted to communicate with cloud servers with stringent latency requirements. The gRPC protocol is highly efficient for transporting structured data through **protocol buffers (Protobufs)**, which is a key component of this protocol. By using Protobufs, the data is packed in a binary format instead of a textual format such as JSON or XML. This format not only reduces the size of the data but is very efficient for serializing and deserializing data compared to JSON or XML. Moreover, the data is transported using HTTP/2 instead of HTTP 1.1. HTTP/2 offers both the request-response model and the bidirectional communication model. This

bidirectional communication model makes it possible for clients to open long-lived connections that speed up the data transfer process significantly. These two technologies make the gRPC protocol 7 to 10 times faster than the REST API.

gNMI is a specific implementation of the gRPC protocol for network management purposes and telemetry applications. It is also a YANG model-driven protocol like NETCONF and offers very few operations compared to NETCONF. These operations include **Get**, **Set**, and **Subscribe**. gNMI is getting more popular for telemetry data collection than for network management. The main reason for this is that it does not provide as much flexibility as NETCONF for network configuration, but it is an optimized protocol when it comes to collecting data from a remote system, especially in real time or near-real time.

Next, we will discuss Python libraries for interacting with network devices.

## Interacting with network devices using SSH-based Python libraries

There are several Python libraries available for interacting with network devices using SSH. Paramiko, Netmiko, and NAPALM are three popular libraries that are available, and we will explore them in the next subsections. We will start with Paramiko.

### Paramiko

The Paramiko library is an abstraction of the SSH v2 protocol in Python and includes both server-side and client-side functionality. We will only focus on the client-side capabilities of the Paramiko library here.

When we interact with a network device, we either try to get configuration data, or we push a new configuration for certain objects. The former is achieved with *show* types of CLI commands, as per the operating system of the device, while the latter may require a special mode for executing the configuration CLI commands. These two types of commands are handled differently when working through Python libraries.

### Fetching device configuration

To connect to a network device (listening as an SSH server) using the Paramiko library, we must use an instance of the **paramiko.SSHClient** class or directly use a low-level **paramiko.Transport** class. The **Transport** class offers low-level methods that provide direct control over sockets-based communication. The **SSHClient** class is a wrapper class and uses the **Transport** class under the hood to manage a session, with an SSH server implemented on a network device.

We can use the Paramiko library to establish a connection with a network device (Cisco IOS XR, in our case) and to run a show command (**show ip int brief**, in our case) like so:

```
#show_cisco_int_pk.py

import paramiko
host='HOST_ID'
port=22
username='xxx'
password='xxxxxx'

#cisco ios command to get a list of IP interfaces
cmd= 'show ip int brief \n'

def main():

 try:
 ssh = paramiko.SSHClient()
 ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
 ssh.connect(host, port, username, password)
 stdin, stdout, stderr = ssh.exec_command(cmd)
 output_lines = stdout.readlines()
 response = ''.join(output_lines)
 print(response)

 finally:
 ssh.close()

if __name__ == '__main__':
 main()
```

The key points of this code example are as follows:

- We created an **SSHClient** instance and opened a connection with the SSH server.
- Since we are not using the host key for our SSH connection, we applied the **set\_missing\_host\_key\_policy** method to avoid any warnings or errors.
- Once the SSH connection had been established, we sent our show command, **show ip int brief**, to the host machine using SSH transport and received the output of the command as an SSH reply.
- The output of this program is a tuple of **stdin**, **stdout**, and **stderr** objects. If our command is executed successfully, we will retrieve the output from the **stdout** object.

The output of this program, when executed on a Cisco IOS XR device, is as follows:

```
Mon Jul 19 12:03:41.631 UTC
Interface IP-Address Status Protocol
```

Loopback0	10.180.180.10	Up	Up
GigabitEthernet0/0/0/0	10.1.10.2	Up	Up
GigabitEthernet0/0/0/0.100	unassigned	Up	Down
GigabitEthernet0/0/0/1	unassigned	Up	Up
GigabitEthernet0/0/0/1.100	150.150.150.1	Up	Up
GigabitEthernet0/0/0/2	unassigned	Shutdown	Down

If you are running this program on another device type, you must change the command that has been set as the **cmd** variable, as per your device's type.

The Paramiko library provides low-level control over network communication, but it can sometimes be quirky due to the non-standard or incomplete implementation of the SSH protocol by many network devices. If you face challenges in using Paramiko with some network devices, it is not you or Paramiko but the way the device expects you to communicate with it. A low-level transport channel can solve these issues, but this requires a bit of complex programming. Netmiko comes to the rescue here.

## Netmiko

Netmiko is an abstracted library for network management that is built on top of the Paramiko library. It eliminates the challenges of Paramiko by treating every network device differently. Netmiko uses Paramiko under the hood and hides many device-level communication details. Netmiko supports several devices from different vendors, such as Cisco, Arista, Juniper, and Nokia.

### Fetching device configuration

To connect to a network device using the show type of CLI commands, we must set a **device\_type** definition that is used to connect with the target network device. This **device\_type** definition is a dictionary that must include the device's type, the host IP or the device's **Fully Qualified Domain Name (FQDN)**, the username, and the password to connect with the device. We can set the port number for SSH connection if the target machine is listening on a port other than 22. The following code can be used to execute the same **show** command that we executed with the Paramiko library:

```
#show_cisco_int_nm.py

from netmiko import ConnectHandler
cisco_rtr = {
 "device_type": "cisco_ios",
 "host": "HOST_ID",
 "username": "xxx",
 "password": "xxxxxxxx",
```

```

 #"global_delay_factor": 2,
}

def main():
 command = "show ip int brief"
 with ConnectHandler(**cisco_rtr) as net_connect:
 print(net_connect.find_prompt())
 print(net_connect.enable())
 output = net_connect.send_command(command)
 print(output)

```

The key points of this example code are as follows:

- We created a network connection using the **ConnectHandler** class using a context manager. The context manager will manage the life cycle of the connection.
- Netmiko offers a simple method called **find\_prompt** for grabbing the prompt of the target device, which is useful for parsing the output of many network devices. This is not required for the *Cisco IOS XR* network device but we used it as a best practice.
- Netmiko also allows us to enter *Enable* mode (it is a command-line prompt, #) for Cisco IOS devices by using the **enable** method. Again, this is not required for this example, but it is a best practice to use it, especially in cases where we are pushing CLI commands for configuration as part of the same programming script.
- We executed the **show ip int brief** command using the **send\_command** method and got the same output that we did for the **show\_cisco\_int\_pmk.py** program.

Based on the code example we shared for the same **show** command, we can conclude that working with Netmiko is much more convenient compared to Paramiko.

### ***IMPORTANT NOTE***

*It is very important to set the correct device type for consistent results, even if you are using devices from the same vendor. This is especially important when using commands for configuring a device. An incorrect device type can give inconsistent errors.*

Sometimes, we execute commands that require more time to complete than normal **show** commands. For example, we may want to copy a file from one location to the other on a device, and we know this can take a few hundred seconds for a large file. By default, Netmiko waits nearly 100 seconds for a command to complete. We can add a global delay factor as part of the device definition by adding a line like the following:

```
"global_delay_factor": 2
```

This will increase the wait time for all the commands for this device by a factor of 2. Alternatively, we can set the delay factor for an individual command with the **send\_command** method by passing the following argument:

```
delay_factor=2
```

We should be adding a delay factor when we expect a significant execution time. When we have to add a delay factor, we should also be adding another attribute as an argument with the **send\_command** method, which will break the wait cycle early if we see a command prompt (for example, # in the case of Cisco IOS devices). This can be set by using the following attribute:

```
expect_string=r'#'
```

## Configuring a network device

In the following code example, we will provide some sample code for configuration purposes. Configuring a device using Netmiko is similar to executing **show** commands as Netmiko will take care of enabling the configuration terminal (if required, as per the device type) and exiting out of the configuration terminal gracefully.

For our code example, we will set a **description** of an interface using Netmiko with the following program:

```
#config_cisco_int_nmk.py
from netmiko import ConnectHandler
cisco_rtr = {
 "device_type": "cisco_ios",
 "host": "HOST_ID",
 "username": "xxx",
 "password": "xxxxxx",
}
def main():
 commands = ["int Lo0 \"description custom_description\"", "commit"]
 with ConnectHandler(**cisco_rtr) as net_connect:
 output = net_connect.send_config_set(commands)
 print(output)
 print()
```

The key points of this code example are as follows:

- For this program, we created a list of three commands (**int <interface id>**, **description <new description>**, and **commit**). The first two commands can be sent as a single command as well, but we have kept them separate for illustration purposes. The **commit** command is used to save changes.
- When we send a command to the device for configuration, we use the **send\_config\_set** method from the Netmiko library to set up a connection for configuration purposes. Successfully executing this step depends on the correct setting of the device type. This is because the device behavior varies from one device to the other for the configuration commands.

- The set of three commands will add or update the **description** attribute for the specified interface.

No special output will be expected from this program, except that the device config prompts with our commands. The console output will look as follows:

```
Mon Jul 19 13:21:16.904 UTC
RP/0/RP0/CPU0:cisco(config)#int Lo0
RP/0/RP0/CPU0:cisco(config-if)#description custom_description
RP/0/RP0/CPU0:cisco(config-if)#commit
Mon Jul 19 13:21:17.332 UTC
RP/0/RP0/CPU0:cisco(config-if)#

```

Netmiko offers a lot more functionality, but we will leave this for you to explore by reading its official documentation (<https://pypi.org/project/netmiko/>). The code examples we've discussed in this section have been tested with a Cisco network device, but the same program can be used by changing the device type and the commands for any other device if the device is supported by Netmiko.

Netmiko simplifies our code for network device interaction, but we are still running the CLI commands for fetching the device configuration or pushing the configuration toward the device. Programmability is not easily to do with Netmiko, but another library called NAPALM is there to help.

## NAPALM

**NAPALM** is an acronym for **Network Automation and Programmability Abstraction Layer with Multivendor**. This library provides the next level of abstraction on top of Netmiko by offering a set of functions as a unified API to interact with several network devices. It does not support as many devices as Netmiko. For release 3 of NAPALM, core drivers are available for the **Arista EOS, Cisco IOS, Cisco IOS-XR, Cisco NX-OS**, and **Juniper JunOS** network devices. However, there are several community-built drivers available for communicating with many other devices, such as **Nokia SROS, Aruba AOS-CX, and Ciena SAOS**.

As we did for Netmiko, we will build NAPALM examples for interacting with a network device. In the first example, we will get a list of IP interfaces, while for the second example, we will add or update the **description** attribute for an IP interface. These two code examples will perform the same operations that we performed using the Paramiko and Netmiko libraries.

### Fetching device configuration

To fetch a device configuration, we must set up the connection to our network device. We will do this in both code examples. Setting up a connection is a three-step process, as explained here:

1. To set up a connection, we must get a device driver class based on the supported device type. This can be achieved using the **get\_network\_driver** function of the NAPALM library.
2. Once we have a device driver class, we can create a device object by providing arguments such as **host id**, **username**, and **password** to the driver class constructor.
3. The next step is to connect to the device using the **open** method of the device object. All these steps can be implemented as Python code, as shown here:

```
from napalm import get_network_driver
driver = get_network_driver('iosxr')
device = driver('HOST_ID', 'xxxx', 'xxxx')
device.open()
```

Once the device's connection is available, we can call methods such as **get\_interfaces\_ip** (equivalent to the **show interfaces** CLI command) or **get\_facts** (equivalent to the **show version** CLI command). The complete code for using these two methods is as follows:

```
#show_cisco_int_npm.py
from napalm import get_network_driver
import json
def main():
 driver = get_network_driver('iosxr')
 device = driver('HOST_ID', 'root', 'rootroot')
 try:
 device.open()
 print(json.dumps(device.get_interfaces_ip(), indent=2))
 #print(json.dumps(device.get_facts(), indent=2))
 finally:
 device.close()
```

The most interesting fact is that the output of this program is in JSON format by default. NAPALM converts the CLI command's output into a dictionary by default that is easy to consume in Python. An excerpt of the output for the previous code example is shown here:

```
{
 "Loopback0": {
 "ipv4": {
 "10.180.180.180": {
 "prefix_length": 32
 }
 }
 }
}
```

```

 },
 },
 "MgmtEth0/RP0/CPU0/0": {
 "ipv4": {
 "172.16.2.12": {
 "prefix_length": 24
 }
 }
 }
}

```

## Configuring a network device

In the following code example, we are using the NAPALM library to add or update the **description** attribute for an existing IP interface:

```
#config_cisco_int_npm.py

from napalm import get_network_driver
import json

def main():

 driver = get_network_driver('iosxr')
 device = driver('HOST_ID', 'xxx', 'xxxx')
 try:
 device.open()

 device.load_merge_candidate(config='interface Lo0 \n'
 napalm_desc \n end\n')
 description
 print(device.compare_config())
 device.commit_config()
 finally:
 device.close()
```

The key points of this code example are as follows:

- To configure the IP interface, we must use the **load\_merge\_candidate** method and pass the same set of CLI commands to this method as we did for the interface configuration with Netmiko.
- Next, we compared the configs before our commands and after the commands using the **compare\_config** method. This indicates what new configuration has been added and what has been removed.
- We applied a commit to all the changes using the **commit\_config** method.

For this example code, the output will show the delta of changes, like so:

```
--
+++
@@ -47,7 +47,7 @@
!
!
interface Loopback0
- description my custom description
+ description napalm added new desc
 ipv4 address 10.180.180.180 255.255.255.255
!
interface MgmtEth0/RP0/CPU0/0
```

Here, the line that starts with - is a configuration to be removed; any line with + at the start is a new configuration to be added.

With these two code examples, we have showed you a basic set of NAPALM features for one device type. The library can be used to configure multiple devices at a time and can work with different sets of configurations.

In the next section, we will discuss interacting with network devices using the NETCONF protocol.

## Interacting with network devices using NETCONF

NETCONF was created for model (object)-driven network management, especially for network configuration. When working with a network device using NETCONF, it is important to understand two capabilities of the device, as follows:

- You can understand the YANG models of the devices you have. Having this knowledge is important if you wish to send the messages in the correct format. Here is an excellent source of YANG models from various vendors:  
<https://github.com/YangModels/yang>.
- You can enable the NETCONF and SSH ports for the NETCONF protocol on the network device for your network device. In our case, we will be using a virtual device from Cisco IOS XR, as we did in our previous code examples.

Before starting any network management-related activity, we must check the device's NETCONF capabilities and the details of the NETCONF data source's configuration. For all the code examples in this section, we will use a NETCONF client library for Python known as **ncclient**. This library provides convenient methods for sending NETCONF RPC requests. We can write a sample Python

program using the **ncclient** library to get the device's capabilities and the device's full configuration, as follows:

```
#check_cisco_device.py

from ncclient import manager

with manager.connect(host='device_ip', username=xxxx, password=xxxxxx,
hostkey_verify=False) as conn:

 capabilities = []
 for capability in conn.server_capabilities:
 capabilities.append(capability)
 capabilities = sorted(capabilities)

 for cap in capabilities:
 print(cap)

 result = conn.get_config(source="running")
 print (result)
```

The **manager** object from the **ncclient** library is used to connect to the device using SSH but for NETCONF port **830** (default). First, we get a list of server capabilities through the connection instance and then print them in a sorted format for the convenience of reading. In the next part of this code example, we initiated a **get-config** NETCONF operation by using the **get\_config** method of the **manager** class library. The output of this program is very long and displays all the capabilities and the device configuration. We leave it to you to explore the output and become familiar with your device's capabilities.

It is important to understand that the scope of this section is not to explain NETCONF but to learn how to use Python and **ncclient** to work with NETCONF. To achieve this goal, we will write two code examples: one for fetching the configuration of device interfaces and another on how to update the description of an interface, which is the same as what we did for the previous Python libraries.

## Getting interfaces via NETCONF

In the previous section, we learned that our device (Cisco IOS XR) supports interfaces by using the **OpenConfig** implementation, which is available at <http://openconfig.net/yang/interfaces?module=openconfig-interfaces>.

We can also check the XML format of our interface's configuration, which we received as the output of the **get\_config** method. In this code example, we will simply pass an XML filter with interface configuration as an argument to the **get\_config** method, as follows:

```
#show_all_interfaces.py

from ncclient import manager
```

```

with manager.connect(host='device_ip', username=xxx,
hostkey_verify=False) as conn:
 result = conn.get_config("running", filter=('subtree',
'<interfaces xmlns= "http://openconfig.net/yang/ interfaces"/>'))
 print (result)

```

The output of this program is a list of interfaces. We will only show an excerpt of the output here for illustration purposes:

```

<rpc-reply message-id="urn:uuid:f4553429-ed6-4c79-aeea-5739993cacf4"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<interfaces xmlns="http://openconfig.net/yang/interfaces">
<interface>
<name>Loopback0</name>
<config>
<name>Loopback0</name>
<description>Configured by NETCONF</description>
</config>
<!--rest of the output is skipped -->

```

To get a selective set of interfaces, we will use an extended version of the XML filter based on the interface's YANG model. For the following code example, we will define an XML filter with the **name** properties of the interfaces as our filtering criteria. Since this XML filter is more than one line, we will define it separately as a string object. Here is the sample code with the XML filter:

```

#show_int_config.py
from ncclient import manager
Create filter template for an interface
filter_temp = """
<filter>
<interfaces xmlns="http://openconfig.net/yang/interfaces">
<interface>
<name>{int_name}</name>
</interface>
</interfaces>
</filter>"""

```

```

with manager.connect(host='device_ip', username=xxx,
hostkey_verify=False) as conn:
 filter = filter_temp.format(int_name = "MgmtEth0/RP0/ CPU0/0")
 result = m.get_config("running", filter)
 print (result)

```

The output of this program will be a single interface (as per the configuration in our device) and look as follows:

```

<?xml version="1.0"?>
<rpc-reply message-id="urn:uuid:c61588b3-1bfb-4aa4-a9de-2a98727e1e15"
 xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
 xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<interfaces xmlns="http://openconfig.net/yang/interfaces">
<interface>
<name>MgmtEth0/RP0/CPU0/0</name>
<config>
<name>MgmtEth0/RP0/CPU0/0</name>
</config>
<ethernet xmlns="http://openconfig.net/yang/interfaces/ ethernet">
<config>
<auto-negotiate>false</auto-negotiate>
</config>
</ethernet>
<subinterfaces>
<!-- ommitted sub interfaces details to save space -->
</subinterfaces>
</interface>
</interfaces>
</data>
</rpc-reply>

```

We can define XML filters in an XML file as well and then read the file's contents into a string object in the Python program. Another option is to use *Jinja* templates if we are planning to use filters extensively.

Next, we will discuss how to update an interface's description.

## Updating the interface's description

To configure an interface attribute such as **description**, we must use the YANG model available at <http://cisco.com/ns/yang/Cisco-IOS-XR-ifmgr-cfg>.

Moreover, the XML block for configuring an interface is different than the XML block we used for getting the interface's configuration. For updating an interface, we must use the following template, which we have defined in a separate file:

```
<!--config-template.xml-->

<config xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0">
 <interface-configurations xmlns="http://cisco.com/ns/yang/ Cisco-IOS-XR-ifmgr-cfg">
 <interface-configuration>
 <active>act</active>
 <interface-name>{int_name}</interface-name>
 <description>{int_desc}</description>
 </interface-configuration>
 </interface-configurations>
</config>
```

In this template, we set the placeholders for the **name** and **description** properties of the interface.

Next, we will write a Python program that will read this template and call the **edit-config** NETCONF operation by using the **edit\_config** method of the **ncclient** library. This will push the template to the candidate database of the device:

```
#config_cisco_int_desc.py
from ncclient import manager
nc_template = open("config-template.xml").read()
nc_payload =
nc_template.format(int_name='Loopback0', int_desc="Configured by
NETCONF")
with manager.connect(host='device_ip', username=xxxx, password=xxx,
hostkey_verify=False) as nc:
 netconf_reply = nc.edit_config(nc_payload, target="candidate")
 print(netconf_reply)
 reply = nc.commit()
 print(reply)
```

It is important to highlight two things here. First, the Cisco IOS XR device has been configured to only accept the new configuration through the candidate database. If we try to set the **target** attribute

to **running**, it will fail. Second, we must call the **commit** method after the **edit-config** operation in the same session to make the new configuration operational. The output of this program will be two OK replies from the NETCONF server, as follows:

```
<?xml version="1.0"?>
<rpc-reply message-id="urn:uuid:6d70d758-6a8e-407d-8cb8-10f500e9f297"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<ok/>
</rpc-reply>
<?xml version="1.0"?>
<rpc-reply message-id="urn:uuid:2a97916b-db5f-427d-9553-de1b56417d89"
xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<ok/>
</rpc-reply>
```

This concludes our discussion of using Python for NETCONF operations. We covered two main operations (**get-config** and **edit-config**) of NETCONF with the **ncclient** library.

In the next section, we will look at integrating with network management systems using Python.

## Integrating with network management systems

Network management systems or network controllers are systems that offer network management applications with **graphical user interfaces (GUIs)**. These systems include applications such as network inventory, network provisioning, fault management, and mediation with network devices. These systems communicate with network devices using a combination of communication protocols such as SSH/NETCONF for network provisioning, SNMP for alarms and device monitoring, and gRPC for telemetry data collection. These systems also offer automation capabilities through their scripting and workflow engines.

The most value-added aspect of these systems is that they aggregate the network device's functionality into a single system (itself) and then offer it through its **North Bound Interfaces (NBIs)**, which are typically REST or RESTCONF interfaces. These systems also offer notifications of real-time events such as alarms through an event-based system such as Apache Kafka. In this section, we will discuss a couple of examples of using the REST API of a NMS We will explore how to integrate with Apache Kafka using Python in *Integrating with event driven systems* section.

To work with a NMS we will use a shared lab offered by Nokia's online developer portal (<https://network.developer.nokia.com/>). This lab has a few Nokia IP routers and an NSP. This shared lab is offered free of charge for a limited time (3 hours per day at the time of writing this book). You will be required to create an account with the developer portal free of charge. When you book a lab for use, you will receive an email with instructions on how to connect to the lab, along with the necessary VPN details. If you are a network engineer and you have access to any other NMS or a controller, you can use that system for the exercises in this section by making the appropriate adjustments.

To consume a REST API from Nokia NSP, we need to interact with the REST API Gateway, which manages several API endpoints for Nokia NSP. We can start working with the REST API Gateway by using location services, as explained next.

## Using location services endpoints

To understand what API endpoints are available, Nokia NSP offers a location services endpoint that provides a list of all API endpoints. To consume any REST API in this section, we will use the **requests** library from Python. The **requests** library is well-known for sending HTML requests to a server using the HTTP protocol, and we have used it in previous chapters. To get a list of API endpoints from the Nokia NSP system, we will use the following Python code to invoke a location services API:

```
#location_services1.py
import requests
payload = {}
headers = {}
url = "https://<NSP URL>/rest-gateway/rest/api/v1/location/ services"
resp = requests.request("GET", url, headers=headers, data=payload)
print(resp.text)
```

This API response will provide you with a few dozen API endpoints in JSON format. You can check the online documentation of Nokia NSP at <https://network.developer.nokia.com/api-documentation/> to understand how each API works. If we are looking for a specific API endpoint, we can change the value of the **url** variable in the aforementioned code example, as follows:

```
url = "https://<NSP URL>/rest-gateway/rest/api/v1/ location/services/endpoints?
endPoint=/v1/auth/token
```

By using this new API URL, we are trying to find an API endpoint for the authorization token (**/v1/auth/token**). The output of the code example with this new URL is as follows:

```
{
 "response": {
 "status": 0,
 "startRow": 0,
 "endRow": 0,
 "totalRows": 1,
 "data": {
 "endpoints": [
 {
 "docUrl": "https://<NSP_URL>/rest-gateway/api-docs#/authent..",
 "effectiveUrl": "https://<NSP_URL>/rest-gateway/rest/api",
 "operation": "[POST]"
 }
]
 },
 "errors": null
 }
}
```

Note that no authentication is required to use the location services API. However, we will need an authentication token to call any other API. In the next section, we will learn how to get an authentication token.

## Getting an authentication token

As a next step, we will use **effectiveUrl** from the output of the previous code example to get the authentication token. This API requires that we pass the *base64* encoding of **username** and **password** as the **Authorization** attribute of the HTTP header. The Python code to call this authentication API is as follows:

```
#get_token.py
import requests
from base64 import b64encode
import json
```

```

#getting base64 encoding

message = 'username' + ':' + 'password'
message_bytes = message.encode('UTF-8')
basic_token = b64encode(message_bytes)
payload = json.dumps({
 "grant_type": "client_credentials"
})
headers = {
 'Content-Type': 'application/json',
 'Authorization': 'Basic {}'.format(str(basic_token,'UTF-8'))
}
url = "https://<NSP SERVER URL>/rest-gateway/rest/api/v1/auth/ token"
resp = requests.request("POST", url, headers=headers, data=payload)
token = resp.json()["access_token"]
print(resp)

```

When executing this Python code, we will get a token for one hour to be used for any NSP API.

```
{
 "access_token": "VEt0LVNBTFhZDQ3MzE5ZjQtNWUxZjQ0YjN1",
 "refresh_token": "UkVUS04tU0FNcWF5ZlMTmQ0ZTA5MDNlOTY=",
 "token_type": "Bearer",
 "expires_in": 3600
}
```

There is also a refresh token available that can be used to refresh the token before it expires. A best practice is to refresh your token every 30 minutes. We can refresh our token using the same authentication token API, but send the following attributes in the body of the HTTP request:

```

payload = json.dumps({
 "grant_type": "refresh_token",
 "refresh_token": "UkVUS04tU0FNcWF5ZlMTmQ0ZTA5MDNlOTY="
})

```

Another good practice is to revoke the token when there is no further need for it. This can be achieved by using the following API endpoint:

```
url = "https://<NSP URL>/rest-gateway/rest/api/v1/auth/ revocation"
```

## Getting network devices and an interface inventory

Once we have received the authentication token, we can use the REST API to get configured data, as well as to add a new configuration. We will start with a simple code example that will get a list of all the network devices in a network that is managed by NSP. In this code example, we will use the token we have already retrieved using the token API:

```
#get_network_devices.py

import requests

pload={}
headers = {
 'Authorization': 'Bearer {token}'.format(token)
}

url = "https://{{NSP_URL}}:8544/NetworkSupervision/rest/api/v1/ networkElements"
response = requests.request("GET", url, headers=headers, data=pload)

print(response.text)
```

The output of this program will be a list of network devices with network device attributes. We skipped showing the output due to this being a large set of data.

In the following code example, we will show how to get a list of device ports (interfaces) based on a filter. Note that we can apply filters to network devices as well. For this code example, we will ask the NSP API to give us a list of ports based on the port name (**Port 1/1/1**, in our case):

```
#get_ports_filter.py

import requests

payload={}
headers = {
 'Authorization': 'Bearer {token}'.format(token)
}

url = "https://{{server}}:8544/NetworkSupervision/rest/api/v1/ ports?filter=(name='Port
1/1/1')"
response = requests.request("GET", url, headers=headers, data=payload)

print(response.text)
```

The output of this program will be a list of device ports called **Port 1/1/1** from all network devices. Getting ports across multiple network devices with a single API is the real value of working with a NMS

Next, we will discuss how to update a network resource using the NMS API.

## Updating the network device port

Creating new objects or updating existing objects is also convenient when using the NMS API. We will implement a case of updating the port description, as we did in previous code examples, using **Netmiko**, **NAPALM**, and **ncclient**. To update a port or interface, we will use a different API endpoint that is available from the **Network Function Manager for Packet (NFMP)** module. NFMP is an NMS module for Nokia network devices under the Nokia NSP platform. Let's look at the steps of updating a port description or making any change to a network resource:

1. To update an object or create a new object under an existing object, we will need the **Object Full Name (OFN)**, also known as the **Fully Distinguishable Name (FDN)**, of any existing object (for updating the object) or a parent object (for creating a new object). This OFN or FDN acts as a primary key for identifying an object uniquely. For Nokia network objects available under the NSP modules, every object has an OFN or FDN attribute. To get an OFN for a port to be updated, we will use the **v1/managedobjects/searchWithFilter** API with the following filter criteria:

```
#update_port_desc.py (part 1)

import requests
import json
token = <token obtain earlier>
headers = {
 'Content-Type': 'application/json',
 'Authorization': 'Bearer {}'.format(token)
}
url1 = "https://NFMP_URL:8443/nfm-p/rest/api/v1/ managedobjects/searchWithFilter"
payload1 = json.dumps({
 "fullClassName": "equipment.PhysicalPort",
 "filterExpression": "siteId ='<site id>' AND portName='123",
 "resultFilter": [
 "objectFullName",
 "description"
]
})
response = requests.request("POST", url1, headers=headers, data=payload1, verify=False)
port_ofn = response.json()[0]['objectFullName']
```

In this code example, we set the name of the object to **fullClassNames**. The object's full class names are available in the Nokia NFMP object model documentation. We set **filterExpression** to search for a unique port based on the device site's ID and port name. The **resultFilter** attribute is

used to limit the attributes that are returned by the API in the response. We are interested in the **objectFullName** attribute in the response of this API.

2. Next, we will use a different API endpoint called **v1/managedobjects/ofn** to update an attribute of the network object. In our case, we are only updating the description attribute. For the update operation, we must set the **fullClassName** attribute in the payload and a new value for the description attribute. For the API endpoint's URL, we will concatenate the **port\_ofn** variable that we computed in the previous step. The sample code for this part of the program is as follows:

```
#update_port_desc.py (part 2)

payload2 = json.dumps({
 "fullClassName": "equipment.PhysicalPort",
 "properties": {
 "description": "description added by a Python program"
 }
})

url2 = "https:// NFMP_URL:8443/nfm-p/rest/api/v1/ managedobjects/" + port_ofn
response = requests.request("PUT", url2, headers=headers, data=payload2,
verify=False)
print(response.text)
```

Network automation is the process of creating and updating many network objects in a specific order. For example, we can update a port before creating an IP connectivity service to connect two or more local area networks. This type of use case requires that we perform a series of tasks to update all the ports involved, as well as many other objects. With the NMS API, we can orchestrate all these tasks in a program to implement an automated process.

In the next section, we will explore how to integrate with Nokia NSP or similar systems for event-driven communication.

## Integrating with event-driven systems

In the previous sections, we discussed how to interact with network devices and network management systems using the request-response model. In this model, a client sends a request to a server and the server sends a response as a reply to the request. The HTTP (REST API) and SSH protocols are based on a request-response-based model. This model works well for configuring a system or getting the operational state of the network on an ad hoc basis or periodically. But what about if something happens in the network that requires the operation team's attention? For example, let's say a hardware failure on a device or a line cable has been cut. Network devices typically raise

alarms in such situations, and these alarms have to reach the operator (via an email, an SMS, or a dashboard) immediately.

We can use the request-response model to poll the network device every second (or every few seconds) to check if there has been any change in the state of a network device or if there is a new alarm. However, this is not an efficient use of the network device's resources and will contribute to unnecessary traffic in the network. What about if the network device or NMS itself reaches out to the interested clients whenever there is a change in the state of critical resources, or an alarm is raised? This type of model is called an *event-driven* model, and it is a popular communication approach for sending real-time events.

Event-driven systems can be implemented either using **webhooks/WebSockets** or using the **streaming** approach. WebSockets offers a bidirectional transport channel over HTTP 1.1 through a TCP/IP socket. Since this bidirectional connection is not using the traditional request-response model, WebSockets is an efficient approach when we want to establish a one-to-one connection between the two systems. This is one of the best options when we need real-time communication between the two programs. WebSockets are supported by all standard browsers, including the one that's available with iPhone and Android devices. It is also a popular choice for many social media platforms, streaming applications, and online gaming.

WebSockets is a lightweight solution for getting real-time events. But when many clients are looking to receive events from one system, using a streaming approach is scalable and efficient. The streaming event-based model typically follows a publisher-subscriber design pattern and has three main components, as described here:

- **Topic:** All streaming messages or event notifications are stored under a topic. We can think of a topic as a directory. This topic helps us subscribe to topics of interest to help us avoid receiving all events.
- **Producer:** This is a program or piece of software that pushes the events or messages to a topic. This is also called the **publisher**. In our case, it will be an NSP application.
- **Consumer:** This is a program that fetches the events or messages from a topic. This is also called a **subscriber**. In our case, this will be a Python program we will write.

Event-driven systems are available for network devices, as well as network management systems. NMS platforms use event systems such as gRPC or SNMP to receive real-time events from network devices, and they offer aggregated interfaces for the orchestration layer or the operational or monitoring applications. For our example, we will interact with an event system from the Nokia NSP platform. The Nokia NSP system offers an event system based on Apache Kafka. Apache Kafka is an open source piece of software that was developed in Scala and Java, and it provides the implementation of a software messaging bus that is based on the **Publisher-Subscriber** design

pattern. Before interacting with Apache Kafka, we will enumerate a list of key **categories** (a term used for topics in Apache Kafka) offered through Nokia NSP, as follows:

- **NSP-FAULT**: This category covers events related to faults or alarms.
- **NSP-PACKET-ALL**: This category is used for all network management events, including keep-alive events.
- **NSP-REAL-TIME-KPI**: This category represents events for real-time streaming notifications.
- **NSP-PACKET-STATS**: This category is used for statistics events.

A full list of categories is available in Nokia NSP documentation. All these categories offer additional filters for subscribing to a certain type of event. In the context of Nokia NSP, we will be interacting with Apache Kafka to create a new subscription and then process the events from the Apache Kafka system. We will start with subscription management.

## Creating subscriptions for Apache Kafka

Before receiving any event or message from Apache Kafka, we must subscribe to a topic or a category. Note that one subscription is only valid for one category. A subscription typically expires after 1 hour, so it is recommended to renew a subscription 30 minutes before its expiry time.

To create a new subscription, we will use the **v1/notifications/subscriptions** API and the following sample code to get a new subscription:

```
#subscribe.py

import requests
token = <token obtain earlier>
url = "https://NSP_URL:8544/nbi-notification/api/v1/ notifications/subscriptions"
def create_subscription(category):
 headers = {'Authorization': 'Bearer {}'.format(token) }
 payload = {
 "categories": [
 {
 "name": "{}".format(category)
 }
]
 }
 response = requests.request("POST", url, json=payload,
 headers=headers, verify=False)
 print(response.text)
```

```

if __name__ == '__main__':
 create_subscription("NSP-PACKET-ALL")
}

The output of this program will include important attributes such as subscriptionId, topicId, and expiresAt, among others, as shown here:

{
 "response": {
 "status": 0,
 "startRow": 0,
 "endRow": 0,
 "totalRows": 1,
 "data": {
 "subscriptionId": "440e4924-d236-4fba-b590-a491661aae14",
 "clientId": null,
 "topicId": "ns-eg-440e4924-d236-4fba-b590-a491661aae14",
 "timeOfSubscription": 1627023845731,
 "expiresAt": 1627027445731,
 "stage": "ACTIVE",
 "persisted": true
 },
 "errors": null
 }
}

```

The **subscriptionId** attribute is used to renew or delete a subscription later. Apache Kafka will create a topic specifically for this subscription. It is provided to us as a **topicId** attribute. We will use this **topicId** attribute to connect to Apache Kafka to receive events. This explains why we call general topics categories in Apache Kafka. The **expiresAt** attribute indicates the time this subscription will expire.

Once a subscription is ready, we can connect to Apache Kafka to receive events, as explained in the next subsection.

## Processing events from Apache Kafka

Writing a basic Kafka consumer takes no more than a few lines of Python code with the **kafka-python** library. To create a Kafka client, we will use the **KafkaConsumer** class from the **kafka-**

**python** library. We can use the following sample code to consume events for our subscription topic:

```
#basic_consumer.py

topicid = 'ns-eg-ff15a252-f927-48c7-a98f-2965ab6c187d'
consumer = KafkaConsumer(topic_id,
 group_id='120',
 bootstrap_servers=[host_id], value_
 deserializer=lambda m: json.loads
 (m.decode('ascii')),
 api_version=(0, 10, 1))

try:
 for message in consumer:
 if message is None:
 continue
 else:
 print(json.dumps(message.value, indent=4, sort_
 keys=True))
except KeyboardInterrupt:
 sys.stderr.write('++++++ Aborted by user ++++++\n')
finally:
 consumer.close()
```

It is important to note that you must use the **kafka-python** library if you are using Python 3.7 or later. If you are using a version of Python that's earlier than 3.7, you can use the **kafka** library. There are known issues with the **kafka** library if we use it with Python 3.7 or later. For example, there is a known issue that **async** has become a keyword in Python 3.7 or later releases, but it has been used as a variable in the **kafka** library. There are also API version issues when using the **kafka-python** library with Python 3.7 or later. These can be avoided by setting a correct API version as an argument (the **0.10.0** version, in this case).

In this section, we showed you a basic Kafka consumer, but you can explore a more sophisticated example in the source code provided with this book by going to [https://github.com/nokia/NSP-Integration-Bootstrap/tree/master/kafka/kafka\\_cmd\\_consumer](https://github.com/nokia/NSP-Integration-Bootstrap/tree/master/kafka/kafka_cmd_consumer).

## Renewing and deleting a subscription

We can renew a subscription with the Nokia NSP Kafka system using the same API endpoint that we used to create a subscription. We will add the **subscriptionId** attribute at the end of the URL, along

with the **renewals** resource, as follows:

```
https://{{server}}:8544/nbi-notification/api/v1/notifications/subscriptions/<subscriptionId>/renewals
```

We can delete a subscription using the same API endpoint with the **subscriptionId** attribute at the end of the URL but using the HTTP **Delete** method. This API endpoint will look as follows for a delete request:

```
https://{{server}}:8544/nbi-notification/api/v1/notifications/subscriptions/<subscriptionId>
```

In both cases, we will not send any arguments in the request body.

This concludes our discussion on integrating with NMS and network controllers using both the request-response model and the event-driven model. Both these approaches will give you a good starting point when it comes to integrating with other management systems.

## Summary

In this chapter, we introduced network automation, along with its benefits and the challenges it provides for telecom service providers. We also discussed the key use cases of network automation. After this introduction, we discussed the transport protocols that are available for network automation to interact with network devices. Network automation can be adopted in many ways. We started by looking at how to directly interact with network devices using the SSH protocol in Python. We used the Paramiko, Netmiko, and NAPALM Python libraries to fetch configuration from a device and we elaborated on how to push this configuration to a network device. Next, we discussed how to use NETCONF with Python to interact with a network device. We provided code examples for working with NETCONF and used the ncclient library to fetch an IP interface configuration. We also used the same library to update an IP interface on a network device.

In the last part of this chapter, we explored how to interact with network management systems such as Nokia NSP. We interacted with the Nokia NSP system using Python as a REST API client and as a Kafka consumer. We provided a few code examples in terms of how to get an authentication token, and then sent a REST API to a NMS to retrieve configuration data and update the network configuration on devices.

This chapter included several code examples to make you familiar with using Python for interacting with devices using SSH, NETCONF protocols, and using an NMS-level REST API. This practical knowledge is critical if you are an automation engineer and looking to excel in your area by using Python capabilities.

This chapter concludes this book. We not only covered the advanced concepts of Python but also provided an insight into using Python in many advanced areas, such as data processing, serverless computing, web development, machine learning, and network automation.

## Questions

1. What is the name of the commonly used class from the Paramiko library for making a connection to a device?
2. What are the four layers of NETCONF?
3. Can you push configuration directly to a **running** database in NETCONF?
4. Why is gNMI better for data collection than network configuration?
5. Does RESTCONF provide the same features as NETCONF but through REST interfaces?
6. What are a publisher and consumer in Apache Kafka?

## Further reading

- *Mastering Python Networking*, by Eric Chou.
- *Practical Network Automation*, Second Edition, by Abhishek Ratan.
- *Network Programmability and Automation*, by Jason Edelman.
- *Paramiko official documentation* is available at <http://docs.paramiko.org/>.
- *Netmiko official documentation* is available at <https://ktbyers.github.io/>.
- *NAPALM official documentation* is available at <https://napalm.readthedocs.io/>.
- *ncclient official documentation* is available at <https://ncclient.readthedocs.io/>.
- *NETCONF YANG models* can be found at <https://github.com/YangModels/yang>.
- *Nokia NSP API documentation* is available at <https://network.developer.nokia.com/api-documentation/>.

## Answers

1. The **paramiko.SSHClient** class.
2. Content, Operations, Messages, and Transport.
3. If a network device does not support a **candidate** database, it typically allows direct updates to be performed for the **running** database.
4. gNMI is based on gRPC, which is a protocol that was introduced by Google for RPC calls between mobile clients and cloud applications. The protocol has been optimized for data transfer, which makes it more efficient in terms of collecting data from network devices compared to configuring them.
5. RESTCONF provides most of the functionality of NETCONF through REST interfaces but it does not expose all the operations of NETCONF.
6. The publisher is a client program that sends messages to a Kafka topic (category) as events, whereas the consumer is a client application that reads and processes the messages from a Kafka topic.





[Packt.com](https://www.packt.com)

Subscribe to our online digital library for full access to over 7,000 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
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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 [packt.com](https://www.packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub.com](mailto:customercare@packtpub.com) for more details.

At [www.packt.com](https://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.

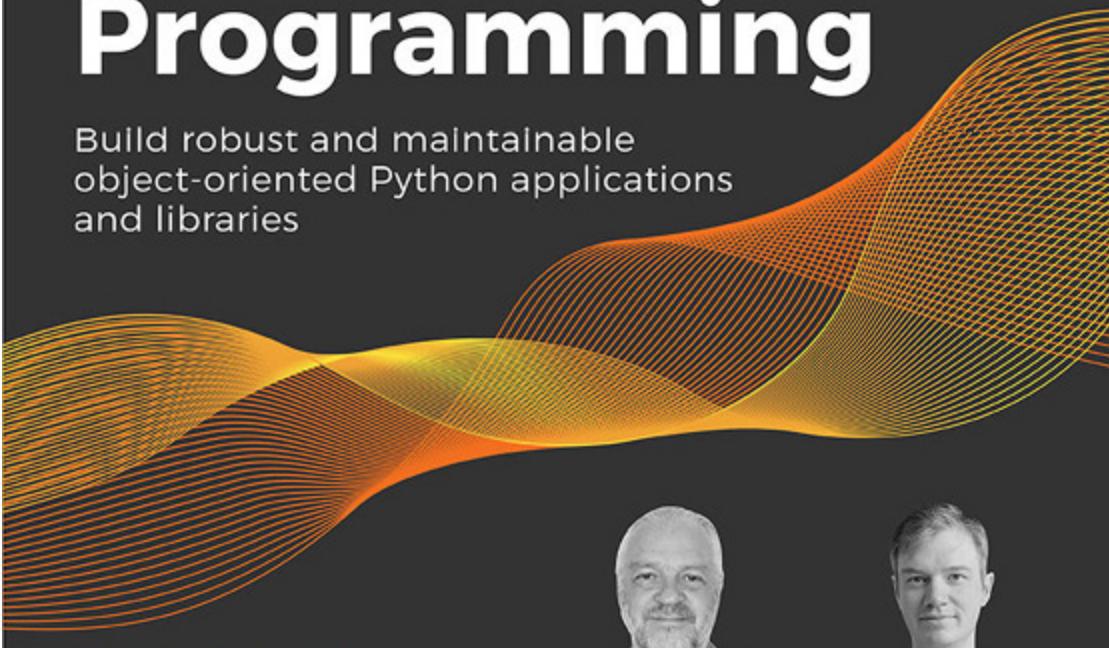
## Other Books You May Enjoy

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

EXPERT INSIGHT

# Python Object-Oriented Programming

Build robust and maintainable  
object-oriented Python applications  
and libraries



**Fourth Edition**



**Steven F. Lott  
Dusty Phillips**

**Packt**

**Python Object-Oriented Programming**

Steven F. Lott, Dusty Phillips

ISBN: 978-1-80107-726-2

- Implement objects in Python by creating classes and defining methods
- Extend class functionality using inheritance
- Use exceptions to handle unusual situations cleanly

- Understand when to use object-oriented features, and more importantly, when not to use them
- Discover several widely used design patterns and how they are implemented in Python
- Uncover the simplicity of unit and integration testing and understand why they are so important
- Learn to statically type check your dynamic code
- Understand concurrency with asyncio and how it speeds up programs

EXPERT INSIGHT

# Expert Python Programming

Master Python by learning the best coding practices and advanced programming concepts



Fourth Edition



Michał Jaworski  
Tarek Ziadé

Packt

Expert Python Programming – Fourth Edition

Michał Jaworski, Tarek Ziadé

ISBN: 978-1-80107-110-9

- Explore modern ways of setting up repeatable and consistent Python development environments
- Effectively package Python code for community and production use
- Learn modern syntax elements of Python programming, such as f-strings, enums, and lambda functions
- Demystify metaprogramming in Python with metaclasses
- Write concurrent code in Python
- Extend and integrate Python with code written in C and C++

## Packt is searching for authors like you

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

## Share Your Thoughts

Now you've finished *Python for Geeks*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here to go straight to the Amazon review page](#) for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

[OceanofPDF.com](https://OceanofPDF.com)