python™ Community Edition

MUHAMMAD ASIF

PYTHON FOR GEEKS

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

Packt>

Python for Geeks

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

Muhammad Asif

Packt

BIRMINGHAM—MUMBAI

Python for Geeks

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.

Table of Contents

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:

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.

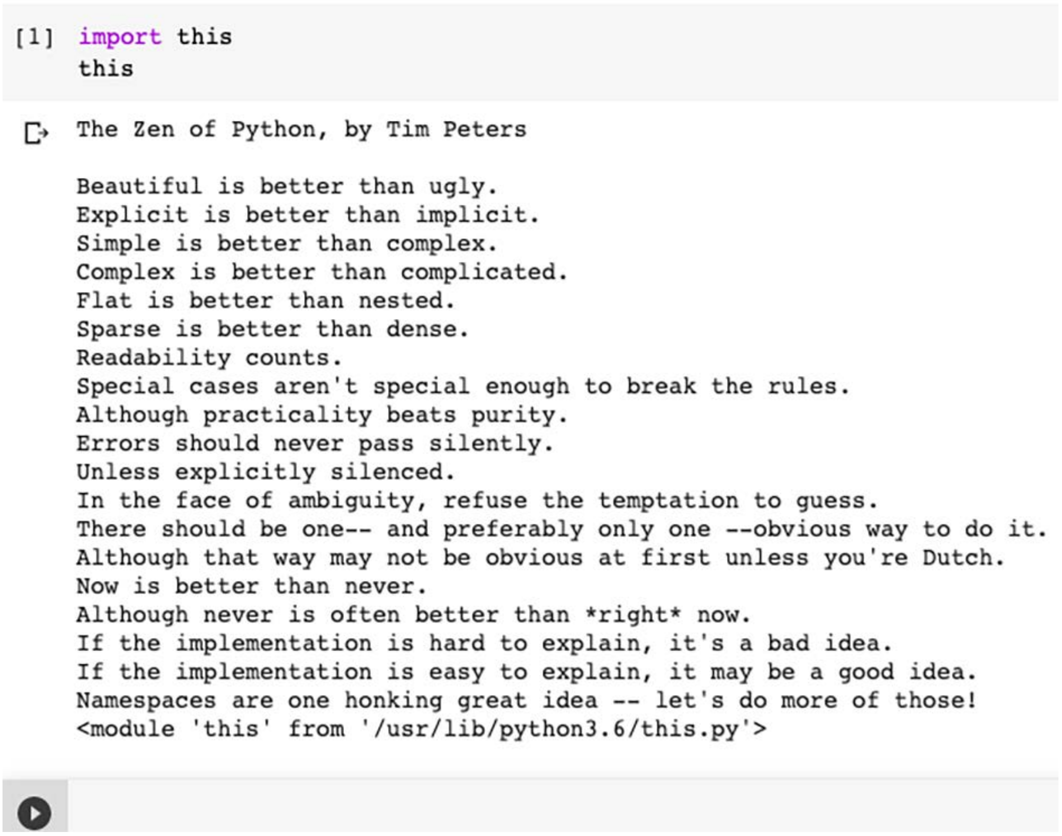
Python culture and community

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:

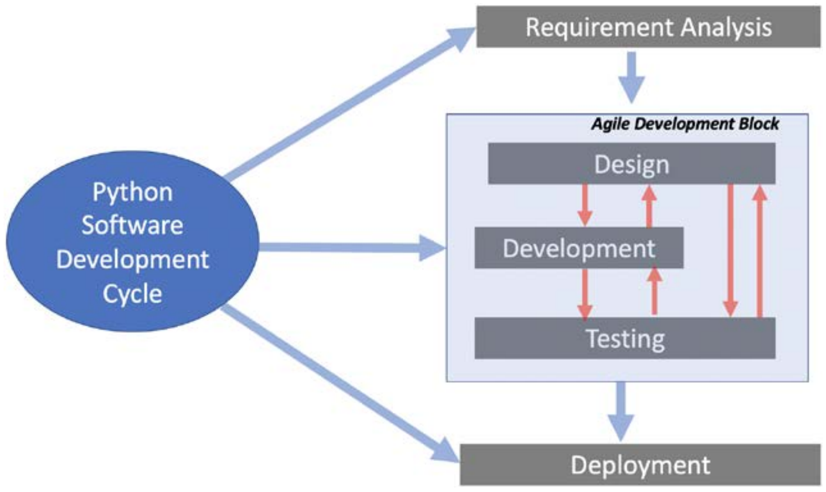


Figure 1.2 – Various phases of a Python project

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 subject 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:

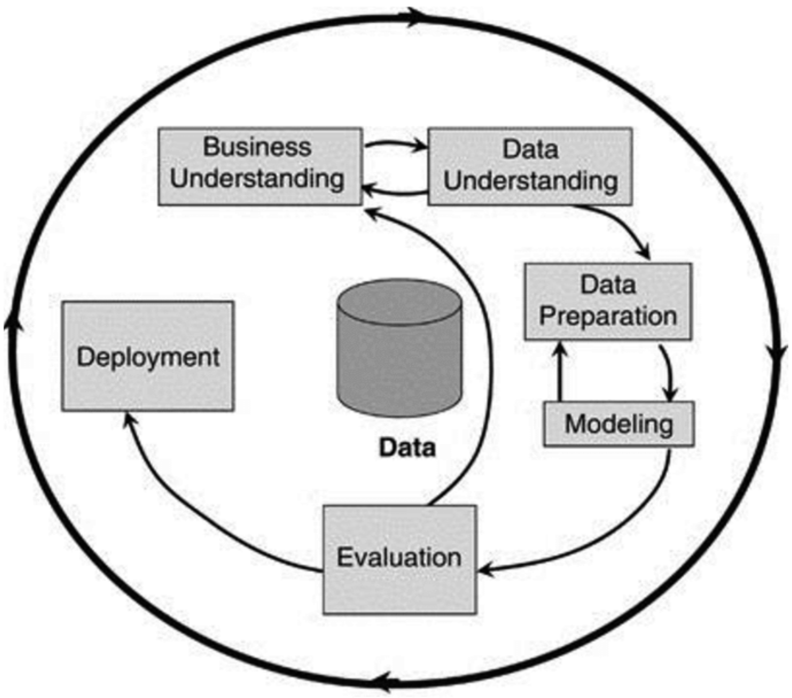


Figure 1.3 – A CRISP-DM life cycle

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 propriety 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.

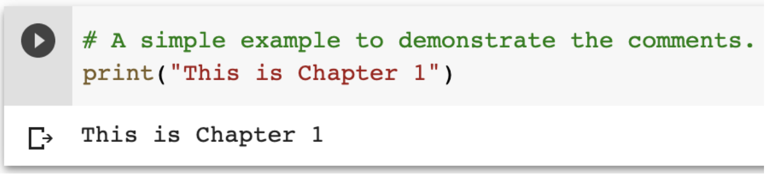
Now, let's look into effective ways of developing Python code.

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:



```
# A simple example to demonstrate the comments.
print("This is Chapter 1")
```

```
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

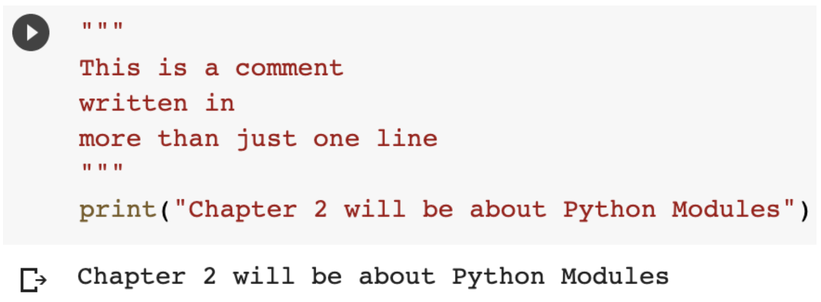
While developing the code, various types of documentation need to be produced, including the following:

- 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:



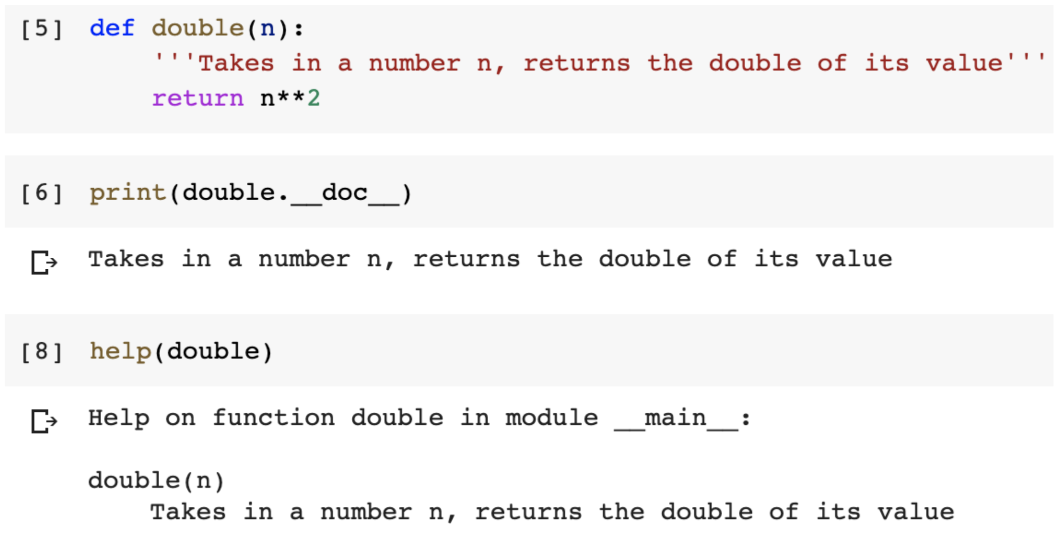
```
"""
This is a comment
written in
more than just one line
"""
print("Chapter 2 will be about Python Modules")
```

```
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)
```

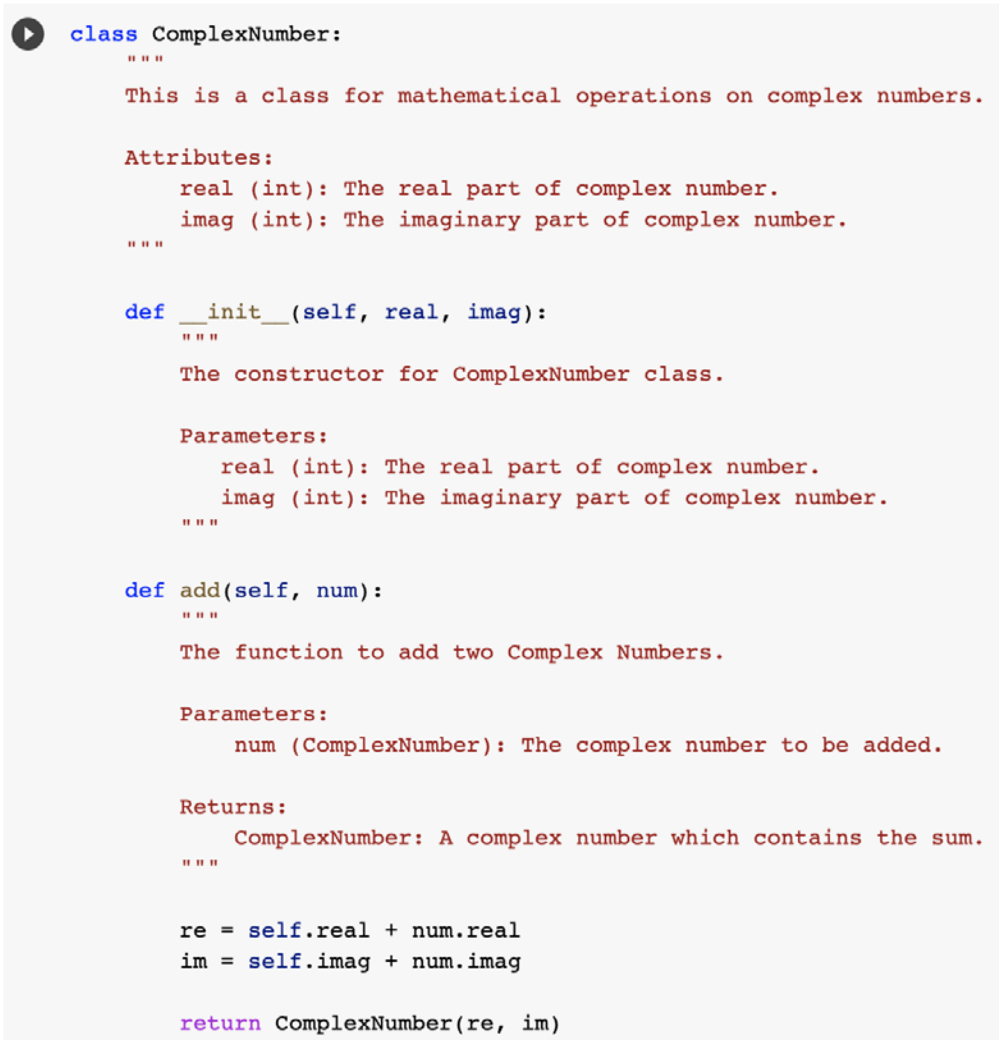
```
Help on function double in module __main__:

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:

  ```
  FileNotFoundError
  ```

- 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.

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 `funct1` 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:

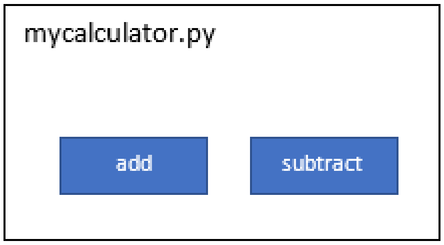


Figure 2.1 – The mycalculator module with add and subtract functions

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:

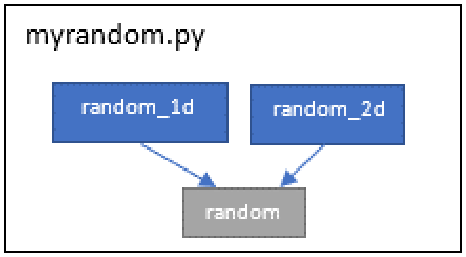


Figure 2.2 – The myrandom module with function dependency on the random library

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:

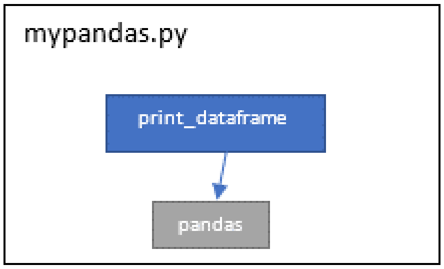


Figure 2.3 – The mypandas module with dependency on a third-party pandas library

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:

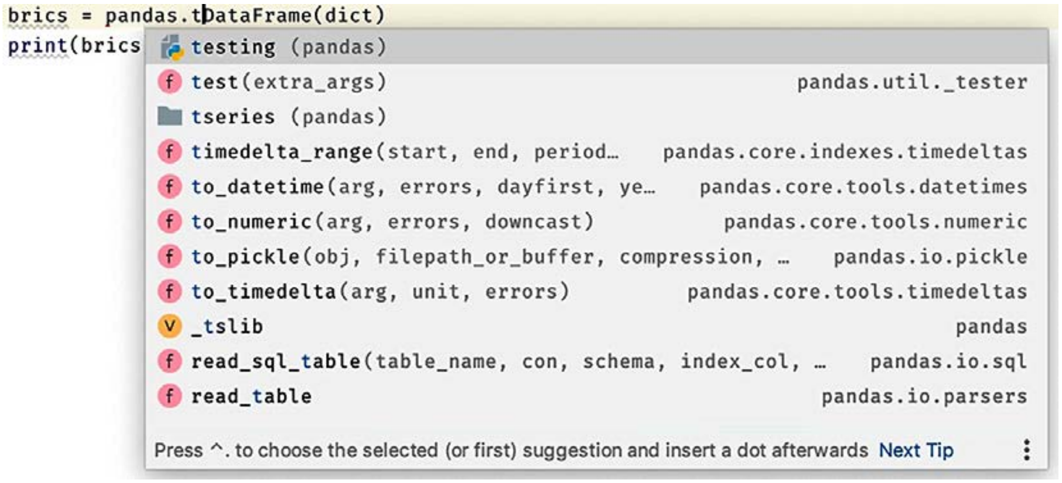
```
def addNumbers(numParam1, numParam2)
   #function code is omitted
Def featureCount(moduleName)
   #function code is omitted
```

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:



```
brics = pandas.tDataFrame(dict)
print(brics
    testing (pandas)
    test(extra_args)                                         pandas.util._tester
    tseries (pandas)
    timedelta_range(start, end, period…      pandas.core.indexes.timedeltas
    to_datetime(arg, errors, dayfirst, ye…      pandas.core.tools.datetimes
    to_numeric(arg, errors, downcast)             pandas.core.tools.numeric
    to_pickle(obj, filepath_or_buffer, compression, …     pandas.io.pickle
    to_timedelta(arg, unit, errors)            pandas.core.tools.timedeltas
    _tslib                                                            pandas
    read_sql_table(table_name, con, schema, index_col, …     pandas.io.sql
    read_table                                             pandas.io.parsers
    Press ^. to choose the selected (or first) suggestion and insert a dot afterwards  Next Tip
```

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

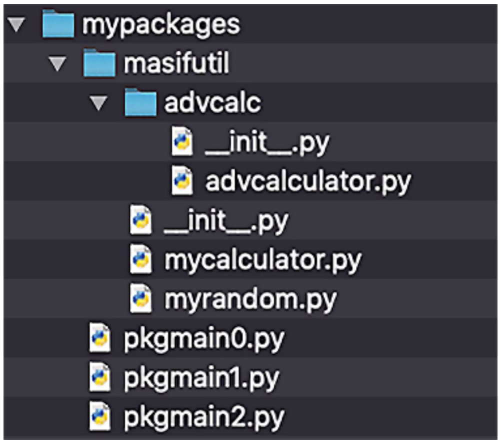
Now we will discuss how to build a package with one sample package example. We will build a `masifutil` package using the following modules and a sub-package:

- 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:



```
mypackages
    masifutil
        advcalc
            __init__.py
            advcalculator.py
        __init__.py
        mycalculator.py
        myrandom.py
    pkgmain0.py
    pkgmain1.py
    pkgmain2.py
```

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

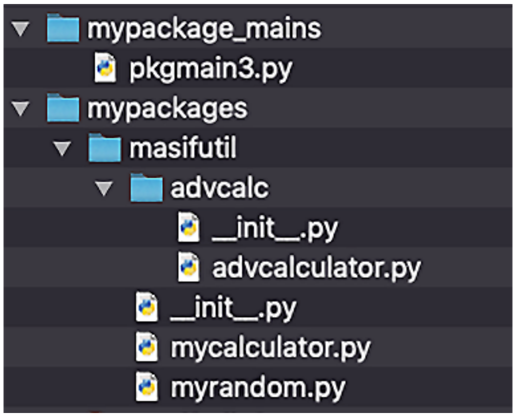
The package we built in the previous subsection is accessible only if the program calling the modules is at the same level as the package location. This requirement is not practical for code reusability and code sharing.

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:



```
▼ mypackage_mains
    pkgmain3.py
▼ mypackages
  ▼ masifutil
    ▼ advcalc
        __init__.py
        advcalculator.py
      __init__.py
      mycalculator.py
      myrandom.py
```

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:

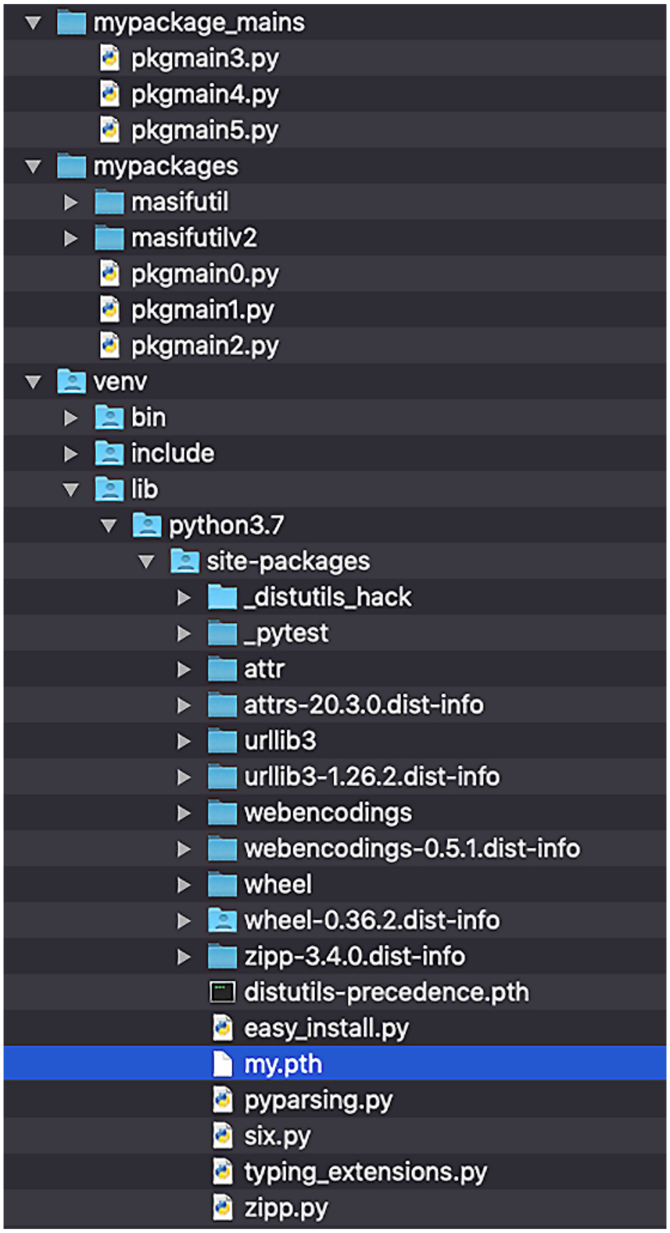


Figure 2.7 – A view of a virtual environment with the my.pth file

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.

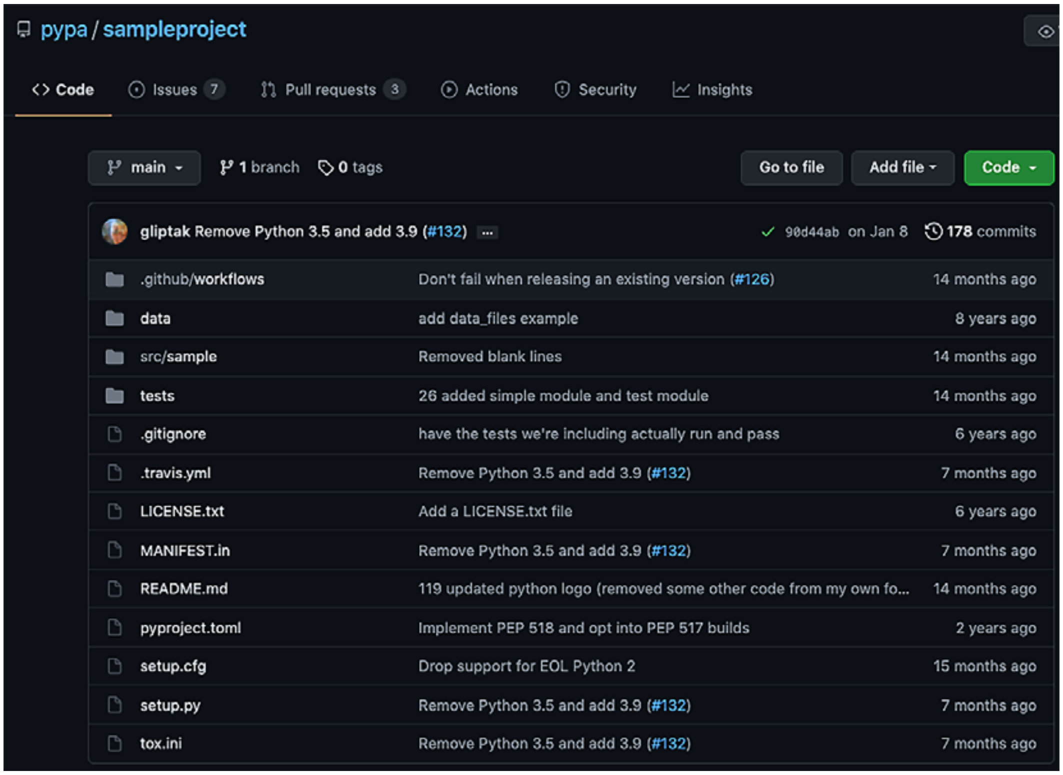
In this section, we will be covering installing and distributing packaging techniques. A few tools that we will use or are at least worth mentioning in this section as a reference are as follows:

- **distutils**: This comes with Python with base functionality. It is not easy to extend for complex and custom package distribution.
- **setuputils**: 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:



gliptak Remove Python 3.5 and add 3.9 (#132) ... ✓ 90d44ab on Jan 8 178 commits

.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

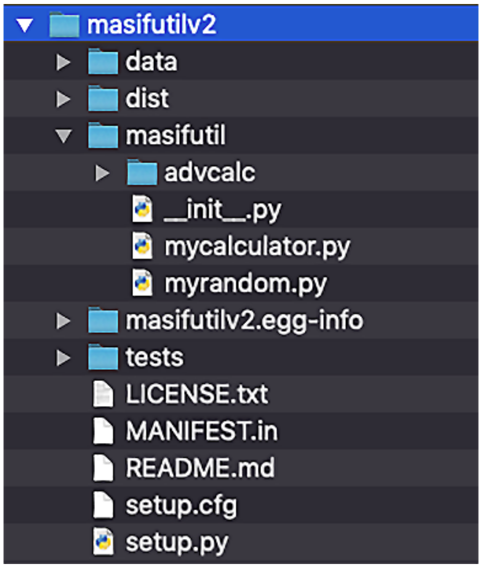
  c) Description

  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:



```
masifutilv2
    data
    dist
    masifutil
        advcalc
        __init__.py
        mycalculator.py
        myrandom.py
    masifutilv2.egg-info
    tests
    LICENSE.txt
    MANIFEST.in
    README.md
    setup.cfg
    setup.py
```

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

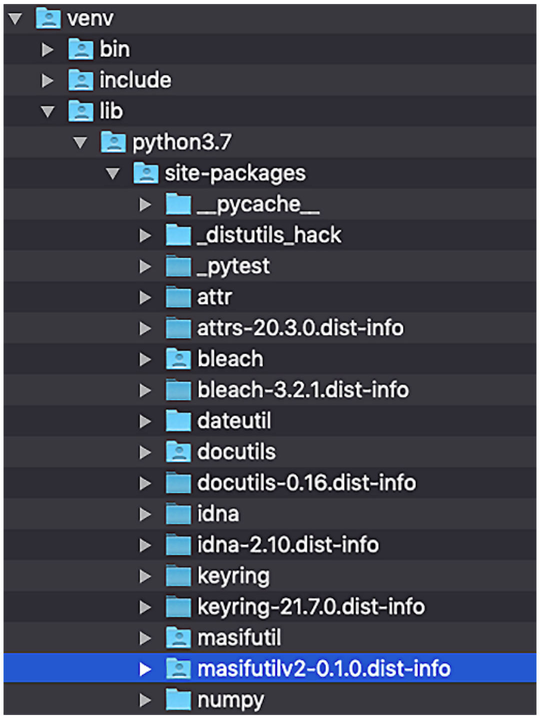
Once we have updated the package with new files, we are ready to install it using the pip utility. The simplest way to install it is by executing the following command with the path to the `masifutilv2` folder:

```
> 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:



```
▼ venv
  ▶ bin
  ▶ include
  ▼ lib
    ▼ python3.7
      ▼ site-packages
        ▶ __pycache__
        ▶ _distutils_hack
        ▶ _pytest
        ▶ attr
        ▶ attrs-20.3.0.dist-info
        ▶ bleach
        ▶ bleach-3.2.1.dist-info
        ▶ dateutil
        ▶ docutils
        ▶ docutils-0.16.dist-info
        ▶ idna
        ▶ idna-2.10.dist-info
        ▶ keyring
        ▶ keyring-21.7.0.dist-info
        ▶ masifutil
        ▶ masifutilv2-0.1.0.dist-info
        ▶ numpy
```

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`.

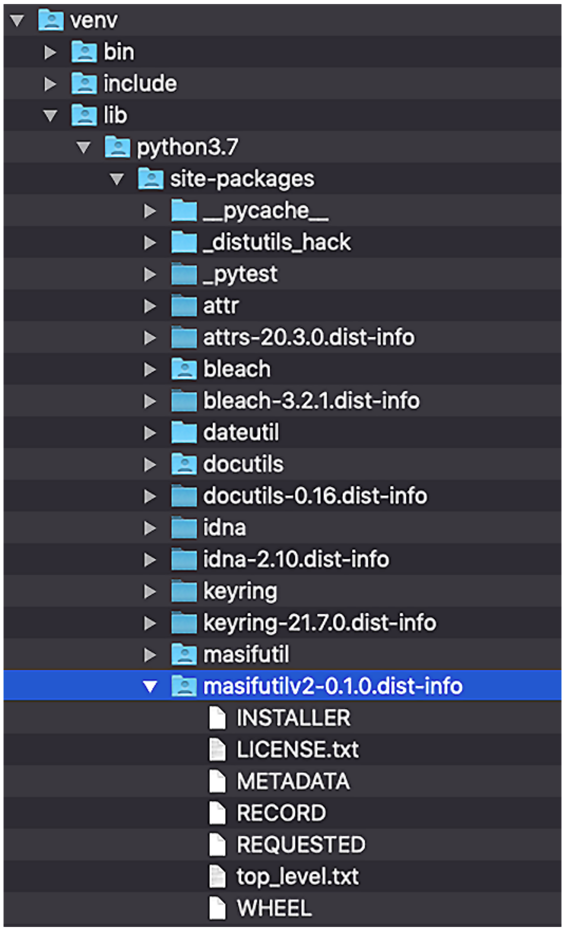
As a next step, we will install the `wheel` package and then reinstall the same package again. Here is the installation command:

```
> 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=038712975b7d7eb1f3fefa799da9e294b34
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:



```
venv
  bin
  include
  lib
    python3.7
      site-packages
        __pycache__
        _distutils_hack
        _pytest
        attr
        attrs-20.3.0.dist-info
        bleach
        bleach-3.2.1.dist-info
        dateutil
        docutils
        docutils-0.16.dist-info
        idna
        idna-2.10.dist-info
        keyring
        keyring-21.7.0.dist-info
        masifutil
        masifutilv2-0.1.0.dist-info
          INSTALLER
          LICENSE.txt
          METADATA
          RECORD
          REQUESTED
          top_level.txt
          WHEEL
```

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
  f5c9f447dbee57c0961b5f8fbf99501
  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.

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 (car.i_color)
    print (car.i_mileage)
    print (car.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 first 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:

```
#carexampl4.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 0x100caae80>
```

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:

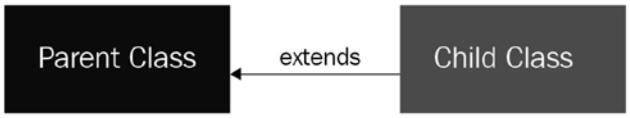


Figure 3.1 – Parent-and-child class relationship

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 `inhertance1.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**:

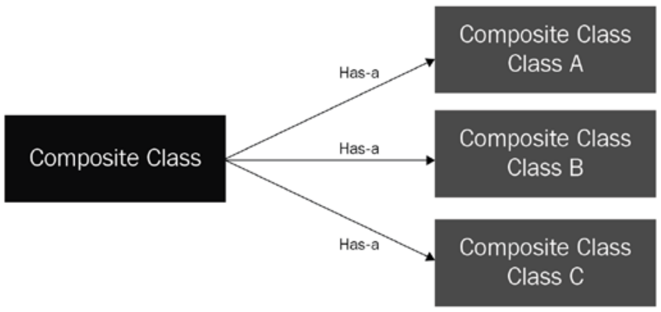


Figure 3.2 – Relationship between a composite class and its component classes

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 duct 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.

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*

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 handles 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_valu

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 occured")
    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:

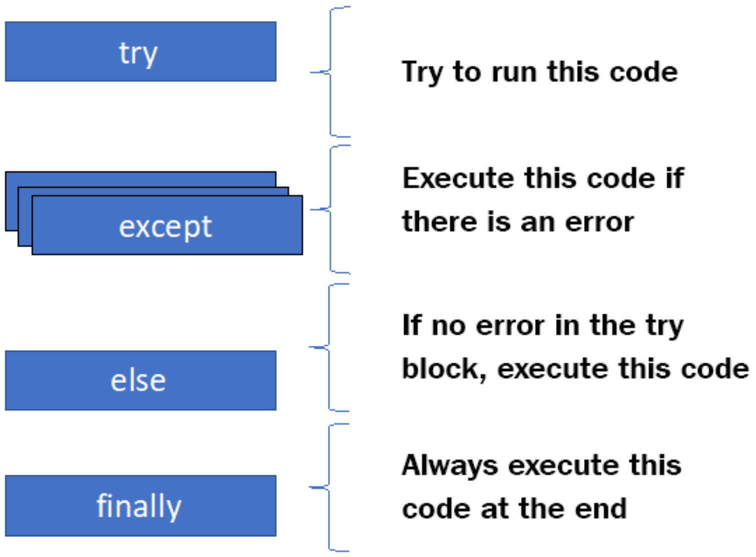

try
Try to run this code
except
Execute this code if there is an error
else
If no error in the try block, execute this code
finally
Always execute this code at the end

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:

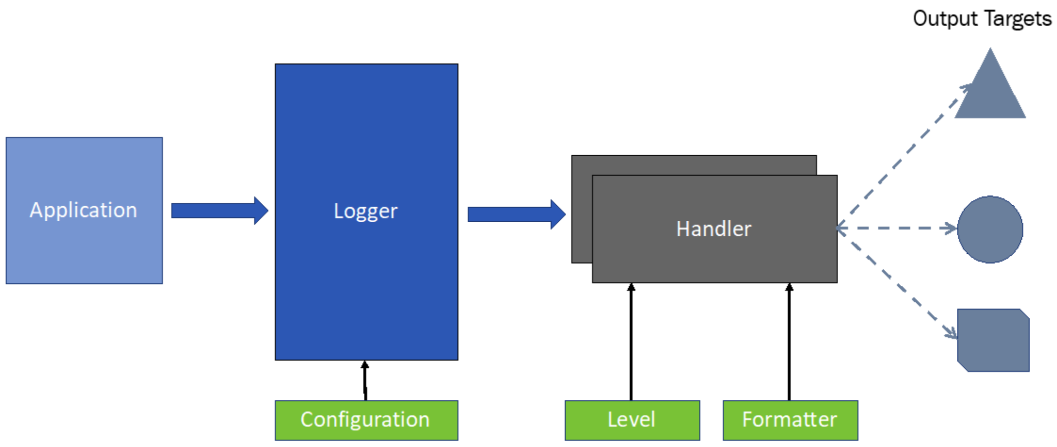


Figure 4.2 – Logging components in Python

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:

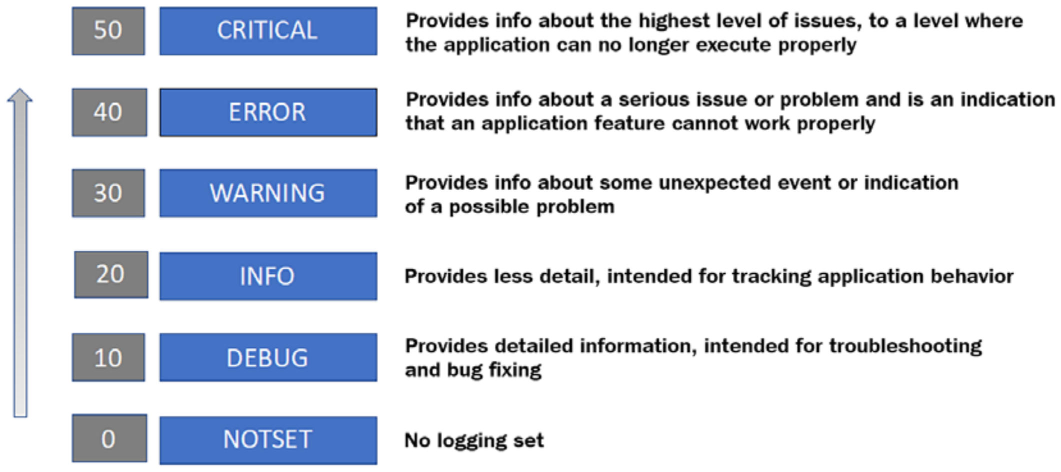

50
CRITICAL
Provides info about the highest level of issues, to a level where the application can no longer execute properly
40
ERROR
Provides info about a serious issue or problem and is an indication that an application feature cannot work properly
30
WARNING
Provides info about some unexpected event or indication of a possible problem
20
INFO
Provides less detail, intended for tracking application behavior
10
DEBUG
Provides detailed information, intended for troubleshooting and bug fixing
0
NOTSET
No logging set

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)

#creating 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.

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:

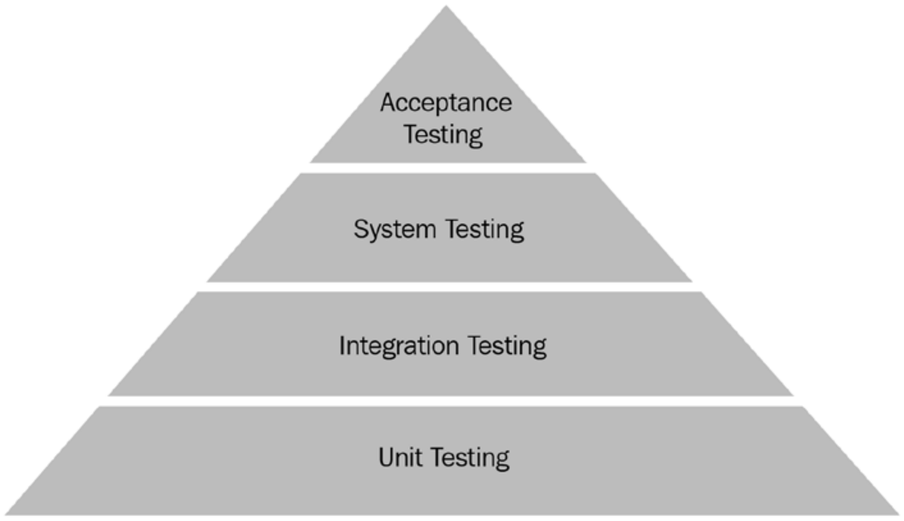


Figure 5.1 – Different levels of testing during software development

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.

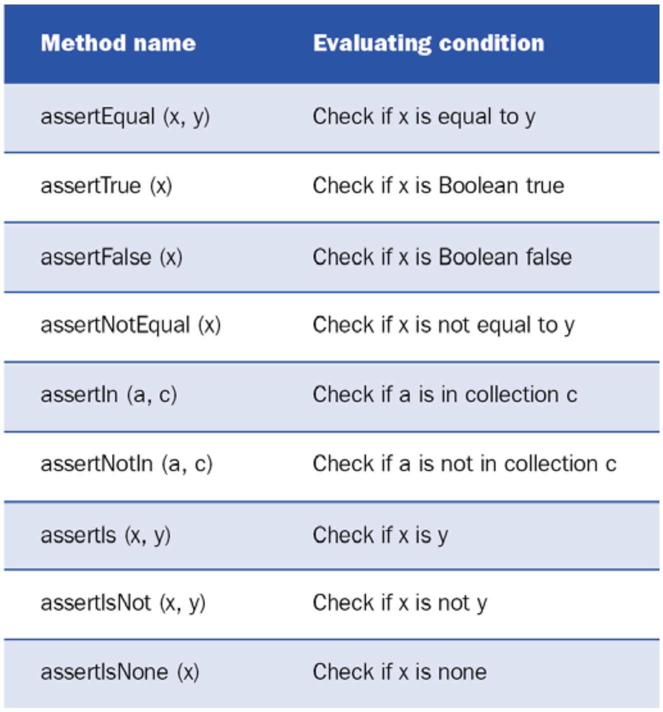
Working with the unittest framework

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
assertEqual (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')
```

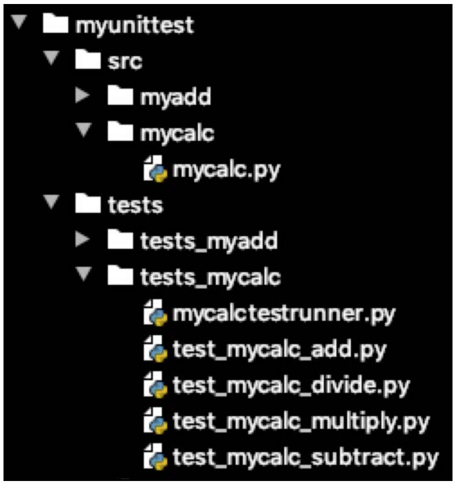
In the preceding code snippet, we added two additional test cases to the `test_add3.py` module. These test cases use the `assertRaises` method to validate if a particular type of exception is thrown or not. In our test cases, we used a single letter (`a`) or two letters (`a` and `b`) as arguments for 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:



```
myunittest
    src
        myadd
        mycalc
            mycalc.py
    tests
        tests_myadd
        tests_mycalc
            mycalctestrunner.py
            test_mycalc_add.py
            test_mycalc_divide.py
            test_mycalc_multiply.py
            test_mycalc_subtract.py
```

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 subtracts 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
Pytho...)                                                [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:

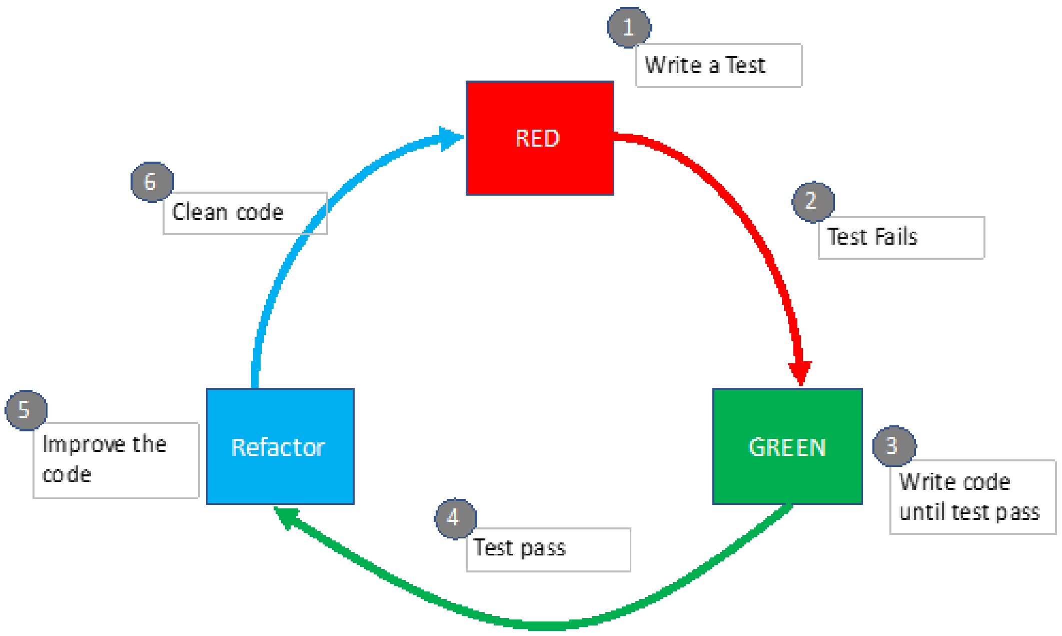


Figure 5.4 – TDD, also known as Red, Green, Refactor

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:

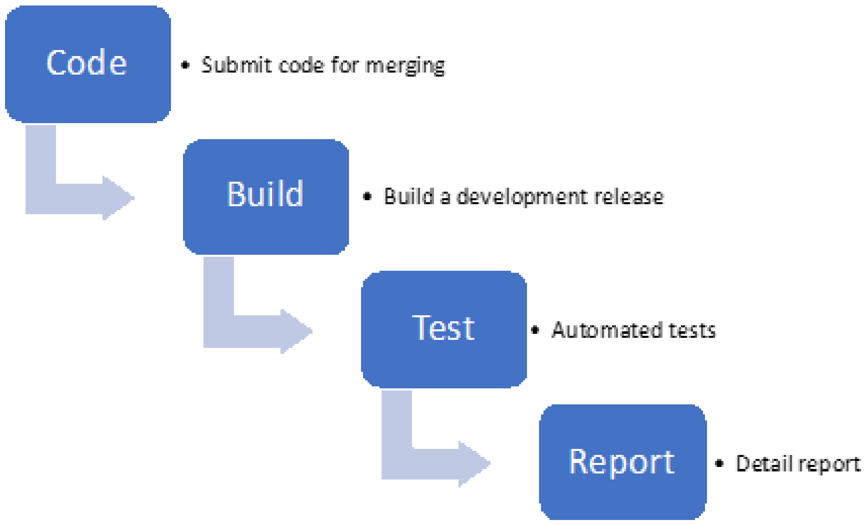


Figure 5.5 – Phases of CI testing

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.

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 `itertool` 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 `functtools` 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:

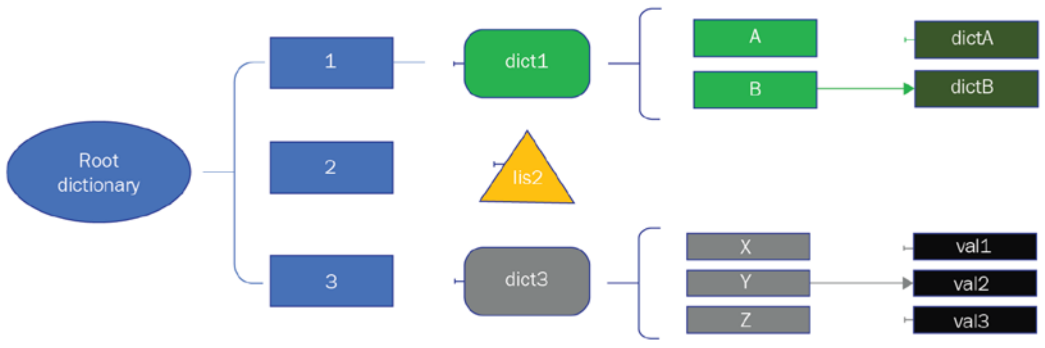


Figure 6.1: An example of a dictionary inside a dictionary

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 comrehension

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.

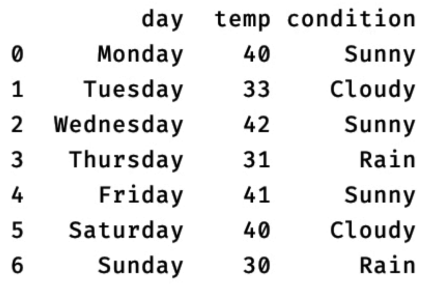
The DataFrame is the key data structure that is used in data analysis. In the remainder of this section, we will be using the DataFrame object extensively in our code examples. Before we discuss any advanced tricks regarding these pandas DataFrame objects, we will do a quick review of the fundamental 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.

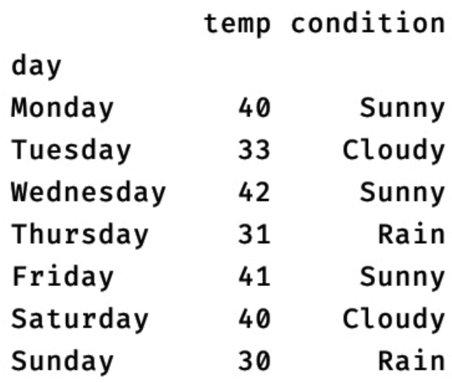
Setting a custom index

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`
- `25th percentiles, 50th percentile, 75th 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:

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 other resources with other threads:

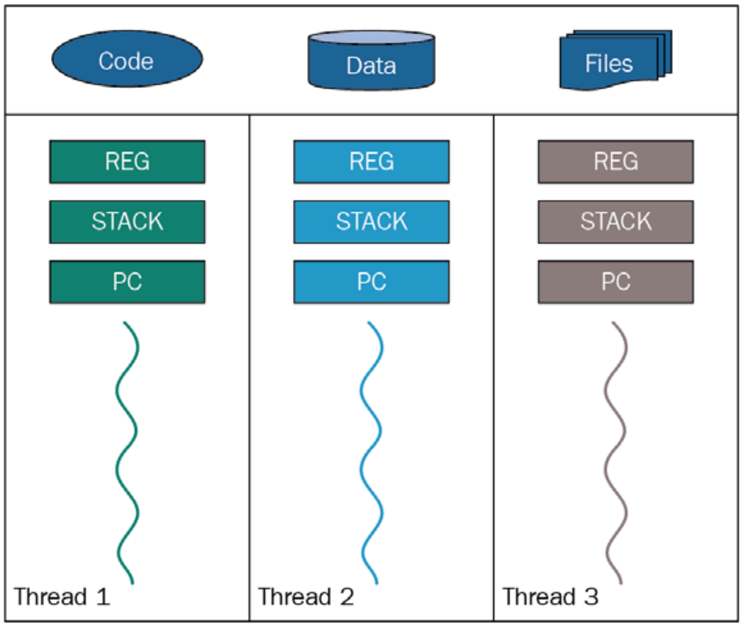


Figure 7.1 – Multiple threads in a process

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:

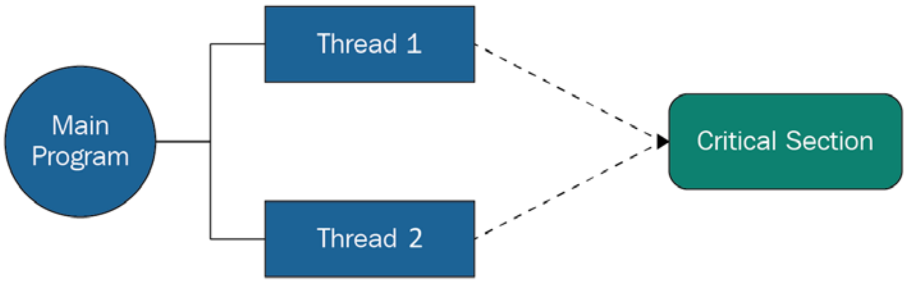


Figure 7.2 – Two threads accessing a critical section of a program

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 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_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.