From the Author of
Java Design Patterns

PYTHON BOOKCAMP

EXERCISES AND PROJECTS

Vaskaran Sarcar

Python Bookcamp

Exercises and Hands-on Projects

Vaskaran Sarcar

Python Bookcamp: Exercises and Hands-on Projects

ISBN-13: 979-8581409275

ASIN: B08FTD48NF

For information on translations, please e-mail **vaskaran@rediffmail.com.**

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at **https://github.com/Vaskaran/PythonBookcamp.** For more detailed information, please email **vaskaran@rediffmail.com**

I dedicate this book to all the unsung heroes and volunteers who are fighting at the front lines of Covid-19 battle to save humanity and this beautiful world.

Contents

About the Author

Acknowledgments

Preface

Chapter 1: Getting Ready for Program Execution

- **Overview**
- **Setting Up the Programming Environment**
  - Using Command Prompt
  - Using IDLE
  - Using PyCharm IDE
- **Executing Simple Programs**
  - Demonstration 1
  - Output
  - Demonstration 2
  - Output
  - Demonstration 3
  - Output

Chapter 2: Comments, Variables, and Operators

- **Using Comments**
  - Demonstration 1
  - Output
  - Analysis
- **Introduction to Variables**
  - Strings vs Numbers
  - int vs float
  - Demonstration 2
  - Output
  - Analysis
  - Naming Conventions
  - Assigning Multiple Variables in a Single Line
  - Assigning Same Value to Multiple Variables
- **Operators**
  - Arithmetic Operators
  - Assignment Operators
  - Comparison (or Relational) Operators

Logical Operators
Bitwise Operators
Precedence of Operators
Operators Associativity
Identity Operators
Membership Operators

Solution to Exercises

General Comments for Case Studies

Case Study I

Chapter 3: Common Data Types

Code Demonstrations with Strings

Code Demonstrations with Numbers
Quick talk on f-strings

Accepting User Inputs
Demonstration 1
Output

Solution to Exercises

CS 1.1 Implementation

Case Study II

Chapter 4: Decision Making

Using an if Statement
Demonstration 1
Output
Using the if-else Statements
Demonstration 2
Output
Demonstration 3
Output
Demonstration 4
Output

Using elif with an if-else Statement
Demonstration 5
Output
Demonstration 6
Output
Analysis

Tautology and Contradictions

Alternative Designs
- Demonstration 7
- Output
- Analysis
- Demonstration 8
- Output

Solution to Exercises

CS 2.1 Implementation

Case Study III

Chapter 5: Iteration using Loops

The Purpose of Iteration

The while Loop
- Demonstration 1
- Output
- Demonstration 2
- Output
- Analysis

The for Loop
- Demonstration 3
- Output
- Analysis
- Demonstration 4
- Output
- Analysis

Use of break Statement
- Demonstration 5
- Output
- Analysis

Use of continue Statement
- Demonstration 6
- Output
- Analysis

Nested Loop
- Demonstration 7
- Output
- Analysis

Solution to Exercises

CS 3.1 Implementation

Case Study IV

Chapter 6: Advanced Data Types

Working with Lists

Working with Tuples

Working with Dictionaries

Working with Sets

Solution to Exercises

CS 4.1 Implementation

Possible improvements

Case Study V

Chapter 7: Functions and Modules

Function Overview

Demonstration 1
Output
Analysis
Argument vs Parameter

Discussion on Function Arguments

Positional Argument
Keyword Arguments
Use of Default Values
Demonstration 2
Output
Demonstration 3
Output
Analysis

Discussion on Return Values

Demonstration 4
Output
Demonstration 5
Output

Lambda Function

Demonstration 6
Output
Demonstration 7
Output

Modules
Demonstration 8
Output
Using Alias
Demonstration 9
Output
Demonstration 10
Output
Quick Discussion on pip
Solution to Exercises
CS 5.1 Implementation
Possible Improvements
CS 5.2 Implementation
Possible Improvements
Case Study VI
Chapter 8: Exception Management
Types of Mistakes
General Philosophy
Common terms
Python Exceptions
Demonstration 1
Case Study with Valid Inputs
Case Study with an Invalid Input
Case Study with another Invalid Input
Key Points of the Exception-handling Mechanism
Demonstration 2
Case Study with Valid Inputs
Case Study with an Invalid Input
Case Study with another Invalid Input
Demonstration 3
Case Study with Valid Inputs
Case Study with Invalid Inputs
Use of pass Statement
Demonstration 4
Case Study with Valid Inputs
Case Study with an Invalid Input
Analysis

Arranging Multiple except Blocks
Case Study with an Invalid Input
Case Study with another Invalid Input
Custom Exception
Demonstration 5
Output
Debugging Code Using assert Statement
Demonstration 6
Output
Demonstration 7
Output
Analysis
Solution to Exercises
CS 6.1 Implementation
Case Study VII
Chapter 9: Programming with Files
Reading from a File
Quick Talk about File Paths
Demonstration 1
Output
Analysis
Writing to a File
Demonstration 2
Output
Analysis
Alternative Way to Read and Write
Demonstration 3
Output
Analysis
Working with Binary Files
Demonstration 4
Output
Renaming and Removing Files
Demonstration 5
Output
Using with Keyword
Demonstration 6
Output for Approach1

Output for Approach2
Output for Approach3

Handling FileNotFoundError
Demonstration 7
Output
Analysis

Solution to Exercises

CS 7.1 Implementation

Chapter 10: Object-Oriented Programming Concepts

Class and Objects

Encapsulation

Abstraction

Inheritance

Polymorphism

Q&A Session

Case Study VIII

Chapter 11: Class, Objects, and Inheritance

Constructing the Building Blocks for OOP
Demonstration 1
Output
Analysis

Constructor
Demonstration 2
Output
Analysis

Looking into Instance Creation Process
Demonstration 3
Output
Analysis

Changing an Attribute Value
Demonstration 4
Output

Working with Default Attributes
Demonstration 5

Output
Analysis

Importing Classes
Demonstration 6
Output
Analysis
Demonstration 7
Output
Demonstration 8
Demonstration 9
Output

Use of Identity Operator
Demonstration 10
Output

Inheritance
Demonstration 11
Output
Analysis
Updated Implementation
Output
Demonstration 12
Output

Private Variables and Methods
Demonstration 13
Output
Analysis

Static Methods and Class Methods
Understanding the Difference
Demonstration 14
Output
Analysis

Solution to Exercises

CS 8.1 Implementation
Possible improvements

Chapter 12: Test Your Code

The Code for Testing

Testing Functions
Demonstration 1
Output
Demonstration 2

Output
Analysis

Testing Class

Demonstration 3
Output

Solution to Exercises

Index

About the Author

Vaskaran Sarcar obtained his Master of Engineering in Software Engineering from Jadavpur University, Kolkata (India), and an MCA from Vidyasagar University, Midnapore (India). He was a National Gate Scholar (2007-2009) and has over 12 years of experience in Education and the IT industry. He devoted his early years (2005-2007) to teach at various engineering colleges, and later he joined HP India PPS R&D Hub Bangalore. He worked there until August 2019. At the time of his retirement from the IT industry, he was a Senior Software Engineer and Team Lead at HP. To follow his dream and passion, Vaskaran is now an independent full-time author. Other books by him include:

- Design Patterns in C# Second Edition (Apress,2020)
- Getting Started with Advanced C# (Apress,2020)
- Interactive Object-Oriented Programming in Java Second Edition (Apress,2019)
- Java Design Patterns Second Edition (Apress,2019)
- Design Patterns in C# (Apress,2018)
- Interactive C# (Apress,2017)
- Interactive Object-Oriented Programming in Java (Apress,2016)
- Java Design Patterns (Apress,2016)
- C# Basics: Test Your Skills (Createspace,2015)
- Operating System: Computer Science Interview Series (Createspace,2014)

Acknowledgments

At first, I thank the Almighty. I sincerely believe that with HIS blessings only, I could complete this book. I extend my deepest gratitude to all my family members, my publisher, the editorial board members, and everyone who directly or indirectly supported this book.

Preface

Welcome to your journey through Python Bookcamp: Exercises and Hand-on Projects. This is an introductory guide to the Python programming. Before you jump into the topics, I want to highlight a few points about the goal and the organization of the book.

- The primary goal of this book is to make you familiar with Python programming as quickly as possible. Once you learn the concepts in this book, you'll be ready to explore further. I have been involved in teaching since 2005. I have taken classes at both engineering and non-engineering colleges. This is the true motivation to introduce a book like this.
- The book helps you to learn the fundamental concepts effectively. What is an effective way? Ok, once you learn a topic, if you repeatedly practice and try to write code, you are on the right track. This is why this book has multiple exercises and hands-on projects. The book follows a steep learning curve, where you keep implementing these case studies using the concepts you learn in a previous chapter.
- Starting from the second chapter, each chapter in this book contains exercises too. These exercises are neither too easy nor too tough. These are within your optimal zone of difficulty. As per Goldilocks rule, you can motivate yourself and act better in these situations. The exercises and case studies help you test your understanding and raise your confidence level.
- Many of us are afraid of fat books because they do not promise that we can learn the subject in one day or 7 days. But you know that learning is a continuous process.

It is hard to achieve any real mastery in 24 hours or 7 days. So, the motto of the book is “To learn the core topics of Python, whatever effort I need to put, I am ok with that.” Still, simple arithmetic says that if you can complete one topic in one day, you can complete the book within 12 days (your learning speed depends only on your concentration level, focus, and dedication). But this arithmetic calculation is secondary! I have designed the book in such a way that upon completion of the book, you will know the core concepts in Python. Most importantly, you’ll know how to learn further.

- Python is a very popular computer language and widely used. Like other popular programming languages, it grows continuously to give us support with additional features and functionalities. When I started writing this book, Python3.8 was the latest version. So, everything in this book should run in Python 3.8 and upcoming versions. I choose this version for another reason. The official website tells us that Python 3.x (Commonly known as Python 3) is the future of the language, but Python 2. x (Also known as Python 2) is the legacy. You may see some old Python projects with Python2, but I recommend you to learn and use Python3.
- Once you are familiar with a good programming language, you find many similarities with other programming languages. As an obvious result, you can motivate yourself to learn a new programming language and this time it will be even easier to grasp.

How the book is organized?

The book has the following organization:

- The first chapter is a warm-up session for you. Here you’ll set up your programming environment and make yourself ready for further learning. In the Chapter 2, you’ll learn about comments, variables, and operators.
- I told you that this book includes multiple exercises and projects. An implementor must know what he/she is going to implement. So, at the beginning of each subsequent chapter, you will get an overview and description of the project that can be implemented. Once you finish reading the chapter, you’ll get the complete implementation of the projects.
- In Chapter 3, you’ll learn about common data types, such as strings and numbers in Python. In Chapter 4, you’ll learn about decision making in your program. Chapter 5 talks about iteration. Here you’ll learn about loops and the usage of break and continue statements. In Chapter 6, you’ll learn advanced data types, such as lists, tuples, sets, and dictionaries. Chapter 7 teaches you how to use functions to make your code more Pythonic. You’ll also learn about modules and their usage. Chapter 8 talks about exceptions and how to manage them. Chapter 9 teaches you file handling mechanisms.
- Chapter 10 and Chapter 11 of this book briefly cover the object-oriented programming basics and show you the usage of classes, objects, and inheritances. Here you also learn about static methods, class methods, and private variables, etc. At the end of the book, there is a chapter (Chapter 12) to show how to write useful tests to verify your code.
- You can download all the source codes of the book from GitHub (I already gave you the online link). I have a plan to maintain the “Errata” and if required, I can also make some announcements there. So, I suggest that you visit those pages to receive any important corrections or updates.

Prerequisite Knowledge

The target readers for this book are those who are new to Python programming. The book will be super easy for the readers with the least coding experience in any other high-level computer language. I assume that you can download a software installer following the online instructions and you already installed Python on your computer. So, I do not spend time on this topic. It is because you can find them easily both online and offline.

Who is this book for?

In short, you can pick the book if the answer is “yes” to the following questions:

- Have you never programmed before, but eager to learn Python?
- Do you have experience with one or more high-level programming languages, but want to learn Python this time?
- Do you want to explore the Python basics step-by-step, but as quickly as possible?
- Do you want to be familiar with object-oriented concepts like polymorphism, inheritance, abstraction, and encapsulation?
- Do you know how to install software on a machine and then set up the coding environment?
- Do you like to review your knowledge before you use Python in advanced fields such as data science, machine learning?

Probably you shouldn’t read this book if the answer is yes to any of the following questions:

- Are you confident about the fundamentals of Python?

- Are you looking for advanced concepts in Python, excluding the topics mentioned previously?
- Do you dislike a book that has an emphasis on exercises?
- “I dislike Windows OS, and PyCharm. I want to learn and use Python without them only.”-Is this statement true for you?

Guidelines for Using This Book

Here are some suggestions so you can use the book more effectively:

- I suggest you reading these chapters sequentially. The reason is that some fundamental techniques/concepts may be discussed in a previous chapter, and I do not repeat the same in a subsequent chapter.
- I also suggest that you complete the exercises in a chapter before you enter a new chapter. This process can give you confidence, which can give you a better pay-off soon.
- Indentation is an important part of Python programming. We consider anything indented as a block of code in Python. Based on your device’s screen size, you may not see the correct indentation in some programs. I suggest that you refer to the actual code in those cases. You can download the actual code from the online link https://github.com/Vaskaran/PythonBookcamp. I mentioned this link at the beginning of this book.
- It is worth mentioning that I could use IDLE, which is Python’s Integrated Development and Learning Environment. It provides some basic functionalities such as syntax highlighting, different uses of colors for better readability. But I have used Python Command Shell (for

the initial chapters where lines of code are very short) and PyCharm Community Edition 2020.1.1 in the Windows 10 environment. PyCharm has many interesting features and when you work with large projects, those features are very useful. The Community edition of PyCharm is also free of cost. If you do not use the Windows operating system, you can also use Visual Studio Code, which is also a source-code editor developed by Microsoft to support Windows, Linux, or Mac operating systems. This multi-platform IDE is also free.

- In some cases, I have shortened the long URLs for a better presentation. For example, you'll see http://bit.ly/python-exceptions instead of https://docs.python.org/3/library/exceptions.html#:~:text=In%20
- In this book, I have written the programs are written and explained the concepts in a way that these different programming environments should not cause any major problem to your learning process.
- I have used Python 3.8 for this book, which was the latest version available at this time of writing. You can surely predict that version updates will come continuously, but I strongly believe that these version details should not matter much to you because I have used the fundamental constructs of Python. So, these codes should execute smoothly in the upcoming versions of Python/PyCharm as well. I also believe that the results should not vary in other environments, but you know the nature of a software-it is naughty. So, if you like to see the same output, it is better to mimic the same environment.
- Remember that you have just started on this journey. As you learn about these concepts, try to write your code. It helps you write better programs.

Conventions in This Book

Here I mention only two points: In many places, to avoid less typing, I have used the word “his” only. Please treat it as “his” or “her” -which applies to you.

Second, all the outputs and codes of the book follow the same font and structure. To draw your attention, in some places, I have made them bold. For example, consider the following code fragment (I’ve taken it from Chapter 8) and the lines in bold.

```
# Previous codes are skipped

except ZeroDivisionError as e:
    print("Invalid input! Your divisor becomes zero!")
    print(f"Error details:{e}")
except ValueError as e:
    print("Invalid input! Provide a correct input next time!")
    print(f"Error details:{e}")
else:
    print(f"Result of the division is : {result}")
print("The program completes successfully.")
```

Finally, I hope that this book can provide help to you and you will value the effort.

Chapter 1: Getting Ready for Program Execution

This chapter briefly talks about the Python language and its importance. Here you'll learn how to set up your programming environment before you execute the programs.

Overview

Python is a computer programming language that is rapidly growing nowadays. The primary reasons for its popularity are simplicity and readability. Guido van Rossum created this in the late 1980s.

You can use Python for various purposes. For example, you may notice its usage in game programming, business applications, tools development, etc. Using Python programs, you can automate boring and lengthy tasks to save your time and energy. In recent years, you notice it in emerging fields like data science and machine learning too. Last but not least is that the Python community is very strong and supportive and they help you grow faster.

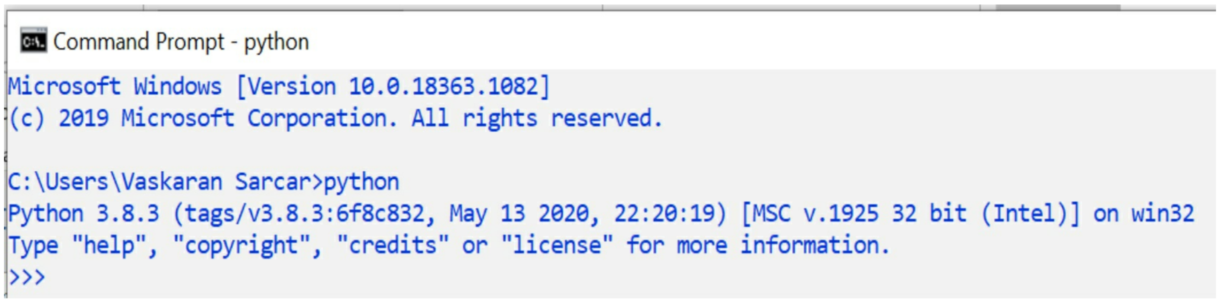
Setting Up the Programming Environment

Python is a high-level language. So, you can avoid direct interaction with registers, memory addresses, call stacks, etc. Instead, you write your program in plain English. So, how the computer will understand the instructions? Well, you may need to be familiar with two terms-compilers and interpreters. I'll cover them later. Now you can keep in mind the simple distinction between these two: An interpreter translates one instruction (or statement) at a time into machine code. But a compiler takes the entire program and then translates it into machine language in one shot. For now, you remember that you need a Python interpreter to interpret these programs. So, you need to get the interpreter and install it before you write your programs. You need to pick the correct interpreter based on your operating system.

A popular programming language grows continuously to give us support for more features and functionalities. When I started writing the book, Python3.8 was the latest version. So, everything in this book should run in Python 3.8 and upcoming versions. I choose this version for another reason. The official website tells us that Python3.x (known as Python 3) is the future of the language, but Python2.x (known as Python 2) is the legacy. So, you may see some old Python projects with Python2, but I recommend you to learn and use Python3.

Using Command Prompt

You can use a text editor to write a Python program. For example, you can use a notepad. But before you do this, let us begin with a simple command prompt to check whether Python is installed in your system. So, open the command prompt and type python (See **Figure 1-1**):



```
Microsoft Windows [Version 10.0.18363.1082]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Vaskaran Sarcar>python
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 1-1: Checking whether Python is installed in the system.

I told you earlier that I assume Python is already installed on your computer. If Python is installed on your computer, you notice the detailed information as shown in Figure 1-1. Now you can use some simple commands for further verification purposes. Each time you enter a command and press the Enter (or Return) key, the statement will be tested. Here are some examples:

```
C:\Users\Vaskaran Sarcar>python
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> 2+3
5
>>> 5>3
True
>>> 2<1
False
>>> print("Hi")
Hi
>>>
```

You can type **exit()** (Or, or **Ctrl-Z plus Return**) to quit from this shell.

Using IDLE

You can also launch IDLE to get a python shell where you can execute python commands. For example, to get IDLE in Windows10, you can type IDLE in the search box as shown in Figure 1-2:

What is a shell? In simple terms, it is an environment that is used to run other programs. We can use shells for both the command-line interface and graphical user interface (GUI). But normally we use it to refer to the command-line interface of the operating system (OS). Developers often call the terms-‘shell’ and ‘terminals’ in the same context interchangeably.

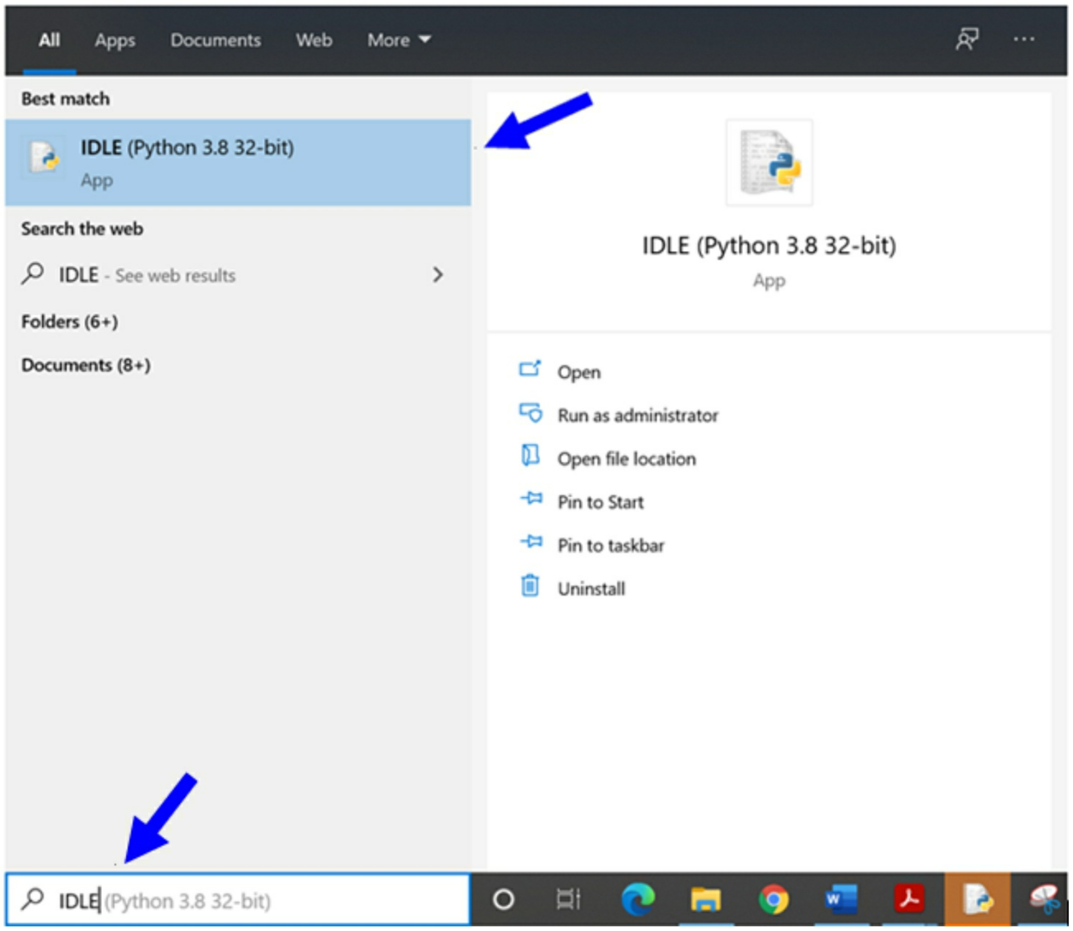
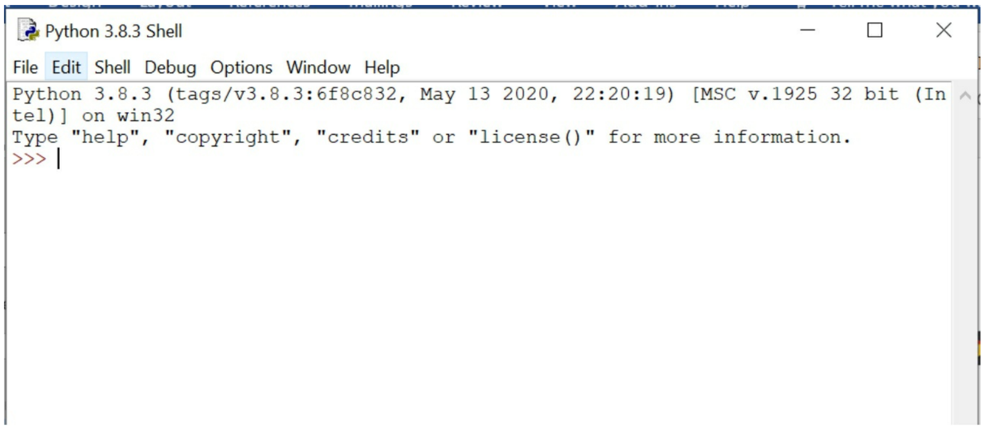


Figure 1-2: Searching IDLE

Once you can see the app, click on it to launch. Now you can see the shell.

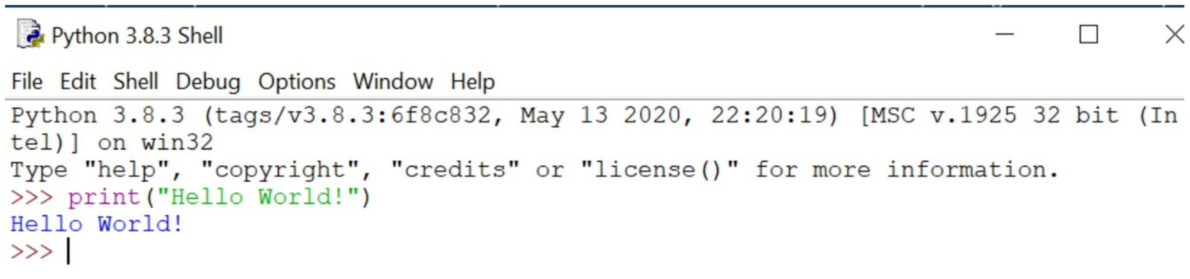


```
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> |
```

Figure 1-3: Python 3.8.3 is launched.

This shell waits to get a particular command from the user, executes the command, and then displays the result. Once this cycle is completed, it waits for the next command to receive from the user. Let us try this.

Type print("Hello World!") after >>> in the shell, and press the **Enter** key. You can immediately see the output Hello world in the next line as shown in the following figure.



```
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello World!")
Hello World!
>>> |
```

Figure 1-4: Printing 'Hello World' using IDLE.

Now type 1+2 in the shell and press enter to get the following output:

```
>>> 1+2
3
```

Now you may try a few more lines of code, and verify the output like the following:

```
>>> 10>7
True
>>> 2+7<5
False
```

Hopefully, you get an idea- how to write some simple programs in a Python shell. To execute some basic commands, or to check whether you are ready for python programming, this process is fine. But the problem is: once you exit from the shell, we lose all these commands. This is why we can use a text file to write a Python program and save the file with the **.py** extension. In short, the file with the **.py** extension is called a python script.

Using PyCharm IDE

I have shown you the use of simple command prompts to make you aware of the alternative ways to run your Python programs. But ask any professional about how they write programs. You’ll come to know that they use specialized text editors or IDE’s (IDE stands for Integrated Development Environment) to write the code. It is because the use of command prompts is not suitable for big programs. IDE’s help you highlight the syntax error and they can provide you automatic suggestions to write error-free code. They also support auto-completion for certain functions and phrases, which are extremely helpful. There are many IDE’s with cross-platform support too. But in the end, a few of them became popular, and widely used for a particular computer language. For example, Java developers often use *Eclipse* when they write programs; similarly, *Visual Studio* is very common for .NET developers. The same concept applies here. For Python programming, there are many IDE’s, and **PyCharm** is very common. I have used this IDE to develop my programs (aka python scripts) for this book.

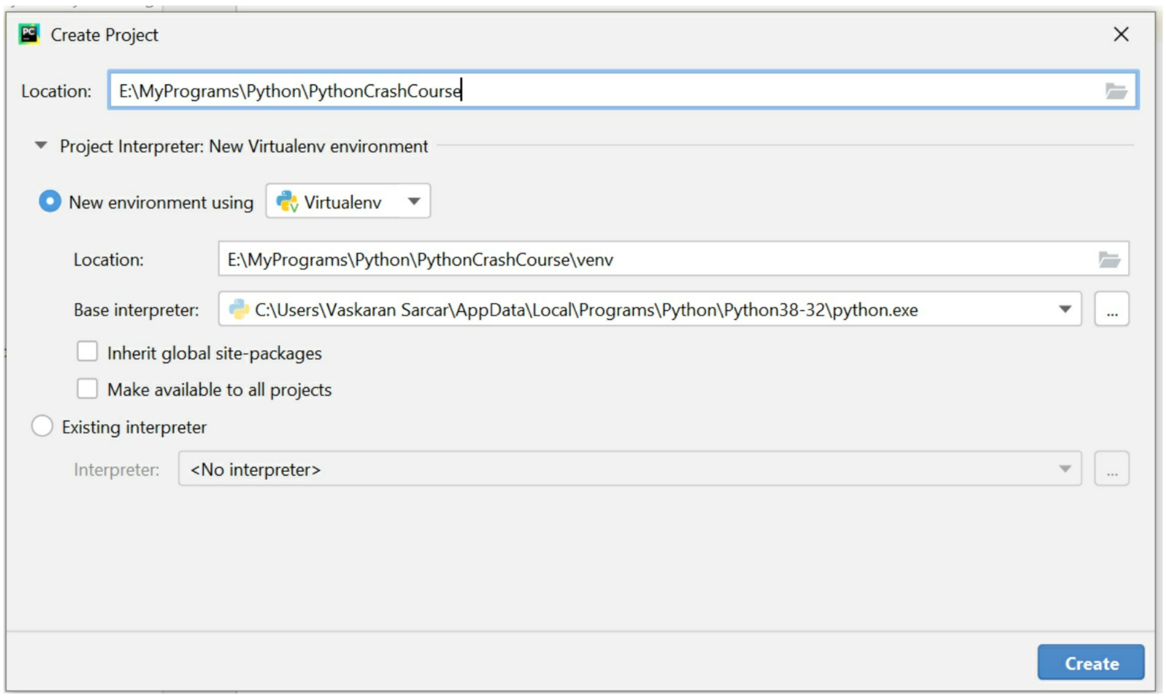
Spyder is another open-source and cross-platform IDE, which we often in Python programming. For my machine learning projects, I use Jupyter Notebook in which you can test Python scripts too. If you install Anaconda distribution on your computer, you can find both Spyder IDE, Jupyter Notebook, and many other things. But I believe that to learn and test simple Python scripts, PyCharm is a pleasant choice for you. It helps you to organize your files, identify syntax errors. You can also set breakpoints to pause at specified lines in your program. You find code refactoring very easy when you use PyCharm.

For the following program, I took some screenshots from the PyCharm IDE. These can help you visualize the execution environment. But next time onwards, I’ll show you the programs and corresponding output only. As said before, to execute the python scripts, PyCharm is NOT mandatory. You can run these programs in various ways (for example using IDLE, Spyder IDE, Jupyter Notebook, etc). So, it makes little sense to take screenshots from PyCharm for each of these programs.

Let us follow these steps:

Step-1: Open PyCharm

Step-2: Click on File > New Project


Create Project

Location: E:\MyPrograms\Python\PythonCrashCourse

Project Interpreter: New Virtualenv environment

New environment using Virtualenv

Location: E:\MyPrograms\Python\PythonCrashCourse\venv

Base interpreter: C:\Users\Vaskaran Sarcar\AppData\Local\Programs\Python\Python38-32\python.exe

Inherit global site-packages

Make available to all projects

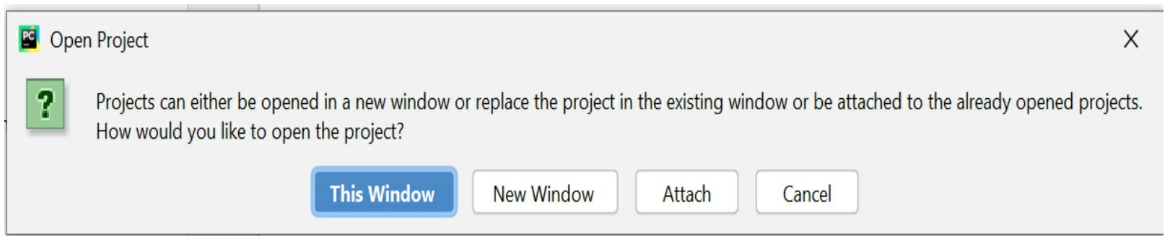
Existing interpreter

Interpreter: <No interpreter>

Figure 1-5: Creating a project in PyCharm.

If you have multiple Python versions installed on your computer, you can see them in the drop-down list in the previous screenshot (Figure 1-5) and you can pick your preferred python version from here. I am using the latest version (3.8) which is available at the time of this writing.

Since I have already executed some python scripts using PyCharm, I get this option. But for the first time users, you do not see this window.


Open Project

Projects can either be opened in a new window or replace the project in the existing window or be attached to the already opened projects. How would you like to open the project?

Figure 1-6: Project open options in PyCharm.

Step-2.1: I opt for a new Window.

Now I get the following screen. See Figure 1-7.

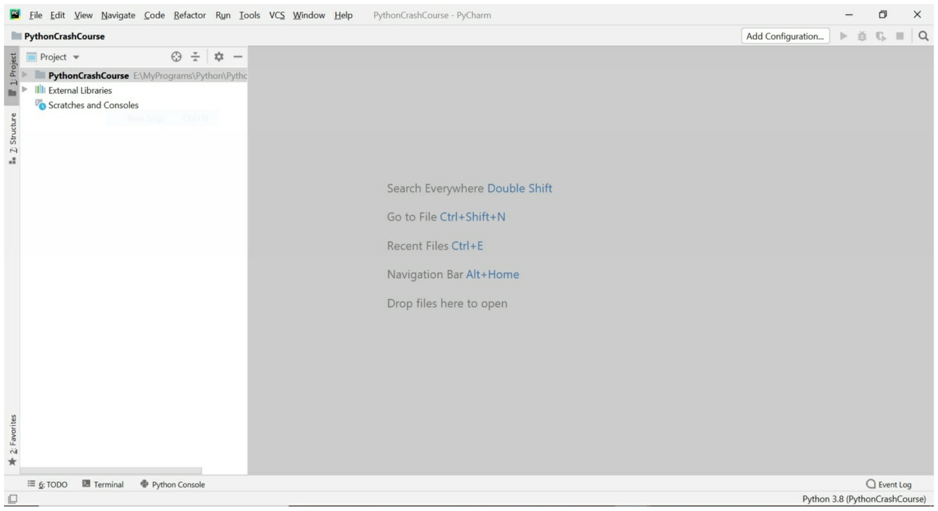


Figure 1-7: The project is opened in a new window.

Step-3: Right-click on the project folder name (PythonCrashCourse). Select “New” and then choose “Python File” (See Figure 1-8)

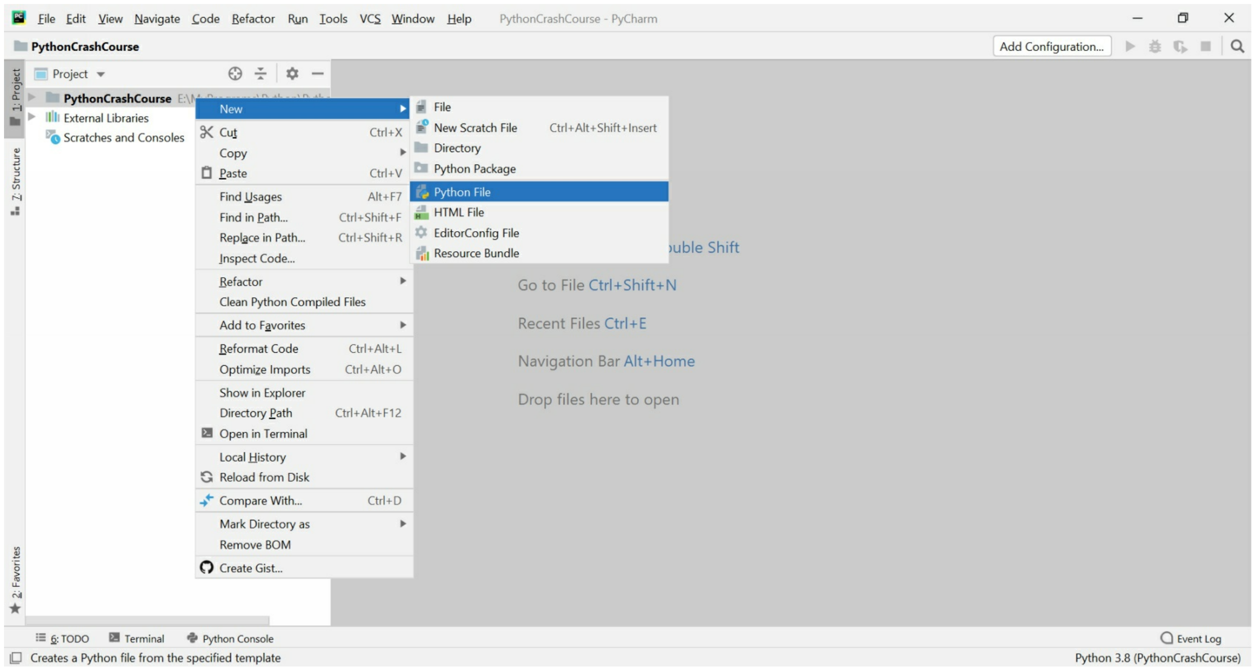


Figure 1-8: Adding a new Python file.

Step-3.1: Name it as hello_world. See Figure 1-9.

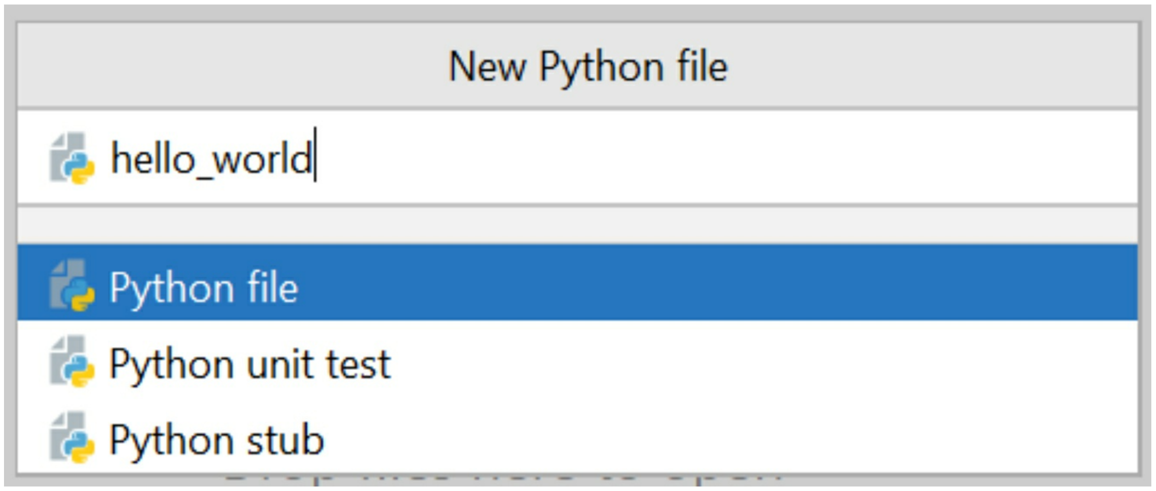


Figure 1-9:Name the file as hello_world.py

Step-3.2: Press Enter. Notice that a new python file is created for you. See Figure 1-10.

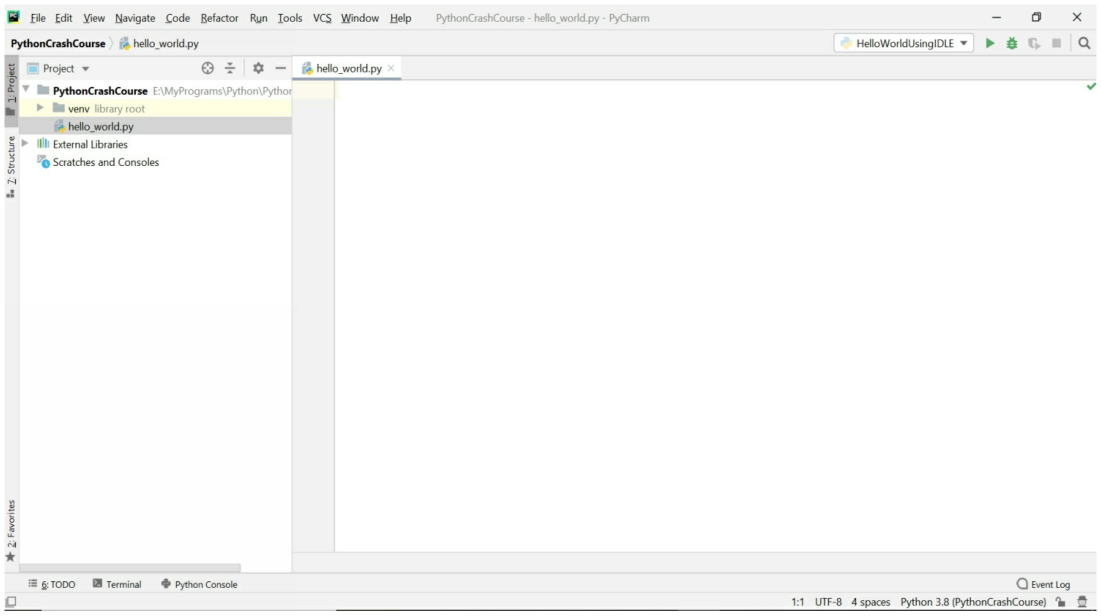

File Edit View Navigate Code Refactor Run Tools VCS Window Help PythonCrashCourse - hello_world.py - PyCharm
PythonCrashCourse hello_world.py
HelloWorldUsingIDLE
Project
hello_world.py
PythonCrashCourse E:\MyPrograms\Python\Python
venv library root
hello_world.py
External Libraries
Scratches and Consoles
6: TODO
Terminal
Python Console
Event Log
1:1 UTF-8 4 spaces Python 3.8 (PythonCrashCourse)

Figure 1-10: The hello_world.py is ready. You can write the code here.

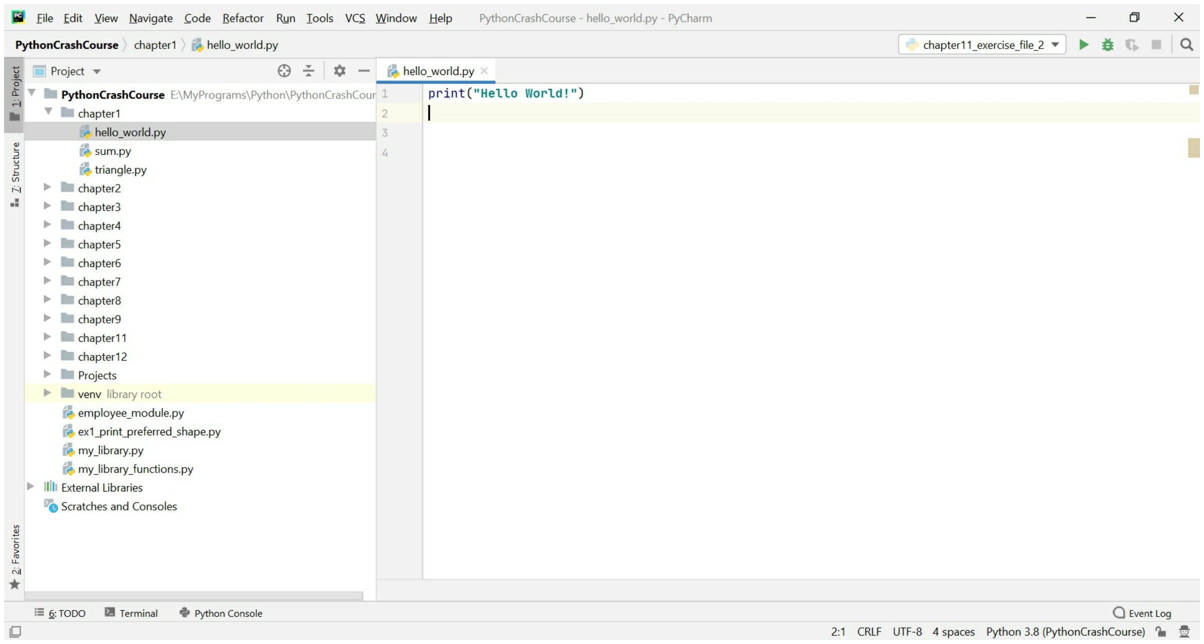
Now you are ready to write your first program in PyCharm.

Executing Simple Programs

Now I show you some simple Python programs using PyCharm IDE.

Demonstration 1

Let us write a simple python program. Type the following line: print("Hello World!") in a python file (Refer to the following screen). Now move the cursor to the next line by pressing the “Enter” key as shown in the following figure. See Figure 1-11. (I have organized the code of this book chapter-wise. So, in this screenshot, you see other files and directories too.)

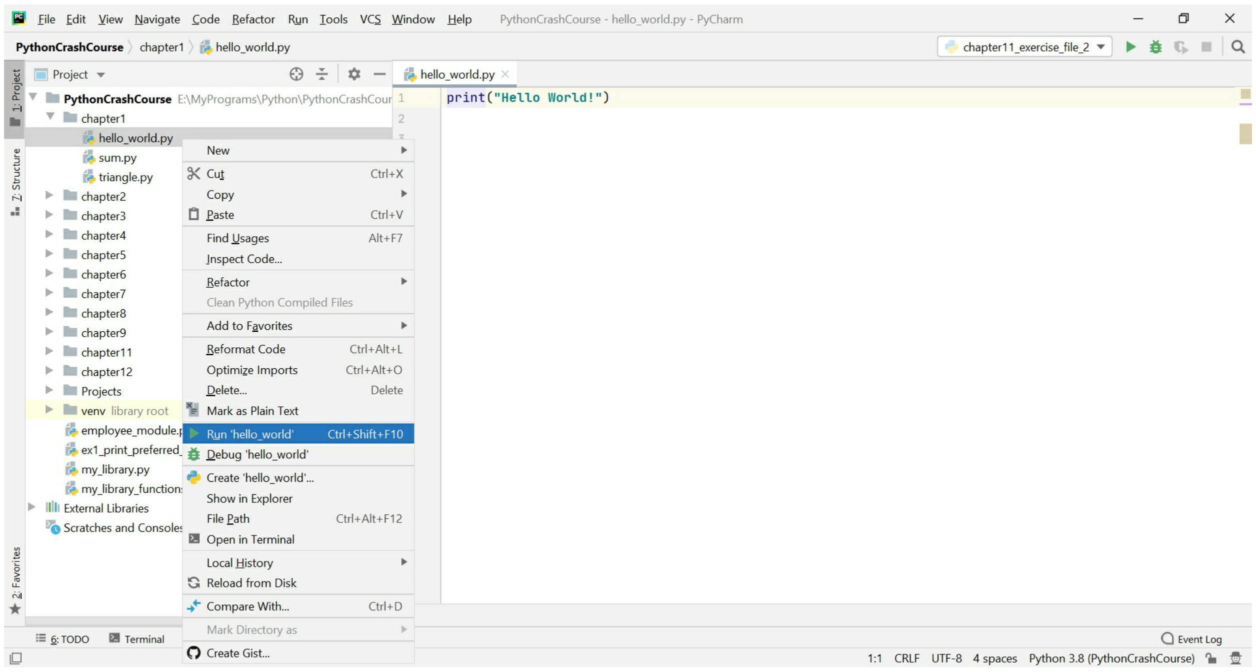


```
print("Hello World!")
```

Figure 1-11: The hello_world.py file contains a line of code.

Save the file (**File >Save All (Ctrl+S)**).

Now you can run the program. **Right-click** on the file name and press the option **Run ‘hello_world’** as shown in the following figure. See Figure 1-12.



```
print("Hello World!")
```

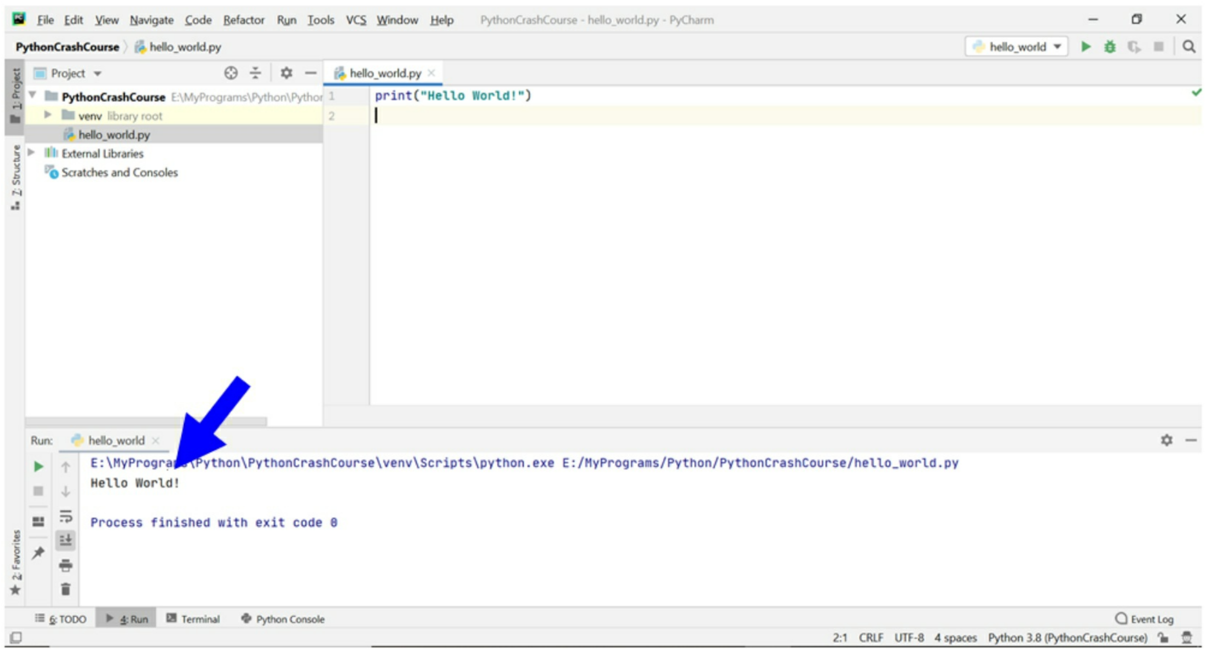
Figure 1-12: Run hello_world.py in PyCharm

Output

Congratulation! You have successfully executed your first python program in PyCharm IDE. You can see the following output:

Hello World!

Now I take a snapshot from PyCharm to depict it better. See Figure 1-13.

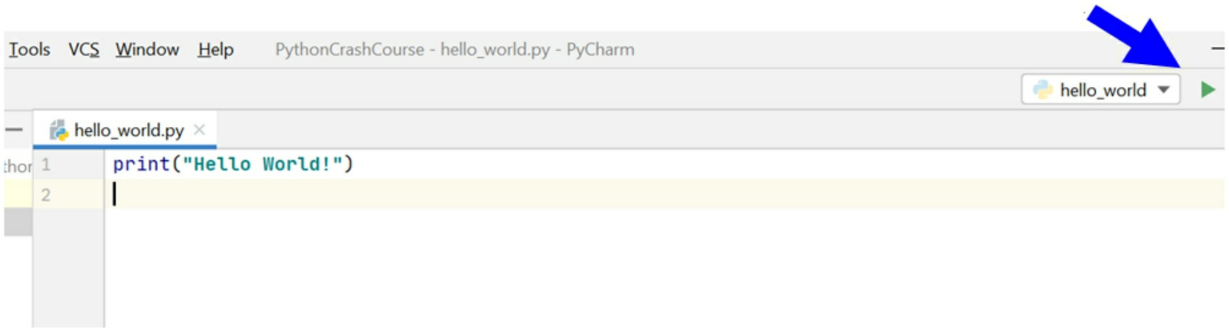


```
print("Hello World!")
```

```
E:\MyPrograms\Python\PythonCrashCourse\venv\Scripts\python.exe E:/MyPrograms/Python/PythonCrashCourse/hello_world.py
Hello World!

Process finished with exit code 0
```

Figure 1-13: The successful execution of the program generates the output.

If you want to run the program again, you can use the following button as shown in the following figure. Notice that ‘ hello_world’ is selected by default before you press the run button. See Figure 1-14.



```
print("Hello World!")
```

Figure 1-14: Alternative option to run hello_world.py

POINTS TO REMEMBER

- Every time you write a new program, you can follow the same approach. Next time onwards, I’ll show you the programs and output only. It is because you can run the python scripts in various ways(for example, you can use a Python shell, you may opt for IDLE, etc). So, it makes little sense to take screenshots from PyCharm in each case.

- Following the PEP8 guideline (You can refer to the link https://www.python.org/dev/peps/pep-0008/), I have named my file name hello_world.py. This guideline suggests that you choose all lowercase names for your modules. But you can use underscore to improve readability. I'll discuss modules later in Chapter 7. For now, you know that a module is a python file with a .py extension.

Let us have some fun with python scripting and continue to write another simple program. In Demonstration2, I am using asterisks (*) to make a shape which makes a triangle shape.

Demonstration 2

Create a new python file. Let us call it **triangle.py** and type the following lines into it.

```
print("*")
print("**")
print("***")
print("****")
print("*****")
```

Now save the file and run the program.

Output

Here is the output.

```
*
```

```
**
***
****
*****
******
```

Let us go through another simple program in which I’m calculating the sum of two numbers.

Demonstration 3

Create a new python file. Let us call it **sum.py** and type the following lines into it.

```
print("The sum of 12 and 5.7 are as follows:")
print(12+5.7)
```

Output

Here is the output.

```
The sum of 12 and 5.7 are as follows:
17.7
```

I know that this chapter was a bit slow, but it is important. These brief discussions will help you understand the theory and do the code exercises in the upcoming chapters.

Chapter 2: Comments, Variables, and Operators

In this chapter, you see some building blocks to develop your Python programs. First, you'll learn how to write comments and use variables in a program. Later you'll learn how to use operators in your program.

Using Comments

It is a standard practice to use comments in your program. These comments can help others to understand your code better. Let us consider a real-life scenario. In a software organization, a group of people creates software for its customers. It is possible that after some years, none of them are available. Either these members move into a different team, or they leave the organization. In such a case, someone needs to maintain the software and continue fixing the bugs for its customers. But it is very difficult to understand the logic if there is no hint or explanation about the program logic. Comments are useful in such scenarios. These help us understand the code better.

In Python, you see the following options for comments:

Case-1: Single line comments using # tags. Here is an example:

```
# This is a single-line comment
```

Case-2: Multi-line docstrings as multi-line comments using three double quotations (or single quotations). You can see one-line docstrings as well. Here I show you the use of docstrings that I use inside a function in Chapter 7. For now, you do not need to understand the code. You only see the texts that begin with triple-double quotations (" " ") and ends with triple-double quotations (" " ")

```
def print_details(name,age):
    """
    This function takes two parameters.
    You can supply the name and age of the user
```

```
in this function.
"""
print(f"Hello {name}!How are you?")
print(f"You are now {age}.")
```

POINTS TO REMEMBER

- Comments are simple notes or some texts. You use them for human readers, but not for the Python interpreter. Python interpreter ignores the text inside a comment block.
- In the software industry, many technical reviewers review your code. The comments help them understand the program logic.
- A developer can forget the logic after some months. These comments can help him recollect his logic.
- Experts prefer to use the many single-line comments using # tags.
- In Python programming, functions, modules, and classes should have docstrings. These will make sense when you see them in Chapter 7 or Chapter 11. You’ll know that a docstring becomes the _doc_ attribute of an object. I leave the discussion at this point.
- In Chapter 7 and Chapter 11, you’ll see that I use docstrings to describe a function and class behavior.
- In this chapter, you learn the use of simple comments that you may see in other’s code.

Demonstration 1

When the Python interpreter sees a comment, it ignores that. To understand this, follow the next program. I repeat that experts suggest that we use single-line comments with # tags. Use of the docstrings is common when you use a function, module, or class. You'll be familiar with them later.

```python
# Testing whether 2 is greater than 1
print(2>1)
'''
I'm using these as multi-line comments.
I use these lines for your reference only.

These are common in class, functions, or modules.
'''

"""
I'm trying to use multi-line comments using three double-quotes.
I use these lines for your reference only.
These are common in class, functions, or modules.
"""

# Now I'm showing multiple single-line comments
# Multiplying 2 with 3
# And printing the result
print(2*3)
```

Output

Here is the output.

```
True
6
```

Analysis

This program uses different comments, and it is easy to understand. You can see that you have received output for the lines print(2>1) and print(2*3) only. The remaining portions(comments) are ignored by the interpreter. Since 2 is bigger than 1, so the output came as True , and when you multiply 2 with 3, the result is 6 .

Introduction to Variables

In Mathematical Algebra, you often write something like x=10. Then you say x is a variable to represent the number 10 . Python works in the same way, except here you can represent both numeric and non-numeric values using variables.

In the following sections, for your easy understanding, I use some string and number variables. You'll learn more about them in the next chapter (Chapter 3). To continue this discussion, I quickly cover two important points:

- What do I mean by the words- strings and numbers?
- What is the difference between int and float ?

Strings vs Numbers

In simple words, strings are texts. They are very common in Python programming. You have already seen the usage of strings in earlier demonstrations. For example, to print Hello World! I wrote the following:

```
print("Hello World!")
```

I can also use a string variable like the following:

```
my_text = "Hello World!"
```

and then print the output using the following line of code:

```
print(my_text)
```

Notice that in both cases, I enclose Hello World! inside the double quotation marks. This is the general format to declare a string variable. You can use single quotations too. Both are fine. Sometimes you may need to use both intelligently. For example, the following code works fine:

```
>>> print("This is Sam's book.")
This is Sam's book.
```

But if you use single quotations like the following, you'll get errors. Here is a sample:

```
>>> print('This is Sam's book.')
  File "<stdin>", line 1
    print('This is Sam's book.')
                       ^
SyntaxError: invalid syntax
```

You can use an escape character to avoid the error. Here is a sample:

```
>>> print('This is Sam\'s book.')
This is Sam's book.
```

You'll learn more about escape characters in Chapter 3.

POINT TO REMEMBER

You can use single quotations or double quotations to print the output messages. Both are fine. Python does not distinguish between single and double quotation marks. You need to remember the simple fact: you enclose a string literal in matching single quotation or double quotation marks.

In most cases, you see me using double quotation marks. It is because I learned Java, C# earlier, and naturally double quotations come in my coding. I have seen people using single quotations to print simple messages and double quotations for formatted strings. But note that this is only a preference, but not a convention. The following link: http://bit.ly/3i8zWAw says: "In plain English: Both types of literals can be enclosed in matching single quotes (') or double quotes ("). They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as triple-quoted strings)." Refer to the official documentation http://bit.ly/python-official-documentation whenever you have similar doubts.

Numbers are also heavily used in Python programming. Like strings, to print a number without a variable, you can just type the number inside the print() function as follows. I keep the inline comments to show what these lines can print for you.

```
print(1)#Prints 1
print(5.7)# Prints 5.7
print(-6.789)# Prints -6.789
```

You can also use number variables like the following:

```
# Using variables
my_int=125
print(my_int)# Prints 125
my_float=25.763
print(my_float)# Prints 25.763
```

I want you to notice that I've NOT enclosed these numbers inside double quotations. This is the general format to declare a number variable. This slight difference in a declaration can make a big difference in output. For example, consider the following code segment, and notice the inline comments to know the output:

```
# Difference between the numbers and the strings
print("1"+ "2") #Prints 12
print(1+2) #Prints 3
```

Now I want you to note another key distinction. This time both are numbers, but they are categorized as int and float . Let us have a look.

int vs float

In Python, ***integers*** are whole numbers; they do not have fractional parts. An integer can be positive, negative, or 0. For example, 23,-32,67,-2,0,25 are examples of integers. In Python, we refer to an integer as an int .

In contrast, 35.75, 45.2, etc. are not integers because they have fractional parts. These are ***floating-point numbers***. In Python, you simply call them the float datatype .

Now you are ready to proceed further. Before I discuss more on Python variables, let me show you a program. This program can make more sense to you when you go through the analysis followed by the output of the program.

Demonstration 2

Now I change the Demonstration1 in the previous chapter (Chapter 1). Here I add a few more lines of code into the program. I create a new python file and name it **hello_world_modified.py.** Then I type the following lines into it.

```
print("*** This program shows the use of a variable. ***")
# mytext is holding the value "Hello World!"
my_text = "Hello World!"
print(my_text)
# my_text is holding a new value
my_text = "Dear Reader, how are you?"
print(my_text)
```

Output

Here is the output.

```
*** This program shows the use of a variable. ***
Hello World!
Dear Reader, how are you?
```

Analysis

This output shows you an alternative way to print “Hello World!” . So, what was the key modification? Instead of writing the following line:

```
print("Hello World!")
```

Now I have written the following lines:

```
my_text = "Hello World!"
print(my_text)
```

You can see that in both cases, I have received the same output. But why should I write more? One possible answer can be found when you notice the last part of the program which is as follows:

```
my_text = "Dear Reader, how are you?"
print(my_text)
```

You can see that you can store different data inside my_text and manipulate them later. Most importantly, you can change the value later. It is the key use of a variable. You got it right; my_text is an example of a variable that you have seen in this demonstration. In this example, you can notice that the initial value (Hello World !) of my_text is changed later in this program.

To explain the program better, I can use comments in this program. You learned about comments already, and here you see another valid use case of them.

Now I summarize the key points:

- Using an assignment operator (=) you can assign a value to a variable.
- You can assign a variable and reassign it again. You do this when it makes sense to you.

Note that the type of variable change if you reassign an expression with a different type. Let us test this in a Python shell:

```
>>> number1=2
>>> number2=3
>>> print(number1+number2)
5
>>> number1=2.0
>>> print(number1+number2)
5.0
```

You can see that when number1 was reassigned to the value 2.0, the resultant sum of number1+number2 becomes **5.0** instead of **5** .

Alternatively, you can test the following block of code. Notice that if you assign 2 to the variable number1 , the Python interpreter can determine that it is an int . But when you assign 2.0 to the variable number1 , it can recognize it as a float data type.

```
>>> number1=2
>>> type(number1)
<class 'int'>
```

```
>>> number1=2.0
>>> type(number1)
<class 'float'>
```

Naming Conventions

You can remember the following points when you use variables in your program.

You use an assignment operator (for example, =) to assign a value to a variable. You'll see different assignment operators shortly.

In real-world applications, you use a meaningful and descriptive name for your variables. Try to avoid one-character variable names except inside some loops or functions. You will know about loops and functions and use them in later chapters. For now, you can note this point. In simple demonstrations to type less, sometimes I ignore this convention. So, in some code fragments, you may see something like x=5 . But in a real-world application, choose a better name, which is meaningful enough. For example, to assign an identification number (say, 5) to an employee in an organization, you can write something like employee_id=5

Your variable name should not clash with Python keywords. A keyword is a reserved word, you cannot use them as ordinary identifiers. Python 3.9 has released just now. You can refer to the online link https://docs.python.org/3/reference/lexical_analysis.html#identifiers to know about the latest keywords. For your immediate reference, I include them here:

```
False     await     else      import    pass
None      break     except    in        raise
True      class     finally   is        return
and       continue  for       lambda    try
as        def       from      nonlocal  while
assert    del       global    not       with
async     elif      if        or        yield
```

So, if you write for=25, you'll get an error. Here is a sample for you:

```
>>> for=25
  File "<stdin>", line 1
```

```
    for=25
       ^
SyntaxError: invalid syntax
>>> for_variable=25
>>> for_variable
25
```

Like a different computer language, Python has many in-built functions. Your variable name should not clash with those names. For example, in Python, you can use the abs() function to calculate the absolute value. Here are some examples.

```
>>> abs(23.7)
23.7
>>> abs(-5.7)
5.7
>>> abs(3-7.5)
4.5
```

So, ideally, you should not use something like abs=45.2. In this context, there is another interesting point to note. Let us say, by mistake, you use abs=45.2 earlier in your code. Now if you call abs(-10) , a Python shell can show you an error. Here I show you such a sample.

```
>>> abs=45.2
>>> print(abs)
45.2
>>> abs(-10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object is not callable
```

A similar error may appear if you write something like abs=45 and then you try to call abs(-10) . Here is a code fragment:

```
>>> abs=45
>>> print(abs)
45
>>> abs(-10)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Both the errors are similar. The only difference is that in the first case, it complains about the float , but in the next case, it says about an int . It is obvious because you know that 45.2 is a floating-point number, but 45 is an integer. So, you understand that this kind of error tries to say that you have used abs before this call. A simple remedy to this is to exit from the shell (or remove the erroneous variable naming practice) and try to use the in-built function again. Here is a code fragment from the Python shell for your reference.

```
C:\Users\Vaskaran Sarcar>python
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32
bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> abs=7
>>> print(abs)
7
>>> abs(-5.7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
>>> exit()

C:\Users\Vaskaran Sarcar>python
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32
bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> abs(-5.7)
5.7
```

You can see that when I see the following error:

(TypeError: 'int' object is not callable)

I get a clue that before this line somewhere I used abs as an Integer. So, I exit from the shell and re-enter the shell again. This activity deletes old assignments. This time I do not perform the illegal variable naming practice too. So, the in-built function abs() again work fine for me.

You can use any of the following naming conventions and you'll not receive any error for any of these. For example, to name a student, you can use your variable name as:

studentName (Lower camel case)
StudentName (Upper camel case, also known as Pascal case)
student_name (Snake style, words are separated by an underscore)
Student_name (Also a snake style, but first word starts with a capital letter)

Variable names can be alphanumerical, but the first letter must start with a letter. For example, student_1 , student1 both are fine, but 1student or 1_student will raise an error. Here is an example for your reference.

```
>>> student_1="John"
>>> print(student_1)
John
>>> student2="Sam"
>>> print(student2)
Sam
>>> 3student="Kate"
  File "<stdin>", line 1
    3student="Kate"
     ^
SyntaxError: invalid syntax
```

You have seen that you can use an underscore in your variable name, but other special characters can raise an error in your program. For example, if you use @ inside your variable name, you'll receive an error. Here is an example.

```
>>> student@name="Jack"
  File "<stdin>", line 1
SyntaxError: cannot assign to operator
```

The variable name should not start or end with an underscore. Though you do not see any error for that, experts strongly discourage this

practice. For example, you should not use something like:

```
_student = ”John”
or
student_= ”John”
```

If you ask me: Among these accepted naming conventions, which one is my favorite? I’ll answer that I like to use the snake style, which is something like the following:

```
student_name=” Jack”
```

I recommend the same practice for you.

Look at the following line again: my_text = "Hello World!". You can see I use the assignment operator (=). The variable (my_text) is placed at the left-hand side of the assignment operator, and the intended value is placed on the right-hand side of the assignment operator. So, you can say that using the previous line of code, I assign the value Hello World! to the variable my_text . Similarly, if I write x=25 , I say that I assign 25 to x . So, you never see the code like:

```
25=x  #This is an error
```

If by mistake, you do this, you’ll receive the syntax error. For your reference, I show the error from a Python shell:

```
>>> 25=x
  File "<stdin>", line 1
SyntaxError: cannot assign to literal
```

If you are coding for the first time, you may be confused about the usage of the assignment operator. Let me explain. From a mathematical point of view, it may appear to you that both these statements: x=y and y=x can have the same meaning, but in programming, it is different. You need to keep the information from previous paragraphs in your mind. To make it clear, you can make a small test using the following code segment in the Python shell:

```
>>> x=25
>>> y=50
>>> print(x)
25
>>> print(y)
50
>>> x=y
>>> print(x)
50
```

You can see that initially, x has the value of 25. But once you use the line of code:

x=y

The current value of y is assigned to x . So, when you print the value of x , you get 50 .

Conversely, if you use the following line of code:

y=x

The current value of x will be assigned to y . Let us test this too.

```
>>> x=25
>>> y=50
>>> y=x
>>> print(x)
25
>>> print(y)
25
```

I hope that you have got the idea!

Assigning Multiple Variables in a Single Line

In Python programming, you can assign multiple variables in one statement. Let us have a look into the following segment:

```
>>> x,y,z = 10,25,-303.5
>>> print(x)
10
>>> print(y)
25
```

```
>>> print(z)
-303.5
```

Here you can see that when you use the line of code:

x,y,z = 10,25,-303.5

The values 10, 25 , and 303.5 are assigned to x, y, and z respectively. A tuple is a comma-separated list of expressions. So, you can say that in the previous example, x, y, and z form one tuple, and 10,25 and -303.5 form another tuple. We refer to this assignment process as a ***tuple assignment***.

You will learn more about tuples in detail in Chapter 6

Assigning Same Value to Multiple Variables

Now let us explore another interesting feature in Python programming. You can assign the same values to multiple variables in one statement. For example, in the following code snippet, I am assigning all the three variables (x,y, and z) the same value (100).

```
>>> x=y=z=100
>>> print(x)
100
>>> print(y)
100
>>> print(z)
100
```

Operators

Operators are special symbols, which are used to carry out some kind of assignment or computations. These operators work on some values, which are termed ***operands***. For example, in the expression: 2+3 , + is an operator, and 2,3 are the operands.

In simple terms: a literal value, say 5 , or a variable, say empId are examples of simple expressions. You can combine values, variables, and operators, etc. to make a complex expression. We can check an expression to get a value at the end.

You have seen how to assign value to a variable using an assignment operator (=). Python supports many other operators which can be classified as follows:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators
- Identity operators
- Membership operators

Let us test some common operators. For your immediate reference, let us go through the following examples. These are very easy to understand. I’m executing them in the Python shell.

Arithmetic Operators

These are used to perform common mathematical operations. Before I use the arithmetic operators, I use two variables, x, and y. I assign them with the initial values 25 and 10 as follows:

```
>>> x=25
>>> y=10
```

Addition operator (+):

```
>>> print(x+y)
35
```

Subtraction operator (-):

```
>>> print(x-y)
15
```

Multiplication operator (*):

```
>>> print(x*y)
250
```

Division operator (/):

```
>>> print(x/y)
2.5
```

Modulus (Or, Remainder) operator (%)

```
>>> print(x%y)
5
```

Explanation: If you divide 25 by 10, 5 is the remainder.

Exponentiation operator (**)

```
>>> print(y**3)
1000
```

Explanation: 10*10*10=1000

Floor Division (//) operator:

```
>>> print(x//y)
2
```

Explanation: You get the answer to the nearest whole number. Consider another example: 11/3=3.666(approx); so, 11//3 gives you the answer as 3.

POINTS TO NOTE

The interactive Python shell can test both expressions and statements. For example, if you type 10, the shell can interpret it as 10. See the following segment:

```
>>> 10
10
```

But when you enter x=10 in the Python shell, it is treated as a correct syntactical statement with no value, and the shell prints nothing. In the next line, when you enter x, the shell can test the value of x. As a result, it can print the value of x   now. This is is why it is interesting to note that in Python shell, when I print the values, instead of typing print(x), you can simply type x to get the value of x. For example, see the following code segment.

```
>>> x=10
>>> x
10
```

But, by mistake, if you try to print the value of another variable, say, y which has not been defined already, you’ll encounter errors. Here is such a segment for your easy reference:

```
>>> y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

For the upcoming examples in this chapter, I use this shorthand form to print the value of a variable in the Python shell.

Assignment Operators

You can use these operators to assign some values to the variables. You can use them for less typing. For example, x+=2 is shorthand for x = x+2 . Similarly, x -= is a shorthand for x = x-2 . Let us test some of them. Before I show them in the following examples, in each case, I assign an arbitrary value 5 to x as follows:

```
>>> x=5
```

Using +=:

```
>>> x=5
>>> x+=2
>>> x
7
```

Using -=:
```
>>> x=5
>>> x-=3
>>> x
2
```

Using *=:
```
>>> x=5
>>> x*=2
>>> x
10
```

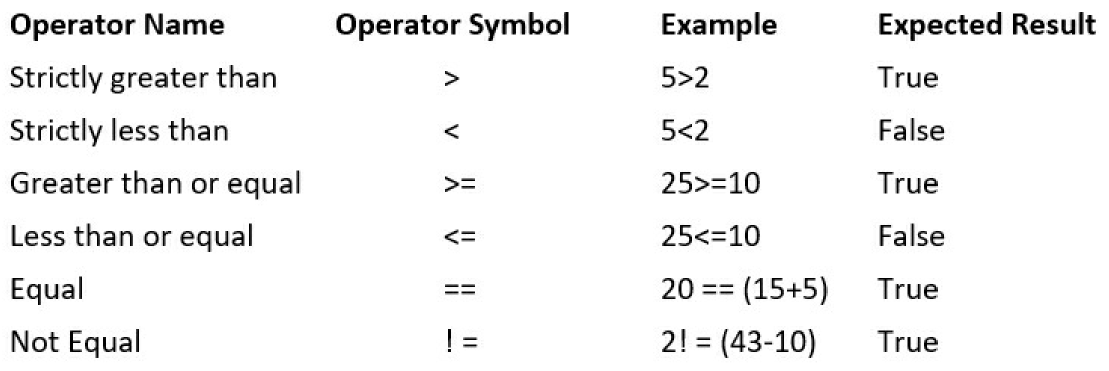
Using //=:
```
>>> x=5
>>> x//=2
>>> x
2
```

Similarly, you can see other assignment operators like ****=, /=, %=,** etc.

Comparison (or Relational) Operators

These operators are used to compare the equalities (or, inequalities) of two values (or, operands). The result of the comparison is a boolean value, i.e., either True or False . You can use Table 2-1 for your reference.

Table 2-1 The relational operators



Operator Name	Operator Symbol	Example	Expected Result
Strictly greater than	>	5>2	True
Strictly less than	<	5<2	False
Greater than or equal	>=	25>=10	True
Less than or equal	<=	25<=10	False
Equal	==	20 == (15+5)	True
Not Equal	! =	2! = (43-10)	True

Here I test some statements using the comparison operators in a

Python shell.

```
>>> 5>2
True
>>> 5<2
False
>>> 20==15+5
True
```

```
# Assigning 10 to x, 15 to y and
# then I'm using comparison operators.
```

```
>>> x = 10
>>> y = 15
>>> x == y
False
>>> x != y
True
>>> x == y-5
True
>>> x >= y
False
>>> x <= y-2
True
```

Logical Operators

You can use these operators when you work on conditional statements. Here you see the uses of "**logical and"**," logical **or"**, and "**logical not"** operators. To understand these, let us suppose you are testing two statements.

- For a Logical OR, if at least one statement is true, the combined result is True, otherwise False. You use **'or'** to denote logical OR.
- For a Logical AND, if both statements are true, the combined result is True, otherwise False. You use **'and'** to denote logical AND.
- A logical NOT does the opposite. It reverses the result.

For example, if the result is True, it reverses it to False and vice versa. You use **'not'** to denote logical NOT.

For an easy understanding, I show you some examples in a Python shell.

```
>>> 5>3 and 3>1
True
>>> 5>3 and 2>4
False
>>> 5>3 or 2>4
True
>>> not(5>3)
False
>>> not(2>3)
True
>>> not(5>3 or 2>4)
False
>>> x=10
>>> y=15
>>> x>=10 and y<5
False
>>> x>5 and y>7
True
```

Bitwise Operators

We use these operators for binary numbers. Here operands are integers, but Python treats them as binary digits. Here we compare bit-by-bit of the binary codes. This is why we call them bitwise operators. The common bitwise operators are:

- Bitwise OR: You use **'|'** to denote bitwise OR. When you compare two bits, the resultant bit is 1, if at least one of the two bits is 1.
- Bitwise AND: You use **'&'** to denote bitwise AND. When you compare two bits, the resultant bit is 1, if both

bits are 1.

- Bitwise XOR: You use **'^'** to denote bitwise XOR. When you compare two bits, the resultant bit is 1, if ONLY one of the two bits is 1.
- Bitwise NOT: You use **' ~ '** to denote bitwise NOT. This operator inverts all the bits in a binary number (i.e, you change 1 to 0 and 0 to 1 for each bit)
- Zero fill left shift: You use **'<<'** to denote the zero-fill left shift operator. Here x<<y simply means that the resultant x will appear with the bits shifted to the left by y places. The 0's are inserted as new bits on the right-hand side.
- Signed right shift: You use **'>>'** to denote the signed right shift operator. Here x>>y simply means that the resultant x will appear with the bits shifted to the right by y places and fills 0 on voids.

Let us understand this better. Consider two numbers 3 and 5. Suppose I represent them in 8-bit numbers. Now,3 can be represented in binary as 0000 0011

5 can be represented in binary as 0000 0101

Applying Bitwise OR on these numbers.

```
0000 0011
0000 0101
_______________
0000 0111
```

So, the result will be: $1*2^0+ 1* 2^1 +1* 2^2 =1+2+4=7$

Let us test this in Python shell now.

```
>>> 3|5
7
```

Applying Bitwise AND on the numbers (3 and 5) now.

```
0000 0011
0000 0101
_______________
```

0000 0001

So, the result will be: $1*2^0 = 1$
Let us test this in Python shell now.

```
>>> 3&5
1
```

Applying Bitwise XOR on the numbers (3 and 5) now.

```
0000 0011
0000 0101
______________
0000 0110
```

So, the result will be: $0*2^0 + 1*2^1 + 1*2^2 = 0+2+4=6$
Let us test this in Python shell now.

```
>>> 3^5
6
```

Understanding bitwise NOT operator is not very easy. Here are two most important points:

- You need to consider two’s complement binary when you apply these operators in Python. A two's complement binary is the same as the classical binary representation for positive integers, but it is slightly different for negative numbers. Negative numbers are represented by performing the two's complement operation on their absolute value.
- Negative numbers are written with a leading one instead of a leading zero. So, if you are using only 8 bits for your twos-complement numbers, then you treat patterns from "00000000" to "01111111" as the whole numbers from 0 to 127, and reserve "1xxxxxxx" for writing negative numbers. A negative number, -x, is written using the bit pattern for (x-1) with all the bits complemented (switched from 1 to 0 or 0 to 1). So, -1 is complement (1 - 1) = complement (0) = "11111111", and -10 is complement (10 - 1) = complement (9) =

complement ("00001001") = "11110110". This means that negative numbers go all the way down to -128 ("10000000").

As per the previous bullet point, you can say:
-4 = Complement for (4-1)= 3 , which says **~3** is **-4**
-6 = Complement for (6-1)= 5, which says **~5** is **-6**
Therefore, in the Python shell you see the following results:

```
>>> ~5
-6
>>> ~3
-4
```

In this context, it is useful to note that you can see the binary representation of a number in string format using the in-built bin() function in Python. Here are some examples for you:

```
>>> bin(2)
'0b10'
>>> bin(3)
'0b11'
>>> bin(~3)
'-0b100'
>>> bin(-4)
'-0b100'
>>> bin(5)
'0b101'
>>> bin(~5)
'-0b110'
>>> bin(-6)
'-0b110'
```

POINTS TO NOTE

The link https://wiki.python.org/moin/BitwiseOperators also says that Python doesn't use 8-bit numbers. To remove portability issues, now it considers an INFINITE number of bits. Thus, the number -5 is treated by bitwise operators as if it were written "...1111111111111111111011"

Applying << on the numbers 3 and 5:
3 is in binary: 0000 0011
Now 3<<2 becomes: 0000 1100 i.e.,12

5 is in binary: 0000 0101

So, 5<<1 becomes 0000 1010 i.e., 10
Let us test these in a Python command shell now.

```
>>> 3<<2
12
>>> 5<<1
10
```

Applying >> on the numbers 3 and 5:
3 is in binary: 0000 0011
Now 3>>2 becomes: 0000 0000 i.e.,0

In the same way, 5>>1 becomes 0000 0010 i.e., 2
In the same way, 5>>2 becomes 0000 0001 i.e., 1

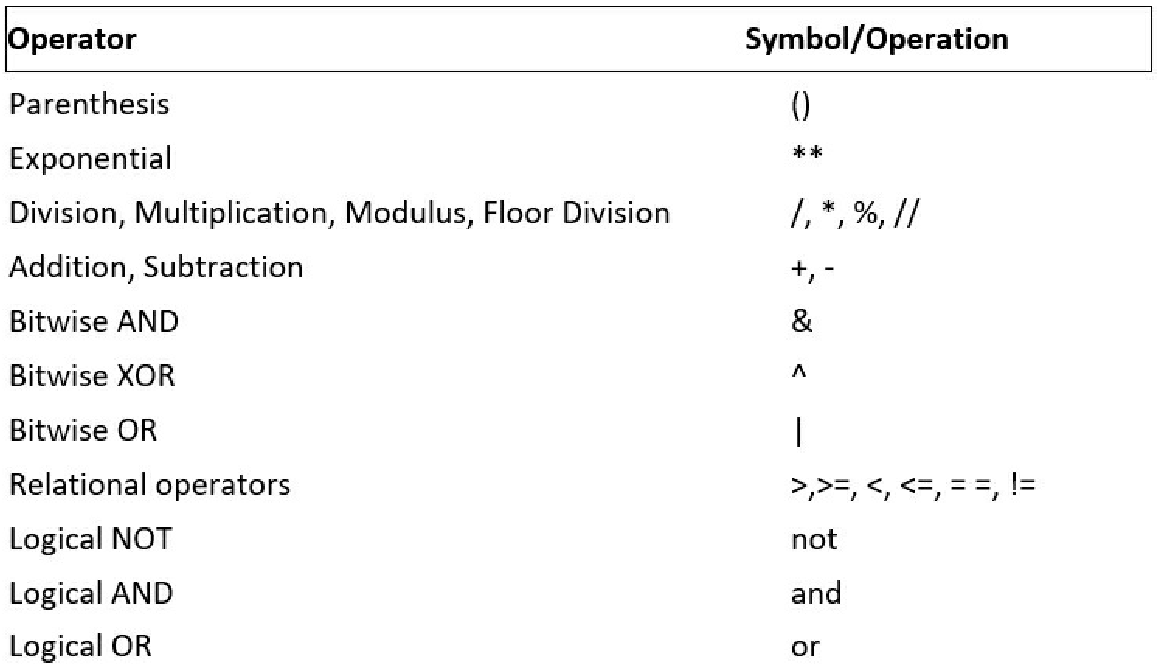
Let us test this in Python shell now.

```
>>> 3>>2
0
>>> 5>>2
1
>>> 5>>1
2
```

Precedence of Operators

The operators follow the hierarchy which is shown in the following table. I’m showing the precedence of the common operators in decreasing order. Please note that parenthesis is not an operator but it has the highest precedence. I do not want you to miss this information. See Table 2-2 for your reference.

Table 2-2 Operator precedence



Operator	Symbol/Operation
Parenthesis	()
Exponential	**
Division, Multiplication, Modulus, Floor Division	/, *, %, //
Addition, Subtraction	+, -
Bitwise AND	&
Bitwise XOR	^
Bitwise OR	|
Relational operators	>,>=, <, <=, = =, !=
Logical NOT	not
Logical AND	and
Logical OR	or

Example:

```
>>> 3*4**2+2
11.0
```

Explanation:
In this expression, ** has the highest precedence here. So, 4**2 computes first and we get 16. And the resultant expression becomes 3*16+2
In this expression, * has the highest precedence. So, 3*16 will compute now and you get 48.
So, now the expression becomes 48+2 which is 50

Example:

```
>>> 1+3&5
4
```

Explanation:
As per the precedence table (See Table 2-2) shown earlier, arithmetic operators have higher precedence than bitwise operators. So, the given expression evaluates in the following order:

```
1+3&5
=4&5
=4
```

Operators Associativity

Sometimes in an expression, you may see the operators with the same precedence. Most of the operators except exponential operator (**) evaluates from left to right. In this context, you can remember that you used assignment operators for variables, such as x=5 . This evaluates from right to left.

Example:

```
>>> 5*8%3
1
```

Here * and % have the same precedence and they both evaluate from left. So, 5*8%3 becomes 40%3 which is 1 . [Notice that, if you evaluate 8%3 first, you’ll get the final result as 5*2=10, which is not correct].

Example:

```
>>> 10*2//3
6
```

Explanation:
In this expression, * and // have the same precedence. Also, both evaluate from the left. So, 10*2//3 becomes 20//3 which is 6 .

Note that you can always change the order of the evaluation if you use parenthesis as follows: 10*(2//3), you get 0 . See the following in a Python Command shell:

```
>>> 10*(2//3)
0
```

Example:

```
>>> x=10
>>> y=20
>>> z=30
>>> (x<y) & (y>z)
False
>>> (x<y) | (y>z)
True
```

Explanation:

Here I present this example to show that you can apply bitwise operators on conditional statements (that includes comparison operators). Here x<y is True , but y>z is False . And True & False results False , but True|False results True .

Identity Operators

You can use ‘ is’ and ‘ is not’ to test whether two objects are the same. Since I discuss class and objects in Chapter 11, I do not include the discussion here.

Membership Operators

You use the operators to test for the membership. In simple words, you can use ‘in’ and ‘ not in’ to test whether an element is present in a sequence. In Chapter4, you’ll see the usage. So, I do not include the discussion here.

EXERCISE

E2.1 Predict the output.

```
x='12.2'
print('x')
print(x)
```

E2.2 Predict the output.

```
x=23
y=10
print(x)
print(y)
print(z)
```

E2.3 How can you assign multiple variables in a single statement? Give an example.

E2.4 Identify the invalid statements among the following:

```
i)abc=1
ii)if=2
iii)$fi=3
iv)_public=4
v)abc$=5
vi)bob's=6
vii)emp_id=7
```

viii)emp id=8

E2.5 Write a program to output the following lines where you use a single variable to store the name(s).

Welcome our favorite hero:
John
He is changing his name to:
Sam

E2.6 Predict the output of the following expression:

2+3*(36/9+1)**2-1

E2.7 Predict the output of the following expression:

3+2&4|2

E2.8 Which one you prefer -more comments inside your code, or fewer comments inside the code?

E2.9 Write a python program to print the following shape:

```
         *
     *       *
  *              *
*-------------------*
```

Solution to Exercises

E2.1.

```
x
12.2
```

E2.2.

```
23
10
Traceback (most recent call last):
  File "E:/MyPrograms/Python/PythonCrashCourse/chapter3/chap3_quiz.py",
line 11, in <module>
    print(z) #error
NameError: name 'z' is not defined
```

E2.3

```
x,y,z=1,2,3
```

E2.4

```
i)abc=1 #ok
ii)if=2 #error; ’if’ is a Python keyword.
iii)$fi=3 #error
iv)_public=4 # will work, but not recommended
v)#abc$=5 #error
vi)bob's=6 #error
vii)emp_id=7 #ok
viii)emp id=8 #error
```

E2.5

```
name="John"
print("Welcome our favorite hero:")
print(name)
name="Sam"
print("He is changing his name to:")
print(name)
```

E2.6

```
>>> 2+3*(36/9+1)**2-1
76.0
```

Explanation:
In this expression, () has the highest precedence. So, (36/9+1) computes first. In this sub-expression, / has higher precedence than +. So, it results in 4.0+1=5.0 and the resultant expression becomes 2+3*5.0**2-1

In this expression, ** has the highest precedence. So, 5.0*2 will compute now. It results in 25.0 and the resultant expression becomes 2+3*25.0-1

In this expression, * has the highest precedence. So, 3*25.0 will compute now. It results in 75.0 and the resultant expression becomes 2+75.0-1

Now + and – has the same precedence in the resultant expression. Both these operators are left-associative (evaluates from left to right). So, the addition will be done first and the resultant expression becomes 77.0-1, which produces the final output as 76.0

E2.7

```
>>> 3+2&4|2
6
```

Explanation:
In this expression, + has the highest precedence, then & and the |. So, the expression evaluates as follows:

```
3+2&4|2
=5&4|2
=4|2
=6
```

E2.8

It depends. If a comment helps another developer to understand or review your code, it has significant value. But you need to use your intelligence. For example, you do not want to include too many unnecessary comments to describe a code that is easy to understand.

E2.9

```
print("         *            ")
print("     *       *        ")
print("  *              *    ")
print("*-------------------* ")
```

General Comments for Case Studies

Before you implement the projects or case studies, I want you to note the following points:

- You'll see the supporting comments in all these implementations. I also add some useful information at the beginning of a project. I use them to help you understand the important segments of code in a project.
- You get the complete code at the end of a chapter. I already discuss these codes (or similar codes) in the chapter or a previous chapter. So, I do not repeat the code explanations. If you do not understand a line of code, read the chapter again. This simple activity helps you to refresh your knowledge.
- In Python programming, you often organize the code using functions. You call them to perform the intended job and make your code more Pythonic. In chapter7, you'll learn about functions in detail. Once you learn them, you can beautify the initial implementations I show you in this book.
- You need to guard your application against unwanted user inputs/scenarios. Chapter 8 teaches you how to do that.
- I show you the simple solutions at the beginning. It is because your initial aim is to meet the least essential requirements. You can ignore the remaining corner cases when you implement them for the first time. Later you can improve these projects. For example, you can ignore invalid user inputs for the initial case studies. After you complete reading Chapter 7 and Chapter 8, you can consider those cases and improve your solutions.

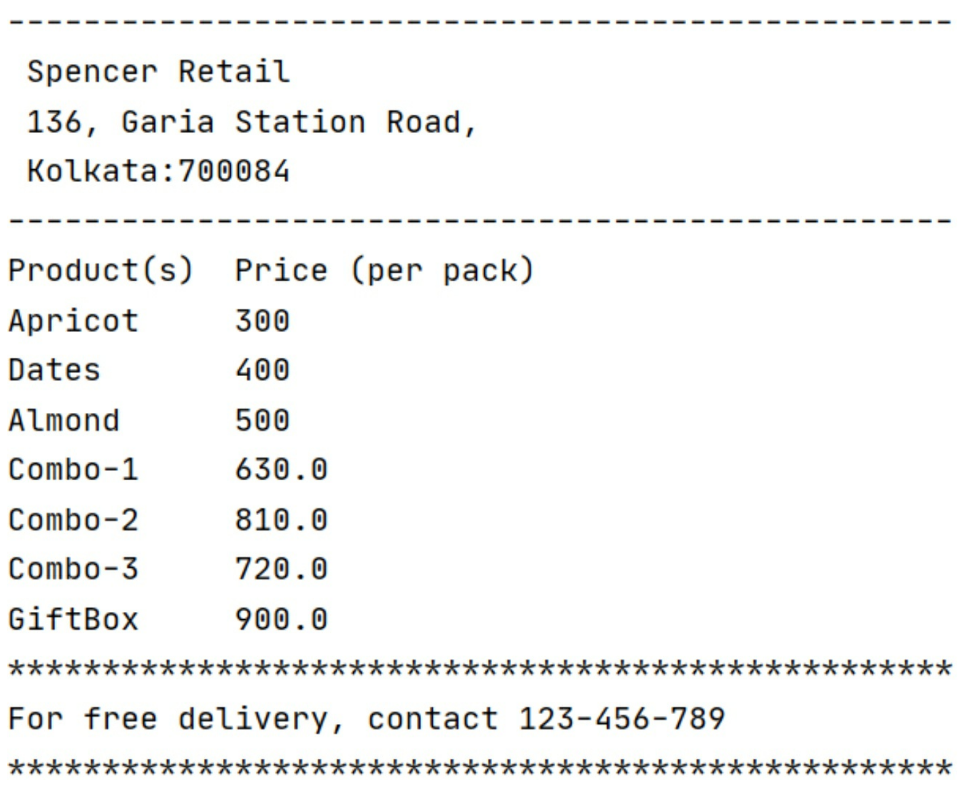
Case Study I

CS 1.1 Problem Statement

Welcome to your first project. As a creator, you always like to imagine the final product in advance. Here I want you to develop a company catalog. Assume that this company sells three different dry fruits-Apricot, Dates, and Almond. The seller can sell individual items or a combination of these items. A gift pack is a special combination that contains all three items. Here are some special considerations:

- If a customer purchases individual items, he does not receive any discount.
- If a customer purchases a combo pack with two unique items, he gets a 10% discount.
- If the customer purchases a gift pack, he gets a 25% discount.

For illustration purposes, I choose an Indian retailer (Spencer). I also show the price of an item in Indian Rupees. The final output should look like the following (See **Figure CS1.1-1)**:



```
--------------------------------------------------
 Spencer Retail
 136, Garia Station Road,
 Kolkata:700084
--------------------------------------------------
Product(s)  Price (per pack)
Apricot     300
Dates       400
Almond      500
Combo-1     630.0
Combo-2     810.0
Combo-3     720.0
GiftBox     900.0
**************************************************
For free delivery, contact 123-456-789
**************************************************
```

Figure CS1.1-1*:The final output of the project.*

Are you ready?

Chapter 3: Common Data Types

This chapter shows you the usage of some common data types and the built-in functions in Python. In simple words, a function is a block of code that helps you to perform an intended job. Using the words “built-in functions”, I mean that these are already written and available for you. So, you can directly use them in your program. You’ll learn more details about functions later in the Chapter7 of this book.

There are many data types in Python. As a beginner, to proceed further, you need to be familiar with strings, numbers, and booleans. These are the most common data types, and you see them almost in every program. We’ll cover them quickly in the upcoming sections.

Instead of creating separate files for each of these small code segments, I place them into a single file. I also keep the comments for your reference. If you want, you can write each segment in separate files, or you can simply execute them in a Python shell. All are fine.

Code Demonstrations with Strings

In Chapter2, I gave you a brief introduction to strings. Here you’ll know more about them. Any string can include letters (A-Z, a-z) and digits (0-9). Other printable characters such as &,$, *, and# can be included in a sting too. You can also include special characters (often termed as **control codes**) followed by a backslash (\). For example, **\n** represents an escape character. When you use it in a string, the character ‘n’ is escaped, and the text cursor moves down to the next line. Here is an example for you:

```
>>> print("Hello\nWorld!")
Hello
World!
```

Similarly, if you want to print a tab between letters, you can use \t . Here is an example:

```
>>> print("Hello\tWorld!")
Hello   World!
```

The **\n** and **\t** are very common. Apart from these, you may see the use of **\b** for backspace, **\a** for sounding a beep sound, etc. But the behavior of \ **b** and **\a** can vary. To illustrate, each of them works fine in a Python command shell, but they do not work in IDLE as per the previous description. To show this, let us use a Python command shell first:

```
>>> print("Abc\bd")
Abd
```

Now execute the same line of code in IDLE. In this case, you see the following output:

```
>>> print("Abc\bd")
Abcd
```

You can see the difference!

Sometimes, you may not want backslash (\ ) to use for escape characters. For example, you may want to print the string with the backslash. In this case, you can use raw strings by adding an **r (or R)** before the double quotation as follows:

```
>>> print(r"Hello\World")
Hello\World
```

Before you read further, I want you to note another particular type of Python statement. I use this approach in various examples and projects in this book to decorate top borders or bottom borders. Here is a sample statement:

```
print("*"*10)
```

How does it work? This statement prints the star 10 times. In the same way, you can use:

```
print("-"*15)
```

to print the character ‘ –‘ 15 times. What is the benefit? You can type less and make your code size short. I show you some sample use cases of this for your immediate reference.

```
>>> print("*"*10)
**********
```

```
>>> print("-"*15)
---------------
>>> print("#"*5)
#####
```

Now I’ll show you some other common use cases of strings. Go through the following code fragments:

Goal:
I want to print HelloWorld! inside single quotations.

Code:

```
print("'HelloWorld!'");
```

Expected Output:

```
'HelloWorld!'
```

Goal:
I want to print HelloWorld! inside double-quotations.

Code:

```
#Trying to print HelloWorld! inside double quotations
print(""HelloWorld""); #Error
```

Expected Output:

```
print(""HelloWorld""); #Error
            ^
SyntaxError: invalid syntax
```

Explanation:
To handle this case, you can use escape characters like the following:

```
#Using the escape characters
print(" \"Hello World! \" ")
```

Here you simply tell Python to insert the character (which is a double quotation in this case) after the backslash. Now you can get the following output:

```
"Hello World! "
```

Alternatively, you can use double quotations inside single quotations as follows:

```
>>> print(' "Hello World!" ')
"Hello World!"
```

Goal:
I want to print HelloWorld! using a string variable.

Code:

```
my_text="Hello World!"
print(my_text)
```

Expected Output:

```
Hello World!
```

Explanation:
You are familiar with this code. Here you print the text message using a string variable named my_text .

Goal:
I want to concatenate multiple strings.

Code:

```
# Concatenation example
text1 = "Hello, Reader!"
text2 = " How are you?"
text3 = " Hope everything is fine."
print(text1+text2+text3)
```

Expected Output:

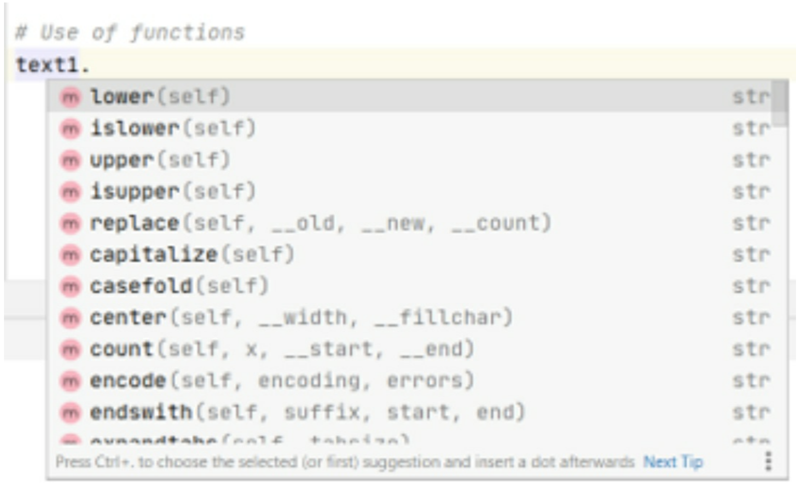
Hello, Reader! How are you? Hope everything is fine.

Explanation:
Here you see the plus (+) operator is in action. Using this, you can concatenate strings. This activity is common in computer programming.

Now I show you the usage of some built-in functions. You will see a detailed discussion on functions in Chapter7 of the book. I told you before that a function is a block of code to perform an intended job. You can write your function(s) or, you can use the functions that are already written in Python. I also told you that by using the words "built-in functions", I mean that you can directly use them in your code. Now the obvious question is: how will you know about the available functions? The use of an IDE can help you in this case. You know that I am using PyCharm IDE for this book. It gives me some special supports when I invoke (or, call) the functions. For example, suppose, I have the following code:

text1="Hello, Reader!"

Now when I write text1 and then put a **dot(.)**, I can see available functions for me. Here is a screenshot for you:



```
# Use of functions
text1.
  m lower(self)                                str
  m islower(self)                              str
  m upper(self)                                str
  m isupper(self)                              str
  m replace(self, __old, __new, __count)       str
  m capitalize(self)                           str
  m casefold(self)                             str
  m center(self, __width, __fillchar)          str
  m count(self, x, __start, __end)             str
  m encode(self, encoding, errors)             str
  m endswith(self, suffix, start, end)         str
  m expandtabs(self, tabsize)                  str
Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards  Next Tip
```

Figure 3-1:Showing available functions when you type a string variable and put a dot(.)

Let us test some of these functions.

Goal:
I want to print HelloWorld! in uppercases and lowercases.

Code:

```
text1 = " Hello, Reader!"
print("The original string is:" + text1)
# Printing text1 in uppercase
print(text1.upper())
# Printing text1 in lowercase
print(text1.lower())
```

Expected Output:

```
The original string is: Hello, Reader!
HELLO, READER!
hello, reader!
```

Explanation:
You can see the function upper() is converting the original string into uppercase characters, and lower() is doing the opposite.

Goal:
I want to use multiple functions together.

Code:

```
# Using multiple functions together
text1 = " Hello, Reader!"
print("The original string is:"+ text1)
print(text1.upper().islower()) #False
print(text1.upper().isupper()) #True
```

Expected Output:

```
The original string is: Hello, Reader!
```

```
False
True
```

Explanation:
This fragment of code shows that you can use multiple functions together. For example, text1.upper().islower() is actually doing two things:
At first, it performs text1.upper() which converts the string into uppercase characters, and then it invokes the islower() function on the resultant string. The islower() function verifies whether the string is in lowercase or not. Since the original string is converted in uppercase characters already, it returns False .
The next line of code is testing the reverse scenario. The isupper() function is used to test whether the resultant string is in uppercase or not. Before you invoke this function, you transformed the string to uppercase characters. This is why you get True in the output.

Goal:
I want to calculate the length of a string.

Code:

```
text1 = "Python"
text2 = "Hello, Jon!"
print("The text1 is: " + text1)
print("Length of text1 is as follows:")
#Length of text1
print(len(text1))
print("The text2 is: " + text2)
print("Length of text2 is as follows:")
#Length of text2
print(len(text2))
```

Expected Output:

```
The text1 is: Python
Length of text1 is as follows:
6
The text2 is: Hello, Jon!
```

```
Length of text2 is as follows:
11
```

Explanation:
The len() function is used to calculate the length of a string. Here I have calculated the length of two different strings using this function.

POINTS TO REMEMBER

Note that I’ve used len(text1) and len(text2). Programmatically, when you write similar codes, you say that you pass text1 and text2 as method arguments (often called parameters). Expert programmers are particular about the terminologies. Ideally, in this case, you should say that len() is a function that can accept a string parameter. But when you use len(text1) , you should say that I’m passing a string argument text1 inside the len() function. I’ve included a discussion on this in Chapter7.

Goal:
I want to examine the index positions of the characters inside a string.

Code:

```
text1 = "Python"
print("The text1 is: " + text1)
# Printing individual characters inside the string
print(text1[0])
print(text1[1])
print(text1[2])
print(text1[3])
print(text1[4])
print(text1[5])
```

Expected Output:

```
The text1 is: Python
P
```

```
y
t
h
o
n
```

Explanation:
Here I have shown you all the index positions from 0 to 4. Notice that the array indexing starts from the 0th position. So, a,b,c,d, and e are stored at index positions 0,1,2,3 , and 4 respectively. This is why, text1[0] prints the first character a, text1[1] prints the next character, and so on. You see the same in many other high-level languages, such as Java, C++.

Goal:
I want to get the first occurrence of a character (or, a word) inside a string.

Code:

```
text = "abcABc"
print("The text is: " + text)
# Printing individual characters inside the string
print("The first occurrence of 'A' is at index:")
print(text.index("A"))
print("The first occurrence of 'c' is at index:")
print(text.index("c"))
print("The first occurrence of 'bcA' is at index:")
print(text.index("bcA"))
```

Expected Output:

```
The text is: abcABc
The first occurrence of 'A' is at index:
3
The first occurrence of 'c' is at index:
2
The first occurrence of 'bcA' is at index:
1
```

Explanation:
As shown here, you can use the index() function to retrieve the first occurrence of a particular character inside a string. Notice that inside the string, we had two c’s; one is at index 2 and another is at 5 . But the function returns the index position 2 . Also, I searched for the index position of the combined characters bcA . So, you can use the same function to find a particular word inside a string too.

Goal:
I want to examine whether an intended character is absent inside a string.

Code:

```
text="Hello, John!"
print(text.index("y"))#error now
```

Expected Output:

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter3/string_usage_file_2.py
line 16, in <module>
    print(text.index("y")) #error now
ValueError: substring not found
```

Explanation:
The error message is self-explanatory. You see this error because the character y   is not present inside the string Hello, John!

Goal:
I want to examine a function that can has multiple parameters. This is why I’m showing you an example using the replace() function.

Code:

```
# I want to examine a function that accepts multiple
# parameters.Using the replace() function for this example.
```

```
text = " Hello, John!"
print("Initial text is :" + text)
print("Replacing the name 'John' with 'Bob' now.")
text = text.replace("John", "Bob")
print("The changed text is :" + text)
```

Expected Output:

```
Initial text is : Hello, John!
Replacing the name 'John' with 'Bob' now.
The changed text is : Hello, Bob!
```

Explanation:
Notice that the replace() function takes two arguments. I use the first one for the string which I replace, and the second one I use for the string which reflects the changed value. So, to change the name John with Bob , I have used text.replace("John", "Bob") , and I hold this changed value into the same string variable text .

Code Demonstrations with Numbers

In Chapter2, I gave you a brief introduction to Numbers. In this section, you are going to explore more about them. Before I proceed further, it's worth remembering some important points which are as follows:

Like strings, to print a number without a variable, you can just type the number inside the print() function. See the following code segment with inline comments.

```
print(1)#Prints 1
print(5.7)# Prints 5.7
print(-6.789)# Prints -6.789
```

You can perform both the basic and complex arithmetic operations inside the print statement. In chapter2, you learned about the precedence and associativity of operators. If you understood the concept, you face no problem to predict the output in advance. Consider the following code segment with comments for your quick reference:

```
#Performing some basic and complex arithmetic operations
print(1+2) #Prints 3
print(10 - 3) #Prints 7
print( 25* 3) #Prints 75
print(12.88/4) #Prints 3.22
print(1+2*3)#Prints 7
print((1+2)*3) #Prints 9
```

You have also used variables for numbers. Here is a sample code segment to examine the usage of number variables:

```
# Using variables
my_int=125
print(my_int)# Prints 125
my_float=25.763
print(my_float)# Prints 25.763
```

Remember that for string variables, the plus (+) operator concatenates the strings, but for numbers, it adds them. Here is a sample for you:

```
#Difference between number and strings
my_string1="10"
my_string2 = "22"
my_int1=10
my_int2=22
print(my_string1+my_string2) #Prints 1022
print(my_int1+my_int2) #Prints 32
```

One question may appear in your mind: whether we can use both numbers and strings together inside a print statement? The answer is yes, and you need to do this very often. A simple solution is to write something like:

```
print("The value inside my_int1 is:", my_int1)
```

Which can produce the following output:

```
The value inside my_int1 is: 10
```

You can follow various approaches to print the output in the console. For example, you can use comma's inside the print() function. You can also use string concatenation (using + operator). You can also use string interpolation (using the format () functions) too. From Python 3.6, format literals (it uses prefix f) are also available. You've just started learning Python programming. So, let us stick to the simple forms at this stage. I show you f-strings usage at the end of this chapter.

But if you want to use the + operator, you need to do a trick, in which you convert them into the same datatype. Otherwise, you'll see errors.

For example, if you write:

```
print("The value inside my_int1 is  :" +my_int1) #error
```

You'll get the errors like:

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/Chapter3/numbers_usage_file_1
line 38, in <module>
    print("The value inside my_int1 is  :" +my_int1)
TypeError: can only concatenate str (not "int") to str
```

The error is self-explanatory. You see this error because you can do concatenation for strings, but not for a string and a number. Here my_int contains a whole number (10). So, programmatically it is an **int**. So, you see the word 'int' in the error message. (When a number has a fractional part, say 10.2, it is a **float** number. I discussed the difference between int and float in Chapter2).

Again, if you write and execute the following line of code:

```
print(my_int1 + " is my first number") #error
```

You will again receive an error:

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/Chapter3/numbers_usage_file_1
```

```
line 38, in <module>
    print(my_int1 + " is my first number") #error
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

This time in the print function, you place the number variable before the string variable. So, you see this error. What is the solution? You have guessed it right! Make both of them the same data type. For example, you can convert the number into a string using the str() function as follows:

```
print("The value inside my_int1 is:" +str(my_int1)) #Ok
```

If you execute this code, you can receive the following output now:

```
The value inside my_int1 is:10
```

Now I show you a function to convert a string to an integer. See the following code segment.

```
my_string1="12"
#Converting a string to an int
print(int(my_string1,10)) #prints 12
```

Here 10 is used to state the base of the number. But all strings are not convertible to an integer. For example, if you try to execute the following code:

```
# All strings are NOT convertible to an int
my_string2 = "abc"
print(int(my_string2, 10))  # Error
```

You'll receive the error like the following (the line number can vary in your file):

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/Chapter3/numbers_usage_file_1
line 48, in <module>
    print(int(my_string2, 10))  # Error
ValueError: invalid literal for int() with base 10: 'abc'
```

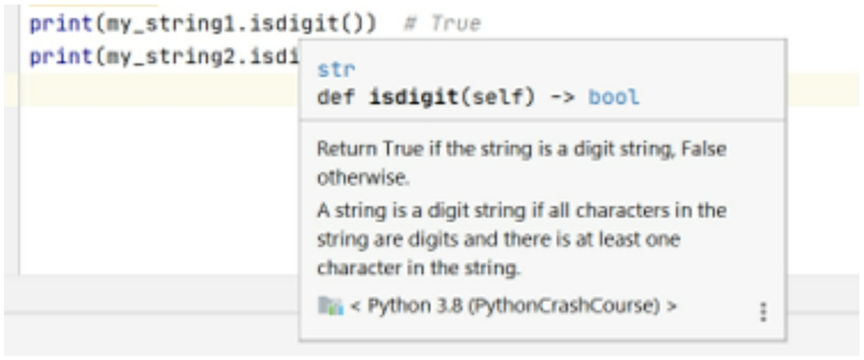
You can use the isdigit() function to test whether a string is convertible to a true digit or not. For example, if you execute the following code segment:

```
my_string1="25"
my_string2 = "abc"
print(my_string1.isdigit())  # True
print(my_string2.isdigit()) # False
```

you’ll receive the following output:

```
True
False
```

If you use PyCharm IDE, when you move your cursor on the function, you can see the function definition easily. For your easy reference, I take a screenshot from this IDE when I move my cursor on the isdigit() function (See Figure 3-2):



```
print(my_string1.isdigit())  # True
print(my_string2.isdi
```

```
str
def isdigit(self) -> bool
```

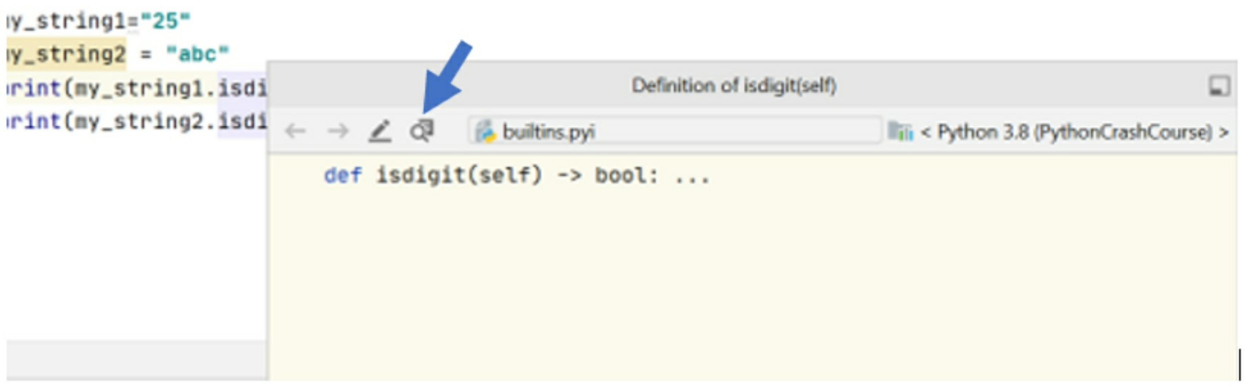
Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

< Python 3.8 (PythonCrashCourse) >

Figure 3-2: Description of isDigit() function from a PyCharm IDE

Or, you can press **Ctrl+Shift+I** to get the following screen (See Figure 3-3)

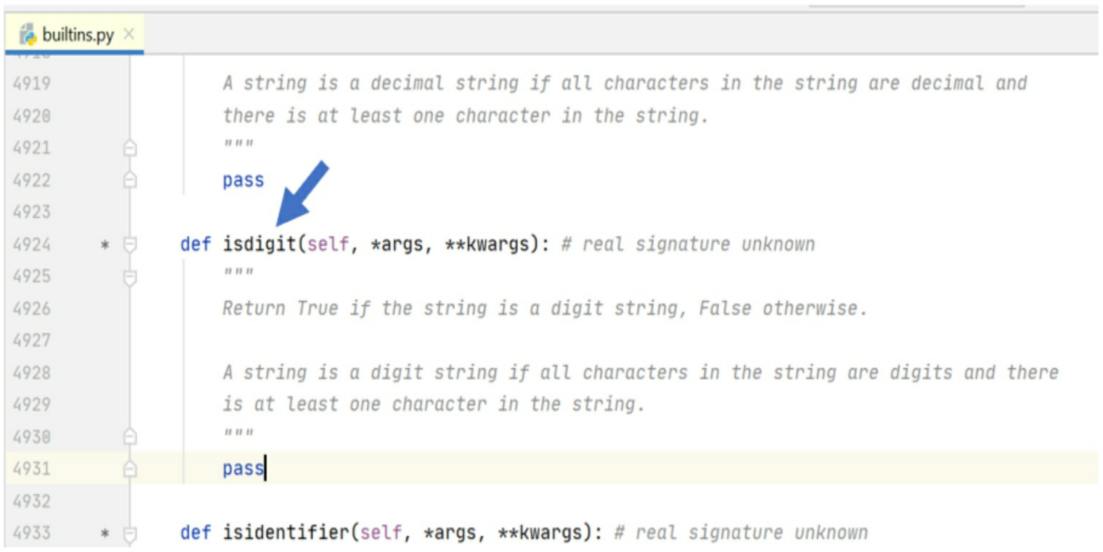

Definition of isdigit(self)
builtins.pyi
< Python 3.8 (PythonCrashCourse) >

```
y_string1="25"
y_string2 = "abc"
rint(my_string1.isdi
rint(my_string2.isdi
```

```
def isdigit(self) -> bool: ...
```

Figure 3-3: Retrieving the definition of isDigit() function

Now see **Show Source(Ctrl +Enter)** to get the details (See Figure 3-4):



```
    A string is a decimal string if all characters in the string are decimal and
    there is at least one character in the string.
    """
    pass

def isdigit(self, *args, **kwargs): # real signature unknown
    """
    Return True if the string is a digit string, False otherwise.

    A string is a digit string if all characters in the string are digits and there
    is at least one character in the string.
    """
    pass

def isidentifier(self, *args, **kwargs): # real signature unknown
```

Figure 3-4: Retrieving the description of isDigit() function from a PyCharm IDE

These simple activities can make your programming life easy. This is another valid use case, which shows the effectiveness of an IDE over a normal Python shell.

Quick talk on f-strings

Python 3.6 introduced the concepts of the f-strings. You can put the letter "**f**" in front of a string and then inject a variable into it. I like this approach because it is easy to read and understand. To inject a variable inside a string, you need to wrap it inside the curly brackets. Let us look at the following example:

```
user_name="John"
print(f"Hello, {user_name}!")
```

This code segment can produce the following output:

```
Hello, John!
```

Now you are ready to examine some built-in functions for numbers. Let us proceed.

Goal:
I want to round a number.

Code:

```
# Rounding a number
print(round(2.7))  # Prints 2
print(round(5.32))  # Prints 5
```

Expected Output:

```
3
5
```

Goal:
I want to find the maximum and minimum from a set of numbers.

Code:

```
#Finding the maximum
print(max(1,2,3,4,5)) # Prints 5

#Finding the minimum
```

```
print(min(1,2,3,4,5)) # Prints 1
```

Expected Output:

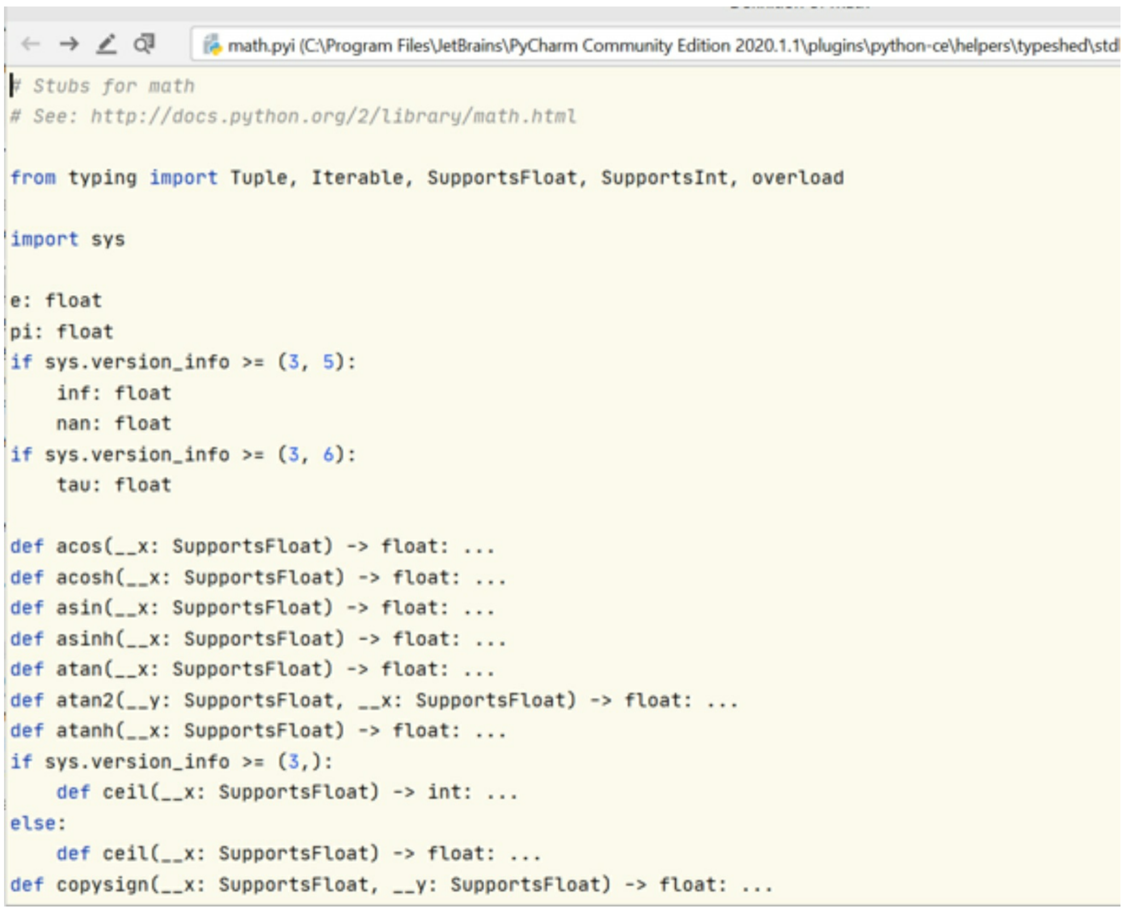
```
5
1
```

Explanation:
You can see the max() function is used to find the greatest among the numbers and the min() function is used to find the smallest among the numbers.

Now I import some math functions and use those functions. So, at the beginning of the file, I write the following:

```
# importing math functions
from math import *;
```

It helps me to get the functions sqrt(), ceil(),floor() etc. If you do not import the math functions, you cannot use (or access) these functions. So, if you see the source (For example, as said before, put your cursor on math and press Ctrl+Shift+I), you can see a screen like the following (see Figure 3-5):


math.pyi (C:\Program Files\JetBrains\PyCharm Community Edition 2020.1.1\plugins\python-ce\helpers\typeshed\std

```
# Stubs for math
# See: http://docs.python.org/2/library/math.html

from typing import Tuple, Iterable, SupportsFloat, SupportsInt, overload

import sys

e: float
pi: float
if sys.version_info >= (3, 5):
    inf: float
    nan: float
if sys.version_info >= (3, 6):
    tau: float

def acos(__x: SupportsFloat) -> float: ...
def acosh(__x: SupportsFloat) -> float: ...
def asin(__x: SupportsFloat) -> float: ...
def asinh(__x: SupportsFloat) -> float: ...
def atan(__x: SupportsFloat) -> float: ...
def atan2(__y: SupportsFloat, __x: SupportsFloat) -> float: ...
def atanh(__x: SupportsFloat) -> float: ...
if sys.version_info >= (3,):
    def ceil(__x: SupportsFloat) -> int: ...
else:
    def ceil(__x: SupportsFloat) -> float: ...
def copysign(__x: SupportsFloat, __y: SupportsFloat) -> float: ...
```

Figure 3-5: A partial snapshot of available math functions

And if you go through the file, you can see the functions named ceil(),floor(),sqrt(),gcd(),factorial() etc. Let us use some of them in the following section:

POINT TO REMEMBER

As said before, do not forget to import the math functions before you use the following functions. In Chapter 7, you'll learn about modules and see import statements. Using an import statement, you make the previously written codes available in a current file. For example, in many project implementations, I use:

from random import randint

It helps me to get the in-built randint() function from the random module. Once I import this statement, I can generate a random number using this function. For example, randint(1,15) can generate a random number between 1 and 15, including both endpoints.

Goal:
I want to get the square root of a number.

Code:

```
# Printing the square root of 25
print(f"Square root of 25 is:{sqrt(25)}")  # 5.0
# Printing the square root of 6.25
print(f"Square root of 6.25 is:{sqrt(6.25)}")  # 2.5
```

Expected Output:

```
Square root of 25 is:5.0
Square root of 6.25 is:2.5
```

Goal:
I want to find the ceiling value and the floor value of a number.

Code:

```
# Printing the ceiling value
# This is the smallest integer less than the number
# you defined.

print(f"The ceiling value of 39.3 is:{ceil(39.3)}")  # 40
# Printing the floor value
# This is the biggest integer NOT greater than the number
# you defined.
print(f"The floor value of 39.3 is: {floor(39.3)}")  # 39
```

Expected Output:

```
The ceiling value of 39.3 is:40
The floor value of 39.3 is: 39
```

Explanation

Refer to the comments for your understanding.

Goal:
I want to find the gcd (greatest common divisor) of two numbers.

Code:

```
# Finding the gcd of 4 and 14
print(f"The gcd of 4 and 14 is:{gcd(4,14)}")  # prints 2
# Finding the gcd of 14 and 63
print(f"The gcd of 63 and 14 is:{gcd(63,14)}")  # prints 7
```

Expected Output:

```
The gcd of 4 and 14 is:2
The gcd of 63 and 14 is:7
```

Goal:
I want to find the factorial of a number.

Code:

```
# Finding the factorial of 5
print(f"The factorial of 5 is: {factorial(5)}")
# Finding the factorial of 7
print(f"The factorial of 7 is: {factorial(7)}")
```

Expected Output:

```
The factorial of 5 is: 120
The factorial of 7 is: 5040
```

Explanation:

The factorial of 5 is: 5*4*3*2*1=120
The factorial of 7 is: 7*6*5*4*3*2*1=5040

Accepting User Inputs

Now I show you how to accept user input in a program. You use the input() function in this context. Using this function, you can pass an optional prompt. The prompt can help a user know about the information he needs to supply.

When you run the following demonstration, you’ll see that the input() function pauses the program and waits for the user’s input. Once you supply the input, you can proceed further.

Demonstration 1

To examine this, let us write a simple program where the user can supply his name and age. I store these inputs inside some variables. Then I print the information with some additional messages. Let us type the following lines in your Python file and save the content. Then run the program.

```
# Supply a name
user_name = input("Enter your name:")
# Enter the age
user_age = input("Enter your age:")
# Using commas
print("Welcome,",user_name, "! you are now ", user_age)
# Using f-strings(Python 3.6 onwards)
print(f"Welcome,{user_name}! you are now {user_age}")
# Printing using string concatenation
print("Welcome," + user_name + "! you are now " + str(user_age))
```

Output

You should see the following output:

```
Enter your name:John
Enter your age:25
Welcome, John ! you are now  25
Welcome,John! you are now 25
Welcome,John! you are now 25
```

You have seen how to supply user inputs to your program. This program also shows the different ways to print the output.

EXERCISE

E3.1 Write a python program to accept two numbers from a user and print the average of them. You can avoid the input validation until you read chapter8.

E3.2 Write a program to output the following using exactly one print statement.
Hello,
Reader!

E3.3 Print the following line. You can test it using single quotations or double-quotations.
The height of Andrew is 6'9"

E3.4 Given the following assignment:

```
x,y=10,2.5
```

Predict the output of the following Python statements:

```
print(x+y)
print(x/y)
print("Difference of x and y is:", x-y)
print("x*y=:"+ x*y)
print(x+ "is stored in x.")
```

E3.5 Predict the output of the following Python code segment:

```
x = 12
#print("x= "+12)
```

E3.6 Write a python program to calculate the area of a circle. Your program should accept the radius from a user. You can assume that the user provides valid inputs only.

Solution to Exercises

E3.1

```
#Solution to Assignment 3.1
#First user input
first_input=input("Enter first number:")
#Converting it to float
first_number=float(first_input)
#Second user input
second_input=input("Enter second number:")
#Converting it to float
second_number=float(second_input)
#Calculating the average
average=(first_number+second_number)/2
print("Average is :",average)
```

Output

```
Enter first number:25.2
Enter second number:75
Average is : 50.1
```

E3.2

```
#Solution to Assignment 3.2
print("Hello,\nReader!")
```

E3.3

```
print('The height of Andrew is 6\'9" ')
```

Alternative answer:

```
print("The height of Andrew is 6'9\"")
```

Output:

```
The height of Andrew is 6'9"
```

E3.4
The output for each of the following statements is shown with inline comments:

```
x,y=10,2.5
print(x+y) #12.5
print(x/y) #4.0
print("Difference of x and y is:", x-y) #7.5
#The following line of code will cause error: can only #concatenate str (not "float") to str
print("x*y=:"+ x*y)
# The following line of code will cause error: unsupported #operand #type(s) for +: 'int' and 'str'
print(x+ "is stored in x.")
```

E3.5

It will print nothing because the print statement is commented.

E3.6

```
#Get the radius from the user and converting it into a float
radius = float(input("Enter the radius of the circle:"))
#Area of a circle=(22/7)*r*r where r is the radius
area = (22/7)*radius*radius
print(f"The area of the circle is:{area} square unit.")
```

Output:

```
Enter the radius of the circle:7.7
The area of the circle is:186.34 square unit.
```

CS 1.1 Implementation

Here I use f-strings to print the seller’s name, address, etc. So, you see the following codes:

```
print(f"{seller_name}")
print(f"{seller_address}")
```

In this implementation, you do not need these f-strings. I could use hard-coded strings like:

```
print("Spencer Retail")
print("136, Garia Station Road,\n Kolkata:700084")
```

But the problem is that when you need to update the seller information, you need to reflect the change in every place. So, I suggest that you use variables in similar places for better maintenance. In that case, you can update the variable in one place and it will reflect the updated result in all locations.

Now go through the complete implementation.

```
# Seller information
seller_name = " Spencer Retail"
seller_address = " 136, Garia Station Road,\n Kolkata:700084"
seller_contact = "123-456-789"

#Decorating top segment
print("-" * 50)
print(f"{seller_name}")
print(f"{seller_address}")

print("-" * 50)
apricot_price = 300
dates_price = 400
almonds_price = 500
apri_dates_combo = (apricot_price + dates_price) * .9  # 10% discount
dates_almon_combo = (dates_price + almonds_price) * .9  # 10% discount
almon_apri_combo = (almonds_price + apricot_price) * .9  # 10% discount
gift_pack = (apricot_price + dates_price + almonds_price) * .75  # 25%
discount
print("Product(s) \tPrice (per pack)")
print(f"Apricot\t\t{apricot_price}")
print(f"Dates\t\t{dates_price}")
print(f"Almond\t\t{almonds_price}")
print(f"Combo-1\t\t{apri_dates_combo}")
print(f"Combo-2\t\t{dates_almon_combo}")
print(f"Combo-3\t\t{almon_apri_combo}")
print(f"GiftBox\t\t{gift_pack}")

# Decorating the bottom segment.
# It contains the contact information.
print("*" * 50)
print(f"For free delivery, contact {seller_contact} ")
print("*" * 50)
```

Case Study II

CS 2.1 Problem Statement

Will you like to attend a test? The test is simple. You need to predict the value of an expression in advance. You can attend this test as many times as you want, but there is a twist. The computer can show you a unique expression each time you play the game. Are you ready?

For a better understanding, I'm supplying some sample output for you:

Case-1: Wrong prediction

```
Predict the value of the following expression: 12*8%3
Enter your answer:5
Your answer is wrong.
The correct answer is:0
```

Case-2: Correct prediction

```
Predict the value of the following expression: 12+(2*3)/4
Enter your answer:13.5
Correct answer.
```

Author's Comment:

In this project, you can consider only one expression and change the data inside it. In that case, you can make a working solution at the end of Chapter 2. But I want you to make an improved solution. A better application can test you with different expressions using varying data. So, I show you this implementation at the end of Chapter 4.

Chapter 4: Decision Making

Decision making is an integral part of programming. You often need to examine certain conditions in a program. Based on those conditions, you control the flow of the program execution. This chapter teaches how to respond based on the state of your program.

We often perform conditional tests using various if -statements. For example, you may see a simple if statement, an if-else chain, or an if-elif-else chain in a program. The choice depends on how many conditions you want to test at a specific point. In the chapter2, you have seen the use of comparison operators. This chapter uses those operators to verify these conditions.

Using an if Statement

Let us begin with an easy program. Here I assume that I have a number variable. If its value is greater than 10, the program can tell that the current value of the variable is bigger than 10.

Demonstration 1

Let us create a new file and type the following into it:

```
a = 11
if a > 10:
    print("The current value inside a is greater than 10.")
```

Output

When you run this program, you’ll see the following output:

```
The current value inside a is greater than 10.
```

Using the **if-else** Statements

There is a potential drawback in the previous program: if a is less than or equal to 10, it will NOT print any output. (To test this, you can change the value of a to a value less than 10 and execute the program again.)

Demonstration 2

Let us make the program better. Now I add a few more lines to the previous demonstration. This time you see the use of an else statement. Go through the following code.

```
else:
    print("The current value of a is less than or equal to 10.")
```

Now you can get an output if the value of a is less than 10. To test this, let us make a=5 and run the following program:

```
a = 5
if a > 10:
    print("The current value inside a is greater than 10.")
else:
    print("The current value of a is less than or equal to 10.")
```

Output

This time you’re expected to see the following output:

```
The current value of a is less than or equal to 10.
```

Demonstration 3

Notice the code in the previous demonstration. You understand that inside the if-statement, I’m testing a condition. If the condition is true, then the indented lines following the if-statement will execute. Otherwise, the indented lines following the else statement will execute. So, you understand that the if-else chain helps you to check whether a certain condition is True or False.

Let us see another example. This time I check whether a value inside a variable is equal to the value of my interest. To make it clear, go through the following example:

```
a = 5
if a == 5:
   print("a is equal to 5.")
else:
   print("a is NOT equal to 5.")
```

Output

We use a single equal to (=) sign is to assign a value to a variable. For example, you can read the code: a=5 as “ Set the value 5 to the variable a ”. You have seen this kind of usage several times.

But we use double equal to (==) for the equality operator. It returns True if values of both sides of the operator match; otherwise, it returns False . So, when you run the previous program, you’ll get the following output:

```
a is equal to 5.
```

POINT TO REMEMBER

The online link: https://docs.python.org/release/3.0.1/whatsnew/3.0.html#changed-syntax says that True, False, and None are reserved words. In Python3.x programming, when you convert int to bool, the boolean value is True for the integers except 0. For better clarity, I show you a few more code fragments. I executed these in a Python3.8.3 command shell:

```
>>> int(False)
0
>>> int(True)
1
>>> bool(-3)
True
>>> bool(5)
True
>>> bool(0)
False
```

Demonstration 4

In the chapter2, you have seen that the != operator does exactly the opposite to the == . (The exclamation sign is for “not”; so, you pronounce “!=” as “Not equal to”. You may also see similar syntaxes in many other programming languages like Java, C#, etc.)

Now I make some minor changes. Here an if statement checks a “Not equal to” condition instead of an “Equal to” condition. Consider the following code segment:

```
a = 25
if a != 7:
    print("a is NOT equal to 7.")
else:
    print("a is equal to 7.")
```

Output

If you run this program now, you’ll get the following output:

```
a is NOT equal to 7.
```

Using elif with an if-else Statement

You have seen the uses of if , if-else , ==, and != . Let us fine-tune the previous programs now. This time you see the use of the elif statement.

It is very common when your program needs to check over two conditions. In such a case, using if-else statements is not enough. To understand this, consider another case. Let us assume instead of saying ‘ Greater than equal to ’ (or, less than equal to), you segregate the case ‘ Equal to’ . For example, let us suppose, in your application, the user supplies a number. Now your program should print whether the number is greater than 10, or less than 10, or equal to 10. The if-elif-else chain can handle such a scenario.

Demonstration 5

Let us type the following lines of code and execute the program. For simplicity, I’m skipping the validation of user input. I assume that he follows the instruction properly, and he does not supply any invalid data.

```
user_input = input("Enter a valid number only:")
# Skipping the validation of the user's input
a = float(user_input)
if a > 10:
    print("The number is greater than 10.")
elif a == 10:
    print("The number is equal to 10.")
else:
    print("The number is less than 10.")
```

Output

Here are some outputs for different user-supplied numbers.

Case1:
```
Enter a valid number only:15
The number is greater than 10.
```
Case2:
```
Enter a valid number only:6
The number is less than 10.
```
Case3:
```
Enter a valid number only:10
The number is equal to 10.
```
Case4:
```
Enter a valid number only:7.9
The number is less than 10.
```

POINTS TO REMEMBER

Python executes only one block from an if-else or if-elif-else chain. The control flows in the sequential order until one condition becomes True . You may see many indented lines of code under any of these blocks (if, elif, or else). When a condition becomes True , all the indented lines inside the block will execute.

You can also note: the else block is not mandatory. In demonstration1, you have seen that you can write a program with an if block only. In that program, else block was not present. Like that case, else block is also not mandatory at the end of an if-elif chain.

One question may come into your mind now: How to handle a scenario that considers more than 3 cases? This is also a common use case. For example, let’s assume that you are about to launch a product. Assume that you decide the following prices for different age groups:

- Kids up to age 10, can use the product for free.
- If the user is above 10 years but less than 20 years, he needs to pay only 1$.
- If the user is between the age of 20 years to less than 30 years, he needs to pay only 2$.
- If the user is between the age of 30 years to less than 40 years, he needs to pay only 3$.
- Users from 40 and above, need to pay 4$.

You can see that you need to check more than 3 conditions in this scenario. Let us see how can we handle this using an if-elif-else chain. Before you go forward, I give you a clue. In an if-elif-else chain, Python does not restrict you to use only one elif statement.

Demonstration 6

Let us type the following lines of code and execute the program. In the analysis section, you'll get an idea about how to improve this program. Like the previous cases, for the sake of simplicity, I'm skipping the validation of user inputs.

```
user_input = input("Enter your age:")
# Skipping the validation of user's input
age = float(user_input)
if age < 10:
    print("Hi Dear. You can use the product for free.")
elif age >= 10 and age <20:
    print("Please donate 1$ for the product.")
elif age >= 20 and age < 30:
    print("Please donate 2$ for the product.")
```

```
elif age >= 30 and age < 40:
    print("Please donate 3$ for the product.")
else:
    print("Please donate 4$ for the product.")
```

Output

Here is the output for different user-supplied numbers.

Case1:

```
Enter your age:9.2
Hi Dear. You can use the product for free.
```

Case2:

```
Enter your age:12.5
Please donate 1$ for the product.
```

Case3:

```
Enter your age:25
Please donate 2$ for the product.
```

Case4:

```
Enter your age:37
Please donate 3$ for the product.
```

Case5:

```
Enter your age :57.3
Please donate 4$ for the product.
```

Analysis

This program gives you an idea about how to test many conditions using the if-elif-else chain. Apart from this, there are several points to note in this program. The following bullet points list the important points:

- You can improve the following code. For example, you can replace the code:

```
elif age>= 10 and age <20:
    # other code
```

using the following code:

```
elif 10 <= age < 20:
    # other code
```

Notice that I’m simplifying the expression. You can do the same in similar places inside the program.

- For better readability, you can use brackets. Here is a sample:

```
elif (age >= 20) and (age < 30):
    print("Please donate 2$ for the product.")
```

I’ve not used brackets in Demonstration 6 because it was NOT mandatory for me.

- Note that I have used only one statement inside an if block, elif block, or an else block. But it is very common when you see many print statements in any of these blocks. For example, the following block of code is perfectly fine:

```
if age < 10:
    print("Hi Dear. You can use the product for free.")
     print("Thank you for choosing the product.")
     //remaining code
```

Tautology and Contradictions

You should be careful about your logic. Sometimes the result of the compound expressions is always False . For example, if you write:

```
if a < 5 and a > 7:  # This condition cannot be satisfied.
    print("The if condition is satisfied.")
```

Think now. Is it possible that a variable is less than 5 but greater than 7 at the same time? The answer is no. We term these as contradictions . We term the opposite scenario as a tautology, where the resultant value of the compound expressions is always True .

Alternative Designs

Sometimes you may see alternative designs in a similar context. I do not recommend these practices, but I want you to note that Python does not complain if you follow them. Let us look into them.

Demonstration 7

In demonstration5, you saw the use of if-elif-else chain. Here you see a program that uses the if-else chain. Write the following program and execute it.

```
user_input = input("Enter a valid number only:")
# For simplicity, skipping the validation of user's input
a = float(user_input)
if a > 10:
    print("The number is greater than 10.")
else:
    if a == 10:
        print("The number is equal to 10.")
    else:
        print("The number is less than 10.")
```

Output

You can test this program against the same inputs that I used in demonstration5. Here, you’ll receive the same output. I do not repeat them here.

Analysis

Have you noticed that in this approach, if you need to check more conditions, the texts are drifted to the right? For example, see the following program. It is another valid program with more conditions. But I continue to use the if-else chain instead of the if-elif-else chain.

```
user_input = input("Enter a valid numbers only:")
# For simplicity, skipping the validation of user's input
a = float(user_input)
if a < 0:
    print("The number is less than 0.")
else:
    if a < 1:
        print("The number is greater than 0 but less than 1.")
    else:
        if a == 1:
            print("The number is equal to 1.")
        else:
            if a == 2:
                print("The number is equal to 2.")
            else:
                if a == 3:
                    print("The number is equal to 3.")
                else:
                    print("The number is greater than 3.")
```

You can see that as the number of conditions grows, the texts are drifting to the right side. This gives you the idea about: why do I prefer an if-elif-else chain in this case. For me, it is a better option and more manageable.

The previous code segment shows you can write nested if (or else) statements too.

Demonstration 8

Let us explore another way of using simple if statements to check a condition. Again, I do not recommend this. I include this for one reason. It should not surprise you when you see similar constructs in other people's code.

In this program, I divide a number by another number. You know that in this case, the divisor should not be zero. So, you can write a simple program like the following:

```
a = input("Enter a valid integer for the dividend:")
b = input("Enter a valid integer for the divisor:")
# Skipping the input validation...
dividend = int(a)
divisor = int(b)
if divisor != 0:
    result = dividend / divisor
    print(f"Division successful.The result is:{result}")
else:
    print("You cannot proceed when the divisor is 0.")
```

This program works for you and it is easily readable. But now I write an alternative version of the previous code. We refer to this as inline-if . You can summarize the concept as:

Do task-1 if the condition is True else do task-2

Here is an example for you:

```
# Assign b before the following statement
a = 5 if b != 0 else 1
print(a)
```

The previous code segment tells that we assign the value of a to 5 if b is not equal to 0. Otherwise, we assign the value 1 to a . So, using inline-if , you can write an alternative version of the program. Notice the changes in bold:

```
a = input("Enter a valid integer for the dividend:")
b = input("Enter a valid integer for the divisor:")
# Skipping the input validation...
dividend = int(a)
divisor = int(b)

# Alternative version
result= 0 # some initial value
result = dividend / divisor if divisor != 0 else print("You cannot set the divisor to 0.")
print(f"The result is:{result}")
```

Output

Here is the output for some valid inputs.

Case1:

```
Enter a valid integer for the dividend:24
Enter a valid integer for the divisor:3
The final result is: 8.0
```

Case2:

```
Enter a valid integer for the dividend:24
Enter a valid integer for the divisor:0
You cannot set the divisor to 0.
The result is: None
```

EXERCISE

E4.1 Is there any error in the following code?

```
#Exercise-4.1
a = 5
if a > 10:
    print("The number is greater than 10.")
elif a == 10:
    print("The number is equal to 10.")
```

E4.2 Predict the output.

```
#Exercise-4.2
a = 5
if a < 7 and a > 9:
        print("The condition is satisfied.")
else:
    print("The condition will never be satisfied.")
```

E4.3 Predict the output.

```
#Exercise-4.3
number = 0
#number = -34
if number:
    print(f"The condition is satisfied.number={number}")
else:
    print(f"The condition is not satisfied.number={number}")
```

E4.4 Give an example of a tautology.

E4.5 Give an example of a contradiction.

Solution to Exercises

E4.1

No, there is no problem because it is not mandatory to place an else block at the end of an if-elif chain. But in this case, you will not receive any output. It is because neither the condition of the if block nor the condition of the elif block is satisfied.

E4.2

The condition will never be satisfied.

Explanation:
It is an example of a contradiction. In other words, the result of the compound expressions is always False in this case.

In this program, if you replace and with or , then the if condition can be satisfied.

E4.3

The condition is not satisfied.number= 0

Explanation:
In Python3.x programming, when you convert int to bool, except 0, Python evaluates all to True . I have already shown you an example in the chapter. You can refer to that discussion if needed.

E4.4

Let us say my_number is a number variable. So, the following expression: my_number>0 or my_number<=0 is always true. So, it is an example of tautology.

E4.5

See Exercise 4.2.

CS 2.1 Implementation

I start with three unique expressions in this project. When you run this application, you get a unique expression in each run. For example, in the sample output you can see that in one run, you get an expression 12*8%3. But in a different run, you get a new expression: 12+(2*3)/4 . To give it more dynamic behavior, I change the data in a particular type of expression too. For example, in one run, you may see the expression: 12*8%3 , but in a different run, you may see 11*8%3. It is because I replace some data in these initial expressions with random data in each run. To do this, I import the randint() function. The randint(a,b) can return a random integer in the range [a, b] including both endpoints. I talked about this function in Chapter 3.

The following code segment with supporting comments can explain it:

```
x = randint(10, 12)
question1 = "2*3-4"
# Replacing 2 in question1 with x
question1 = question1.replace("2", str(x))
question2 = "1+(2*3)/4"
# Replacing 1 in question2 with x
question2 = question2.replace("1", str(x))
question3 = "5*8%3"
# Replacing 5 in question3 with x
question3 = question3.replace("5", str(x))
```

I use another important function eval() in this implementation. This function has 3 parameters, the first one is mandatory and the last two are optional. I do not need these optional parameters now. We can pass a string to eval() that can evaluate it as a Python expression. To show how it works, I present you following code segment:

```
>>> eval("1+2+3")
6
>>> eval("2*5+2")
12
```

Now you are ready to go through the complete implementation. Here is the complete code:

```
from random import randint

x = randint(10, 12)
#First question with initial data
question1 = "2*3-4"
# Replacing 2 in question1 with x
question1 = question1.replace("2", str(x))

#Second question with initial data
question2 = "1+(2*3)/4"
# Replacing 1 in question2 with x
question2 = question2.replace("1", str(x))

#Third question with initial data
question3 = "5*8%3"
# Replacing 5 in question3 with x
question3 = question3.replace("5", str(x))

# The randint(1,3) function returns a number
# between 1 and 3 (both included)
pick_question = randint(1,3)
if pick_question == 1:
    quiz = question1
elif pick_question == 2:
    quiz = question2
```

```
else:
    quiz = question3

print(f"Predict the value of the following expression:{quiz}")
# User's input
user_input = input("Enter your answer:")
# Converting it to float
predicted_value = float(user_input)
actual_value = eval(quiz)
if predicted_value == actual_value:
    print("Correct answer.")
else:
    print("Your answer is wrong.")
    print(f"The correct answer is:{actual_value}")
```

Case Study III

CS 3.1 Problem Statement
Will you like to play a game? It is simple but interesting. The computer will pick a number between 1 to 15 for you. You need to guess the number. But here is the challenge. To win the game, you need to make a correct guess within three attempts. Note another point: Each time you play the game, the computer can pick a different number for you. Are you ready?

For a better understanding, I show you some sample output:

Case-1: You have lost the game.

```
The computer has picked a random number for you.
Clue: It is between 1 and 15 (both inclusive).
Can you guess it within 3 attempts? Make a try.
Enter your answer:5
It is high. Try again!
Enter your answer:3
It is high. Try again!
Enter your answer:1
It is low. Try again!
The computer picked the number:2.
You have lost the game now!
```

Case-2: You have won the game.

```
The computer has picked a random number for you.
Clue: It is between 1 and 15 (both inclusive).
Can you guess it within 3 attempts? Make a try.
Enter your answer:10
It is low. Try again!
Enter your answer:14
Excellent. You have guessed it right in 2 attempt(s).
```

Author’s Comment:

For a simple illustration, the computer picks the number between 1 and 15. The game will be more challenging when it picks the number from a broader range. Similarly, it becomes easy, if it picks the number from a narrow range. Once you see the solution, I encourage you to vary the range through user input.

Chapter 5: Iteration using Loops

In programming, iteration is used for repetition. When you write a program or implement an algorithm, you may need to iterate over a certain part of the code until a specific condition meets. Loop statements help you iterate over those portions of the code. In this chapter, you will learn about various kinds of loop statements. You can use them with conditional statements to make smart programs.

The Purpose of Iteration

The first question that may come into your mind is: why do I need loop statements (or, iterations)? To understand the answer, let's write a simple program that can print 1, 2, and 3 in separate lines.

```
print("Printing 1 to 3:")
print(1)
print(2)
print(3)
```

When you execute this code, you see the following output:

```
Printing 1 to 3:
1
2
3
```

It is a simple program, and this output is obvious. You should not have any problem to understand this program. Now I give you a task: I want you to write a program which can print up to 5000 or 50000 . If you follow the previous approach, -how many print statements are required? Can you imagine this? Even if you use a simple copy/paste technique, there is a substantial amount of work. This is an impractical and tedious approach. Is there any alternative? Yes, you can exercise a better approach. Loop statements are ready to help you in similar situations.

There are different loop statements. You can use them at your convenience. For now, let’s start with while loops.

The while Loop

The while loop continues executing as long as a certain condition is true. It has the following form:

```
while “Your specified condition(s)”:
        statement-1
        statement-2
        statement-3
        ….
```

If you look closely at the previous syntax, you will notice the following points:

- It starts with a reserved keyword while
- There is a colon (:) after the condition(s) you mentioned.
- There can be many statements inside the while loop. You indent these lines.
- You check the condition before you enter this block of statements. If the condition is true, the control can enter the block and execute these statements. You repeat this block of statements till the condition is true.
- In short, to come out from a while loop, the condition needs to be false. Otherwise, the block of statements inside the loop will continue executing.
- When the condition becomes false, the block of statements inside while loop will no longer execute. Then we say that the control has exited from the while loop. It also means that if the condition is false at the beginning, you cannot enter the body of a while loop at all.
- If you enter a loop but do not satisfy the exit criteria, you fall into the trap of an infinite loop.

- Notice the body of the while loop. You can see that those are following correct indentation (same number of spaces from left). This correct indentation is important. Indentations tell you which statements are inside the while loop, and which are not.

In the following example, I’m introducing a variable called current_number . In the beginning, I set its value to 1. Then I introduce a while statement. In this loop, I check a condition. The condition says that if the current_number is less than or equal to 10, I'll enter the loop. Then I print the current value of this variable, increment the value of the variable by 1, and repeat this process. The following flowchart can depict the scenario (See Figure 5-1):

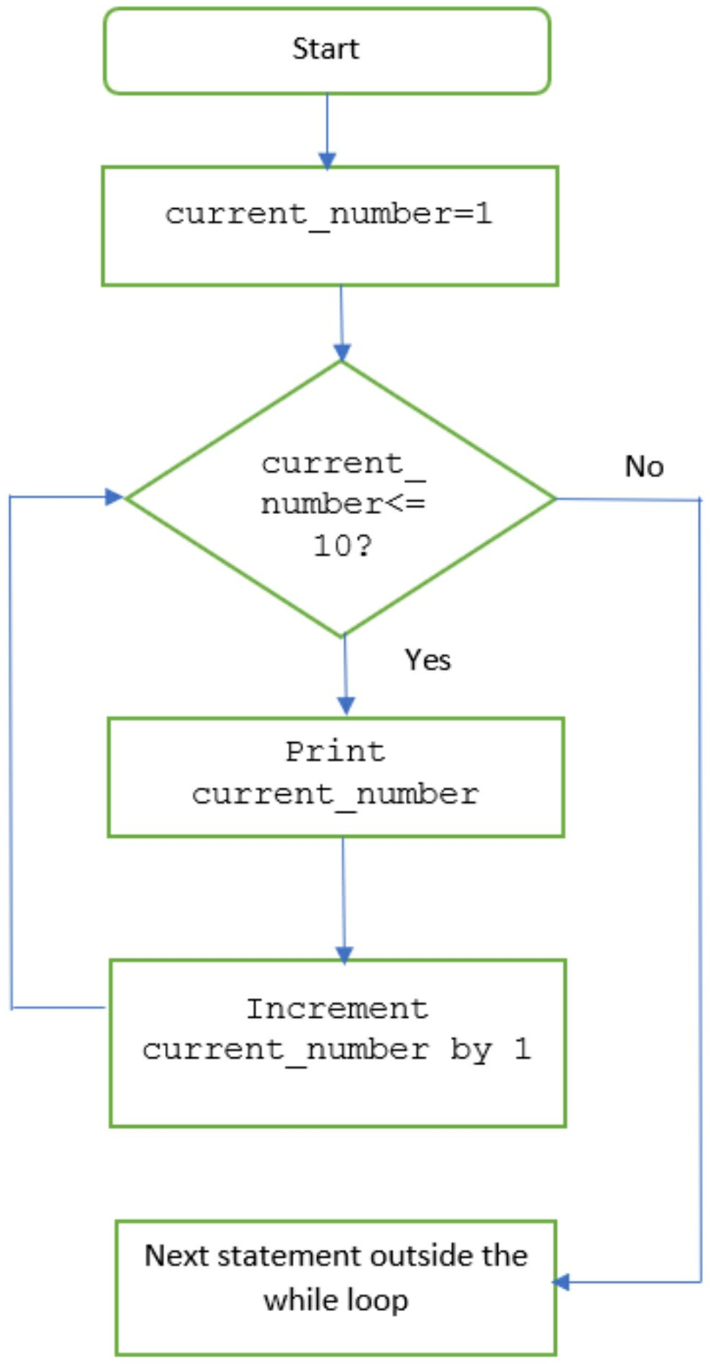


Figure 5-1: Flowchart of a while loop.

Arbitrarily, I have chosen the number 10. But you can choose any number you want. Based on my conditional criteria, I can print 1 to 10 using this while loop. To make the program short and simple, I have used only one print statement outside the while loop. Let us follow demonstration

1 now. I’ve kept many comments for your easy understanding.

Demonstration 1

I have created the file while_loop_ex_1.py . It has the following content:

```
# Printing 1 to 10 using while loop
print("Printing 1 to 10 using while loop:")
current_value = 1
while current_value <= 10:
    print(current_value)
    # incrementing the value
    # Following line is a shortcut for :
    # current_value=Current_value+1
    current_value += 1
# This statement is placed outside the while loop
print("Job has been done. Exit from the while loop.")
```

Output

When you run this program, you’ll see the following output:

```
Printing 1 to 10 using while loop:
1
2
3
4
5
6
7
8
9
10
Job has been done. Exit from the while loop.
```

I have already mentioned that I have chosen the number 10 at random. So, if you want a print up to 500 or 5000 etc., you can change the condition inside while loop. For example, you can replace 10 with your intended value.

Before you move into the next section, let me show you a program. This program causes an infinite loop. I’ve made demonstration2 for this purpose. It is as follows:

Demonstration 2

Here is the complete program.

```
# An incorrect implementation of while loop.
# It creates an infinite loop.
i = 0
while i != 1:
    print(f"I cannot come out from the loop.i={i}.")
```

Output

When you run this program, you’ll see the program falls into the infinite loop. So, it keeps printing the following lines:

```
I cannot come out from the loop.i=0.
I cannot come out from the loop.i=0.
I cannot come out from the loop.i=0.
I cannot come out from the loop.i=0.
I cannot come out from the loop.i=0.
…………
…………
```

Analysis

I hope that you understand the problem. It is because you have not implemented the proper logic. You need to change the value of i that was set to 0. But the control inside the while loop cannot come out unless you make i=1 . Therefore, you need to be careful enough when you implement your logic using a while loop.

The for Loop

Using while loops are common in programming. Notice the construction and usage of the while loop again. You can see that there are three major parts which are as follows:

- Initialization- You use it before the control enters the while loop
- Condition- You use this to determine whether the control enters the loop. If you enter the loop, the condition is further checked to determine whether the control will be inside the loop, or it will exit from the loop.
- Update-You have to implement the proper logic to update your intended variable (which you use in the condition of the while loop). This helps you to come out from the loop and process the next statement if any. If the control cannot exit from the loop, you fall into an infinite loop. You have seen an example of an infinite loop in Demonstration2.

There is another loop called for loop in Python. In certain situations, this loop provides a convenient way to express the steps of a while loop. To understand it better, I make an equivalent program of Demonstration1. This program shows you the use of a for loop.

Demonstration 3

Here is the complete program.

```
# Printing 1 to 10 using for loop
print("Printing 1 to 10 using a 'for' loop:")
# I use the range(1,11) function to print
# values 1 to 11-1=10
for current_value in range(1, 11):
    print(current_value)
# This statement is placed outside the for loop
print("Job has been done. Exit from the 'for' loop.")
```

Output

If you run this program, you get the following output:

```
Printing 1 to 10 using a 'for' loop:
1
2
3
4
5
6
7
8
9
10
Job has been done. Exit from the 'for' loop.
```

Analysis

You can see that demonstration 3 and demonstration 1 generate the equivalent output. This time I have replaced the word ‘while’ with the word ‘for’ in the print statement.

You have seen an example of a for loop in demonstration 3. Now I show you a few more use cases using this kind of loop. Let us continue reading.

You often use the for loops when you traverse an iterable element like list, tuple, strings, etc. What is an iterable element? In simple terms, an iterable element is such an element that can be looped over. The general syntax to traverse and print the content of an iterable element is as follows:

```
for element in iterable_element:
    print(element)
```

In the next chapter, you’ll see the list data type in detail. To understand the upcoming example, it is enough for you to know that a list can store a sequence of elements. These elements may come from the same data type or different data types. You can define a list something like:

```
list_name=[value1,value2,value3,..]
```

Now assume you have a list of employee names as follows:

```
employee_names = ["Sam", "Bob", "George", "Kate", "Julie"]
```

Here, to print the employee names, you can use the following code segment:

```
for employee in employee_names:
    print(employee)
```

POINTS TO REMEMBER

You may be new to programming and unaware of this format. So, you can pick each name from this list one-by-one and then print those names. It’s ok for a small set of names. But consider a scenario where your list contains many names, or when the list changes very often. In these cases, to access the list elements, the for loop can provide the best potential support to you.

I have used the word *‘iterable’* previously. When you’ll learn more, you’ll know that an iterable object has a method called iter() . You can use it to get an iterator. For example, for the following list:

```
employees = ["Sam", "Bob", "George"]
```

I can print the values inside the list using the iter() and next() methods. To show this, I am using a Python command shell now. Look into these code fragments:

```
>>> employees = ["Sam", "Bob", "George"]
>>> iterator=iter(employees)
>>> next(iterator)
'Sam'
>>> next(iterator)
'Bob'
>>> next(iterator)
'George'
```

Demonstration4 provides a sample program for you and it is as follows:

Demonstration 4

In this program, the first time when you pass through the for loop, the first element of the list, “Sam” is assigned to the variable employee . Next time, the second element in the list, “Bob” is assigned to the variable employee . And it continues like this until you reach the end of the list. Let us go through the complete program.

```
# Printing employee names using for loop
employee_names = ["Sam", "Bob", "George", "Kate", "Julie"]
print("Printing employee names using a 'for' loop:")
for employee in employee_names:
    print(employee)
```

Output

Run this program and see the following output.

```
Printing employee names using a 'for' loop:
Sam
Bob
George
Kate
Julie
```

Analysis

I want you to note that in demonstration 4; I have written:

for **employee** in employee_names:
    print(employee)

I have chosen the variable name as an employee . You are free to choose any valid name. For example, the following code can produce the same output:

```
for i in employee_names:
    print(i)
```

But you’ll probably agree with me that ‘ employee’ is a better choice than ‘i’ in this example.

POINT TO REMEMBER

We can summarize the difference between a for loop and a while loop as follows: A while loop runs as long as the specified condition is true. In contrast, a for loop is suitable for a collection of items where you want to execute a block of code for each item in the collection.

In previous demonstrations, I have shown you the use of the range() function. The general form of this function is:

```
range(begin, end, step)
```

where

- You use the begin parameter for the first value. The default value is 0.
- You use the end parameter for the one past the last value
- You use the step parameter to denote the increment or decrement value.

You can use the step parameter wisely. For example, the following code segment can print 0,3,6, and 9 .

```
# Printing 0,3,6,9 using for loop
for current_value in range(0, 11, 3):
    print(current_value)
```

Notice that this code segment prints the values in the following order:

```
0
0+3=3
3+3=6
6+3=9
```

The next value 9+3=12 is beyond the limit/range (11-1=10). So, you do not see 12 in this output.

You can use the for loop to print these values from the reverse direction. Here is the code segment for your reference:

```
# Printing 0,3,6,9 using for loop
```

```
for current_value in range(9, -1, -3):
    print(current_value)
```

Notice that this code segment prints the values in the following order:

```
9
9-3=6
6-3=3
3-3=0
```

The next value 0-3=-3 is beyond the limit/range. So, you do not see -3 in this output.

Use of break Statement

You need to exit from a loop based on a certain condition(s). For example, suppose you have the following program:

```
current_value = 1
while current_value <= 5:
    print(current_value)
    current_value += 1
```

You get the following output when you execute the program:

```
1
2
3
4
5
```

But let’s assume you do not want to go to the end of the loop. You decide that you’ll exit from this loop when the current_value becomes 3.

Let me change the while block now. Now I insert another if block inside it. This if block has a break statement (break is a keyword in Python) and the changed version is:

```
while current_value <= 5:
    print(current_value)
    if (current_value == 3):
        print("Exit the loop when currnet_value becomes 3.")
        break
    # incrementing the value
    current_value += 1
```

If you execute this segment of code, you see the following output:

```
1
2
3
Exit the loop when current_value becomes 3.
```

You can see that a break statement allows the loop to terminate prematurely. Now go through the following demonstration. It contains more comments for your easy understanding.

Demonstration 5

Here is the complete program.

```
print("***Using a break statement inside a while loop.***" )
current_value = 1
print(f"Initially, current_value={current_value}”)
print("Exit the loop when current_value becomes 3.")
while current_value < 5:
    # incrementing the value
    current_value += 1
    print(f"The current value={current_value}")
    if current_value == 3:
        break
    print("I'm still inside the while loop.")
# This statement is placed outside the while loop
print("Job has been done. Exit from the while loop.")
```

Output

When you run this program, you’ll see the following as output:

```
***Using a break statement inside a while loop.***
Initially, current_value=1
Exit the loop when current_value becomes 3.
The current value=2
I'm still inside the while loop.
The current value=3
Job has been done. Exit from the while loop.
```

Analysis

Notice that the program does not print the statement "I'm still inside the while loop." when the current_value becomes 3. It is because I place this statement after the break statement. Therefore, when a break statement executes, the control jumps out of the loop.

POINT TO NOTE

You may wonder why do I complicate this thing? You may suggest if I need to exit at 3, why should I continue till 5? So, I could change the condition of the while loop as follows:

```
while current_value <= 3:
#rest of the code
```

Wait! This is a simple demo and the need to exit from the loop is not important. But consider a case when you generate some random numbers inside a loop. Here, you are not sure about the number in advance. Also, you may design the application in such a way that for a particular random number inside a loop, you do not process further and you exit from the loop. In cases like this, a break statement is useful.

Use of continue Statement

You have seen the use of a break statement. There is another interesting keyword in Python called continue . Using this keyword, you can skip some statements inside a loop in an iteration. You place these statements after the keyword continue .

There is a reason for this. When a specific condition is met, you may not want to exit entirely from a loop. Instead, in this iteration, you may want to skip the remaining statements of the loop and go back to the beginning of the loop. The continue keyword is made for this purpose. To understand the usage better, I use a similar example which you see in Demonstration5. The only difference is: this time, I use continue instead of break . I also make some meaningful changes to print the messages.

See the output and follow the analysis section. These help you understand the working mechanism easily. Let us follow demonstration 6.

Demonstration 6

Here is the complete program.

```
print("***Using a continue statement inside a while loop.***")
current_value = 1
print(f"Initially, current_value={current_value}")
print("I skip the code when current_value becomes 3.")
while current_value < 5:
    # incrementing the value
    current_value += 1
    print(f"The current value={current_value}")
    if current_value == 3:
        continue
    print("I'm still inside the while loop.")
# This statement is placed outside the while loop
print("Job has been done. Exit from the while loop.")
```

Output

When you run this program, you’ll see the following output:

```
***Using a continue statement inside a while loop.***
Initially, current_value=1
I skip the code when current_value becomes 3.
The current value=2
```

```
I'm still inside the while loop.
The current value=3
The current value=4
I'm still inside the while loop.
The current value=5
I'm still inside the while loop.
Job has been done. Exit from the while loop.
```

Analysis

Notice that the program does not print the statement "I'm still inside the while loop." when current_value becomes 3. It is because I place this statement after the continue statement. In short, when the program executes the continue statement, it skips the remaining part of the loop for this iteration.

Notice that you can see the print statement in other iterations. The loop also continues even after the continue statement. But in case of a break , the control jumps out from the loop. You have seen that behavior in demonstration 5.

POINTS TO NOTE

The concept of while loop, break, and continue statements are very similar when you program with C, C++, Java, or C#.

Nested Loop

Now I show you an example of a nested loop. In the previous chapter, you have seen that a program can contain an if statement inside another if statement. We call it a nested if statement. Similarly, you can have nested loops, which means you can have a loop that is contained in another loop. It is useful when you want to repeat an iterative process.

Demonstration 7

The following is an example that has two for loops. For each iteration of the “outer for loop”, the “inner for loop” execute completely. See the following demonstration.

```
print("*** Nested loop example.***")
colors = ["Red","Green","Yellow"]
fruits = ["Mango","Banana"]
for color in colors:
    for fruit in fruits:
        print(color,fruit,end="     ")
    print()
```

Output

Here is the output:

```
*** Nested loop example.***
Red Mango     Red Banana
Green Mango     Green Banana
Yellow Mango     Yellow Banana
```

Analysis

You can see that I use a parameter called end inside the print() function. It becomes available in Python3. By default, the parameter is ‘ \n’ which is a newline character. You can pass any other character or string to it. For example, you can see that I pass some spaces in the previous code segment. I use this function in Chapter 9 as well.

EXERCISE

E5.1 Assume that there is an application where users supply some valid numbers only. It is because you do not need to write extra code to validate the input. Now write a python program where a user can keep supplying the numbers, and for each input, it can print whether it is a positive number or not. The program will end when the user types quit.

E5.2 Create a list of numbers including integers and floats. Make sure that you have at least 5 numbers.

a) Using a for loop, print the content of this list.

b) Print the elements from index position 2 to 5.

E5.3 Predict the output of the following code segment:

```
for i in range(10, 20, 5):
    print(i)
```

E5.4 Predict the output of the following code segment:

```
for i in range(20,9,-5):
    print(i)
```

E5.5 How many asterisks do you expect to see from the following code segment?

```
for i in range(20,9,-5):
    print(i)
```

Solution to Exercises

E5.1

```
# Solution to Assignment 5.1
# Initially flag contains an empty string.
# We need a string other than quit to enter
# into the while loop to proceed further.
flag = ""
while flag != 'quit':
    user_input = input("Enter a valid number(Type quit to end the program): ")
    # If the user does not type 'quit'
    # we can convert the valid user input to float
    if user_input != 'quit':
        user_input = float(user_input)
    else:
        flag = 'quit'
        break; # exit from the loop
    if user_input > 0:
        print("You have supplied is a positive number.")
    elif user_input < 0:
        print("The supplied number is a negative number.")
    else:
        print("You've entered 0.")

print("Thank you. It is the end of the program.")
```

Here is a sample output:

```
Enter a valid number(Type quit to end the program): 12
You have supplied is a positive number.
Enter a valid number(Type quit to end the program): -35
The supplied number is a negative number.
Enter a valid number(Type quit to end the program): 12.5
You have supplied is a positive number.
Enter a valid number(Type quit to end the program): quit
```

```
Thank you. It is the end of the program.
```

Author’s Comment:
You may think: Can I use a number (an integer, or a float) flag to exit from the program? I suggest you not to do so. It is because, in that case, your program can suffer from a potential drawback. When you’ll learn about anti-patterns, you’ll know that there is an anti-pattern called **Zero Means Null** . The proposed program can suffer from it if you use a number flag. For example, let’s assume that when a user press -999, you’ll end the program. But if you treat -999 to exit from the loop, what will you do if you need this number? So, you cannot treat -999 like any other number now. Therefore, experts suggest that we choose the exit criterion wisely.

E5.2

```
# Solution to Assignment 5.2
# A list of numbers
number_list = [1, 2.2, 4.4, 5, 6, 9.3, 102]
print("Printing the numbers inside number_list using a 'for' loop:")
for number in number_list:
    print(number)

print("Now printing the numbers which have the index between 2 and 5:")
for index in range(2, 6):
    print(number_list[index])
```

Output:
Here is the output:

```
Printing the numbers inside number_list using a 'for' loop:
1
2.2
4.4
5
6
9.3
102
Now printing the numbers which have the index between 2 and 5:
4.4
```

```
5
6
9.3
```

E5.3

```
10
15
```

Author’s Comment:
The next value 15+5=20 falls outside the upper limit (20-1=19)

E5.4

```
20
15
10
```

E5.5

```
4
```

Author’s Comment:
These four asterisks appear due to the ‘ i’ values: 10, 5, 0 and -5

CS 3.1 Implementation

I make this implementation using the randint() function, if-elif-else chain, and while loop. You have seen randint() in Chapter 3, if-elif-else chain in Chapter 4, and while loop in Chapter 5. Refer to the supporting comments and strings that I use in print() functions. These help you understand the code. Here is the complete implementation for you.

```
from random import randint

picked_number = randint(1, 15)
print("The computer has picked a random number for you.")
print("Clue: It is between 1 and 15 (both inclusive).")
print("Can you guess it within 3 attempts? Make a try.")
```

```
no_of_attempt = 0
guess = False
user_input = 0  # an initial value
while no_of_attempt < 3:
    #User's input
    user_input = int(input("Enter your answer:"))
    no_of_attempt += 1
    if user_input == picked_number:
        guess = True
        break
    elif user_input > picked_number:
        print("It is high. Try again!")
    else:
        print("It is low. Try again!")
if guess:
    print("Excellent. You have guessed it right.")
    print(f"you've taken {no_of_attempt} attempt(s).")
else:
    print(f"The computer picked the number:{picked_number}.")
    print("You have lost the game now!")
```

Case Study IV

CS 4.1 Problem Statement

Let us design a multiple-choice question bank. Each question has four options. The user needs to pick the correct answer among these options. Once the test is over, he can see the score.

For a better understanding, I supply some sample output for you:

```
Welcome to the MCQ test.
==========================
Q1.What is the value of the expression:2*3-4?
(a)1
(b)2
(c)3
(d)None.
Type your answer(a/b/c/d):b

Q2.What is the value of the expression:1+(2*3)/4?
(a)1.5
(b)3
(c)3.5
(d)None.
Type your answer(a/b/c/d):d

Q3.The set data type can hold duplicate values. The statement is:
(a)False
(b)True
(c)Partially correct.
(d)None.
Type your answer(a/b/c/d):a

Your Score:2 out of 3
```

Author’s Comment:

Once you get the idea, you can add more questions to this test. The test is more challenging when you do not pick a fixed set of questions. Instead, you can select a specific number of questions at random from an extensive set of questions. At this stage, you can work with a fixed number of questions only.

Chapter 6: Advanced Data Types

This chapter shows you the use of some advanced data types which are very common in Python programming. We'll cover them quickly in the upcoming sections.

Working with Lists

Lists are used to store a sequence of elements. In a list, you can have just a few items or millions of items. Here you may see the presence of mixed data types too. Formally, you can say a list can contain a sequence of objects which may come from different data types.

You define a list something like the following:

```
list_name=[value1,value2,value3,..]
```

You can see that I am separating the values inside the square bracket [ ]. This kind of arrangement can help you to store multiple values into a single variable. For example, in the following code segment, I store three different names(strings) inside a list, called mylist1. Then I print the contents of the list.

```
# A list with three strings
my_list1 = ["John", "Bob", "Sam"]
print("The my_list1 is as follows:")
print(my_list1)
```

When you run this code segment, you can get the following output:

```
The my_list1 is as follows:
['John', 'Bob', 'Sam']
```

Similarly, you can have another code segment, where you use another list, called my_list2 to hold the value of 5 numbers and print the contents as follows:

```
# A list with five numbers
```

```
my_list2 = [1, 2, 3.7, 4, 5.2]
print("The my_list2 is as follows:")
print(my_list2)
```

When you run this code segment, you can get the following output:

```
The my_list2 is as follows:
[1, 2, 3.7, 4, 5.2]
```

You may notice that I use the list names as my_list1,my_list2, etc. In real-world programming, you may see that developers use “plural names” such as names, numbers, etc. It is because a list usually contains multiple elements.

You can also declare an empty list and later add elements using a special function called append(). But wait! Before you read further, let me quickly show some of the common use cases of lists. I’m keeping the comments for your easier understanding.

Lastly, like the previous chapter, instead of creating separate files for each of these small code segments, I have put them into a single file. But it’s a choice. So, if you want, you can always write a new program in a new file and execute it.

Now go through the following code fragments:

Goal:
To show you that a list can store mixed data types.

Code:

```
# A list can store mixed data types
my_list = ["John", 12, "Sam", True, 50.7]
print(my_list)
print("----------------")
```

Expected Output:

```
['John', 12, 'Sam', True, 50.7]
```

Goal:
Printing the elements of a list using an index. The indexing starts with 0 from the extreme left.

Code:

```
# You can use a list index. The usage is similar to strings.
my_list = ["John", 12, "Sam", True, 50.7]
print(my_list[0])  # John
print(my_list[1])  # 12
print(my_list[2])  # Sam
print(my_list[3])  # True
print(my_list[4])  # 50.7
# Error: List index out of range
# print(my_list[5])
```

Expected Output:

```
John
12
Sam
True
50.7
```

Explanation:
Notice that the line print(my_list[5]) is commented. There is currently no element at index position 5 in my_list . So, you'll receive an error if you try to use mylist[5] now. This error is similar to the following:

```
Traceback (most recent call last):
File
"E:/MyPrograms/Python/PythonCrashCourse/Chapter4/list_usages_file_1.py"
line 14, in <module>
    print(mylist[5])
IndexError: list index out of range
```

Note: I am placing several code segments in the same file. So, in the previous segment, line number 14 indicates the error location in my file, which may differ in your case. The same comment applies to similar error messages in this book.

Goal:
Printing the elements of a list from the extreme right. In this case, indexing starts from -1.

Code:

```
# You can use list indexing from the right end.
# In this case, it starts from -1
my_list = ["John", 12, "Sam", True, 50.7]
print(my_list[-1])  # 50.7
print(my_list[-2])  # True
print(my_list[-3])  # Sam
print(my_list[-4])  # 12
print(my_list[-5])  # John
# Error: List index out of range
# print(my_list[-6])
```

Expected Output:

```
50.7
True
Sam
12
John
```

Explanation:
Notice that the line print(mylist[-6]) is commented in the code segment. There is currently no element at index position -6 in my_list . So, you'll receive an error if you try to use mylist[-6] now. Similar to the previous case, the error may appear as follows:

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/Chapter4/list_usages_file_1.py"
line 25, in <module>
```

```
    print(mylist[-6])
IndexError: list index out of range
```

Goal:
I want to print only a specific portion of the list. Refer to the supportive comments.

Code:

```
# Printing list elements starting from a particular position
my_list = ["John", 12, "Sam", True, 50.7]
print("The original list is as follows:")
print(my_list)
print("Printing the elements starting from index position 2 to end:")
print(my_list[2:])
print("Printing the elements starting from index position 1 to 3(i.e.4-1):")
print(my_list[1:4])
```

Expected Output:

```
The original list is as follows:
['John', 12, 'Sam', True, 50.7]
Printing the elements starting from index position 2 to end:
['Sam', True, 50.7]
Printing the elements starting from index position 1 to 3(i.e.4-1):
[12, 'Sam', True]
```

Explanation:
When you use mylist[2:], you consider elements starting from index 2 to the end of the list. When you use mylist[1:4], you consider elements starting from index 1 to index 4-1 , i.e. 3 .

Goal:
You can reassign a new value in the list. Here is an example.

Code:

```
# You can reassign a new value inside the list
my_list = ["John", 12, "Sam", True, 50.7]
print("The original list is as follows:")
print(my_list)
print("Changing the element at index 2.")
my_list[2] = "Bob"
print("Now the list is as follows:")
print(my_list)
```

Expected Output:

```
The original list is as follows:
['John', 12, 'Sam', True, 50.7]
Changing the element at index 2.
Now the list is as follows:
['John', 12, 'Bob', True, 50.7]
```

Explanation:
You can see that I've set a new value at index position 2. So, instead of 'Sam' , you see 'Bob' inside this modified list.

Goal:
I want to concatenate multiple strings. There are various ways to accomplish this task. The simplest among them is to use the "+" operator. One sample usage of this is shown in the following example when I concatenate two lists called my_list1 and my_list2 .

Code:

```
# Concatenation example
my_list1 = ["John", 12, 50.7]
my_list2 = ["Sam", 25, "John", False, 100.2]
print("Original lists are :")
print(my_list1)
print(my_list2)
print("After concatenating the lists, you get the following list:")
print(my_list1 + my_list2)
```

Expected Output:

```
Original lists are :
['John', 12, 50.7]
['Sam', 25, 'John', False, 100.2]
After concatenating the lists, you get the following list:
['John', 12, 50.7, 'Sam', 25, 'John', False, 100.2]
```

Explanation:
Now you see an example to concatenate multiple lists. This is very common in Python programming. The concatenated list can have duplicates; for example, once you concatenate the lists, notice that ‘ John’ from both lists are present in the new list.

Goal:
I want to print a specific number of elements from a list. To demonstrate this, in this example, I print the last 3 elements, the last 2 elements, and the last element of a list.

Code:

```
# Printing a specific number of elements of a list from the # end.
my_list = ["John", 12, "Sam", True, 50.7]
print("The original list is:")
print(my_list)
print("The last 3 elements of the list are:")
print(my_list[-3:])
print("The last 2 elements of the list are:")
print(my_list[-2:])
print("The last element of the list is:")
print(my_list[-1])
```

Expected Output:

```
The original list is:
['John', 12, 'Sam', True, 50.7]
The last 3 elements of the list are:
['Sam', True, 50.7]
The last 2 elements of the list are:
```

```
[True, 50.7]
The last element of the list is:
50.7
```

Explanation:
This segment gives you the idea: how to print a specific element or a specific portion of elements inside a list (starting from the end location).

Goal:
I remove an element from a list. Here I use the del() function.
At first, I remove the element from index position 2. In the next step, I again remove another element from the modified list. This time I remove the element from index location 3.

Code:

```
# Removing an element using del()
my_list = ["John", 12, "Sam", True, 50.7]
print("The original list is:")
print(my_list)
print("Removing the element at index 2.")
del (my_list[2])
print("Now the list is:")
print(my_list)
print("Removing the element at index 3 from this updated list.")
del (my_list[3])
print("The updated list is:")
print(my_list)
```

Expected Output:

```
The original list is:
['John', 12, 'Sam', True, 50.7]
Removing the element at index 2.
Now the list is:
['John', 12, True, 50.7]
Removing the element at index 3 from this updated list.
The updated list is:
```

```
['John', 12, True]
```

Explanation:
This example shows the repetitive use of the del() function to modify a list.

Goal:
Removing an element from a list. Here I use remove() function.

Code:

```
# Removing an element using remove()function
my_list = ["John", 12, 25, 12, "Sam", True, 50.7]
print("The original list is:")
print(my_list)
print("Removing the first occurrence of 12 inside the list.")
my_list.remove(12)
print("Now the list is:")
print(my_list)
```

Expected Output:

```
The original list is:
['John', 12, 25, 12, 'Sam', True, 50.7]
Removing the first occurrence of 12 inside the list.
Now the list is:
['John', 25, 12, 'Sam', True, 50.7]
```

Explanation:
Initially, there were two 12 inside the list. The remove() function has removed the first occurrence of 12. It was present before element 25. Notice that another 12 is still present in this modified list.

Goal:
In Python, elements are case-sensitive. For example, in the following code, the remove() function can correctly remove the ‘Sam’ but not the ‘sam’ which appeared before ‘Sam’ .

Code:

```
# Elements are case-sensitive
my_list = ["John", 12, "sam", 25.7, "Sam", True]
print("The original list is:")
print(my_list)
print("Removing the first occurrence of 'Sam' inside the list.")
my_list.remove('Sam')
print("Now the list is:")
print(my_list)
```

Expected Output:

```
The original list is:
['John', 12, 'sam', 25.7, 'Sam', True]
Removing the first occurrence of 'Sam' inside the list.
Now the list is:
['John', 12, 'sam', 25.7, True]
```

Goal:
Removing an implement from a list using pop() . Inside this function, you need to supply the index location.

Code:

```
# Removing an element using pop()
my_list = ["John", 12, 25, 12, "Sam", True, 50.7]
print("The original list is:")
print(my_list)
print("Removing an element inside the list at index 3 using pop().")
my_list.pop(3)
print("Now the list is:")
print(my_list)
```

Expected Output:

```
The original list is:
['John', 12, 25, 12, 'Sam', True, 50.7]
Removing an element inside the list at index 3 using pop().
```

Now the list is:

```
['John', 12, 25, 'Sam', True, 50.7]
```

Explanation:
Initially, my_list contained two 12. The first 12 was at index location 1 and the second 12 was at index location 3. So, the following line of code: my_list.pop(3) removes the second occurrence of 12 from the original list.

Goal:
You want to examine whether a particular element is currently present inside a list.

Code:

```
# Checking whether an element is present inside a list
my_list = ["John", "Bob", "Sam", "Ester", 1, 2, 3, 4]
print("Is 'Sam' present inside the list?")
print('Sam' in my_list)  # True
print("Is 'Jeniffer' present inside the list?")
print('Jennifer' in my_list)  # False
print("Is 3 present inside the list?")
print(3 in my_list)  # True
print("Is 5 present inside the list?")
print(5 in my_list)  # False

# Checking whether an element is absent inside a list
print("Is 'Jeniffer' NOT present inside the list?")
print('Jennifer' not in my_list)  # True
```

Expected Output:

```
Is 'Sam' present inside the list?
True
Is 'Jeniffer' present inside the list?
False
Is 3 present inside the list?
True
Is 5 present inside the list?
```

```
False
Is 'Jeniffer' NOT present inside the list?
True
```

Explanation:
The output message is self-explanatory. You can note that“ **not in”** performs the reverse check. Notice that the following code segment:

```
print('Jennifer' not in my_list)  # True
```

outputs the following:

```
True
```

Goal:
I want to find the maximum element and minimum element from a list.

Code:

```
# Finding the maximum and minimum from a list
# This list contains the numbers only
my_list = [1, 23, 56.2, -3.7, 999]
print("The original list is:")
print(my_list)
print(f"The largest number is:{max(my_list)}")  # 999
print(f"The smallest number is:{min(my_list)}")  # -3.7
print("---------------")
```

Expected Output:

```
The original list is:
[1, 23, 56.2, -3.7, 999]
The largest number is:999
The smallest number is:-3.7
```

Note:
The max() and min() functions can work if the list contains numbers only; otherwise, you’ll encounter errors. For example, the following code segment:

```
#For this segment, you'll receive an error
my_list = [1, 23, 56.2, -3.7, 999, "abc", "bob"]
```

```
print("The original list is:")
print(my_list)
print(f"Largest number is :{max(my_list)}")
print(f"Smallest number is :{min(my_list)}")
```

Can generate the following error:

```
The original list is :
[1, 23, 56.2, -3.7, 999, 'abc', 'bob']
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter4/list_usage_file_2.py",
line 15, in <module>
    print(f"Largest number is :{max(my_list)}")
```
TypeError: '>' not supported between instances of 'str' and 'int'

Goal:
Testing max(), min() on boolean values.

Code:

```
# Testing booleans with max() and min()
my_list = [0.75, True, False, 0.5, 0.6, 1, 0]
print("The original list is:")
print(my_list)
print(f"The largest number is:{max(my_list)}")  # True
print(f"The smallest number is:{min(my_list)}")  # False
```

Expected Output:
```
The original list is:
[0.75, True, False, 0.5, 0.6, 1, 0]
The largest number is:True
The smallest number is:False
```

Explanation:
By design, if you use these Boolean values in numerical context, there is no

error; True is treated as 1 and False is treaded as 0. So, in this case, you can see the largest number is True and the smallest number is False .

To test this fact, in PyCharm, I rearrange the list. Notice that I have interchanged the positions of True and 1 in this list. Also, I have interchanged the position of False and 0 here.

```
# Testing booleans with max() and min()
my_list = [0.75, 1, 0, 0.5, 0.6, True, False]
print("The original list is:")
print(my_list)
print(f"The largest number is:{max(my_list)}")  # 1
print(f"The smallest number is:{min(my_list)}")  # 0
```

Now run this modified segment. You can see the following output:

```
The original list is as follows
[0.75, 1, 0, 0.5, 0.6, True, False]
The largest number is:1
The smallest number is:0
```

The online link https://docs.python.org/3/library/stdtypes.html#boolean-values from python doc confirms this saying:
"Boolean values are the two constant objects False and True. They are used to represent truth values (although other values can also be considered false or true). In numeric contexts (for example when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively. In numeric contexts (for example when used as the argument to an arithmetic operator), they behave like the integers 0 and 1."

Goal:
I want to add one element at the end of a list. I use the append() function for this purpose. Initially, I append 25 at the end of the list. Later I append another element " Bob " to the modified list.

Code:

```
# I want to add an element at the end of a list
my_list = ["John", 12, "Sam", True, 50.7]
```

```
print("The original list is:")
print(my_list)
print("Appending 25 at the end of the list.")
my_list.append(25)
print("Now the list is:")
print(my_list)
print("Appending another element 'Bob' now.")
my_list.append("Bob")
print("The modified list:")
print(my_list)
```

Expected Output:

```
The original list is:
['John', 12, 'Sam', True, 50.7]
Appending 25 at the end of the list.
Now the list is:
['John', 12, 'Sam', True, 50.7, 25]
Appending another element 'Bob' now.
The modified list:
['John', 12, 'Sam', True, 50.7, 25, 'Bob']
```

Note:
Using append (), you can add one element at a time. If you try to append multiple elements like the following:

```
my_list.append(10,20)
```

You will receive the error similar to the following:

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter4/list_usage_file_2.py",
line 52, in <module>
    my_list.append(10,20) #error
TypeError: append() takes exactly one argument (2 given)
```

Goal:

Using append(), you can add a single element only. But you can use this function to add a list that contains multiple elements. Let us see how it looks.

Code:

```
my_list = ["John", 12, "Sam", True, 50.7]
print("The initial list is:")
print(my_list)
print("Appending [10,'Bob',100.2] at the end of list:")
my_list.append([10, 'Bob', 100.2])
print("The modified list:")
print(my_list)
```

Expected Output:

```
The initial list is:
['John', 12, 'Sam', True, 50.7]
Appending [10,'Bob',100.2] at the end of the list:
The modified list:
['John', 12, 'Sam', True, 50.7, [10, 'Bob', 100.2]]
```

Explanation:
Notice that in this modified list, the final element is again a list. And it is NOT like other elements in my_list .

Goal:
You can add multiple elements to a list using the extend() function. Here I show you an example.

Code:

```
# You can add multiple elements to a list
# using extend() function.
# Here is an example.
my_list = ["John", 12, "Sam", True, 50.7]
print("The original list is:")
print(my_list)
print("Adding 10,'Bob', and 100.2 at the list end.")
my_list.extend([10, 'Bob', 100.2])
```

```
print("Now the list is:")
print(my_list)
```

Expected Output:

```
The original list is:
['John', 12, 'Sam', True, 50.7]
Adding 10,'Bob', and 100.2 at the list end.
Now the list is:
['John', 12, 'Sam', True, 50.7, 10, 'Bob', 100.2]
```

Additional note:
You can compare the result with the previous example when you used the append() function. You can see that when you use append() , you got the modified list as
['John', 12, 'Sam', True, 50.7, **[10, 'Bob', 100.2]**]
But using extend() , you have received the following output:

['John', 12, 'Sam', True, 50.7, **10, 'Bob', 100.2**]

Goal:
You saw that you can add the elements at the end of a list. But you have the option to add an element in a particular position. The insert() function is useful in this case. In the following example, I insert an element “Jack” into a list at index location 3.

Code:

```
my_list = ["John", 12, "Sam", True, 50.7]
print("The original list is:")
print(my_list)
print("Adding the element 'Jack' at index 3.")
my_list.insert(3, "Jack")
print("Now the list is as follows:")
print(my_list)
```

Expected Output:

```
The original list is:
```

```
['John', 12, 'Sam', True, 50.7]
Adding the element 'Jack' at index 3.
Now the list is as follows:
['John', 12, 'Sam', 'Jack', True, 50.7]
```

Goal:
You can sort your list that contains the same datatype. Here is an example.

Code:

```
my_list = [33, 11, 555, 77, 111, 333]
print("The initial list is:")
print(my_list)
print("Using sort() on my_list now.")
my_list.sort()
print("Now the list is:")
print(my_list)
```

Expected Output:

```
The initial list is:
[33, 11, 555, 77, 111, 333]
Using sort() on my_list now.
Now the list is:
[11, 33, 77, 111, 333, 555]
```

Note:
Two points to remember when you use the sort() function:

- It changes the original list. If you want to avoid modifying the original list, you can refer sorted() function which I discuss next.
- If you have mixed data types in your list and you apply sort() on that, you'll encounter errors. Here is an example, for the following code segment:

```
my_list = ["John", 12, "Sam", True, 50.7]
my_list.sort()#error now
```

you’ll see the following error:

```
TypeError: '<' not supported between instances of 'int' and 'str'
```

Goal:
As said before, if you want to prevent modification of the original list, you can use the sorted() function. Here I show you an example.

Code:

```
my_list = [33, 11, 555, 77, 111, 333]
#my_list = ["sam","bob","jack"]
print("Initially, my_list is:")
print(my_list)
print("Printing the sorted list now.")
print(sorted(my_list))
print("The my_list now:")
print(my_list)
```

Expected Output:

```
Initially, my_list is:
[33, 11, 555, 77, 111, 333]
Printing the sorted list now.
[11, 33, 77, 111, 333, 555]
The my_list now:
[33, 11, 555, 77, 111, 333]
```

Additional note:
Keep in mind that you should use this function on the same datatypes.

Working with Tuples

Tuples are another important data type and similar to lists. But there are some noticeable differences which are as follows:

- Tuples are immutable. This means once created, you cannot incorporate changes in them. But you have seen that lists can be modified. For example, you reassigned a value in a list, you extended a list, etc. So, lists are mutable.
- When you create a tuple, you use codes something like the following (here I have chosen the variable name as my_tuple):

```
my_tuple = ("John", 12, "Sam", True, 50.7)
```

you can see the declaring a tuple is similar to a list, but this time, I put elements inside the round brackets - (,)

All the functions that are available for lists are not available for tuples. For example, if you write the following:

```
my_tuple = ("John", 12, "Sam", True, 50.7)
my_tuple.remove(12) #error
```

You can notice the error saying:

```
AttributeError: 'tuple' object has no attribute 'remove'
```

Or, if you write the following:

```
my_tuple.append("Jack") #error
```

You can notice the error saying:

```
AttributeError: 'tuple' object has no attribute 'append'
```

But there are some similarities with lists too. You can also convert a list to a tuple. So, let’s examine some built-in functions for tuples now.

Goal:
Declaring a tuple and printing the elements inside it.

Code:

```
my_tuple = ("John", 12, "Sam", True, 50.7)
print("The content of my_tuple is:")
print(my_tuple)
```

Expected Output:

```
The content of my_tuple is:
('John', 12, 'Sam', True, 50.7)
```

Goal:
I want to access the tuple elements.

Code:

```
# Indexing is similar to lists
my_tuple = ("John", 12, "Sam", True, 50.7)
print("The content of my_tuple is:")
print(my_tuple)
# Printing the first element
print("The first element is:")
print(my_tuple[0])
print("The last element is:")
print(my_tuple[-1])
print("Printing the elements from index 1 to index 3.")
print(my_tuple[1:4])
print("Printing the elements from index 2 to end.")
print(my_tuple[2:])
```

Expected Output:

```
The content of my_tuple is:
('John', 12, 'Sam', True, 50.7)
The first element is:
John
The last element is:
50.7
Printing the elements from index 1 to index 3.
(12, 'Sam', True)
Printing the elements from index 2 to end.
('Sam', True, 50.7)
```

Goal:
I show you that you cannot reassign the value inside a tuple.

Code:

```
# You cannot reassign the value inside a tuple.
my_tuple = ("John", 12, "Sam", True, 50.7)
print("The content of my_tuple is:")
print(my_tuple)
print("Trying to replace 'Sam' with 'Bob':")
my_tuple[2]= 'Bob' #error
```

Expected Output:
You are supposed to see the error similar to the following:

```
The content of my_tuple is:
('John', 12, 'Sam', True, 50.7)
Trying to replace 'Sam' with 'Bob':
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter4/tuple_usage_file.py",
line 29, in <module>
    my_tuple[2]= 'Bob' #error
TypeError: 'tuple' object does not support item assignment
```

Explanation:
Tuples are immutable by design. So, you cannot modify them.

Goal:
Converting a list to a tuple.

Code:

```
my_list = ["John", 12, "Sam", True, 50.7]
print("The content of my_list is:")
print(my_list)
```

```
# Converting the list to a tuple
my_tuple=tuple(my_list)
print("The content of my_tuple is:")
print(my_tuple)
```

Expected Output:

```
The content of my_list is:
['John', 12, 'Sam', True, 50.7]
The content of my_tuple is:
('John', 12, 'Sam', True, 50.7)
```

Explanation:
You can use the built-in tuple() function to convert a list to a tuple.

Goal:
Reversing a tuple.

Code:

```
my_tuple = (1, 2, 3, 4, 5)
print("The content of my_tuple is:")
print(my_tuple)

print("Reversing the tuple:")
rev_tuple = tuple(reversed(my_tuple))
print("The content of rev_tuple is:")
print(rev_tuple)
```

Expected Output:

```
The content of my_tuple is:
(1, 2, 3, 4, 5)
Reversing the tuple:
The content of rev_tuple is:
(5, 4, 3, 2, 1)
```

Working with Dictionaries

Now you see the use of another important data type. You call it a dictionary . These are some noticeable characteristics of this datatype:

- It is a key-value pair.
- The keys are unique.
- Keys and values can be of any type.
- Dictionaries are indexed through its keys.
- When you create a dictionary, you use codes something like the following (here I have chosen the variable name as my_dictionary):

  my_dictionary = {1: "John", 2: 12, 3: "Sam", 4: True, 5: 50.7}

  you can see that I put elements inside the curly brackets –{ , } now.

Let us examine some built-in functions for dictionaries which are as follows.

Goal:
I want to create a dictionary and print the details inside it.

Code:

```
# A dictionary with 5 key-value pair
my_dictionary = {1: "John", 2: 12, 3: "Sam", 4: True, 5: 50.7}
print("The my_dictionary contains:")
print(my_dictionary)
```

Expected Output:

```
The my_dictionary contains:
{1: 'John', 2: 12, 3: 'Sam', 4: True, 5: 50.7}
```

Goal:

I want to use different types of keys in my dictionary. In the following code segment, you’ll see that first three keys are numbers and remaining keys are strings.

Code:

```
# A dictionary with 5 key-value pair
# Choosing different types of keys in the same dictionary
my_dictionary = {1: "John", 2: 12, 3: "Sam", 'fourth': True, 'fifth': 50.7}
print("The my_dictionary contains:")
print(my_dictionary)
print("---------------")
```

Expected Output:

```
The my_dictionary contains:
{1: 'John', 2: 12, 3: 'Sam', 'fourth': True, 'fifth': 50.7}
```

Goal:
I want to print values for particular keys in a dictionary.

Code:

```
# I want to print values for particular keys
my_dictionary = {1: "John", 2: 12, 3: "Sam", 'fourth': True, 'fifth': 50.7}
print("The my_dictionary contains:")
print(my_dictionary)
print("Value at key 1:", my_dictionary[1])
print("Value at key 2:", my_dictionary[2])
print("Value at key 3:", my_dictionary[3])
print("Value at key 'fourth':", my_dictionary['fourth'])
print("Value at key 'fifth'::", my_dictionary['fifth'])
```

Expected Output:

```
The my_dictionary contains:
{1: 'John', 2: 12, 3: 'Sam', 'fourth': True, 'fifth': 50.7}
Value at key 1: John
Value at key 2: 12
Value at key 3: Sam
Value at key 'fourth': True
Value at key 'fifth':: 50.7
```

Goal:
I assign different values for the same key in a dictionary and want to see the effect. I also want to check -how many elements are present in the dictionary?

Code:

```
# If you assign different values for the same keys
# last assigned value will be kept
my_dictionary = {1: "John", 2: "Sam", 3: "Jack",1: "Bob"}
print("The my_dictionary contains:")
print(my_dictionary)
print("Value at key 1:", my_dictionary[1])
print(f"Number of contents:{len(my_dictionary)}")
```

Expected Output:

```
The my_dictionary contains:
{1: 'Bob', 2: 'Sam', 3: 'Jack'}
Value at key 1: Bob
Number of contents:3
```

Explanation:
It’s important to note that in a case like this, the dictionary will keep the last assigned value. For example, initially, I assigned John to key 1 , but later I assigned Bob to key 1 . So, when I print the details of the dictionary, you can see that the dictionary holds the last assigned value Bob for key 1 . The last line in the output shows that you can use the len() function to determine the total number of elements in your dictionary.

Working with Sets

Sets are a special kind of datatype that does not hold duplicate values. You can think of it as a combination of a list and a dictionary. For example, like dictionaries, you use curly brackets { and } but like lists, you do not have any key-value pair. The following is an example of a set:

```
my_set1 = {1, 2, 3, "Jack", "Bob"}
```

Alternatively, to create a set, you can use the built-in set() function like the following:

```
myset=set(iterable_element)
```

Where " iterable_element "indicates that you can use something like a list, or a tuple like the following:

```
#Alternative way to create a set
#Using a list now to create a set
my_set = set([1, 2, 3, "Jack", "Bob"])

#Using a tuple to create a set
my_set = set((1, 2, 3, "Jack", "Bob"))
```

Now go through the following code fragments to get some idea about set data types.

Goal:
I supply duplicate values to sets. Then I test whether these sets contain duplicates.

Code:

```
my_set1 = {1, 2, 3, "Jack", 2, "Bob", 3, 1}
print("The my_set1 contains:")
print(my_set1)

my_set2 = {"Sam", "Bob", "Jack", "Sam", "Jack", "Ester"}
print("The my_set2 contains:")
print(my_set2)
```

Expected Output:

```
The my_set1 contains:
{1, 2, 3, 'Jack', 'Bob'}
```

```
The my_set2 contains:
{'Sam', 'Jack', 'Bob', 'Ester'}
```

Explanation:
You can see that sets can’t have duplicates.

Goal:
Sets are mutable. In the following example, I’m adding 6 to the list and then removing 2 from the set.
Code:

```
# Sets are mutable
my_set = {1, 2, 3, 4, 5}
#my_set = {}#it is treated as empty dictionary
#my_set=set()#this is ok for an empty set
print("The my_set contains:")
print(my_set)
print("Adding 6 to the set now.")
my_set.add(6)
print("Now the set is:")
print(my_set)
print("Removing 2 from the set now.")
my_set.remove(2)
print("Now the my_set is:")
print(my_set)
```

Expected Output:

```
The my_set contains:
{1, 2, 3, 4, 5}
Adding 6 to the set now.
Now the set is:
{1, 2, 3, 4, 5, 6}
Removing 2 from the set now.
Now, the my_set is:
{1, 3, 4, 5, 6}
```

Explanation:

Here is an important point to note: if you just use the following:

```
my_set = {}#it is treated as empty dictionary
```

Python considers my_set as an empty dictionary, not an empty set. So, in this case, If you use the following code fragment:

```
my_set = {}
my_set.add(6)
```

you'll see the following error:

```
AttributeError: 'dict' object has no attribute 'add'
```

So, what is the remedy? You can opt for an alternative way to create a set. For example, the following line of code can create an empty set.

```
my_set=set()#this is ok for an empty set
```

and now the following lines of codes will work fine for you:

```
print("The set,called 'my_set' is as follows:")
print(my_set)
print("Adding 6 to the set now.")
my_set.add(6)
```

Goal:
You can iterate over strings. So, you can pass a string argument to the set() function.

Code:

```
# Strings are iterable. So, you can use strings to set().
my_str = "vaskaran"
my_set = set(my_str)
print("The my_set contains:")
print(my_set)
```

Expected Output:

```
The my_set contains:
{'n', 's', 'k', 'a', 'v', 'r'}
```

Additional note:
You can see that my_set is unordered and it does not contain the duplicates.

Goal:
In the previous code segment, you saw that sets are unordered. Now you see that you cannot access the set elements referring to an index. Here is an example.

Code:

```
# Sets are unordered. You cannot access the elements by referring to an index
my_set = set([2,"abc",1,5])
print("The my_set contains:")
print(my_set)
print("Trying to access the 0th element.")
print(my_set[0])#error
```

Expected Output:

```
The my_set contains:
{1, 2, 5, 'abc'}
Trying to access the 0th element.
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter4/set_usage_file.py",
line 59, in <module>
    print(my_set[0])#error
TypeError: 'set' object is not subscriptable
```

Goal:
You cannot access set elements by referring to an index. But You can loop through the elements. Here is an example.

Code:

```
# You cannot access set elements by referring to an index.
# But You can loop through the elements.
```

```
my_set = set([2, "hello", 1, 5])
print("The my_set contains:")
for item in my_set:
    print(item)
```

Expected Output:

```
The my_set contains:
{1, 2, 'hello', 5}
Trying to access the 0th element.
The my_set contains:
1
2
hello
5
```

Goal:
Removing an element from a set. Here I show you the use of both remove and discard. The difference is: discard() does not raise an error if the element is not present, but remove() raises an error in that case.

Code:

```
# Remove an element from a set
my_set = set([1,2,3,4,5])
print("The my_set contains:")
print(my_set)
print("Removing 5 using remove().")
my_set.remove(5)
print("Now the my_set contains:")
print(my_set)
print("Removing 4 now using discard().")
my_set.discard(4)
print("Now the my_set contains:")
print(my_set)
```

Expected Output:

```
The my_set contains:
{1, 2, 3, 4, 5}
Removing 5 using remove().
Now the my_set contains:
{1, 2, 3, 4}
Removing 4 now using discard().
Now the my_set contains:
{1, 2, 3}
```

Note:

Append the following code:

```
# Trying to remove 4 when it is absent in the set.
my_set.discard(4) # no error is raised
my_set.remove(4)  # error is raised
```

If you execute the code, you see that remove() raises an error because 4 is not present in my_set now. The error is similar to the following:

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter4/set_usage_file.py",
line 82, in <module>
    my_set.remove(4)  # error is raised
KeyError: 4
```

But the discard() does not raise any error for this.

EXERCISE

E6.1 Create a list that contains more than 2 elements. Then remove the last two elements from the list.

E6.2 Create a tuple with more than 5 elements. Then print the 3rd element of it. Can you print the 3rd element from last?

E6.3 Create a tuple with elements 1,2,2,3,4,4,4,5.Then reverse the tuple and print how many times 4 appears in this tuple.

E6.4 How a tuple is different from a list?

E6.5 Predict the output:

```
my_tuple = (1, 2, 2, 3, 4, 4, 5)
my_set = set(my_tuple)
print("The my_set is:")
print(my_set)
```

E6.6 Can you get any error for the following code?

```
my_list=["red", "blue"]
my_set = set(my_list)
print("The my_set is:")
print(my_set)
my_set.discard("green")
```

E6.7 Create a dictionary and print the details. Can you print the value for a particular key?

Solution to Exercises

E6.1

```
my_list = ["John", 12, 25, 12,"Sam", True, 50.7]
print("The original list is:")
print(my_list)
#Removing the last two elements from the list
del(my_list[-2:])
print("Now the list is:")
print(my_list)
```

Output:

```
The original list is:
['John', 12, 25, 12, 'Sam', True, 50.7]
Now the list is:
['John', 12, 25, 12, 'Sam']
```

E6.2

```
my_tuple = ("John", 12, 25, 12, "Sam", True, 50.7)
print("The original tuple is:")
print(my_tuple)
#Printing the 3 element
print("The third element:")
print(my_tuple[2])
print("The third element from last:")
print(my_tuple[-3])
```

E6.3

```
my_tuple = (1, 2, 2, 3, 4, 4, 4, 5)
print("The original tuple is:")
print(my_tuple)

print("The reversed tuple is:")
rev_tuple = tuple(reversed(my_tuple))
print(rev_tuple)
print(f"The number of 4 in this tuple:{rev_tuple.count(4)}")
```

E6.4
Tuples are immutable but lists are mutable.

E6.5

The my_set is:
{1, 2, 3, 4, 5}

Explanation:
Sets do not contain duplicates.

E6.6
The element “green” is not present in the set. If you use remove() , you see a KeyError , but discard() does not raise any such error. The discard() method removes the element if it is present in the set.

E6.7
Do it yourself. I have shown you code samples in this chapter.

CS 4.1 Implementation

I use a dictionary to store answers to the questions. I also use the len() function to get the total number of questions in this question bank. To beautify the output, I print the questions and corresponding options in the new lines. Here is the complete implementation for you.

```
# All questions
question1 = "Q1.What is the value of the expression:2*3-4?" \
            "\n(a)1" \
            "\n(b)2" \
            "\n(c)3" \
            "\n(d)None."
question2 = "\nQ2.What is the value of the expression:1+(2*3)/4?" \
            "\n(a)1.5" \
            "\n(b)3" \
            "\n(c)3.5" \
            "\n(d)None."
question3 = "\nQ3.The set data type can hold duplicate values." \
            "The statement is:" \
            "\n(a)False" \
            "\n(b)True" \
            "\n(c)Partially correct." \
            "\n(d)None."

# Storing the questions with answer keys
# inside the following dictionary.
question_bank = {question1: "b",
                 question2: "c",
                 question3: "a"}
print("Welcome to the MCQ test.")
print("="*25)
score = 0  # initial value
for key in question_bank:
    print(key)
    user_input = input("Type your answer(a/b/c/d):")
    if user_input == question_bank[key]:
        score += 1
print(f"\nYour Score:{score} out of {len(question_bank)}")
```

Possible improvements

- Use a function to make a better implementation. In chapter7, you’ll learn about functions. At the end of Chapter 7, you’ll see a better solution.
- You can store more questions and pick a subset of these questions at random.

Case Study V

CS 5.1 Problem Statement

This time you make a calculator. Using the calculator, you should able to perform the basic operations-addition, subtraction, multiplication, and division. Here I provide you a sample output when the user supplies the valid inputs:

```
========================
This is a simple calculator.
It supports the following operations:
i)Addition
ii)Subtraction
iii)Multiplication and
iv)Division.
========================
Enter the first number:12.5
Enter the next number:3
Enter an operator(+,-,*,/): *
The final result is:37.5
```

Author's Comment:

Chapter 8 can help you evaluate all the valid inputs. In this project, you add one simple validation. I want that you consider a case when a user supplies an invalid operator. In this case, your application should tell the user about this error. Here is a sample output for this negative scenario:

```
========================
This is a simple calculator.
It supports the following operations:
i)Addition
ii)Subtraction
iii)Multiplication and
iv)Division.
========================
Enter the first number:2
Enter the next number:5
Enter an operator(+,-,*,/): >
Invalid operator. Cannot compute the result.
```

CS 5.2 Problem Statement

You can organize the CS 4.1 implementation in a better way. Your guess is correct. I want that you use functions in that code.

Chapter 7: Functions and Modules

This chapter covers user-defined functions, lambda functions, and modules. These are very common in Python programming. You use them to organize your code.

Function Overview

We can think of a function as a logical set of statements that perform a specific task. You use functions to divide our code into manageable pieces. They allow you to optimize and reuse your code. Consider a simple scenario. Suppose you have written a function that can calculate a sum of two numbers. So, whenever you want to compute the sum of two numbers, instead of rewriting the code, you can simply invoke this function.

When you see a function usage in someone's code, you notice the following things:

- They define the function to describe behaviors.
- They call the function single or multiple times.

An ordinary function has a name, a body. It can have zero, one, or more parameters. You will be familiar with all of them shortly. Before I discuss more on functions, let me cover the following points:

A function definition starts with the executable code called, def. Then you type the function name with parameters (if any). Finally, a def statement ends with a colon. I told you that a function may not have parameters. So, let me show you a simple function with no parameter. This function can print hello.

```
def print_hello():
    print("Hello")
```

You can invoke this function using the following code: print_hello(). You'll see the complete demonstration shortly.

The statements of the body of the function are indented. This format is used to show that all these statements belong to the function.

POINTS TO REMEMBER

If you use PyCharm IDE, once you press enter to go to the next line, this indentation is automatic. Otherwise, you can simply use the same characters for indentation. For example, you can use a tab or space; but you should not mix both.

Your function can have parameters. For example, consider the following function which takes two parameters-the first one is for a name , the second one is for age . Here is the function, called print_details for you:

```
def print_details(name,age):
    print(f"Hello {name}! How are you?")
   print(f"You are now {age}.")
```

As a result, I can use the following line of code:

```
print_details("Bob",20)
```

When I execute this code, I expect to see the following output:

```
Hello Bob! How are you?
You are now 20.
```

You can have function documentation. Function documentation helps others to understand *what the function does*. You can also print this documentation using **<your_function_name>.__doc__** . Note that you're seeing the use of double underscores here. To understand this, let me add some documentation to the previous function which is as follows:

```
def print_details(name,age):
   """
   This function takes two parameters.
   You can supply the name and age of the user
   in this function.
   """
   print(f"Hello {name}!How are you?")
   print(f"You are now {age}.")
```

Now you can print this documentation using the following line of code:

```
print(print_details.__doc__)
```

Demonstration 1

Go through the following demonstration. It contains all the functions and associated codes that I have discussed so far.

```
#Function example-1
print("***Function example-1.***")


def print_hello():
    print("Hello")


print_hello()

#Function example-2
print("\n***Function example-2.***")


def print_details(name, age):
    """
    This function takes two parameters.
    You can supply the name and age of the user
    in this function.
    """
    print(f"Hello {name}! How are you?")
    print(f"You are now {age}.")


print_details("Bob", 20)
```

Output

Here is the output.

```
***Function example-1.***
Hello

***Function example-2.***
Hello Bob! How are you?
You are now 20.
```

Analysis

Notice that when I call print_details("Bob", 20) , I pass two arguments in the function call. The first one is a string and the second is a number. This shows the fact that a function can have multiple parameters that can be of different types.

Before you see some other characteristics of functions, let's have a quick discussion about argument and parameter.

Argument vs Parameter

People often use the words arguments and parameters interchangeably. But expert programmers are particular about this. The variables in a function definition are called parameters of the function. For example, when you see the following function definition:

```
def print_details(name,age):
        //some code
```

The name and age are called parameters(or, formal parameters) of the function print_details . But, when you invoke the function using the following code: print_details("Bob",20), we say that "Bob" and 20 are the arguments you've passed to this function. Similarly, in the line print_details("Sam",30) , "Sam" and 30 are arguments. Many developers refer to them as actual parameters as well.

You can say that we pass the arguments to a function. These values are assigned to the function parameters. But I already mentioned that some programmers do not emphasize these terms too much. So, they interchangeably use these terms.

You know about arguments and parameters now. Let us continue the discussion on some other characteristics of functions.

You can repeat function calls and you can vary the arguments. In demonstration1, you have seen the following line:

```
print_details("Bob",20)
```

You can repeat the line as many times as you want. You can also vary the arguments of your function. So, the following segment of code:

```
def print_details(name,age):
    """
    This function takes two parameters.
    You can supply the name and age of the user
    in this function.
    """
    print(f"Hello {name}! How are you?")
    print(f"You are now {age}.")


print_details("Bob", 20)
#You can repeat function calls.
print_details("Bob", 20)
# You can vary the arguments of the functions
print_details("Sam", 35)
```

Can produce the following output.

```
Hello Bob! How are you?
You are now 20.
Hello Bob! How are you?
You are now 20.
Hello Sam! How are you?
You are now 35.
```

Discussion on Function Arguments

Now you know about function arguments and parameters. These are essential to understand the upcoming section. Let us take a deeper look at a function argument now.

Positional Argument

In the previous segment of code, instead of writing the following code:

```
print_details("Bob",20)
```

if you write:

```
print_details(20,"Bob") # Unexpected outcome
```

You'll receive the following output:

```
Hello 20! How are you?
You are now  Bob
```

Oh..no! How is this possible? Yeah, you've guessed it right. You needed to pass the arguments in the correct order. For example, in this case, 20 is assigned for the name parameter(as it was in the first position in function definition) and 'Bob' is assigned for the age parameter(which is in the second position in function definition). Unless you specify these differently, by default, Python expects that you pass the arguments following the order of the parameters which are defined in the function definition. When the values are matched in this way, you say that you are following **positional arguments.**

Keyword Arguments

Let us invoke the function differently. This time, I use the following lines:

```
print_details(age=20,name="Bob") # Expected outcome
print_details(name="Bob", age=20) # Again expected outcome
```

As mentioned in the comments, in both cases, you get the expected result. You can verify the result when you execute the code. For example, in the previous demonstration, execute the following two lines of code. You'll see the following output:

```
Hello Bob! How are you?
You are now 20
Hello Bob! How are you?
You are now 20
```

So, what we see now is that when you pass arguments in this **name-value** pair, you will always see the expected result. Programmatically, these are called **keyword arguments.**

Use of Default Values

You always want to make your programming life easier. So, instead of supplying values in each case, you may be interested to use some default values. There are several reasons for this, but I am listing a few of them:

- You discover that in a specific function call, a particular argument always gets the same value.
- You want to type less.
- You do not wish to pass all values in your function invocation.
- You are not sure about the accepted value of a parameter.

And so forth.

To address this type of concern, let's modify the previous function definition as follows:

```
#Using default values in a function
def print_details(name="Sam",age=35):
    """
    This function takes two parameters.
    You can supply the name and age of the user
    in this function.
    By default, the name is 'Sam' and the age is 35.
    """
    print(f"Hello {name}! How are you?")
    print(f"You are now {age}.")
```

Notice the presence of (name="Sam",age="35") in the updated function. I've updated the function documentation too. You can see that I've added the following line:

```
By default, the name is 'Sam' and the age is 35.
```

You know that it is a documentation comment and it is optional for you. But it can provide better readability. The changes in the modified function definition tell us- now onwards, you can invoke the function with or without parameters. It is because you have supplied default values for all these parameters. Let us test this. Use the following lines of codes now and refer to the associated comments for a better understanding.

```
print_details()#Will take both the default values
print_details(name="Jack")#Will take age=35 as default
print_details(age=45)#Will take name="Sam" as default
#None of the default values are considered now
print_details("Bob",20)
```

When you execute this segment of code, you're expected to see the following output:

```
Hello Sam! How are you?
You are now 35.
Hello Jack! How are you?
You are now 35.
Hello Sam! How are you?
You are now 45.
Hello Bob! How are you?
You are now 20.
```

Demonstration 2

Here is the complete demonstration for you.

```
#Using default values in a function
def print_details(name="Sam",age=35):
    """
    This function takes two parameters.
    You can supply the name and age of the user
    in this function.
    By default, the name is 'Sam' and the age is 35.
    """
    print(f"Hello {name}! How are you?")
    print(f"You are now {age}.")
```

```
print_details()  #Will take both the default values
print_details(name="Jack")  #Will take age=35 as default
print_details(age=45)  #Will take name="Sam" as default
#None of the default values are considered
print_details("Bob", 20)
```

Output

Here is the output. I’ve marked the default values in the output in bold.

Hello **Sam**! How are you?
You are now **35**
Hello Jack! How are you?
You are now **35**
Hello **Sam**! How are you?
You are now 45
Hello Bob! How are you?
You are now 20

Demonstration 3

In this demonstration, I calculate the sum of the two numbers. Here is my function:

```
def calculate_sum(x, y):
    return x + y
```

Using this function, I can calculate the sum of 12 and 15 . Using the same function, I can also the sum of 20.5 and 37 . So, you can see that I do not need to write the programming logic repeatedly. This demonstrates the concept of code reuse using functions. Let us go through it.

```
def calculate_sum(x, y):
    return x + y


total = calculate_sum(12, 15)
print(f"The sum of 12 and 15 is:{total}")

total = calculate_sum(20.5, 37)
```

```
print(f"The sum of 20.5 and 37 is:{total}")
```

Output

Here is the output.

```
The sum of 12 and 15 is:27
The sum of 20.5 and 37 is:57.5
```

Analysis

If you write the following:

```
total = calculate_sum('10','20') #1020
print(f"The sum of '10" and '20' is:{total}")
```

you see the following output:

```
The sum of '10" and '20' is:1020
```

You can guess the reason. This time you used two strings, instead of two numbers. This is why you see a concatenated string (1020) in the output.

Discussion on Return Values

In demonstration1, you used the function print_hello() as follows:

```
def print_hello():
    print("Hello")
```

and when you used the following line: print_hello(), you could see the output immediately. Again, in demonstration3, you saw the following lines:

```
def calculate_sum(x, y):
    return x + y


total = calculate_sum(12, 15)
print(f"The sum of 12 and 15 is:{total}")
```

but this time the calculate_sum() function took two numbers as arguments; adds them and stores the result into another variable called total .

So, you could see the result when you display the value inside the total variable.

This example shows you a simple fact: When you use a function, you do not need to print the result immediately. Instead, you can store the value that comes out from a function into another variable. This value is called the **return value** of a function.

You often see that a function returns only one value. But using a smart program, you can return multiple values from a function too. I have already shown you a function with no return values and a function with only one return value. This time let us concentrate on a function that can return multiple values.

Demonstration 4

In this example, I have a simple list called initial_list . This list contains the values 1,2,3,4, and 5. I also have a function called make_double(). This function takes a list as an argument, makes the elements double. Finally, it appends the result into another list called double_list which was initially empty.

When you invoke the make_double() function and pass the initial_list as an argument, I show you a way how to return multiple values from a function. I print the double_list at the end of this program. You can see all the double values of the original integers which I initially supplied through a list.

```
def make_double(input_list):
    """
    It is a function that can return multiple values.
    Each element in the list will be doubled
    by this function.
    """
    for element in input_list:
        double_list.append(2 * element)


print("The initial_list is :")
print(initial_list)
print("Calling the function make_double() now.")
```

```
make_double(initial_list)
print("The double_list is :")
print(double_list)
print("The initial_list at present :")
print(initial_list)  #unchanged initial list
```

Output

Here is the output.

```
The initial_list is :
[1, 2, 3, 4, 5]
Calling the function make_double() now.
The double_list is :
[2, 4, 6, 8, 10]
The initial_list at present :
[1, 2, 3, 4, 5]
```

Demonstration 5

In the previous demonstration, you did not modify the original list. But in some cases, once you create a new list, you do not need to maintain the old/original list. But keep in mind that this can be risky.

The following demonstration shows you such an example, where you remove the elements from the initial list, make them double, and create a new list. So, in the end, when you print the original list, you'll see that the initial list is empty.

```
initial_list = [10, 20, 30, 40, 50]
double_list = []


def make_double(input_list):
    """
    It is a function that can return multiple values.
    Each element in the list will be doubled
    by this function.
    """
    while input_list:
        element = initial_list.pop()
        #print(element)
        double_list.append(2 * element)

print("The initial_list is:")
print(initial_list)
print("Calling the function make_double() now.")
make_double(initial_list)
print("The double_list is:")
print(double_list)
print("The initial_list at present :")
print(initial_list)
```

Output

Here is the output.

```
The initial_list is:
[10, 20, 30, 40, 50]
Calling the function make_double() now.
The double_list is:
[100, 80, 60, 40, 20]
The initial_list at present :
[]
```

Lambda Function

Lambda functions fall into advanced concepts, but I want you to know about this. A lambda function is a function with no name. This anonymous function can take one or multiple arguments, but a single expression. A common function starts with the def keyword. A lambda function starts with a lambda keyword.

Demonstration 6

This demonstration uses two lambda functions. The first one does not take any argument. It simply prints Hello, Reader! The next one takes one argument and doubles a number.

```
print("***Lambda function example-1.***")
say_hello = lambda: print("Hello, Reader!")
say_hello()

print("***Lambda function example-2.***")
make_double = lambda x:x * 2
print(f"Double of 10 is:{make_double(10)}")
print(f"Double of 25.35 is:{make_double(25.35)}")
```

Output

Here is the output.

```
***Lambda function example-1.***
Hello, Reader!
***Lambda function example-2.***
Double of 10 is:20
Double of 25.35 is:50.7
```

Demonstration 7

You can use lambda functions as an argument for another function. For example, in the following in-built map() function, I use a lambda function and a list as arguments. The initial list contains some numbers. I use the lambda function to increment each number by 2 in the list.

The map function requires a function object and any number of iterables, such as a list or a dictionary. So, in the following demonstration, I pass a lambda function and a list to fulfill the need. Go through the following demonstration and output for your reference.

```
original_list = [1, 2, 3, 4, 5]
```

```
# Adds 2 to each item in the list
new_list = list(map(lambda x: x + 2, original_list))
print("The original list is as follows:")
print(original_list)
print("The updated list is as follows:")
print(new_list)
```

Output

Here is the output.

```
The original list is as follows:
[1, 2, 3, 4, 5]
An updated list is as follows:
[3, 4, 5, 6, 7]
```

POINT TO NOTE

In this example, you see a warning message which says: PEP 8: E731 do not assign a lambda expression, use a def. It means that Python recommends you to write code like:

```
def say_hello():
    print("Hello, Reader!")


say_hello()
```

In the end, it is your choice which approach you want to follow.

Modules

In real-world applications, normally the code size is gigantic. If you place the entire code is in a single place, you create a big file. This kind of practice is commonly discouraged. Instead, Python allows you to place your code in a separate file with a .py extension. Using an import statement, you make these codes available in a current file. What are the advantages? Go through the following bullet points:

- You make the current program file small and focus on the high-level logic in this file.
- The inner working logic is hidden. Because that part of the code is written in a separate file that is not immediately visible to the user.

A module can contain many things. For example, you can place variables, functions, and classes in a module. In Chapter 11, I have shown you how to import a single class, multiple classes, or the entire module. I have also discussed what are the recommended practices. To avoid repetition, now I show you some simple usage of a module that contains the variables and functions only.

Let us make a file called **my_library.py** and place the following codes into it. You can see that this file contains a list, a dictionary, and two functions.

On my computer, I’ve organized the codes chapter-wise. For example, I stored all programs of the chapter7 inside a directory named chapter7 . The parent directory of the **chapter7** is **PythonCrashCourse** . If I store the **my_library.py** inside the **chapter7** directory, every time I refer to this module, I need to mention it as **chapter7.my_library** . To avoid less typing, I store this inside **PythonCrashCourse** . Now I can directly refer to the module without mentioning the **chapter7** .

```
# A simple list
my_list = [1, 2, 3, 4, 5]

# A simple dictionary
my_dictionary = {1: "Jack", 2: "Kate", 3: "Bob"}


def make_total_double(first, second):
    """
    This function takes two numbers
    and returns the double.
    """
    total = first + second
    return total * 2


def make_average(first, second, third=0):
    """
    This function takes three numbers and
    returns the average.
    The third number is optional.
    """
    total = first + second + third
    return total / 3
```

Now create a file using_molules_example_1.py inside chapter7 and write the following lines of code:

```
from my_library import my_list, my_dictionary

print("Accessing some list elements:")
print(f"my_list[0]={my_list[0]}")
print(f"my_list[3]={my_list[3]}")

print("\nAccessing some dictionary elements:")
print(f"my_dictionary[1]={my_dictionary[1]}")
print(f"my_dictionary[2]={my_dictionary[2]}")
```

If you run this code, you see the following output:

```
Accessing some list elements:
my_list[0]=1
my_list[3]=4

Accessing some dictionary elements:
my_dictionary[1]=Jack
my_dictionary[2]=Kate
```

You can see that I can access the list and dictionary elements from this file, though these elements live in a separate file. I can access them because I use the following statement at the beginning of the file:

```
from my_library import my_list, my_dictionary
```

In the same way, you can use the functions from my_library.py . To test this, make the following function available using an import statement. Now you write the following lines of code in this file.

```
from my_library import make_total_double
print("Using the make_total_double function now.")
result = make_total_double(20, 30.5)
print(result)
```

This code segment can produce the following output:

```
Using the make_total_double function now.
101.0
```

Hope you understand the key usage of the module. Let us gather all these points in the following demonstration.

Demonstration 8

Here is the content of using_molules_example_1.py for you.

```
from my_library import my_list, my_dictionary, make_total_double
print("Accessing some list elements:")
print(f"my_list[0]={my_list[0]}")
print(f"my_list[3]={my_list[3]}")

print("\nAccessing some dictionary elements:")
print(f"my_dictionary[1]={my_dictionary[1]}")
print(f"my_dictionary[2]={my_dictionary[2]}")

print("\nUsing the make_total_double function now.")
result = make_total_double(20, 30.5)
print(result)
```

Output

```
Accessing some list elements:
my_list[0]=1
```

```
my_list[3]=4

Accessing some dictionary elements:
my_dictionary[1]=Jack
my_dictionary[2]=Kate

Using the make_total_double function now.
101.0
```

You can see that when Python reads the following line:

```
from my_library import my_list, my_dictionary, make_total_double
```

It makes my_list, my_dictionary, and make_total_double(…) available in the current program file. Already this line of code is long. The my_library.py may contain additional functions and variables. So, you may want to import the entire content from the file into your current program. The following example illustrates this case where I make the key changes in bold.

import my_library # imports the whole module.

print("Accessing some list elements:")
print(f"my_list[0]={**my_library.**my_list[0]}")
print(f"my_list[3]={**my_library.**my_list[3]}")

print("\nAccessing some dictionary elements:")
print(f"my_dictionary[1]={**my_library**.my_dictionary[1]}")
print(f"my_dictionary[2]={**my_library**.my_dictionary[2]}")

print("\nUsing the make_total_double function now.")
result = **my_library.**make_total_double(20, 30.5)
print(result)

When you run this program, you see the same output which you saw in the previous demonstration.

POINT TO REMEMBER

You can import the whole module, or you can import a particular function from a module. I have shown you both approaches. If you import the whole module, you need to use the following syntax: module_name.function_name() to access a function from the module. The same comment applies to variables and classes of the module. In Chapter 11, I discuss more on modules.

Using Alias

Consider a typical case when you see a big module name and you need to refer to it many times from different files. Here typing this long name repeatedly becomes a boring activity for you. Consider another case. Suppose you import a module in the current program. But a function in the current program and a function in the module both may have the same name. How can you proceed now? In such a situation, you can use an alias. You can consider the alias as a short name for your function or module.

Let us test a function alias. In the following demonstration, I make an alias for the function make_total_double function. I have chosen the alias name as mtd . This is why you see the following line of code:

```
from my_library import make_total_double as mtd
```

The remaining program is easy to understand. Go through it.

Demonstration 9

For the following demonstration, I’ve created a new file, called using_function_alias.py . Here is the content.

```
from my_library import make_total_double as mtd

print("\nUsing the make_total_double function now.")
print("Here mtd is an alias for make_total_double(...)")
result = mtd(20,30.5)
print(result)
```

Output

When you run the program, you see the following output:

```
Using the make_total_double function now.
Here mtd is an alias for make_total_double(...)
101.0
```

Let us test a module alias. In the following demonstration, I make an alias for the module my_library . I have chosen the alias name as ml . So, you see the following line of code in the upcoming demonstration:

```
import my_library as ml
```

Demonstration 10

For the following demonstration, I’ve created another new file, called using_module_alias.py . Here is the content.

```
import my_library as ml # imports the whole module.
print("Accessing a list element:")
print(f"my_list[0]={ml.my_list[0]}")

print("\nAccessing a dictionary element:")
print(f"my_dictionary[1]={ml.my_dictionary[1]}")

print("\nUsing the make_total_double function now.")
result = ml.make_total_double(20, 30.5)
print(result)
```

Output

When you run the program, you see the following output:

```
Accessing a list element:
my_list[0]=1

Accessing a dictionary element:
my_dictionary[1]=Jack

Using the make_total_double function now.
101.0
```

Quick Discussion on pip

The pip is the standard package manager for Python. You often use it to install the additional packages (or modules) that you do not find in the Python standard library. I give you an example. In Chapter 3, I imported the math module to get functions sqrt(), ceil(),floor() etc. So, I used the following line:

```
from math import *;
```

Now the question may come to your mind: where does this math module reside? You can find it in the Python module index which is at https://docs.python.org/3/py-modindex.html.

But you cannot see the NumPy library here. If you work on machine learning and you use Python, you often need this library. In that case, you need to install this library using the following command:

```
pip install numpy
```

So, you can see that the install command helps you to install a module using pip .

If you are using Python3 >=3.4, you already have pip on your computer. For example, in my Windows system, I can use the following command to check whether pip is installed on my computer:

```
C:\Users\Vaskaran Sarcar>py -m pip --version
pip 19.2.3 from C:\Users\Vaskaran
Sarcar\AppData\Local\Programs\Python\Python38-32\lib\site-packages\pip
(python 3.8)
```

For other operating system commands and to know more about pip you can refer to the online link https://pip.pypa.io/en/stable/installing/ This book targets the domain of beginners and I do not need the outside packages to describe these materials. So, I exclude the detailed discussion of pip in this book.

EXERCISE

E7.1 Can you predict the output of the following code segment?

```
print("Exercise-7.1")


def print_x(x):
    print(x)


def print_y(y):
    print(y)


def main():
    print_x(10)
    print_y(20)


main()
```

E7.2 Can you predict the output of the following code segment?

```
def print_me(x):
    x += 2
    print(x)


def print_me(x):
    x += 3
    print(x)


print_me(5)
```

E7.3 Can you predict the output of the following code segment?

```
def print_me(x):
    x += 2
    print(x)


def print_me(y):
    y += 3
    print(y)


print_me(5)
```

E7.4 Can you predict the output of the following code segment?

```
x = 10
print(f"x={x}")


def print_me(x):
    x += 2
    print(f"Now x={x}")


print_me(x)
print(f"Here x={x}")
```

E7.5 Can you predict the output of the following code segment?

```
def print_me(x):
    print(x)

def print_me(x,y):
    print(x)
    print(y)

print_me(5)
print_me(5,7)
```

E7.6 Suppose you have a list of numbers. Using a lambda function, can you increase each number by 5%?

Solution to Exercises

E7.1

```
10
20
```

Explanation:
This is straightforward. The **main()** function calls the other two functions. So, when you call main(), it calls print_x() and print_y() too.

E7.2
8

Explanation:
Method overloading allows you to use the same name for multiple functions which can be worked differently. We often see it in **C#, Java, C++,** etc. But Python does not this concept by default. If you want the concept, you need to write your code differently.

Python does not show you any error if you use the same name for two functions, but it calls the latest one. In this exercise, both functions exist with the same name print_x()**.** But the last same-named function increments the value of x by 3 . This is why print_x(5) results in 5+3=8

E7.3

```
8
```

Explanation:
Previous explanations can explain this one, too. I have put this exercise for you to note the following fact: *If you only change the method parameter from x to y; it does not mean that you have a different function.*

E7.4

```
x=10
Now x=12
Here x=10
```

Explanation:

The initial value of x is 10 . Then I called print_me() which increments the value to 12 . But notice that this change has happened inside the function body. It means that I worked on a local copy of x . So, when I come out from the function body, I again have the value 10 for x .

We lose the value of a local variable between function invocations. You can use the same local variable for different functions too. In simple terms, the memory of a local variable is used when it is in the scope. When you leave the scope, you free the memory.

Sometimes, you may want to a have variable that does not die before your program ends. In such a case, you can use a global variable. Since global variables are not for any specific functions, you place them outside of all functions. In Python, global is a reserved word. You can use it as follows:

```
print("Example of a global variable.")
x = 10
print(f"x={x}")


def print_me():
    global x
    x += 2
    print(f"Now x={x}")


print_me()
print(f"Here x={x}")
print("-" * 30)
```

Now you can get the following output:

```
Example of a global variable.
x=10
Now x=12
Here x=12
```

E7.5

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter7/chap7_exercise.py",
line 79, in <module>
```

```
    print_me(5)  # error now
TypeError: print_me() missing 1 required positional argument: 'y'
```

Explanation:
I have placed all exercises in the same file, so you see line number 79. This line number can vary if you make separate files for each exercise. The important part is to recognize the error in the pointed area. Here you see the error because I invoke print_me() with one argument.

In exercise E7.2 , you have seen the explanation. If you have multiple functions with the same name, Python considers the latest one. So, in this case, Python wants us to supply two arguments because the latest definition of **print_me()** has two parameters.

E7.6

```
original_list = [100, 200, 300, 400, 500]
# Adds 2 to each item in the list
new_list = list(map(lambda x: x * 1.05, original_list))
print("The original list is as follows:")
print(original_list)
print("The updated list is as follows:")
print(new_list)
```

Output

```
The original list is as follows:
[100, 200, 300, 400, 500]
The updated list is as follows:
[105.0, 210.0, 315.0, 420.0, 525.0]
```

CS 5.1 Implementation

I use many small functions in this implementation. I convert a valid operand into a float before I calculate the result. So, you'll find codes like:

```
usr_input1 = input("Enter first number:")
first_number = float(usr_input1)
```

I store valid operators in a list. Here is the code:

```
valid_operators = ["+", "-", "*", "/"]
```

If the operator is not included in this list, I report an error. I use an if-else chain to do this task:

```
if usr_opr in valid_operators:
    compute(first_number, usr_opr, second_number)
else:
    print("Invalid operator. Cannot compute the result.")
```

The remaining codes are easy to understand. Refer to the supporting comments if you need them. Here is the complete implementation:

```
print("=" * 25)
print("This is a simple calculator.")
print("It supports the following operations:")
print("i)Addition"
      "\nii)Subtraction"
      "\niii)Multiplication and "
      "\niv)Division.")
print("=" * 25)
valid_operators = ["+", "-", "*", "/"]



def add_numbers(num1, num2):
    """
    Adds the numbers.
    """
    return num1 + num2
```

```
def subtract_numbers(num1, num2):
    """
    Subtracts the numbers.
    """
    return num1 - num2


def multiply_numbers(num1, num2):
    """
    Multiplies the numbers.
    """
    return num1 * num2


def divide_numbers(num1, num2):
    """
    Divide num1 by num2.
    """
    return num1 / num2


def compute(num1, operator, num2):
    """
    This function computes the final result.
    """
    result = 0  #default value
```

```
      if operator == '+':
        result = add_numbers(num1,num2)
    elif operator == '-':
        result = subtract_numbers(num1,num2)
    elif operator == '*':
        result = multiply_numbers(num1,num2)
    else:
        result = divide_numbers(num1, num2)
    print(f"The final result is:{result}")


def main():
    """
    This is the top-level function.
    It calls the compute() function.
    """
    usr_input1 = input("Enter the first number:")
    first_number = float(usr_input1)
    usr_input2 = input("Enter the next number:")
    second_number = float(usr_input2)
    usr_opr = input("Enter an operator(+,-,*,/): ")
    if usr_opr in valid_operators:
        compute(first_number, usr_opr, second_number)
    else:
        print("Invalid operator. Cannot compute the result.")
```

```
main()
```

Possible Improvements

- Try to validate all the user inputs when you learn the exception handling mechanism. You’ll see a sample implementation at the end of Chapter 8.
- You can support more operations in your calculator.

CS 5.2 Implementation

Here is an improved and organized solution for CS 4.1 using functions.

```
# All questions
question1 = "Q1.What is the value of the expression:2*3-4?" \
            "\n(a)1" \
            "\n(b)2" \
            "\n(c)3" \
            "\n(d)None."
question2 = "\nQ2.What is the value of the expression:1+(2*3)/4?" \
            "\n(a)1.5" \
            "\n(b)3" \
            "\n(c)3.5" \
            "\n(d)None."
question3 = "\nQ3.The set data type can hold duplicate values." \
            "The statement is:" \
            "\n(a)False" \
            "\n(b)True" \
            "\n(c)Partially correct." \
            "\n(d)None."

# Storing the questions with answer keys
# inside the following dictionary.
question_bank = {question1: "b",
                question2: "c",
                question3: "a"}


def run_test(questions):
    """
        This function takes the question bank as a parameter.
        You can supply the question bank with answer keys
        in this function.
        """
    print("Welcome to the MCQ test.")
    print("=" * 25)
    score = 0  # initial value
    for key in questions:
        print(key)
        user_input = input("Type your answer(a/b/c/d):")
        if user_input == question_bank[key]:
            score += 1
    print(f"\nYour Score:{score} out of {len(questions)}")


run_test(question_bank)
```

Possible Improvements

- Validate all the user inputs when you learn the exception handling mechanism in Chapter 8.

Case Study VI

CS 6.1 Problem Statement

In CS 5.1, you made a calculator that performs the basic operations-addition, subtraction, multiplication, and division. In Chapter8, you learn about exception handling. Upon completion, you should be able to evaluate the user inputs and valid operations. Now I want you to make a better implementation using that knowledge.

The output should not change for valid inputs. But for a negative input, the application should raise exceptions. For example, the application should detect whether you try to divide a number by zero. If so, it raises the exception. I give you three sample outputs to understand it better.

Here is the **sample-1**. A user enters an invalid number. Since the number is invalid, you do not need to ask for an operator. So, you can report the issue immediately.

```
==========================
This is a simple calculator.
It supports the following operations:
i)Addition
ii)Subtraction
iii)Multiplication and
iv)Division.
==========================
Enter the first number:23
Enter the next number:asc
Invalid input.Details: could not convert string to float: 'asc'
```

Here is the **sample-2**. A user enters an invalid operator.

```
==========================
This is a simple calculator.
It supports the following operations:
i)Addition
ii)Subtraction
iii)Multiplication and
iv)Division.
==========================
Enter the first number:12.5
Enter the next number:3
Enter an operator(+,-,*,/): >?
Error details: Invalid operator.
```

Here is the **sample-3**. A user tries to do an invalid operation.

```
=========================
This is a simple calculator.
It supports the following operations:
i)Addition
ii)Subtraction
iii)Multiplication and
iv)Division.
=========================
Enter the first number:27.3
Enter the next number:0
Enter an operator(+,-,*,/): /
Invalid Operation.Details: float division by zero
```

Chapter 8: Exception Management

Coding is fun. You may face continuous challenges and encounter sudden surprises during your program executions. Many of the failures are beyond the control of a programmer. These surprises may occur for various reasons. I list some of them as follows:

- There are some careless mistakes (such as typos) in the program.
- The program logic is incorrect.
- A developer ignores/overlooks the possible loopholes in the code, etc.

Programmers often term these unusual situations as *exceptions*. Handling these exceptions is essential when you write an application. The core concept is not new. It has been around for some time. Python has its own set of exceptions. These are available for your immediate use. You can also write a custom exception to handle a specific situation in your way. This chapter covers a detailed discussion about the exception handling mechanism.

Types of Mistakes

You can broadly classify the mistakes (or errors) in a program in two categories:

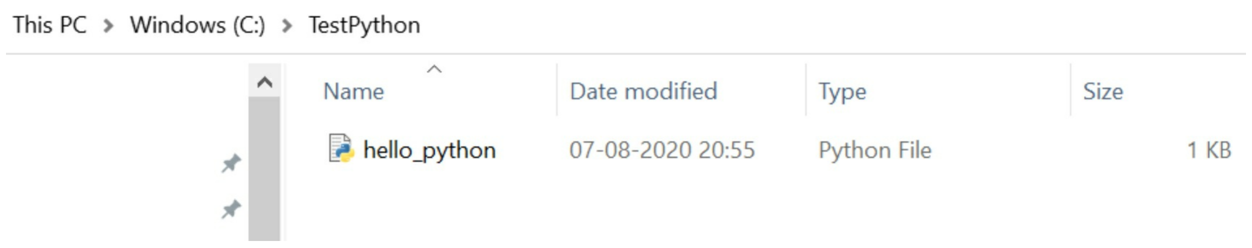
- Before you execute a program, often termed as compile-time errors. (It is important to note that there are some differences between a compiler and interpreter. We often term Python as an interpreted language. It is partially true because there is a step for compilation too. You will see the discussion shortly).
- When the program is running, often termed as run-time errors.

When I say “Before you execute a program”, I mean that some errors are detected by Python in the early development stage. Python acts as your friend to figure out the error details. These errors are simple to detect and fix. For example, it may point out some line number (where the error is encountered) and display a brief description of the error. Once you correct it, you need to recheck whether the updated program is ready for execution or not. Sometimes you need to fix multiple errors. In those cases, we need multiple rechecks.

I told you earlier that “Python is an interpreted language”. This statement is partially true. To elaborate, let me write the following two lines of code in a notepad and save it as hello_python.py :

```
print("Hello Python!")
print("This is Vaskaran.")
```

I put this file in my directory C:\TestClass . So, currently, on my computer I can see this (Refer Figure 8-1):



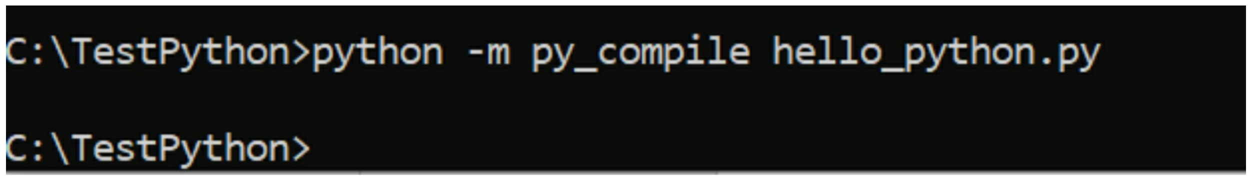
```
This PC > Windows (C:) > TestPython
Name            Date modified       Type          Size
hello_python    07-08-2020 20:55    Python File   1 KB
```

Figure 8-1:The hello_python.py is placed inside C:\TestClass.

Using the command prompt, I enter the directory which contains the python file. Then I execute the following command:

```
python -m py_compile hello.py
```

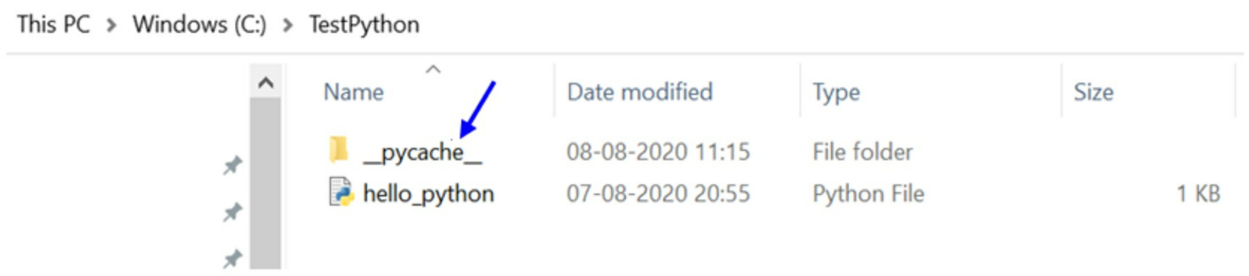
Here is the screenshot (See Figure 8-2):



```
C:\TestPython>python -m py_compile hello_python.py

C:\TestPython>
```

Figure 8-2: The hello_python.py is compiled successfully.

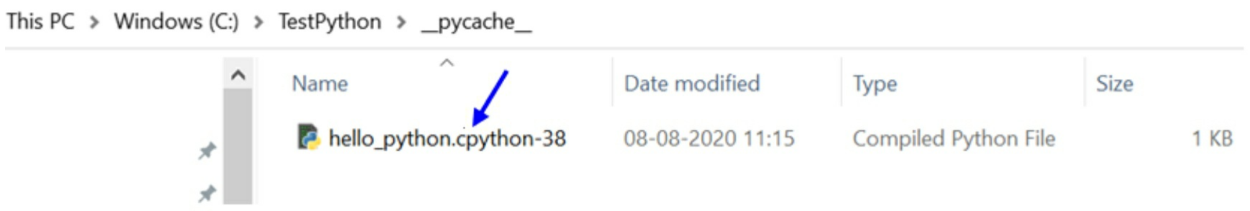
Now, you’ll notice that a folder called __pycache__ is created. (Note that here you see a double underscore)


This PC > Windows (C:) > TestPython

Name	Date modified	Type	Size
__pycache__	08-08-2020 11:15	File folder	
hello_python	07-08-2020 20:55	Python File	1 KB

Figure 8-3: A new folder __pycache__ is created inside C:\TestPython

If you enter this newly created folder, you will see a compiled Python file that has a name similar to hello_python.cpython-38 .You see ‘38’ because I used the Python version is 3.8. So, this number can vary in your case. Notice the following screen (See Figure 8-4):


This PC > Windows (C:) > TestPython > __pycache__

Name	Date modified	Type	Size
hello_python.cpython-38	08-08-2020 11:15	Compiled Python File	1 KB

Figure 8-4: The current contents inside __pycache__

Now you can run this compiled file using the following command from the command prompt:

```
python hello_python.cpython-38.pyc
```

Here is a screenshot for you (See Figure 8-5):

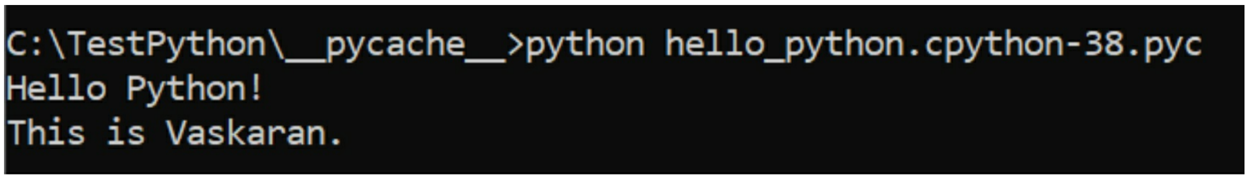

```
C:\TestPython\__pycache__>python hello_python.cpython-38.pyc
Hello Python!
This is Vaskaran.
```

Figure 8-5: Successful execution of the file inside __pycache__ directory

Note that if there is any syntax error in your file, Python does not

generate a compiled file for you. To test this, consider a new file, called hello_python2.py . This file includes some typographical errors. Here is the content of it:

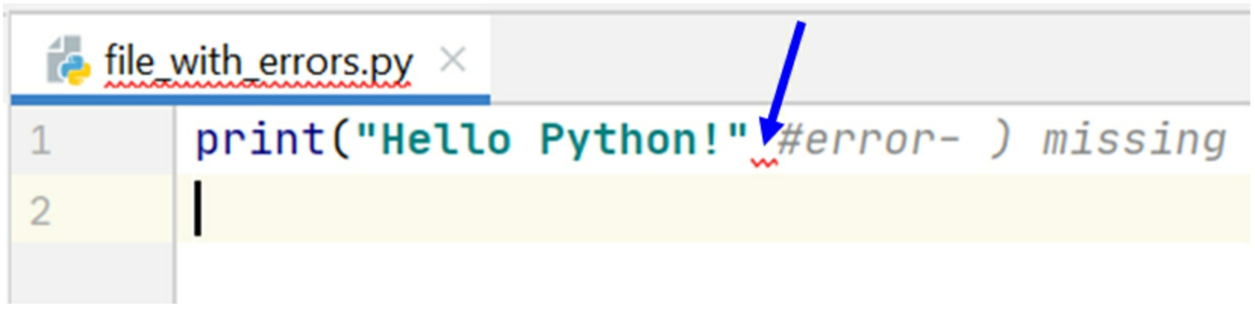
```
print("Hello Python!" #error- ) missing
```

If you try the previous way to generate a compiled file from it, you get some errors. I show the important error segment for your immediate reference.

```
C:\TestPython>python -m py_compile hello_python2.py
Traceback (most recent call last):
  File "C:\Users\Vaskaran
Sarcar\AppData\Local\Programs\Python\Python38-32\lib\py_compile.py",
line 144, in compile
    code = loader.source_to_code(source_bytes, dfile or file,
  File "<frozen importlib._bootstrap_external>", line 846, in source_to_code
  File "<frozen importlib._bootstrap>", line 219, in
_call_with_frames_removed
  File "hello_python2.py", line 2

    ^
SyntaxError: unexpected EOF while parsing

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "C:\Users\Vaskaran
Sarcar\AppData\Local\Programs\Python\Python38-32\lib\py_compile.py",
line 209, in main
    compile(filename, doraise=True)
  File "C:\Users\Vaskaran
Sarcar\AppData\Local\Programs\Python\Python38-32\lib\py_compile.py",
line 150, in compile
    raise py_exc
__main__.PyCompileError:   File "hello_python2.py", line 2

    ^
SyntaxError: unexpected EOF while parsing
```

…

If you use PyCharm IDE, it can point the error easily with the red markers (See Figure 8-6):



```
print("Hello Python!" #error- ) missing
```

Figure 8-6: The PyCharm IDE shows the typographical error.

Move your cursor to the highlighted area. You understand that you forgot to put the closing bracket for the print() function.

This type of error is common, but sometimes hard to find with human eyes at the very beginning. So, a Python compiler or an IDE like PyCharm can help you in these situations.

On the other hand, run-time errors are challenging. Here, you get a green signal from the compiler and everything appears to be fine. But your program still produces wrong results and may terminate prematurely. Here are some examples of run-time errors that can be caused for the following operations:

- Dividing an integer by 0
- Try to convert an invalid string to an Integer
- An expected file is NOT found, or it doesn't exist at all, and so forth.

General Philosophy

The primary purpose of exception handling is to handle errors that arise during program execution. You can use it effectively for many other purposes, too. Programmers are often lazy to analyze all the possible errors in advance. It can make sense for a small application where debugging the code is easy. But consider the case when you have a big application with many small functions. If you need to consider all possible errors in every code segment, the task can be repetitive.

Consider a simple example. Suppose your application computes something and you periodically store the result in a disk. These operations can be interrupted in various ways. Here are some possibilities:

- A user provides one or more invalid inputs.
- At some stage, you encounter something unwanted, say a division by zero operations.
- The disk does not have enough space.
- The disk is not fully functional. So, you cannot save the result.

and so forth.

For the ultimate safety, let’s assume that you check all possible error conditions in advance. Under this method, the error detecting and handling code become a significant portion of the entire program. This code is repetitive and scattered. It is not desirable.

Exception handling mechanisms can act as a bridge between these possibilities. You understand that at a high level, you can consider two imaginable types of errors-one to detect a computational error, and another one to detect disk errors. So, you make two handlers to handle these situations. Then you organize your code in such a way that if any of these errors occur, you refer to the corresponding error handler.

POINT TO REMEMBER

Usually, the problems that can cause an exception are different. For example, an arithmetic problem such as division by 0 differs completely from a memory shortage problem in a disk. Therefore, use a particular handler to manage a specific type of problem.

Common terms

You can define an exception as an event that breaks the normal execution flow of the program during the runtime of a program. Python creates some special objects called exceptions to manage the errors in those situations.

We say that a particular block of code ***raises*** (or throws) an exception. We call the act of responding to an exception ***catching*** an exception. We refer to the code that handles the exception as an ***exception handler***. You can see multiple exception handlers in a program. They help you catch different types of exceptions in the code. It is also possible that different handlers handle the same exception, but in general, they do not live in the same place.

If you have the correct exception handlers, your program continues to run. Otherwise, the program may halt and die prematurely. In these cases, follow the traceback, which includes a detailed report about the situation.

POINTS TO REMEMBER

An exception handling mechanism deals with runtime errors, and if not handled properly, an application will produce unwanted output and it may die prematurely. Therefore, try to write applications that can detect and handle surprises gracefully and prevent the premature death of the application.

Python Exceptions

Python generates an exception object with a raise statement. Once it is raised, Python follows the common exception handling mechanism, i.e. instead of proceeding with the next statement, the current calling chain searches for the handler that can handle the situation. If it finds such a handler, it can access the exception object for more information. Otherwise, the program aborts with an error message.

Python’s exception handling philosophy is a little different from other common programming languages such as Java, for example. In Java, developers check all possible errors at the beginning. It is because handling a run-time exception is a costly operation. We call this style as ***look before you leap*** (LBYL). Python likes to deal with exceptions after they occur. Yes, it is risky. But they encourage you to write code, which is less cumbersome and easily readable. We call this approach "***Easier to Ask for Forgiveness than Permission*** (EAFP).

It is often easier to ask for forgiveness than to ask for permission.

—Grace Hopper, American computer scientist

Python has its own set of exceptions. It has a hierarchical structure. At the time of this writing, Python 3.9 is just released. So, you can get the latest list from the link: http://bit.ly/python-exceptions. Instead of filling the pages with the complete list of exceptions, I prefer to show a subset from it. You can get the idea from the following list:

```
BaseException
     SystemExit
    …
     Exception
     ArithmeticError
     FloatingPointError
     ZeroDivisionError
     AssertionError
     …
     ValueError
     UnicodeError
     …
     …
```

Notice the indentation. It is deliberate. Each exception type is a class. These classes follow the inheritance hierarchy. The hierarchical structure helps us to determine the parent-child relationship. Class, objects, and inheritance make the heart of object-oriented programming (OOP). You may be unfamiliar with OOP. Do not worry, it's ok. I have included a discussion about them in the last part of the book. Following the list, it's enough for you to know that a ZeroDivisionError is one type ArithmeticError which is again one type of Exception . But SystemExit is not an Exception type, but it is a BaseException type. You can get the full meaning of each type from the Python documentation. You can be familiar with them as you continue writing programs.

POINT TO REMEMBER

Python’s exception handling mechanism follows the object-oriented programming (OOP) style. Chapter 10 and Chapter 11 in this book discuss the OOP. You do not need to learn the OOP in detail to understand the material in this chapter.

Demonstration 1

Let us examine the case when you do not handle the situation. You leave the decision to Python to manage the error. The following program seems to be fine and compiles successfully. But it can raise the exception during runtime because of invalid input (s). Let us test this program with some valid and invalid inputs.

This program expects you to supply two valid integers. Then it displays the result of the division. Here is the program.

```
# exception_handling_case_study_1.py

a = input("Enter a valid integer:")
b = input("Enter another valid integer:")
c = int(a)/int(b)
print(f"The result of a/b is:{c}")
print("The program completes successfully.")
```

Case Study with Valid Inputs

You supply two valid inputs 7   and 2   in this case. And everything seems to work fine. Here is the output.

```
Enter a valid integer:7
Enter another valid integer:2
The result of a/b is:3.5
The program completes successfully.
```

Case Study with an Invalid Input

This time you supply 7 and 0 . In this case, the application raises a run-time error. It is because you try to divide 7 by 0. Here are the inputs and corresponding output.

```
Enter a valid integer:7
Enter another valid integer:0
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter8/exception_handling_ca
line 3, in <module>
    c = int(a)/int(b)
ZeroDivisionError: division by zero
```

Notice that Python generates a traceback for you. Following this report, you can identify that at line 3, you have encountered the problem. It is because there is an attempt of a division by zero. Once this error is raised, Python does not execute the next statement anymore. So, you do not see the line " The program completes successfully." in this output. Python names this kind of error as ZeroDivisionError .

Case Study with another Invalid Input

This time you supply abc and 2 . Since you cannot convert abc to a valid integer, the application produces a different run-time error. Here is the output.

```
Enter a valid integer:abc
Enter another valid integer:2
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter8/exception_handling_ca
line 3, in <module>
    c = int(a)/int(b)
ValueError: invalid literal for int() with base 10: 'abc'
```

Following this report, you can identify that this time you have encountered the problem because of an invalid literal for an integer. Python names this kind of error as ValueError . Once this error is raised, Python does not execute the next statement anymore. So, you do not see the line " The program completes successfully." in this output.

Before you read further, I want you to note that an in-built Python function can raise the exception too. Here is such an example. I run this segment in a command shell.

```
>>> mylist=[1,2,3,4]
>>> mylist
[1, 2, 3, 4]
>>> mylist[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

You can raise an exception directly too. Here is an example:

```
>>> raise ArithmeticError("Have Fun!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArithmeticError: Have Fun!
```

Key Points of the Exception-handling Mechanism

In this section, I highlight some key points about the exception handling mechanism. I suggest that you revisit these points as per your need.

- An exceptional object is created when a run-time error occurs. An exception is an object to describe an erroneous situation.
- An application can raise surprises during the application's runtime. If such a situation occurs, in programming terminology, you say that the application has raised an exception, or it has thrown an exception.
- You can guard the situation using a try-except block. You instruct Python on what to do if an exceptional situation occurs. You place the code that may raise an

exception inside a try block. You handle an exceptional situation inside an except block. If the codes inside the try block execute without an exception, the program control bypasses completely the except blocks.

- In the previous demonstration, you have seen that the same program can generate a variety of exceptions. So, you may notice multiple except blocks with a try block.
- When a particular except block handles the sudden surprise (the exception), you say that the except block has caught the exception.
- Let us assume a typical case. In a program, you have the code to handle the ZeroDivisionError , but you do not have except block to handle ValueError . In this case, if the ValueError occurs inside the try block, your program ends prematurely.
- When an exception occurs, Python shows you an error report. You can use this report to know the error location and the details of it.

Demonstration 2

In demonstration2, you'll examine all the key points that we have just discussed. Follow the code now. Again, you see case studies with valid and invalid inputs. This program is a modified version of demonstration1. So, it expects you to supply two valid integers before it displays the result of the division.

```
# try_except_example_1.py

print("The following program can handle the ZeroDivisionError and
ValueError both.")
a = input("Enter a valid integer:")
b = input("Enter another valid integer:")
try:
    result = int(a) / int(b)
    print(f"The result of the division is: {result}")
except ZeroDivisionError as e:
    print("Invalid input! Your divisor becomes zero!")
    print(f"Error details:{e}")
except ValueError as e:
    print("Invalid input! Provide a correct input next time!")
    print(f"Error details: {e}")
print("The program completes successfully.")
```

Case Study with Valid Inputs

You supply 2 valid inputs 7 and 2 in this case. And everything seems to work fine. Here is the output.

```
The following program can handle the ZeroDivisionError and ValueError
both.
Enter a valid integer:7
Enter another valid integer:2
The result of the division is: 3.5
The program completes successfully.
```

Case Study with an Invalid Input

This time you supply 7 and 0. Since the divisor is 0 before the division operation, the application could produce a run-time error, but you have handled it using an except block. Here is the output.

```
The following program can handle the ZeroDivisionError and ValueError
both.
Enter a valid integer:7
Enter another valid integer:0
Invalid input! Your divisor becomes zero!
Error details: division by zero
The program completes successfully.
```

Notice that this time the program ends gracefully. You also see the line “ The program completes successfully.” in this output. It is because the error was caught successfully in an except block. This except block handles the ZeroDivisionError. In this block, you also print the custom messages to the user. When you do this, you provide better security to your application. It is because you do not disclose important details like your program file name, file location, etc. A skilled hacker can do illegal activities using this information.

Case Study with another Invalid Input

This time you supply 5 and abc . Since abc cannot be converted to a valid integer, the application could produce a run-time error. But in this program, I have another except block to handle this type of errors. We call it a ValueError . Here is the output.

The following program can handle the ZeroDivisionError and ValueError both.

```
Enter a valid integer:5
Enter another valid integer:abc
Invalid input! Provide correct input next time!
Error details:invalid literal for int() with base 10: 'abc'
The program completes successfully.
```

Once again, since the program ended gracefully, you see the line “ The program completes successfully.” in this output. Notice that in this case, the error was caught in a different except block. Again, you have done an excellent job to prevent the skilled attacker by not disclosing the important details of the file.

Demonstration 3

In this demonstration, I show you two important case studies. First, you’ll improve demonstration2. Then you’ll examine a case when a user provides different invalid inputs. Let us begin.

I create a new file and name it use_of_try_except_and_else.py .

Notice the program logic in the previous program and consider the division operation again. You understand that if a run-time error occurs during the division operation; the program does not print the result. This logic is correct, but you can beautify your code. It is a better idea to put the result of the division in an else block. It can show whether the code in the try block passes well. And as a next step, the program control enters the else block.

So, in this example, I’ve moved the line: print(f"The result of the division is: {result}") into the else block. Here is the complete demonstration which can generate the same output.

```
# use_of_try_except_and_else.py

print("The following program can handle the ZeroDivisionError and
ValueError both.")
a = input("Enter a valid integer:")
b = input("Enter another valid integer:")
try:
    result = int(a) / int(b)
    #print(f"Result of the division is: {result}")
except ZeroDivisionError as e:
    print("Invalid input! Your divisor becomes zero!")
    print(f"Error details:{e}")
except ValueError as e:
    print("Invalid input! Provide a correct input next time!")
    print(f"Error details:{e}")
else:
    print(f"The result of the division is : {result}")
print("The program completes successfully.")
```

Case Study with Valid Inputs

Let us test the program now. Supply two valid inputs: 15 and 3. Everything seems to work fine. Here is the output.

```
The following program can handle the ZeroDivisionError and ValueError both.
Enter a valid integer:15
Enter another valid integer:3
The result of the division is: 5.0
The program completes successfully.
```

Case Study with Invalid Inputs

Let us pass abc and 0 this time to generate a run-time error. Notice that both inputs are invalid. Here is the output.

```
The following program can handle the ZeroDivisionError and ValueError both.
Enter a valid integer:abc
Enter another valid integer:0
Invalid input! Provide correct input next time!
Error details:invalid literal for int() with base 10: 'abc'
The program completes successfully.
```

Analysis

Here, both the inputs are invalid. The first one is abc, which you cannot convert to a valid integer. The second input is 0. It makes the divisor zero. But the program handled the first issue, which is a ValueError .Once you fix this error, you can see the next error. For example, supply 7 and 0 this time to get the following output:

```
The following program can handle the ZeroDivisionError and ValueError both.
Enter a valid integer:7
Enter another valid integer:0
Invalid input! Your divisor becomes zero!
Error details: division by zero
The program completes successfully.
```

You can see that the program handles the error which appears first. In this context, the next section is very important. Read it carefully.

POINTS TO REMEMBER

When you assume that a particular segment of code may raise a run-time error or exception, you place that segment of code into a try block. To handle a specific run-time error, you place an appropriate **except** block to handle the error, print user-friendly messages and suppress important details to prevent malicious attacks. Finally, if there are codes that depend on the successful completion of the **try** block, you place them into the **else** block. In short, keeping this information in mind, you can design a **try-except-else** block.

Use of pass Statement

Sometimes you do not want users to see the exception details and allow the program to fail silently if a typical error occurs. You may want this to give the user a feel that everything is fine. For example, the upcoming program calculates the aggregate of two valid integers. If the user passes any invalid input, you catch the exception, but silently skip the exception details to the user using the pass statement. The following demonstration describes such a scenario.

Demonstration 4

Here is the complete demonstration.

```
# use_of_pass.py

print("This program prints the sum of two valid integers.")

first_input = input("Enter a valid integer:")
```

```
second_input = input("Enter another valid integer:")
total = 0  #default value
try:
    a = int(first_input)
    b = int(second_input)
except ValueError as e:
    pass
else:
    total = a + b
    print(f"Sum of numbers:{total}")

print("The program completes successfully.")
```

Case Study with Valid Inputs

You supply two valid inputs 27 and 5 in this case. And everything seems to work fine. Here is the output.

```
This program prints the sum of two valid integers.
Enter a valid integer:27
Enter another valid integer:5
Sum of numbers:32
The program completes successfully.
```

Case Study with an Invalid Input

Let us test the code with one set of invalid output too. Pass 27 and abc this time to generate a run-time error. Here is the output.

```
This program prints the sum of two valid integers.
Enter a valid integer:27
Enter another valid integer:abc
The program completes successfully.
```

Analysis

This time, you do not see the sum; the program logic tells us that if everything goes well inside the try block, you calculate the aggregate of the valid integers inside the else block. But it is NOT the case here, because you have supplied a string abc which is not a valid integer.

Arranging Multiple except Blocks

In a program, you can expect different errors, and you guard them with appropriate except blocks. But you may not anticipate everything in advance. So, you may want to have a general except block that can handle remaining exceptional situations. For example, in demonstration2, you have handled both ZeroDivisionError and ValueError , but what happens if there is a different exception occurs in your program? In such a case, you can use a general except block to catch the remaining errors. Here is such an example when you see the following code:

```
except Exception as e:
    print("An unknown error occurred.")
```

I suggest that you add this block to your code. Ideally, it should be placed, after all the anticipated except block, something like the following:

```
try:
    #Some code
except ZeroDivisionError as e:
    print(f"Error details:{e}")
except ValueError as e:
    print(f"Error details:{e}")
except Exception as e:
    print(f"Error details:{e}")
```

This arrangement is important. In the last block, you use the Exception class, which is a common base class for non-system-exiting exceptions. You may not know about “class” at this time. It is ok. You can understand the key thing: if you place a more generic or broader except block before a specific except block, your code will NOT reach the specific except block.

Let us have a test in the following demonstration. First, I’ve placed the following block before other except blocks:

```
except Exception as e:
        print("An unknown error occurred.")
```

Here is the complete program where the except block arrangement is not proper.

# try_with_multiple_except_example2.py

```
print("***The following program can handle the non-system-exiting
exceptions.***")
a = input("Enter a valid integer:")
b = input("Enter another valid integer:")
try:
    result = int(a) / int(b)
except Exception as e:
    print("An unknown error occurred.")
    print(f"Error details: {e}")
except ZeroDivisionError as e:
    print("Invalid input! Your divisor becomes zero!")
    print(f"Error details: {e}")
except ValueError as e:
    print("Invalid input! Provide a correct input next time!")
    print(f"Error details: {e}")
#except Exception as e:
    #print("An unknown error occurred.")
```

```
else:
    print(f"Result of the division is : {result}")
print("The program completes successfully.")
```

Now run the program against the invalid inputs that you saw in demonstration2.

Case Study with an Invalid Input

First, you supply 7 and 0. Here is the output.

```
***The following program can handle the non-system-exiting exceptions.***
Enter a valid integer:7
Enter another valid integer:0
An unknown error occurred.
Error details: division by zero
The program completes successfully.
```

Case Study with another Invalid Input

This time you supply 5 and abc . Since abc cannot be converted to a valid integer, the application could produce a run-time error, and here is the output.

```
***The following program can handle the non-system-exiting exceptions.***
Enter a valid integer:5
Enter another valid integer:abc
An unknown error occurred.
Error details:invalid literal for int() with base 10: 'abc'
The program completes successfully.
```

In both cases, the program terminates gracefully, and you see the line “ An unknown error occurred . ” in the output. You can see that the top-level except block was able to handle both kinds of errors. But you are unable to see the specific messages like: “ Invalid input! Your divisor becomes zero!”, or “ Invalid input! Provide a correct input next time!" in the output.

If you want to see those specific messages, you need to place the except blocks properly. For example, here is a correct arrangement:

```
#previous codes
except ZeroDivisionError as e:
   print("Invalid input! Your divisor becomes zero!")
   print(f"Error details:{e}")
except ValueError as e:
   print("Invalid input! Provide a correct input next time!")
   print(f"Error details:{e}")
except Exception as e:
   print("An unknown error occurred.")
```

Now if you test your program against the same inputs, you'll get the similar output that you saw in demonstration2.

POINTS TO REMEMBER

When you deal with multiple except blocks, you need to place more specific except blocks first. In other words, you should place the except blocks from most specific to most general.

Custom Exception

You can create a custom exception using the following code segment:

```
class MyException(Exception):
   pass
```

Let us use it now. The following program asks for user input. To make the example short and simple, I assume that the user needs to supply an integer that is less than or equal to 100. Otherwise, the program raises this custom exception, called MyException .

Demonstration 5

Here is the complete demonstration.

# custom_exception.py

```
#Following example uses a custom exception
```

```
class MyException(Exception):
    """ This is a custom exception."""
    pass


try:
    user_input = int(input("Enter an integer which is not greater than 100:"))
    if user_input > 100:
        raise MyException("You have made the integer greater than 100.")
    print(f"Well done.You have entered:{user_input}")
except MyException as e:
    print(f"Custom exception is raised.Error Details:{e}")
except ValueError as e:
    print(f"Here is the error Details:{e}")
```

Output

Here is a sample output. This time user supplies an integer less than 100.

```
Enter an integer which is not greater than 100:23
Well done.You have entered:23
```

Here is another sample output. This time user supplies an integer that is greater than 100.

```
Enter an integer which is not greater than 100:102
Custom exception is raised.Error Details:You have made the integer greater than 100.
```

Here is another sample output. This time user does not supply an integer.

```
Enter an integer which is not greater than 100:abc
Here is the error Details:invalid literal for int() with base 10: 'abc'
```

Debugging Code Using assert Statement

The assert statements help you debug your code. Using the assert keyword, you can test a condition. If the condition is false, the program raises an AssertionError . Here is a simple example for you.

Demonstration 6

In this example, I have a list called my_list . It contains 4 elements. Now I test whether the length of the list is greater than 5. This condition returns False . Since I have used the assert statement, an AssertionError is raised. It also displays the supporting message in the output. This supporting message is optional for you.

```
my_list = [1, 2, 3, 4]
#If the condition is False, AssertionError is raised.
assert len(my_list) >= 5, "List length is less than 5."
```

Output

Here is a sample output.

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter8/using_assert.py", line
3, in <module>
    assert len(my_list) >= 5, "List length is less than 5."
AssertionError: List length is less than 5.
```

Demonstration 7

You can use try-except to handle this exception. Otherwise, you have seen that the program will halt and produce a traceback. The following code segment shows the use of try-except to handle the AssertionError .

```
# using_assert.py
my_list = [1, 2, 3, 4]

try:
   assert len(my_list) >= 5, "List length is less than 5."
except AssertionError as e:
    print(f"Error details: {e}")
```

Output

Here is the output.

```
Error details: List length is less than 5.
```

Analysis

In demonstration 6 and demonstration 7, I assume that system variable __debug__ is True by default. You can turn it off. For example, you can use -O (O is in the capital) or -OO in the command line. The -O flag turns on the basic optimization. The -OO discards docstrings too, besides basic optimizations. Here is a sample:
Since 2 is always less than 3, you get the AssertionError if you write :

```
>>> assert 2>3
```

Here is the output.

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Now I use -O flag.

```
C:\Users\Vaskaran Sarcar>python -O
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> assert 2>3
>>>
```

Notice that you do not see the AssertionError this time.

There is an alternative way. You can set an environment variable to set this flag. Here is a sample. I do this in a Windows10 system.

C:\Users\Vaskaran Sarcar>**SET PYTHONOPTIMIZE=TRUE**

```
C:\Users\Vaskaran Sarcar>python
```

```
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> assert 2>3
>>>
```

Again, you do not see the AssertionError . It is because you made it off already. You can reset it back. Here is the sample:

```
>>> exit()

C:\Users\Vaskaran Sarcar>SET PYTHONOPTIMIZE=
C:\Users\Vaskaran Sarcar>python
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:20:19) [MSC v.1925 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> assert 2>3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

I hope that you have enjoyed this chapter. Now it’s time for exercises and projects.

EXERCISE

E8.1 Predict the output when you run the following code.

```
try:
    result = 15/0
except ArithmeticError:
    print("Caught the ArithmeticError.Your divisor is zero.")
except ZeroDivisionError:
    print("Caught ZeroDivisionError.Correct the divisor which is zero at present.")
```

E8.2 Predict the output

```
my_string="Hello"
assert my_string== "Hi", "You should use 'Hi'"
```

E8.3 Predict the output

```
try:
    raise BaseException('BaseException raised.')
```

```
except Exception as e:
    print(f"Error Details:{e}")
```

E8.4 Write a program in which a user can continue entering integers. The program should continue, even if the user provides an invalid input. The user can type ‘q’ to quit the program.

Solution to Exercises

E8.1

Caught the ArithmeticError.Your divisor is zero.

Explanation:
ArithmeticError is a built-in exception. It can handle the exceptions when the code raises an arithmetic error. The built-in exceptions like ZeroDivisionErro r, FloatingPointError also belong to this category.

For example, you can think of a Vehicle as the base class for both Bus and Train . So, when you talk about a bus or train, you are talking about a specific vehicle. Similarly, ZeroDivisonError class is a specific type of ArithmeticError class.

You may not know the concept of class/object at this moment. But you can know a simple fact. It says that both ArithmeticError and ZeroDivisionError could handle the error. But I placed AirthmeticError before ZeroDivisonError . So, we catch the error inside the ArithmeticError block. If you change their order, as follows:

```
try:
   result = 15/0
except ZeroDivisionError:
   print("Caught ZeroDivisionError.Correct the divisor which is zero at
present.")
except ArithmeticError:
   print("Caught the ArithmeticError.Your divisor is zero.")
```

You can see a different output:

Caught ZeroDivisionError.Correct the divisor which is zero at present.

I have shown you how to arrange multiple except blocks in your code. You can read that section again.

E8.2

This code segment raises the AssertionError if my_string is not Hi.(I assume that system variable __debug__ is True in this case)

```
>>> my_string="Hello"
>>> assert my_string== "Hi", "You should use 'Hi'"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: You should use 'Hi'
```

E8.3
This catch block does not catch BaseException . It is because BaseException is the parent class of Exception . Here is a sample output:

```
Traceback (most recent call last):

  File "E:/MyPrograms/Python/PythonCrashCourse/chapter8/exercise8.3.py",
line 2, in <module>

    raise BaseException("BaseException raised.")

BaseException: BaseException raised.
```

E8.4
Here is a sample program.

```
print("Exercise-8.4")
print("*" * 20)


def test_me():
    flag = True

        while flag:
        user_input = input("Keep entering a valid integer.(Type q to quit):")
        if user_input == 'q':
            break
        try:
            display_me = int(user_input)
            print(f"Correct.You entered:{display_me}")
        except Exception as e:
            print(f"Error:{e}")
        #This statement is placed outside the while loop
    print("End of the exercise.")


test_me()
```

Here is a sample output:

```
Exercise-8.4
********************
Keep entering a valid integer.(Type q to quit):12
Correct.You entered:12
Keep entering a valid integer.(Type q to quit):342
Correct.You entered:342
Keep entering a valid integer.(Type q to quit):-5
Correct.You entered:-5
Keep entering a valid integer.(Type q to quit):2.3
Error:invalid literal for int() with base 10: '2.3'
Keep entering a valid integer.(Type q to quit):abc
Error:invalid literal for int() with base 10: 'abc'
Keep entering a valid integer.(Type q to quit):56
Correct.You entered:56
Keep entering a valid integer.(Type q to quit):q
End of the exercise.
```

CS 6.1 Implementation

Here is the solution to CS 6.1. You can consider it as an improved solution for CS 5.1 because it can report a wide range of invalid inputs.

```
print("=" * 25)
print("This is a simple calculator.")
print("It supports the following operations:")
print("i)Addition"
      "\nii)Subtraction"
      "\niii)Multiplication and "
      "\niv)Division.")
print("=" * 25)
valid_operators = ["+", "-", "*", "/"]


def add_numbers(num1, num2):
    """
    Adds the numbers.
    """
    return num1 + num2


def subtract_numbers(num1, num2):
    """
    Subtracts the numbers.
    """
    return num1 - num2


def multiply_numbers(num1, num2):
    """
    Multiplies the numbers.
    """
    return num1 * num2



def divide_numbers(num1, num2):
    """
    Divide num1 by num2.
    """
    return num1 / num2



def compute(num1, operator, num2):
    """
    This function computes the final result.
    """
    result = 0  #default value

        if operator == '+':
        result = add_numbers(num1, num2)
    elif operator == '-':
        result = subtract_numbers(num1, num2)
    elif operator == '*':
        result = multiply_numbers(num1, num2)
    #elif operator == "/":
    else:
        result = divide_numbers(num1, num2)
    print(f"The final result is:{result}")
```

```
def main():
    """
    This is the top-level function.
    It calls the compute() function.
    """
    try:
        usr_input1 = input("Enter the first number:")
        first_number = float(usr_input1)
        usr_input2 = input("Enter the next number:")
        second_number = float(usr_input2)
        usr_opr = input("Enter an operator(+,-,*,/): ")
        if usr_opr not in valid_operators:
            raise Exception("Invalid operator.")
        compute(first_number, usr_opr, second_number)
    except ZeroDivisionError as e:
        print(f"Invalid Operation.Details: {e}")
    except ValueError as e:
        print(f"Invalid input.Details: {e}")
    except Exception as e:
        print(f"Error details: {e}")


main()
```

Case Study VII

CS 7.1 Problem Statement
This time you make a report card of students. Assume that a student joins a beginner's course in Python. To get a grade card, a student needs to meet the following criteria:

- He needs to submit two assignments (50 marks each) before he appears in the final examination (which is 100 marks).
- You calculate the final score based on these assignment scores and the final examination score. Consider 25% of total marks in assignment and 75% marks in the final examination to prepare the grade card. When a student scores more than 90, he gets A+, which means Outstanding. If he scores more than 80, he gets A, which means Very Good. If the score is 70 or above, he gets B, which means Good. Consider a score that is less than 70 as Fail.
- Try to manage your application in a better way. For example, if any of the assignment scores is greater than 50, your application should report an error. The same rule applies if anyone inputs a final examination score greater than 100.
- You save these student records in text files, which have names like student_name.txt. You store these files in a separate folder. You can name it *GradeScores*.

I give you three sample outputs to understand it better.

Here is the **sample-1**. A user enters the valid inputs.

```
Enter the student name:Ravi S
Assignment-1 score:25.5
```

```
Assignment-2 score:34.5
Exam score:87
Final score:80.25
Grade: A(Very Good)
The current working directory:
E:\MyPrograms\Python\PythonCrashCourse\Projects
Get the report card at:
E:\MyPrograms\Python\PythonCrashCourse\Projects\GradeScores\Ravi S.txt
```

Here is a sample report card that you store. The content of the text file Ravi S.txt may look like the following:

```
***Report Card***
***Course name: Python for Beginners***
==================================================
Student Name:Ravi S
Assignment-1 Score:25.5
Assignment-2 Score:34.5
Exam Score:87.0
Final score:80.25
Grade/Remark: A(Very Good)
```

Here is the **sample-2**. A user enters an assignment score that is above 50:

```
Enter the student name:John
Assignment-1 score:23
Assignment-2 score:51.5
Error: Assignment2 score cannot be greater than 50.
Provide the correct input next time!
```

Here is the **sample-3**. A user enters the final examination score that is above 100:

```
Enter the student name:Kate
Assignment-1 score:23
```

```
Assignment-2 score:45.3
Exam score:105
Error: Final exam score cannot be greater than 100.
Provide the correct input next time!
```

Chapter 9: Programming with Files

The first question that may come to your mind, why do you need a file? There is a simple answer to this question. If you process an enormous amount of data, supplying inputs one-by-one is not a pleasant idea. Also, taking one input at a time from the user makes the program execution slow. So, a better approach is you store all these inputs in a file. Then you let your program read the date from it directly. Besides, it is useful to write a detailed log into a file for further processing. This chapter discusses these kinds of basic operations.

Reading from a File

Let us read from a text file. I create a text file and save it as OriginalFile.txt in my preferred location C:\TestData. Here is the content of the file:

Dear Reader, this is my test file.
It is originally stored at C:\TestData in my system.
I hope you'll enjoy learning.

Now I have the file; so, I can read its content using a Python program. In the upcoming demonstration, I show you two different approaches. You can use them to read the data from a file. The first one is easy to understand, but not convenient. The second one is a better approach where I use a for loop to read the content of the file. Before you see the demonstration, read the following content.

Quick Talk about File Paths

Experts suggest that we organize the code properly. The current working directory and the files you use -rarely reside in the same directory. So understanding the file paths is important.

If the python program and the files you use, stay at the same location, you do not need to mention the file location. In my case, they stay at different locations. So, I mention the full path of a saved file, called OriginalFile.txt. I use the double backslashes (\\) inside the open() function as follows: 'C:\\TestData\\OriginalFile.txt'

It's worth remembering that we use single backslashes are for escape characters in Python.

POINT TO REMEMBER

Windows systems use the backslash (\) instead of a forward slash (/) to display file paths. But you can use forward slashes too in your program.

You may see the use of both ***absolute paths*** and ***relative paths***. Absolute paths talk about the exact location of a file. For example, in my windows system, C:\TestData\AnotherDirectory is an example of an absolute path. Relative paths talk about the position relative to some other place in a file system. For example, you can use TestData\AnotherDirectory as a relative path. Developers often append a relative path to an absolute path to form a new absolute path. Consider an example. Suppose you append AnotherDirectory\SubDirectory to the absolute path C:\TestData . This activity gives you get a new absolute path C:\TestData\AnotherDirectory\SubDirectory

The directory location in which your Python program resides is your current working directory. To know about a current working directory, you can use the following code:

```
import os
print(f"The current working directory is: {os.getcwd()}")
```

For example, when I run this code, I get the following output:

```
The current working directory is:
E:\MyPrograms\Python\PythonCrashCourse\chapter9
```

Now I can append a relative path to this location to make a new absolute path. The following code segment shows how to retrieve a current working directory before you append a relative path to it.

#get_current_working_directory.py

```
import os

current_path = os.getcwd()
print("The current working directory is as follows:")
print(current_path)

relative_path = "AnotherDirectory\\SubDirectory"
new_path = os.path.join(os.getcwd(),relative_path)
print("The new path is as follows:")
print(new_path)
```

I ran this program and got the following output:

```
The current working directory is as follows:
E:\MyPrograms\Python\PythonCrashCourse\chapter9
The new path is as follows:
E:\MyPrograms\Python\PythonCrashCourse\chapter9\AnotherDirectory\SubD
```

I hope you have got the difference between an absolute path and a relative path now. Let us concentrate on the upcoming demonstration now.

In the upcoming program, I use the following line:

file_object = open('C:\\TestData\\OriginalFile.txt','r')

Notice that I have passed two arguments inside the open() function. I use these arguments to mention the file location and the mode of the operation. The open() function returns an object which I store in file_object . I use this file_object for further processing. The second argument is optional. In this chapter, I've used the following modes. These are very common.

- **'r' :** used for reading. The file pointer stays at the beginning of the file.
- **'w' :** used for writing. If the file doesn't exist, Python creates the file for you. Otherwise, it deletes the contents of the existing file.
- **'a' :** used to append the data at the end of the file. The file pointer stays at the end of the file. If the file doesn't exist, Python creates the file for you.
- **'r+' :** used for both reading and writing. The file pointer stays at the beginning of the file.

- In demonstration4, I use **rb** and **wb** modes as well. These are like **r** and **w** modes, but the only difference is that you use them for binary files.

In other people's code, you can see the use of **w+,** and **a+** modes as well. These have some special capabilities. For example, if the file does not exist, ' **a** ' mode creates a file for writing, but you can use **'a+'** mode both for reading and writing. In short, you use **'+ '** for both reading and writing. Apart from these common modes, you can see ' **x** ' for exclusive creation (if file exists), **'t'** for text mode. For backward compatibility, there is **'U'** mode. You can have a quick look at https://docs.python.org/3.3/library/functions.html#open to know more about them. You can remember these modes easily when you use them in your code.

Next, you see the following code segment:

```
print("Reading each line one by one:")
first_line = file_object.readline()
second_line = file_object.readline()
third_line = file_object.readline()
```

This code segment shows that using the function readline(), you can read a complete line from a file. Inside OriginalFile.txt, there are three lines. So, I use readline() three times to print the content.

Before I move to approach2, I want you to note that readline() inserts a line break after each line. So, when you use this function in your output, you can notice a line gap between these lines. Here is the sample:

```
Dear Reader, this is my test file.

It is originally stored at C:\TestData in my system.

I hope you'll enjoy learning.
```

Now notice the use of the keyword argument end . It becomes available in Python3. You saw an example of this function in Chapter 5 as well. The end= tells what you should print after a function processes other argument(s).

For example, if I write the following code:

```
file_object = open('C:\\TestData\\OriginalFile.txt', 'r')
```

```
for current_line in file_object:
    print(current_line, end='**')
```

I’ll get the following output:

```
Dear Reader, this is my test file.
**It is originally stored at C:\TestData in my system.
**I hope you'll enjoy learning.
**
```

In the upcoming example, I use the following code:

```
for current_line in file_object:
    print(current_line, end='')
```

If you execute this code segment, you notice the following output without the line breaks:

```
Reading entire file using for loop:
Dear Reader, this is my test file.
It is originally stored at C:\TestData in my system.
I hope you'll enjoy learning.
```

I show you another approach. Here I read the entire content of the file using the read() method. This method can read the entire content of the file and store it in a long string. Finally, I print the content. So, you see the following code in approach3:

```
file_content = file_object.read()
print(file_content)
```

Now go through the following demonstration and output.

Demonstration 1

Here is the complete program.

```
# reading_text_file_example_1.py

file_object = open('C:\\TestData\\OriginalFile.txt', 'r')
print("---Approach:1---")
print("Reading each line one by one:")
```

```
first_line = file_object.readline()
second_line = file_object.readline()
third_line = file_object.readline()
print(first_line)
print(second_line)
print(third_line)
#print(first_line, end='')
#print(second_line, end='')
#print(third_line, end='')
file_object.close()

print("---Approach:2---")
print("Reading entire file using for loop:")
file_object = open('C:\\TestData\\OriginalFile.txt', 'r')
for current_line in file_object:
    print(current_line, end='')
file_object.close()

print("---Approach:3---")
file_object = open('C:\\TestData\\OriginalFile.txt', 'r')
print("Using the read() method.")
file_content = file_object.read()
print(file_content)
```

Output

Here is the output.

```
---Approach:1---
Reading each line one by one:
Dear Reader, this is my test file.

It is originally stored at C:\TestData in my system.

I hope you'll enjoy learning.

---Approach:2---
Reading entire file using for loop:
```

```
Dear Reader, this is my test file.
It is originally stored at C:\TestData in my system.
I hope you'll enjoy learning.

---Approach:3---
Using the read() method.
Dear Reader, this is my test file.
It is originally stored at C:\TestData in my system.
I hope you'll enjoy learning.
```

Analysis

From the output, you can see that in approach1, you get a line break between the lines. It is because readline() inserts a line break after each line. But now you know the use of keyword end ; so, if you write the following segment:

```
print(first_line, end='')
print(second_line, end='')
print(third_line, end='')
```

You'll not see the line breaks between the lines in the output.

POINTS TO REMEMBER

I use open() and close() in the demonstration1. You know that I use them to open the file and close the file. This approach suffers from a potential drawback. If there is a bug that prevents the close() method to execute, the file may never close. In that case, the data can be lost or corrupted. Also, as a precautionary step, if you close the file early, you may work with a closed file. It also causes unexpected results. So, if you are unsure when to close a file, it's a better practice to leave the decision to Python. You can use the 'with' keyword in this context. You'll see a demonstration on this topic later in this chapter. Once you learn this approach, I recommend you to follow the same in your code.

Writing to a File

In demonstration1, you see how to open and read from a text file. In this section, let’s write something to the file.

In the upcoming example, I use the same file, OriginalFile.txt. This time I append a few more lines into it.

Demonstration 2

Here is the program for you.

```
# writing_to_file.py

# Opening the file in 'a' mode
my_file = open("C:\\TestData\\OriginalFile.txt", "a")
my_file.write("This is a new line.")
my_file.write("\nThis is another new line.I append this too.")
my_file.close()
```

Output

After you execute this program, go to the file location and open the file to verify the result. Here is the output.

```
Dear Reader, this is my test file.
It is originally stored at C:\TestData in my system.
I hope you'll enjoy learning.
This is a new line.
This is another new line.I append this too.
```

Analysis

Each time you execute the program, these lines are added. Notice that I’ve added the escape character **\n** before the last line, which I print. So, you see this line starts from a new line.

Alternative Way to Read and Write

Demonstration1 and demonstration2 show you how to open a file, read from a file, write to a file, and finally close the files. In the approach3 of demonstration1, you saw the usage of the read() method. It has an optional parameter to show the buffer size. By default, it is -1 , which means the entire file. So, in that example, I did not pass any argument inside this method. In the upcoming demonstration, you see an alternative usage of it. Here I specify the buffer size. This is useful when you care about memory resources. Consider a case: If you load the entire content of an extensive file at once, you can face a memory shortage problem.

In the upcoming example, I use the same file, OriginalFile.txt to read the data from it. But this time I specify the buffer size. So, you see the following line of code:

```
content=input_file.read(15)
```

It tells you that I read 15 bytes at a time. To verify this, while writing, I use the following line of code:

```
output_file.write(content+ "\n")
```

So, when you get the output, you see that each line contains a maximum of 15 characters. If you remove the escape character **\n** while writing the lines in the output file, you see no difference between the input and output files.

Demonstration 3

The previous section describes the most important characteristic of this program. The remaining code is easy to understand. In brief, I want you to note the following points:

I read from a file and write to a different file. So, I need two files to open and close. The input_file represents my input file, and the output_file represents my output file.

I could use r+ mode here. It is because, in the specified location, I already had a file called OutputFile.txt . If in your case, there is no such file, you'll encounter errors something like the following:

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/Chapter9/reading_writing_using
line 3, in <module>
    output_file= open('C:\\TestData\\OutputFile.txt','r+') #error if the file
```

```
doesn't exist
FileNotFoundError: [Errno 2] No such file or directory:
'C:\\TestData\\OutputFile.txt'
```

In that case, I suggest you using **w** mode (as shown in the commented line). You have already seen the use of **r** and **w** mode. So, this time I wanted you to show the use of **r+** mode. This is why I use this mode in this program.

Here is the complete program for you.

```
# reading_writing_using_buffer_size.py

# Opening two files-one for reading and one for writing
input_file = open('C:\\TestData\\OriginalFile.txt', 'r')
# Will NOT work if the file does not exist
output_file= open('C:\\TestData\\OutputFile.txt','r+')
# Will work even if the file does not exist
#output_file = open('C:\\TestData\\OutputFile.txt', 'w')  content =
input_file.read(15)
while len(content):
    output_file.write(content)
    #output_file.write(content + "\n")
    content = input_file.read(15)
input_file.close()
output_file.close()
```

Output

Execute this program. Then go to the output file location and open the file to verify the result. Here is the output.

```
Dear Reader,thi
s is my test fi
le.
It is origi
nally stored at
C:\TestData in
my system.
```

```
I h
ope you'll enjo
y the learning.
```

Analysis

As said before, if you avoid the newline character (\n) and use the following line:

```
output_file.write(content)
```

instead of using the following line:

```
output_file.write(content+ "\n")
```

you can see the same content as in the input file which is as follows:

```
Dear Reader, this is my test file.
It is originally stored at C:\TestData in my system.
I hope you'll enjoy learning.
```

You may not notice the changes made to a file until you close it. So, I suggest that you close the file before you make any changes to it.

Working with Binary Files

Till now I was working with simple text files. Let us now work with a binary file. A binary file is a non-text file such as images or videos. To do reading from a binary file, I store a file called MyImage.png inside the location C:\TestData

In the upcoming example, you'll see the following lines of code:

```
input_file = open("C:\\TestData\\MyImage.png", "rb")
output_file = open("C:\\TestData\\OutputImage.png", "wb")
```

This segment is almost like the previous demonstration. The only difference is that this time I've used ' **rb'** and ' **wb'** instead of **'r'** and **'w'** (or **'r+'**). Also, no escape character is present when I write to the output file.

Demonstration 4

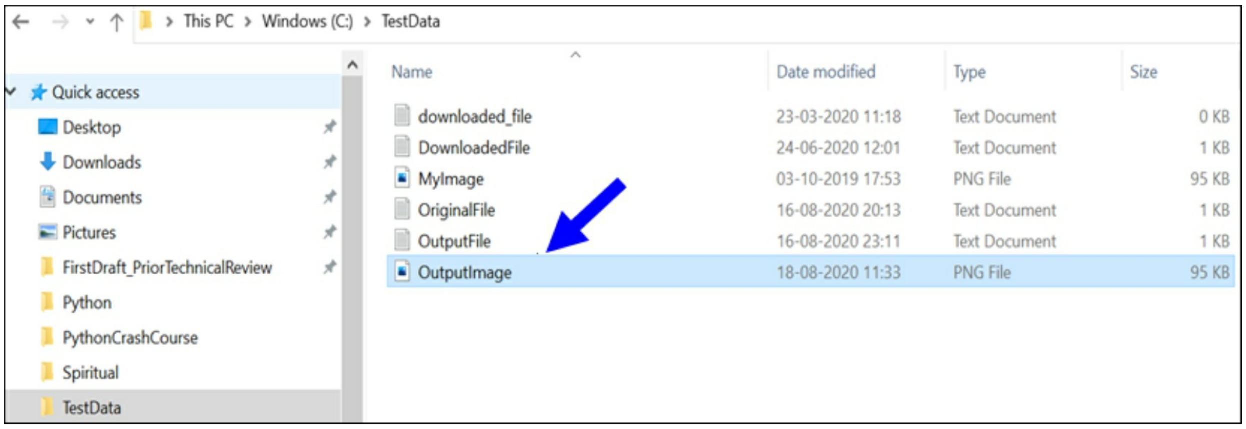
Here is the complete program for you.

# working_with_binary_files.py

```
# Opening two files-one for reading and one for writing
input_file = open("C:\\TestData\\MyImage.png", "rb")
output_file = open("C:\\TestData\\OutputImage.png", "wb")
content = input_file.read(15)
while len(content):
    output_file.write(content)
    content = input_file.read(15)
input_file.close()
output_file.close()
```

Output

Execute the program and go to the output file location. You can see a new image called OutputImage.png which is a duplicate of the original image. I'm taking a partial snapshot from my machine to show this:



Name	Date modified	Type	Size
downloaded_file	23-03-2020 11:18	Text Document	0 KB
DownloadedFile	24-06-2020 12:01	Text Document	1 KB
MyImage	03-10-2019 17:53	PNG File	95 KB
OriginalFile	16-08-2020 20:13	Text Document	1 KB
OutputFile	16-08-2020 23:11	Text Document	1 KB
OutputImage	18-08-2020 11:33	PNG File	95 KB

Figure 9-1: OutputImage.png is created inside c:\TestData

Renaming and Removing Files

You can rename and remove an ***existing*** file. The following program can show this. In the previous program, you’ve created OutputImage.png . Now you can rename it using the rename function. This function has the following format: rename(‘old_file_name’,’new_file_name’) . To delete a file, you can use the remove() function which has the following format: remove(“file_name”).

Before you see them in demonstration 5, go through the following notes:

- In the following program, I rename the file first. Then I delete the file. So, you see the use of rename before remove .
- To use the functions, rename, and remove , I needed to use the following line at the beginning:

```
from os import rename, remove
```

Demonstration 5

Here is the complete program for you.

# rename_and_remove_file.py

```
from os import rename, remove
#Renaming the image
rename("C:\\TestData\\OutputImage.png","C:\\TestData\\OutputImage1.png"
#Deleting the file
remove("C:\\TestData\\OutputImage1.png")
```

Output

After you execute this program, go to the output file location. This time you do not see any file called OutputImage.png or OutputImage1.png. It is because in the first step; I rename OutputImage.png to OutputImage1.png. Later, I delete this file. If you disable the code for the deletion operation, you can see OutputImage1.png in this location.

Using with Keyword

Closing a file is an important activity. When you close a file, you free system resources. This activity also allows other code to use this file now. If you forget to close one small file, you may not see the immediate effect. But if you have too many open files, you may see the impact of memory leaks, which may end your program abnormally. This is why I told you earlier that you should close the file properly to avoid unwanted situations. If you are unsure when to close a file, you can leave the decision to Python. The 'with' keyword in this context can serve this purpose.

Demonstration 6

First, I rewrite "Approach1" in demonstration1. It is as follows:

# using_with_keyword.py

```
# Using the ‘with’ keyword in this example.
# Python closes the file when it is no longer needed.
with open('C:\\TestData\\OriginalFile.txt', 'r') as file_object:
    print("---Approach:1---")
    print("Reading each line one by one:")
    first_line = file_object.readline()
    second_line = file_object.readline()
    third_line = file_object.readline()
    print(first_line, end='')
    print(second_line, end='')
    print(third_line, end='')
```

Output for Approach1

```
---Approach:1---
Reading each line one by one:
Dear Reader, this is my test file.
It is originally stored at C:\TestData in my system.
I hope you'll enjoy learning.
```

Now I rewrite "Approach2" in demonstration1. It is as follows:

```
# Using the ‘with’ keyword in this example.
# Python closes the file when it is no longer needed.
with open('C:\\TestData\\OriginalFile.txt', 'r') as file_object:
    print("---Approach:2---")
    print("Reading entire file using for loop:")
    for current_line in file_object:
        print(current_line, end='')
```

Output for Approach2

```
---Approach:2---
Reading entire file using for loop:
Dear Reader, this is my test file.
It is originally stored at C:\TestData in my system.
I hope you'll enjoy learning.
```

Now I rewrite “Approach3” in demonstration1. It is as follows:

```
# Using the ‘with’ keyword in this example.
# Python closes the file when it is no longer needed.
with open('C:\\TestData\\OriginalFile.txt', 'r') as file_object:
    print("---Approach:3---")
    print("Using the read() method.")
    file_content = file_object.read()
print(file_content)
```

Output for Approach3

```
---Approach:3---
Using the read() method.
Dear Reader, this is my test file.
It is originally stored at C:\TestData in my system.
I hope you'll enjoy learning.
```

I hope you have got an idea about how to use the ‘with’ keyword in your program.

Handling FileNotFoundError

In Chapter 8, you learned how to manage run-time errors or exceptions. You can use the knowledge to make a better program. When you work with files, handling missing files is a common activity. Your program may not find a file for various reasons. Here are some of them:

- The file resides in a different location,
- You misspelled the file name,
- The intended file does not exist at all, etc.

So, it’s a better practice to put the code in the try-except block to avoid unexpected outcomes.

Demonstration 7

Here I change the approach3 of the previous demonstration to give you an idea about it. In this example, I use the file name OriginalFile_1 which does not exist inside C:\TestData on my computer.

```
# Using try-except to handle FileNotFoundError
try:
    my_file = 'C:\\TestData\\OriginalFile_1.txt'
    with open(my_file, 'r') as file_object:
        print("Using the read() method.")
        file_content = file_object.read()
    print(file_content)
except FileNotFoundError as e:
    print(f"The file {my_file} is not found.")
    print(f"Error details:{e}")
```

Output

Here is the output:

```
The file C:\TestData\OriginalFile_1.txt is not found.
Error details:[Errno 2] No such file or directory:
'C:\\TestData\\OriginalFile_1.txt'
```

Analysis

If you do not handle this error using try-except , you can see the following error:

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter9/handling_file_not_fou
line 4, in <module>
   with open(my_file, 'r') as file_object:
FileNotFoundError: [Errno 2] No such file or directory:
'C:\\TestData\\OriginalFile_1.txt'
```

EXERCISE

E9.1 Write a program to print the content of a file.

E9.2 Write a program to copy an image.

E9.3 Write a program to rename a file.

E9.4 Write a program to remove a file from a specific location.

E9.5 Write a program to count an approximate number of words in a text file.

E9.6 Write a program to print first ‘n’ lines from a file. A user can supply the value of ‘n’.

Solution to Exercises

E9.1, E9.2, E9.3, E9.4
I already showed you the possible solutions to these exercises in this chapter. This time I want you to write better programs. For example, you can write a small function to make a solution for E9.1. You can use your knowledge of exception management in your implementation. Here, I show you a sample solution for E9.1

```
def print_file_content(input_file):
    """ This function can print the content of a text file."""
    try:
        with open(input_file,"r") as file_object:
            file_content = file_object.read()
    except FileNotFoundError as e:
        print(f"The file {input_file} is not found.")
        print(f"Error details:{e}")
    else:
        print(file_content)



print_file_content('C:\\TestData\\OriginalFile.txt')
```

Author’s comment:
You see an else block here. You understand that this block executes only if the code in the try block passes well.

E9.5

```
def count_words(input_file):
    """
    This function counts the approximate number
    of words in a text file.
    """
    try:
        with open(input_file,"r") as file_object:
            file_content = file_object.read()


          except FileNotFoundError as e:
        print(f"The file {input_file} is not found.")
        print(f"Error details:{e}")
    else:
        separate_words = file_content.split()
        word_count = len(separate_words)
        print("The content of the file:")
        print("-" * 20)
        print(file_content)
        print("-"*20)
        print(f" The file has {word_count} words (approx).")



count_words('C:\\TestData\\sample_text_file.txt')
```

Sample output:

```
The content of the file:
```

```
--------------------
Sky is blue.
Apple is red.
Sam is a good boy.

--------------------
The file has 11 words (approx).
```

Author’s comment:
You can enhance this solution. For example, you can ask the filename of a user. I left this exercise for you.
You can make an alternative solution too. For example, you can use a for loop instead of using the len() function. Here is a sample:

```
#word_count = len(separate_words)
# Alternative solution
word_count = 0
for word in separate_words:
    word_count += 1
#remaining code.
```

E9.6
I use the OriginalFile.txt for this example.

```
def print_lines(input_file, lines):
    """ This function prints first ‘n’ lines from a file."""
    try:
        with open(input_file, "r") as file_object:
            count = 0
            while count < lines:
                line_content = file_object.readline()
                print(line_content,end='')
                count += 1
    except FileNotFoundError as ex:
        print(f"The file {input_file} is not found.")
        print(f"Error details:{ex}")
```

```
try:
    user_input = input('Enter how many lines you want to print from the file? ')
    number_of_lines = int(user_input)
    print_lines('C:\\TestData\\OriginalFile.txt',number_of_lines)
except ValueError as e:
    print("Invalid input! Provide a correct input next time!")
    print(f"Error details:{e}")
except Exception as e:
    print("An unknown error occurred.")
    print(f"Error details:{e}")
```

Here is a sample output:

```
Enter how many lines you want to print from the file? 2
Dear Reader, this is my test file.
It is originally stored at C:\TestData in my system.
```

Here is another sample output:

```
Enter how many lines you want to print from the file? 2.3
Invalid input! Provide correct input next time!
Error details:invalid literal for int() with base 10: '2.3'
```

Author’s comment:
You can make a better solution. For example, you can ask the file name too from a user. I left this exercise for you.

CS 7.1 Implementation

In this implementation, you see the use of the makedirs() function that comes from the os module. Here is the function documentation from PyCharm IDE for your immediate reference:

```
def makedirs(name, mode=0o777, exist_ok=False):
    """makedirs(name [, mode=0o777][, exist_ok=False])

      Super-mkdir; create a leaf directory and all intermediate ones.  Works like
```

mkdir, except that any intermediate path segment (not just the rightmost)will be created if it does not exist. If the target directory already exists, raise an OSError if exist_ok is False. Otherwise, no exception is raised. This is recursive.

```
  """
```

This is why I import the os module at the beginning of this implementation. Go through the function documentation to understand the usage of different functions. Here is the complete implementation for you:

```
import os



class ScoreExceedsError(Exception):
    """ This is a custom exception."""
    pass



def calculate_grade(a1_score,
                    a2_score,
                    e_score):
    """
    This function calculates the final score.
    Here is the consideration:
    25% of the total marks in assignment and
    75% of the total marks make the grade card.
    """
    final_score = (a1_score + a2_score) * .25\
                  + e_score * .75
    return final_score


def make_grade(score):
    """ This function makes the grade."""
    grade = ""
    if score > 90:
        grade = "A+(Outstanding)"
    elif score > 80:
        grade = "A(Very Good)"
    elif score >= 70:
        grade = "B(Good)"
    else:
        grade = "F(Fail)"
    return grade


def save_scores(name,
                assign1,
                assign2,
                exam,
                score,
                grade):
    """
```

```
This function stores the result in a text file.
It picks the current working directory.Then create a
directory(if it does not exist),called GradeScores.
All records are stored as test files inside
this directory.
"""
try:
    # Retrieve the current working directory
    current_path = os.getcwd()
    print("The current working directory:")
    print(current_path)
    # Adding a relative path to the current path
    relative_path = "GradeScores"
    new_path = os.path.join(current_path,
                            relative_path)
    # Making the directory if it does not exist
    if not os.path.exists(new_path):
        os.makedirs(new_path)
    new_path = new_path + '\\' + name + '.txt'
    print("Get the report card at:")
    print(new_path)
    with open(new_path,'w') as file_object:
        file_object.write("***Report Card***")
        file_object.write("\n***Course name: Python for Beginners***\n")
        file_object.write("=" * 50)
```

```
            file_object.write(f"\nStudent Name: {name}")
            file_object.write(f"\nAssignment-1 Score:{assign1}")
            file_object.write(f"\nAssignment-2 Score:{assign2}")
            file_object.write(f"\nExam Score:{exam}")
            file_object.write(f"\nFinal score:{score}")
            file_object.write(f"\nGrade/Remark: {grade}")
    except FileNotFoundError as e:
        print(f"The file is missing.Details:{e}")
    except Exception as e:
        print(f"Error details: {e}")


def main():
    """
    It is the top level function for
    this application.
    """

           try:
        student_name = input("Enter the student name:")
        assign1 = float(input("Assignment-1 score:"))
        if assign1 > 50:
            raise ScoreExceedsError("Assignment1 score cannot be greater than
50.")
        assign2 = float(input("Assignment-2 score:"))
        if assign2 > 50:
            raise ScoreExceedsError("Assignment2 score cannot be greater than
50.")
        exam = float(input("Exam score:"))
        if exam > 100:
            raise ScoreExceedsError("Final exam score cannot be greater than
100.")
    except ValueError as e:
        print(f"Invalid input.Details:{e}")
    except ScoreExceedsError as e:
        print(f"Error: {e}")
        print("Provide the correct input next time!")
    except Exception as e:
        print(f"Error details: {e}")
    else:
        final_score = calculate_grade(assign1,assign2,exam)
        print(f"Final score:{final_score}")
        grade_score = make_grade(final_score)
        print(f"Grade: {grade_score}")
        save_scores(student_name,
                    assign1,
                    assign2,
                    exam,
                    final_score,
                    grade_score)
```

```
main()
```

Chapter 10: Object-Oriented Programming Concepts

Welcome to the basics of object-oriented programming (OOP). I remind you that Python supports both procedural programming and object-oriented programming. The OOP in Python is not exactly like other languages like Java, C#, etc. There are some differences. Those differences are not your focus in this chapter. Instead, to get the most benefit, here you'll learn the basic terms in OOP. These are independent of any programming language. In the next chapter, you'll see how some of these concepts can be implemented in Python.

Let us begin with some fundamental questions and answers. For example, why do you need this kind of programming? Or how these concepts can make your life easy? If you know the answers, the learning path will be easy, and you can relate these concepts in various ways. But before you start, there are two warning messages for you:

- Do not lose your motivation if you cannot understand everything after the first pass. Sometimes it is complex, but gradually it will be easier for you.
- Many developers criticize the concepts of OOP. But you remember that the human minds are not always comfortable with new concepts or ideas. So, before you criticize these concepts, understand and use them in various applications. Finally, make your decision-whether to appreciate or criticize.

Now let us begin the journey…

Computer programming started with binary code. We used mechanical switches to load the programs. It is easy to assume that a programmer's life was very challenging in those days. To make their lives easy, some high-level programming languages were developed. In those languages, we began to use some simple English-like instructions. Remember that a computer can understand instructions in a binary language only. So, the compiler’s job was to translate these English-like instructions into binaries. Eventually, these high-level languages gained in popularity.

Over a period, computer capacity and capabilities increased a lot. Then developers started developing complex applications. Unfortunately, none of the programming languages were mature enough to handle them effectively. Some primary concerns were as follows:

- How can I avoid duplicate efforts? Or, how can you reuse the existing code?
- How can I control the use of global variables in a shared environment?
- How can I debug the code when too much jumping between levels occurs in a program? (For example, a C programmer can use a goto statement to make an unconditional jump in his applications.)
- How can I make a new engineer’s life easier?
- How can I maintain a large codebase in a better way?

To solve these problems, expert programmers made some changes. They started breaking the large problems into smaller problems. The idea behind this philosophy was very simple. *It states that if you can solve these smaller problems, eventually you’ll solve the big problem.* So, they started portioning the big problems into small chunks. And the concept of functions (or procedure or subroutines) was developed. They dedicated each function to solve one small problem. So, managing these functions and the interactions among them became the key focus and it created the concept of structured programming. Structured programming was a big hit. It is because managing small functions is easy and you can debug them easily. At the same time, developers also started limiting the use of global variables. They started replacing them with local variables in the functions wherever possible.

Structured programming was popular for almost two decades. During this time, the capacity of hardware increased a lot. So, developers wanted to solve more complex tasks. Over the period, the limitations of structured programming became more prominent. For example, consider the following cases:

- Suppose, in a program, you have used a particular data type across many functions. Later you discover that you need to change the data type. As a result, you need to make changes in all functions in this application.

- The key component of structured programming is data and functions. It is difficult to model all real-world scenarios with them. In the real world, whenever you create a product, there are two areas you need to focus on.

    - *Purpose.* Why do you need the product?
    - *Behavior*. How can the product make your life easier?

Then the idea of objects came into existence. It will be good to know that Alan Curtis Kay is widely considered the one father of object-oriented programming. He used this naming along with some colleagues at Palo Alto Research Center (PARC). It was formerly known as Xerox PARC.

POINTS TO REMEMBER

Structured programming makes a program more efficient by enforcing a logical structure. You can do this type of programming in different programming languages. It also relates to the basis of structured programming. Here you see that functions (or procedures) play a vital role in the data. For simplicity, you can think of structured programming without object-oriented functionalities/features as procedural programming. So, the fundamental difference between procedural (or purely structured) programming and object-oriented programming can be summarized as *In object-oriented programming, instead of focusing on* the operations on data, you focus on the data itself.

In OOP, there are some important principles. I'll cover them quickly in this chapter, and you will get a brief introduction to each of them. I recommend you to read and understand these concepts clearly.

Class and Objects

These are at the core of OOP. A **class** is a blueprint or the template for its objects. **Objects** are instances of a class. Each object has its state, behavior, and identity. In simple language, in structured programming, you segregate the problem into small functions. But in OOP, you divide the problem into objects. I assume that you are familiar with data types like int, float, etc. You know that these are primitive data types. We also call them built-in data types because they are already defined in a computer language. But when you create your data type, let's say, Player, you need to create a Player class. When you need to create an integer variable, you mention the int first. In the same way, when you create a Player object (e.g., Michael), you mention your Player class first.

Let us look into this. Are you familiar with the game of football (or *soccer*, as it's known in the United States)? Like any other game, the players are selected for their skills. These players have the least level of match fitness and some important athletic capabilities. So, when I say that Ronaldo is a footballer (a.k.a. soccer player), you can predict that Ronaldo has these basic abilities and some skills specific to football. Notice that you can make these assumptions even though Ronaldo is unknown to you). This is why you can say that Ronaldo is an object of a Footballer class.

Note It may appear to you it is a chicken-or-the-egg type of dilemma. Perhaps if I say, "X is playing like Ronaldo," then, in that case, Ronaldo is acting as a class. However, in object-oriented design, you make things simple. You ask," Who comes first?" The answer to this question is the class in your application.

Now consider another footballer, Beckham . You can predict again that if Beckham is a footballer, then he must be excellent in many aspects of football. Also, he must have a minimum fitness level to take part in a match.

Now let's assume Ronaldo and Beckham both are taking part in the same match. It is easy to predict that although both are footballers, their playing style and performance differ from each other in that match. In the same way, in the OOP world, the performing objects differ from each other, even though they belong to the same class.

You can consider different domains. For example, you can say that a Dog is an object from a Mammal class, a Bus is an object from a Vehicle class, and so on.

In short, *in the real-world scenario, each of the objects must have two basic characteristics: state and behavior*. Consider the objects—Ronaldo or Beckham from the Footballer class again. You may notice that they have states like “playing state” or “non-playing state.” In the playing state, they can show different skills (or behaviors)—they can run, they can kick, they can pass the ball, and so forth.

In a non-playing state, the behavior will also change. In this state, they can take a much-needed nap, or they can eat their meals. Or they can simply relax by doing activities like reading a book, watching a movie, and so forth.

In the same way, you can also say that the televisions in your home, at any moment, can be an “on” state or an “off” state. It can display different channels if, and only if, it is in *switched on* mode. It shows nothing if it is in *switched off* mode.

So, to begin with, in object-oriented programming, you can ask the following questions:

- What are the possible states of my objects?
- What are the different functions (behaviors) that they can perform in those states?

Once you get the answers to these questions, you are ready to proceed. Software objects follow the same pattern in any object-oriented program. We store their states in fields (variables). We describe their capabilities (behaviors) through different methods (or functions).

Encapsulation

In OOP, you do not allow your data to flow freely inside the system. Instead, you wrap the data and functions into a single unit (i.e., in a class). It is important to note that you often see the term ‘**method’** instead of ‘**function’**. In the OOP context, functions are often referred to as methods. Though some

technical guys are very particular about these terms. They say when you invoke a function through an object of the class, you should mention it as a method, but not a function. Otherwise, you can mention it as a function.
The purpose of encapsulation is at least one of the following:

- Putting restrictions so that the components of an object cannot be accessed directly.
- Binding the data with methods that will act on that data (i.e., forming a capsule)

In some OOP languages, the hiding of the information is not implemented by default. So, they come up with an additional term called *information hiding*.

Later you see that data encapsulation is one of the key features in a class. If you want to promote security, your data should not be visible from the outside world. Only through the methods defined inside the class, you can access these data. Thus, you can think of these methods as the interface between the objects' data and the outside world (i.e., your program).

In short, encapsulation is a process through which you bundle your data and associated operations (i.e., functions or methods) that can work on these data. At the same time, you define the way to access these data.

Abstraction

Abstraction shows only essential features and hiding the background details of implementation. It is very much related to encapsulation, but there is a difference. Let us understand it using a simple day-to-day scenario.

When you press a button on the remote control of your TV, you do not worry about the internal circuits of the TV or how the remote control works. You know that different buttons on the remote control have different functionalities. As long as they work, you are happy. So, a user is isolated from the complex implementation details. These details are encapsulated within the remote control (and TV). At the same time, the common operations that can be performed through remote control can be thought of as an abstraction in the remote control. A manufacturer can further enhance this feature when the same remote can also perform on a different model or product. For example, you can use a DVD player remote control to control the volume of a TV.

As a developer, you need to filter out which information is essential for you. Let us assume you make an application for an organization. When you analyze the information, you may see that you can have access to the employee name, his (or her) identification number, address, contact details, age, hobbies, spouse name, dependent name, previous experience details, etc. But when you create an Employee class for your application, you may find that we do not need all these details. Instead, you can simply include the employee name, the identification number, and the contact address in an Employee class. We term the way you filter out the unwanted information and pick the essential information as an abstraction process.

Inheritance

Whenever we talk about reusability, normally we refer to inheritance. It is a process in which one object acquires the properties of another object. Consider an example. You use vehicles for transportation. Now you know that Bus is one type of Vehicle because it fulfills the basic criteria of a Vehicle . Similarly, Train is another type of Vehicle . In the same way through a ‘ Goods train ’ and a ‘ Passenger train’ are different but both of them fulfill the basic criteria of a Train , which in turn is a Vehicle . So, we say that you can support hierarchical classifications using inheritance.

Using inheritance, you create a new child class from an existing class. You refer to the existing class as a parent class. A parent class is also referred to with different names. For example, a C# programmer often calls a parent class a base class. But a Java programmer refers to it as a superclass.

In short, you place a parent class one level up than a derived class in a hierarchical chain. Then you can add or change the functionalities (methods) inside a derived class. When you do this, you say “I’m overriding the functionalities into the child class”. Keep in mind that these modifications should not change the original architecture. For example, let’s assume you have two classes called ‘ Bus’ and ‘ Vehicle’ . And you derive the Bus class from the Vehicle class. Now you add or change the functionalities in the Bus class. These modifications should not affect the original functionalities in the Vehicle class. The key advantage is that you can type less and avoid lots of duplicate codes in a program.

Polymorphism

Polymorphism is associated with *one name with many forms*. Consider the behavior of your pet dog. When it sees an unknown person, it is angry and starts barking a lot. But when it sees you, it makes different noises and behaves differently. In the coding world, you can also consider the addition method. If the operands are numbers, addition means the sum or total of the numbers. But if the operands are strings, you expect to get a concatenated string. For example, 2.5+3 = 5.5 but “Hello” +”reader?” should produce Hello, reader? Notice that in both cases, I use + operation. Polymorphism can be of two types.

- ***Compile-time polymorphism****:* The compiler can decide very early which method to invoke when you compile the program. We also refer to it as static binding or early binding.
- ***Runtime polymorphism****:* The actual method calls are resolved at runtime. At compile time, you cannot predict which method will be invoked at run-time. Let us consider an example. Suppose you generate a random number at the very first line in a program. Now you have the following logic: if the generated number is an even number, you call a method, Method1() , which prints “ Hello ”. Otherwise, you’ll call a method whose name is the same, but it prints “ Hi ”. In this case, you do not know in advance - whether you will see “Hello” or “Hi” before the program’s execution. So, we term this as dynamic binding or late binding.

Now I add a Q&A session for you. This can help you to review your understanding.

Q&A Session

10.1 What are the key features of object-oriented programming?

Class and objects are the backbones of OOP. Apart from them, the following are the key features of object-oriented programming:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

10.2 How is an object different from a class?

A class is a template or a blueprint. Objects are made from a class. An object is an instance of a class. An object is a physical entity. You need to allocate memory for it. But class is a logical entity, and it does not allocate memory in the system.

10.3 How does an abstraction differ from an encapsulation?

Abstraction focuses on the noticeable behavior of an object. But encapsulation focuses on the implementation part of that behavior. Encapsulation helps you to bundle your data. It also hides some information that you do not want to disclose to the outside world. In this context, you can refer to the example that I discuss in this chapter.

10.4 What is the key advantage associated with the inheritance mechanism?

By reusing the existing code, you can save time and effort. At the same time, this mechanism helps you to avoid duplicate codes in your application.

10.5 What are the characteristics of object-oriented programming?

Here are some important characteristics:

- Your focus is on data, not on functions. So, you divide your program into objects, not into functions.
- You do not allow data to flow freely. You use methods to access them.
- Your objects communicate through methods.

- The outside world cannot access the data directly.
- Your application can adapt to new changes easily. It promotes easy maintenance.

Case Study VIII

CS 8.1 Problem Statement

Let us assume you need to make an application that can design a computer science course in various institutions. Here are the assumptions:

- The course includes three subjects. Two subjects are common across all institutions. Let us assume these two subjects are mathematics and soft-skills. But the third one is a specialized subject that can vary across the institutions.
- A user can supply the institution name and the third subject name when he initializes an object. But these are optional for him.
- By default, the institution's name is St. Stephen College , and the specialized subject is Compiler design .
- To reduce the code size, you can assume there is no invalid user input to consider.

I give you two sample outputs.

Sample-1: A User supplies the institution name as Presidency and the compulsory third subject as Python Programming.

```
Institution name: Presidency college.
Computer science course includes:
1:Mathematics.
2:Soft-skills.
3:Python Programming
----------
```

Sample-2: A User does not supply the institution name and the compulsory third subject.

```
Institution name: St. Stephen college.
Computer science course includes:
1:Mathematics.
2:Soft-skills.
3:Compiler design.
----------
```

Author’s Comment:
Here I use the concepts of class, object, and inheritance in OOP. You will:

- See the use of different constructors.
- Know about how to use a superclass constructor effectively.
- See the use of default values in your code.

I follow a simple template design pattern in this project. I want you to know this name because it is a famous Gang of Four design pattern.

Chapter 11: Class, Objects, and Inheritance

Let us do some object-oriented programming exercise in Python. Here I make things very simple. So, I'll ignore some typical corner cases to make your journey easy.

Constructing the Building Blocks for OOP

Let us recollect what you have learned in the previous chapter. Class and objects are the building blocks in OOP. To represent a real-world entity, we create a class, and then we create objects from it. So, you understand that to create objects from a class; you need to have the class first. Since we make objects from a particular class, we define the common behavior of these objects inside the class.

You may note that objects and instances are often used interchangeably. It is because we refer to the process of creating an object from a class as instantiation.

POINTS TO REMEMBER

- You start with a class, which is the architectural blueprint for you. A class defines the structure and behavior of the objects. From a single blueprint, you can create multiple buildings. Similarly, from a single class, you can construct multiple objects (or instances). (As said before, I ignore some typical corner cases when I make this statement. For example, a true singleton class cannot have multiple instances).
- Developers often refer to the terms function and method interchangeably. In Python, a method

is usually associated with objects/classes. In other words, you use a method for a particular object. A method can access the data that is contained in a class.

- Classes help you create a new data type. Objects hold the data (fields) and methods. When you create an object from a class, you supply the name of the object. We expose object behaviors through these methods.
- In OOP, objects are self-contained. The encapsulated code can include methods, variables, etc. When different methods are called, actually the function defined in the class is invoked. So, it is important to know which object calls the function.
- This type of construct allows you to write modular programming. It is useful for easy maintenance.

Let us model a simple class, called Student . This class tells that it is a " Student class ", and it has one behavior which can say " Hello " to a student. Python helps you create the class using the following code:

```
class Student:
   """ This is a simple Student class."""
   name = "Student class"

     def say_hi(self,student_name):
       """ A simple method to say hello to a student."""
       print(f"Hello {student_name}!")
       print(f"You are using {self.name} now.")
```

There are some new things to learn, but don't worry! You'll be familiar with the structure shortly. I suggest that you continue reading the following information carefully.

Notice that I use the class keyword to create a class. The class has a name, and the line ends with a colon (**:**).

In the next line, you see a docstring which tells about the class. In real-world programming, docstrings are used to specify what a class can do in your code.

POINTS TO REMEMBER

I want you to note the following points carefully:

- The coding convention for a class name is: the first letter of each word is capitalized and you should not separate words using underscores. For example, MyClass, Student, ColorContainer, etc. are the standard class names. Instance names are lowercase case letters, and you can have underscores between the words.
- It’s a better practice to maintain a meaningful docstring in your class, and function(s). I recommend you to follow this practice. To type less, in the remaining chapter, I use some code fragments without docstrings. There I focus on some other aspects of a class.

The indented code shows the class body. You can see this class has a variable, called name and a function, called say_hi(). Collectively, you can call them class members.

Till now, you have the class definition only; it does not allocate any computer memory.

You can see the ‘ self ’ parameter in the function definition. We use it to refer to the actual object. This parameter must appear before any other parameter(s). It is included in the method definition for a special purpose.

When you call this method using an object of this class, this method automatically passes the self argument. You can use it to access the variable name that is defined inside the class.

POINT TO REMEMBER

The self parameter in a method definition comes before any other parameter(s) in the method definition. It has a special purpose. ***A Python method requires the object to be passed as the first argument to a class to distinguish the object from other objects of the class***. It means that when you invoke a method using a class instance, the self argument is automatically passed. And it helps you to access individual attributes or methods of the class. (For a static method, ‘ self ’ is NOT required. You can skip that part for now).

If you are familiar with Java or C#, you’ll find that **self** in Python is similar to **this** in Java or C#.

Unlike C structures or Java classes, you can skip declaring the name variable in advance. Instead, you can create it on the fly in Python. I’ll discuss this shortly.

Since I have created a class called Student, I can now create an instance (or, object) from it. Let us name the instance as object1 . I use the following code:

```
#Creating an object from Student class
object1=Student()
```

The Student class has defined the method say_hi() which takes two arguments. I said before that self argument is passed automatically. So, when I invoke the method using this object, I pass only a student name as follows: #invoking the method
object1.say_hi("John")

If you execute this code fragment, you see the following output:

```
Hello  John!
```

```
You are using Student class now.
```

Let us create another object, called object2 . Then invoke the say_hi() method with a different name (Kate) as follows:

```
#Creating another object from Student class
object2=Student()
#invoking the method
object2.say_hi("Kate")
```

This time you get the following output:

```
Hello Kate!
You are using Student class now.
```

Hope you have got the idea! Now you are ready to execute your first object-oriented program. Let us go through the complete code in demonstration1.

Demonstration 1

Create a new python file. Let us call it **class_object_demo_1.py** and type the following lines into it.

```
# The class definition


class Student:
    """ This is a simple Student class."""
    name = "Student class"

    def say_hi(self,student_name):
        """ A simple method to say hello to a student."""
        print(f"Hello {student_name}!")
        print(f"You are using the {self.name} now.")


#Creating an object from Student class
object1 = Student()
#Invoking the method
object1.say_hi("John")
```

```
#Creating another object from Student class
object2 = Student()
#Invoking the method
object2.say_hi("Kate")
```

Output

When you run this program, you will get the following output.

```
Hello  John!
You are using the Student class now.
Hello  Kate!
You are using the Student class now.
```

Analysis

If you have a Java or C# background, you may find some similarities. For example, I define an attribute(name) in the class before I use it in a print statement. But Python gives you more flexibility. You can define the attribute on the fly. You’ll see the example in demonstration3.

POINTS TO REMEMBER

A class is a logical entity. Once you instantiate a class, you create objects. These objects occupy memories in your system. So, the objects are physical entities.

Constructor

You use constructors to run initialization codes and make the objects. Constructors can be both parameterized and non-parameterized. When you use a parameterized constructor, you can pass a different number of arguments to them.

In Python, there is a special method, called __init__(). It runs automatically whenever you create an instance from the class. Following the convention, this method has two leading and two trailing underscores.

Here is an example of a non-parameterized constructor in a class. Ideally, ‘non-parameterized’ means that there is no parameter. But in Python, __init__(self) takes the argument self automatically, and you do not need to pass anything inside it. So, we often refer to this constructor as a non-parameterized constructor.

```
def __init__(self):
    print("You do not need to supply any parameter.")
```

Consider the following code now. Here you see a class with a non-parameterized constructor as follows:

```
class Student:
    def __init__(self):
        print("You do not need to supply a parameter.")

# Creating an object from Student class.
# Using the non-parameterized constructor.
object1 = Student()
```

If you execute this code fragment, you can see the following output:

```
You do not need to supply a parameter.
```

Now, look into a constructor that accepts one argument (name) from you.

```
def __init__(self, name):
    print("You need to supply a name.")
```

Likewise, here is another constructor that needs two arguments (name and roll_no) from you.

```
def __init__(self,name,roll_no):
    print("Supply a name and a roll number.")
```

and so on.

Let us consider a class with a constructor. In the following case, the Student class has a constructor that has one parameter. I define it as follows:

```
class Student:
    def __init__(self, name):
        print(f"You've supplied the name:{name}")
```

Whenever you create an object from this class, you need to supply a value for the name parameter. The following code with supportive comments shows you such a usage:

```
# Creating an object from Student class.
# Using the constructor that accepts one parameter.
student_john = Student("John")
```

If you execute this code, you get the following output:

```
You've supplied the name:John
```

Hope you understand the concept. Let us examine a complete program in demonstration2, and follow the discussion.

POINTS TO REMEMBER

Constructors are used for initialization. For now, to make these simple, I use some simple print statements in them. In the upcoming examples, you’ll see the true initialization statements inside these constructors.

Demonstration 2

Create a new python file. Let us call it **class_object_demo2.py** and type the following lines into it.

```
class Student:
    """ This is a Student class with a constructor."""

    def __init__(self, name, roll_no):
        """
        The name and roll_no is used to
        identify a student properly.
        """
        print(f"The student name is: {name}")
        print(f"The student roll number is: {roll_no}")


# Creating an object from Student class.
# Using the constructor that accepts two parameters.
student_robin = Student("Robin", 2)
```

Output

When you run this program, you see the following output.

```
The student name is: Robin
The student roll number is: 2
```

Analysis

In demonstration 1, I created objects from the Student class, but you do did not see any constructor in that example. It was possible because if you do not supply any constructor for your class, Python calls the superclass constructor for you. To understand ‘Superclass’, you need to learn inheritance, which I discuss later in this chapter. In Python 3.x, the object class is the root of all classes. Once you learn inheritance, you’ll understand that class Student and class Student(object) are essentially the same. So, once you learn inheritance, you can read this paragraph again.

In this demonstration, when __init__(…) was called, the arguments- Robin and 2 are assigned to the name and roll_no attribute of the Student class. You may ask me: what is the advantage? The answer is: you can make different objects where you set different values for these attributes. For example, the following line of code:

```
student_kate = Student("Kate", 4)
```

can create another instance of Student class. Using the constructor, you set name and roll_no attributes with the values- Kate and 4 . This time your instance name is student_kate . In other words, the variable student_kate holds the instance which has Kate as a student name and 4 as her roll number.

Looking into Instance Creation Process

In OOP, understanding the object creation process is important. I want you to understand it clearly. In demonstration 1, there is a name attribute but there is no constructor. In demonstration2, you see no attribute, but you notice a constructor. Now we’ll look into these in detail.

Notice that in demonstration 2, when I create the instance, student_robin using the following line:

```
student_robin = Student("Robin",2)
```

Python calls the constructor _ _init__(self, name, roll_no) to create the object. You can see that I pass two arguments Robin and 2   during the object creation. I use the following code to print those values:

```
print(f"The student name is:{name}")
print(f"The student roll number is:{roll_no}")
```

Now I make some minor changes in demonstration2 and rename it demonstration 3. It is almost a replica of demonstration2, but this time I use the following statements:

```
print(f"The student name is:{self.name}")
print(f"The student roll number is:{self.roll_no}")
```

Have you noticed that this time you see the use of ‘ self ’? Fine, first go through the following demonstration. Then follow the discussion to understand it.

Demonstration 3

Here is the complete demonstration. Create a new python file. Let us save it with a new name, say **class_object_demo3.py,** and type the following lines into it.

```
class Student:
    """ This is a Student class with a constructor."""
    def __init__(self, name, roll_no):
        """
        The name and roll_no is used to
        identify a student properly.
        """
        print("This constructor has two parameters.")
        #initialize name
        self.name = name
        #initialize roll_no
        self.roll_no = roll_no
        print(f"The student name is: {self.name}")
        print(f"The student roll number is: {self.roll_no}")

# Creating an object from Student class.
# Using the constructor that accepts two parameters.
student_jack = Student("Jack", 3)
```

Output

When you run this program, you can expect to see the following output which is similar to demonstration2.

```
This constructor has two parameters.
The student name is: Jack
The student roll number is:3
```

Analysis

Now analyze the changes. Notice that the dot(.) notation helps you to access the attribute of a particular instance. This is why I use the following lines:

```
print(f"The student name is:{self.name}")
print(f"The student roll number is:{self.roll_no}")
```

to access the name and roll_no attributes of an instance. This demonstration shows you another important characteristic. I talked about it earlier. Have you noticed that I didn’t declare name and roll_no in advance in the class? In Python programming, you can create data fields on the fly. This is why, when I use the following lines inside the constructor:

```
   self.name=name
   self.roll_no=roll_no
```

there was no problem though I haven’t defined name and roll_no variables earlier.

Changing an Attribute Value

You can update the attribute value inside an object (or instance). For example, in demonstration 3, you create an instance, called student_jack as follows:

```
student_jack = Student("Jack",3)
```

Notice that inside student_jack , the attribute name has the value Jack and the attribute roll_no has the value 3 . Let us say, you want to assign different values in this instance. The easiest way is to directly access the attribute and change the value as follows:

```
# Changing the attribute values of student_jack
student_jack .name = "Bob"
student_jack .roll_no = 5
```

One interesting point: student_jack is a better name than object1 or object2 , but it is not good when you replace the name Jack with Bob . My aim here is to show you how to change these values. I always suggest picking a meaningful name for your object.

Alternatively, you can define a function inside your class to update an instance value. So, once you initialize the object, you can use the instance method to update the attribute values. In the upcoming demonstration, you'll see both usages.

Demonstration 4

In this demonstration, I update a student's name and roll_no multiple times. Initially, I create a Student class object, called student_1 which has the name, Jack. I assign his roll_no to 3 . It is exactly similar to demonstration3.

Next, I change the attribute values directly to Bob and 4 using the following lines:

```
student_1.name="Bob"
student_1.roll_no=5
```

Again, I update the values to Harry and 6 , but this time I use a method. I define it as follows:

```
def update_student_details(self,new_name,new_roll_no):
        self.name=new_name
        self.roll_no=new_roll_no
```

I invoke this method using the dot operator as follows:

```
student_1.update_student_details("Harry",6)
```

The remaining code is easy to understand. To test this, create a new python file and save it with a new name, say **class_object_demo4.py.** Then type the following lines into it. For your easy understanding, you can follow the associated comments.

```
class Student:
    """ This is a Student class with a constructor."""

    def __init__(self, name, roll_no):
        """
        The name and roll_no is used to
        identify a student properly.
        """
        #initialize name
        self.name = name
        #initialize roll_no
        self.roll_no = roll_no

    def get_details(self):
        """
        This method prints the detail of a student.
        """
        print("The current student's detail is:")
        print(f"Name: {self.name}")
        print(f"Roll number: {self.roll_no}")

    def update_student_details(self, new_name, new_roll_no):
        """
        This method is used to update a student's detail.
        :param new_name: Updated student name.
        :param new_roll_no: Updated student roll_no
        :return: None
        """
        self.name = new_name
        self.roll_no = new_roll_no

# Creating an object from Student class.
student_1 = Student("Jack", 3)
#printing the details
student_1.get_details()
# Changing the attribute values of student_1
student_1.name = "Bob"
student_1.roll_no = 5
print("---After the first update.---")
student_1.get_details()

#Second update using the method 'update_student_details'
student_1.update_student_details("Harry", 6)
print("---After the second update.---")
student_1.get_details()
```

Output

You can get the following output when you execute this program.

```
The current student's detail is:
Name: Jack
Roll number: 3
---After the first update.---
The current student's detail is:
Name: Bob
Roll number: 5
---After the second update.---
```

```
The current student's detail is:
Name: Harry
Roll number: 6
```

Working with Default Attributes

Till now you have seen the demonstrations where I pass the values for the attributes of an instance. But it is not mandatory. Consider our Student class again. So far, I was passing the name and roll_no of a student. But consider a case, where all students belong to the same college or university. Here, if you want your Student class object to having the college name, you do not need to pass the repeated information.

Demonstration 5

The following example shows such a case. Here a student object automatically belongs to a college, called St.Stephen . This example is like the previous demonstration, but I have excluded the update method. Here my focus is on the default attribute value. I have accomplished it using the following line inside the constructor:

```
self.college="St.Stephen"
```

To include the college information, I have changed the get_details() method. Let us create a new python file and save it with a new name, say **class_object_demo5.py.** Then include the following lines to verify the result:

```
class Student:
    def __init__(self,name,roll_no):
        #initialize name
        self.name = name
        #initialize roll_no
        self.roll_no = roll_no
        self.college = "St.Stephen"

    def get_details(self):
        print("The current student's detail is:")
        # Formatting the output using f-string
        # This syntax is available from Python 3.6 onwards)
        print(f"Name: {self.name}")
        print(f"Roll number: {self.roll_no}")
        print(f"College: {self.college}")

# Creating an object from Student class.
student_1 = Student("Jack", 3)
#printing the details
student_1.get_details()

# Creating another object from Student class.
student_1 = Student("Kate", 4)
#printing the details
student_1.get_details()
```

Output

Here is the output of the program.

```
The current student's detail is:
Name: Jack
Roll number: 3
College: St.Stephen
The current student's detail is:
Name: Kate
Roll number: 4
College: St.Stephen
```

Analysis

Notice that you do not pass the college name in the constructor. Still, you have an attribute to hold the default college name. Each student object can contain this information inside it.

Importing Classes

In a complex real-world application, the code size is gigantic. If you write the entire code into a single file, you create a huge file. This kind of practice is commonly discouraged. Instead, Python allows you to store segregate your code in modules. In Chapter 7, I told you that a module can contain many things such as variables, functions, and classes. In this chapter, our focus is on classes. So, here I show you how to import classes from a module.

To show you a simple example, I introduce a class called Employee. It is like the Student class that we discussed so far. The only differences are: instead of a roll number and a college name, now you see an employee id and a company name. I choose the default company name as Abc Ltd. Here is the class that I store in a separate file called employee_module.py. I keep lots of comments for your easy understanding.

I’ve organized all the codes chapter wise. For example, I store all programs of Chapter 11 inside a directory named chapter11 . The parent directory of the **chapter11** is **PythonCrashCourse** . If I store the **employee_module.py** inside the **chapter11** directory, every time I refer to this module, I need to mention it as **chapter11.employee_module** . So, to type less, I store the file **employee_module.py** in **PythonCrashCourse** . Now I can directly refer to the module without mentioning the chapter11.

```
# This is an Employee class where you can
# find the name, employee id, and company details.
# It is used in chapter11.

"""

Class
---------
Employee
    This is a simple class. In the constructor,
    you can store employee name, employee id and
    company information. It has a method, called
    get_details() to print these information.
```

```
"""

class Employee:
    """
    It is a simple class to store
    basic employee information.
    """
    def __init__(self, name, emp_id):
        # Initialize name
        self.name =Iname
        # Initialize emp_id
        self.emp_id = emp_id
        self.company = "Abc Ltd."

     def get_details(self):
        print("\nThe current employee details are:")
        # Formatting the output using f-string
        print(f"Name: {self.name}")
        print(f"Id: {self.emp_id}")
        print(f"Company: {self.company}")
```

Demonstration 6

Create a new python file. Save it with a new name, say **importing_single_class.py,** and type the following lines into it.

```
from employee_module import Employee

# Creating an object from the Employee class.
emp_1 = Employee("Jack", 3)
# Printing the details
emp_1.get_details()

# Creating another object from the Employee class.
emp_2 = Employee("Kate", 4)
# Printing the details
```

```
emp_2.get_details()
```

Output

Once you run the program, you can see the following output.

```
The current employee details are:
Name: Jack
Id: 3
Company: Abc Ltd.

The current employee details are:
Name: Kate
Id: 4
Company: Abc Ltd.
```

Analysis

You can see that **importing_single_class.py** contains only a few lines of code. I can create the Employee class instances in the current file because I imported this class from employee_module .

Demonstration 7

You have just seen a demonstration where you imported a class from a module. You can import multiple classes from a module, or you can import the entire module. To show you an example, I add the following class inside the file employee_module.py that I used earlier:

```
class PersonalDetails:
    """
    This class is used to show the
    background details of an employee.
    """
    def __init__(self, name, current_company,
                 work_experience, age, address):
        # Initialize department
        self.name = name
        self.current_company = current_company
        self.work_experience = work_experience
        # Initialize age
        self.age = age
        self.address = address

    def background_details(self):
        print("-" * 10)
        print("Here are the background details:")
        #Formating the output using f-string
        print(f"Name: {self.name}")
        print(f"Current company: {self.current_company}")
        print(f"Age: {self.age},Address: {self.address}")
        print(f"Experience: {self.work_experience} yrs.")
        print("-" * 10)
```

Create a new python file. Save it with a new name, say **importing_multiple_classes.py.** Then type the following lines into it.

```
from employee_module import Employee, PersonalDetails

# Creating an object from Employee class.
emp_1 = Employee("Jack", 3)
# Setting background details for emp_1
emp_1_details = PersonalDetails(emp_1.name,
                                emp_1.company,
                                10.2,
                                39,
                                "21,Abc Road,USA")
# Printing the details
emp_1.get_details()
emp_1_details.background_details()
```

Output

Run the program. You can see the following output.

```
The current employee details are:
Name: Jack
Id: 3
Company: Abc Ltd.
----------
Here are the background details:
Name: Jack
Current company: Abc Ltd.
Age: 39,Address: 21,Abc Road,USA
Experience: 10.2 yrs.
----------
```

It is a recommended practice to make a module with related classes. In demonstration7, you have seen that **employee_module** contains **Employee** class and **PersonalDetails** class. These two are related classes.

Demonstration 8

When you import an entire module, you can access its classes using the dot notation. To illustrate, consider the following example. It is an alternative version of the previous demonstration.

```
import employee_module

# Creating an object from the Employee class.
emp_1 = employee_module.Employee("Jack", 3)
# Setting background details for emp_1
emp_1_details= employee_module.PersonalDetails(
    emp_1.name,
    emp_1.company,
    10.2,
    39,
    "21,Abc Road,USA")
# Printing the details
```

```
emp_1.get_details()
emp_1_details.background_details()
```

When you run the program, you’ll see the same output which you saw in demonstration7.

Demonstration 9

Now I show another way of importing classes from a module. It is not a recommended practice, but I present this to you for the sake of completeness purpose.

You get the same output as in demonstration8(or demonstration7) if you change the first line in the previous demonstration as follows:

```
from employee_module import *  # NOT a recommended practice
#remaining code as it is
```

It is not a recommended practice due to the following reasons:

- You cannot directly see which classes are imported in the current file.
- We do not need all the imported classes in the current program file.
- If the program file has a same-named class, there will be a name collision.

To illustrate the third bullet point, let us add another Employee class in the program file.

```
from employee_module import *  # NOT a recommended practice


class Employee:
    def __init__(self):
        print("Inside  the constructor of Employee class")


# Creating an object from the Employee class.
emp_1 = Employee("Jack", 3) ##Name collision
```

```
# Setting background details for emp_1
emp_1_details = PersonalDetails(
    emp_1.name,
    emp_1.company,
    10.2,
    39,
    "21,Abc Road,USA")
# Printing the details
emp_1.get_details()
emp_1_details.background_details()
```

If you run this program, you can get an error like the following:

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter11/importing_all_classes
line 10, in <module>
    emp_1 = Employee("Jack",3)
TypeError: __init__() takes 1 positional argument but 3 were given
```

So, what is the remedy? In this case, you can import the entire module and use the dot notation to point to the correct class. Here is a sample implementation.

```
import employee_module


#from employee_module import *  # NOT a recommended practice


class Employee:
    """ This employee class stays in the current file."""
    def __init__(self):
        print("Inside  the constructor of Employee class")


# Creating an object from the Employee class.
#emp_1 = Employee("Jack", 3) #Name collision
emp_1 = employee_module.Employee("Jack", 3)
```

```
# Setting background details for emp_1
#emp_1_details = PersonalDetails(
emp_1_details = employee_module.PersonalDetails(
    emp_1.name,
    emp_1.company,
    10.2,
    39,
    "21,Abc Road,USA")
# Printing the details
emp_1.get_details()
emp_1_details.background_details()

emp_2 = Employee()  # Uses current file’s Employee class
```

Output

Run the program. You can see the following output.

```
The current employee details are:
Name: Jack
Id: 3
Company: Abc Ltd.
----------
Here are the background details:
Name: Jack
Current company: Abc Ltd.
Age:39,Address:21,Abc Road,USA
Experience:10.2 yrs.
----------
Inside  the constructor of the Employee class
```

The last line shows that you can use the Employee class that stays in the current file.

Use of Identity Operator

In Chapter 2, I told you I discuss the identity operators in this chapter. Now you know about objects. So, you are ready for it.

The identity operator can tell whether two comparing objects are the same object with the same memory location. In the following demonstration, you see three objects. I call them jack, jack_2, and jack_3 respectively. All these objects have the same content. But you'll see that I create jack_2 separately, but I create jack_3 from an existing object, jack. So, jack_3 and jack are the same, but jack and jack_2 are not the same objects. You can use the is and is not operators to check this exactness. The ' is' operator returns True if both variables point to the same object. The ' is not' can check the reverse.

Demonstration 10

Here is the complete demonstration.

```
class Student:
    """ This is a Student class with a constructor."""
    def __init__(self, name, roll_no):
        """
        The name and roll_no is used to
        identify a student properly.
        """
        self.name = name
        #initialize roll_no
        self.roll_no = roll_no
        print(f"Name: {self.name}")
        print(f"Roll number is: {self.roll_no}")


# Creating an object from Student class.
# Using the constructor that accepts two parameters.
print("Creating jack object:")
jack = Student("Jack", 3)
print("\nCreating jack_2 object:")
jack_2 = Student("Jack", 3)

print("\nUse of identity operator:")
print("jack is jack_2?", jack is jack_2) # False
print("Introducing jack_3 object")
jack_3 = jack
print("jack_3 is jack?", jack_3 is jack) # True
print("jack_3 is jack_2?", jack_3 is jack_2) # False
```

Output

Here is the output.

```
Creating jack object:
Name: Jack
Roll number is: 3

Creating jack_2 object:
Name: Jack
Roll number is: 3

Use of identity operator:
jack is jack_2? False
Introducing jack_3 object
jack_3 is jack? True
jack_3 is jack_2? False
```

You can use these operators to check if the variable is of a certain type. Here is a short code segment for you:

```
>>> x=10
>>> type(x) is int
True
>>> type(x) is str
False
>>> type(x) is float
False
>>> type(x) is not float
True
```

So, in demonstration10, if you append the following lines:

```
print("The type(jack) is Student?")
print(type(jack) is Student) # True
print("The type(jack) is int?")
print(type(jack) is int) #False

print("The type(jack) is NOT int?")
print(type(jack) is not int) # True
```

You see some more output for these new lines of code. These are as follows:

```
The type(jack) is Student?
True
The type(jack) is int?
False
The type(jack) is NOT int?
True
```

Inheritance

The primary aim of inheritance is to promote reusability and eliminate redundancy in code. It also shows how a child class can get the features (or characteristics) of its parent class. A parent class is placed at a higher level in the class hierarchy, and a child class can derive from it. So, we also refer to a *child class is as a **derived class** or **subclass**. We often refer to a parent class as a **superclass**.*

In short, when you make a specialized version of a parent class, you can avoid writing the code from scratch. Instead, you use inheritance to get the attributes and methods of the parent class. A child class can use any number of attributes or methods of the parent class. It can also define new attributes and methods of its own.

Let us start with a simple example. In the following example, Shape is the parent class. You can make any class a parent class. So, the syntax of creating a parent class is the same as creating any other class. Here is the Shape class for you. You can see that it has a constructor, and a method called about_me() .

```
class Shape:
    """ This is the parent class """
    def __init__(self):
        self.type = "Shape"

    def about_me(self):
        print(f"I am a {self.type}.")
```

To create a child class, you use the parent class as a parameter. In our upcoming example, the Rectangle class inherits from the Shape class. So, you notice the following line:

```
class Rectangle(Shape):
    #Remaining code
```

In the following example, I use one Shape object and one Rectangle object. The Shape object(shape_ob) can access the about_me() method. It is obvious. But notice that the Rectangle object(rectangle_ob) can also access the method. It is possible because the Rectangle class inherits from the Shape class.

Demonstration 11

Here is the complete demonstration. I have created a new Python file, named inheritance_ex1.py, and place the following code inside it.

```
class Shape:
    """
    It is the Shape class.
    It is the parent class of Rectangle
    in this example.
    """

    def __init__(self):
        self.type = "Shape"

    def about_me(self):
        print(f"I am a {self.type}.")


class Rectangle(Shape):
    """
    This is the Rectangle class which
    inherits from  the Shape class
    """


print("***Simple inheritance example.***")
shape_ob = Shape()
shape_ob.about_me()

rectangle_ob = Rectangle()
rectangle_ob.about_me()
```

Output

Here is the output.

```
***Simple inheritance example.***
I am a Shape.
I am a Shape.
```

Analysis

You can see that the Rectangle class object can access the parent class method. I show this for a demonstration purpose. We know that you can have a different type of shape, such as a rectangle, a circle, or a square. So, we can override the parent class method inside the Rectangle class. It is better to use the child class constructor to tell that it is a Rectangle type. Let us update demonstration10 now.

Updated Implementation

Here is an updated implementation of Demonstration10.

```
class Shape:
    """ This is the parent class """

    def __init__(self):
        self.type = "Shape"

    def about_me(self):
        print(f"I am a {self.type}.")

class Rectangle(Shape):
    """
    This is the Rectangle class which
    inherits from  the Shape class
    """

    def __init__(self):
        self.type = "Rectangle"

    def about_me(self):
        print(f"I am a {self.type}.")

print("***Simple inheritance example.***")
shape_ob = Shape()
print(f"The shape_ob.type={ shape_ob.type}")
shape_ob.about_me()

rectangle_ob = Rectangle()
print(f"The rectangle_ob.type={ rectangle_ob.type}")
rectangle_ob.about_me()
```

Output

Here is the output of this implementation.

```
***Simple inheritance example.***
The shape_ob.type=Shape
I am a Shape.
The rectangle_ob.type=Rectangle
I am a Rectangle.
```

This updated implementation shows you the following characteristics:

- You can modify the attribute of the parent class inside the child class.

- If needed, you can override a parent class method. When I say “Overriding” a parent class method, notice that the method names in the parent class and child class are the same.

You may often see a class with many attributes and methods. In a specialized version of this class, you do not change all of them. Instead, you modify a few of them. It is also possible that you add a few attributes or methods in the specialized (or child) class. The upcoming demonstration shows you a similar usage.

In demonstration6, you saw an Employee class as follows:

```
class Employee:
    """Employee class is the parent class."""

    def __init__(self, name, emp_id):
        # Initialize name
        self.name = name
        # Initialize emp_id
        self.emp_id = emp_id
        self.company = "Abc Ltd."
```

Make an inheritance hierarchy by adding a Developer class now. A Developer is a special type of Employee . So, we do not need to change the basic employee information, such as employee name, company name, employee id. You can inherit them from the Employee class. Later you can add a designation attribute in the Developer class. You can achieve this when you call the __init()__ method of the parent class using the super() function. The super() function is a special function that allows you to call a parent class method. The name comes from a convention of calling a parent class a superclass and a child class a subclass. This is why you’ll see the following code in the upcoming demonstration:

```
class Developer(Employee):
    """Developer class inherits from Employee"""

    def __init__(self, name, emp_id):
        """ Initialize starts from parent class."""
        super().__init__(name, emp_id)
        # This is a class-specific attribute
        self.designation = "Developer"
```

Now you can add a method to describe a developer detail. I present the following demonstration to show you the usage.

Demonstration 12

Here is the complete demonstration. I have created a new Python file, named inheritance_ex2.py, and place the following code inside it. Here is the complete demonstration.

```
class Employee:
    """Employee class is the parent class."""

    def __init__(self, name, emp_id):
        # Initialize name
        self.name = name
        # Initialize emp_id
        self.emp_id = emp_id
        self.company = "Abc Ltd."


class Developer(Employee):
    """Developer class inherits from Employee"""

    def __init__(self, name, emp_id):
        """ Initialize starts from parent class."""
        super().__init__(name, emp_id)
        # This is a class specific attribute
        self.designation = "Developer"

    def developer_details(self):
        """Prints the developer details."""
        print(f"Name: {self.name}")
        print(f"Id= {self.emp_id}")
        print(f"Designation= {self.designation}")
        print("-" * 10)
```

```
# Creating an instance of the Developer class
john = Developer("John S", "E001")
john.developer_details()
kate = Developer("Kate W", "E002")
kate.developer_details()
```

Output

Here is the output.

```
Name: John S
Id= E001
Designation= Developer
----------
Name: Kate W
Id= E002
Designation= Developer
----------
```

POINT TO REMEMBER

In demonstration11, if you replace the following line:

```
super().__init__(name, emp_id)
```

with the following line:

```
Employee.__init__(self, name, emp_id) #Also works
```

You get the same output. The difference is that when you use super() , you do not pass the parent class name and the self parameter. But this technique is not suitable in the long run. For example, consider a case, when you need to change the design or inheritance hierarchy. Since you hard-code the inherited class name, you need to refactor your code. This is the reason an expert programmer normally prefers to use super() in his code. This approach promotes indirection because we do not need to mention the delegate class by name.

Private Variables and Methods

If you are familiar with object-orient programming languages like Java, C#, C++, etc., you notice the use of public, private, and protected variables (and methods). If not, let me give you the idea with simple examples.

Consider your ATM card or Credit card credentials. You do not share this information with anybody else. These are your private data. In my case, my kids are among the trusted ones who know where I keep my ATM cards at my home. You can call it the protected data. In programming languages like Java, apart from the containing class, the child classes can access the protected variables (and methods). In simple words, you extend the visibility to some extent, but not for everyone. Finally, anyone can access public information. So, if I place a noticeboard outside my house to say when I like to meet my visitors - this is a piece of public information. Similarly, the advertisement for recruitment or visiting hours of a hospital represents the public data. In programming, public data has maximum visibility.

In Java, C#, or C++, declaring a variable public, private, or protected is straightforward. For example, you use the keyword “private” to declare a private variable. Now you may want to know whether Python supports a similar concept. The simple answer is: Python does not have real private variables. In Python programming, you can use the underscore(s) in a similar context.

You can use two underscores at the beginning of a variable (or a method) to mark it as a private variable (or a private method). It tells that you cannot access the variable (or the method) in a usual way. Let us test this with an example.

Demonstration 13

Consider the following class.

```
class Parent:
    """ This is the parent class"""
    def __init__(self):
      self.i = 1  #public variable
      self.__j = 10  #private variable

    def __private_show_me(self):
      print(f"i={self.i}")
      print(f"__j={self.j}")

    def about_me(self):
      print(f"i={self.i}")
      print(f"__j={self.__j}")
```

Before you proceed further, make an object of Parent class as follows:

```
parent_ob = Parent()
```

In the Parent class, i is not a private variable. You can use it directly as follows:

```
print(f"parent_ob.i={parent_ob.i}")
```

Which can output the following:

```
parent_ob.i=1
```

But __j is the private variable. So, if you try to access it directly as follows:

```
print(parent_ob.__j)#will cause error
```

You get the error like the following:

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter11/private_variable_and
line 19, in <module>
    print(parent_ob.__j)#will cause error
AttributeError: 'Parent' object has no attribute '__j'
```

Now see the methods. The about_me() method is not private. So, you can access this method directly. Since this method stays in the Parent class, it can access the private variable too. So, the following code can work for you:

```
parent_ob.about_me()
```

It can produce the following output:

```
i=1
__j=10
```

Finally, ___private_show_me() is a private method. So, if you try to access it in the following way:

```
parent_ob.__private_show_me() #will cause error
```

You see the error again, which is like the following:

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter11/private_variable_and
line 23, in <module>
    parent_ob.__private_show_me() #will cause error
AttributeError: 'Parent' object has no attribute '__private_show_me'
```

Hope you have got the idea! Now I present you the complete demonstration with supportive comments. If you uncomment the corresponding codes, you will get errors. I have kept them for your reference purpose.

```
class Parent:
   """ This is the parent class"""

    def __init__(self):
      self.i = 1  #public variable
      self.__j = 10  #private variable

    def __private_show_me(self):
      print(f"i={self.i}")
      print(f"__j={self.j}")

    def about_me(self):
      print(f"i={self.i}")
      print(f"__j={self.__j}")
```

```
parent_ob = Parent()
print(f"parent_ob.i={parent_ob.i}")
#print(parent_ob.__j)#will cause error
print("Calling about_me() now:")
parent_ob.about_me()
#parent_ob.__private_show_me() #will cause error
```

Output

Here is the output.

```
parent_ob.i=1
Calling about_me() now:
i=1
__j=10
```

Analysis

Is there any way to access the private variables? Yes, you have. I’ve shown you one way in the previous demonstration when a public method could access the private data. Is there any alternative way? Yes, the following code can work for you:

```
print(parent_ob._Parent__j)#it works
```

it can print the output: 10 Notice this again. You can see that _parent_ob._Parent__j represents the format <instance_name>._Classname__privatevariable

Python saves the private variable in this different format. As a result, you cannot use this variable outside of the class in the usual way. We call this **name mangling**. This is useful to avoid the ambiguity of names defined by a subclass. In other words, if you do not want a subclass to use its parent class attributes, you can name those attributes with double leading underscores and no trailing underscore. Actually, in this context, Python allows you to use at most one trailing underscore, which means you can use a variable like __z or __z_.

Static Methods and Class Methods

Till now, you have seen the usage of instance methods. In OOP, they are very common. Now I show you static methods and Python-specific class methods. These are useful when you learn advanced programming in Python. It may seem complicated at the beginning, but a careful analysis can make things clear to you.

Let us begin with a class with a constructor and an instance method. You are familiar with this:

```
class Color:
    """ This is the parent class"""
    fav_color = "Green"

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

    def instance_method(self):
        print("I am an instance method.")
        print("Call me with an instance and a dot operator.")
        print(f"My favorite color is: {self.fav_color}")
        print("+"*15)
```

You understand that I can create an object of Color class and call the instance_method . Here is a sample code segment:

```
#Creating an object from Student class
my_color = Color("Blue")
print("Calling the instance_method() now.")
my_color.instance_method()
```

If you run this code, you can get the following output:

```
Calling the instance_method() now.
I am an instance method.
Call me with an instance and a dot operator.
My favorite color is: Blue
+++++++++++++++
```

Till this point, everything is straightforward. Now I introduce a static method inside the Color class. You can see the static methods in other languages too, such as Java, C#. Read the following points for static methods in Python:

- You use a decorator @staticmethod before the method definition.
- This time you do not use self parameter. Instead, you use the class name Color to invoke the variable fav_color .

```
@staticmethod
def static_method():
    print("I am a static method.")
    print("You can call me without a class instance.")
    print(f"My favorite color is: {Color.fav_color}")
    print("+" * 15)
```

The following code segment can invoke the static method as follows:

```
Color.static_method()
```

and produce the following output:

```
I am a static method.
You can call me without a class instance.
My favorite color is: Green
+++++++++++++++
```

Now I introduce a class method inside the Color class. It’s a special inclusion in Python programming. Notice the following points in this segment:

- You use a decorator @classmethod before the method definition.
- This time you do not use self parameter. Instead, you use cls with a dot operator to invoke the variable fav_color . It is similar when you invoke an instance method. The only difference is that instead of using the instance name, you use cls . I choose the name ‘cls’ following the convention. So, if you use any other name, that will work too, but experts do not recommend that.

```
@classmethod
def class_method(cls):
    print("I am a class method.")
    print("You can call me without a class instance.")
    print("In general, you use cls as the class parameter.")
    print(f"My favorite color is: {cls.fav_color}")
    print("+" * 15)
```

The following code segment can invoke the class method as follows:

```
Color.class_method()
```

and produce the following output:

```
I am a class method.
You can call me without a class instance.
In general, you use cls as the class parameter.
My favorite color is: Green
+++++++++++++++
```

Understanding the Difference

Let me change the color now. You can append the following code and run the program again.

```
print("Changing the color inside the instance now.")
my_color.fav_color = "Red"

print("Calling the instance_method() now.")
my_color.instance_method()

print("Calling the static_method() now.")
Color.static_method()

print("Calling the class_method() now.")
Color.class_method()
```

This time you can see these extra outputs:

```
Changing the color inside the instance now.
Calling the instance_method() now.
I am an instance method.
Call me with an instance and a dot operator.
My favorite color is: Red
+++++++++++++++
Calling the static_method() now.
I am a static method.
You can call me without a class instance.
My favorite color is: Green
+++++++++++++++
Calling the class_method() now.
I am a class method.
You can call me without a class instance.
In general, you use cls as the class parameter.
My favorite color is: Green
+++++++++++++++
```

Notice that the color " Red " is reflected when you use the instance method. But the static method or the class method did not change the color. It is because both the class method and the static method are bound to a class, but not to an object.

Now we do some further analysis. See the code examples of instance methods, static methods, and class methods again. Can you remember the functions in Chapter 7 ? If you compare these codes with those functions, you understand that my_color.instance_method() transforms the code to instance_method(my_color).

You called the static method and the class method using class names, which are as follows:

```
Color.static_method()
Color.class_method()
```

It is interesting to note that you can call them in an alternative way using the instance name. So, the following code will work too:

```
#Also works
my_color.static_method()
my_color.class_method()
```

You understand that behind the scene, my_color.static_method() actually transforms to static_method(). It means no additional argument is added in the function call.

But my_color.class_method() call transforms to class_method(type(cls)) . This is helpful when you use inheritance. It helps you to know about-which class calls this method. To understand this go through the following demonstration and output.

Demonstration 14

In this example, there are two classes- Parent and Child . As per their names, the Parent is a superclass and the Child is a derived class. The Parent class contains an instance method, a static method, and a class method. I use both a Parent class object and a Child class object to access the methods. In the output, you can see which type of object calls the class method. For example, you can notice the following lines in this output:

A class method is called.<class '__main__.**Parent**'>
A class method is called.<class '__main__.**Child**'>

Here is the complete demonstration:

```
class Parent:
    """ This is a Parent class."""

    def instance_method(self):
        print(f"An instance method is called.{self}")

    @staticmethod
    def static_method():
        print("A static method is called.")

    @classmethod
    def class_method(cls):
        print(f"A class method is called.{cls}")


class Child(Parent):
    """ This is a Child class."""
    pass
```

```
#Creating an object from Parent class
parent_ob = Parent()

print("Using the Parent class object.")
parent_ob.instance_method()

Parent.static_method()
parent_ob.static_method() # Also ok

Parent.class_method()
parent_ob.class_method() # Also ok

print("*"*20)

print("Using the Child class object.")
child_ob = Child()
child_ob.instance_method()

Child.static_method()
child_ob.static_method()# Also ok

Child.class_method()
child_ob.class_method()# Also ok
```

Output

Here is the output.

```
Using the Parent class object.
An instance method is called.<__main__.Parent object at 0x0170C058>
A static method is called.
A static method is called.
A class method is called.<class '__main__.Parent'>
A class method is called.<class '__main__.Parent'>
********************
Using the Child class object.
An instance method is called.<__main__.Child object at 0x0170C070>
A static method is called.
A static method is called.
```

```
A class method is called.<class '__main__.Child'>
A class method is called.<class '__main__.Child'>
```

Analysis

From this output, you can tell whether the Parent class object calls the class method or the derived class object calls it. You’ll find this mechanism useful when you make an advanced application. But static methods do not tell any such information. It is an important difference between a class method and a static method.

EXERCISE

E11.1 Can you predict the output of the following code segment?

```
class Student:
    class_name = "Student"

    def __init__(self, roll_number):
        self.roll_number = roll_number

    def about_me(self,name):
        """ A simple method """
        print(f"Hello,{name}!")
        print(f"You belong to {self.class_name} class!")
        print(f"Your roll number is:{self.roll_number}")


student_1 = Student(25)
student_1.about_me("Mike")
```

E11.2 Suppose you replace the following line in E11.1:

```
student_1 = Student(25)
```

with

```
student_1 = Student()
```

Can you predict the output?

E11.3 Can you predict the output of the following code segment?

```
class College:
    college = "Abc"

    def __init__(self,name,stud_name,roll_number):
        self.college = name
        self.stud_name = stud_name
        self.roll_number = roll_number
```

```
     def about_me(self):
      print(f"College name is: {self.college}")
      print(f"The student name is: {self.stud_name}")
      print(f"The roll number is: {self.roll_number}")


student_1 = College("St. Stephen","Bob",2)
student_1.about_me()
```

E11.4 Can you predict the output of the following code segment?

```
class Student:
   def __init__(self, roll_number, name="Mike"):
      self.name = name
      self.roll_number = roll_number

     def about_me(self):
      print(f"Name is: {self.name}")
      print(f"The roll number is: {self.roll_number}")


student_1 = Student("Bob",1)
print("\nStudent-1 detail:")
student_1.about_me()

print("\nStudent-2 detail:")
student_2 = Student(2)
student_2.about_me()
```

E11.5 Can you predict the output of the following code segment?

```
class Parent:
   @staticmethod
   def static_method():
      print("Static method inside Parent is called.")

     @classmethod
   def class_method(cls):
      print(f"Class method is called.Invoker type:{cls}")


class Child(Parent):
   """ This is a Child class."""

     @staticmethod
   def static_method():
      print("The static method inside Child is called.")


parent_ob = Parent()
child_ob = Child()
parent_ob.class_method()
child_ob.class_method()
parent_ob.static_method()
child_ob.static_method()
```

E11.6 Can you predict the output of the following code segment?

```
class ParentClass:
     @staticmethod
```

```
    def static_method():
        print("Static method inside Parent is called.")


class ChildClass(Parent):
    pass


ParentClass.static_method()
ChildClass.static_method()
```

E11.7 Can you predict the output of the following code segment?

```
ParentClass.static_method()
ChildClass.static_method()

class SampleClass:
   """ This is the parent class"""

    def __init__(self):
     self.i = 1
     self.__j = 10

    def about_me(self):
     print(f"i={self.i}")
     print(f"__j={self.__j}")


sample = SampleClass()
sample.about_me()
print(sample.i)
```

E11.8 Suppose, in E11.7, append the following line:

```
print(sample.__j)
```

Can you predict the output?

E11.9 You have seen the differences between a class method and a static method. Can you tell some similarities between them?

Solution to Exercises

E11.1

```
Hello,Mike!
You belong to Student class!
Your roll number is:25
```

E11.2

In this case, you'll receive the error like:

```
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter11/chap11_exercise.py",
line 19, in <module>
    student_1 = Student() #error. Ex-11.2
TypeError: __init__() missing 1 required positional argument: 'roll_number'
```

E11.3

In this case, you have updated the college name in the constructor, you see the updated college name as St. Stephen. Now you receive the following output.

```
College name is: St. Stephen
The student name is: Bob
The roll number is: 2
```

E11.4

In chapter7, you learned about default values. This code segment uses the same concept. So, when you do not pass the student name, it accepts the default name as Mike. But notice that you need to supply the roll number, which you cannot bypass.

```
Student-1 detail:
Name is: 1
The roll number is: Bob

Student-2 detail:
Name is: Mike
```

```
The roll number is: 2
```

E11.5
You can refer to the discussion about static methods and class methods in this chapter to understand the output.

```
Class method is called.Invoker type:<class '__main__.Parent'>
Class method is called.Invoker type:<class '__main__.Child'>
Static method inside Parent is called.
The static method inside Child is called.
```

E11.6
Commonly, you invoke a static method (or a class method) using the class name. You can refer to the discussion about static methods in this chapter to understand the output.

```
Static method inside Parent is called.
Static method inside Parent is called.
```

E11.7

```
i=1
__j=10
1
```

E11.8
This time you see an error.It is because you try to access a private data using an object in an illegal way. If needed, refer the discussions on private variables in this chapter.Here is the output.

```
i=1
__j=10
1
Traceback (most recent call last):
  File
"E:/MyPrograms/Python/PythonCrashCourse/chapter11/chapter11_exercise_f
line 19, in <module>
    print(sample.__j) #Error #Ex-11.8
AttributeError: 'SampleClass' object has no attribute '__j'
```

E11.9

Here are the similarities:

- Both the class method and the static method are bound to a class, but not to an object.
- In general, we use class names to invoke a class method or a static method. Remember that you cannot use a class name to invoke an instance method.

CS 8.1 Implementation

In this implementation, Engineering is the parent class. The ComputerScience class derives from this class. So, you see the following code:

```
class ComputerScience(Engineering):
    #remaining code skipped
```

Mathematics and Soft-skills are common subjects across all institutions. So, I initialize these two subjects in the Engineering class constructor (i.e., in its __init()__() method).

The default institution name and special subject (the third subject) is used in the derived class constructor. You call the superclass constructor to get the common subject's name. So, you see this:

```
def __init__(self, college_name="St. Stephen",
             special_subject="Compiler design."):
        """ Initialize starts from parent class."""
        super().__init__(college_name)
        self.subject_3 = special_subject
```

The remaining codes follow the specification of this project/case study. Here is the complete demonstration for you:

```
class Engineering:
    """Engineering is the parent class."""

      def __init__(self, college_name):
        # Initialize name
        self.college = college_name
        self.subject_1 = "Mathematics."
        self.subject_2 = "Soft-skills."


class ComputerScience(Engineering):
    """
    ComputerScience class inherits from Engineering.
    Default institution is St.Stephen.
    """

    def __init__(self, college_name=" St. Stephen",
                 special_subject="Compiler design."):
        """ Initialize starts from parent class."""
        super().__init__(college_name)
        self.subject_3 = special_subject

    def course_details(self):
        """Prints the course details of an institution."""
        print(f"Institution name:{self.college} college.")
        print("Computer science course includes:")
        print(f"1:{self.subject_1}")
        print(f"2:{self.subject_2}")
        print(f"3:{self.subject_3}")
        print("-" * 10)


# Computer science course at Presidency College
cs_course1 = ComputerScience(" Presidency",
                             "Python Programming")
cs_course1.course_details()

# Computer science course at St.Stephen College
cs_course2 = ComputerScience()
cs_course2.course_details()
```

Possible improvements

- To reduce the code size, I skipped the exception handling mechanism and user inputs in this project implementation. You can consider them to make a better implementation. I leave this improvement exercise to you.

Chapter 12: Test Your Code

Testing software is always important. It gives us confidence. The presence of test teams is not limited to software organizations. You see them everywhere. These teams test a product in various ways and generate the reports. It's possible, there are different modules in software and you work only in a particular module. So, even if you write the correct code, your job is not over. The test teams can integrate your code with other's code to verify the software. But every time you cannot wait for the entire test report to come. You first want to know whether your code works fine. So, you can write some small tests. You term them as unit tests. These small tests can check whether a class or a function shows the expected behavior. This chapter gives you a quick overview of this topic.

I repeat: there are different types of tests. The detailed coverage of them is beyond the scope of this chapter. This chapter shows you the tests which help you examine the functions and classes in your code. Here you'll also learn how to run many tests together.

The Code for Testing

You write tests to verify some code. So, at first, you need to have the code. (It is possible that you write test cases in advance when the development code is not ready. I ignore this possibility here). My primary goals are to show you:

- How to write a test for a function.
- How to run this test.
- How to run multiple tests together.

I start with a module that has two functions inside it. I name the module my_library_functions.py . I store it inside the main directory, PythonCrashCourse . The code inside the module is simple. The supporting documentations tell you :

- The make_total_double() accepts two numbers and returns double of their sum.

- The make_avarage() function takes three numbers and returns the average of them. In this function, the first two are compulsory arguments. But the final one is an optional argument.
- To make these functions simple and easy, I exclude the validation part.

Here are the contents of the module my_library_functions.py

```
# I write test cases to verify the following functions.

""" Support to use custom functions.

Functions
---------
make_total_double()
    This function takes two numbers and returns the double.


make_total_double()
    This function takes three numbers and returns the average.
    The third number is optional.


"""


def make_total_double(first,second):
    """
    This function takes two numbers and returns the double.
    """
    total = first + second
    return total * 2


def make_average(first,second,third=0):
    """
    This function takes three numbers and returns the average.
    The third number is optional.
    """
    total = first + second + third
    return total / 3
```

POINT TO REMEMBER

In chapter7, I told you I’ve organized the codes for this book chapter wise. So, I store all programs for chapter12 inside a directory named chapter12. The parent directory of the chapter12 is PythonCrashCourse. Suppose I store the file my_library_functions.py inside the chapter12 directory. In that case, I cannot use the following line:

```
from my_library_functions import make_total_double
```

Instead, I need to use it as follows:

```
from chapter12.my_library_functions import make_total_double
```

So, to avoid less typing, I follow this structure. This helps me to refer to the module without mentioning the chapter12.

Testing Functions

But before you write your test code, import the unittest module. To test a function from the specified location, you need to import the function too. So, you see the following lines at the beginning:

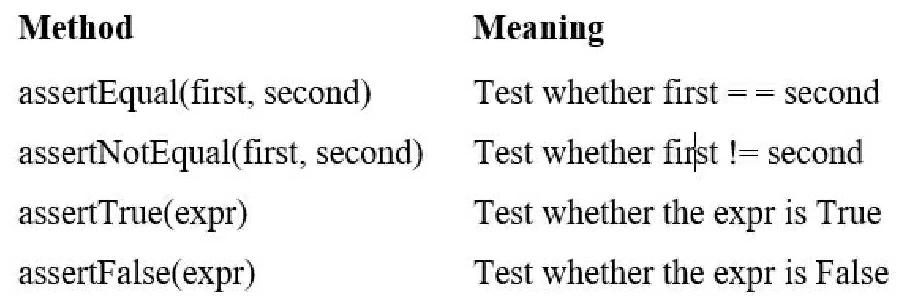
```
import unittest
from my_library_functions import make_total_double
```

Then you create a class with a meaningful name. Its better to include the word Test in this context. This class should inherit from unittest.TestCase . So, you see the following line in the upcoming code:

```
class TestEmployee(unittest.TestCase):
```

Now you are ready to write your test. It’s better to choose a name which starts with ‘test’. Inside the test_make_total_double () function, you see the use of *assert* methods. In testing, assertions are very common. You can use various assertion methods. These methods help you test the actual value against the expected value. Table 12-1 shows you four of them. These are enough for you to understand the content of this chapter.

Table 12-1 The assert methods that you can use in Chapter12



Method	Meaning
assertEqual(first, second)	Test whether first = = second
assertNotEqual(first, second)	Test whether first != second
assertTrue(expr)	Test whether the expr is True
assertFalse(expr)	Test whether the expr is False

For example, in the following tests, I pass two numbers- 25 and 75.3 . If you add them and double this sum, you should get(25+75.3)*2=200.6 . So, I write the following code:

```
self.assertEqual(200.6, resultant_value)
```

Now if the function returns any other value, the test containing this statement will fail. You can use an optional message inside this method. This message can help you when the test fails. For example, if you write the following:

```
self.assertEqual(200.6+1, resultant_value, "The actual value and expected value does not match")  #test fails
```

The test will fail and display this message. I pick a sample output fragment for your reference. Notice the bold lines in this segment:

```
Launching unittests with arguments python -m unittest E:/MyPrograms/Python/PythonCrashCourse/chapter12/run_single_test.py in E:\MyPrograms\Python\PythonCrashCourse\chapter12
FAILED (failures=1)

The actual value and expected value does not match
200.6 != 201.6
```

```
Expected :201.6
Actual   :200.6
<Click to see difference>

Traceback (most recent call last):
  File "C:\Program Files\JetBrains\PyCharm Community Edition 2020.1.1\plugins\python-ce\helpers\pycharm\teamcity\diff_tools.py", line 32, in _patched_equals
    old(self, first, second, msg)
  File "C:\Users\Vaskaran Sarcar\AppData\Local\Programs\Python\Python38-32\lib\unittest\case.py", line 912, in assertEqual
    assertion_func(first, second, msg=msg)
  File "C:\Users\Vaskaran Sarcar\AppData\Local\Programs\Python\Python38-32\lib\unittest\case.py", line 905, in _baseAssertEqual
    raise self.failureException(msg)
AssertionError: 201.6 != 200.6 : The actual value and expected value does not match
…
```

One last point for you: You'll see the following if block in this example:

```
if __name__ == '__main__':
    unittest.main()
```

The special variable __name__ is set when the program executes. When you run this file as the main program, the __name__ is set to __main__ . Here, you call unittest.main() to run the test case. A test framework can work differently. If a test framework imports this file, the __name__ may not set to __main__ . In that case, it will not execute this block.

Demonstration 1

I have created the file run_single_test.py which has a single test. Here is the content:

```
import unittest
from my_library_functions import make_total_double


class LibraryFunctionsTestCases(unittest.TestCase):
    """ Tests for my_library_functions """

    def test_make_total_double(self):
        """ Can we make the total double for the numbers 25,75.3 """
        resultant_value = make_total_double(25, 75.3)
        self.assertEqual(200.6, resultant_value)

if __name__ == '__main__':
    unittest.main()
```

Output

Here is a sample output:

```
Launching unittests with arguments python -m unittest
E:/MyPrograms/Python/PythonCrashCourse/chapter12/run_single_test.py in
E:\MyPrograms\Python\PythonCrashCourse\chapter12


Ran 1 test in 0.023s

OK

Process finished with exit code 0
```

Demonstration 2

This demonstration shows you can execute multiple tests together. In real-world programming, this is a common practice. I have created a new file run_multiple_tests.py which has three tests. Now I import everything from my_library_functions.py . So, I can access both the methods- make_total_double() and make_average() in my current program file. Here is the content of this file:

```
import unittest
```

```
# importing everything from the module
import my_library_functions as ml

class LibraryFunctionsTestCases(unittest.TestCase):
    """ Tests for my_library_functions """

      def test_make_total_double(self):
        """ Can we make the average of
        the numbers 25,75.3 """
        resultant_value = ml.make_total_double(25, 75.2)
        self.assertEqual(200.4, resultant_value)  #test passes

      def test_make_average_two_numbers(self):
        """ Can we make the the average of the
        numbers 25,75.8,and 0(optional)?"""
        resultant_value = ml.make_average(25, 75.8)
        self.assertEqual(33.6, resultant_value) # test passes
        #self.assertEqual(40, resultant_value) # test fails

      def test_make_average_three_numbers(self):
        """ Can we make the average of the
        numbers 10,20.9 and 30 ?"""
        resultant_value = ml.make_average(10, 20.9, 30)
        self.assertEqual(20.3,resultant_value)


if __name__ == '__main__':
    unittest.main()
```

I remind you that you may not see the correct indentation on your device. So, if required, you can refer to the actual code. It is available online at https://github.com/Vaskaran/PythonBookcamp.I told you about this link at the beginning of this book. You can download the complete code from there. The same comment applies for all code in this book.

Output

Here is the sample output from my computer:

```
Testing started at 10:15 ...
E:\MyPrograms\Python\PythonCrashCourse\venv\Scripts\python.exe
"C:\Program Files\JetBrains\PyCharm Community Edition
2020.1.1\plugins\python-ce\helpers\pycharm\_jb_unittest_runner.py" --path
E:/MyPrograms/Python/PythonCrashCourse/chapter12/run_multiple_tests.py

Ran 3 tests in 0.003s

OK
Launching unittests with arguments python -m unittest
E:/MyPrograms/Python/PythonCrashCourse/chapter12/run_multiple_tests.py
in E:\MyPrograms\Python\PythonCrashCourse\chapter12


Process finished with exit code 0
```

Analysis

Let us deliberately fail one test case and check the output. It can depict a common scenario when you run multiple tests together. To make the test fail, I replace the following line:

```
self.assertEqual(33.6,resultant_value) # test passes
```

with the following one:

```
self.assertEqual(40,resultant_value) # test fails
```

And execute the tests again. You can see the following output. Once again, I highlight the important lines in bold.

```
Testing started at 10:17 ...
E:\MyPrograms\Python\PythonCrashCourse\venv\Scripts\python.exe
"C:\Program Files\JetBrains\PyCharm Community Edition
2020.1.1\plugins\python-ce\helpers\pycharm\_jb_unittest_runner.py" --path
E:/MyPrograms/Python/PythonCrashCourse/chapter12/run_multiple_tests.py
Launching unittests with arguments python -m unittest
```

```
E:/MyPrograms/Python/PythonCrashCourse/chapter12/run_multiple_tests.py
in E:\MyPrograms\Python\PythonCrashCourse\chapter12


33.6 != 40

Expected :40
Actual   :33.6
<Click to see difference>

Traceback (most recent call last):
  File "C:\Program Files\JetBrains\PyCharm Community Edition
2020.1.1\plugins\python-ce\helpers\pycharm\teamcity\diff_tools.py", line 32,
in _patched_equals
    old(self, first, second, msg)
  File "C:\Users\Vaskaran
Sarcar\AppData\Local\Programs\Python\Python38-32\lib\unittest\case.py",
line 912, in assertEqual
    assertion_func(first, second, msg=msg)
  File "C:\Users\Vaskaran
Sarcar\AppData\Local\Programs\Python\Python38-32\lib\unittest\case.py",
line 905, in _baseAssertEqual
    raise self.failureException(msg)
AssertionError: 40 != 33.6

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "C:\Users\Vaskaran
Sarcar\AppData\Local\Programs\Python\Python38-32\lib\unittest\case.py",
line 60, in testPartExecutor
    yield
  File "C:\Users\Vaskaran
Sarcar\AppData\Local\Programs\Python\Python38-32\lib\unittest\case.py",
line 676, in run
    self._callTestMethod(testMethod)
  File "C:\Users\Vaskaran
```

```
Sarcar\AppData\Local\Programs\Python\Python38-32\lib\unittest\case.py",
line 633, in _callTestMethod
    method()
  File
"E:\MyPrograms\Python\PythonCrashCourse\chapter12\run_multiple_tests.py
line 19, in test_make_average_two_numbers
    self.assertEqual(40, resultant_value) # test fails


Ran 3 tests in 0.015s

FAILED (failures=1)

Process finished with exit code 1

Assertion failed

Assertion failed

Assertion failed
```

Testing Class

The first two demonstrations in this chapter show you how to write tests for functions. Now you see tests for class. This time, I do not create a new class. Instead, I reuse the employee_module , which you saw in Chapter 11. This module has two classes- Employee and PersonalDetails . Here I write a test to verify the instance creation process for the Employee class. I import this class to my current location before I write a test for it. This test is useful because if you cannot create the instance of the class, there is no use for it.

Demonstration 3

I have created the file test_employee_class.py which has the following content:

```
import unittest
from employee_module import Employee
```

```
class TestEmployee(unittest.TestCase):
    """ Tests for Employee class in employee_module """

    def test_employee_creation(self):
        """ Can we store the Employee details correctly? """
        emp_john = Employee('John','E001')
        self.assertEqual(emp_john.name, 'John', 'Employee name does not
match.')
        self.assertEqual(emp_john.emp_id, 'E001', 'Employee ID does not
match.')
        self.assertEqual(emp_john.company, 'Abc Ltd.', 'The company
information is not found.')



if __name__ == '__main__':
    unittest.main()
```

Output

Here is a sample output from my computer:

```
Testing started at 11:05 ...
E:\MyPrograms\Python\PythonCrashCourse\venv\Scripts\python.exe
"C:\Program Files\JetBrains\PyCharm Community Edition
2020.1.1\plugins\python-ce\helpers\pycharm\_jb_unittest_runner.py" --path
E:/MyPrograms/Python/PythonCrashCourse/chapter12/testing_employee_clas
Launching unittests with arguments python -m unittest
E:/MyPrograms/Python/PythonCrashCourse/chapter12/testing_employee_clas
in E:\MyPrograms\Python\PythonCrashCourse\chapter12



Ran 1 test in 0.003s

OK

Process finished with exit code 0
```

EXERCISE

E12.1 You saw that the Employee class in employee_module has a method get_details() which simply prints some information. Let us change the method with a return statement as follows (I kept the old codes commented for your reference):

```
def get_details_modified(self):
    """
    Modifying this method with return
    value for testing in Chapter-12
    """
    #print("The current employee details are as follows:")
    #Formating the output using f-string
    #print(f"Name:{self.name},Id:{self.emp_id},Company:
{self.company}")
    return f"Name:{self.name},Id:{self.emp_id},Company:
{self.company}"
```

Can you write a test to verify this modified method in the Employee class?

E12.2 Can you write a test case to test some in-built Python functions?

Solution to Exercises

E12.1

```
import unittest
from employee_module import Employee


class TestEmployeeMethod(unittest.TestCase):
    def test_get_details_modified(self):
        """ Does the get_details() work correctly? """
        emp_kate = Employee('Kate', 'E002')
        self.assertTrue('Kate' in emp_kate.get_details_modified(), 'The
employee name is not found.')
        self.assertTrue('E002' in emp_kate.get_details_modified(), 'The
employee id is not found.')
        self.assertTrue('Abc Ltd.' in emp_kate.get_details_modified(), 'The
company information is not found.')


if __name__ == '__main__':
    unittest.main()
```

E12.2

```
import unittest
from math import sqrt, floor, gcd


class TestEmployeeMethod(unittest.TestCase):
    def test_inbuilt_functions(self):
        """ Do the methods sqrt(),floor(), and gcd() correctly? """
        self.assertEqual(5.0, sqrt(25), 'The sqrt(25) does not give the expected
result.')
        self.assertEqual(100, floor(100.2), 'The floor(100.2) does not give the
expected result.')
        self.assertEqual(2, gcd(4, 10), 'The gcd(4,10) does not give the expected
result.')
```

```
if __name__ == '__main__':
    unittest.main()
```

Index

A

absolute path, 212, 213
Abstraction, 239, 241
Arguments, 154
assertEqual, 295, 296, 297, 299, 300, 301
assertFalse, 295
assertNotEqual, 295
assertTrue, 295, 304
Associativity, 39

B

break Statement, 103

C

Case Study, 47, 74, 91, 113, 148, 178, 188, 189, 191, 192, 194, 196, 198, 209, 243
Class, 187, 236, 241, 245, 278, 301
Class Methods, 278
Comments, 15, 16, 46
continue Statement, 106
Contradictions, 82

D

Default Attributes, 257
Default Values, 154
Demonstration, 10, 13, 14, 16, 20, 68, 75, 76, 78, 79, 80, 82, 83, 84, 97, 99, 101, 104, 106, 108, 151, 156, 158, 159, 161, 165, 167, 187, 191, 193, 195, 199, 201, 215, 217, 219, 221, 222, 223, 225, 248, 251, 253, 255, 258, 261, 263, 264, 270, 273, 275, 282, 296, 297, 301
Dictionaries, 135

E

elif, 75, 78, 79, 80, 81, 82, 83, 84, 86, 88
else, xxiii, 23, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 90, 111, 173, 175, 193, 195, 196, 198, 207, 227, 228, 231, 275
Encapsulation, 238, 241

F

FileNotFoundError, 219, 225, 226, 227, 228, 229

float, 18, 19, 20, 22, 23, 24, 59, 60, 70, 71, 79, 80, 83, 84, 110, 178, 179, 236
for Loop, 98
f-strings, 60, 63
Function Arguments, 153
Functions, 149, 293, 294

I

Identity Operator, 266
IDLE, xxii, 3, 4, 5, 6, 12, 50
if-else, 75, 76, 78, 79, 83
inheritance, 269
Inheritance, 239, 241, 269
Instance, 247, 253
int, 18, 19, 20, 22, 24, 25, 59, 60, 61, 71, 77, 85, 88, 130, 188, 189, 191, 192, 193, 194, 195, 197, 198, 200, 205, 206, 229, 236
Iteration, 93

K

Keyword, 154, 223

L

Lambda, 160
Lists, 114

M

Membership, 29, 40
Modules, 149, 162

N

name mangling, 278
Numbers, 18, 19, 58
numpy, 168

O

Objects, 236, 241, 245, 246
Operators, 15, 28, 29, 31, 32, 33, 34, 38, 39, 40, *See*

P

pass Statement, 195
pip, xi, 168
Polymorphism, 240, 241
Positional, 153

Positional Argument, 153
PyCharm, xxi, xxiii, 5, 6, 7, 9, 10, 11, 12, 53, 62, 63, 126, 150, 183, 184, 295, 298, 299, 302

Q

Q&A Session, 241

R

Relative paths, 212
remove, 24, 37, 120, 121, 122, 132, 144, 159, 218, 222, 226
rename, 222, 226

S

Sets, 138, 139
Solution, 43, 70, 88, 109, 110, 144, 171, 204, 227, 288, 304
Solution to Exercises, 43, 70, 88, 109, 171, 204, 227, 288, 304
Static Methods, 278
Strings, 18, 49

T

Tautology, 82
Testing, 16, 125, 292, 294, 298, 299, 301, 302
Tuples, 131, 134

V

Variables, 15, 17, 27, 28

W

while Loop, 94

Z

zone, xviii