

Python Programming for Students

Explore Python in multiple dimensions with project-oriented approach



Nidhi Grover Raheja

bpb



Python Programming for Students

Explore Python in multiple dimensions with project-oriented approach



Python Programming for Students

*Explore Python in multiple dimensions with
project-oriented approach*

Nidhi Grover Raheja



www.bpbonline.com

Copyright © 2024 BPB Online

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

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

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

First published: 2024

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55516-084

www.bpbonline.com

Dedicated to

My Family:

***Rakesh Kumar Grover with Saroj Grover & Isha
Ashok Kumar Raheja and Renu Raheja***

My husband:

Sameer Raheja

&

*Our daughter **Cherika***

About the Author

Nidhi Grover Raheja is actively working as Technical Trainer in the domains of Python Programming, Data Analytics and Visualization Tools. She possesses over a decade experience and is associated with numerous reputed educational and training institutions in the role of Technical Trainer and Guest Lecturer. She qualified UGC-NET (Lectureship) and GATE in Computer Science. After completing MCA from GGSIPU, Delhi she accomplished M.Tech (CSE) from DCRUST, Sonipat with Distinction. Her interest areas include Python programming with Machine Learning, Deep Learning, Natural Language Processing, Statistical Analysis and Visualization Tools including Tableau and Microsoft Power BI. She endeavors not only training students with experiential learning approach but also continuously tries to shape up their careers with best of skills and knowledge as per standards.

About the Reviewer

Shivkumar Ramanna Chandey is a seasoned technical reviewer with an insatiable passion for cutting-edge technology and a knack for dissecting complex concepts. He currently works as an Assistant Professor in the Department of Computer Science and Information Technology at Nirmala Memorial Foundation College of Commerce and Science, Kandivali East – 400101, affiliated to the University of Mumbai. With more than 05 years of experience in the tech teaching industry, he has honed his skills to ensure that every product, project, or manuscript he reviews receives the meticulous attention it deserves.

He has written various research papers that have been published in reputed International Journals, National Journals, and Conference Proceedings. His area of research includes Cloud Computing, Digital Communication, Blockchain, Cyber Security, Open-Source Software, etc.

By keeping the motto of “Learn Something About Everything and Everything About Something,” he has explored various technical fields. His expertise spans a wide spectrum of technologies, including Cloud Computing, Data Science, Blockchain, Web Services, Cyber Security, Web Development, Various Programming Languages, DBMS, Linux, and FOSS, still exploring and learning new technologies to keep himself updated in this digital world. He has flexibility in technical areas and utilizes these skills to solve problems by making use of intellectual thinking.

Acknowledgement

I want to express my deepest gratitude to my family and friends for their unwavering support and encouragement throughout this book's writing, especially my parents, my husband & my daughter for their unconditional love and support.

I am also grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. It was a long journey of revising this book, with valuable participation and collaboration of reviewers, technical experts, and editors.

I would also like to acknowledge the valuable contributions of my colleagues and co-worker during many years working in the education industry, who have taught me so much and provided valuable feedback on my work.

Finally, I would like to thank all the readers who have taken an interest in my book and for their support in making it a reality. Your encouragement has been invaluable.

Preface

Python is a powerful, versatile programming language. Python is challenging other programming languages like Java, C#, etc. with its simple syntax and wide range of applications.

Python is a very promising programming language in today's rapidly evolving technological landscape thanks to its applicability in a wide range of domains, including task-specific python programs, standalone GUI applications, creating sustainable websites, creating interactive games, data analytics and machine learning, artificial intelligence, etc.

This book gives readers the opportunity to learn all facets of Python programming through the use of clear and engaging examples, practical codes, examples of completed projects, and exercises based on unsolved assignments.

Each project presented in the book offers a taste of a real-world approach to problem solving while providing the advantages of experiential learning, which allows readers to learn by doing. Readers will enjoy learning Python thanks to the abundance of examples, programming illustrations, and relevant project assignments.

Chapter 1: Getting Started with Programming in Python – This segment will aid the readers to learn Python programming in a quick and easy way through a series of simple interesting examples. In this chapter the set-up process of the Python development environment is discussed with illustrative screenshots. Furthermore, readers will also learn how to write variables, literals, keywords, and comments in Python. Learners will also get a deep insight upon various data types, Input-Output process, types of operators, type-conversions, and Namespace in Python.

Chapter 2: Flow Control Concepts – This chapter introduces the audience to the core concepts flow control and its types in Python. Readers will walk

through various working examples of conditional decision-making flow control using if...else statement and iterative flow control using loops in Python. This chapter will lay a firm base for developing problem solving approaches while writing more complicated applications in Python.

Chapter 3: Data Structures and Algorithms – This chapter gives an overview of data structures and related algorithms. Here readers will get an overview of the fundamental data structures supported in Python such as List, Tuple, Set, Dictionary and Comprehensions. The later sections of this chapter describe various sorting and searching algorithms implemented in Python. By the end of this unit readers will be able to apply different algorithmic approaches to solve real time problems at hand.

Chapter 4: Functions in Python – This chapter is based on creating, calling, and managing functions in Python. The readers will learn about predefined functions in Python and create customizable user-defined functions for a specific functionality. Here, the learners will also get an introduction to recursion approach of writing functions. The later section of this chapter deals with creating and managing modules and packages in Python.

Chapter 5: Object-oriented Programming Concepts – This section focuses on the major concepts of Object-Oriented Programming approach including class & Object, Data Hiding, Data Abstraction, Inheritance and its types, Polymorphism and basics of overloading using Python. This chapter will lead the users towards the path of real time project development.

Chapter 6: Turtle Programming in Python – This section deals with creating graphics using Turtle library in Python. After reading this chapter the learners will be able to draw different shapes, fill colours and create attractive designs using Python and Turtle library. Furthermore, learners will get a glimpse of creating animated Turtle graphics in Python.

Chapter 7: Database Handling Using SQLite – This section deals with the creation and management of data using SQLite database with Python. Here the focus will be to integrate SQLite3 module with Python for developing real time database applications. By the end of this chapter

readers will be able to develop CRUD applications in Python using SQLite database.

Chapter 8: GUI Application Development Using Tkinter – This unit deals with developing Graphical User Interface (GUI) applications using Tkinter library. Python collectively with Tkinter provides a quick and easy way to create GUI applications. Throughout this unit readers will walk through numerous examples of developing standard GUI based desktop applications.

Chapter 9: Game Development with PyGame – This section takes us into the fascinating world of game development using the PyGame library. This chapter will help readers to learn the PyGame library from basic to advance with the help of simple and well-explained examples. After reading this unit readers will be able to develop simple games in Python.

Chapter 10: Mobile App Development with Kivy – This section deals with creating simple mobile applications in Python using the Kivy library in Python. Here readers will learn the basics of mobile application by creating simple applications and creating .apk files for the same. These .apk files created will help to users to deploy and use mobile applications in android phones.

Chapter 11: Image and Video Processing with Python – This chapter showcases the techniques of manipulating images and video frames. After reading this unit, readers will be able to modify images and videos with ease. Image and video processing is necessary in several multimedia applications. Therefore, this chapter will enable users to manipulate images and videos for such applications.

Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/9f68d3>

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

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

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at

www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

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

Piracy

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

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

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

For more information about BPB, please visit www.bpbonline.com.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Table of Contents

1. Getting Started with Programming in Python

- Introduction
- Structure
- Objectives
- Features of Python
- Installing Python
- Keywords
- Identifier
- Comments
- Variable and data types
 - Datatypes in Python*
 - Numbers*
 - Dictionary*
 - Boolean*
 - Set*
 - Sequence types*
 - Type conversion in Python
 - Implicit type conversion*
 - Explicit type conversion*
 - Input/output using Python
 - Output formatting using format()*
 - Operators and expressions
 - Arithmetic operators*
 - Assignment operators*

Comparison operators
Logical operators
Bitwise operators
Identity operators
Membership operators
Operator precedence

Namespaces in Python

Conclusion

Points to remember

Exercise

Sample project with solution
Practice project

2. Flow Control Concepts

Introduction

Structure

Objectives

Decision-making in Python

Workflow of if...elif...else

The if statement

The if...else statement

The if...elif...else statement

Nested if

Loop control in Python

Python for Loop

Python while Loop

Infinite while Loop

Nested loop

Flow control statements in Python

Conclusion

Points to remember

Exercise

Sample project with solution

Practice project

3. Data Structures and Algorithms

Introduction

Structure

Objectives

Introduction to PyCharm IDE

Installation steps of PyCharm IDE

Built-in data structures

String

List

Tuple

Set

Python set operations

Dictionary

User-defined data structures

Linked list

Stack

Queue

Sorting algorithms

Time complexity and space complexity

Bubble sort

Selection sort

Insertion sort

Searching algorithms

Linear search

Binary search

Conclusion

Points to remember

Exercise

Sample project with solution

Practice project

4. Functions in Python

Introduction

Structure

Objectives

Introduction to functions

Benefits of using functions

Functions versus methods

Types of Python function

Function declaration and calling

Function arguments

Types of function arguments

Default arguments

Keyword arguments

Required arguments

Variable-length arguments

Recursion in Python

Anonymous functions

Example 1: use of lambda function to find the maximum of two numbers

Example 2: To filter out only even numbers from a list of numbers

Example 3: To find the cube of all elements in a list

Scope and lifetime of variables

Modules and packages

Conclusion

Points to remember

Exercise

Sample project with a solution

Practice project

5. Object-oriented Programming Concepts

Introduction

Structure

Objectives

Introduction to programming paradigms

Procedural programming

Object-oriented programming

Object-oriented programming concepts

Class and objects

Class attributes and methods

Built-in attributes of class

Constructors in Python

Parameterized constructor

Non-parameterized constructor

Default constructor

Encapsulation and data hiding

Inheritance in Python

Types of Inheritance

Single Inheritance

Multilevel Inheritance

Hierarchical Inheritance

Multiple Inheritance

Hybrid Inheritance
Method Resolution Order
super() in Python
 Super function in single inheritance
 Super function in multiple inheritance

Polymorphism in Python
 Compile-time Polymorphism
 Method and constructor overloading
 Operator overloading
 Run-time polymorphism

Conclusion

Points to remember

Exercise

Sample project with solution
 Practice project

6. Turtle Programming in Python

Introduction

Structure

Objectives

Turtle programming in Python

Plotting with Turtle

Creating shapes with Turtle

Drawing connecting lines

Draw square, rectangle, and triangle

Draw star pentagon, hexagon, and octagon

Draw circle and oval

Draw spiral

Fill colors in shapes

Event programming with Turtle

Mouse events

Key events

Conclusion

Points to remember

Exercise

Sample project with solutions

Practice project

7. Database Handling Using SQLite

Introduction

Structure

Objectives

Introduction to data and database

Relational versus non-relational database

Relational databases

Non-relational databases

SQLite for database handling

Downloading SQLite

Installing SQLite in command-line

GUI tools for SQLite

SQLite working in Python

Connecting SQLite database in Python

Datatypes in SQLite

Exception handling tasks

Database management with SQLite

Create new database

Commands in SQLite

Data Definition Language (DDL) commands—CREATE, ALTER, and DROP

SQLite table constraints

Data Manipulation Language (DML) commands

Data Query Language (DQL) command—SELECT

Clauses in SQLite commands

Aggregate functions

Joins in SQLite

Parameterized Query and sub-queries in SQLite

BLOB and DATE TIME in SQLite

Conclusion

Points to remember

Exercise

Sample project with solution

Practice project

8. GUI Application Development Using Tkinter

Introduction

Structure

Objectives

GUI programming in Python

Getting started with Tkinter

Introducing Tkinter widgets

Button

Label

Entry widget

Text widget

Radiobutton

Checkbutton

Combobox
Listbox
Menu
Spinbox
The Treeview Widget and Treeview Scrollbar
 Example 1
 Example 2
Label frame
Messagebox
Tkinter filedialog
 Returning a file path
 Saving a file
 Select directory
Geometry management in Tkinter
 Organizing widgets with layout managers
 Pack layout
 Grid layout
 Place layout
Event binding
Conclusion
Points to remember
Exercise
 Sample project with solution
 Practice project

9. Game Development with PyGame

Introduction
Structure
Objectives
Introduction to PyGame

Installing PyGame

Getting started with PyGame

Color object in pygame

Surface and shapes in pygame

Images in pygame

Events in pygame

Adding text and music in pygame

Sprites and collisions

Conclusion

Points to remember

Exercise

Sample project with solution

Practice project

10. Mobile App Development with Kivy

Introduction

Structure

Objectives

Introduction to Kivy

Characteristics of Kivy

Installation of Kivy

Kivy app life cycle

Widgets and layouts

UX widgets, events, and binding function

Geometry management using layout managers

Basics of KV language

Loading the KV File

Syntax guidelines for KV file

Modules and widgets

Events and properties

Dynamic class

Widget reference

User pages with multiple screens

Package Kivy applications with buildozer

Conclusion

Points to remember

Exercise

Sample project with solution

Practice project

11. Image and Video Processing with Python

Introduction

Structure

Objectives

Introduction to image processing

Manipulating images

Image processing libraries in Python

Reading an image

Grayscale conversion and image blurring

Image edge detection

Object detection in image

Image resize and rotation

Image addition, subtraction, and blending

Video processing tasks in Python

Capture Live video from Webcam

Conclusion

Points to remember

Exercise

Sample project with solution

Practice project

Appendix

Multiple choice questions

Solutions

Index

CHAPTER 1

Getting Started with Programming in Python

Introduction

In this chapter, we will discuss the basics of Python programming language. Python is an object-oriented high-level programming language that is easy to write and understand, more interactive, interpreted, and meant for general purposes. Guido van Rossum designed “Python” at **Centrum Wiskunde & Informatica (CWI)** in the Netherlands and released its first version in 1991. Guido Van Rossum, being a fan of a famous TV show in The Netherlands, “Monty Python’s Flying Circus,” named the language after **Monty Python**. Python is a powerful scripting language but can be used as an efficient programming language and also to develop a variety of applications. Python has been an open-sourced language since the beginning, and its source code is also available under the **GNU General Public License (GPL)**.

Structure

In this chapter, we will discuss the following topics:

- Features of Python
- Installing Python
- Keywords
- Identifier
- Comments
- Variable and data types

- Type conversion in Python
- Input/output using Python
- Operators and expressions

Objectives

By the end of this chapter, the readers will know the important characteristics of Python that make it a popular general-purpose language among users. They will be able to set-up a Python development environment in their systems, making them ready for programming. This chapter focuses on Python basics such as defining variables, literals, keywords, expressions, and comments. By learning about various data types, type conversion concepts, input/output processes, and different operators, readers will be able to adopt problem-solving approaches and write simple beginner-level programs in Python.

Features of Python

Python is a scripting language, being interpreted as a high-level programming language, developed for the purpose of fulfilling general programming requirements. The following features of Python make it very popular among its users:

- **Easy to understand:** When we read or write the Python program, we can feel like reading or writing simple English statements. This makes it a beginner's choice of language.
- **Multipurpose in nature:** Python is a general-purpose language that enables the development of a wide variety of applications such as text processing applications, Web programming, machine learning applications, games, and so on.
- **Python is interpreted:** We are not required to compile Python programs explicitly. Internally, the Python interpreter will take care of the compilation process as well.
- **Open-source language:** We can use Python software without any license, and it is freeware. Its source code is open so that we can

customize it based on our requirements.

- **Dynamically typed:** In Python, we are not required to declare the type for variables before assigning values. Rather, the type of value assigned to the variable will determine its datatype automatically. Hence, Python is considered a dynamically typed language.
- **Automatic memory management:** Python removes those objects that are no longer in use. It frees up the memory space occupied by such objects automatically using an internal Garbage Collector.
- **Supports platform independence:** Once we write a Python program, it can run on any platform without rewriting it once again, thus providing the feature of Platform Independence.
- **Platform portability:** Python programs are portable, that is, we can migrate from one platform to another very easily. Python programs will provide the same results on any platform.
- **Support for libraries:** Support for various third-party tools and utilities is available.

Installing Python

For most of Unix systems, Linux, and MAC OS, Python is pre-installed. For Windows Operating Systems, users can easily download the latest Python release from its official download page

<https://www.python.org/downloads/>. The code given in this book is implemented on Python 3 release Python 3.11.0. We can click on the desirable operating system; for example, click Windows to view download options for the selected operating system. As per your system configurations, you may choose to download Installer for 32-bit or 64-bit operating systems. This will automatically begin downloading the installer executable file. Once the download is complete, click the executable file to start the installation process as shown in *Figure 1.1*:

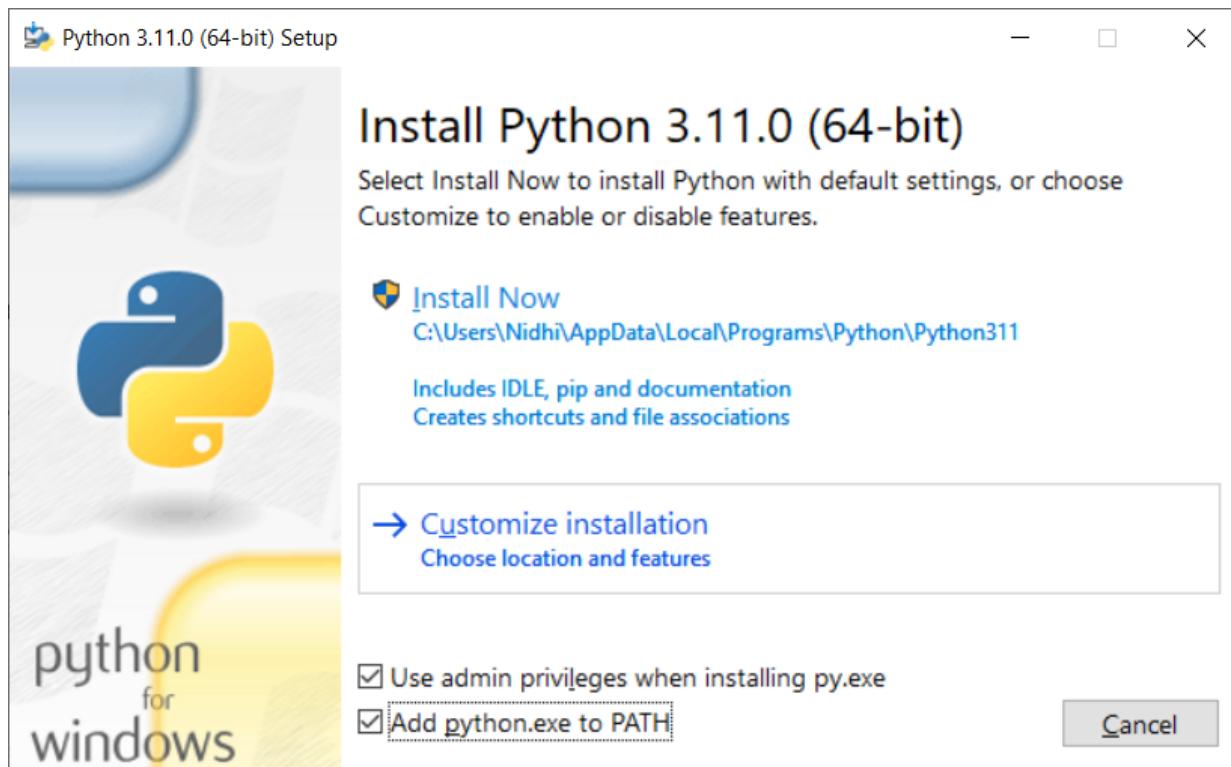


Figure 1.1: Installation starting process

Make sure to check both the checkboxes as highlighted in the preceding figure. This will enable the system to use Admin privileges during the installation process and automatically add the address of the **python.exe** file in the **PATH** variable of system settings. Next, click on **Install Now** to begin the set-up procedure. Refer to *Figure 1.2*:

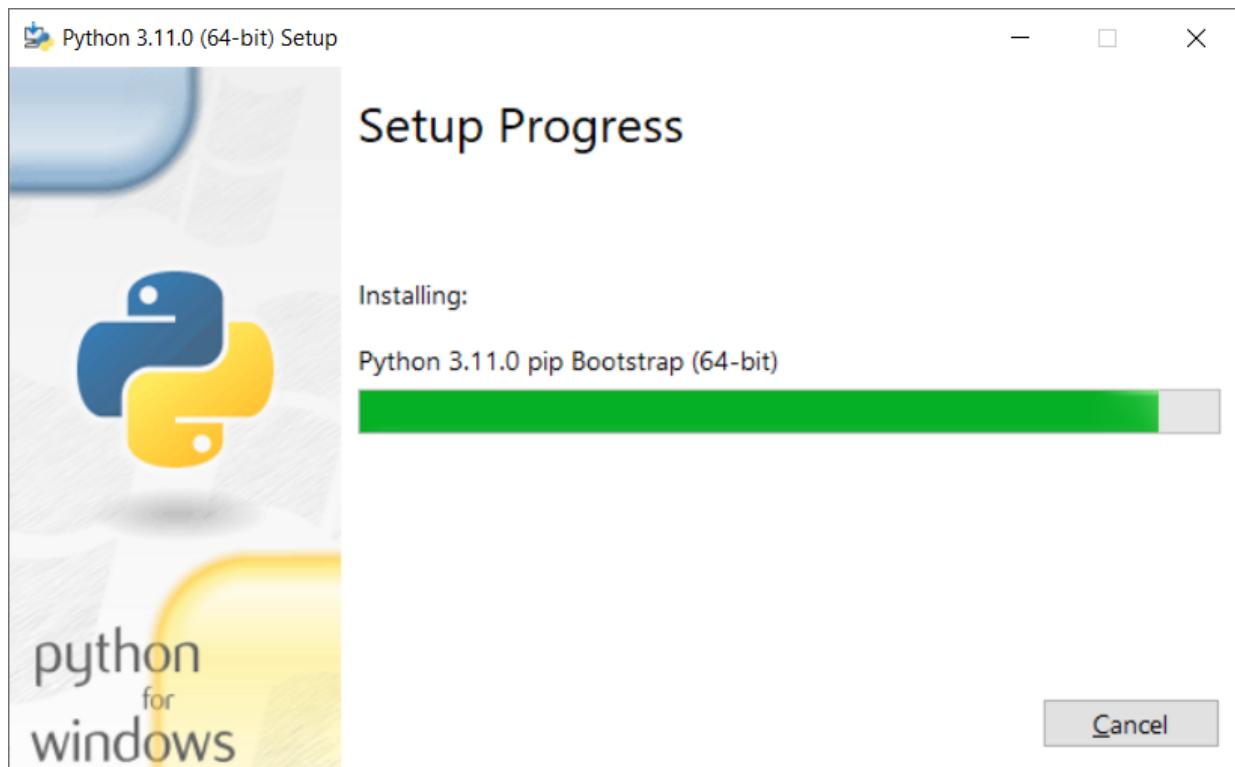


Figure 1.2: Installation in progress

Once the set-up is complete, we can see the successful set-up message, as shown in [*Figure 1.3*](#):

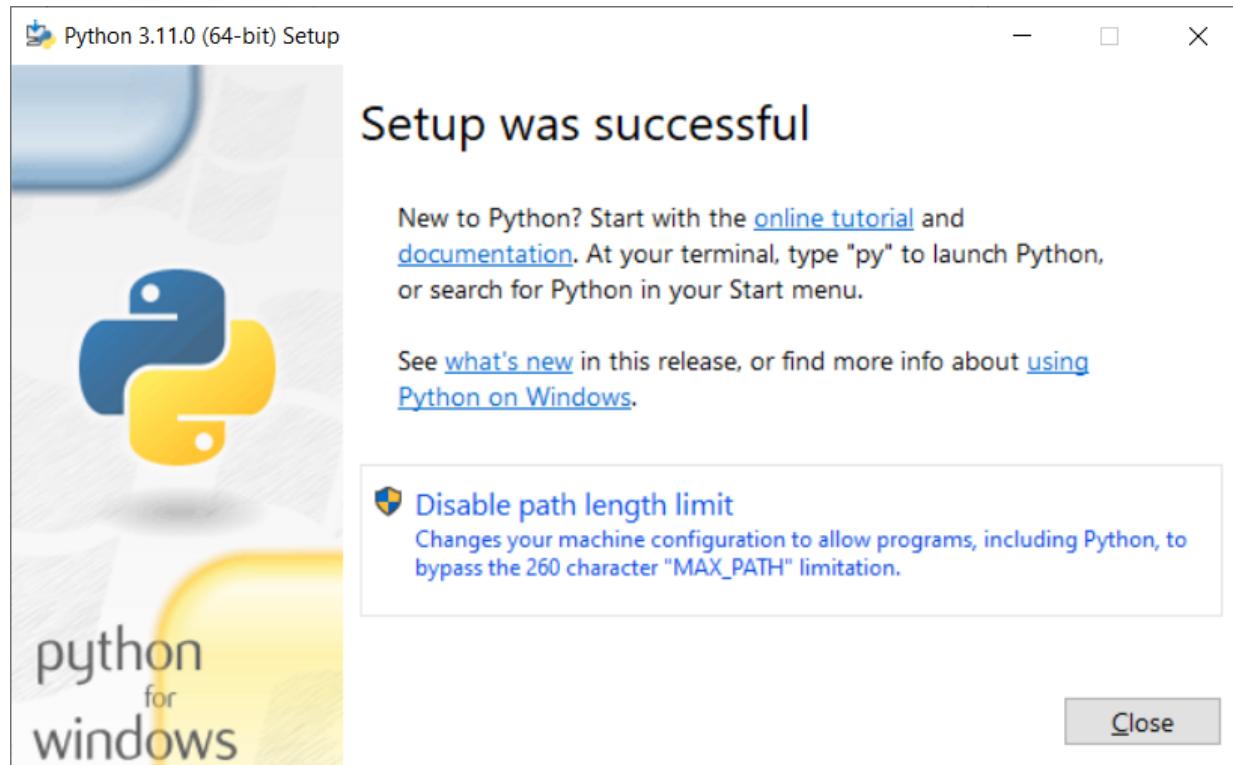
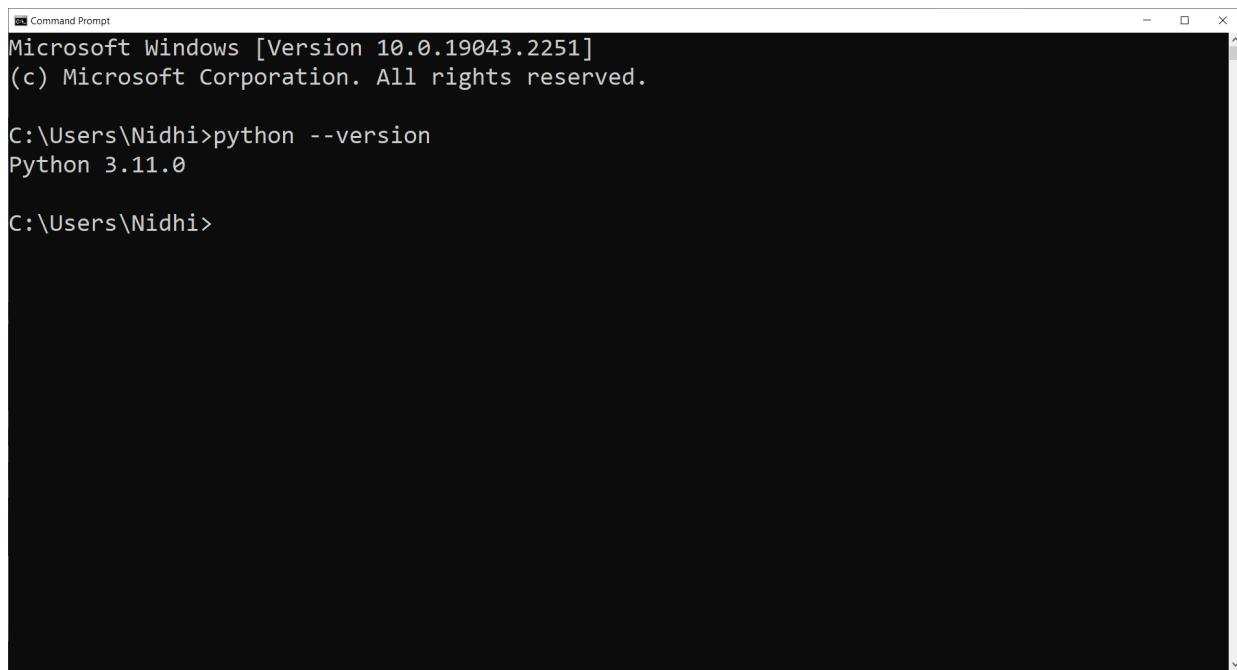


Figure 1.3: Set-up success screen

Click on the **Close** button to exit the set-up window. Now, we are ready to check Python installed on our system. After successful installation, open the command prompt and type the following command on the prompt:

python --version

The preceding command shows the installed version of Python, as shown in *Figure 1.4*:



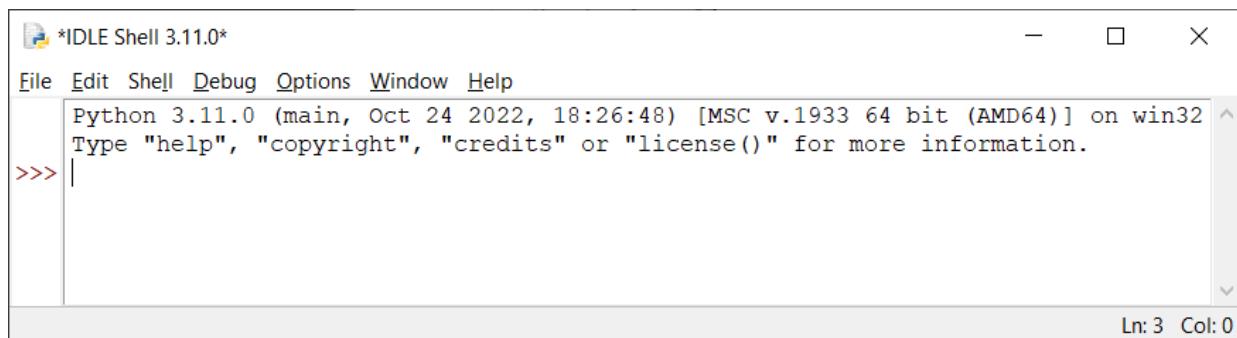
```
Command Prompt
Microsoft Windows [Version 10.0.19043.2251]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Nidhi>python --version
Python 3.11.0

C:\Users\Nidhi>
```

Figure 1.4: Check the installed Python version in the command prompt

Once we install Python 3 successfully on our system, we can use a very simple yet powerful built-in **Integrated Development Environment (IDE)** known as **IDLE**. To start IDLE, click on the **Start** menu, then select **Python 3.11 Folder**. Select **IDLE** (Python 3.11 64-bit). Refer to [Figure 1.5](#):



```
*IDLE Shell 3.11.0*
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> |
```

Figure 1.5: IDLE shell 3.11.0

Now, it is time to write our first command in Python IDLE Shell. Here in IDLE Shell, we can write and execute a single command at a time but not the entire Python program in one go. To solve this, we will create a new editing window in the IDLE shell. Go to **File** menu | **New File** or press

Ctrl + N. This will open a new untitled window where we can write entire Python code and execute it to see the output, as shown in [Figure 1.6](#):

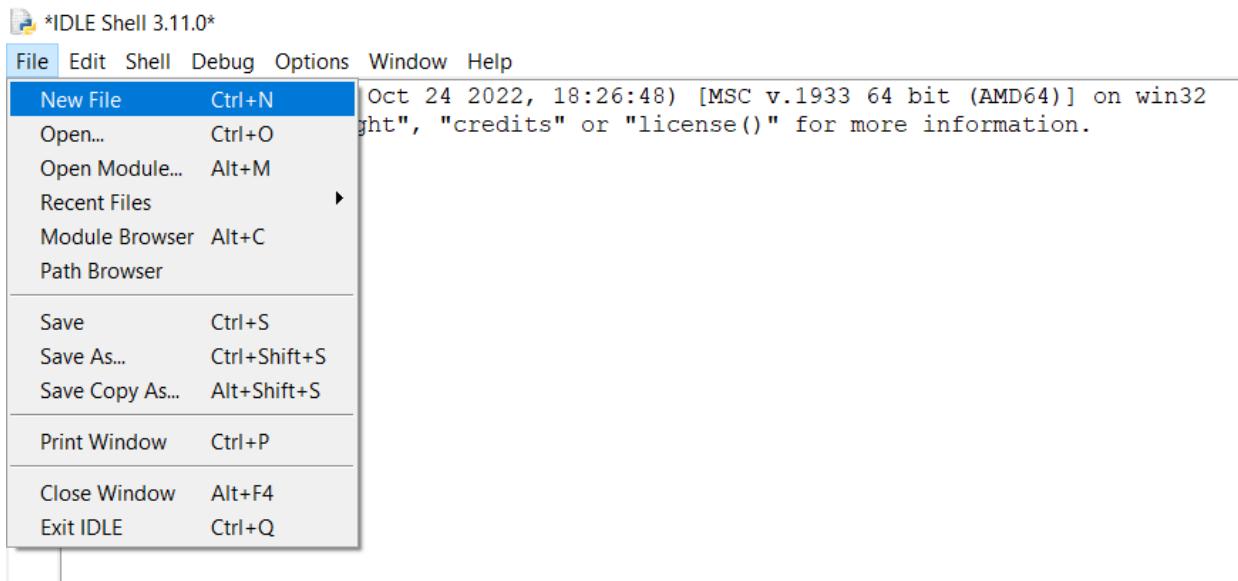
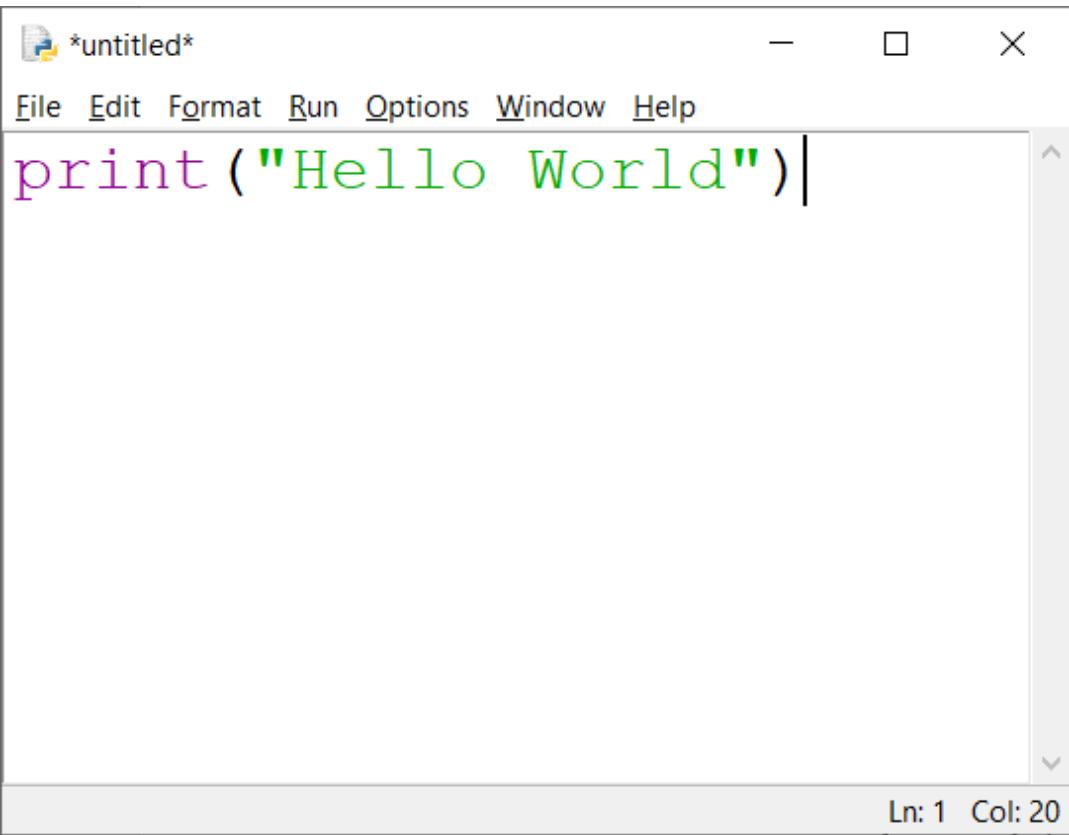


Figure 1.6: Opening new module in IDLE

Let us now write Python code to print the **Hello World** message. For this, we can use the print method in which the **Hello World** message is enclosed in double quotes or single quotes within parentheses brackets. Refer to [Figure 1.7](#):



The image shows a screenshot of a text editor window titled '*untitled*'. The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The main text area contains the Python code 'print ("Hello World")'. The status bar at the bottom right indicates 'Ln: 1 Col: 20'.

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

Figure 1.7: Print the “Hello World” command

Before running, we need to save the module first. For this, click on **File** menu | **Save As**, as shown in *Figure 1.8*:

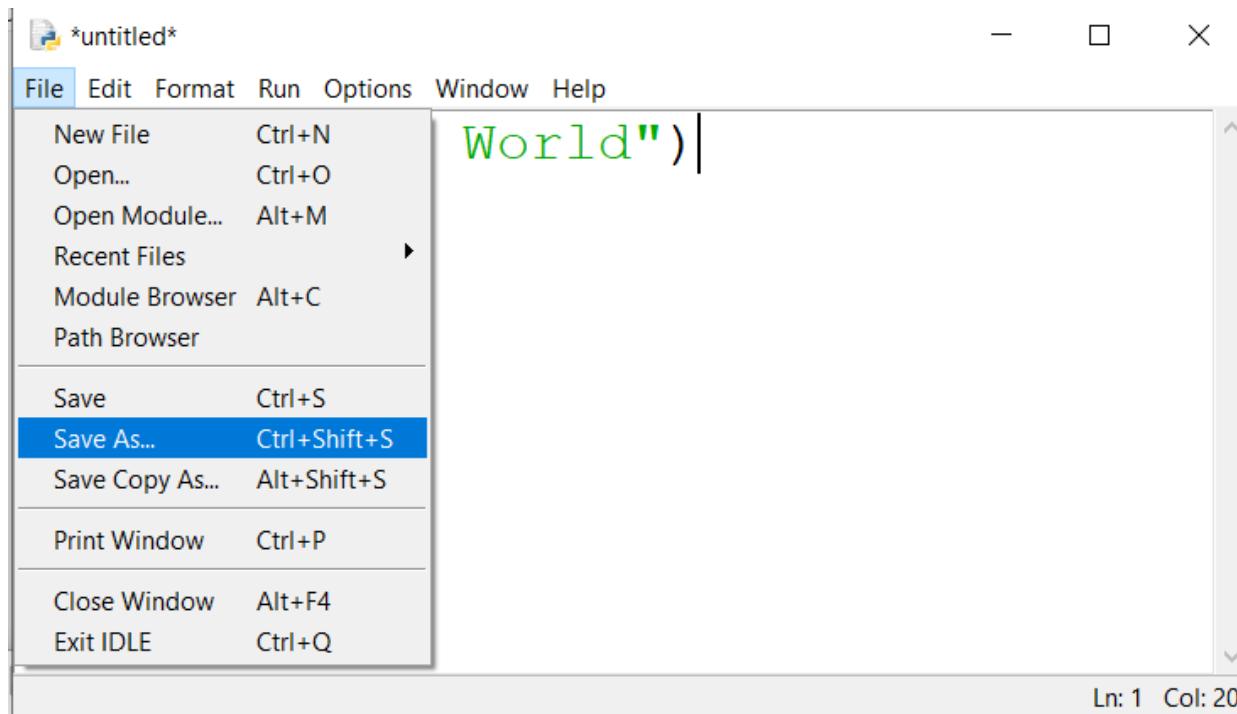


Figure 1.8: Saving module with name `First.py`

Give an appropriate name to the file, such as `First.py`, as shown in [Figure 1.9](#):

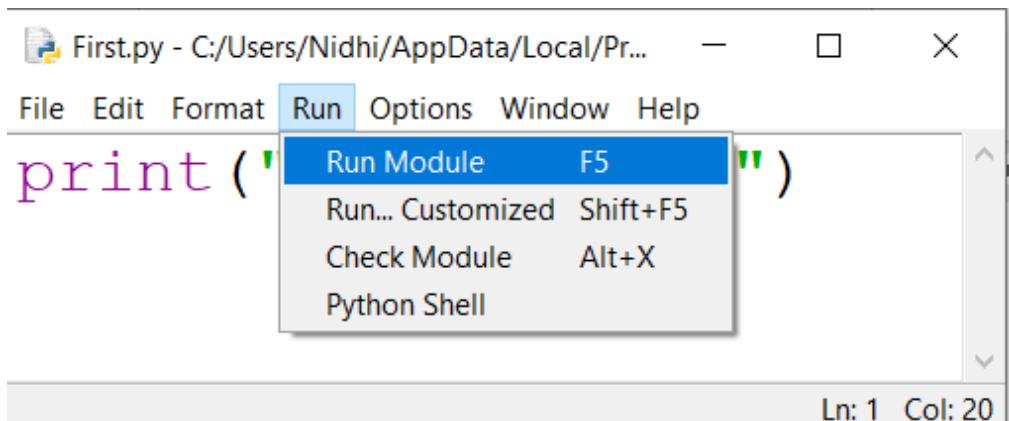
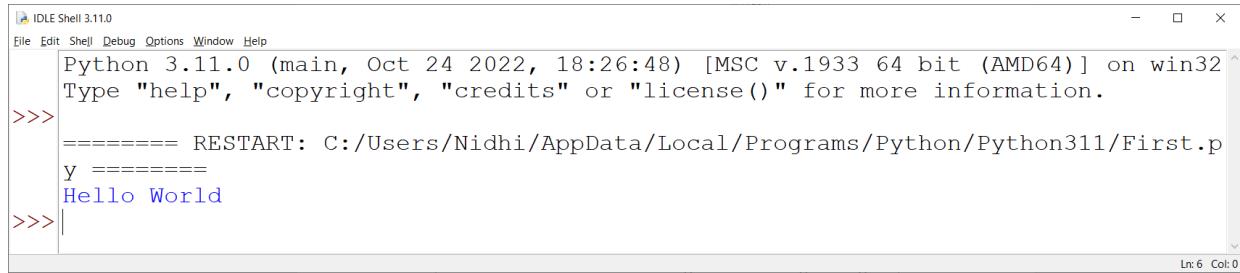


Figure 1.9: Run `First.py` module

Finally, to run the code in `First.py`, click on the **Run** menu | **Run Module** or press **F5** to run the current module. As we run the module, we can see the output in the IDLE shell window. The message **Hello World** is printed in the IDLE Shell prompt, as shown in [Figure 1.10](#):



```

IDLE Shell 3.11.0
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> ====== RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/First.py ======
Hello World
>>>

```

Figure 1.10: Output after executing *First.py*

Keywords

Python keywords are specially reserved words that convey a special meaning to the compiler/interpreter. Each keyword has a special meaning and a specific function. These keywords cannot be used as variables. Some examples of keywords are **True**, **False**, **class**, and so on. The entire list of keywords is given in [Table 1.1](#):

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

Table 1.1: List of keywords in Python

Identifier

A name in a Python program is called an identifier. It can be a class name, function name, module name, or variable name. There are certain rules that should be kept in mind while naming an identifier. These rules are listed as follows:

- The first character of the Identifier name must be an alphabet or underscore (`_`).

- All the characters in the Identifier name except the first character can be an alphabet in upper-case (A–Z), lower-case (a–z), an underscore, or a digit (0–9). For example, num, _num, and num123 are valid identifiers.
- Identifier name must not contain any white space, or special character (!, @, #, \$, %, ^, &, and *). For example, n um, n\$um, and 123num are invalid identifiers.
- No keyword can be used as an identifier name.
- Identifier names are case sensitive, that is, ContactNo, Contactno, contactno, and contact_No are all different.
- Examples of valid identifiers are ab789, _a, a_78, ab, and so on.
- Examples of invalid identifiers: 7a, a!7, a 7, @qw, and so on.

Comments

Comments are those statements that are not executed but rather completely ignored by the Python interpreter. Comments act as clues or indications that make code more understandable for fellow programmers. In Python, the following two types of comments are used:

- Single-Line Comment
- Multiline Comment

As the name suggests, a single-line comment begins with a # symbol and ends in the same line. On the other hand, a multiline comment extends to multiple lines that begin and end with triple quotes, either “” or “”.

Comments are demonstrated in the following code:

```
# Declaring a number num1
num1 = 10
'''Declaring address
of a person
with his location'''
```

```
address = "Delhi"
```

The statements in lines 1, 3, to 5 are comments that are non-executable statements.

Variable and data types

A variable is defined as a storage area to hold some assigned value. For example, `age = 10`. Here, `age` is a variable that is assigned a value 10. Once an object is assigned to a variable, you can refer to the object by that name. Suppose we create a variable:

```
a = 10
```

This assignment creates an integer object with the value 10 and assigns the variable `a` to point to that object. Now, consider the following statement:

```
b = a
```

Here, Python does not create another object, but rather, it creates a new reference, `b`, which points to the same integer object that points to. However, if we assign a new value to reference `b`, such that:

```
b = 20
```

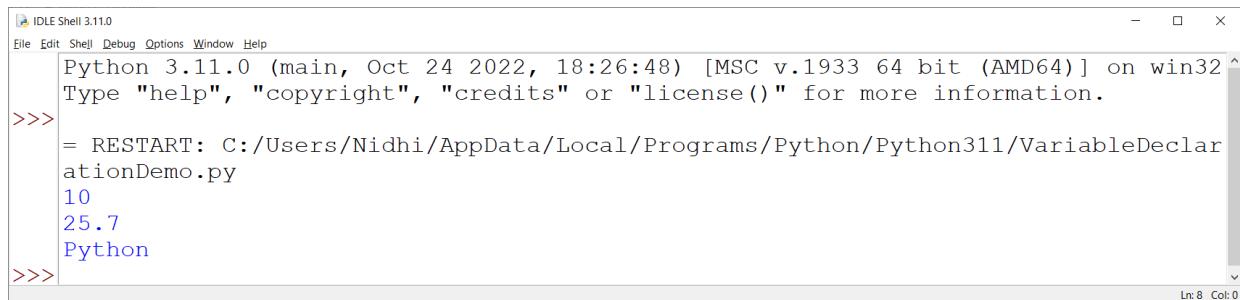
Here, Python creates a new integer object with the value 20, and now `b` becomes a reference to it. In the following example, we create variables with different types of values in IDLE IDE and view the output.

```
# Declaring variables x, y, z and assigning values
x, y, z = 10, 25.7, 'Python'
# Display values of variables using print method
print(x) # prints 10
print(y) # prints 25.7
```

```
print(z) # prints Python
```

Output:

The output can be seen in *Figure 1.11*:



The screenshot shows the Python 3.11.0 IDLE Shell window. The title bar reads "IDLE Shell 3.11.0". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main window displays the following text:

```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> = RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/VariableDeclarationDemo.py
10
25.7
Python
>>>
```

The status bar at the bottom right shows "Ln: 8 Col: 0".

Figure 1.11: Output

The naming mechanism works in-line with Python's object system, that is, everything in Python is an object. All the data types, such as numbers, strings, functions, and classes, are all objects. A name acts as a reference to get the objects. There are three most commonly used methods of constructing a multi-word variable name:

- **Camel case:** Second and subsequent words are capitalized to make word boundaries easier to see. Example: `salaryOfEmployee`.
- **Pascal case:** Identical to Camel Case, except the first word is also capitalized. Example: `SalaryOfEmployee`.
- **Snake case:** Words are separated by underscores. Example: `salary_of_employee`.

As per the recommendation of **Python Enhancement Proposal (PEP) 8**, Style Guide for Python Code, Snake Case should be used for functions and variable names, and Pascal Case should be used for class names.

Datatypes in Python

In a programming language, Data Types are used to define the category of a given variable based on the value assigned to it. We do not need to explicitly define the type of a variable, and the value assigned to the variable will

decide the data type of the variable. *Figure 1.12* describes the different types of data types in Python:

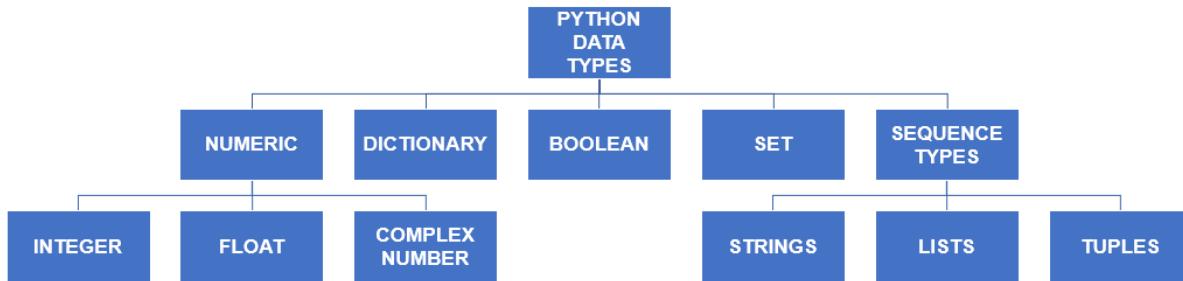


Figure 1.12: Flowchart showing data types in Python

Let us briefly discuss each of these data types with examples.

Numbers

Number data types store numeric values. They are immutable data types, that is, changing the value of a number data type results in a newly allocated object. *Table 1.2* summarizes the types of Numbers in Python:

Integer Numbers	Floating point Numbers	Complex Numbers	Binary, Octal, and Hexadecimal Numbers
Integer Numbers that can be both positive and negative numbers with no decimal part in them. Example: 123, -56,	The decimal numbers with both integer and fractional parts in it. Example: -14.45 and 0.67 are examples	The complex numbers of the form $a+bj$ such that "a" forms the real part and "b" forms the imaginary part of the given complex number. Example: 2+1.6j	The decimal system is base 10, where only 10 symbols (0–9) can be used to represent a number. Similarly, a Binary number is defined with the prefix 0b or 0B denotes base 2, an Octal number with the prefix 0o or 0O denotes base 8, and a hexadecimal number with the prefix 0x or 0X denotes base 16. Example: Represent 50 in other formats. Here, 50 = (in Binary) 0b110010

707, and –222 are examples of Integers.	of Floating-point numbers.	Here 2 is the real number, and 1.6 is the imaginary number	50 = (in Octal) 0062 50 = (in Hexadecimal) 0x32
---	----------------------------	--	--

Table 1.2: Types of Numbers in Python

Python supports three different numerical types – int (signed integers), floating point real values, and complex numbers. For example:

```
# Python program to demonstrate Numeric Data Type

x = 5

print("Type of x: ", type(x))

y = 5.0

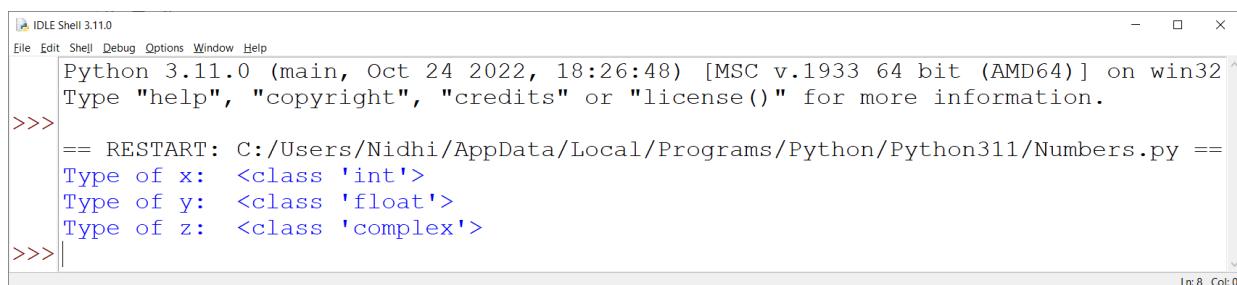
print("Type of y: ", type(y))

z = 2 + 4j

print("Type of z: ", type(z))
```

Output:

The output can be seen in *Figure 1.13*:



```
IDLE Shell 3.11.0
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> == RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/Numbers.py ==
Type of x: <class 'int'>
Type of y: <class 'float'>
Type of z: <class 'complex'>
>>> Ln: 8 Col: 0
```

Figure 1.13: Output

The **type()** function in the preceding example is used to get the type of an object, which is nothing but the class to which the object belongs. Now, let us see a demonstration in Python to convert a decimal number to other number formats:

```

# Program to convert a Decimal number into other
# number formats

decimal_num = 180

print("The Decimal Number is : ", decimal_num)

print("Converted to Binary : ", bin(decimal_num))

print("Converted to Octal : ", oct(decimal_num))

print("Converted to Hexadecimal : ",
hex(decimal_num))

```

Output:

The output can be seen in *Figure 1.14*:

```

IDLE Shell 3.11.0
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> = RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/ConvertDecimalToOctalHexBinary.py
The Decimal Number is : 180
Converted to Binary : 0b10110100
Converted to Octal : 0o264
Converted to Hexadecimal : 0xb4
>>>

```

Figure 1.14: Output

In the preceding program, we have used built-in functions **bin()**, **oct()**, and **hex()** to convert the given decimal number 180 into Binary, Octal, and Hexadecimal number systems.

Dictionary

Python's dictionaries consist of key-value pairs. A dictionary key can be almost any Python type but is usually numbers or strings. Values can be any arbitrary Python object. Dictionaries are enclosed by curly braces ({}), and their values can be assigned and accessed using its key in square braces ([]).

Dictionaries have no ordering among their elements. For example:

```
my_dictionary = {'name': 'sumit', 'code':149, 'dept': 'IT'}
```

For example:

```
# To create a dictionary
```

```
new_dict1 = {1: "January", 2: "February", 3: "March", 4: "April", 5: "May"}
```

```
# To display dictionary
```

```
print("Dictionary new_dict1 is: ", new_dict1)
```

```
# To create a dictionary using a dict class
```

```
new_dict2 = dict({6: "June", 7: "July", 8: "August", 9: "September"})
```

```
# To display new dictionary
```

```
print("Dictionary new_dict2 is: ", new_dict2)
```

```
# To view the type of Dictionaries
```

```
print("Type of new_dict1: ", type(new_dict1))
```

```
print("Type of new_dict2: ", type(new_dict2))
```

```
# To access values 'January' and 'February' using their keys
```

```
print("Value of key 1: ", new_dict1[1])
```

```
print("Values in key 2: ", new_dict1[2])
```

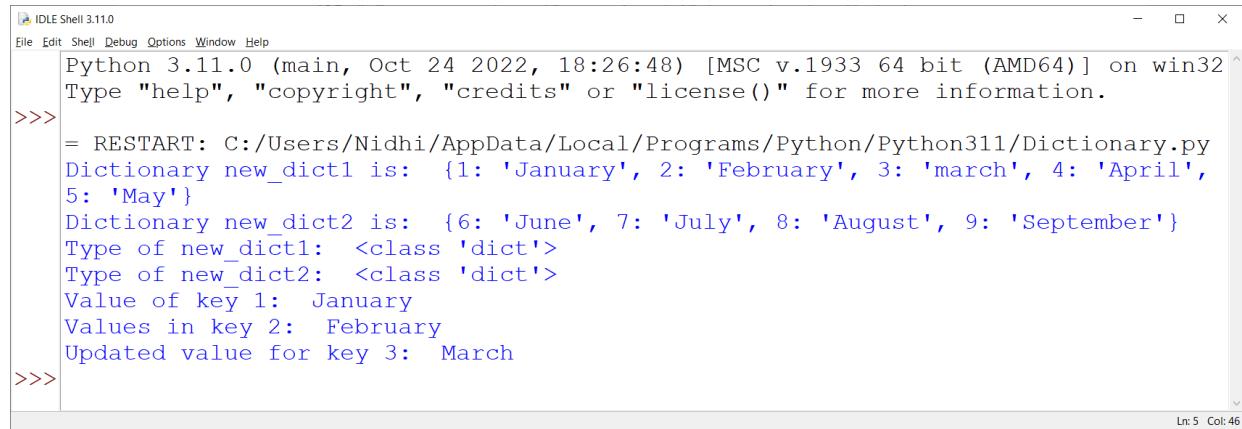
```
# To change the value of a key and display updated value
```

```
new_dict1[3] = "March"
```

```
print("Updated value for key 3: ", new_dict1[3])
```

Output:

The output can be seen in *Figure 1.15*:



```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> = RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/Dictionary.py
Dictionary new_dict1 is:  {1: 'January', 2: 'February', 3: 'march', 4: 'April', 5: 'May'}
Dictionary new_dict2 is:  {6: 'June', 7: 'July', 8: 'August', 9: 'September'}
Type of new_dict1:  <class 'dict'>
Type of new_dict2:  <class 'dict'>
Value of key 1:  January
Values in key 2:  February
Updated value for key 3:  March
>>>
```

Figure 1.15: Output

Boolean

Booleans represent one of two values: **True** or **False**. In programming, you often need to know if an expression is **True** or **False**. You can evaluate any expression in Python and get one of two answers: **True** or **False**. When you compare two values, the expression is evaluated, and Python returns the Boolean answer. The **bool()** function allows you to evaluate any value and give you **True** or **False** in return.

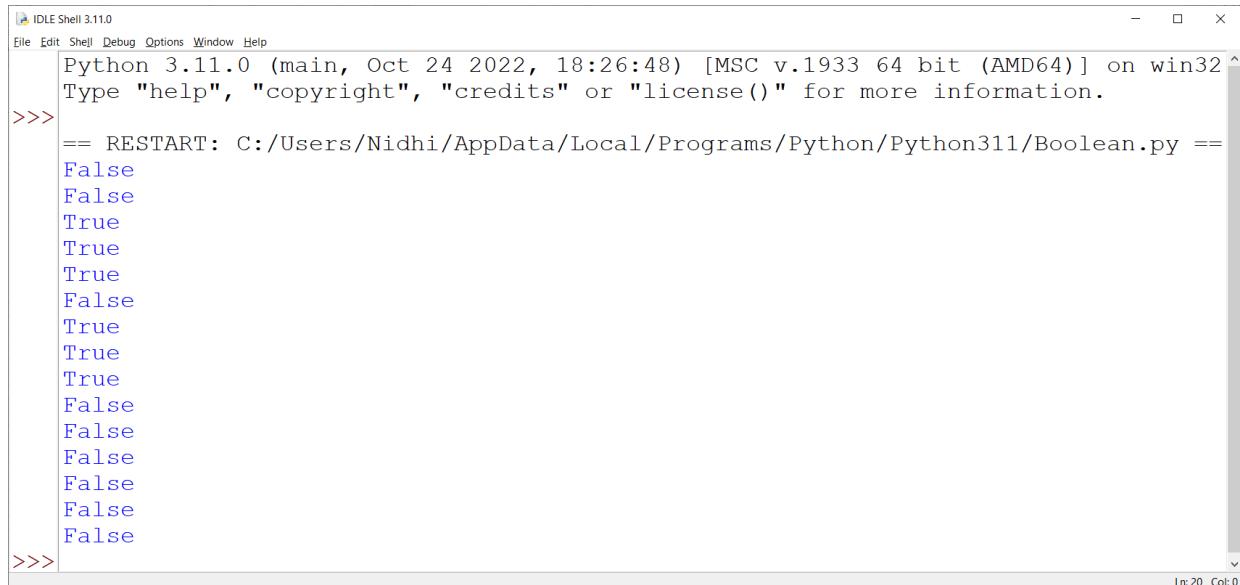
```
# Check for Boolean True/ False in Expressions
print(100 > 200)
print(100 == 200)
print(100 < 200)

# Check for Boolean True/ False in different values
# of variables
a = "Python"
b = 50
print(bool(a))
```

```
print(bool(b))  
print(bool(0))  
print(bool(-10))  
print(bool(["January", "February", "March",  
"April"]))  
print(bool(True))  
print(bool(False))  
print(bool(None))  
print(bool(""))  
print(bool(()))  
print(bool([]))  
print(bool({}))
```

Output:

The output can be seen in *Figure 1.16*:



The screenshot shows the IDLE Shell 3.11.0 interface. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The title bar says "IDLE Shell 3.11.0". The main window displays the following text:

```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>> == RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/Boolean.py ==  
False  
False  
True  
True  
True  
False  
True  
True  
False  
False  
False  
False  
False  
>>>
```

The status bar at the bottom right shows "Ln: 20 Col: 0".

Figure 1.16: Output

From the output, we can note that a value is evaluated as **True** if it has some valid value in it. Any string, except empty strings, is **True**, and any number except 0 is **True**. Any list, tuple, set, and dictionary are **True**, except the empty ones. Thus, the values such as empty (), [], {}, "", the number 0, and the value **None**, value **False** all evaluate to **False**.

Set

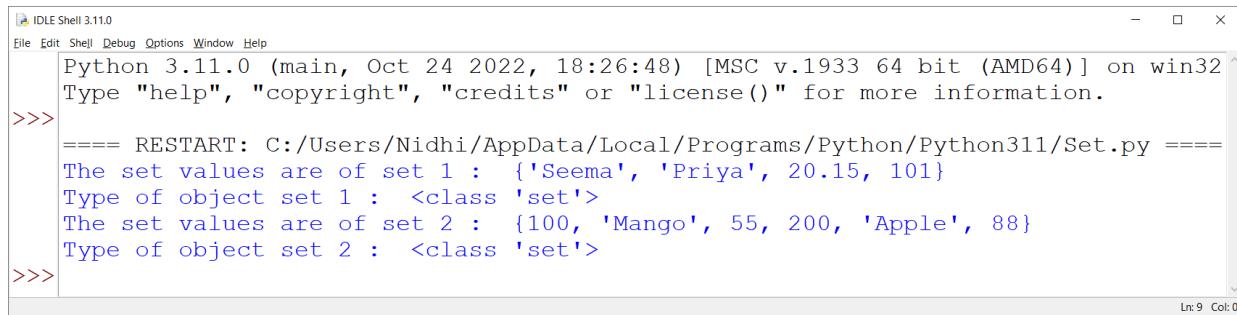
Python set is the collection of the unordered dataset. It is enclosed by the {}, and each element is separated by the comma (,). Duplicate elements are not allowed. Sets are mutable, which means that we can modify it after its creation. There is no index for the elements of the set, that is, we cannot directly access any element of the set by the index. However, we can print them all together, or we can get the list of elements by looping through the set. Example: `set1 = {'Dog', 'Cat', 'Donkey', 'Cow'}`. We can create a Set by either enclosing values in the curly brackets {} or by using a `set()` class.

For example:

```
# create a set using curly brackets{},  
new_set1 = {101, 20.15, "Seema", "Priya"}  
print("The set values are of set 1 : ", new_set1)  
print("Type of object set 1 : ", type(new_set1))  
  
# To create a set using set class  
new_set2 = set({55, 100, 88, 200, "Apple",  
"Mango"})  
print("The set values are of set 2 : ", new_set2)  
print("Type of object set 2 : ", type(new_set2))
```

Output:

The output can be seen in [*Figure 1.17*](#):



```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> ===== RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/Set.py =====
The set values are of set 1 :  {'Seema', 'Priya', 20.15, 101}
Type of object set 1 :  <class 'set'>
The set values are of set 2 :  {100, 'Mango', 55, 200, 'Apple', 88}
Type of object set 2 :  <class 'set'>

>>>
```

Figure 1.17: Output

Sequence types

The most basic data structure in Python is the sequence. A sequence is a group of items with a deterministic ordering. The order in which we put them in is the order in which we get an item out from them. Each element of a sequence is assigned a number—its position or index. The first index is zero, and the second index is one, and so forth. There are three basic sequence types in Python, namely, Strings, Lists, and Tuples.

Let us briefly discuss about each of these sequences.

Strings

Strings in Python are identified as an immutable contiguous set of characters represented in quotation marks. A single-line string is represented by enclosing the set of characters in either single-quotes or double-quotes, whereas triple-quotes represent a multiline string. Characters in a string can be accessed using a slice operator [] from both directions. Just like most programming languages, the indexing of Python strings begins from 0 while traversing in a forward direction from the left to the right end. However, while moving backward from the right to the left end, the index begins from -1. Refer to [Figure 1.18](#).

Suppose a string is: **str1="welcome"**

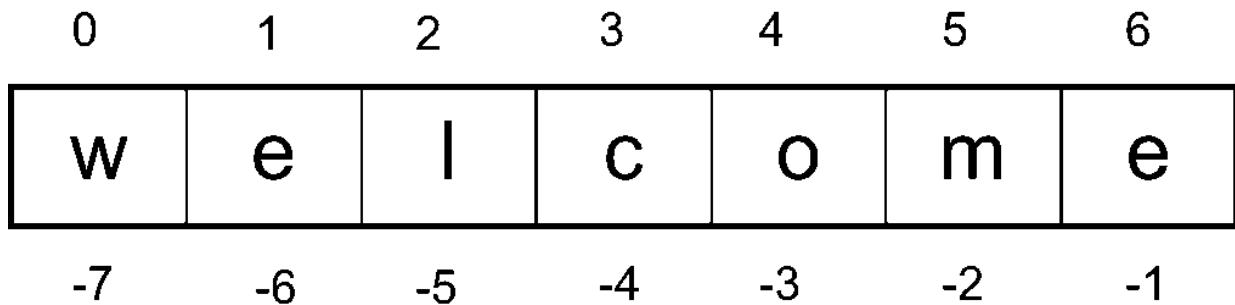


Figure 1.18: Positive and negative indexing in string

Where:

```
str1[0]='w'      str1[1]='e'      str1[2]='l'
str1[3]='c'

str1[-4]='c'     str1[-3]='o'     str1[-2]='m'
str1[-1]='e'
```

For example, refer to the following code:

```
text1 = "Welcome to learn Python"

print(type(text1))

# To display the entire string

print(text1)

# To access first character of a string 'w'

print(text1[0])

# To access third character of a string 'l'

print(text1[2])

# To access last character of a string 'n'

print(text1[-1])

# To access second last character of a string 'o'
```

```
print(text1[-2])
```

Output:

The output can be seen in *Figure 1.19*:



The screenshot shows the Python 3.11.0 IDLE Shell interface. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following text:

```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> = RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/StringBasic.py
<class 'str'>
Welcome to learn Python
W
l
n
o
>>>
```

The text "Welcome to learn Python" is printed to the shell, followed by the characters "W", "l", "n", and "o" on separate lines. The status bar at the bottom right shows "Ln: 6 Col: 23".

Figure 1.19: Output

Escape Sequences in Strings

Sometimes, we want Python to interpret a character or sequence of characters within a string differently. This may occur in one of the following two ways:

- We may want to suppress the special interpretation that certain characters are usually given within a string.
- We may want to apply special interpretation to characters in a string that would normally be taken literally.

We can accomplish this using a backslash (\) character. A backslash character in a string indicates that one or more characters that follow it should be treated specially. This is referred to as an escape sequence because the backslash causes the subsequent character sequence to *escape* its usual meaning.

Table 1.3 features escape sequences that cause Python to suppress the usual special interpretation of a character in a string:

Escape Sequence	Escaped Interpretation
\'	Literal single quote (') character

Escape Sequence	Escaped Interpretation
\"	Literal double quote (") character
\\\	Literal backslash () character
\a	ASCII Bell (BEL) character
\b	ASCII Backspace (BS) character
\f	ASCII Formfeed (FF) character
\n	ASCII Linefeed (LF) character
\r	ASCII Carriage Return (CR) character
\t	ASCII Horizontal Tab (TAB) character
\uxxxx	Unicode character with 16-bit hex value xxxx
\Uxxxxxxxxx	Unicode character with 32-bit hex valuexxxxxxxx
\v	ASCII Vertical Tab (VT) character
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

Table 1.3: Escape sequences in Python strings

Lists

The Python List is an ordered sequence of elements that may be of the same or different datatypes. It contains items separated by commas and enclosed within square brackets ([]). To store all student's names, we can use list type, for example, `student_list = ['Sumit', 'Aman' , 'Priya', 'Rohit', 'Raman', 'Namita', 'Aman']`. We can create a list using the following two ways:

- By enclosing elements in the square brackets [].
- Using a `list()` class.

For example:

```
student_list = [ 'Sumit', 'Aman' , 'Priya', 'Rohit', 'Raman', 'Namita', 'Aman']
```

```

# To display list elements and type of the list
object

print("Original Student List : ", student_list)
print("Type of object : ", type(student_list))

# Accessing first element of list

print("First element in List : ", student_list[0])

# To modify 2nd element of the list from "Aman" to
"Aman Sharma"

student_list[1] = "Aman Sharma"

# Display updated list elements

print("Updated List : ", student_list)

# To create new list of students' marks using a
list() class method

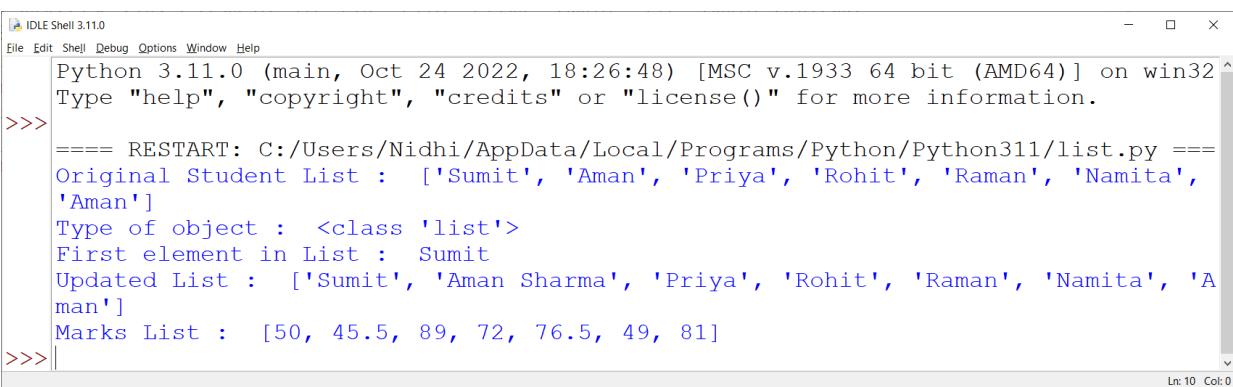
new_list = list([50, 45.5, 89, 72, 76.5, 49, 81])

print("Marks List : ", new_list)

```

Output

The output can be seen in *Figure 1.20*:



```

IDLE Shell 3.11.0
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> ===== RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/list.py ====
Original Student List :  ['Sumit', 'Aman', 'Priya', 'Rohit', 'Raman', 'Namita',
'Aman']
Type of object :  <class 'list'>
First element in List :  Sumit
Updated List :  ['Sumit', 'Aman Sharma', 'Priya', 'Rohit', 'Raman', 'Namita', 'A
man']
Marks List :  [50, 45.5, 89, 72, 76.5, 49, 81]
>>> Ln: 10 Col: 0

```

Figure 1.20: Output

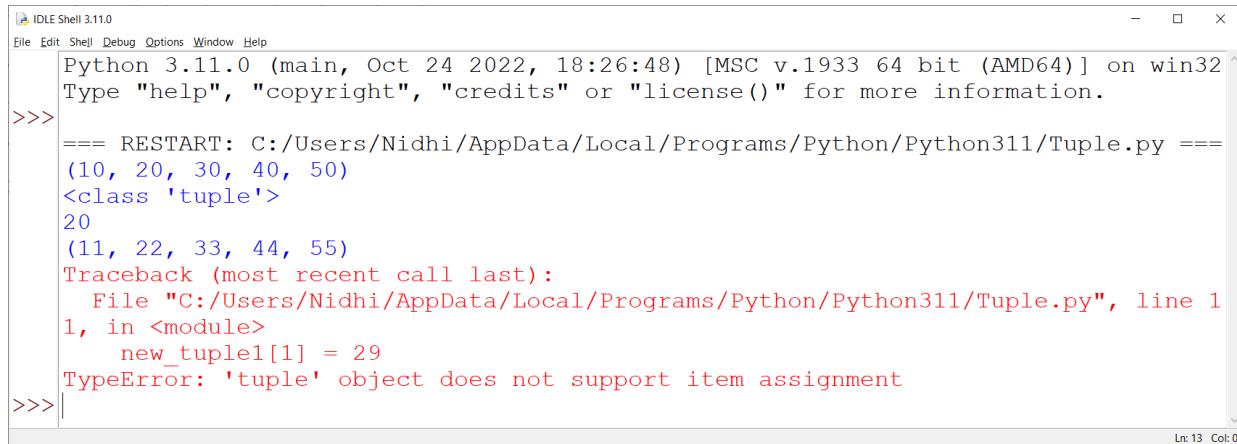
Tuples

A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses. List is enclosed in the brackets ([]), and their elements with size can be modified, whereas tuples are enclosed in parentheses (()) and cannot be updated. Tuples can be thought of as read-only lists. For example:

```
# To create a tuple
new_tuple1 = (10, 20, 30, 40, 50)
print(new_tuple1)
print(type(new_tuple1))
# Accessing 2nd element of a tuple
print(new_tuple1[1])
# To create a tuple using a tuple() class
new_tuple2 = tuple((11, 22, 33, 44, 55))
print(new_tuple2)
# To modify 2nd element of tuple
new_tuple1[1] = 29
# TypeError: 'tuple' object does not support item assignment
```

Output:

The output can be seen in *Figure 1.21*:



```
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> === RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/Tuple.py ===
(10, 20, 30, 40, 50)
<class 'tuple'>
20
(11, 22, 33, 44, 55)
Traceback (most recent call last):
  File "C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/Tuple.py", line 1
1, in <module>
    new_tuple1[1] = 29
TypeError: 'tuple' object does not support item assignment
>>> Ln: 13 Col: 0
```

Figure 1.21: Output

Type conversion in Python

The process of converting the value of one data type (integer, string, float, and so on) to another data type is called **Type Conversion**. Python has the following two types of type conversion:

- Implicit type conversion
- Explicit type conversion

Implicit type conversion

In Implicit Type Conversion, Python automatically converts one data type to another data type. Here, Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

For example: Converting **integer** to **float**:

```
# Declare and assign 3 integer numbers
num1, num2, num3 = 45, 30, 65
# Find sum of 3 numbers and display the sum with
# its type
sum = num1 + num2 + num3
print("Sum of 3 numbers : : ", sum)
```

```

print("Datatype of sum is : ", type(sum))

# Find Mean of 3 numbers and display the mean with
# its type

mean = sum / 3

print("Mean of 3 numbers : ", mean)

print("Datatype of Mean : ", type(mean))

```

Output:

The output can be seen in *Figure 1.22*:

```

IDLE Shell 3.11.0
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>>
= RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/ImplicitConversion.py
Sum of 3 numbers : : 140
Datatype of sum is :  <class 'int'>
Mean of 3 numbers :  46.666666666666664
Datatype of Mean :  <class 'float'>
>>>

```

Figure 1.22: Output

Here, in the output, we can see that it has a float data type because Python always converts smaller data types to larger data types to avoid the loss of data.

Explicit type conversion

Explicit type conversion means that the programmer manually changes the data type of one object to another with the help of an in-built Python function. Supposing we try to add a string (higher) data type to an integer (lower) data type. We will not be able to do this implicitly and may encounter a **TypeError** exception in this case. Thus, we need to explicitly typecast and change the higher datatype into the lower datatype manually in this scenario.

Syntax for explicit type conversion: **<desired_datatype> (Expression)**

We can use the following predefined functions in <desired_datatype> to perform explicit type conversion. Refer to [Table 1.4](#):

Function	Purpose
<code>int()</code>	The <code>int()</code> function helps to convert a floating-point number or string into an integer data type.
<code>float()</code>	Using <code>float()</code> function, we can convert an integer or string into a floating-point number.
<code>str()</code>	The <code>str()</code> function is used to convert any data type into the string data type.

Table 1.4: Typecasting functions in Python

For example:

```

num_int = 101
num_float = 22.0
num_str = "123"

print("Orginal Datatype of num_int : ",
type(num_int))

print("Orginal Datatype of num_float : ",
type(num_float))

print("Orginal Datatype of num_str : ",
type(num_str))

print(".....Explicit
Typecasting.....")

print("Datatype of num_int after Typecasting into
String : ",type(str(num_int)))

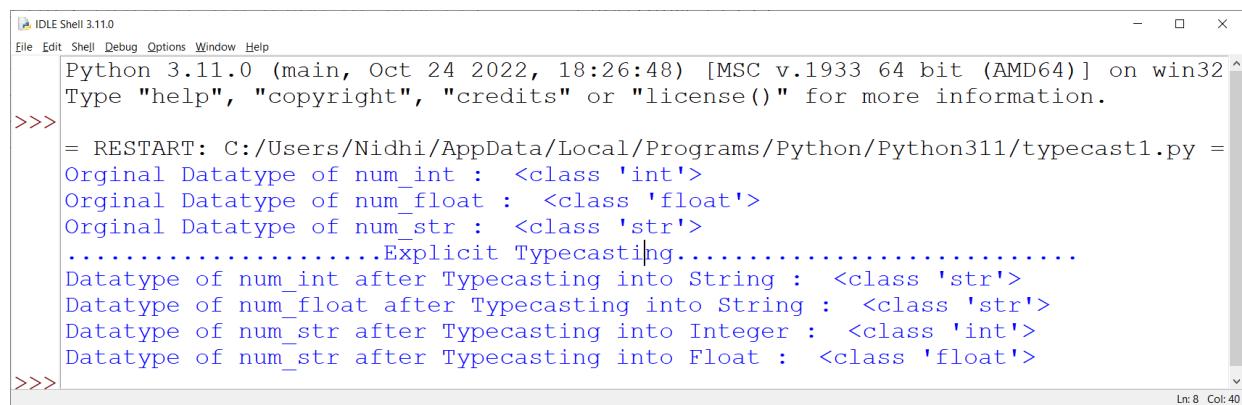
print("Datatype of num_float after Typecasting into
String : ",type(str(num_float )))

```

```
print("Datatype of num_str after Typecasting into Integer : ", type(int(num_str)))  
print("Datatype of num_str after Typecasting into Float : ", type(float(num_str)))
```

Output:

The output can be seen in *Figure 1.23*:



The screenshot shows the Python IDLE Shell 3.11.0 interface. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following text:

```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>> = RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/typecast1.py =  
Orginal Datatype of num_int : <class 'int'>  
Orginal Datatype of num_float : <class 'float'>  
Orginal Datatype of num_str : <class 'str'>  
.....Explicit Typecasting.....  
Datatype of num_int after Typecasting into String : <class 'str'>  
Datatype of num_float after Typecasting into String : <class 'str'>  
Datatype of num_str after Typecasting into Integer : <class 'int'>  
Datatype of num_str after Typecasting into Float : <class 'float'>  
>>>
```

Figure 1.23: Output

Input/output using Python

The main aim of any programming language is to make interaction with the users by fetching input from the user through the **Keyboard** input device and providing processed solutions as output to the user via the **Monitor** screen. The basic workflow of input/output process is shown in *Figure 1.24*:

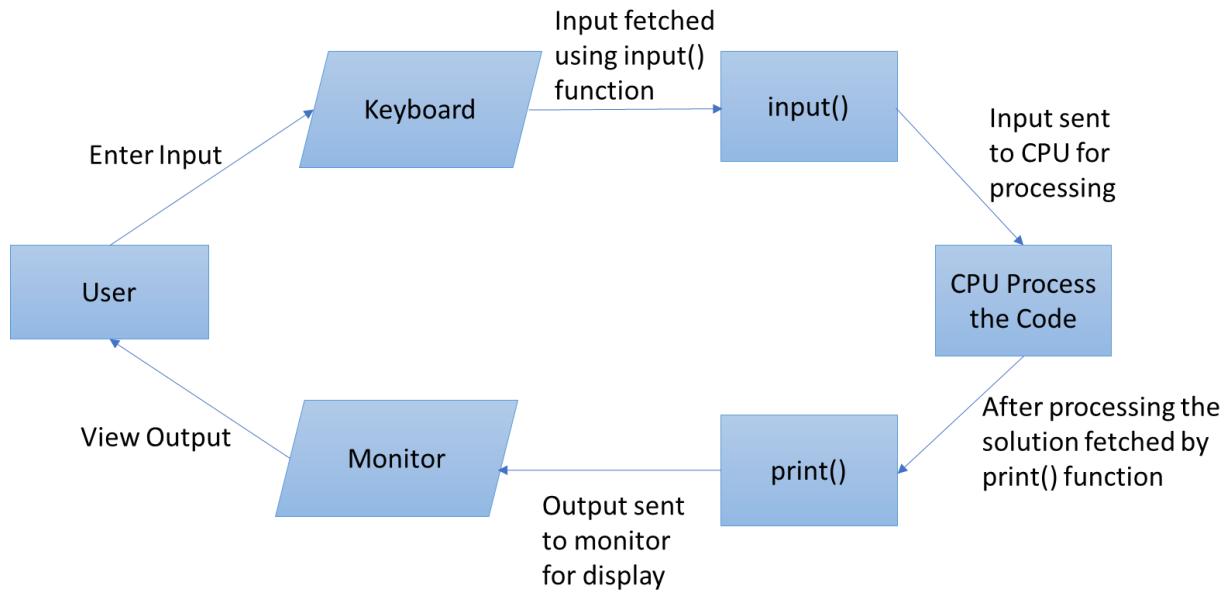


Figure 1.24: Input/output process in Python

The functions like `input()` and `print()` are widely used for standard input and output operations, respectively. The syntax for `input()` is as follows:

```
input([prompt])
```

Here, the prompt is the string we wish to display on the screen.

It is important to note that the `input()` function takes only string-type input. If we wish to input an Integer, Float, or any other type, then we need to explicitly typecast the default String input to the desired datatype.

Similarly, we use the `print()` function to output data to the standard output device (screen).

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

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

- Here, `*objects` is the value(s) to be printed. (* means any number of values).
- The `sep` refers to the separator, and it is used between the values. It defaults into a space character.

- After all values are printed, the **end** is printed. Here, '\n' refers to a new line.
- The file is the object where the values are printed, and its default value is **sys.stdout** (screen).
- Various forms of **print()** function:

```
print('Hello User')      # print simple string
"Hello User"
```

Output: Hello User

```
print(1,2,3,4)      # print multiple comma
separated outputs
```

Output: 1 2 3 4

```
print(1,2,3,4,sep='#',end='&')      # values
separated by '#' and end '&'
```

Output: #1#2#3#4&

For example:

```
# Here we will take an integer and String as input
from the user.
```

```
print("*****Hello
User*****")
```

```
name = input("Enter your name : ")
```

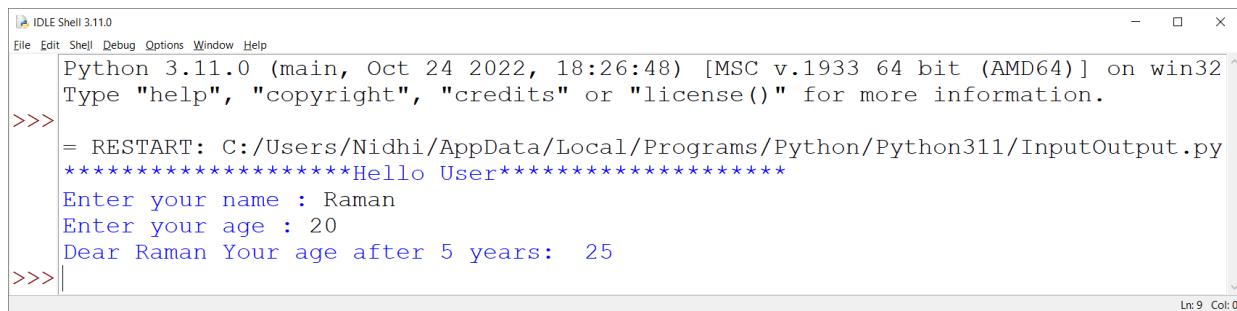
```
age = int(input("Enter your age : "))
```

```
new_age = age + 5
```

```
print("Dear " + name + " Your age after 5 years: ",
new_age)
```

Output:

The output can be seen in *Figure 1.25*:



```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> = RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/InputOutput.py
*****Hello User*****
Enter your name : Raman
Enter your age : 20
Dear Raman Your age after 5 years:  25
>>> |
```

Figure 1.25: Output

In the preceding code, in line 6, we can see the “+” operator being used with the string **“Dear”** and a variable name which is of String datatype itself. In Python, the “+” operator can be used to join or concatenate two strings automatically, although when used with two numbers, the “+” operator returns their sum. Here, the “+” operator shows different functionality based on the type of data it is used with. This concept is known as Operator Overloading in Python, where an operator displays different behavior with different sets of data operands.

Output formatting using **format()**

Sometimes, we would like to format our output to make it look attractive. This can be done by using the **str.format()** method. The **str.format()** is a string formatting method in Python3 that enables to concatenate elements within a string through positional formatting along with multiple substitutions and value formatting. The string that is passed as parameters into the format function is replaced and concatenated into the placeholders defined by **{}**.

Syntax: **{ } .format(value)**

- **{}** refers to the string to format, and the value parameter is any integer, floating point constant, string, character, or variable.
- It returns a formatted string with a parameter value placed in the placeholder **{}**. The placeholder may have indexes 0, 1, 2. as parameters to define the order of replacing values defined in **format()**.

For example:

```
# Demo of format()

print ("{}, Hello ! Welcome to python lecture."
       .format("Good Evening..."))

# Demo of format() with a String variable

str1 = "This lecture is about {}"

print (str1.format("Python Programming"))

# Demo of format() with Integer variables as
parameters

num1 = 10

num2 = 20

Num3 = 30

print('The value of num1 is {} and num2 is
      {}'.format(num1,num2))

# Demo of format with indexed parameters where {0}
represents first parameter

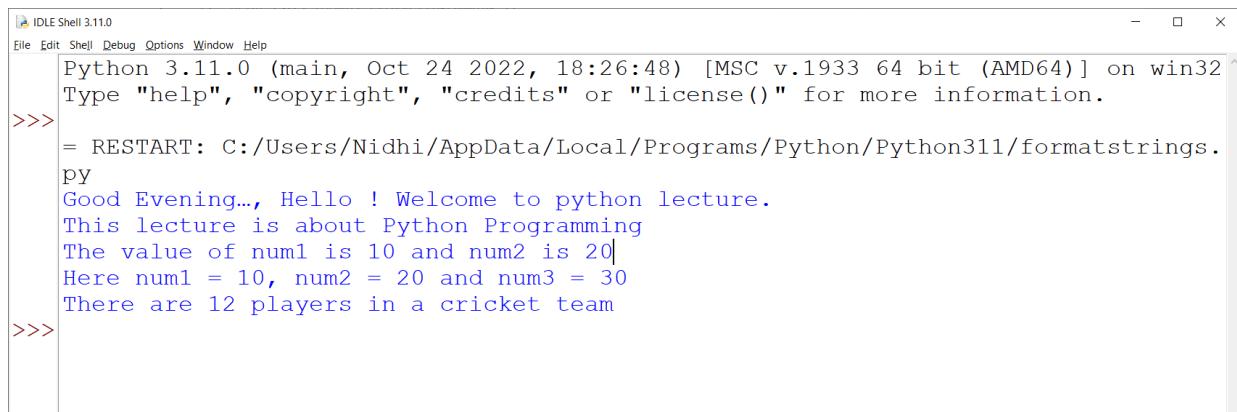
print('Here num1 = {0}, num2 = {1} and num3 =
      {2}'.format(num1,num2,num3))

# Demo of format() with a numeric constant value

print ("There are {} players in a cricket team"
       .format(12))
```

Output:

The output can be seen in *Figure 1.26*:



```

IDLE Shell 3.11.0
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>>
= RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/formatstrings.py
Good Evening..., Hello ! Welcome to python lecture.
This lecture is about Python Programming
The value of num1 is 10 and num2 is 20
Here num1 = 10, num2 = 20 and num3 = 30
There are 12 players in a cricket team
>>>

```

Figure 1.26: Output

Operators and expressions

Operators are special symbols that perform operations on variables and values. These variables or values are known as operands. So, we may infer that operators operate upon operands. A sequence of operands and operators, such as $a + b - 10$, is called an **Expression**. Python supports many operators for combining data objects into expressions. There are different sets of operators present in Python, namely, Arithmetic Operators, Assignment Operators, Comparison Operators, Logical Operators, Bitwise Operators, Identity Operators, and Membership Operators. Let us discuss each of them briefly.

Arithmetic operators

Arithmetic operators are used to perform mathematical operations such as addition, subtraction, multiplication, and so on. *Table 1.5* displays a list of Arithmetic Operators in Python.

Operator	Operation	Syntax	Example
+	Addition of two operands	$a + b$	<code>print(2 + 3)</code> # Output: 5

Operator	Operation	Syntax	Example
-	Subtraction of two operands	a - b	<code>print(5 - 3)</code> # Output: 2
*	Multiplication of two operands	a * b	<code>print(2 * 3)</code> # Output: 6
/	Division operator (/) returns the quotient obtained after dividing the first operand by the second	a / b	<code>print(10 / 2)</code> # Output: 5.0
//	Floor Division operator (//) divides the first operand by the second and ignores the decimal value if present in Quotient.	a // b	<code>print(15 // 2)</code> # Output: 7
%	Modulo operator (%) for a%b returns remainder obtained after division of a and b	a % b	<code>print(5 % 2)</code> # Output: 1
**	Power operator (**) for a**b returns the power of a raised to b	a ** b	<code>print(4 ** 2)</code> # Output: 16

Table 1.5: List of arithmetic operators in Python

Assignment operators

Assignment operators are used to assign values to variables. The assignment operator works in such a way that it assigns the value from the right to the variable on to the left. So, its associativity is from right to left. Refer to [Table 1.6](#) where a = 10:

Operator	Description	Syntax	Example

Operator	Description	Syntax	Example
=	Assign the value of the right side of the expression to the left side operand	c = a + b	c = a + 2 print(c) # Output: 12
+=	Add and Assign: Add right side operand with left side operand and then assign to left operand	a += b	a += 1 print(a) # a = a + 1 = 11
-=	Subtract and Assign: Subtract the right operand from the left operand and then assign it to the left operand. True if both operands are equal	a -= b	a -= 1 print(a) # a = a - 1 = 9
*=	Multiply and Assign: Multiply the right operand with the left operand and then assign it to the left operand	a *= b	a * = 2 print(a) # a = a * 2 = 20
/=	Divide and Assign: Divide left operand with right operand and then assign to left operand	a /= b	a /= 2 print(a) # a = a / 2 = 5.0
%=	Modulus and Assign: Takes modulus using left and right operands and assigns the result to the left operand	a %= b	a %= 2 print(a) # a = a % 2 = 0
//=	Floor Division and Assign: Floor divide the left operand with the right operand and then assign the value to the left operand	a //= b	a // = 2 print(a) # a = a//2 = 5

Operator	Description	Syntax	Example
<code>**=</code>	Exponent and Assign: Calculate exponent (raise to power) value using operands and assign value to left operand	<code>a **= b</code>	<code>a **= 2</code> <code>print(a)</code> <code># a= a**2 = 100</code>

Table 1.6: List of assignment operators in Python

Comparison operators

Comparison operators, also known as **Relational operators**, are used in comparing values and return either **True** or **False** as the result. The different comparison operators are Greater than, Greater than or equal to, less than, lesser than or equal to, equal to, and not equal to. In [Table 1.7](#), let a = 10:

Operator	Description	Syntax	Example
<code>></code>	Greater than operator returns True if the left operand is greater than the right operand	<code>a > b</code>	<code>print(a > 20)</code> # results in False
<code><</code>	Less than operator returns True if the left operand is less than the right operand	<code>a < b</code>	<code>print(a < 20)</code> # results in True
<code>==</code>	Equal to operator returns True if both operands are equal operand	<code>a == b</code>	<code>print(a == 20)</code> # results in False
<code>!=</code>	Not equal to operator returns True if both operands are not equal.	<code>a != b</code>	<code>print(a != 20)</code> # results in True

<code>>=</code>	Greater than or equal to operator returns True if the left operand is greater than or equal to the right operand.	<code>a >= b</code>	<code>print(a >= 20)</code> # results in False
<code><=</code>	Less than or equal to operator returns True if the left operand is less than or equal to the right operand	<code>a <= b</code>	<code>print(a <= 20)</code> # results in True

Table 1.7: List of comparison operators in Python

Logical operators

Logical operators are used to check whether an expression is True or False. They are used in decision-making. Logical operators perform Logical AND, Logical OR, and Logical NOT operations. It is used to combine conditional statements. In [Table 1.8](#), let `a = 15`.

Operator	Description	Syntax	Example
<code>and</code>	Logical AND operator returns True if both the operands (or Expressions) are True at the same time.	<code>Expr1 and Expr2</code>	<code>print((a>10) and (a<20))</code> # returns True
<code>or</code>	Logical OR operator returns True if either of the operands (or Expressions) is True.	<code>Expr1 or Expr2</code>	<code>print((a>10) or (a>20))</code> # returns True

Operator	Description	Syntax	Example
not	Logical NOT operator returns True if the operand is False or returns False if the operand is True. Thus, it inverts the output of expression.	Not (Expression)	<code>print(not((a>10) and (a<20)))</code> # returns False

Table 1.8: List of logical operators in Python

Bitwise operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name, and perform the bit-by-bit operations. These are used to operate on binary numbers. Consider two variables: a = 10 (0000 1010) in binary format and b = 4 (0000 0100) in binary format. Refer to [Table 1.9](#):

Operator	Description	Syntax	Example
&	Bitwise AND operator that sets each bit to 1 if both bits are 1; otherwise results 0.	a & b	<code>print(a & b)</code> # Output: 0 (0000 0000)
	Bitwise OR operator that sets each bit to 1 if one of the two bits is 1 or both bits are 1. It results 0 only if both bits are 0.	a b	<code>print(a b)</code> # Output: 14 (0000 1110)
^	Bitwise Exclusive OR (XOR) operator that sets each bit to 1 if only one of the two bits is 1. It results 0 if both bits are either 0 or 1.	a ^ b	<code>print(a ^ b)</code> # Output: 14 (0000 1110)
~	Bitwise NOT operator inverts all bits, that is, inverts bit 0 to 1 and vice-versa.	~a	<code>print(~a)</code> # Output: -11

Operator	Description	Syntax	Example
>>	Bitwise Signed Right Shift operator shifts the bits of the number to the right and fills 0 on the extreme left. It fills 1 in the extreme left bit position in case of a negative number. The effect is like dividing the number with the power of 2 equivalent to the number of shifts.	a >> n Here, n is the number of shifts.	For unsigned number Let a = 10 = 0000 1010 (Binary) a >> 1 = 0000 0101 = 5 a >> 2 = 0000 0010 = 2 For signed numbers use 2's complement form Let a = -10 print(a >> 1) # Output: -5 print(a >> 2) # Output: -3

Operator	Description	Syntax	Example
<code><<</code>	Bitwise left shift operator. It Shifts the bits of the number to the left and fills 0 on voids right as a result. The effect is similar to multiplying the number with the power of 2 equivalent to the number of shifts.	b << n Here, n is the number of shifts.	For unsigned number. Let c = 5 = 0000 0101 (Binary) c << 1 = 0000 1010 = 10 c << 2 = 0001 0100 = 20 For signed negative number Let d = -10 <code>print(d << 1)</code> # Output: -20 <code>print(d << 2)</code> # Output: -40

Table 1.9: List of bitwise operators in Python

Identity operators

In Python, the membership operators—is and is not being used to check if two values are located on the same part of the memory. Two variables that have equal values do not imply that they are identical. Consider two integers or two strings having the same values. Then, they are considered equal as well as identical because they have immutable constant values. Now consider two lists, tuples, sets, and so on, having the same values. They are considered equal but not identical because the interpreter locates them separately in memory, although they are equal in terms of values. Let us consider the following values:

Integers: a = 10, b = 10

Strings: str1 = 'Hello', str2 = 'Hello'

Lists: list1 = [1,2,3,4,5], list2 = [1,2,3,4,5]

Refer to *Table 1.10*:

Operator	Description	Example
is	True if the operands are identical (It refers to the same object, that is, the same memory location)	<code>print(a is b)</code> #returns True <code>print(str1 is str2)</code> #returns True <code>print(list1 is list2)</code> #returns False
is not	True if the operands are not identical (It do not refer to the same object, that is, different memory locations)	<code>print(a is not b)</code> #returns False <code>print(str1 is not str2)</code> #returns False <code>print(list1 is not list2)</code> #returns True

Table 1.10: List of identity operators in Python

Membership operators

In Python, in and not in are the two membership operators. They are used to check whether a value or variable exists in a sequence such as string, list, tuple, set, and dictionary or not. In a dictionary, we can only check for the existence of a key but not the value. Let us consider the following values:

String: str1 = 'Hello world'

Dictionary: `dict1 = {1 : 'a', 2 : 'b', 3 : 'c', 4 : 'd'}`

Refer to [Table 1.11](#):

Operator	Description	Example
<code>in</code>	True if value/variable is found in the sequence	<code>print('H' in str1)</code> # return True <code>print('hello' in str1)</code> # returns False <code>print(2 in dict1)</code> # returns True <code>print(7 in dict1)</code> # returns False
<code>not in</code>	True if value/variable is not found in the sequence	<code>print('K' not in str1)</code> # returns True <code>print(9 not in dict1)</code> # returns True

Table 1.11: List of membership operators in Python

Operator precedence

Consider the following expression:

`10 + 5 * 10`

Here is confusion or ambiguity to calculate this expression. Should Python perform the addition $10 + 5$ first and then multiply the sum by 10, or should the multiplication $5 * 10$ be performed first and the addition of 10 later. If the first approach is used, then the output should be 150, and if the second approach is followed, then the output should be 60. The solution to this ambiguity is given by the concept of Precedence.

All operators that the language supports are assigned a precedence. In an expression, all operators of the highest precedence are evaluated first. Once those results are obtained, operators of the next highest precedence are evaluated. This continues until the expression is fully evaluated. Note that operators having equal precedence are performed in left-to-right order. [Table 1.12](#) shows the order of precedence of the Python operators you have seen so far, from lowest to highest:

Precedence Order	Operator	Description
Highest precedence	**	Exponentiation
	+x, -x, ~x	Unary positive, unary negation, bitwise negation
	*, /, //, %	Multiplication, division, floor division, modulo
	+, -	Addition, subtraction
	<<, >>	Bit shifts
	&	Bitwise AND
	^	Bitwise XOR
		Bitwise OR
	==, !=, <, <=, >, >=, is, is not	Comparisons, identity
	not	Boolean NOT
	and	Boolean AND
Lowest precedence	or	Boolean OR

Table 1.12: Precedence order of operators in Python

For example: The result for Expression $2 * 3 ** 4 * 5$ is 810 as per preceding table order.

Namespaces in Python

The word Namespace is made up of two words: Name (which means name, a unique identifier) and Space (related to scope or location for access). A namespace is a simple system to control the names in a program. It ensures that names are unique and will not lead to any conflict. Python implements namespaces in the form of dictionaries. It maintains a name-to-object mapping where names act as keys and objects as values. We may think namespace just like a surname for the name. For example, in a class, there may be many students named “Raman,” but their surnames differentiate them from each other and make them unique entities in the class. Refer to [*Figure 1.27*](#):

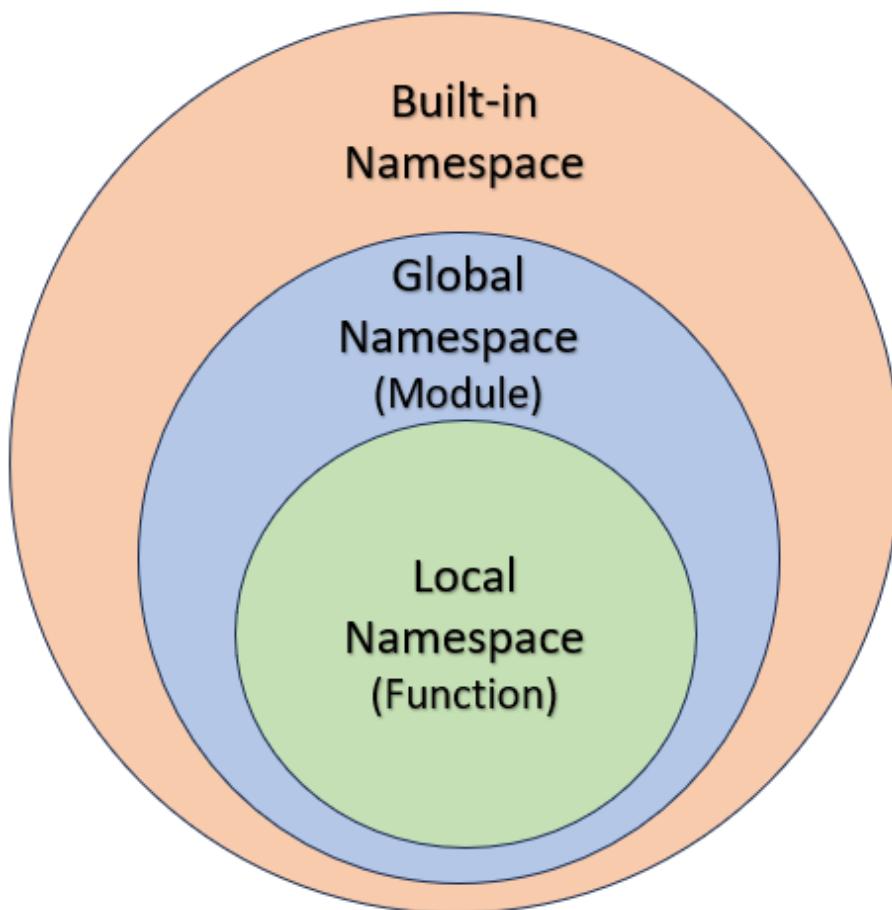


Figure 1.27: Types of namespaces

Namespace helps the Python interpreter understand which exact method or variable is trying to point out in the code. The types of Namespace are listed as follows:

- **Local Namespace:** This namespace covers the local names inside a function. Python creates this namespace for every function called in a program. It remains active until the function returns.
- **Global Namespace:** This namespace covers the names of various imported modules used in a project. Python creates this namespace for every module included in your program. It will last until the program ends.
- **Built-in Namespace:** This namespace covers the built-in functions and built-in exception names. Python creates it as the interpreter starts and keeps it until you exit.

Namespaces make our programs immune from name conflicts. However, it does not give us a free ride to use a variable name anywhere we want.

Python restricts names to be bound by specific rules known as a scope. The scope determines the parts of the program where you could use that name without any prefix. We will discuss more about Namespace and scope concepts in the upcoming chapters.

Conclusion

In this chapter, we have discussed the basics of writing simple programs in Python. We discussed the setup process of the Python development environment. After that, we also learnt how to write variables, literals, keywords, and comments in Python. We also came across various data types in Python, along with implicit and explicit types conversion. In another section of this chapter, we briefly discussed the Input–Output process for user interaction with the system, different types of operators, and Namespace concept in Python. After completion of this chapter, readers will be able to understand the core programming concepts of Python language. In the upcoming chapters, we will learn more about advanced programming concepts such as loops, decision control, and Functions in Python.

Points to remember

- Python is a scripting language, being interpreted high-level programming language developed for the purpose of fulfilling general

programming requirements.

- Python Keywords are special reserved words that convey a special meaning to the compiler/interpreter.
- A name in a Python program is called an identifier, which can be a class name, function name, module name, or variable name.
- Various Datatypes in Python: Numbers, Boolean, Set, Dictionary, and Sequence Types (Strings, Lists, and Tuples).
- There are different set of operators present in Python, namely, Arithmetic operators, Assignment Operators, Comparison Operators, Logical Operators, Bitwise Operators, Identity Operators, and Membership Operators.
- A namespace is a simple system to control the names in a program and ensure that names are unique and will not lead to any conflict.
- The three types of namespace supported in Python are Local Namespace (Function), Global Namespace (Module), and Built-in Namespace.

Exercise

Attempt the following projects.

Sample project with solution

Create a simple calculator in Python that can perform addition, subtraction, multiplication, and Division in Python.

Solution:

```
# Print message for user to select the relevant
operator for specific calculation
```

```
operator = input(''
```

Please type in the operator for calculation:

+ For Addition

```
- For Subtraction
* For Multiplication
/ For Division
''')

# Prompt the user to enter two numbers as input for
# calculation

operand_1 = int(input('Enter the first number: '))
operand_2 = int(input('Enter the second number: '))

# Specify conditional statements for calculator

# The first condition checks whether the user has
# entered '+' operator for addition

# If '+' operator is selected then add the two
# operands and display result

if operator == '+':
    print('{} + {} = '.format(operand_1,
operand_2))
    print(operand_1 + operand_2)

# The second condition is specified using elif i.e.
# else if conditional statement.

# It checks whether the user has entered '-'
# operator for Subtraction

# If '-' operator is selected then subtract the two
# operands and display result

elif operator == '-':
```

```
    print('{} - {} = '.format(operand_1,
operand_2))

    print(operand_1 - operand_2)

# The Third condition is specified using elif i.e.
# else if conditional statement.

# It checks whether the user has entered '*' operator for Multiplication

# If '*' operator is selected then multiply the two operands and display result

elif operator == '*':
    print('{} * {} = '.format(operand_1,
operand_2))

    print(operand_1 * operand_2)

# The Fourth condition is again specified using elif i.e. else if conditional statement.

# It checks whether the user has entered '/' operator for Division

# If '/' operator is selected then divide the two operands and display result

elif operator == '/':
    print('{} / {} = '.format(operand_1,
operand_2))

    print(operand_1 / operand_2)

# The last condition is specified using else conditional statement.
```

```

# This statement will be implemented when none of
# the above condition applies.

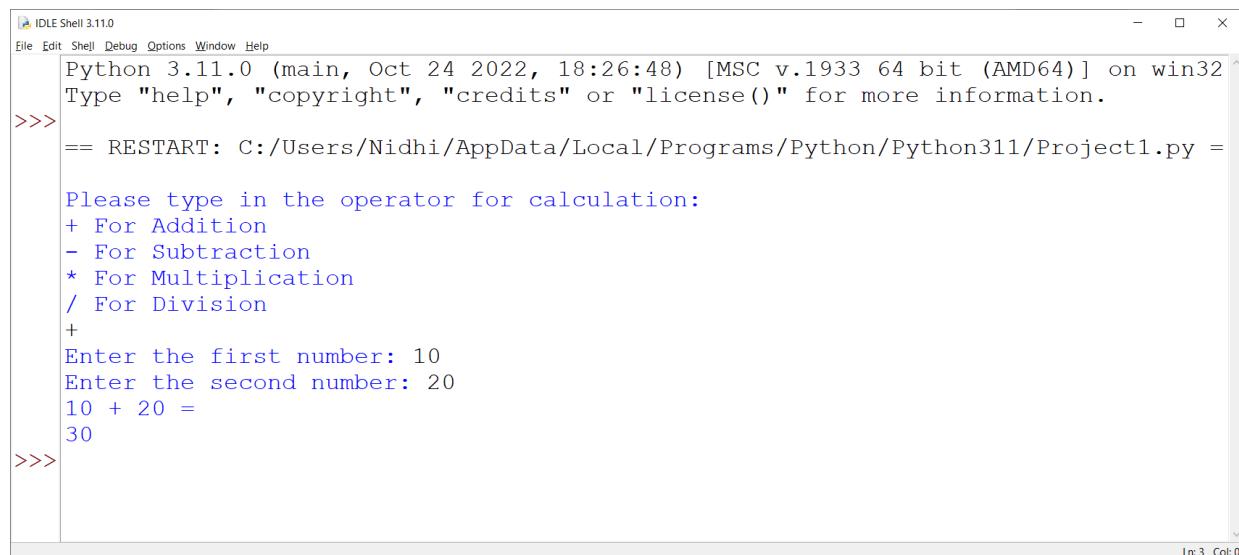
# Specifically, when user enters any other key from
# keyboard except '+', '-', '*' and '/'.

else:
    print('You have not typed a valid operator,
please run the program again.')

```

Output:

The output can be seen in *Figure 1.28*:



```

IDLE Shell 3.11.0
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> == RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/Project1.py =
Please type in the operator for calculation:
+ For Addition
- For Subtraction
* For Multiplication
/ For Division
+
Enter the first number: 10
Enter the second number: 20
10 + 20 =
30
>>>

```

Figure 1.28: Output for Project 1

Practice project

Create a Mad Libs Generator Game to create a new story every time with a different set of inputs.

To put it simply, MadLibs is an interesting game where you compose a tale with blanks in it and then ask another player to fill them in. The outcome is frequently humorous and funny because only you know what is happening in your story. The first step in creating the game is to develop a tale, which we will then translate into a MadLib game and implement it in Python code.

The following given is a short story—The Hare and the Tortoise:

Once upon a time, there was a hare who was friends with a tortoise. Both lived near the forest. One day, he challenged the tortoise to a running race. Seeing how slow the tortoise was going, the hare thought he would win this easily. So he decided to sleep for some time while the tortoise kept on going. He had a beautiful dream of eating carrots. When the hare woke up, he saw that the tortoise was already at the finish line. Much to his distress, the tortoise won the race while he was busy sleeping.

This story, when reframed with blanks for the MadLib game, will be as follows:

Once upon a time there was a ____ who was friends with a _____. Both lived near the _____. One day, he challenged the _____ to a _____ race. Seeing how slow the _____ was going, the _____ thought he'll win this easily. So he decided to _____ for some time while the _____ kept on going. He saw _____ dream of eating _____. When the _____ woke up, he saw that the _____ was already at the finish line. Much to his _____, the _____ won the race while he was busy _____.

Here, we can see a lot of missing words. As per parts-of-speech in English grammar, we will ask the user to enter these blanks and recreate the story with user inputs. Write code to implement the MadLib game in Python.

Hint: Ask the user to enter missing values such as the name of the animal or place, and so on. Then, put those values to recreate the story.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord.\(bpbonline.com](https://discord(bpbonline.com)



CHAPTER 2

Flow Control Concepts

Introduction

In the previous chapter, we have learnt about basic programming concepts in Python. We got a fair idea of how to declare and assign variables along with identifier naming conventions, various Datatypes in Python, and Type conversion, that is, Implicit and Explicit conversion, with hands-on examples. We also discussed in detail the Input/output process, the working of different types of operators, and Namespace in Python. In this chapter, we will take a look at Flow Control concepts in Python. We will go through numerous working examples of decision-making processes using if...else statement and iterative control using types of loops. By the end of this chapter, we will also discuss applications of Break, Continue, and Pass statements while writing code in Python. This chapter will help us to develop solutions for more complicated decision-based and iterative problems in Python.

Structure

In this chapter, we will discuss the following topics:

- Decision-making in Python
- Workflow of if...elif...else
 - IF statement
 - If... Else statement
 - If... Elif... Else statement

- Nested If
- Loop control in Python
 - Python for Loop
 - Python while Loop
 - Infinite while Loop
 - Nested Loop
- Flow control statements in Python

Objectives

This chapter gives a detailed knowledge of different flow control and decision-making constructs in Python. Multiple working examples related to conditional decision-making using if...else statement and iterative flow control using while and for loops have also been discussed. This will enable the readers to develop a firm base for writing more complicated and decision-making applications in Python. The knowledge gained in this chapter will help the audience understand the concepts and code used in upcoming chapters.

Decision-making in Python

Decision-making is the anticipation of conditions occurring during the execution of a program and specified actions taken according to the conditions. Decision structures evaluate multiple expressions, which produce **TRUE** or **FALSE** as the outcome. A control flow of a program refers to the sequence in which the code contained in the program executes. In Python, the control flow of a program is managed by conditional statements, loops, and function calls.

Workflow of if...elif...else

In Python, we use the if statement to execute a block of code only when a specific condition is satisfied. There are three forms of the **if...else** statement, and they are as follows:

- The **if** statement
- The **if...else** statement
- The **if...elif...else** statement

Figure 2.1 provides a flowchart for the **if-elif-else** decision-making process:

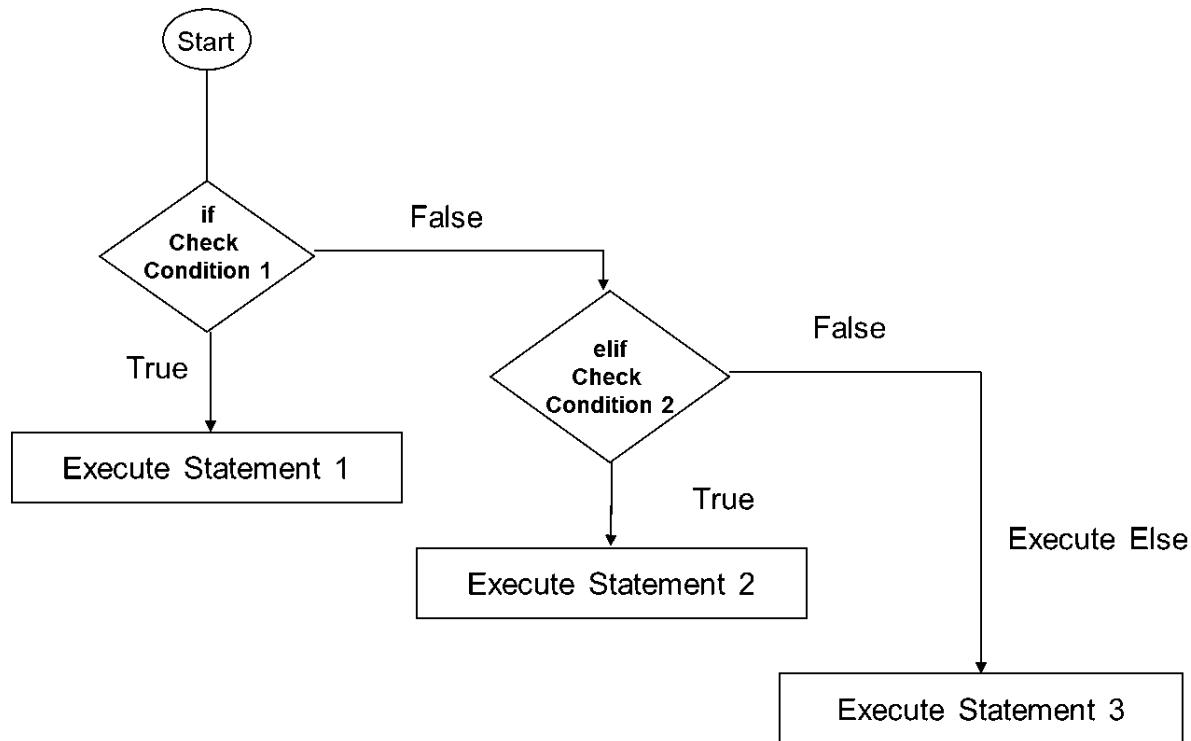


Figure 2.1: Flowchart for the if-elif-else decision-making process

The flowchart shown in *Figure 2.1* gives an overview of the decision-making process for the combined **if-elif-else** statement. Here, the Python program begins by first checking the test condition-1 for **True** or **False**. If the test condition results to **True**, only then the statement(s) in the body of the **if** statement is(are) executed. Otherwise, the control skips to check for condition-2 in **elif** (an acronym for **else-if**). If condition-2 evaluates to **True**, then the statement(s) in the **elif** block is(are) executed. Otherwise, the control jumps out to execute statement 3. This hierarchy is referred to as the **if-elif-else** ladder because it resembles the structure of a ladder in terms of **if** statements. If one of the conditions turns out to be

true, then the rest of the ladder is just bypassed, and the body of that particular block is executed. A Python program may contain an **if** statement followed by multiple **elif** statements and a single **else** statement at the end.

The if statement

The syntax of **if** statement in Python is as follows:

```
if test-condition:
```

```
    # body of if statement
```

The **if** statement evaluates a test condition for True/False. If evaluated to True, then the statement(s) inside the **if** block is(are) executed. Otherwise, the control bypasses the body of the **if** block to execute statements given outside the block. Now, let us demonstrate an example of the **if** statement in the following implementation to check if the number entered by the user is in the range of -100 to 100:

```
num = int(input("Enter a number to check if it is
in range (-100 to 100) : "))

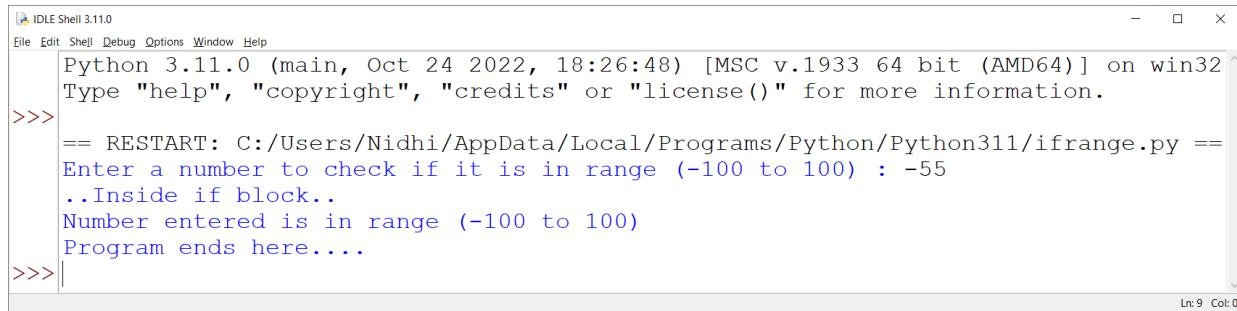
if (num > -100 and num < 100):
    print(..Inside if block..)

    print("Number entered is in range (-100 to
100)")

print("Program ends here....")
```

Output:

The output can be seen in *Figure 2.2*:



```
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> == RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/ifrange.py ==
Enter a number to check if it is in range (-100 to 100) : -55
..Inside if block..
Number entered is in range (-100 to 100)
Program ends here.....
>>> Ln: 9 Col: 0
```

Figure 2.2: Output

Note here that the input entered by the user using the `input()` function is by default of string type. That is why it has to be converted to integer type using the `int()` function as in Line 1. Moreover, we can see the use of logic **and** operator to check simultaneously for two conditions, whether `num > -100` and `num < 100`, as in line 2.

The if...else statement

An **if** statement can have an optional **else** clause. This form of **if** statement is used to decide among two possible decisions based on the **True/False** result of the test condition. The syntax of if...else statement is as follows:

```
if test-condition:
    # block of code if condition is True
else:
    # block of code if condition is False
```

Here, if **test-condition1** evaluates to **True**, then statements in the if block are executed. Otherwise, if **test-condition1** evaluates to **False**, then statement(s) in the else block **is(are)** executed. Now, let us demonstrate an example of checking whether a given number is even or odd using **if...else**:

```
number = int(input("Enter a number to check if even
or odd : "))
if number%2==0:
```

```

print("....Inside if block....")
print('It is Even number divisible by 2')
else:
    print("....Inside else block....")
    print('It is odd number!!')
print('Program ends here!!')

```

Output:

The output can be seen in *Figure 2.3*:

```

IDLE Shell 3.11.0
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> = RESTART: C:\Users\Nidhi\AppData\Local\Programs\Python\Python311\Ifstatement.py
Enter a number to check if even or odd : 51
....Inside else block....
It is odd number!!
Program ends here!!
>>>

```

Figure 2.3: Output

In the same way, readers are suggested to try implement the same code of **if...else** block by giving an even number in user input and analyzing the obtained output.

The **if...elif...else** statement

As seen previously, the **if...else** statement is used to execute a block of code among two possible choices only. But what if we need to make a decision among more than two alternatives? In such a scenario, we may use the **if...elif...else** statement for decision-making. Following is the syntax to be followed for **if...elif...else** statement:

```

if test-condition1:
    # code block 1

```

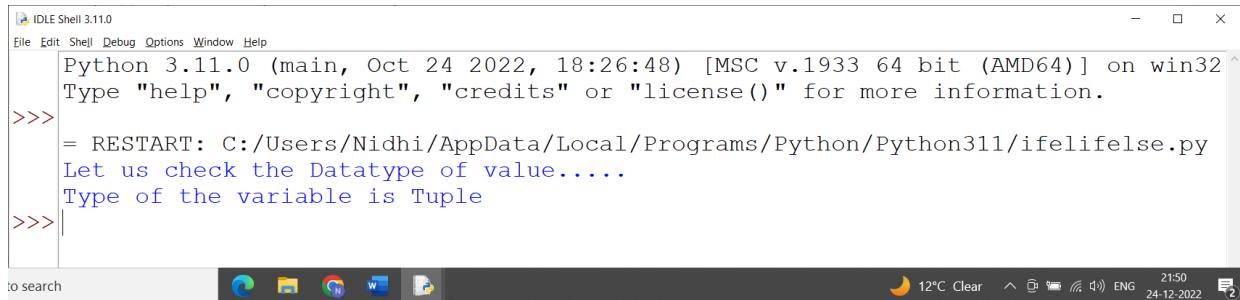
```
elif test-condition2:  
    # code block 2  
else:  
    # code block 3
```

Here, If **test-condition1** evaluates to **True**, then code block 1 is executed. Otherwise, if **test-condition1** evaluates to **False**, then **condition2** is evaluated. If **test-condition2** is **True**, code block 2 is executed, and otherwise, code block 3 is executed. Here, we see a demonstration of **if-elif-else** statements to find the correct datatype of the value given by the user.

```
value1 = (1,2,3,4)  
print("Let us check the Datatype of value.....")  
if (type(value1) == int):  
    print("Type of the variable is Integer")  
elif (type(value1) == str):  
    print("Type of the variable is String")  
elif (type(value1) == tuple):  
    print("Type of the variable is Tuple")  
elif (type(value1) == dict):  
    print("Type of the variable is Dictionaries")  
elif (type(value1) == list):  
    print("Type of the variable is List")  
else:  
    print("Type of the variable given is  
Undefined!!")
```

Output:

The output can be seen in *Figure 2.4*:



```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> = RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/ifelifelse.py
Let us check the Datatype of value.....
Type of the variable is Tuple
>>>
```

Figure 2.4: Output

Here, we can see multiple **elif** statements to check for different conditions to conclude a decision. In any case, only a single code block, whether **if** or **elif** or **else** is executed based on the **True** or **False** result obtained after evaluating the test condition.

Nested if

We can use an **if** statement inside another **if** statement, and so on. This is known as a nested **if** statement. The syntax of nested **if** statement is as follows:

```
# The outer if statement
if condition1:
    # Code statement(s)
# The inner or nested if statement
    if condition2:
        # Code statement(s)
```

Here, we can add **else** and **elif** statements also to the inner **if** statement, if required. We can also insert the inner **if** statement inside the outer **else** or **elif** statements. We can nest multiple layers of **if..elif..else** statements as per our requirement. In the following example, we have nested

if statements to find the maximum of three numbers by taking input from the user:

```
# Taking 3 numbers from user input to check which
# is greater

num1 = int(input("Enter 1st number : "))

num2 = int(input("Enter 2nd number : "))

num3 = int(input("Enter 3rd number : "))

# To find largest number using nested if...else

if num1 >= num2:

    if num1 >= num3:

        print(num1, "is the largest number of all")

    else :

        print(num3, "is the largest number of all")

else :

    if num2 >= num3:

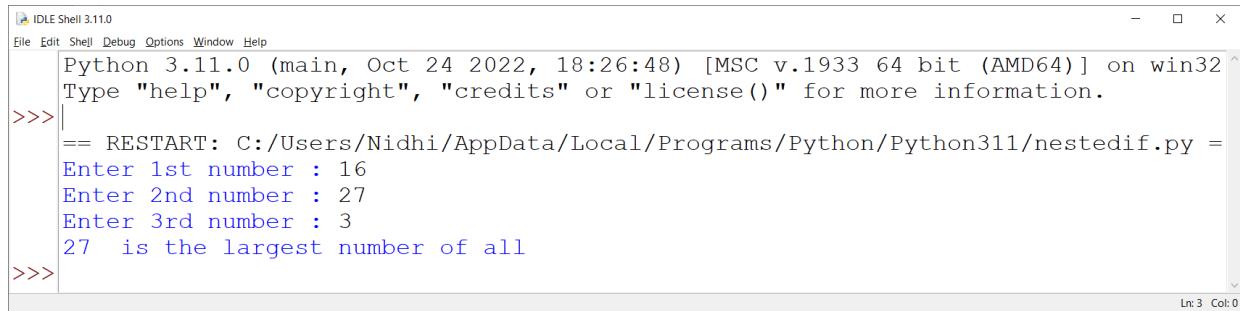
        print(num2, " is the largest number of
all")

    else :

        print(num3, " is the largest number of
all")
```

Output:

The output can be seen in *Figure 2.5*:



```
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> == RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/nestedif.py =
Enter 1st number : 16
Enter 2nd number : 27
Enter 3rd number : 3
27  is the largest number of all
>>> Ln: 3 Col: 0
```

Figure 2.5: Output

Loop control in Python

Loops are capable of repeating or iterating some specific lines of code multiple times. For example, if we want to show a certain message to a user 1,000 times, then we can use a loop to iterate and display the message 1,000 times successfully. Apart from this, there are some other advantages of loops in Python, which are as follows:

- Helps in code re-usability.
- No need to write the same lines of code again and again.
- Enables easy traversal of elements in data structures such as arrays, lists, and so on.

The flowchart given in [Figure 2.6](#) shows the general working of the loop in Python. It starts with initializing an iteration variable and checking a test condition to decide the workflow of the loop. If the condition evaluates to **True**, then the statement(s) contained in the body of the loop is(are) executed, after which the loop control continues with the next iteration and repeats all the steps again. This process continues till either the loop condition fails or the number of iterations specified gets completed. After this, the loop exits, executing the code statements followed after the loop.

Refer to [Figure 2.6](#):

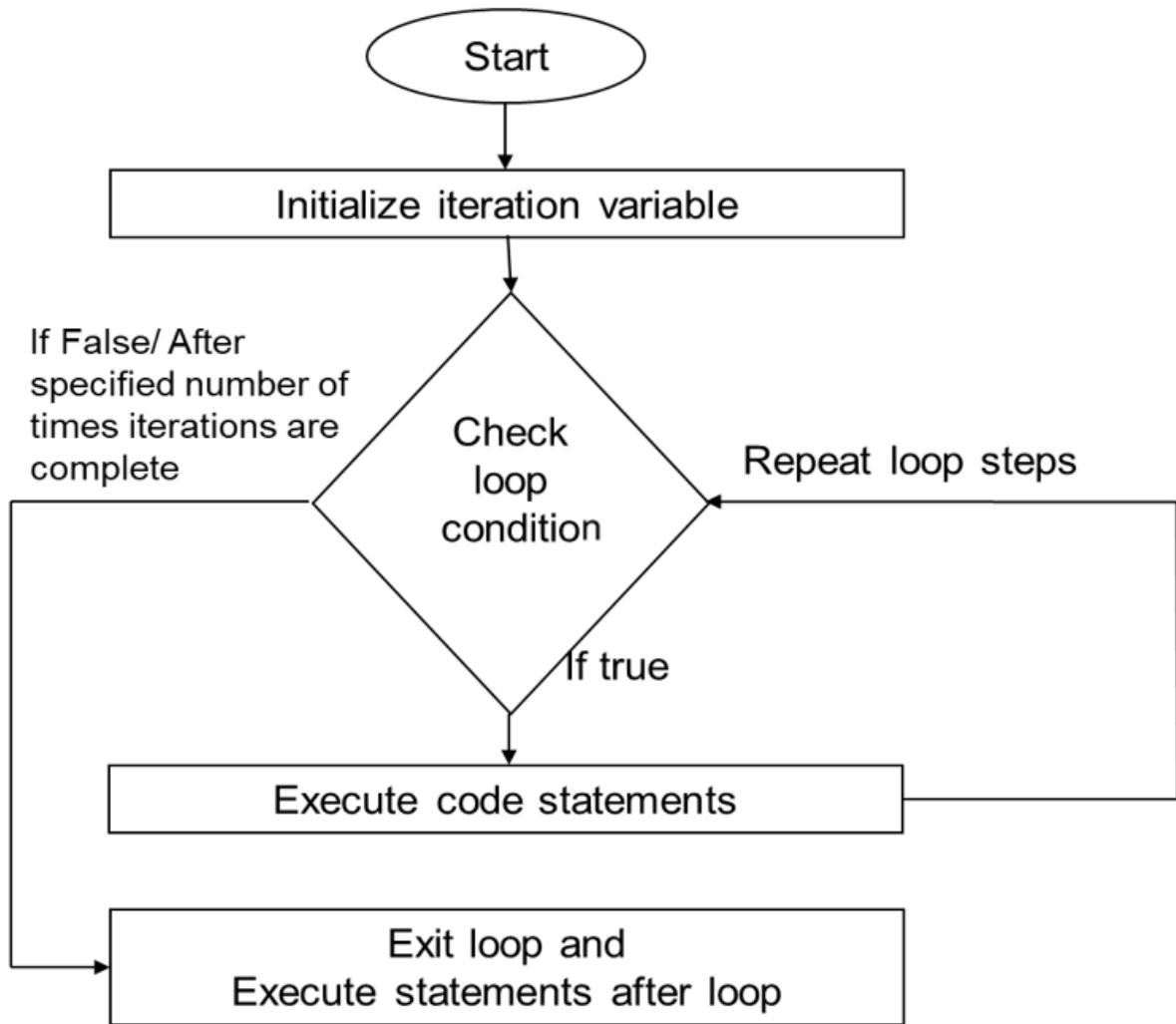


Figure 2.6: Flowchart to display the working of loops

There are two types of loops in Python, as explained in [Table 2.1](#):

Type of Loop	Working
for loop	For loop should be used if the number of repetitions or iterations is known in advance. It is used in the case where we need to execute code statements until the given condition is satisfied. It is an entry-controlled or pre-tested loop.
while loop	The while loop should be used when the number of iterations is not known in advance. In the while loop, the block of code statements is

executed until the specified condition is satisfied. It is an entry-controlled or pre-tested loop.

Table 2.1: Types of loops

Python for Loop

The Python For loop is used to iterate over the items of any sequence such as String, Tuple, List, Set, or Dictionary. The syntax used in **For** loop is as follows:

```
for <iterating_var> in sequence:  
    # Body of for loop  
    else: {optional}  
        # Statement(s)
```

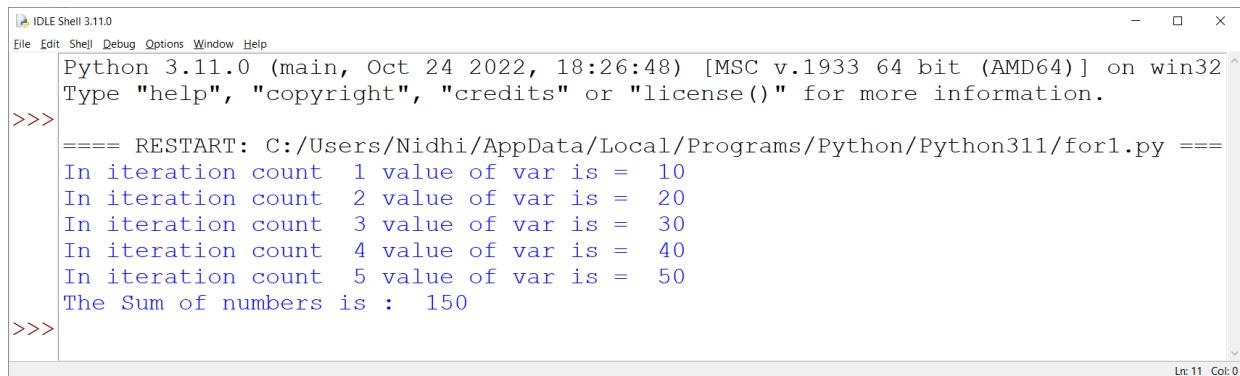
The indented statements inside the for loops are executed once for each item in a sequence. The iteration variable takes the value of the next item of the sequence each time through the loop. A **for** loop can have an optional else block, which is executed when the loop is finished. In the following code, we demonstrate the working of a **for** loop to calculate the sum of items declared in a tuple:

```
numbers = (10, 20, 30, 40, 50)  
sum = 0  
count = 1  
for var in numbers:  
    print("In iteration count ", count, "value of  
var is ", var)  
    sum = sum + var  
    count = count + 1  
else:
```

```
print('The Sum of numbers is : ', sum)
```

Output:

The output can be seen in *Figure 2.7*:



```
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> ===== RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/for1.py ====
In iteration count 1 value of var is = 10
In iteration count 2 value of var is = 20
In iteration count 3 value of var is = 30
In iteration count 4 value of var is = 40
In iteration count 5 value of var is = 50
The Sum of numbers is : 150
>>>

Ln: 11 Col: 0
```

Figure 2.7: Output

The output displays five iterations along with the updated value of the iteration variable **var** for each loop iteration. We can analyze that the value of variables **sum** and **count** also get updated during each iteration. Finally, when the loop iterations are completed, the accompanying **else** block executes to print the sum of tuple items as output.

Python while Loop

Python while loop is used to execute a specific target statement as long as the specified test condition is **True**. The syntax of a while loop is as follows:

```
while test-condition:
    # Body of while loop
    else: {optional}
        # Statement(s)
```

Here, a **while** loop evaluates the test condition for **True/False**. If the condition evaluates to **True**, the code inside the **while** loop is executed, and the condition is evaluated again to continue the next iteration. This process is repeated until the test condition evaluates to **False**, and the loop stops

iterating further. A while loop can have an optional else block as well, which is executed when the loop is finished. The following example checks for Even and Odd numbers between 1 and 10 using a **while** loop:

```
start = 1
end = 20

# while loop to check for even/ odd numbers from
start to end

print("Checking for Even/ Odd numbers in range 1-
10..")

num = start

while num <= end:

    if(num%2 == 0):
        print(num, " is even number")

    elif num==0:
        print(num, " is zero")

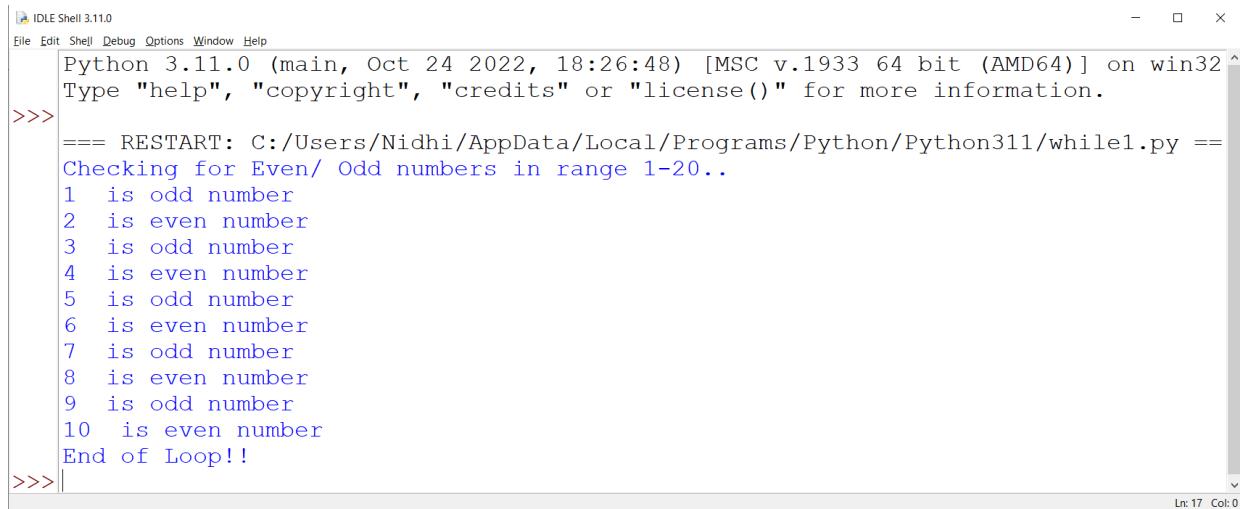
    else:
        print(num, " is odd number")

    num = num + 1

else:
    print("End of Loop!!")
```

Output:

The output can be seen in *Figure 2.8*:



The screenshot shows the Python IDLE Shell 3.11.0 interface. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following text:

```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> === RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/while1.py ==
Checking for Even/ Odd numbers in range 1-20..
1  is odd number
2  is even number
3  is odd number
4  is even number
5  is odd number
6  is even number
7  is odd number
8  is even number
9  is odd number
10 is even number
End of Loop!!
>>> Ln: 17 Col: 0
```

Figure 2.8: Output

Infinite while Loop

If the condition of a loop is always **True**, the loop runs for an infinite number of times or until the system runs out of memory space. For example:

```
# Syntax of infinite while loop

while True:

    # Body of while loop

    else: {optional}

    # Statement(s)
```

In the preceding syntax, the condition is always True. Hence, the loop body will run for an infinite number of times. Once the loop ends, the control moves to execute the **else** block statements. Here is an example to calculate the square root of an input entered by a user an infinite number of times:

```
# import the math module

import math

while True:
```

```

        num = int(input("Enter a number to see its
square root: "))

        # calculate square root using math.sqrt() and
print the result

        sqroot = math.sqrt(num)

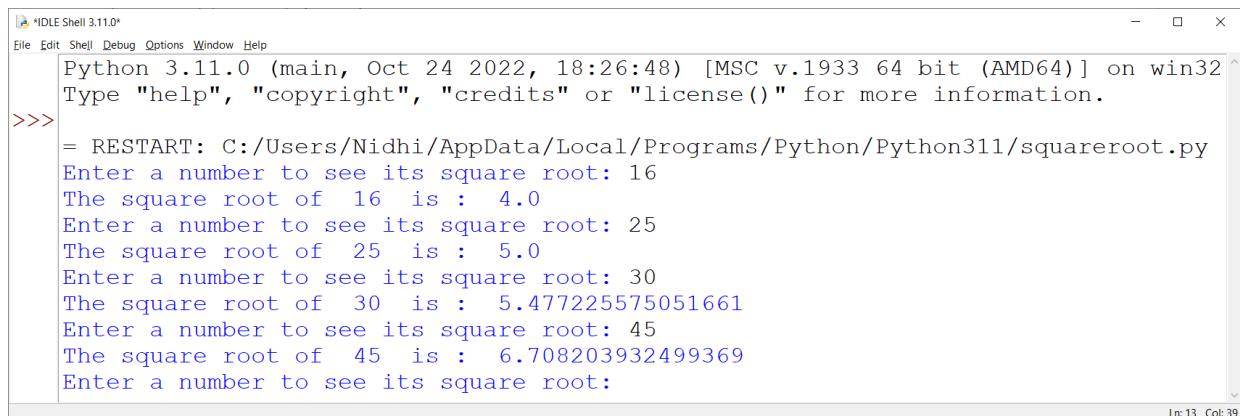
        print("The square root of ", num, " is :
", sqroot)

        # We must press Ctrl+c to exit from the loop

```

Output:

The output can be seen in *Figure 2.9*:



The screenshot shows the Python 3.11.0 IDLE Shell window. The title bar reads "IDLE Shell 3.11.0". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following text:

```

Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>>
= RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/squareroot.py
Enter a number to see its square root: 16
The square root of  16  is :  4.0
Enter a number to see its square root: 25
The square root of  25  is :  5.0
Enter a number to see its square root: 30
The square root of  30  is :  5.477225575051661
Enter a number to see its square root: 45
The square root of  45  is :  6.708203932499369
Enter a number to see its square root:

```

The status bar at the bottom right shows "Ln: 13 Col: 39".

Figure 2.9: Output

In the preceding *Figure 2.9*, to calculate the square root of a number, we import the math module in Line 2. We use the **sqrt()** function present in the math module to calculate the square root of a number entered by the user. It keeps iterating infinite times until the user presses *Ctrl + C* as a keyboard interrupt to exit the loop forcefully.

Nested loop

A loop may contain another loop inside it. A loop inside another loop is called a nested loop. Python does not impose any limit on the number of loops that can be nested inside a loop or on the levels and types of loops

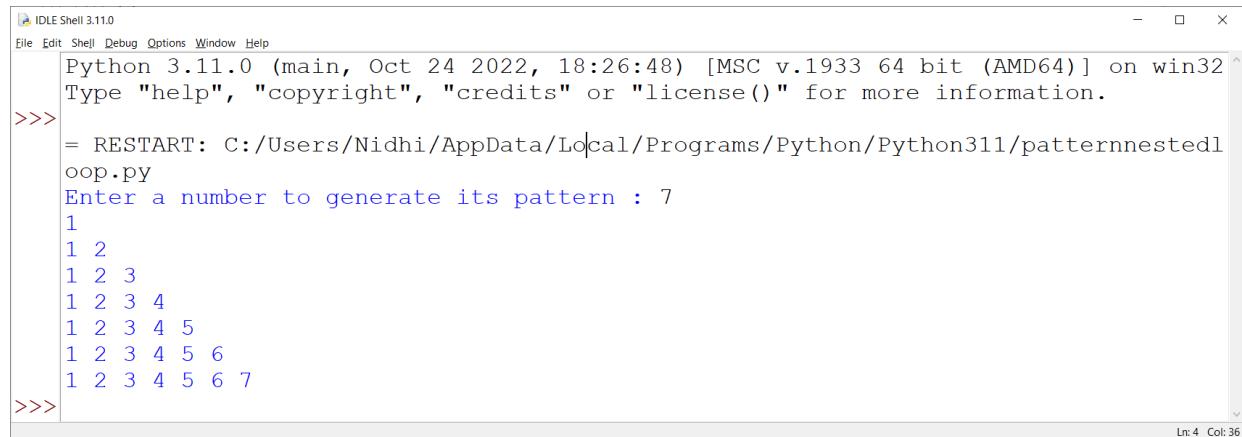
while nesting. Thus, any type of loop (for or while) may be nested within another loop (**for** or **while**) any number of times. Let us see an example of using nested loops in Python for printing a pattern based on the number input given by the user:

```
num = int(input("Enter a number to generate its pattern : "))

for i in range(1,num + 1):
    for j in range(1,i + 1):
        print(j, end = " ")
    print()
```

Output:

The output can be seen in *Figure 2.10*:



The screenshot shows the Python IDLE Shell 3.11.0 interface. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The title bar says "IDLE Shell 3.11.0". The main window displays the following text:

```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> = RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/patternnestedloop.py
Enter a number to generate its pattern : 7
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
```

The status bar at the bottom right shows "Ln: 4 Col: 36".

Figure 2.10: Output

Flow control statements in Python

Flow Control Statements change the flow of execution of code in a loop or deviate it from its regular sequence. There are three types of flow control statements in Python, namely:

- Break statement

- Continue statement
- Pass statement

The working of each flow control statement is described in [Table 2.2](#):

Type of Statement	Working	Syntax
Break	<ul style="list-style-type: none"> ❖ Break statement halts the loop and brings the flow control out of the loop to the next statement after the loop. ❖ In the case of nested loops, it halts and jumps out of the inner loop first and then proceeds to outer loop. ❖ It is used in the scenario where we need to break the loop for a given condition. 	break
Continue	<ul style="list-style-type: none"> ❖ The continue statement brings the program flow control to the beginning of the loop. ❖ It forces to skip the remaining lines of code inside the loop and jump to the next iteration. ❖ It is used for the scenario where it is required to skip some specific code for a specific condition. 	continue
Pass	<ul style="list-style-type: none"> ❖ It is used to execute nothing i.e. the pass statement can be used to execute empty in the scenario where we do not want to execute the code statements. ❖ It just makes the control to pass by without executing any code. If we want to bypass any code pass statement can be used. 	pass

Table 2.2: Types of flow control statements

The following example shows the working of the break statement with a while loop to display the first 5 multiples of 2:

```
# Code to find first 5 multiples of 2

var = 1

while (var <= 10):
    print('2 * ',(var), '=' ,2 * var)

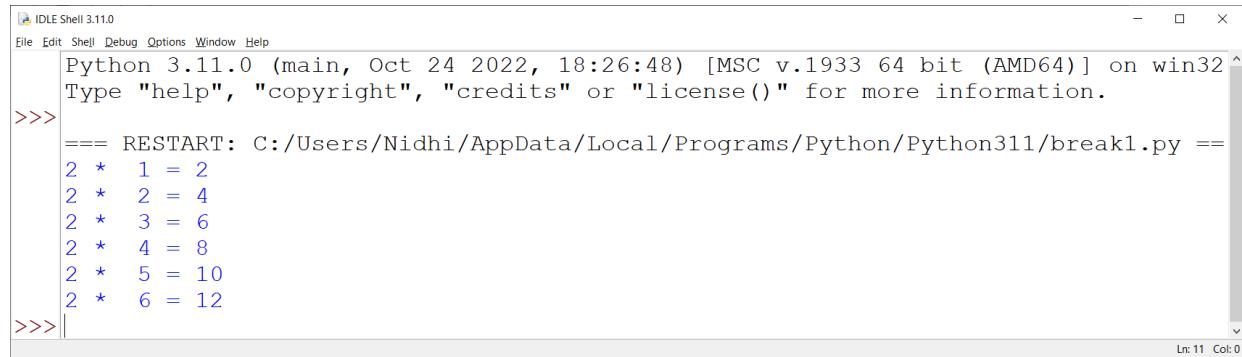
    # Check if var>5 to break out of loop otherwise
    # increase var by 1

    if var > 5:
        break

    var = var + 1
```

Output:

The output can be seen in *Figure 2.11*:



```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>> === RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/break1.py ==
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12

>>> Ln:11 Col:0
```

Figure 2.11: Output

In Python, we can also skip the current iteration of the while loop using the `continue` statement. The following code demonstrates the use of the `continue` statement to print all even numbers between 1 and 10:

```
# To print all even numbers from 1 to 10

num = 0

print("All even numbers from 1-10 are : ")

while num < 10:

    num += 1

        # Check if num not divisible by 2 i.e., for odd
        number

            # continue with the next iteration of the loop
            otherwise

                # for even number print num in Line 11.

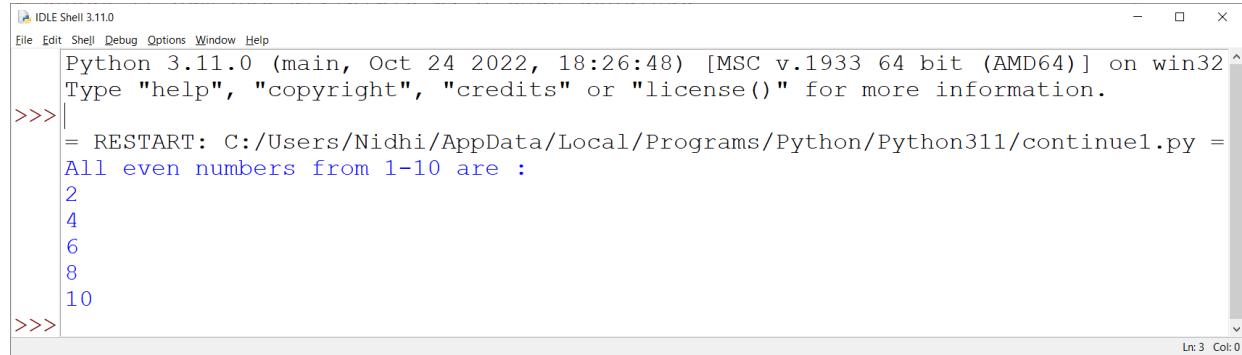
                if (num % 2)!= 0:

                    continue

                print(num)
```

Output:

The output can be seen in *Figure 2.12*:



```
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> = RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/continuel.py =
All even numbers from 1-10 are :
2
4
6
8
10
>>> Ln: 3 Col: 0
```

Figure 2.12: Output

Now, suppose we have a loop or a function that is not implemented yet or is still under construction, although we may implement it in the future. In such cases, we can use the `pass` statement for No Operation. Consider the following example for a `pass` statement:

```
age = int(input("Enter your age: "))

# use pass inside if statement

# if enetered age > 100 then simply pass

# else print a message for user

if age > 100:

    pass

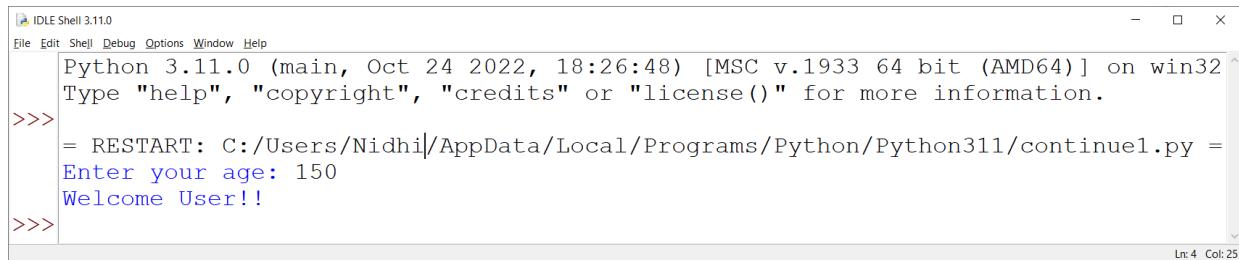
else:

    print("Your age is : ", num)

print('Welcome User!!')
```

Output:

The output can be seen in *Figure 2.13*:



```
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> = RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/continuel.py =
Enter your age: 150
Welcome User!!
>>>
Ln: 4 Col: 25
```

Figure 2.13: Output

Conclusion

In this chapter, we discussed the core concepts of decision-making, loop control, and flow control statements in Python. The if...elif...else statement enables controlled decision-making in Python with a single if block followed by multiple (and optional) elif blocks and an else block (also optional) at the end. There are two types of loops present in Python, which allow iterating a block of code multiple times or till a test condition results True. We can also use flow control statements such as break, continue, and pass statements to customize our decision-making in an application.

Points to remember

- The if...elif...else statement is used for decision-making in Python.
- The looping constructs, such as while and for loop, allow repeated execution of a block of code a specific number of times or on the basis of the True/False result of a test-condition.
- The for loop statement iterates over a range of values or a sequence such as String, List, Tuple, and so on.
- The statements under the body of a while loop are iterated over and again until the test-condition of the while results to False.
- If the condition of the while loop initially resulting in False, then the loop body does not execute even once.
- If the test-condition specified in the loop does not terminate in a finite number of times, then it will become an infinite or never-ending loop.

- The break statement immediately exits a loop, skipping the rest of the loop's body. Execution continues with the statement immediately following the body of the loop.
- On the contrary, in the case of the continue statement, the control jumps to the beginning of the loop for the next iteration.
- We use a Pass statement to execute No-operation and just pass the code.

Exercise

Attempt the following projects.

Sample project with solution

Create text-based Adventure Game using a single player to demonstrate the usage of decision-based flow control in Python.

This text-based adventure game is an application that consists of a series of questions or riddles. The user has to enter his name to begin the game or play as “Guest” with three lifelines at hand. The user must answer the questions in either Yes/ No or True/ False.

(Note: You may try implementing the program with different answers and choices). If the user answers all answers correctly in any of the three lifelines, then he wins. Otherwise, he loses the game.

Solution:

```
import time

print("You have only 3 lifelines to answer all
questions correct and win!!")

lifeline =3

#Storing the user's name. If no name given then
assume Guest.

username = input("\nEnter your name?")
```

```
if username=="":  
    username="Guest"  
  
print ("\nWelcome, " + username +", \nGame Rule :  
You can give answer as True/False or Yes/No.")  
  
time.sleep(3)    # sleep(3) pauses the program for  
3ms.  
  
while(lifeline>=1):  
    print("Your current Lifeline Number is : ",  
lifeline)  
  
    true_ans = ["T", "t", "True", "Yes", "y",  
"yes", "Y"]  
  
    false_ans = ["F", "f", "False", "No", "no",  
"N", "n"]  
  
    correct = 0 #Storing the correct answers  
  
    print("\n.....The game  
begins.....")  
  
    print("\nHere is the first Question.")  
  
    time.sleep(3)  
  
    print ("\nCanberra is the captial of  
Australia?")  
  
    choice = input("Type your Answer Here : ")  
  
    if choice in true_ans:  
  
        correct += 1 #If correct, the user gets one  
point  
  
    else:
```

```
    correct += 0 #If incorrect, 0 is awarded for  
that answer
```

```
    print ("\nThe islands of Seychelles are located  
in Pacific Ocean?")
```

```
    choice = input("Type your Answer Here : ")
```

```
    if choice in false_ans:
```

```
        correct += 1
```

```
    else:
```

```
        correct += 0
```

```
    print ("\nA normal Human has 210 bones?")
```

```
    choice = input("Type your Answer Here : ")
```

```
    if choice in false_ans:
```

```
        correct += 1
```

```
    else:
```

```
        correct += 0
```

```
    print ("\n Did Dmitri Mendeleev designed  
Periodic table in Chemistry?")
```

```
    choice = input("Type your Answer Here : ")
```

```
    if choice in true_ans:
```

```
        correct += 1
```

```
    else:
```

```
        correct += 0
```

```
    print ("\nIf speed of a moving car and distance
travelled by it is given, can we calculate time
taken for this journey?")

    choice = input("Type your Answer Here : ")

    if choice in true_ans:
        correct += 1

    else:
        correct += 0

    print ("Sun rises from the West?")
    choice = input("Type your Answer Here : ")

    if choice in false_ans:
        correct += 1

    else:
        correct += 0

    print("\nTask over, " + username +". You got",
correct, "out of 6 correct.")

    # Check if correct answers are equal to 6 or
not.

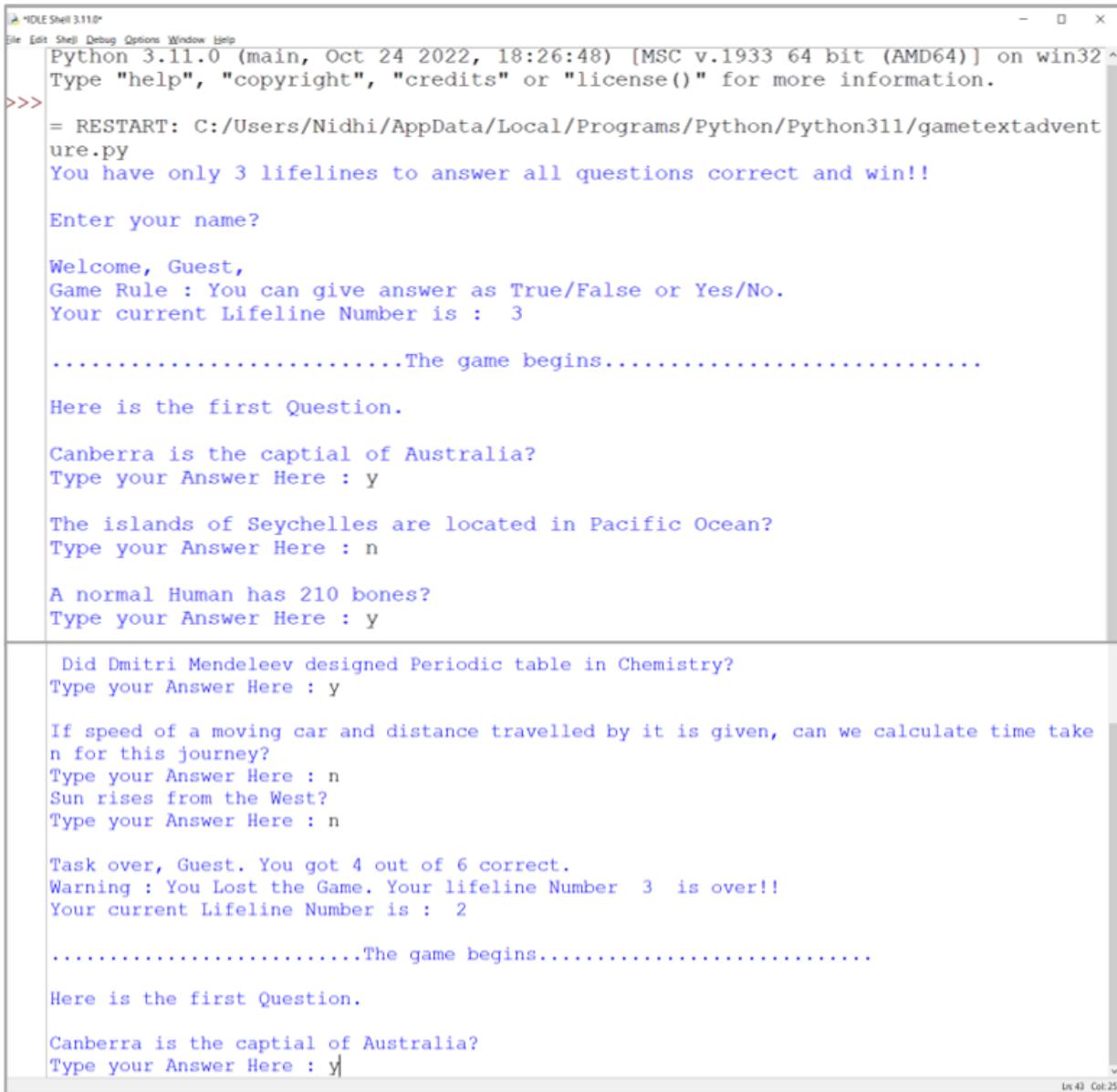
    # If all correct answers then exit the loop
else keep continuing the game.

    if correct == 6:
```

```
        print("Congratulations!!You won the
Game!!")
        break
    else:
        print("Warning : You Lost the Game. Your
lifeline Number ",lifeline," is over!!")
        lifeline = lifeline - 1
else:
    print("!! Game Over !! Better Luck Next Time
!!")
```

Output:

The output can be seen in *Figure 2.14*:



```
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>>
= RESTART: C:/Users/Nidhi/AppData/Local/Programs/Python/Python311/gametextadventure.py
You have only 3 lifelines to answer all questions correct and win!!

Enter your name?

Welcome, Guest,
Game Rule : You can give answer as True/False or Yes/No.
Your current Lifeline Number is : 3

.....The game begins.....  

Here is the first Question.

Canberra is the capital of Australia?
Type your Answer Here : y

The islands of Seychelles are located in Pacific Ocean?
Type your Answer Here : n

A normal Human has 210 bones?
Type your Answer Here : y

Did Dmitri Mendeleev designed Periodic table in Chemistry?
Type your Answer Here : y

If speed of a moving car and distance travelled by it is given, can we calculate time taken for this journey?
Type your Answer Here : n
Sun rises from the West?
Type your Answer Here : n

Task over, Guest. You got 4 out of 6 correct.
Warning : You Lost the Game. Your lifeline Number 3 is over!!
Your current Lifeline Number is : 2

.....The game begins.....  

Here is the first Question.

Canberra is the capital of Australia?
Type your Answer Here : y
```

Figure 2.14: Output for project 1

Practice project

Create a Number Guessing application for a user in Python by giving him clues at every step. The number-guessing game is built on the player's notion to estimate a number between the provided range. If the player predicts the desired number, the player wins; otherwise, the player loses the game. Because this game has limited efforts, thus, the player needs to

indicate the number of the limited attempts. Otherwise, the player will lose the game.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 3

Data Structures and Algorithms

Introduction

In previous chapters, we learned the basics of writing simple programs in Python using various datatypes, decision-making with **if-elif-else** block, and loop constructs such as **while** and **for** loops. In this chapter, we will extend our knowledge by learning coding on a new platform known as PyCharm IDE. We have already discussed about the basics of built-in data structures such as Lists, Tuples, Set, Dictionary, and String with a few working examples in [*Chapter 1, Getting Started with Programming in Python*](#). Here, we will discuss these topics in detail, along with a view of their utility in real-time projects. Furthermore, we will have a detailed discussion on various user-defined data structures such as Tree, Heap, Linked List, Stack, Queue, and Graph. After this, we will also have a look at various searching and sorting algorithms and their applications in Python.

Structure

In this chapter, we will discuss the following topics:

- Introduction to PyCharm IDE
- Built-in data structures
 - String
 - List
 - Tuple
 - Set
 - Dictionary
- User-defined data structures

- Linked list
- Stack
- Queue
- Sorting algorithms
 - Time complexity and space complexity
 - Bubble Sort
 - Selection Sort
 - Insertion Sort
- Searching algorithms
 - Linear Search
 - Binary Search

Objectives

By the end of this chapter, the reader will be able to implement different built-in and user-defined data structures in Python. Users will also get in-depth knowledge of various sorting and searching techniques used in Python.

Introduction to PyCharm IDE

In this chapter, we introduce a new platform known as PyCharm IDE, for executing Python applications. Programmers frequently use PyCharm, an **Integrated Development Environment (IDE)** made by the Czech company JetBrains, for the Python programming language. It was first released in February 2010 and is written in Java and Python. Linux, macOS, and Windows versions all support PyCharm. It offers a graphical debugger, integrated unit testing, coding aid, support for Django Web development, and Anaconda platform integration.

Installation steps of PyCharm IDE

To install PyCharm IDE, visit its official website <https://www.jetbrains.com/pycharm/> and follow the given steps:

1. Click on the **Download** menu button to download the PyCharm IDE setup.

2. As per requirement, we may choose the **Community Edition** or **Professional Edition** for downloading its .exe file for the Windows operating system. The professional edition of PyCharm requires a subscription, while the community edition is free.
3. Once the download is complete, run the .exe file to install PyCharm.
4. Click on the **Finish** button to complete the setup. On starting the PyCharm IDE, we can see the welcome page of the IDE.
5. To create a new application, click on the **New Project** option, give an appropriate name for the project, such as **MyProject**, and click the **Create** button.
6. To add a new Python file in the project, Right click on the project and select **New | Python File**. Give an appropriate name for a Python file with **.py** extension, such as **DemoApplication1.py**. Once all configurations are complete, we can see the new project screen, as shown in *Figure 3.1*:

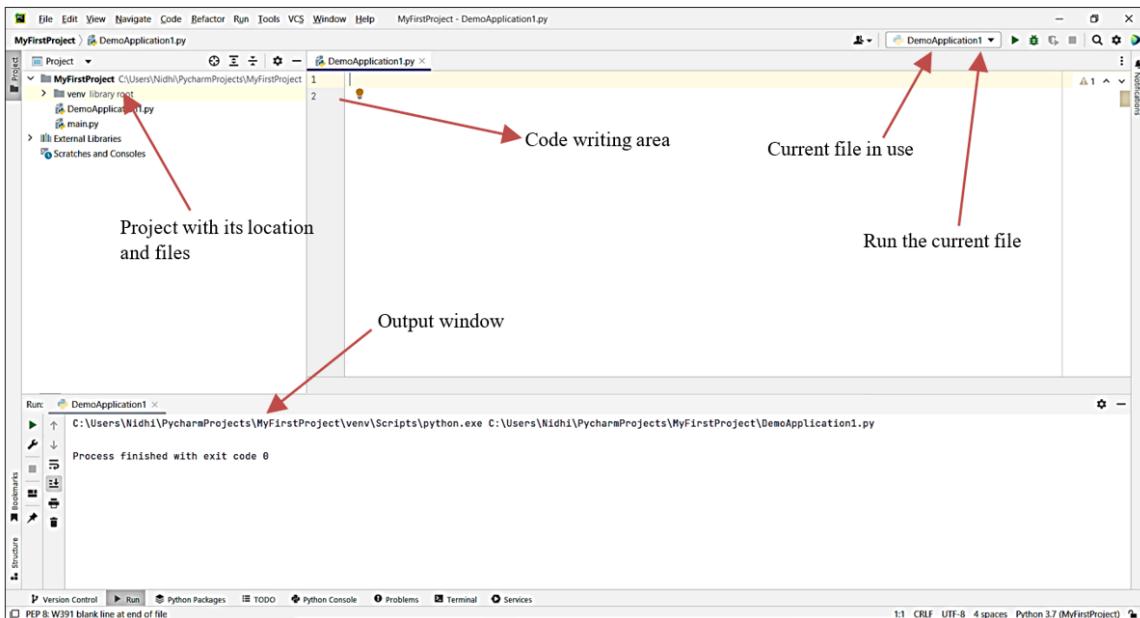


Figure 3.1: Component details of PyCharm IDE screen

Now, we can write and execute Python code in our newly created **DemoApplication1.py** file in the **MyProject** application. To run an application, click on **Run Menu**, and then click on **Run DemoApplication1.py**.

Built-in data structures

A collection of information obtained through measurements, study, observations, or analysis is referred to as data. Facts, numbers, names, figures, or even descriptions of things may be included. Python's data structures are a useful way to organize and store data in a computer so that it may be used and processed effectively, depending on the situation. Python data structures can be broadly classified into two groups, namely, Built-in and User-defined Data Structures.

Figure 3.2 shows the different categories of data structures in Python:

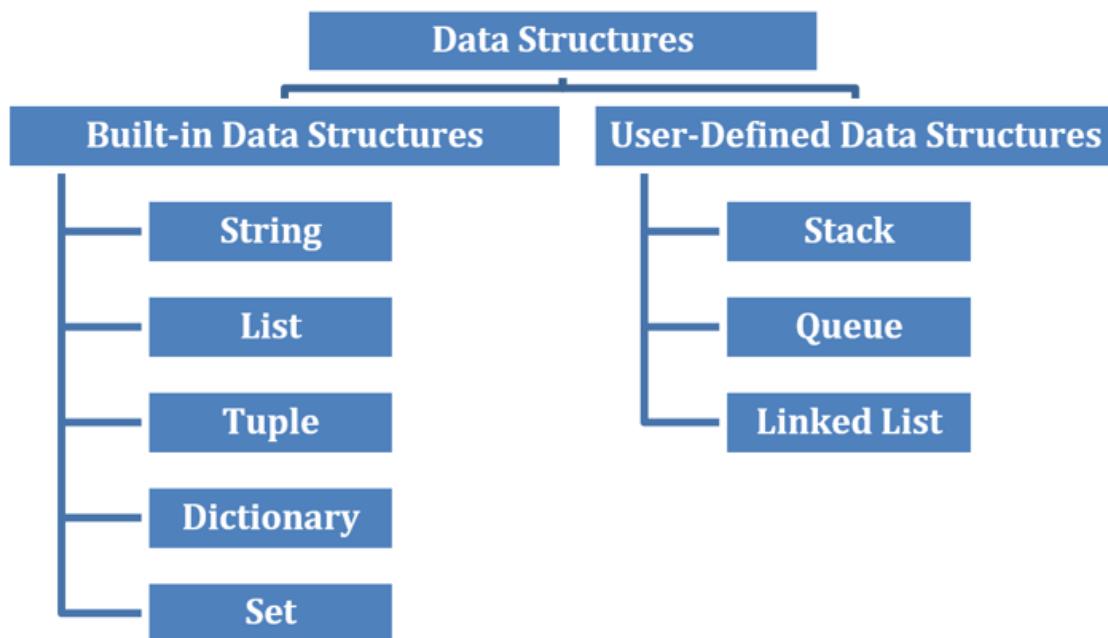


Figure 3.2: Various data structures in Python

String

As we already know, a string is a sequence of characters. For example, “welcome” is a string containing a sequence of characters “w,” “e” “l,” “c,” “o,” “m,” and “e.” We may use single quotes or double quotes to represent a string in Python. We can also create a multiline string in Python, using triple double quotes “”” or triple single quotes ‘’. Readers can revisit Chapter 1, Getting Started with Programming in Python again, to revise the basics of Strings. There are various operators that enable users to manipulate strings in an efficient manner. A summary of string operators is given in *Table 3.1*:

Operator	Description
+	The concatenation operator adds values on either side of the operator.
*	The repetition operator creates new strings, concatenating multiple copies of the same string.
[]	The slice operator gives the character from the given index.
[start: stop: step]	<p>The Range Slice operator gives the characters in the given range only. The parameters are as follows:</p> <p>start => Starting index where the slicing of the object starts.</p> <p>stop => Ending index where the slicing of the object stops.</p> <p>step => An optional argument that determines the increment between each index for slicing.</p>
in not in	The membership operator returns true if a character exists in the given string. On the contrary, not in operator returns true if a character does not exist in the given string.
r/R	Raw String operator suppresses the actual meaning of Escape characters. The “r” can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark. Raw strings do not treat the backslash as a special character; rather, every character we put into a raw string remains as it is.
%	<p>Format operator performs String formatting such that:</p> <p>%s: Used for string format specifier.</p> <p>%d: Used for decimal integers.</p> <p>%o or %0: Used for octal integers.</p> <p>%x or %X: Used for Hexadecimal integers.</p>

Table 3.1: List of String operators

Let us demonstrate the usage of String operators with the following example. For this, we shall write the given code in **DemoApplication1.py**, created in the project **MyFirstProject** in PyCharm IDE. The code example is as follows:

```

str1 = "Hello, Trainees!"

str2 = "Welcome to learn Python."

str3 = "Hello, Trainees!"

# Compare str1 and str2

```

```
print(str1 == str2)

# Compare str1 and str3

print(str1 == str3)

# Concatenation using + operator

result = str1 + str2

print(result)

# Slice str1 from 1st index to 10th index with skip of
# 2 characters

print(str1[1:10:2])

# Membership test using in and not in

print('a' in 'Java')      # returns True

print('n' not in 'Python') # returns False

# String formatting using %

print("His name is %s and age is %d years!" % ('John',
25))

# Raw string using r or R

print(r'C:\Folder\SubFolder\File1')

# Repeat the string "Bye" 2 times

result = "Bye" * 2

print(result)
```

Output:

The output can be seen in [*Figure 3.3*](#):

```

False
True
Hello, Trainees!Welcome to learn Python.
el,Ta
True
False
His name is John and age is 25 years!
C:\Folder\SubFolder\File1
ByeBye

```

Figure 3.3: Output

Apart from the previously mentioned operators, there are various string methods and functions present in Python that enable users in quick string manipulation. Both function and method seem to be similar, but there is a slight difference between the two. A function can be called only by its name, as it is defined independently. However, methods cannot be called by their name; we need to call the method using the object of that class in which it is defined. Following are some of these string methods. The syntax to access the method is **Stringname.method()**. Let **string str1 = " Hello to Python World"**.

Refer to [Table 3.2](#) for an illustration of the string methods with respective examples:

String Method	Description	Example
<code>count(str, beg= 0, end=len(string))</code>	It counts the number of occurrences of substring str a string from beginning index beg till index “end.”	<code>print(str1.count('Pytho'))</code> Output: 1

String Method	Description	Example
<code>endswith(sufx, beg=0, end=len(string))</code>	It is used to find if a string ends with a suffix sufx or not, starting the search from the beginning to end index.	<pre>print(str1.endswith('m')) Output: False</pre>
<code>find(str, beg=0, end=len(string))</code>	Used to find if a substring str occurs in a string beginning search from index beg till index end. If str is found, then its index is returned. Otherwise, it returns -1.	<pre>print(str1.find('to ')) Output: 6 It searches for substring 'to ' in str1 and returns 6</pre>
<code>index (str, beg=0, end=len(string))</code>	It works similar to find() but with a difference—if str is not found, then it raises ValueError exception.	<pre>print(str1.index("python")) Output: ValueError: substring not found</pre>

String Method	Description	Example
<pre>replace(oldstring, newstring, count)</pre> <p>*Note: Here, the count represents the number of replacements of an old substring with a new one.</p>	<p>It replaces each matching occurrence of a substring with another string. If the count parameter is not specified, the <code>replace()</code> method replaces all occurrences of the old substring with the new string.</p>	<pre>str2 = 'hot coffee serv hot weather' # replacing 'hot' with print(str2.replace('hot 'cold'))</pre> <p>Output: cold coffee se in cold weather</p>

String Method	Description	Example
isalnum() isalpha() isdigit()	<p>Here, isalnum() returns true if the string has at least one character and the characters are either alphabets or numbers only.</p> <p>The isalpha() returns True if all characters in the string are alphabets only, and isdigit() returns true if a string contains numeric values only.</p>	<pre>new_str = "abc123" print(new_str.isalnum()) print(new_str.isalpha()) print(new_str.isdigit()) Output: True False False</pre>
islower() isupper()	<p>islower() returns true if all string characters are in lowercase and false otherwise.</p> <p>isupper() returns True if all string characters are in uppercase; else, False.</p>	<pre>print("hello".islower()) Output: True print(("HELLO").isupper) Output: True</pre>

String Method	Description	Example
strip() rstrip() lstrip()	strip() removes the leading and trailing whitespaces from the string, whereas the rstrip() method removes all trailing whitespaces only, and lstrip() removes only the leading whitespace in a string.	<pre>str2 = ' Python Demo ' print(str2.strip()) Output: Python Demo</pre>
split(separator, maxsplit)	It uses a delimiter or separator (by default, space) to split strings into a list of substrings in Python. If maxsplit is specified, the list will have a maximum of maxsplit+1 items. Both parameters are optional.	<pre>fruits = 'Apple, Mango Peach, Olive' # When maxsplit is 2 print(fruits.split(',')) # When no maxsplit is given print(fruits.split(','))</pre>
		Output: ['Apple', ' Mango', ' Peach, Olive'] ['Apple', ' Mango', ' B Peach', ' Olive']

String Method	Description	Example
splitlines(keeplineends) Here, <code>keeplineends</code> is optional, specifying if the line ends be included (True) or not (False). The default is False.	It splits a string consisting of multiple lines at the line end and returns a list of all lines in the string.	<code>str3 = '''I want to learn Python Coding.''' print(str3.splitlines()) Output: ['I', 'want to', 'Python Coding']</code>

Table 3.2: String methods with examples

Let us see [Table 3.3](#) for some string functions:

String function	Description	Example
<code>len(string)</code>	It gives the length of a string, that is, the total number of characters present in the string.	<code>str2 = "Hello World" print(len(str2)) Output: 11</code>
<code>max(string)</code>	It gives the maximum alphabetical character from the given string as output.	<code>str2 = "Hello World" print(max(str2)) Output: 'r'</code>
<code>min(string)</code>	It gives the minimum alphabetical character from the given string as output.	<code>str2 = "hello" print(min(str2)) Output: 'e'</code>
<code>str(object)</code>	This function converts the given object of any data type into a corresponding string.	<code>num = 101.24 s1 = str(num) print(s1, type(s1)) Output:101.24 <class 'str'></code>

Table 3.3: String functions with examples

List

The list is one of the popular datatypes available in Python. It consists of a sequence of comma-separated elements that may or may not be of the same

datatype enclosed in square brackets []. The list indices start at 0, and lists can be sliced, concatenated, and so on. For example, consider a list: `list1 = ['English', 'Maths', 90, 80]`. Various list methods and functions are given in [Table 3.4](#). Python includes the following list of methods:

List method	Description	Example
<code>append()</code>	This method appends a new element at the end of the current list.	<pre>countries = ['India', 'Australia', 'Dubai'] countries.append('US') print(countries) Output: ['India', 'Australia', 'Dubai', 'US']</pre>
<code>clear()</code>	This method returns an empty list after removing all elements in the current list.	<pre>numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9,] numbers.clear() print('List after clear(): ', numbers) Output: List after clear(): []</pre>
<code>copy()</code>	This method creates and returns a copy of the current list.	<pre>list1 = [1, 2, 3, 4, 5] copy_list = list1.copy() print('Copied List is : ', copy_list) Output: Copied List: [1, 2, 3, 4, 5]</pre>
<code>count()</code>	This method returns the count of occurrences of an element within the list.	<pre>numbers = [12, 33, 51, 12, 61, 12, 7, 9] count = numbers.count(12) print('Count of 12 is ', count) Output: Count of 12 is 3</pre>

List method	Description	Example
<code>extend()</code>	It adds one or more elements present in another list to the end of the current list.	<pre>numbers = [1, 2, 3] new_numbers = [4, 5] new_numbers.extend(numbers) print('List after extend() :', new_numbers) Output: List after extend(): [1, 2, 3, 4, 5]</pre>
<code>index()</code>	Returns the index of the first occurrence of the element with the specified value.	<pre>countries = ['India', 'Australia', 'Dubai'] index = countries.index('Dubai') print(index) Output: 2</pre>
<code>insert()</code>	Adds an element at the specified position.	<pre>vowels = ['a', 'e', 'i', 'u'] # 'o' is inserted at index 3 (4th position) vowels.insert(3, 'o') print('New List:', vowel) Output: New List: ['a', 'e', 'i', 'o', 'u']</pre>
<code>pop()</code>	It removes the element at the specified position in the current list.	<pre>numbers = [22, 33, 55, 77] rem_element = numbers.pop(2) print('Removed Element: ', rem_element) print('Updated List: ', numbers) Output: Removed Element: 55 Updated List: [22, 33, 77]</pre>

List method	Description	Example
remove()	It removes the first occurrence of an element with the specified value from the current list.	<pre>numbers = [22, 77, 33, 55, 77, 99, 111, 77] numbers.remove(77) print('Updated List: ', numbers) Output: Updated List: [22, 33, 55, 77, 99, 111]</pre>
reverse()	Reverses the order of the list.	<pre>numbers = [22, 33, 5, 77, 99, 111] numbers.reverse() print('Reversed List:', numbers) Output: Reversed List:[111, 99, 77, 5, 33, 22]</pre>
sort()	Sorts the list.	<pre>numbers = [17, 11, 33, 77, 55, 22, 9] numbers.sort() print(numbers) Output: [9, 11, 17, 22, 33, 55, 77]</pre>

Table 3.4: List methods with examples

Let us see [Table 3.5](#) for some list functions. Consider a list: **numbers = [17, 11, 3, 7, 5, 2, 9]**:

List function	Description	Example
min(list_name)	Returns the minimum value from a list in Python.	<pre>print(min(numbers)) Output: 2</pre>
max(list_name)	Returns the largest value from a list in Python.	<pre>print(max(numbers)) Output: 17</pre>

List function	Description	Example
<code>len(list_name)</code>	Returns the number of elements in a list in Python.	<pre>numbers = [17, 11, 7, 5, 2, 9] print(len(numbers))</pre> <p>Output: 6</p>
<code>list(sequence)</code>	Converts the sequence to a list in Python.	<pre>seq = "Hello ALL !!" list1 = list(seq) print(list1)</pre> <p>Output: ['H', 'e', 'l', 'l', 'o', ' ', 'A', 'L', 'l', ' ', '!', '!']</p>

Table 3.5: List functions with examples

Tuple

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. Tuples are different from Lists because they cannot be changed, unlike lists. Moreover, tuples may or may not use parentheses (), whereas lists use square brackets []. Creating a tuple is as simple as putting different comma-separated values. For example, `tup1 = ('physics', 'chemistry', 1997, 2000)`. The methods and functions available for tuple manipulation are given in [Table 3.6](#).

Consider a tuple: `tuple1 = ('h', 'e', 'l', 'l', 'o')`

Tuple method	Description	Example
<code>count()</code>	This method counts and returns the number of times a specific value occurs in a tuple.	<pre>print(tuple1.count('l'))</pre> <p>Output: 2</p>
<code>index()</code>	Searches the tuple for a specified value and returns the position of the first occurrence of where the value was found.	<pre>print(tuple1.index('l'))</pre> <p>Output: 2</p>

Table 3.6: Tuple methods with examples

Similar to the list data structure, Tuples also has various functions such as `min()`, `max()`, `len()`, and `tuple(sequence)`. Readers can try to implement these functions on their own.

Set

A set in Python is an unordered collection of unique elements, which may be of different data types. We cannot access items in a set by referring to an index because sets are unordered, and its elements are not indexed. However, we can iterate through its elements using a for loop and search for a value in a set by using the `in` membership operator. The mutable elements, such as lists, sets, or dictionaries, cannot be used as members in a set. Various set methods are discussed in [Table 3.7](#):

Set method	Description	Example
<code>update()</code>	It updates a set with the union of this set and unique elements of other collection types, such as lists, tuples, sets, and so on.	<pre>set1 = {'Zara', 'Lacoste', 'Fossil'} list1 = ['Apple', 'Meta', 'Google', 'Apple'] set1.update(list1) print(set1) Output: {'Zara', 'Fossil', 'Apple', 'Google', 'Lacoste', 'Meta'}</pre>

Set method	Description	Example
add()	It adds an element to a set.	<pre>numbers = {7, 2, 11, 3, 4, 5} print('The Initial Set:', numbers) numbers.add(22) print('The Updated Set:', numbers) Output: The Initial Set: {2, 3, 4, 5, 7, 11} The Updated Set: {2, 3, 4, 5, 7, 11, 22}</pre>
discard()	It removes a specified item.	<pre>set1 = {'Zara', 'Lacoste', 'Fossil'} value = set1.discard('Lacoste') print('Set after discard(): ', set1) Output: Set after discard(): {'Fossil', 'Zara'}</pre>
remove()	It removes a specified element. If the element is not a member, it raises a KeyError.	<pre>set1 = {'Zara', 'Fossil'} removedValue = set1.remove('Lacoste') print('Set after remove(): ', set1) Output: KeyError: 'Lacoste'</pre>

Table 3.7: Set methods with examples

Similar to other data structures, Set also has functions such as **len()**, **min()**, **max()**, and **set()**. Readers can try implementing these functions themselves.

Python set operations

Mathematical set operations like union, intersection, subtraction, and symmetric difference can be performed using pre-defined set methods such as **union()**, **intersection()**, **difference()**, and so on.

Set union

The union operation on two sets **X** and **Y** includes all the elements of set **X** and **Y**. We use the **|** operator or the **union()** method to perform the set union operation. Refer to *Figure 3.4*:

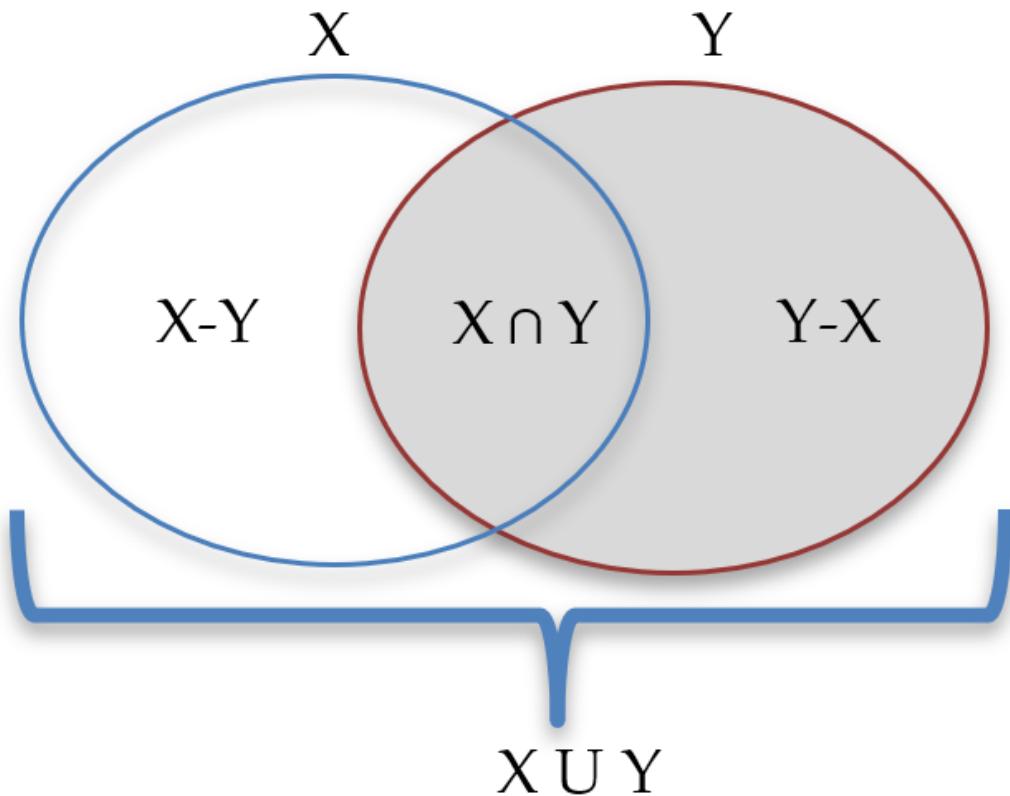


Figure 3.4: Venn diagram showing set operations for set X and Y

Set intersection

The intersection operation on two sets, **X** and **Y**, includes only the common elements between both **X** and **Y**. For this, we can use either the **&** operator or the **intersection()** method.

Set Difference

The difference operation ($X - Y$) on two sets, X and Y , includes the elements of set X that are not present on set Y . We use the “-” operator or the **difference()** method to perform set differences.

Set Symmetric Difference

The symmetric difference operation on two sets X and Y includes all elements of X and Y , excluding their common elements. For this, we can use either the \wedge operator or the **symmetric_difference()** method in Python. Following is a demonstration of various Set operations:

```
# Declare two sets
X = {11, 33, 55, 66, 77}
Y = {22, 33, 44, 66}
print('Union with |:', X | Y)
print('Union with union():', X.union(Y))
print('Intersection with &:', X & Y)
print('Intersection with intersection():',
X.intersection(Y))
print('Difference with - :', X - Y)
print('Difference with difference():', X.difference(Y))
print('Symmetric Difference with ^:', X ^ Y)
print('Now with symmetric_difference():',
X.symmetric_difference(Y))
```

Output:

The output can be seen in the following *Figure 3.5*:

```

Union with |: {33, 66, 11, 44, 77, 22, 55}
Union with union(): {33, 66, 11, 44, 77, 22, 55}
Intersection with &: {33, 66}
Intersection with intersection(): {33, 66}
Difference with - : {11, 77, 55}
Difference with difference(): {11, 77, 55}
Symmetric Difference with ^: {22, 55, 11, 44, 77}
Now with symmetric_difference(): {22, 55, 11, 44, 77}

```

Figure 3.5: Output

Dictionary

Python dictionary is another popular data structure representing an unordered collection of elements in the form of **key: value** pairs enclosed in curly braces. Keys are the unique elements within a dictionary while values may repeat. The values in a dictionary can take any datatype, but keys must be immutable types such as strings, numbers, or tuples. Various methods and functions are available for dictionary manipulation, as explained in [Table 3.8](#).

Let us define a dictionary say, `dict1 = {'s_id':101, 's_name':'Charlie', 's_age': 20}` for a student object:

Dictionary method	Description	Example
<code>get()</code>	This method returns the value corresponding to the given key.	<code>print(dict1.get('s_age'))</code> Output: 20
<code>items()</code>	This method returns dictionary elements in the form of a list of tuples for each key-value pair.	<code>print(dict1.items())</code> Output: <code>dict_items([('s_id', 101), ('s_name', 'Charlie'), ('s_age', 20)])</code>
<code>keys()</code>	This method returns a list of all keys in the dictionary.	<code>print(dict1.keys())</code> Output: <code>dict_keys(['s_id', 's_name', 's_age'])</code>

Dictionary method	This Description method removes the key: value pair from the dictionary with the key specified as the method argument.	<pre>print(dict1.pop('s_age'))</pre> <p>Example</p> <pre>Output: 20</pre> <pre>{'s_id': 101, 's_name': 'Charlie'}</pre>
update()	This method updates the dictionary with the specified key: value pair. In case the pair is already present, then it gets updated. Otherwise, a new pair is added to the dictionary.	<pre>dict1.update({'s_degree': 'B.Tech'})</pre> <pre>print(dict1)</pre> <p>Output:</p> <pre>{'s_id': 101, 's_name': 'Charlie', 's_age': 20, 's_degree': 'B.Tech'}</pre>

values()	This method returns a list of all values in the dictionary.	<pre>print(dict1.values())</pre> <p>Output:</p> <pre>dict_values([101, 'Charlie', 20])</pre>
-----------------	---	--

Table 3.8: Dictionary methods with examples

Furthermore, readers can try to implement functions such as **min()**, **max()**, **len()**, and **dict()** with dictionary elements and analyze the results obtained.

User-defined data structures

User-defined data structures are those data structures that are not directly supported or defined for use in Python but can be implemented by a user with the help of built-in data structures. There are many data structures that can be implemented this way, and some of them are as follows:

- Linked list

- Stack
- Queue
- Tree
- Graph, and so on.

The scope of this book is limited to the discussion of Linked List, Stack, and Queue only. Now, we shall see these user-defined data structures in detail.

Linked list

A linked list is a linear data structure that is made up of individual nodes that are stored separately and not at contiguous memory locations. Each node element comprises two components, namely, **Data** and **Next**.

Here, **Data** refers to the stored data value, whereas the **Next** component refers to the address of the next connected node element. The nodes in a linked list are linked using addresses, as shown in *Figure 3.6*. The last node, **D4**, connects to the **Null** value because no further connecting node is present thereafter:

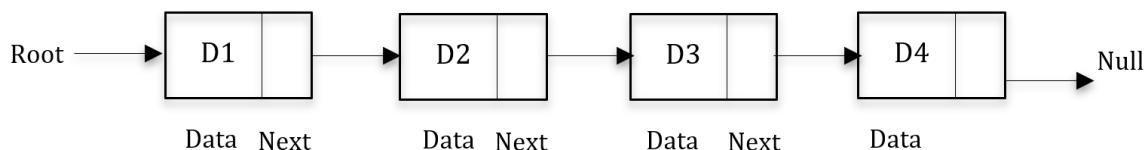


Figure 3.6: Linked list nodes demonstration

Let us demonstrate the working of a linked list using list data structure with the following example:

```

list1 = ['one', 'two', 'three']
print(list1)

list1.append('four') # Adding elements at the end of
# Linked list

list1.append('five')

list1.insert(0, 'zero') # Inserting new element in the
# beginning

print(list1)

```

```
list1.remove('three') # Remove an element from Linked List
print(list1)
```

Output:

The output can be seen in *Figure 3.7*:

```
['one', 'two', 'three']
['zero', 'one', 'two', 'three', 'four', 'five']
['zero', 'one', 'two', 'four', 'five']
```

Figure 3.7: Output

Stack

A stack is another linear structure in which data is inserted and removed from the same end, thereby following the **Last-In-First-Out (LIFO)** approach. Some examples are a stack of books or a stack of plates where the last kept plate is picked up first for use. Insertion in the stack is known as **push()** and deletion as **pop()** operation.

In *Figure 3.8*, we can see a simple stack with elements being inserted and popped out from the same end known as **Top**. If all values get deleted from the stack, the condition is known as **Underflow**, and if we try to add more elements to Stack even if its length is full, then it is called **Overflow** condition.



Figure 3.8: Demonstration of stack push and pop operations

Let us demonstrate the working of Stack using a list data structure with the following example:

```

stack1 = ['one', 'two', 'three']
print(stack1)

top = len(stack1) - 1 # Top position is the last index
# of stack list

print("Current Top is : ", top)

stack1.append('four') # push element to top position

stack1.append('five')

print(stack1)

top = len(stack1) - 1 # Find updated Top position

print("Updated Top is : ", top)

stack1.pop() # Pop an element from stack

print(stack1)

top = len(stack1) - 1 # Find updated Top position

print("Updated Top is : ", top)

```

Output:

The output can be seen in [Figure 3.9](#):

```

['one', 'two', 'three']
Current Top is : 2
['one', 'two', 'three', 'four', 'five']
Updated Top is : 4
['one', 'two', 'three', 'four']
Updated Top is : 3

```

Figure 3.9: Output

Queue

A queue is also a linear structure similar to stack but allows insertion of elements from one end and deletion from the other end. For example, a queue of people

standing to book movie tickets. A queue data structure follows **First-In-First-Out (FIFO)**. The end that allows deletion is known as the front of the queue, and the other end that allows insertion is known as the rear end. If all values get deleted from the queue, the condition is known as underflow, and adding more elements even if the queue is full is known as the **Overflow condition**. Refer to [Figure 3.10](#):

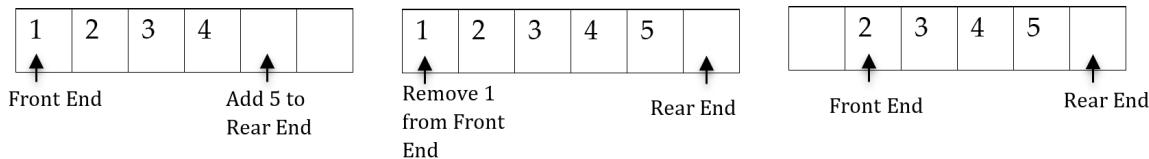


Figure 3.10: Demonstration of queue insertion and deletion

Now, let us see a demonstration of Queue Insertion and Deletion in Python:

```
queue1 = ['one', 'two', 'three']

print(queue1)

queue1.append('four') # Pushing new elements in rear
end of Queue

queue1.append('five')

print(queue1)

print("First element : ", queue1[0]) # Print head
element of Queue

size = len(queue1) # Print the last or tail element

print("Last element : ", queue1[size - 1])

queue1.remove(queue1[0]) # Popping an element from
front queue

print(queue1)
```

Output:

The output can be seen in [Figure 3.11](#):

```

['one', 'two', 'three']
['one', 'two', 'three', 'four', 'five']
First element : one
Last element : five
['two', 'three', 'four', 'five']

```

Figure 3.11: Output

Sorting algorithms

Sorting is the act of placing data in a specific direction, either in ascending or descending order. It represents data in a more understandable way. It is important to choose efficient sorting methods. Otherwise, it may take a significant number of computational resources to sort huge-size data.

Time complexity and space complexity

In order to solve a problem, there may be many different approaches available. Therefore, we must compare the performance of these algorithmic solutions in order to choose the best one to solve the problem. While analyzing an algorithm, we consider its time and space complexity:

- **Time complexity** is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. To compare time complexity, we mostly consider O-notation because it describes the execution time in the worst case.
- **Space complexity** is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

Bubble sort

Bubble sort, also popularly called sinking sort, is a popular sorting method that repeatedly steps through the list to sort it. In this algorithm, adjacent elements are compared and swapped if they are not in the correct order. It is named so because in its working, the smaller or larger elements *bubble* to the top of the list.

Let a given list of numbers [30, 9, 6, 3, 2] has to be sorted in ascending order using Bubble sort. Various passes of the bubble sort process are illustrated in [Figure 3.12](#):

Pass 1: Compare adjacent elements					Pass 2: Compare adjacent elements				
Step1	30	9	6	3	2	9	6	3	2
Step2	9	30	6	3	2	6	9	3	2
Step3	9	6	30	3	2	6	3	9	2
Step4	9	6	3	30	2	6	3	2	30
End	9	6	3	2	30	6	3	2	9

Pass 3: Compare adjacent elements					Pass 4: Compare adjacent elements				
Step1	6	3	2	9	30	3	2	6	9
Step2	3	6	2	9	30	6	9	30	30
End	3	2	6	9	30	2	3	6	9

Figure 3.12: Demonstration of Bubble Sort and its passes

Here, in the preceding example, there are 4 passes and 10 comparisons in all. Thus, we may conclude that bubble sort with n elements requires $n-1$ number of passes and in total $n(n-1)/2$ number of comparisons. Let us demonstrate the code for Bubble sort as follows:

```
num_list = [30, 9, 6, 3, 2]
n = len(num_list)
for i in range(n-1):
    for j in range(n-1-i):
        if num_list[j]>num_list[j+1]:
            num_list[j], num_list[j+1] = num_list[j+1], num_list[j]
print('Array after sorting')
print(num_list)
```

Output:

The output can be seen in [Figure 3.13](#):

```
Array after sorting
[2, 3, 6, 9, 30]
```

Figure 3.13: Output

Selection sort

The selection sort algorithm starts by finding the smallest number in the list and exchanging it with the first number. The next smallest number is found and exchanged with the second number, and so on. The use of the selection sort to order the five elements of the array is described in [Figure 3.14](#). The smallest element of each pass is written on a grey background, and the smallest element is in a circle:

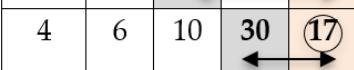
Pass 1:		Smallest element is 4 and is swapped with first element 30
Pass 2:		Next smallest element is 6 and is swapped with second element 10
Pass 3:		Next smallest element is 10 and is swapped with third element 17
Pass 4:		Next smallest element is 17 and is swapped with fourth element 30
		End of Passes with completely sorted list

Figure 3.14: Demonstration of selection sort and its passes

If there are n elements to be sorted, then the process mentioned previously should be repeated $n-1$ times (that is, $n-1$ passes) to get the required result. The code for selection sort is demonstrated as follows:

```
num_list = [30, 10, 17, 4, 6]
n = len(num_list)
for i in range(n-1):
    smallest = i
    for j in range(i+1, n):
        if (num_list[j] < num_list[smallest]):
            smallest = j
    num_list[i], num_list[smallest] =
    num_list[smallest], num_list[i]
print('Array after sorting')
```

```
print(num_list)
```

Output:

The output can be seen in *Figure 3.15*:

```
Array after sorting
[4, 6, 10, 17, 30]
```

Figure 3.15: Output

Insertion sort

Insertion sort is a simple sorting algorithm that picks up one value at a time to place it at its correct position in the unsorted list. When playing cards, people often use the insertion sort methodology to sort cards. Let us assume a list of numbers [30, 10, 17, 4, 6] to demonstrate the steps of Insertion sort. The algorithm starts by picking an element and comparing it with elements present before it in the list, with elements being swapped until they are arranged in the correct order.

Refer to *Figure 3.16*:

<i>Pass 1:</i>	<table border="1"><tr><td>30</td><td>10</td><td>17</td><td>4</td><td>6</td></tr></table>	30	10	17	4	6	Compare 10 with all elements before it i.e., with 30. Since $10 < 30$ so swap them.
30	10	17	4	6			
<i>Pass 2:</i>	<table border="1"><tr><td>10</td><td>30</td><td>17</td><td>4</td><td>6</td></tr></table>	10	30	17	4	6	Compare 17 with 30 and 10. Since $17 < 30$ so swap them but $17 > 10$ so skip.
10	30	17	4	6			
<i>Pass 3:</i>	<table border="1"><tr><td>10</td><td>17</td><td>30</td><td>4</td><td>6</td></tr></table>	10	17	30	4	6	Compare 4 with 30, 17 and 10. Since 4 is less than all of them so swap them to place 4 before all these elements.
10	17	30	4	6			
<i>Pass 4:</i>	<table border="1"><tr><td>4</td><td>10</td><td>17</td><td>30</td><td>6</td></tr></table>	4	10	17	30	6	Compare 6 with 30, 17 and 10 and 4. Since 6 is less than 30, 17 and 10 so swap with them but $6 > 4$ so skip.
4	10	17	30	6			
	<table border="1"><tr><td>4</td><td>6</td><td>10</td><td>17</td><td>30</td></tr></table>	4	6	10	17	30	End of Passes with completely sorted list
4	6	10	17	30			

Figure 3.16: Demonstration of insertion sort and its passes

Let us implement the logic of Insertion sort in the following Python code:

```
num_list = [30,10,17,4,6]
n = len(num_list)
for i in range(1,n):
    temp = num_list[i]
    j = i - 1
```

```

while(j >= 0 and temp < num_list[j]):
    num_list[j + 1] = num_list[j]
    j -= 1
    num_list[j + 1] = temp

print('Array after sorting')
print(num_list)

```

Output:

The output can be seen in *Figure 3.17*:

```

Array after sorting
[4, 6, 10, 17, 30]

```

Figure 3.17: Output

Table 3.9 gives the contrasting features of Bubble, Insertion, and Selection sort. It summarizes Time complexity (Best Case, Average Case, and worst case) and Space complexity:

Type of sorting	Work procedure	When to use	Best case	Avg. case	Worst case	Space comp.
Bubble sort	Compares adjacent element pairs and swaps if they are not ascending.	Bubble sort is fastest on an extremely small or nearly sorted set of data.	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$

Type of sorting	Work procedure	When to use	Best case	Avg. case	Worst case	Space comp.
Selection sort	This sorting technique repeatedly finds the minimum element and exchanges it with the leftmost unsorted element.	Selection Sort, on the other hand, does not work well with already sorted data because it will keep iterating through over and over, even though it may already be sorted.	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there.	It works well with nearly or fully sorted data. It iterates through the data, checking and comparing each item, and sorts them as needed. This is ideal when we have frequent incoming data that is being added to the collection, so it needs to be continuously sorted. It can figure out where to place it with a single pass.	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$

Table 3.9: Features of bubble, selection, and insertion sort

Note: Space Complexity of all sorting algorithms is the same, that is, $O(1)$. It is because we are not actually creating new arrays, and all the swapping occurs in the same place. We are not creating a new variable for each element.

Searching algorithms

Searching for data stored in different data structures is a crucial part of every single application. There are many different algorithms available to use when searching, and each has different implementations and relies on different data structures to get the job done. In Python, the easiest way to search for an object is to use Membership Operators in and not in to search for elements in Strings, Lists, Set, Dictionary, and Tuples data structures. Apart from these, we can implement the following techniques for searching an element in Python.

Linear search

Linear search is one of the simplest searching algorithms and the easiest to understand. We can think of it as a more intense version of the membership in operator. Linear search consists of iterating over a sequence or a collection data structure and returning the index of the first occurrence of an element once it is found:

```
num_list = [11, 22, 44, 0, 11, 99, 55]
index = -1
length = len(num_list)
search = int(input("Enter a number to search : "))
# The range(start, end) function returns a sequence of
# numbers,
# starting from start value (0 by default) till end
# value.
for i in range(0, length):
    if num_list[i]==search:
        index = i
        break
if index==-1:
    print('Element not found')
else:
    print('Element found at index ', index)
```

Output:

The output can be seen in *Figure 3.18*:

```
Enter a number to search : 99
Element found at index 5
```

Figure 3.18: Output

Binary search

Binary search uses the divide and conquer methodology for sorting a list of elements. It requires the sequence of elements to be sorted before beginning the search process. Initially, we set up two variables, namely, low and high, in the list. Here, low refers to the starting index of the list (default, 0), and high is the last index of the list. The algorithm starts by comparing the search value to the element in the middle of the sequence. Based on this comparison, the algorithm proceeds with the following steps:

- If the value at the mid index is the element we are looking for (best case), then the result is the mid index, and the procedure ends.
- If the number to be searched is greater than the middle number, then we shift the search towards the right side of mid by setting low = mid + 1.
- If the number to be searched is less than the middle number, then we shift the search towards the left side of mid by setting high = mid - 1.

The implementation of binary search is as follows:

```
num_list = [0, 11, 22, 44, 55, 99]
search = int(input('Enter a number to search '))
low = mid = 0
high = len(num_list) - 1
index = -1
while low <= high:
    mid = (high + low) // 2      # mid is the middle
    index = -1
    if num_list[mid] == search:
        index = mid
        break
    elif num_list[mid] < search:
        low = mid + 1
    else:
        high = mid - 1
print('Element found at index', index)
```

```
if num_list[mid] < search:
    low = mid + 1
elif num_list[mid] > search:
    high = mid - 1
else:
    index = mid
    break
if index == mid:
    print("Number found at index ", index)
else:
    print("Number Not Found!!")
```

Output:

The output can be seen in *Figure 3.19*:

```
Enter a number to search 55
Number found at index 4
```

Figure 3.19: Output

Conclusion

In this chapter, we have come across various built-in and user-defined data structures along with their working examples. We also learned about different sorting and searching techniques that can be used for developing efficient applications in Python. This chapter will lay a firm base for real-time project development and solving complex programming problems. In the upcoming chapters, we will be applying the different data structures and their methods, as learned in this chapter, more frequently.

Points to remember

- Python data structures can be broadly classified into two groups, namely, built-in and user-defined data structures.
- Built-in data structures include strings, list, tuple, set, and dictionary.
- User-defined data structures include linked list, stack, and queue.
- Sorting algorithms such as Bubble Sort, Insertion sort, and Selection Sort enable users to arrange the data in ascending or descending order.
- Searching techniques such as Linear Search and Binary Search enable searching an element in a sequence of given items.

Exercise

Attempt the following projects.

Sample project with solution

Create a personal telephone directory storing details such as names, numbers, alternate number, e-mail id, and so on. The contacts book enables searching through names or contact numbers. It also allows users to enter new contacts, delete a specific contact, or clear the directory completely:

```
# This project uses user-defined functions for
implementation.

# Readers can refer Chapter-4 to learn Function
Handling mechanism.

import sys # importing the module sys for using exit()
function

def initiate_contactlist():

    contacts = int(input("Please enter the number of
contacts: "))

    details = 5

    contact_book = []

    for i in range(contacts):
```

```
        print("\nEnter details for contact %d :" % (i + 1))
        print("Details marked(*) are mandatory!!")

print(".....")
.....
```

temp_list = []

for j in range(details):

if j == 0:

temp_list.append(str(input("Enter Contact Name* : ")))

if temp_list[j] == '' or temp_list[j] == ' ':

sys.exit("Cannot leave Contact Name blank!!")

if j == 1:

temp_list.append(int(input("Enter contact number* : ")))

if j == 2:

temp_list.append(input("Add alternate contact number: "))

if temp_list[j] == '' or temp_list[j] == ' ':

temp_list[j] = None

if j == 3:

temp_list.append(str(input("Enter e-mail address: ")))

if temp_list[j] == '' or temp_list[j] == ' ':

```
        temp_list[j] = None

    if j == 4:
        temp_list.append(str(input("Enter
category(Home/Family/Friends/Work/Others): ")))

        if temp_list[j] == "" or temp_list[j]
== ' ':
            temp_list[j] = None

    contact_book.append(temp_list)

print(contact_book)

return contact_book

def menu_options():

print("_____"
)

print("\t\t\tPHONE CONTACTS DIRECTORY")

print("_____"
)

print("\tPress desired option of Contacts Book\n")
print("1. Press 1 to Add new Contact")
print("2. Press 2 to Remove a Contact")
print("3. Press 3 to Clear Contacts Book")
print("4. Press 4 to search a Contact")
print("5. Press 5 to View Contacts")
print("6. Exit Contacts Book")
choice = int(input("Please enter your choice: "))
```

```
    return choice

def add_new_contact(obj):
    cont_list = []
    for i in range(len(obj[0])):
        if i == 0:
            cont_list.append(str(input("Enter contact
name: ")))
        if i == 1:
            cont_list.append(int(input("Enter contact
number: ")))
        if i == 2:
            cont_list.append(input("Enter alternate
contact number: "))
        if i == 3:
            cont_list.append(str(input("Enter e-mail
address: ")))
        if i == 4:
            cont_list.append(
                str(input("Enter
category(Home/Family/Friends/Work/others): ")))
    if not obj: # check if contact book object obj is
empty
        obj = cont_list
    else:
        obj.append(cont_list)
    return obj
```

```
def delete_contacts(obj):
    if not obj:
        print("Contacts Book is empty!!")
    else:
        cont_name = str(input("Enter that contact name
to delete: "))
        temp = 0
        for i in range(len(obj)):
            if cont_name == obj[i][0]:
                temp += 1
                print(obj.pop(i))
                print("This contact is removed")
        return obj
    if temp == 0:
        print("Contact does not exist!!")
        return obj

def delete_all(obj):
    if not obj: # check if contact book object obj is
empty
        print("Contacts Book is empty!!")
    else:
        return obj.clear()

def search_contact(obj):
    if not obj: # check if contact book object obj is
empty
```

```
    print("Contacts Book is empty!!")

else:
    choice = int(input("For searching contact
press\n1. Name\n2. Number\n"))

    temp = []
    flag = -1 # flag represents search result
    if choice == 1: # For search using Contact
Name
        c_name = str(input("Please enter the name
of the contact you wish to search: "))

        for i in range(len(obj)):
            if c_name == obj[i][0]:
                flag = i
                temp.append(obj[i])

    elif choice == 2: # For search using Contact
Number
        c_number = int(input("Please enter the
number of the contact you wish to search: "))

        for i in range(len(obj)):
            if c_number == obj[i][1]:
                flag = i
                temp.append(obj[i])

    else:
        print("Invalid search criteria")
        return -1

    if flag == -1:
```

```
        return -1
    else:
        display_contacts(temp)
        return flag

def display_contacts(obj):
    if not obj:
        print("Contacts Book is empty!!")
    else:
        for i in range(len(obj)):
            print(obj[i])

def exit_book():
    sys.exit()

# Main Code to start Contacts Book

print("*****")
print("Welcome to Personal Contacts Book")
print("*****")
option = 1
obj = initiate_contactlist()
while True:
    option = menu_options()
    if option == 1:
```

```

        if not obj: # check if contact book object is
empty

            obj = initiate_contactlist()

        else:

            obj = add_new_contact(obj)

    elif option == 2:

        obj = delete_contacts(obj)

    elif option == 3:

        obj = delete_all(obj)

    elif option == 4:

        result = search_contact(obj)

        if result == -1:

            print("The contact does not exist!! Search
again")

    elif option == 5:

        display_contacts(obj)

    elif option == 6:

        exit_book()

    else:

        print("Enter a valid choice!!")

```

Output:

The output can be seen in *Figure 3.20*:

```
*****
* Welcome to Personal Contacts Book
*****8****

Please enter the number of contacts: 2

Enter details for contact 1 :
Details marked(*) are mandatory!!
.....
Enter Contact Name*: Sameer
Enter contact number*: 3344567
Add alternate contact number:
Enter e-mail address:
Enter category(Home/Family/Friends/Work/Others):

Enter details for contact 2 :
Details marked(*) are mandatory!!
.....
Enter Contact Name*: Nidhi
Enter contact number*: 5678456
Add alternate contact number:
Enter e-mail address:
Enter category(Home/Family/Friends/Work/Others):
[['Sameer', 3344567, None, None, None], ['Nidhi', 5678456, None, None, None]]

-----
PHONE CONTACTS DIRECTORY
-----
Press desired option of Contacts Book

1. Press 1 to Add new Contact
2. Press 2 to Remove a Contact
3. Press 3 to Clear Contacts Book
4. Press 4 to search a Contact
5. Press 5 to View Contacts
6. Exit Contacts Book
Please enter your choice: 1
Enter contact name: Guest
Enter contact number: 6372823
Enter alternate contact number:
Enter e-mail address:
Enter category(Home/Family/Friends/Work/others):

-----
PHONE CONTACTS DIRECTORY
-----
Press desired option of Contacts Book

1. Press 1 to Add new Contact
2. Press 2 to Remove a Contact
3. Press 3 to Clear Contacts Book
4. Press 4 to search a Contact
5. Press 5 to View Contacts
6. Exit Contacts Book
Please enter your choice: 5
['Sameer', 3344567, None, None, None]
```

Figure 3.20: Output

Practice project

Create a Dice-Roller Game for two players who alternately roll the dice. The one who scores the most wins the game. Restrict the total attempts of each player to a particular number. The initial score of both players is zero (0). Each player will throw the dice once, and the output (a number in the range 1–6) will be added to the current score of the player. If in the first attempt of dice throw, the outcome is number “6” then the user gets an additional attempt, and both the outcomes are added together (For example, the first attempt output is 6, and the additional attempt output is 2. Then the combined score would be $6+2 = 8$). On the contrary, if, in the first attempt, a player gets a number less than “6” then his score is updated, and the other player gets a chance to roll the dice.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 4

Functions in Python

Introduction

In the previous chapters, we discussed different Python data structures and their associated methods and functions. This chapter extends the discussion by introducing users to the concept of modularity by using different types of functions, such as built-in functions, user-defined functions, and anonymous functions, with different types of arguments used for creating more dynamic and interactive applications. The concepts of Recursion, variable scope and lifetime will also be covered in depth. Apart from this, users will also get a firm understanding of using and creating different types of modules and packages in Python applications.

Structure

In this chapter, we will discuss the following topics:

- Introduction to functions
- Function arguments
- Recursion in Python
- Anonymous functions
- Scope and lifetime of variables
- Modules and packages

Objectives

By the end of this chapter, the readers will be able to create more modular applications by using different types of functions and dividing the entire application across multiple modules, performing some logical tasks. This chapter will help users understand how complex and big applications can be created and maintained across modules.

Introduction to functions

A segment of code that accomplishes a particular task is known as a Function. Creating functions enables developers to create modular applications, achieving a high degree of code reusability. This means that we may call a function as many times as we require and reuse its code with a different set of arguments rather than creating the same piece of code repeatedly for different input variables. This not only saves time and effort but also reduces the overall **line of code (LOC)** in an application.

Benefits of using functions

There are numerous advantages of including functions in an application, such as:

- We can avoid duplicating the same code block in a program by using functions.
- Once defined, Python functions can be called repeatedly from any location inside a program.
- If a Python program grows in size, it can be divided into numerous logically manageable functions for better handling.
- The most notable advantage of using Python functions is that they can be called with different sets of arguments to return desirable output each time.

Functions versus methods

Although a function and method seem to be the same in their work, there lies a slight difference between them. A method may be defined as a function which is part of a certain class and, therefore, can only be accessed by using

an instance or object of that class. On the other hand, a function refers to a standalone function that is not restricted to access by an object only. Let us consider the following example to understand the difference of function and method:

```
# Consider a string object
str1 = "Welcome to Python Lecture"
# count() is a method called using string object
str1
frequency = str1.count('e')
# len() is a function. It is not called with the
# string object str1
# Rather str1 object passed as a parameter to len()
# function
length = len(str1)
```

In this example, we understand that **count()** is used as a method called with a string object, whereas **len()** is used as a function called directly without using a string object. Thus, we may assume that all methods are functions while all functions are not methods.

Types of Python function

There are three categories of functions in Python programming language, defined as follows:

- **Pre-defined functions:** As the name suggests, these are already defined built-in standard library functions that are ready for immediate use. The standard I/O functions, such as **print()** and **input()**, along with different functions related to Python data types, such as **len()** and **count()**, fall under the category of Pre-defined functions. Readers may refer to [*Chapter 3, Data Structures and Algorithms*](#), to go through the different pre-defined functions for each data type.

- **User-defined functions:** Sometimes, it is desirable to create new functions based on user requirements to perform a specific task. Such functions are known as User-defined functions.
- **Anonymous (or Lambda) Functions:** Another category of functions known as anonymous functions is also used in Python. These types of functions are anonymous since they do not have a name associated with them and are not declared with the standard **def** keyword. Rather, they are defined using the **lambda** keyword, and therefore, these are also called as lambda functions.

Function declaration and calling

The following syntax can be used to declare a user-defined function:

```
def function_name(arguments):
    # function body
return
```

Here, let us see what different elements in the preceding syntax mean:

- **def:** The **def** keyword marks the beginning of a function declaration.
- **function_name:** It refers to any logical name given to the function by the user.
- **arguments:** This represents the set of values passed to the function as parameters.
- **return (optional):** The **return** statement returns a value and control from the function to the target location where the function is being called. If used without a value, then the return statement simply returns control to the target location.

In order to use the function at any point in the program, we need to call the function by its name, followed by **parenthesis()**. Let us see an example of declaring a function and calling it for use:

```
# Declare a function to print welcome message
```

```
def welcomeMessage(name):  
    username = name  
    if username:  
        print("Welcome " + str(username) + "!!")  
    else:  
        print("Welcome Guest!!")  
    return  
  
# Calling welcomeMessage() function to execute  
# Passing name variable as argument in function  
name = input("Enter your name: ")  
welcomeMessage(name)
```

Output:

Refer to *Figure 4.1*:

```
Enter your name: Admin  
Welcome Admin!!
```

Figure 4.1: Output

In the preceding example, we have created a function named **welcomeMessage()** to print a message for a user with a name. Here, when the function is called in line 12, the control of the program goes to the function definition at line 2 and starts executing the function body from lines 3 to 8. Once all the statements inside the function are executed, the program control jumps to the next statement after the function call.

Function arguments

As seen in the syntax of function declaration, the function name is followed by several comma-separated values in parenthesis. These values accepted by the function are known as Arguments. More often, the words arguments and parameters are used interchangeably, but they have a bit of conceptual difference between them. Parameters are the names used in the function declaration, and arguments will be mapped to these defined parameters for use, whereas arguments are the actual values provided to the function at the time of calling it.

Types of function arguments

There are four different types of arguments that may be used for calling a function in Python. Let us discuss each in detail with examples.

Default arguments

If no value is provided for the argument when the function is called, a default argument is a type of parameter that accepts a default value as input. It is important to note that the non-default parameter always follows the default parameter. Following is an example demonstrating the usage of default arguments:

```
def enterDetails(age, username = 'Guest'):  
    print("Your entered details are : ")  
    print("User Name is: ", username)  
    print("User Age is: ", age)  
  
# Calling the function and passing only one argument  
  
enterDetails(25) # Default value of username used  
  
# Calling the function and passing both the arguments  
  
enterDetails(30, "Michael")
```

Output:

Refer to *Figure 4.2*:

```
        Your entered details are :  
        User Name is: Guest  
        User Age is: 25  
        Your entered details are :  
        User Name is: Michael  
        User Age is: 30
```

Figure 4.2: Output

Keyword arguments

Keyword arguments, also known as Named arguments, are those values that, when passed into a function, are recognized using specific parameter names. A keyword argument contains a parameter name, an assignment operator “=” and a value. We demonstrate the working of Keyword arguments in the following example:

```
def enterDetails(age, username):  
    print("Your entered details are : ")  
    print("User Name is: ", username)  
    print("User Age is: ", age)  
# Calling the function with keyword arguments  
enterDetails(username= "Admin", age=25)
```

Output:

Refer to *Figure 4.3*:

```
        Your entered details are :  
        User Name is: Admin  
        User Age is: 25
```

Figure 4.3: Output

Required arguments

Required arguments are those arguments that are given to a function in a pre-defined ordered sequence while calling. It is important to note that the count of the required arguments in the method call must match the count of arguments provided while defining the function:

```
def enterDetails(age, username):  
    print("Your entered details are : ")  
    print("User Name is: ", username)  
    print("User Age is: ", age)  
  
# Calling the function with required arguments  
enterDetails(25, "Admin")  
enterDetails("Admin")
```

Output:

Refer to *Figure 4.4*:

```
Traceback (most recent call last):  
  Your entered details are :  
  User Name is: Admin  
  User Age is: 25  
  File "C:\Users\Nidhi\PycharmProjects\MyProject2\PythonDemo.py", line 7, in <module>  
    enterDetails("Admin")  
TypeError: enterDetails() missing 1 required positional argument: 'username'
```

Figure 4.4: Output

Here, we can consider that when the function is called with both arguments, as in line 6, then the function works fine. On the other hand, `TypeError` is generated while executing line 7 where only one argument is passed in the function call.

Variable-length arguments

In some cases, it may be necessary to call a function with a different number of parameters at different times. In order to call a function with as many arguments as we want, we can use special characters such as ***args** and ****kwargs**. Here, ***args** is used for **Non-Keyword Arguments**, whereas ****kwargs** is used to call functions with **Keyword Arguments**. Consider the following example, demonstrating the use of variable-length arguments:

```
def func1(*args): # Here args can take any number
                  # of parameters

    list1 = []
    for i in args:
        list1.append(i.upper())
    return list1

output = func1() # Calling function with no
                 # arguments

print("The updated output of *args : ", output)

output = func1("Apple", "Mango") # Calling function
                                # with 2 arguments

print("The updated output of *args : ", output)

def func2(**kwargs): # Here kwargs can take any
                    # number of keyword parameters

    list1 = []
    for key, value in kwargs.items():
        list1.append([key, value])
    return list1
```

```
# Calling function with 'n' number of keyword arguments
output = func2(num1=10, num2=20, num3=30, num4=40,
num5=50)
print("The updated output of **kwargs: ", output)
```

Output:

Refer to *Figure 4.5*:

```
The updated output of *args : []
The updated output of *args : ['APPLE', 'MANGO']
The updated output of **kwargs: [['num1', 10], ['num2', 20], ['num3', 30], ['num4', 40], ['num5', 50]]
```

Figure 4.5: Output

Recursion in Python

We understand a function in Python can call other functions to execute their functionality. However, in certain cases, it may be required for a function to call itself. Such types of constructs are known as Recursive functions. The process of recursion is a frequent idea in maths and programming. It denotes that a function makes a repeated call to itself. The developer should exercise extreme caution when using recursion because it may result in writing a function that never ends or consumes excessive amounts of memory or processing resources. *Figure 4.6* gives a fair idea of how recursion works:

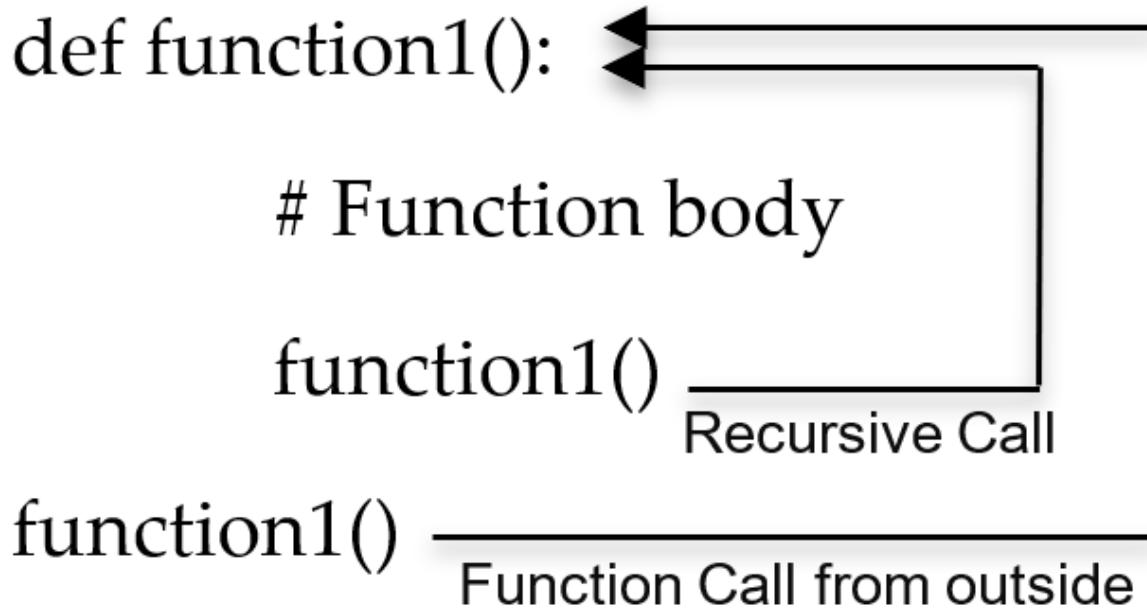


Figure 4.6: How recursion works

Here, we can see that **function1()** is called from outside once, and then during the execution of the function body, a call to itself is made. This self-calling process will continue till some limit is applied by the user programmatically. Let us consider an example to implement recursion while calculating the factorial of a number. The factorial of a number **n** is found by multiplying the number by itself and **n-1** until the calculation reaches 1. For example: Factorial of 5 = $5*4*3*2*1 = 120$:

```

def func_fact(num):
    if num == 0:
        return 1
    else:
        return num * func_fact(num - 1)  #
Recursive call

number = int(input("Enter a number to find
factorial : "))

```

```
print("The factorial of {0} is: ".format(number),  
func_fact(number))
```

Output:

Refer to *Figure 4.7*:

```
Enter a number to find factorial : 5  
The factorial of number 5 is: 120
```

Figure 4.7: Output

Anonymous functions

A lambda function in Python is a distinct category of functions in Python that does not have a function name with it. To build a lambda function, we use the **lambda** keyword instead of the **def** keyword, as used in the normal function declaration. It is important to note that a lambda function can have only one expression, although it can have any number of arguments. The syntax used to create a lambda function is as follows:

```
variable = lambda argument(s): Expression
```

Here, in the syntax, **argument(s)** refers to any value(s) passed to the lambda function. The **Expression** is the calculation executed, and its output is returned. Moreover, we have defined a variable and assigned the lambda function to it. This variable is used to call and execute the respective lambda function. Let us see a few demonstrations of anonymous functions.

Example 1: use of lambda function to find the maximum of two numbers

```
find_max = lambda x, y: x if (x > y) else y  
print(find_max(51, 71))
```

Output:

Example 2: To filter out only even numbers from a list of numbers

Here, we can use the **filter()** function along with the lambda function. The **filter()** function accepts two arguments: a function and an iterable, such as lists, tuples, or strings. The two function works upon all the list items and returns a new list containing only the filtered elements returned by the lambda function:

```
origin_list = [11, 15, 14, 16, 18, 111, 13, 112, 55]  
even_list = list(filter(lambda num: (num%2 == 0), origin_list))  
print(even_list)
```

Output:

```
[14, 16, 18, 112]
```

Example 3: To find the cube of all elements in a list

Here, we can make use of **map()** function. Similar to the **filter()** function, the **map()** function also accepts two arguments: a function and an iterable such as lists, tuples, or strings. The **map()** function is called with a lambda function and a list of items. Unlike **filter()**, the **map()** returns a new list with updated elements returned by the lambda function after working on each element.

```
list1 = [1,2,3,4,5,6,7]  
cube_list = list(map(lambda num: num**3, list1))  
print(cube_list)
```

Output:

```
[1, 8, 27, 64, 125, 216, 343]
```

Scope and lifetime of variables

The scope of a variable is defined as the region where this variable can be accessed, whereas the lifetime of a variable is the interval for which a variable resides in the memory for use. The lifespan and accessibility of a variable depend upon its declaration in the program. For example, the scope and lifetime of a variable **var** declared inside a function **func()** is till the function exits and is used. Python variables can be categorized into the following three types based on their scope:

- **Local variables:** The variables declared for use inside a function have a local scope within the function. Such variables are inaccessible outside the function. These are known as local variables due to their local scope.
- **Global variables:** A global variable is one that is declared outside of the function or in the global scope of the Python program. This indicates that a global variable may be accessed for use from within or outside the function. However, by simply declaring a variable as global, we can only access that variable but cannot modify it from inside any function. The solution for this is to declare the variable by using a global keyword. Please note that we use the global keyword to read and write a global variable from inside a function. On the other hand, using the **global** keyword outside a function has no effect.
- **Nonlocal variables:** A function defined inside another function is called a nested function, and the enclosing function is called an outer function. In these nested functions, we may use variables whose local scope is not defined. These variables can neither be exclusively in the local nor in the global scope. Therefore, these variables are known as nonlocal variables, which are created using **nonlocal** keywords. Let us demonstrate all the variable scopes with the following example:

```
def outer_func(): # Outer Function
    lvar = "local variable of outer_func()"
    global gvar
    gvar = "Global variable"
    print("Inside outer_func() lvar: ", lvar)
```

```

def inner_func(): # Nested Function
    mvar = "Local variable of inner_func()"
    nonlocal lvar # Declare mvar as nonlocal
variable
        lvar = "NonLocal variable of inner_func()"
# Change lvar value
        print("Inside inner_func() lvar : ", lvar)
        print("Inside inner_func() gvar : ", gvar)
        print("Inside inner_func() mvar : ", mvar)
    inner_func() # Call inner_func()
    print("Inside outer_func() mvar: ", lvar)
    print("Inside outer_func() gvar: ", gvar)
    # print("Inside outer_func() mvar: ", mvar) # mvar inaccessible
outer_func()
print("Outside outer_func() gvar: ", gvar)
# print("Outside outer_func() mvar: ", lvar) # Error lvar inaccessible
# print("Outside outer_func() mvar: ", mvar) # Error mvar inaccessible

```

Output:

Refer to *Figure 4.8*:

```
Inside outer_func() lvar: local variable of outer_func()
Inside inner_func() lvar : Non Local variable of inner_func()
Inside inner_func() gvar : Global variable
Inside inner_func() mvar : Local variable of inner_func()
Inside outer_func() mvar: Non Local variable of inner_func()
Inside outer_func() gvar: Global variable
Outside outer_func() gvar: Global variable
```

Figure 4.8: Output

Modules and packages

Our Python program may have numerous lines of code as it expands. We can use modules to partition codes into several files according to their functionality rather than placing everything in a single file. Our code is better organized and simpler to maintain as a result. A *module* is a file that has code in it to carry out a certain function. A module may include classes, functions, and different variables. The Python standard library already has several pre-defined modules that can be used as per requirements. However, it is also possible to create user-defined modules with a **.py** extension and import them when required. In order to use any module, we can use the **import** keyword with the name of the required module. Let us see an example to import a math module and use it.

AddNumbers.py

```
def add(num1, num2):
    print("Addition of numbers: ", num1 + num2)
```

Demo.py

```
import AddNumbers as an # user-defined module
import math as mt      # built-in module
n1 = 144
```

```
n2 = 20

print("The square root of {0} is : ".format(n1),
mt.sqrt(n1))

an.add(n1, n2)
```

Output after executing **Demo.py**:

Refer to *Figure 4.9*:

```
The square root of 144 is : 12.0
Addition of numbers: 164
```

Figure 4.9: Output

In this example, we first created a module called **AddNumbers.py** in which an **add()** function is defined. Next, another module, **Demo.py**, is created in which we import **AddNumbers.py** and a built-in module math as **mt** using the **import** keyword. The **as** keyword is used to create alias names that can be used instead of actual module names. Please note that creating an alias name is optional. In **Demo.py**, in line 5, we use the alias name “**mt**” to access the **sqrt()** function of the math module in order to calculate the square root of a number. Then, in line 6, we use alias **an** to access **add()** defined in the **AddNumbers.py** module.

Another important term used frequently in Python is called *Package*, as we know that a Python module may contain several classes, functions, variables, and so on. On the other hand, a Python package is like a folder containing several modules as files logically grouped together for an application. We will see a lot of examples of packages in the coming chapters.

Conclusion

This chapter deals with the concepts of functions and modules in Python. With the help of various examples, readers have well understood different types of functions, such as pre-defined functions, user-defined functions and anonymous (lambda) functions. The section on function arguments helped

readers to carefully analyze different types of arguments that can be used in functions for creating dynamic applications. The last section of this chapter revolves around creating and managing built-in and user-defined modules that can be used across applications.

Points to remember

- A segment of code that accomplishes a particular task is known as a Function, whereas a Method may be defined as a function, which is part of a certain class, and therefore, can only be accessed by using an instance or object of that class.
- There are three categories of functions supported in Python, namely, pre-defined Functions, User-defined Functions, and Anonymous (or Lambda) Functions.
- The four different types of arguments that may be used for calling a function are Default Arguments, Keyword Arguments, Required Arguments, and Variable-length Arguments.
- The process of recursion is a frequent idea in math and programming. It refers to a function making repeated calls to itself.
- The scope of a variable is defined as the region where this variable can be accessed, whereas the lifetime of a variable is the interval for which a variable resides in the memory for use.
- A module is a file that has code in it to carry out a certain function. A module may include classes, functions, and different variables.
- A Python package is like a folder containing several modules as files logically grouped together for an application.

Exercise

Attempt the following project.

Sample project with a solution

Create a simple application for the **Human Resource (HR)** Manager of a company to enter details of a new employee, edit details of employees, and delete certain details of resigned employees using functions.

This project is a completely function-oriented program with separate functions created for adding new employee details, displaying employee details with edit details and calculating net salary for employees. The program begins with calling the **employee()** function, which contains nested functions for various tasks. These nested functions are called as per user requirements. The global variable count tracks for number of employees added.

Note: We can improve this project and make it more interactive by using Object oriented approach.

```
count = 0 # global variable to contain count of
employees

def employee(): # outer function containing nested
function

    def enter_new(en, dept, desig, bs, age,
status):
        # nested function to add new employee detail

        e_name = en
        department = dept
        designation = desig
        basic_salary = bs
        eage = age
        current_status = status
        global count
        count = count + 1
```

```
        display(e_name, department, designation,
basic_salary, eage, current_status)

def display(en, dept, desig, bs, age, cs):
    # nested function to display employee detail
    global count
    print("Total employees : ", count)
    print("*****The details entered are
: *****")
    print("The name of employee: ", en)
    print("The department of employee: ", dept)
    print("The designation of employee: ",
desig)
    print("The basic salary of employee: ", bs)
    print("The age of employee: ", age)
    print("The current status of employee: ",
cs)
    if cs=="Resigned":
        print("This record of resigned employee
will be removed!!")
    else:
        calc_salary(bs)
        answer = input("Do you want to edit
details?")
        if answer=="yes":
            edit_details()
```

```
        else:
            pass

    def edit_details(): # nested function edit
        employee detail

            print("Enter new details for this
Employee")

            name = input("Enter Name : ")
            dept = input("Enter Department : ")
            desig = input("Enter Designation : ")
            bs = int(input("Enter Basic Salary : "))
            age = int(input("Enter Age : "))
            cs = input("Enter current Employee status :
")
            display(name, dept, desig, bs, age, cs)

    def calc_salary(salary): # nested function find
        net salary

        if (salary > 20000):
            tax = 0.15 * salary
            net_sal = salary - tax
            print("The net salary is", net_sal)

        else:
            net_sal = salary
            print("No Tax Deductions")
```

```
        print("The net salary is", net_sal)

    # call enter_new() inside employee() function to
    # add new employee

        enter_new("John", "Marketing", "Assistant",
10000, 20, "Present")

        enter_new("Simon", "Production", "Manager",
30000, 35, "Present")

        enter_new("Piya", "Project Management",
"Developer", 50000, 25, "Present")

        enter_new("Kylie", "Marketing", "Assistant",
20000, 20, "Resigned")

# Call employee() function to begin employee
management for HR

employee()
```

Output:

Refer to *Figure 4.10*:

```
Total employees : 1
*****The details entered are : *****
The name of employee: John
The department of employee: Marketing
The designation of employee: Assistant
The basic salary of employee: 10000
The age of employee: 20
The current status of employee: Present
No Tax Deductions
The net salary is 10000
Do you want to edit details?no
Total employees : 2
*****The details entered are : *****
The name of employee: Simon
The department of employee: Production
The designation of employee: Manager
The basic salary of employee: 30000
The age of employee: 35
The current status of employee: Present
The net salary is 25500.0
Do you want to edit details?yes
Enter new details for this Employee
Enter Name : Simon Paul
Enter Department : Production Unit
Enter Designation : Manager
Enter Basic Salary : 30000
Enter Age : 35
Enter current Employee status : Present
Total employees : 2
*****The details entered are : *****
The name of employee: Simon Paul
The department of employee: Production Unit
The designation of employee: Manager
The basic salary of employee: 30000
The age of employee: 35
The current status of employee: Present
The net salary is 25500.0
Do you want to edit details?no
Total employees : 3
*****The details entered are : *****
The name of employee: Piya
The department of employee: Project Management
The designation of employee: Developer
The basic salary of employee: 50000
The age of employee: 25
The current status of employee: Present
The net salary is 42500.0
Do you want to edit details?no
Total employees : 4
*****The details entered are : *****
The name of employee: Kylie
The department of employee: Marketing
The designation of employee: Assistant
The basic salary of employee: 20000
The age of employee: 20
The current status of employee: Resigned
This record of resigned employee will be removed!!
```

Figure 4.10: Output

Practice project

Create a currency converter to convert the currency of one country to another country on a real-time basis.

Hint: To create a currency converter application, we need to install a request module in Python using the command:

pip install requests

For practice purposes, readers must try to create a simple console application first for the currency converter project. Later, users can try to create an interactive GUI application for this project by using the Tkinter library or a mobile application using the Kivy library. Both Tkinter and Kivy libraries are discussed in detail in *Chapters 8 and 10*, respectively.

CHAPTER 5

Object-oriented Programming Concepts

Introduction

In the previous chapters, we have learned about various programming constructs that are available in Python. This chapter gives a deep insight into modeling real-time problems in the form of classes and objects using Object-oriented programming approaches. Various OOP techniques, such as Encapsulation, Data Hiding, Inheritance, and Polymorphism, are discussed with well-framed examples. Thus, this chapter will enable readers to understand the idea of implementing object-oriented programming in Python.

Structure

In this chapter, we will discuss the following topics:

- Introduction to programming paradigms
- Class and objects
- Constructors in Python
- Encapsulation and data hiding
- Inheritance in Python
- Polymorphism in Python

Objectives

By the end of this chapter, the readers will have a deeper understanding of real-time problem modeling in terms of classes and objects. Various OOP concepts, such as Inheritance and Polymorphism discussed in this chapter, will sharpen the programming skills of users.

Introduction to programming paradigms

Programming paradigms refer to the way of writing and organizing a program code. It represents a set of rules, conventional ideas, and procedures widely followed by developers to create more structured programming solutions. Each programming language depicts a unique programming style that implements a particular programming paradigm. Although there are many different types of programming paradigms available, the two most popularly used approaches are as follows:

- Procedural programming
- Object-oriented programming

Procedural programming

Imperative programming or procedural programming is a style of writing where statements are organized into several procedure calls (also known as subroutines or functions). Each procedure contains a stepwise list of instructions for accomplishing a specific task. Languages such as FORTRAN, ALGOL, COBOL, BASIC, Pascal, and C were primarily procedural. [Figure 5.1](#) features the procedure-oriented paradigm:

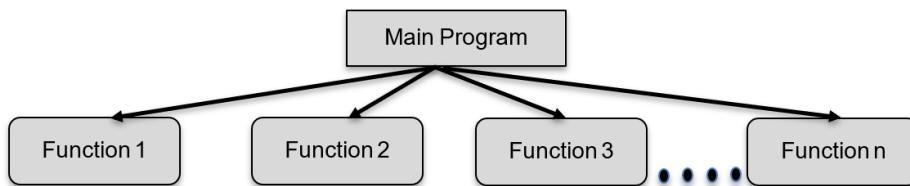


Figure 5.1: Procedure-oriented paradigm

The procedural-oriented programming lacks the feature of code reusability. It means that the same computer code must be written multiple times if required for repeated usage. It is a challenging task to model real-world objects using procedural programming.

Object-oriented programming

The programming paradigm known as **Object-oriented Programming (OOP)** is based on the ideas of classes and objects. It is used to organize a computer program into basic, reusable blueprints of code, often referred to as a class, which consists of data and methods to carry out some specific tasks. In order to use a class, it is required to create instances known as objects of classes. A few examples of popular object-oriented programming languages are C++, Java, C#, and Python. [Figure 5.2](#) features the object-oriented paradigm:

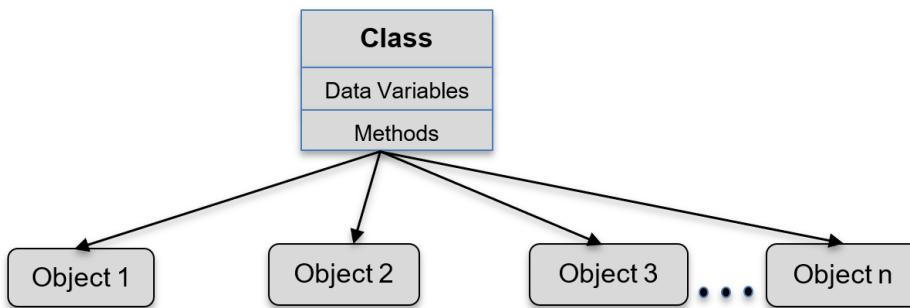


Figure 5.2: Object-oriented paradigm

Object-oriented programming concepts

The essence of the object-oriented programming paradigm consists of designing an application using classes and objects. Python has always been an object-oriented language that provides rich support for object-oriented programming concepts, including class, object, method, inheritance, polymorphism, data abstraction, and encapsulation.

Class and objects

In Python, everything can be considered as an object of a class. A **Class** serves as a user-defined blueprint or template from which objects can be created. The class offers a way to group together functionality and data in a single unit. It contains the state (attribute) and action (behavior) of the object. Here, the state represents data variables, and the methods represent actions that enable modifying the state. In order to use class, we are required to instantiate it by creating its instances called **Objects**. Thus, we may define an object as an entity with a state and actions. An object has access to the data and methods of its class. As there is no limit on the number of class objects, we may create any number of objects for a class. *Figure 5.3* shows modeling real-world problems into classes and objects:

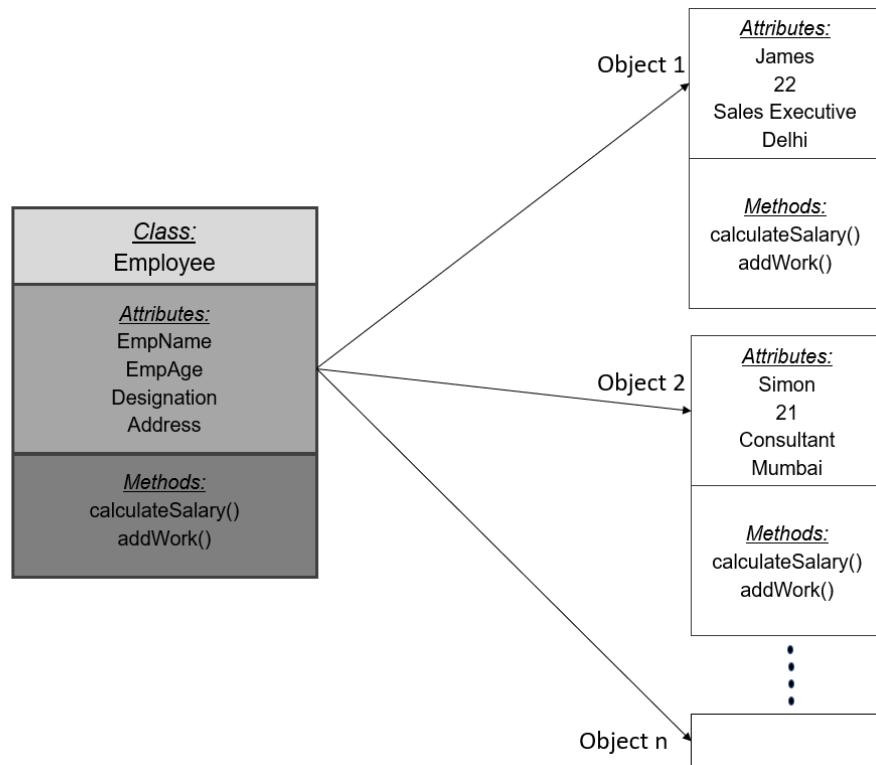


Figure 5.3: Modelling real-world problems into classes and objects

The following is the syntax for creating a new class:

```
class Class_Name:  
    '''Optional class documentation string'''  
    # class body or suite  
    objName = Class_Name()
```

Here, the keyword **class** followed by the class name is used to create a new class in Python. The **docstrings** are optional strings that serve as our code's description and can be accessed via

`ClassName.__doc__` attribute. The class suite refers to the set of attributes and methods in the class. A new object `objName` can be created by simply calling the class name with parenthesis.

Class attributes and methods

A class encapsulates or binds together the data members (attributes) and methods that use these data members together in one unit. In Python, there exist different types of attributes and methods in a class. The two types of data members or variables that can be used in a class are as follows:

- **Class variable:** A class variable is the one that is shared by all instances of a class. These variables are defined within a class body but are present outside any of the class's methods. Thus, these variables exist purely in class scope and can be accessed in the following ways as per requirement:
 - To access the class variable inside the constructor, use either `self` or class name.
 - To access the class variable inside the instance method, use either `self` or class name.
 - To access the class variable from outside the class, use object name or class name.
- **Instance variable:** An instance variable is a type of variable that is defined inside the `__init__()` method and other instance methods of a class. Objects do not share instance variables; therefore, each object keeps its own separate copy with different values of the instance attribute. The two ways to access instance variables are as follows:
 - Inside the class in instance method by using `self` for object reference.
 - Outside the class by using the method `getattr()` with the following syntax:

```
getattr(Object name, 'instance variable')
```

Let us understand the usage of instance and class variables with the following example:

```
class Department:  
    location = 'None'  # Class variable 'location'  
  
    def __init__(self, dname, dmgr):  # constructor  
        self.dname = dname  # Instance variable dname  
        self.manager = dmgr  # Instance variable manager  
    Department.location = "India" # Class variable location  
  
    # Instance method display()  
    def display(self):  
        print('Inside instance method display())')  
        # To access using all variables using self  
        print(self.dname, self.manager, self.location)
```

```

# To access class variable using class name
print(Department.location)

# Now create objects d1 and d2
d1 = Department("Marketing", "Raman")
d2 = Department("Finance", "Anil")
# To access instance variable and class variable using object
print('**** Details of First Object ****')
print('Name -> ', d1.dname, ', Manager ->', d1.manager)
print('Location : ', d1.location)
d1.display()
print('**** Details of Second Object ****')
# To access instance variable using getattr()
print('Name -> ', getattr(d2, 'dname'))
print('Manager ->', getattr(d2, 'manager'))
# Access class variable using class name
print('Location : ', Department.location)
d2.display()

```

Output:

Figure 5.4 shows the output:

	**** Details of Second Object ****
**** Details of First Object ****	
Name -> Marketing , Manager -> Raman	Name -> Finance
Location : India	Manager -> Anil
Inside instance method display()	Location : India
Marketing Raman India	Inside instance method display()
India	Finance Anil India
	India

Figure 5.4: Output

Here, we have one class variable named **location** and two instance variables defined inside the **__init__()** method. We can see how class variables and instance variables can be accessed at different locations both inside and outside the class.

The three different types of methods present as class members are discussed as follows:

- **Instance method:** Instance methods are defined inside a class to use and modify instance variables by applying a sequence of operations on them. An instance method is bound to the class object. By default, any method declared inside a class is treated as an instance method unless the user explicitly declares it as a class or static method. While declaring the method, the keyword **self** is given as the first parameter, which refers to the current object in use. An instance method can be called using an object name with a dot (.) operator:

```

class Department:

    def __init__(self, dept_id, dname):
        self.dept_id = dept_id # Instance variable
        self.dept_name = dname # Instance variable

    # Instance method display() to access instance variable
    def display(self):
        print('DeptID:', self.dept_id, 'Dept Name:', self.dept_name)

    # Instance method to modify instance variable and add new
    def edit(self, dname, dmgr):
        self.dept_name = dname
        self.manager = dmgr # add new attribute to current
        object

obj1 = Department(1001, "Marketing") # Create object obj1
obj1.display() # Call instance method
new_dept_name = input('Enter new Department name')
new_mgr_name = input('Enter Manager name to update')
# call edit() to update instance variable dept_name and add
manager
obj1.edit(new_dept_name, new_mgr_name)
print("**** Updated Details ****")
print("Dept ID : ", obj1.dept_id)
print("Dept Name : ", obj1.dept_name)
print("Manager Name : ", obj1.manager)

```

Output:

Figure 5.5 shows the output:

```
Dept ID: 1001 Dept Name: Marketing
Enter new Department name Marketing & Finance
Enter Manager name to update Raman Kumar
**** Updated Details ****
Dept ID : 1001
Dept Name : Marketing & Finance
Manager Name : Raman Kumar
```

Figure 5.5: Output

- **Class method:** As the name suggests, a class method is linked to the class itself and not to individual class objects. Only class variables are accessible to a class method. By altering the value of a class variable that would affect all class objects, it can change the state of the class. If we use only class variables inside a method, then we must explicitly declare that method as a class method by using the **@classmethod** annotation or **classmethod()** function. While defining a class method, the keyword **cls** is used as the first parameter, which refers to the class. A class method can be called using the class name with the dot (.) operator. For example:

```
class Department:

    location = 'None' # class variable for dept location

    def __init__(self, dname, dmgr):
        self.dname = dname
        self.manager = dmgr

    # Define class method to change dept_loc
    @classmethod
    def change_loc(cls, new_loc):
        # class_name.class_variable
        cls.location = new_loc

    # Instance method to display()
    def display(self):
        print('Department: ', self.dname, ', Manager:', self.manager)
```

```

print('Location:', Department.location)

obj1 = Department('Sales', 'Kartik Kumar')
obj1.display()
# To update department location
new_loc = input("Enter dept location to update")
Department.change_loc(new_loc)
obj1.display()

```

Output:

Figure 5.6 shows the output:

```

Department : Sales , Manager: Kartik Kumar
Location: None
Enter dept location to update India
Department : Sales , Manager: Kartik Kumar
Location: India

```

Figure 5.6: Output

- **Static method:** Similar to class method, a static method is also linked to the class itself and not any class objects. But unlike class methods and instance methods, a static method does not have access to the class and instance variables. This is so because, in its declaration, no **self** or **cls** arguments are used. Hence, a static method is used to perform a standalone task, but it cannot modify the state of a class or an object. A static method can be called using the class name or object name with the dot operator. A static method may act as a utility method that can be called from another method using **self** and without any object reference. Let us consider an example of a static method:

```

class Department:

    # Define static method in class
    @staticmethod
    def staticmethod1(num):
        print('Inside static method num is : ', num)

    # Define instance method and call static method from inside
    def method2(self):
        print('Inside instance method....')
        print('Calling static method...')


```

```

        self.staticmethod1(202)

# Call static method using class name
Department.staticmethod1(101)
# Call instance method using class object
d1 = Department()
d1.method2()

```

Output:

Figure 5.7 shows the output:

```

Inside static method num is : 101
Inside instance method....
Calling static method..
Inside static method num is : 202

```

Figure 5.7: Output

Built-in attributes of class

A Python class comprises additional built-in class attributes that provide details about the class, in addition to the other properties. *Table 5.1* lists the built-in class attributes:

S. no.	Attribute	Description
1	<code>__dict__</code>	This attribute provides the dictionary containing the information about the class namespace.
2	<code>__doc__</code>	It represents a string representing the class documentation
3	<code>__name__</code>	The class name can be accessed using <code>__name__</code> attribute.
4	<code>__module__</code>	It is used to access the name of the module containing the class. Every module in Python has a special attribute called <code>__name__</code> . The value of <code>__name__</code> attribute is set to <code>__main__</code> when the module is run as the main program.
5	<code>__bases__</code>	It gives a tuple containing the names of all base classes of a specified class.

Table 5.1: Built-in class attributes

Consider the following code:

```

class Project:
    """This class contains current project details"""

    def __init__(self, name, id, duration):

```

```

        self.pname = name
        self.pid = id
        self.duration = duration

    def display(self):
        print("Name: %s, ID: %d, Duration: %d " %(self.pname,
self.pid, self.duration))

proj1 = Project("Website", 1001, 180)
print("proj1.__doc__      => ", proj1.__doc__)
print("proj1.__dict__     =>", proj1.__dict__)
print("proj1.__module__   =>", proj1.__module__)
print("proj1.__class__    =>", proj1.__class__)

```

Output:

Figure 5.8 shows the output:

```

proj1.__doc__      => This class contains current project details
proj1.__dict__     => {'pname': 'Website', 'pid': 1001, 'duration': 180}
proj1.__module__   => __main__
proj1.__class__    => <class '__main__.Project'>

```

Figure 5.8: Output

Constructors in Python

The task of creating and initializing an object is fulfilled by a special method known as a **Constructor** defined inside a class. A constructor is automatically called when a new class object is created. The syntax for creating a constructor is as follows:

```

def __init__(self, arguments):
    # Statements

```

Similar to Python functions, the constructor declaration also starts with the **def** keyword followed by the **__init__()** method, prefixed and suffixed with double underscores. It takes one mandatory argument called **self** that refers to the current instance of the class along with any number of other arguments as per requirement. Unlike functions, a constructor is automatic and does not return any value. There are three types of constructors in Python, namely:

- Parameterized constructor
- Non-parameterized constructor
- Default constructor

Parameterized constructor

In a constructor definition, it may accept multiple arguments along with **self**. Such a type of constructor is known as a parameterized constructor. The values for the data members can be assigned inside the class using these parameters. Let us consider an example:

class Product:

```
    quantity = 0
```

```
    color = 'None'
```

```
# Constructor Type => Parameterized
```

```
def __init__(self, qty, clr):
```

```
    print("Demo for parametrized constructor")
```

```
    self.quantity = qty
```

```
    self.color = clr
```

```
def display(self):
```

```
    print("The value of quantity : ", self.quantity)
```

```
    print("The value of color : ", self.color)
```

```
obj1 = Product(10, 'Red') # First Object created with parameters
```

```
obj1.display()
```

```
obj2 = Product(7, 'Blue') # Second Object created with parameters
```

```
obj2.display()
```

Output:

Figure 5.9 shows the output:

```
Demo for parametrized constructor
The value of quantity :  10
The value of color :  Red
Demo for parametrized constructor
The value of quantity :  7
The value of color :  Blue
```

Figure 5.9: Output

Non-parameterized constructor

A constructor is referred to be a non-parameterized one when it only accepts the **self** argument and does not accept any other parameters from the object. Using a non-parameterized constructor, each object is initialized with the same set of default values. Let us consider an example:

```
class Product:  
    quantity = 0  
    color = 'None'  
  
    # Constructor Type => Non-parameterized  
    def __init__(self):  
        print("Demo for Non-parametrized constructor")  
        self.quantity = 10  
        self.color = "Green"  
    def display(self):  
        print("The value of quantity : ", self.quantity)  
        print("The value of color : ", self.color)  
  
# Create object with non-parameterized constructor  
obj1 = Product()  
obj1.display() # call display()  
obj2 = Product()  
obj2.display() # call display()
```

Output:

Figure 5.10 shows the output:

```
Demo for Non-parametrized constructor  
The value of quantity : 10  
The value of color : Green  
Demo for Non-parametrized constructor  
The value of quantity : 10  
The value of color : Green
```

Figure 5.10: Output

Default constructor

In case no class constructor is given by the user, Python creates a default constructor automatically during the program compilation process. This auto-generated default constructor has no code in its definition, and therefore, it is also regarded as an empty constructor. For example:

```
class Product:  
    quantity = 0  
    color = 'None'  
  
    # No user defined constructor given  
    def display(self):  
        print("The value of quantity : ", self.quantity)  
        print("The value of color : ", self.color)  
  
obj = Product()  
obj.display()
```

Output:

Figure 5.11 shows the output:

```
The value of quantity :  0  
The value of color :  None
```

Figure 5.11: Output

Encapsulation and data hiding

One of the essential concepts of object-oriented programming is **Encapsulation**. It refers to the binding of attributes and methods together into a single unit called a class. Encapsulation restricts access to and modification of a class's attributes and methods by other classes. This enables us to achieve the feature of **Data Hiding**. Access modifiers restrict the use of a class's variables and methods. Private, public, and protected are the three different forms of access modifiers offered by Python. However, direct access modifiers are not available, but we may use the **single _** or **double __** underscore as a prefix to indicate protected and private attributes (hidden variables) in a class. The three types of variables as per access modifiers are as follows:

- **Public Member:** It is accessible anywhere from inside or outside the class.
- **Private Member:** It is accessible within the class only and not outside.
- **Protected Member:** It is accessible from within the class and its sub-classes only.

Python requires the usage of setters and getters in order to provide good encapsulation. To access and edit data members, use the getter and setter methods, respectively. Consider the following example:

```

class Product:
    def __init__(self, p, q, n):
        self.__maxSellingPrice = p # Private data member
        self._quantity = q         # Protected data member
        self.name = n              # Public data member

    # getter method to return __maxSellingPrice
    def getMaxSellingPrice(self):
        return self.__maxSellingPrice

    # setter function to change __maxSellingPrice
    def setMaxSellingPrice(self, price):
        self.__maxSellingPrice = price

obj1 = Product(800, 10, "Commodity")
# Trying to change price outside class
obj1.__maxSellingPrice = 1050
print("The max selling price using direct : ",
obj1.getMaxSellingPrice())

# Using setter function to change price
obj1.setMaxSellingPrice(1050)
print("The max selling price using setter : ",
obj1.getMaxSellingPrice())

# Direct access protected data member
print('The quantity of product is :', obj1._quantity)
# Direct access public data member
print('The name of product is :', obj1.name)

```

Output:

Figure 5.12 shows the output:

```

The max selling price using direct : 800
The max selling price using setter : 1050
The quantity of product is : 10
The name of product is : Commodity

```

Figure 5.12: Output

In this example, the variable `__maxSellingPrice` is declared as a private attribute inside `__init__()`. In line 17, we attempted to change the `__maxSellingPrice` value outside of the class. As it is a private variable, the output does not reflect this change. Thus, we must use the setter function `setMaxSellingPrice()`, which accepts price as an argument, in order to change the value of `__maxSellingPrice`.

Inheritance in Python

The ability of one class to derive or inherit properties from another class is known as **Inheritance**. The class that derives properties from another class is called the **Derived class** or **Child class**, and the class from which the properties are being derived is called the **Base class** or **Parent class**. The advantages of inheritance include the following:

- It accurately depicts relationships in the real world.
- It offers code reusability of parent class by child subclass. Additionally, it enables us to expand a class's features without actually changing it.
- Because of its transitive nature, if a class B inherits from class A, then all of class B's subclasses will also automatically inherit from class A, forming a hierarchy of classes.

Types of inheritance

Figure 5.13 shows the types of class inheritance supported by Python:

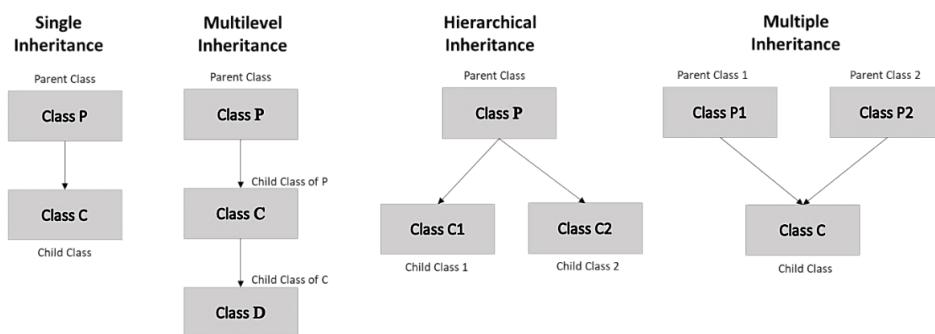


Figure 5.13: Types of Inheritance in Python

The different types of inheritance in Python are as follows.

Single inheritance

A child class inherits from a single-parent class in a **single inheritance**. Here, we have a parent class and a child class.

```

class Parent:      # Base class
    def base_info(self):
        print('Inside Base class method')

class Child(Parent): # Child class
    def child_info(self):
        print('Inside Child class method')

# Create object of Child class
obj1 = Child()

# Access Parent class info Child class object
obj1.base_info()
obj1.child_info()

```

Output:

Figure 5.14 shows the output:

```

Inside Base class method
Inside Child class method

```

Figure 5.14: Output

Multilevel inheritance

Let us assume three classes P, C, and D. Here, P is the Parent class (superclass), C is the child class of P, and D is the child class of C. This forms a hierarchy of classes on multiple levels. This type of inheritance is called **multilevel inheritance**:

```

class Parent:      # Base class
    def base_info(self):
        print('Inside Parent class method')

class Child1(Parent): # Child class of Parent
    def child1_info(self):
        print('Inside Child1 class method')

class Child2(Child1): # Child class of Child1
    def child2_info(self):

```

```

        print('Inside Child2 class method')

# Create object of Child2 class
obj1 = Child2()

# Access Parent class info Child class object
obj1.base_info()
obj1.child1_info()
obj1.child2_info()

```

Output:

Figure 5.15 shows the output:

```

Inside Parent class method
Inside Child1 class method
Inside Child2 class method

```

Figure 5.15: Output

Hierarchical inheritance

A single-parent class gives rise to multiple child classes under **hierarchical inheritance**. Thus, we can say that there is one parent class and several child classes:

```

class Parent:          # Base class

    def base_info(self):
        print('Inside Parent class method')

class FirstChild(Parent): # First Child class of Parent

    def first_child_info(self):
        print('Inside First Child class method')

class SecondChild(Parent): # Second Child class of Parent

    def sec_child_info(self):
        print('Inside Second Child class method')

# Create object of Child classes
obj1 = FirstChild()
obj2 = SecondChild()

```

```
# Access Parent class info Child class object
obj1.base_info()
obj1.first_child_info()
obj2.base_info()
obj2.sec_child_info()
```

Output:

Figure 5.16 shows the output:

```
Inside Parent class method
Inside First Child class method
Inside Parent class method
Inside Second Child class method
```

Figure 5.16: Output

Multiple inheritance

One child class may inherit from several parent classes under **multiple inheritance**. Therefore, we have a single-child class and numerous parent classes. It is conceptually opposite to Hierarchical Inheritance:

```
class Parent1:          # Base class
    def parent1_info(self):
        print('Inside Parent1 class method')

class Parent2:          # Base class
    def parent2_info(self):
        print('Inside Parent2 class method')

class Child(Parent1, Parent2): # Child class of Parent
    def child_info(self):
        print('Inside Child class method')

# Create object of Child classes
obj1 = Child()

# Access Parent class info using Child class object
obj1.parent1_info()
```

```
obj1.parent2_info()  
obj1.child_info()
```

Output:

Figure 5.17 shows the output:

```
Inside Parent1 class method  
Inside Parent2 class method  
Inside Child class method
```

Figure 5.17: Output

Hybrid inheritance

Hybrid inheritance refers to the type of inheritance that combines multiple distinct inheritance types together. Generally, a combination of multiple and multilevel inheritance types gives rise to Hybrid Inheritance. *Figure 5.18* depicts the working of hierarchical inheritance in Python:

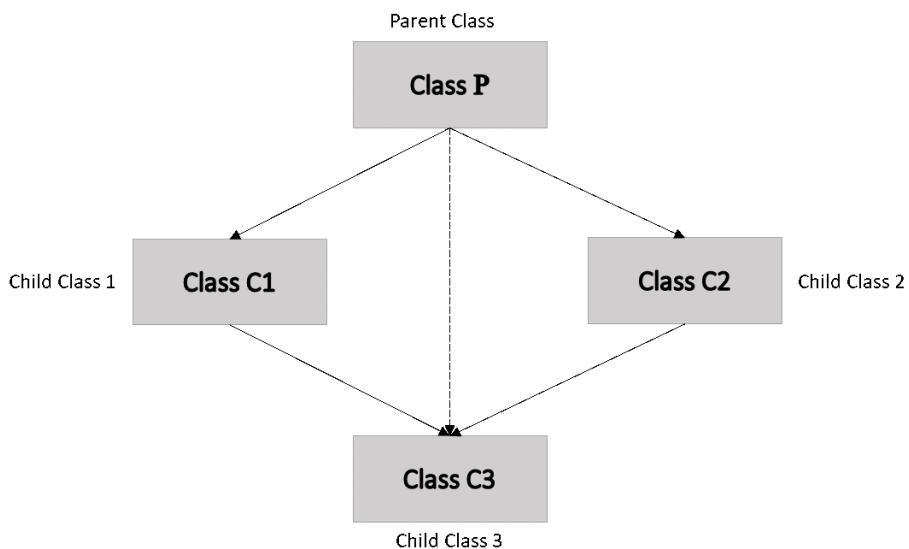


Figure 5.18: Hybrid inheritance and diamond ambiguity

Let us consider an example of Hybrid Inheritance:

```
class GrandParent:  
    def grandparent_info(self):  
        print('Inside Grand Parent class method')  
  
class Parent1(GrandParent):      # Base class  
    def parent1_info(self):
```

```

        print('Inside Parent1 class method')

class Parent2(GrandParent):      # Base class
    def parent2_info(self):
        print('Inside Parent2 class method')

class Child(Parent1, Parent2):  # Child class of Parent
    def child_info(self):
        print('Inside Child class method')

# Create object of Child classes
obj1 = Child()

# Access Parent class info using Child class object
obj1.grandparent_info()
obj1.parent1_info()
obj1.parent2_info()
obj1.child_info()

```

Output:

Figure 5.19 shows the output:

```

Inside Grand Parent class method
Inside Parent1 class method
Inside Parent2 class method
Inside Child class method

```

Figure 5.19: Output

In this example, we can see Hybrid inheritance representing a combination of hierarchical and multiple inheritance. Here, the class **GrandParent** is inherited by two child classes named **Parent1** and **Parent2**, which are further inherited by class **Child**. Hybrid Inheritance raises a programming problem known as **Diamond Inheritance** ambiguity. It occurs when two classes, say X and Y, inherit from a superclass P, whereas another class Z inherits from both classes X and Y. Suppose a method named **display()** is present in either Class X or Class Y or in both. In this case, it is unclear which version of **display()** Class Z should inherit. This problem is solved using the **Method Resolution Order** followed by the derived class.

Method Resolution Order

While executing multiple inheritance in Python, any particular method or attribute is, first of all, searched in the current child class. If not found, then parent classes are searched in left-to-right order as per the inheritance sequence declared in the child class. This order of searching is known as **Method Resolution Order (MRO)**. There are the following two ways to view the MRO of a class:

- Use the **Classname.mro()** method, which returns a list of class sequence
- Use the **Classname.__mro__** attribute, which returns a tuple of the class sequence

The following are some rules for working with MRO:

- Prior to searching in the base classes, the first rule is to search in the subclass.
- The second rule is that when there is multiple inheritance, the base classes are searched from left to right. For instance, if class C is syntactically known as class C (A, B), it will first search in A before moving on to B.
- The third rule is that it will not search the same class twice.

Let us discuss more with an example:

```
class First:  
    def __init__(self):  
        print("In First class")  
        super().__init__() # In MRO hierarchy call __init__()  
  
class Second(First):  
    def __init__(self):  
        print("In Second class") # In MRO hierarchy call  
        __init__()  
        super().__init__()  
  
class Third(Second):  
    def __init__(self):  
        print("In Third class")  
        super().__init__()  
  
obj1 = Third()  
print(Third.__mro__)
```

Output:

Figure 5.20 shows the output:

```

In Third class
In Second class
In First class
(<class '__main__.Third'>, <class '__main__.Second'>, <class '__main__.First'>, <class 'object'>)

```

Figure 5.20: Output

This example shows the calling of the `__init__()` method according to the method resolution order that can be observed in the output.

super() in Python

In Python, a very special function known as `super()` is present. As the name suggests, `super()` has indeed very powerful applications in class inheritance. We can use the `super()` function in child classes to refer to the parent class methods. A temporary object of the parent class is returned by the `super` function, enabling us to call a parent class method inside a child class method. The advantages of employing the `super()` function in Python classes are as follows.

- To access the parent class's methods, we do not need to remember or specify the name of the parent class.
- The `super()` function is applicable to both single inheritance and multiple inheritance.

Super function in single inheritance

Let us consider an example where the `super()` function is used in a single inheritance. Here, we have a parent class `Project` and its sub-class `Staff`, such that the sub-class method `display()` tries to access its base class method `project_name()` using `super()`:

```

class Project:
    def project_name(self):
        return 'Software Development'

class Staff(Project):
    def display(self):
        # Calling the superclass method using super()function
        p_name = super().project_name()
        print("Current Project under work: ", p_name)

# Creating object of Staff class
obj1 = Staff()
obj1.display()

```

Output:

Figure 5.21 shows the output:

Figure 5.21: Output

Super function in multiple inheritance

Let us consider an example of super with multiple inheritance:

```
class First:
    def __init__(self, num):
        print("In First class num = ", num)
        # In MRO hierarchy call __init__() of class Second
        super().__init__(num+10)

class Second:
    def __init__(self, num):
        print("In Second class num = ", num)
        # In MRO hierarchy call __init__() of object class
        super().__init__()

class Third(First, Second):
    def __init__(self, num):
        print("Starting in class Third num = ", num)
        super().__init__(num+10)
        print("Back in Third class num = ", num)

obj1 = Third(10)
print(Third.__mro__)
```

Output:

Figure 5.22 shows the output:

```
Starting in class Third num =  10
In First class num =  20
In Second class num =  30
Back in Third class num =  10
(<class '__main__.Third', <class '__main__.First', <class '__main__.Second', <class 'object'>)
```

Figure 5.22: Output

Here, `super().__init__()` calls and executes the next `__init__()` method according to the **Method Resolution Ordering (MRO)** given in the output of the previous program. Because the `super()` call determines the subsequent method in the MRO at each step, for cooperative subclassing, the `super()` calls are required in the parent classes also.

Polymorphism in Python

To understand the process of polymorphism, let us first break this word into two simple words, that is, “Poly,” which means many, and “Morphs,” meaning forms. In computer programming, polymorphism refers to using the same function name multiple times with distinct signatures for different tasks. Here, signatures refer to the type and/or the number of arguments used in the function definition. Polymorphism is of the following two types:

- Compile-time Polymorphism
- Run-time Polymorphism

Compile-time Polymorphism

When it is possible to determine which method definition will be called at the time of program compilation by simply examining the method signatures, it is called **Compile-time polymorphism**. It is also known as static polymorphism or early binding. The concept of overloading methods, constructors, and operators is used to achieve compile-time polymorphism.

Method and constructor overloading

Python does not support method overloading with user-defined methods. This is because if we declare multiple method definitions with the same name but different numbers or types of arguments, then the Python interpreter will always take into consideration only the last method definition while ignoring all others. However, method overloading is defined internally for built-in methods. Let us see an example of overloading with `len()` method for different arguments:

```
print("For String : ", len("Hello Python"))
print("For List : ", len(["C++", "Python", "Java", "C#"]))
print("For Dictionary : ", len({"EID": 1001, "Age": 20}))
```

Output:

Figure 5.23 shows the output:

```
For String : 12
For List : 4
For Dictionary : 2
```

Figure 5.23: Output

Similarly, the idea of having multiple constructors in a class, each with a different set of parameters, is called **constructor overloading**. However, this concept of constructor overloading is also not supported by Python. If we specify multiple constructors, the interpreter will only take into

account the last constructor declared in class, and it throws **TypeError** if the list of the arguments does match the definition of the last constructor.

Operator overloading

Operator overloading refers to altering an operator's default behavior, based on the operands (values) we use. In other words, we can apply the same operator with different types of arguments to achieve different outcomes. For instance, when used with numbers, the “+” operator will carry out an addition operation, whereas on used with strings, it will perform string concatenation. To execute overloading operations on user-defined custom objects in Python, various **Magic Methods** are available. In Python, Magic Methods, also known as Dunder Methods, are a group of special methods that begin and end with the double underscores. These methods are not meant to be invoked directly by the user but are invoked internally from the class on the occurrence of a certain action. For example, on adding two numbers using the “+” operator, the `__add__()` magic method will be called and executed automatically. In order to display the overloaded behavior of pre-defined operators in a custom class, the corresponding magic method must be overridden. Suppose we want to use the “+” operator with the objects of a user-defined class; for this, we must override `__add__()` magic method.

Table 5.2 lists the names of the Magic Methods that can overload the relational, assignment, and mathematical operators in Python:

Operator Name	Symbol	Magic Method
Addition	+	<code>__add__(self, other)</code>
Subtraction	-	<code>__sub__(self, other)</code>
Multiplication	*	<code>__mul__(self, other)</code>
Division	/	<code>__div__(self, other)</code>
Floor Division	//	<code>__floordiv__(self, other)</code>
Modulus	%	<code>__mod__(self, other)</code>
Power	**	<code>__pow__(self, other)</code>
Power	**=	<code>__ipow__(self, other)</code>
Less than	<	<code>__lt__(self, other)</code>
Greater than	>	<code>__gt__(self, other)</code>
Less than or equal to	<=	<code>__le__(self, other)</code>
Greater than or equal to	>=	<code>__ge__(self, other)</code>
Equal to	==	<code>__eq__(self, other)</code>
Not equal	!=	<code>__ne__(self, other)</code>

Table 5.2: Magic methods in Python

Let us see an example of using the magic method for overloading the less than operator to compare the price and the addition operator to add product quantities for objects of the **Product** class:

```
class Product:
```

```

def __init__(self, qty, price):
    self.quantity = qty
    self.price = price

def __lt__(self, obj2):    # overload < operator
    return self.price < obj2.price

def __add__(self, obj2):    # overload < operator
    return self.quantity + obj2.quantity

prod1 = Product(5, 100)
prod2 = Product(7, 80)

print("Product-1 cheaper than product-2 : ", prod1 < prod2)  #
prints False

print("Total products : ", prod1 + prod2)  # print sum of
quantities

```

Output:

Figure 5.24 shows the output:

```

Product-1 cheaper than product-2 :  False
Total products :  12

```

Figure 5.24: Output

Run-time polymorphism

The technique of resolving a call to an overridden method at run-time is known as **Runtime polymorphism**, also called **Dynamic Method Dispatch**. Python's idea of **Method Overriding** is used to implement it. It is referred to as method overriding when a child class method has the same name, parameters, and return type as a parent class method but implements it differently. In this scenario, the parent class's method is referred to as the overridden method, and the method defined in the child class is known as the overriding method. When an overridden method is called, which version of the overridden method will be executed is decided at the run-time using MRO. Let us consider an example of implementing run-time Polymorphism in Python:

```

class Staff():

    def calculate(self):  # Parent class
        print("Inside calculate() of Staff")

    def display(self):

```

```

        print("Inside display() of Staff")

class Developer(): # Parent class
    def calculate(self): # Parent class
        print("Inside calculate() of Developer")
    def display(self):
        print("Inside display() of Developer")

class PythonExpert(Staff, Developer): # Defining child class
    def display(self): # Child's display method
        print("Inside display() of PythonExpert")

obj = PythonExpert() # Driver's code
obj.display()
obj.calculate()
print(PythonExpert.mro())

```

Output:

Figure 5.25 shows the output:

```

Inside display() of PythonExpert
Inside calculate() of Staff
[<class '__main__.PythonExpert'>, <class '__main__.Staff'>, <class '__main__.Developer'>, <class 'object'>]

```

Figure 5.25: Output

In this case, we can observe that as per MRO, first of all, any method will be searched in the **PythonExpert** class; if not found, then in class **Staff** and last in class **Developer**. Using a similar concept, when a constructor exists in the subclass, then the parent class constructor is not available to sub-class. Rather, only the sub-class constructor is accessible from the sub-class object. Here, we observe that the sub-class constructor is overriding or replacing its parent class constructor. This is called **Constructor Overriding**. Readers are suggested to try an example of Constructor Overloading in Python for practice.

Conclusion

In this chapter, we came across the Object-oriented paradigm and its implementation in Python. We discussed major OOPs concepts such as class, object, Encapsulation, Data Hiding, Inheritance, and Polymorphism with numerous examples. By understanding the concepts followed in this chapter, users can develop their skills by efficiently modeling real-world problems in the form of classes and objects.

Points to remember

- The programming paradigm known as Object-oriented Programming (OOP) is based on the ideas of classes and objects.
- A Class serves as a user-defined blueprint or template from which objects can be created. Class offers a way to group together functionality and data in a single unit.
- The task of creating and initializing an object is fulfilled by a special method known as a constructor defined inside a class.
- Encapsulation refers to the binding of attributes and methods together into a single unit called a Class.
- The ability of one class to derive or inherit properties from another class is known as Inheritance.
- In computer programming, polymorphism refers to using the same function name multiple times with distinct signatures for different tasks.

Exercise

Attempt the following project.

Sample project with solution

Create a Typing Tester to display the user's typing speed in Characters per second/ minute and Words per second/ minute, along with the total time taken for typing and accuracy score.

The code is given as follows:

```
import random # Used to call random()

import time # Used to calculate time

class Project:

    def __init__(self): # Constructor
        self.sent_para = "" # Initialise empty strings
        self.typed_para = ""

    def readtext(self): # To read text from file
        f = open('D:/BPB Publications/input.txt').read()
        self.sentences = f.split('\n')
        self.sent_para = random.choice(self.sentences)

    def error_rate(self, sent_para, typed_para):
```

```

# To calculate error rate by matching read string with typed
string

    error_count = 0

    length = len(self. sent_para)
    for character in range(length):
        try:
            if sent_para[character] != typed_para[character]:
                error_count += 1

        except:
            error_count += 1
    error_percent = error_count/length * 100
    return error_percent

def calc(self):

    # Allow user to type the string read from file and calculate
    accuracy score

        print("Type below paragraph with a few or no mistakes:
\n")
        print(self.sent_para)
        start_time = time.time()
        self.typed_para = input("Write above text here : ")
        if self.typed_para == "":
            print("Empty String not Allowed!!.. Try Again..")
            self.readtext()
            self.calc()
        else:
            end_time = time.time()
            time_taken = end_time - start_time
            error_percent = self.error_rate(self.sent_para,
self.typed_para)
            print("\n")

```

```

accuracy_score = 100 - error_percent

if accuracy_score < 50:

    print(f"Your accuracy rate {accuracy_score} is
less than 50%. Try again to find for typing speed!!.")

else:

    speed = len(obj1.typed_para) / time_taken
    print("*****YOUR SCORE REPORT*****")
    print(f"Your speed is {speed} words/sec")
    print(f"The error rate is {error_percent}")
    print(f"The accuracy score is {accuracy_score}")

obj1 = Project()
obj1.readtext()
obj1.calc()

```

Output:

Figure 5.26 shows the output:

Type the below paragraph with a few or no mistakes:

It was raining heavily
 Write above text here :
 Empty String not Allowed!!.. Try Again.
 Type the below paragraph with a few or no mistakes:

It was raining heavily
 Write above text here : *It was raining heavily*

*****YOUR SCORE REPORT*****
 Your speed is 1.1630000758182222 words/sec
 The error rate is 4.545454545454546
 The accuracy score is 95.45454545454545

Figure 5.26: Output

Practice project

Create a general knowledge quiz of 10 questions for students. Each question has four alternatives, out of which the user must choose only one correct answer. For each correct answer, 1 point is

added to the total score, whereas deducting 1 point for each incorrect answer. At the end of the quiz, display all the correct answers and calculate the total score for the user.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 6

Turtle Programming in Python

Introduction

In the previous chapters, we have discussed different aspects and basic programming constructs of Python language by creating different user-interactive applications. This chapter introduces the readers to an altogether different type of Python programming by creating graphic objects, animations, and small interactive games using a popular in-built library known as Turtle. After reading this chapter, users will be able to create different interesting games by controlling user actions through mouse and keyboard events.

Structure

In this chapter, we will discuss the following topics:

- Turtle programming in Python
- Creating shapes with turtle
- Fill colors in shapes
- Event programming using turtle

Objectives

By the end of this chapter, the readers will have a deeper understanding of creating colored shapes and moving objects of different sizes, controlled by a user's mouse click or key press. This will enable readers to create their own imaginative graphics and interesting games.

Turtle programming in Python

Python's Turtle library is used to make graphics, images, and games. It was created in 1967 by Cynthia Solomon, Wally Feurzeig, and Seymour Papert. It came with the initial version of the Logo programming language. The Logo programming language was quite popular among young learners because it makes it simple to create visually appealing graphs for the screen.

Python comes with a built-in module called Turtle that functions like a virtual canvas on which we may create beautiful images and forms. We can draw on the screen with the provided on-screen pen. The turtle library's main objective is to acquaint young programmers with programming. They can learn how to use Python to program in a fun and interactive way with the aid of Turtle's library. Because it allows for the creation of distinctive shapes, eye-catching images, and a variety of activities, it is advantageous to both naïve and seasoned programmers. Simple animation and small games can also be created using Turtle.

Plotting with Turtle

To begin with the programming using Turtle, we need to ensure that our system is well configured with Python Environment set-up. We can do Turtle-based programming on Python IDLE shell, PyCharm IDE, or Jupyter Notebook. Since Turtle is a built-in Python library that automatically comes along with the standard Python package, there is no need to install it separately. It can be simply imported into the Python environment by using the following **import** command:

```
import turtle
```

The turtle module is first imported, and after that, a window can be constructed if desired. Otherwise, a default window will be produced, and finally, the turtle can be instructed to draw on this window. An important point to note is that the turtle just paints in the same manner as it moves. Python turtle library consists of various methods and functions that can be used to create nice-looking designs and graphs. Let us see these methods with their details in *Table 6.1*:

Method	Description
<code>Turtle()</code>	Creates and returns a new turtle object.
<code>showturtle()</code>	Makes the Python turtle visible on the window screen.
<code>hideturtle()</code>	Makes the Python turtle invisible on the window screen.
<code>home()</code>	Moves the turtle to the center of the screen and points the turtle to the east.
<code>penup()</code>	The turtle will be lifted off the canvas by using <code>.penup()</code> , and if it is moved while in penup state, it will not draw. It is useful for only repositioning the turtle without actually drawing.
<code>pendown()</code>	It is the default state of the turtle that ensures that turtle draws when it is moving with <code>forward()</code> or <code>setpos()</code> .
<code>color(color name)</code>	Changes the color of the turtle's pen.
<code>fillcolor(color name)</code>	Changes the color of the turtle will use to fill a polygon.
<code>heading()</code>	Returns the current heading.
<code>turtle.setheading(to_angle)</code> or <code>turtle.seth(to_angle)</code>	To set the turtle orientation by giving <code>to_angle</code> . Some common directions in degrees are: 0—east, 90—north, 180—west and 270—south
<code>position()</code>	Returns the current position.
<code>dot()</code>	Leave the dot at the current position.
<code>stamp()</code>	Leaves an impression of a turtle shape at the current location.

Method	Description
shape(shape name)	Shape names can be arrow, classic, turtle, square, or circle.
bgcolor()	Change the background color of canvas window.
done()	It is generally the last statement in a turtle graphics program to complete program execution.
mainloop()	It holds the window and waits for the user to do some action or close the window.

Table 6.1: Turtle library drawing methods

Here is an example of turtle program. We will initialize a variable **my_turtle** to create a new turtle screen object:

```
import turtle
my_turtle = turtle.getscreen() # Creating turtle screen
turtle.mainloop() # To stop the screen to display
```

Output:

The output can be seen in [Figure 6.1](#):

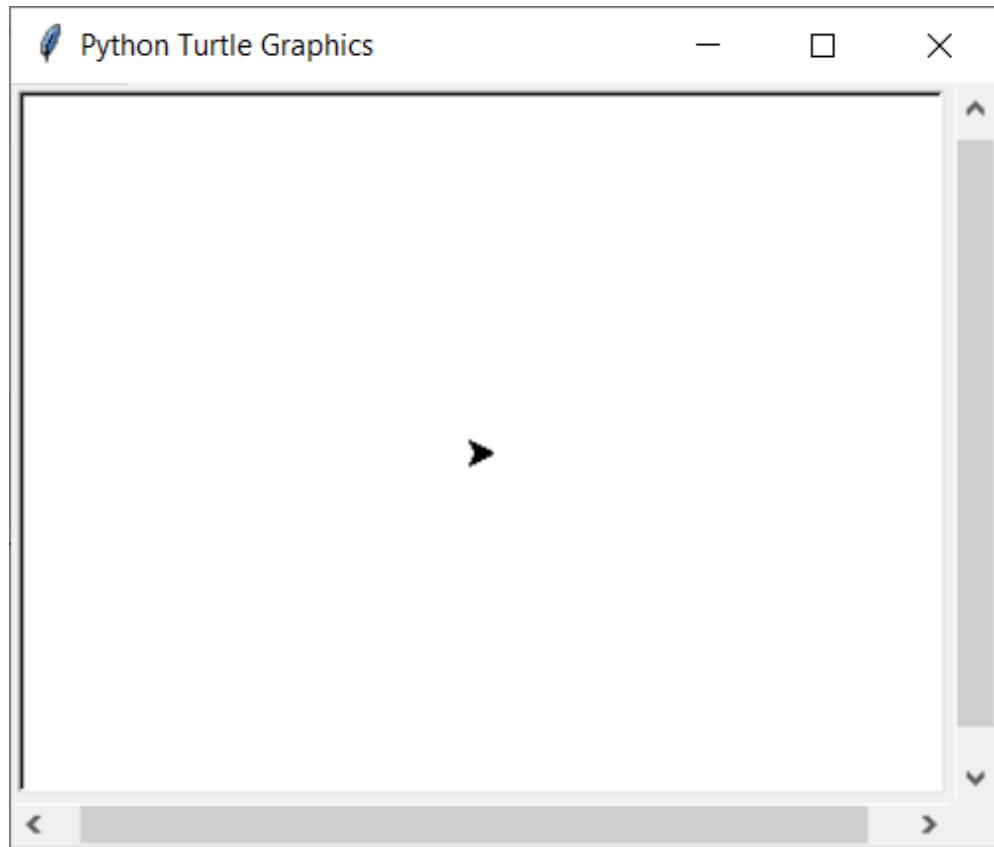


Figure 6.1: A basic turtle figure

The output displays a little triangle in the middle of the screen, and it is known as a turtle. The screen functions like a canvas, whereas the turtle acts like a pen. To create the desired shape, you can move the turtle. The size, color, and speed of the turtle are among its variable characteristics. Unless we instruct it otherwise, it can be moved in a specified direction and will continue to go in that way. Let us see another example to configure a pen and change its shape with color:

```
import turtle

turtle.showturtle()          # Displays turtle on
screen

turtle.shape("square")       # changes turtle shape
to square

turtle.pencolor("red")       # set pen color to red
```

```
turtle.penup()                      # pick up pen
without drawing

turtle.fillcolor("blue")            # fills turtle shape
to blue

turtle.pendown()                   # pick down pen to draw

turtle.forward(100)                # move turtle forward by
100 units

turtle.home()                      # moves turtle to screen
center and point east

turtle.done()                       # complete the execution
```

Output:

The output can be seen in *Figure 6.2*:

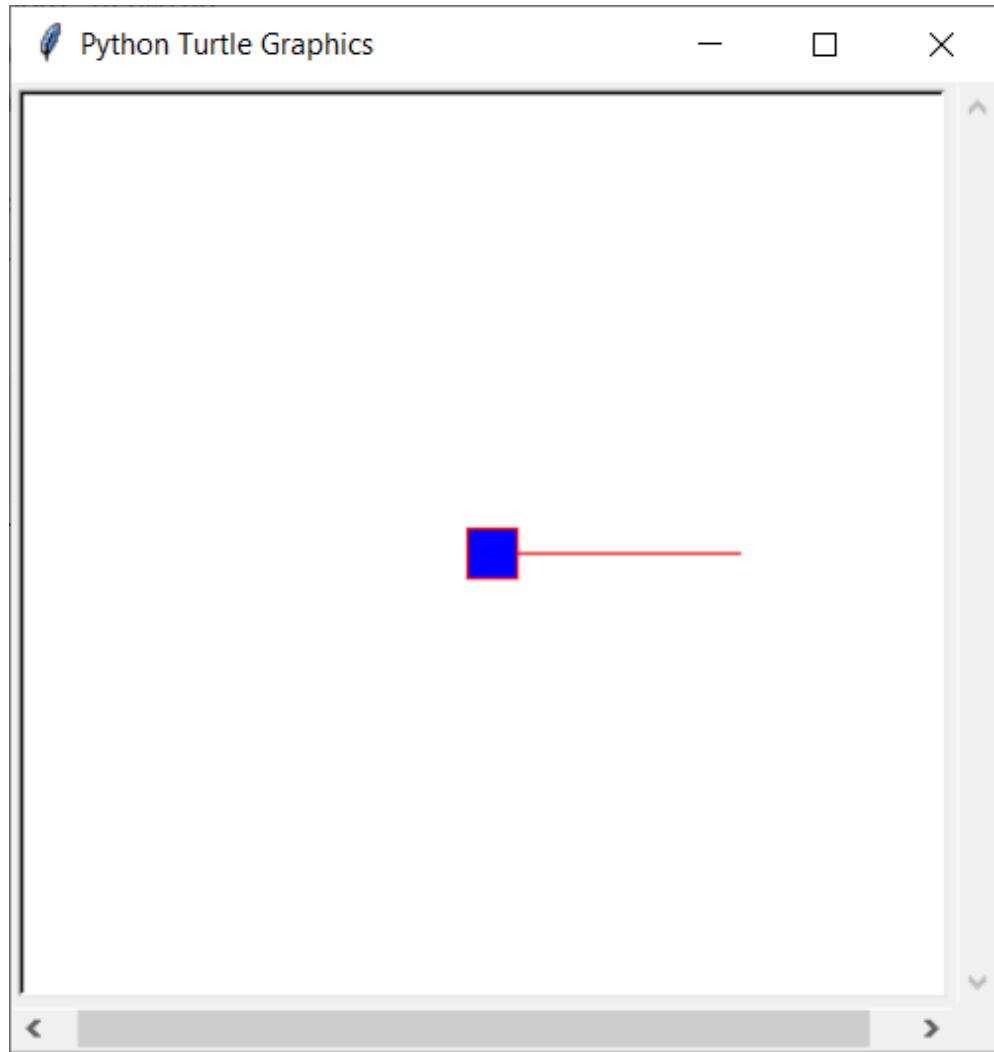


Figure 6.2: A coloured square shape turtle figure

Creating shapes with Turtle

Using the turtle library in Python, we can draw different shapes by moving the turtle object in different directions on the screen canvas. The Python Turtle canvas is divided into four quadrants by an x- and y-axis, and the turtle always starts at (0, 0), which is the exact center of the canvas. On a turtle canvas, we may use this code to draw the four quadrants. The turtle is returned to (0, 0), which is the canvas's middle when the `home()` function is called.

Figure 6.3 features the four quadrants:

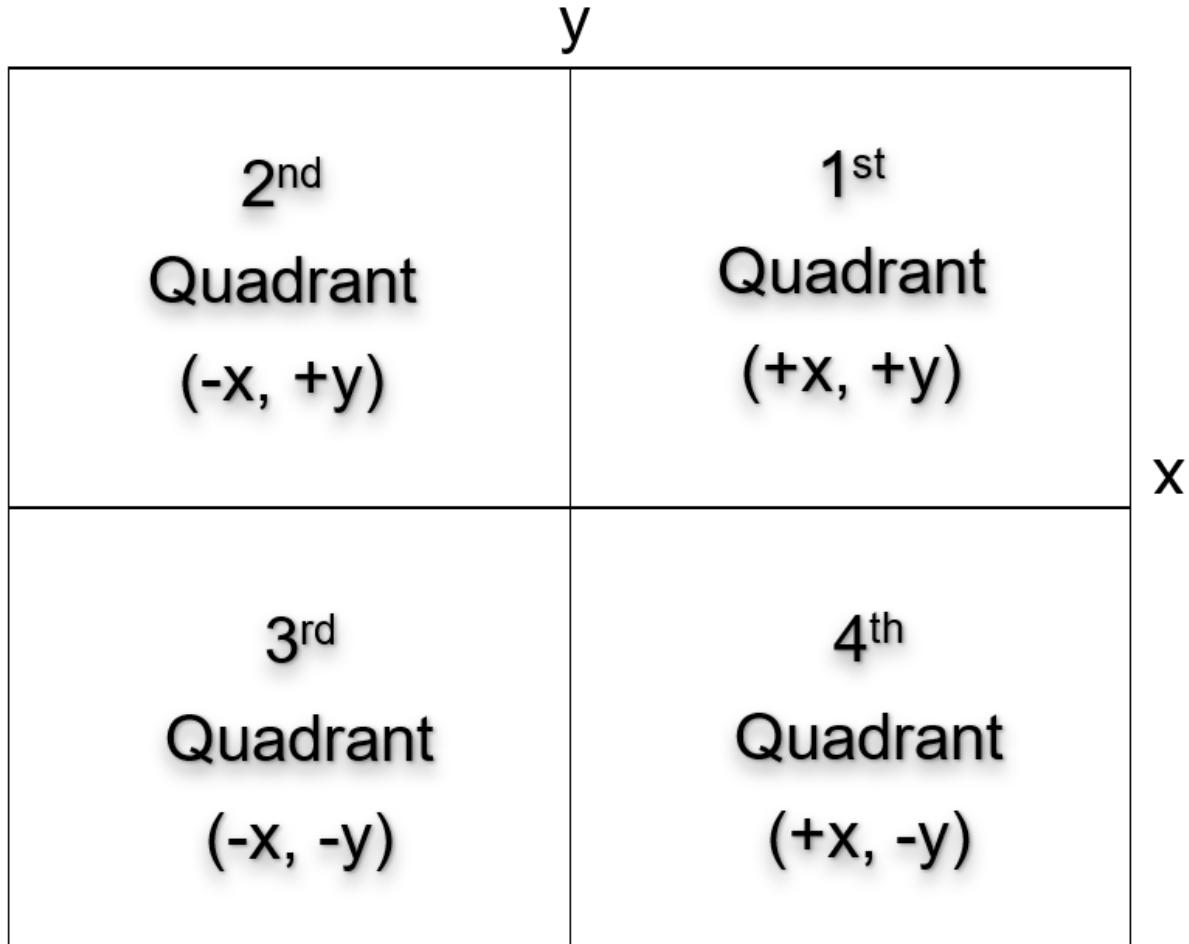


Figure 6.3: Four quadrants of turtle window

Various methods present in turtle library enable the turtle pointer to move and create different shapes. [Table 6.2](#) features a list of such methods:

Method	Description
forward(Value pixels)	Move turtle forward by specified number of pixels.
backward(Value pixels)	Move turtle backward by given number of pixels.
right(Angle in degrees)	Turns the turtle by given angle in clockwise direction.
left(Angle in degrees)	Turns the turtle by given angle in counter clockwise direction.

goto(x, y)	Move and set the turtle to the coordinate position (x, y).
begin_fill()	Marks the starting point for a filled polygon.
end_fill()	Ends the polygon and fills it with the current fill color.
speed()	To change the turtle's drawing speed values ranging from 1 to 10, with 10 being the quickest, and the default value is 3. We can also use the strings slowest, normal, fast, and fastest.

Table 6.2: Turtle movement control methods

Drawing connecting lines

Let us consider a basic example to understand the workings of turtle programming for creating different objects. The following code shows the basic movement of the turtle cursor in forward direction with a left turn:

```
import turtle
my_window = turtle.Screen()
my_window.bgcolor("white") # set white background color
my_pen = turtle.Turtle() # create a new turtle object.
my_pen.forward(100) # turtle moves forward
my_pen.left(90) # Counterclockwise turn turtle by 90
my_pen.forward(80) # turtle moves forward
my_pen.color("black") # set black color of pen
my_pen.pensize(14) # set pen size to 14 points
turtle.done()
```

Output:

The output can be seen in *Figure 6.4*:

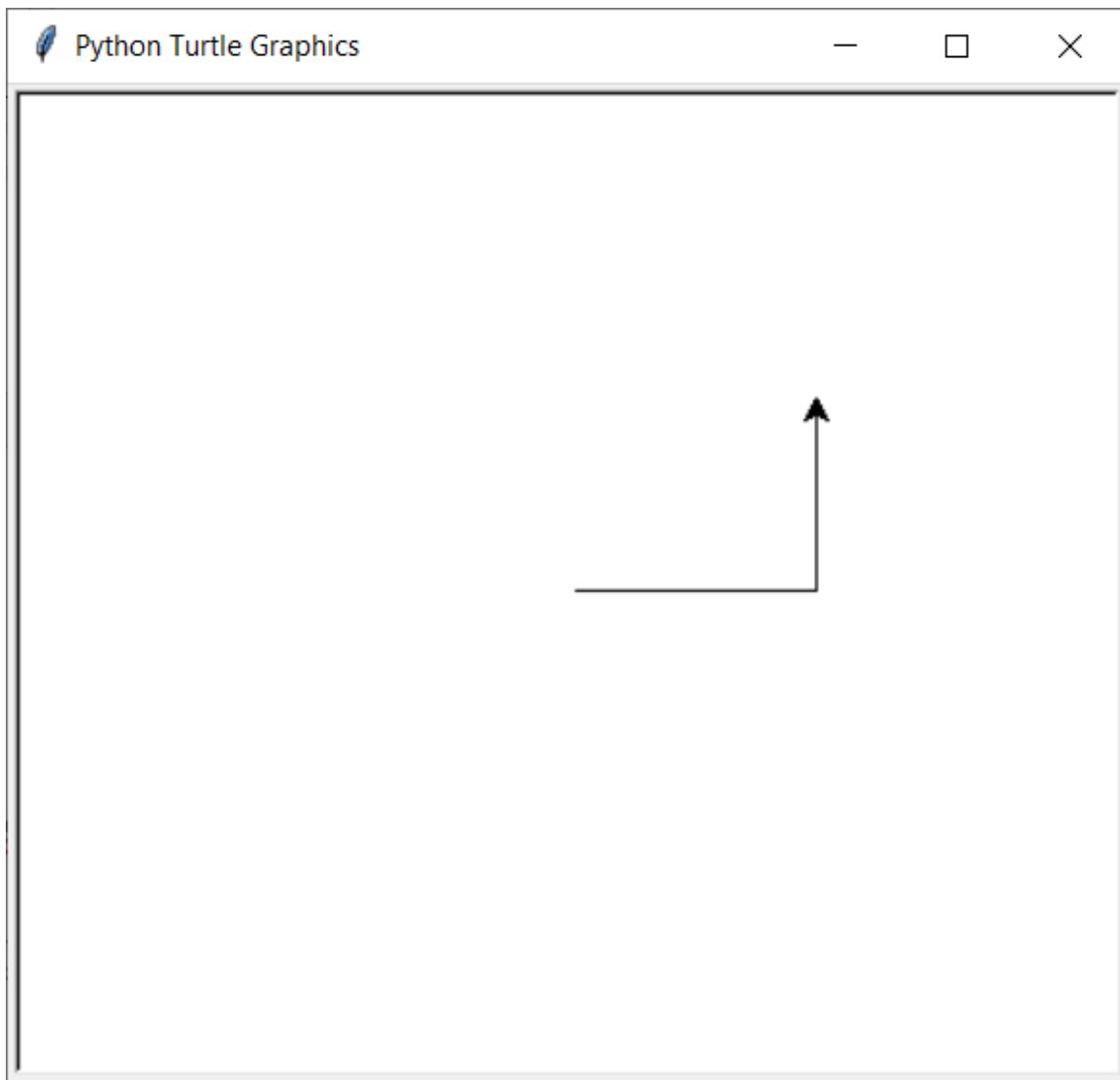


Figure 6.4: Turtle creating lines

Draw square, rectangle, and triangle

Another example depicts drawing basic shapes of square and rectangle on a canvas window using turtle methods such as **penup()**, **pendown()**, **forward()**, **right()**, and so on, to stop or move the turtle cursor in different directions as per requirement:

```
import turtle

# Create a new canvas window with white background
color
```

```
window1 = turtle.Screen()
window1.bgcolor("white")
# Create a new turtle object to draw square
square_turtle = turtle.Turtle()
square_turtle.shape("turtle")
square_turtle.color("red")
# To draw the first edge of the square
# Originally the turtle is moved forward by 100
pixel units
square_turtle.forward(100)
# Move right by 90 degrees to turn in downwards
direction.
square_turtle.right(90)
# Move forward by 100 units to draw the second edge
of square
square_turtle.forward(100)
# Again turn to right by 90 degrees and move
forward by 100 pixels
square_turtle.right(90) # Third edge of square
square_turtle.forward(100)
square_turtle.right(90) # Fourth edge of square
square_turtle.forward(100)
square_turtle.penup() # Stop cursor from drawing
# Create Rectangle with length and breadth as 100
and 200 pixels
```

```
rec_turtle = turtle.Turtle()
rec_turtle.shape("arrow")
rec_turtle.color("blue")
rec_turtle.penup() # Stop turtle from drawing
while moving
rec_turtle.goto(-200, 100) # Set turtle location
(-200, 100)

rec_turtle.pendown() # Start drawing by turtle
rec_turtle.forward(100)
rec_turtle.right(90)
rec_turtle.forward(200)
rec_turtle.right(90)
rec_turtle.forward(100)
rec_turtle.right(90)
rec_turtle.forward(200)

# Create a Triangle
tri_turtle = turtle.Turtle()
tri_turtle.shape("circle")
tri_turtle.color("green")
tri_turtle.penup()
tri_turtle.goto(200, -100)
tri_turtle.pendown()

# Draw 3 sides of triangle
for i in range(3):
```

```
tri_turtle.forward(100)  
tri_turtle.left(120)  
  
# Wait for user mouse click to exit canvas window  
window1.exitonclick()
```

Output:

The output can be seen in *Figure 6.5*:

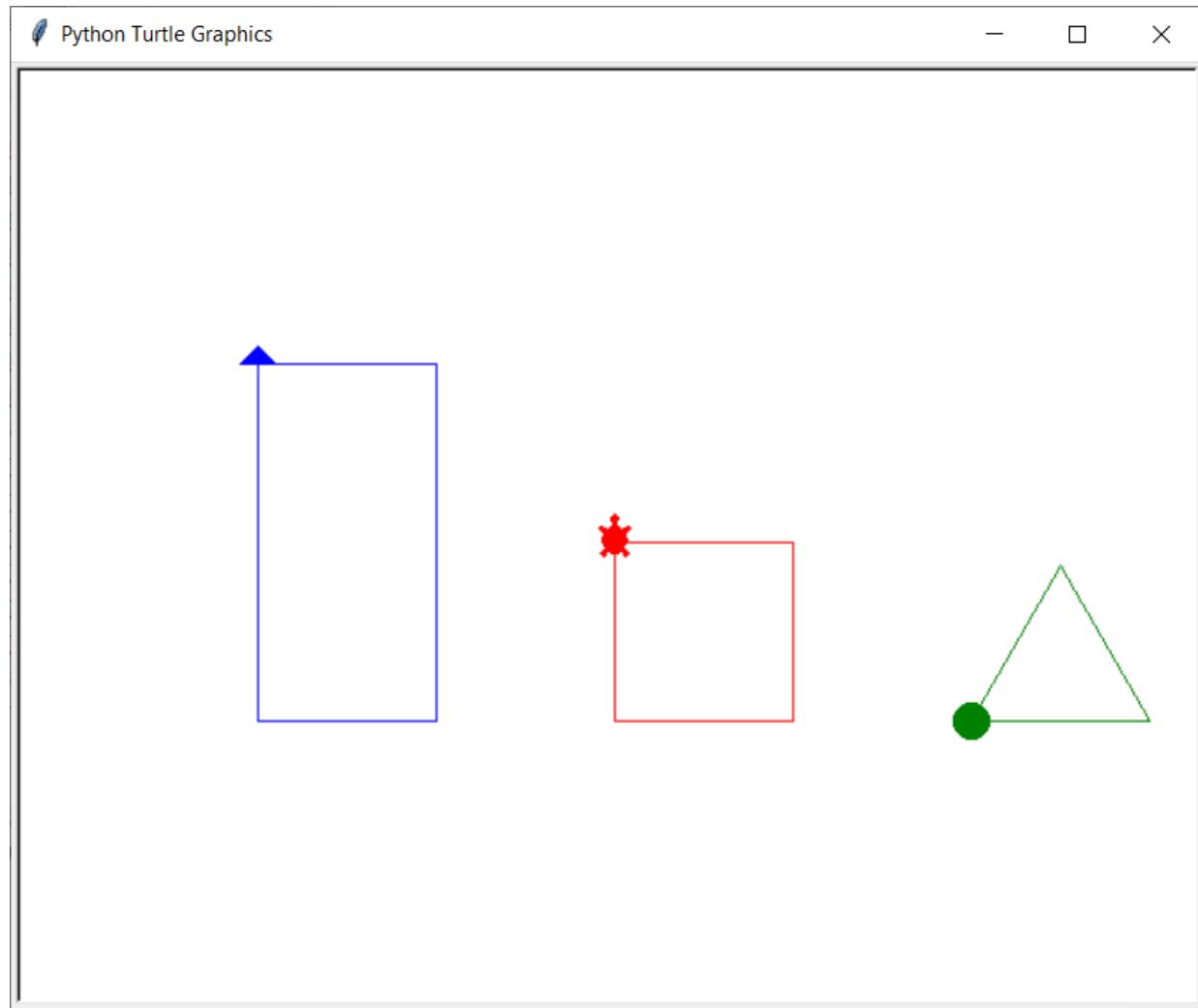


Figure 6.5: Turtle creating different shapes

To draw a square and rectangle, we can create one line as the first edge using the **forward()** method and then use the **right()** method to turn the turtle

by 90 units in a downward direction to create the next edge, and so on. Python Turtle shapes include triangles, which are fairly simple to draw. Simple turtle functions, such as **forward()**, **right()**, and so on, are all that we have used. We used a 120-degree angle in order to draw an equivalent triangle; therefore, we passed 120 on the **left()** function.

Draw star pentagon, hexagon, and octagon

In the following example, we see how to create some other shapes, such as star pentagon, hexagon, and octagon in turtles. Let us consider the following code to draw different shapes:

```
import turtle

# Create a new canvas window with white background
color

window1 = turtle.Screen()
window1.bgcolor("white")

# Create a star shape

star_turtle = turtle.Turtle()
star_turtle.color("green")
star_turtle.shape("turtle")

# executing loop 5 times for a star

for i in range(5):

    # moving turtle 100 units forward
    star_turtle.forward(100)

    # rotating turtle 144 degree right
    star_turtle.right(144)

# Create a Hexagon
```

```
hex_turtle = turtle.Turtle()
hex_turtle.shape("square")
hex_turtle.color("purple")
hex_turtle.penup()
hex_turtle.goto(150, 200)
hex_turtle.pendown()
for i in range(6):      # Draw six sides of Hexagon
    hex_turtle.forward(100)
    hex_turtle.right(60)

oct_turtle = turtle.Turtle()      # Create an Octagon
oct_turtle.shape("arrow")
oct_turtle.color("orange")
oct_turtle.penup()
oct_turtle.goto(-250, 100)
oct_turtle.pendown()
for i in range(8):      # Draw 8 sides of Octagon
    oct_turtle.forward(100)
    oct_turtle.right(45)

window1.exitonclick()
```

Output:

The output can be seen in *Figure 6.6*:

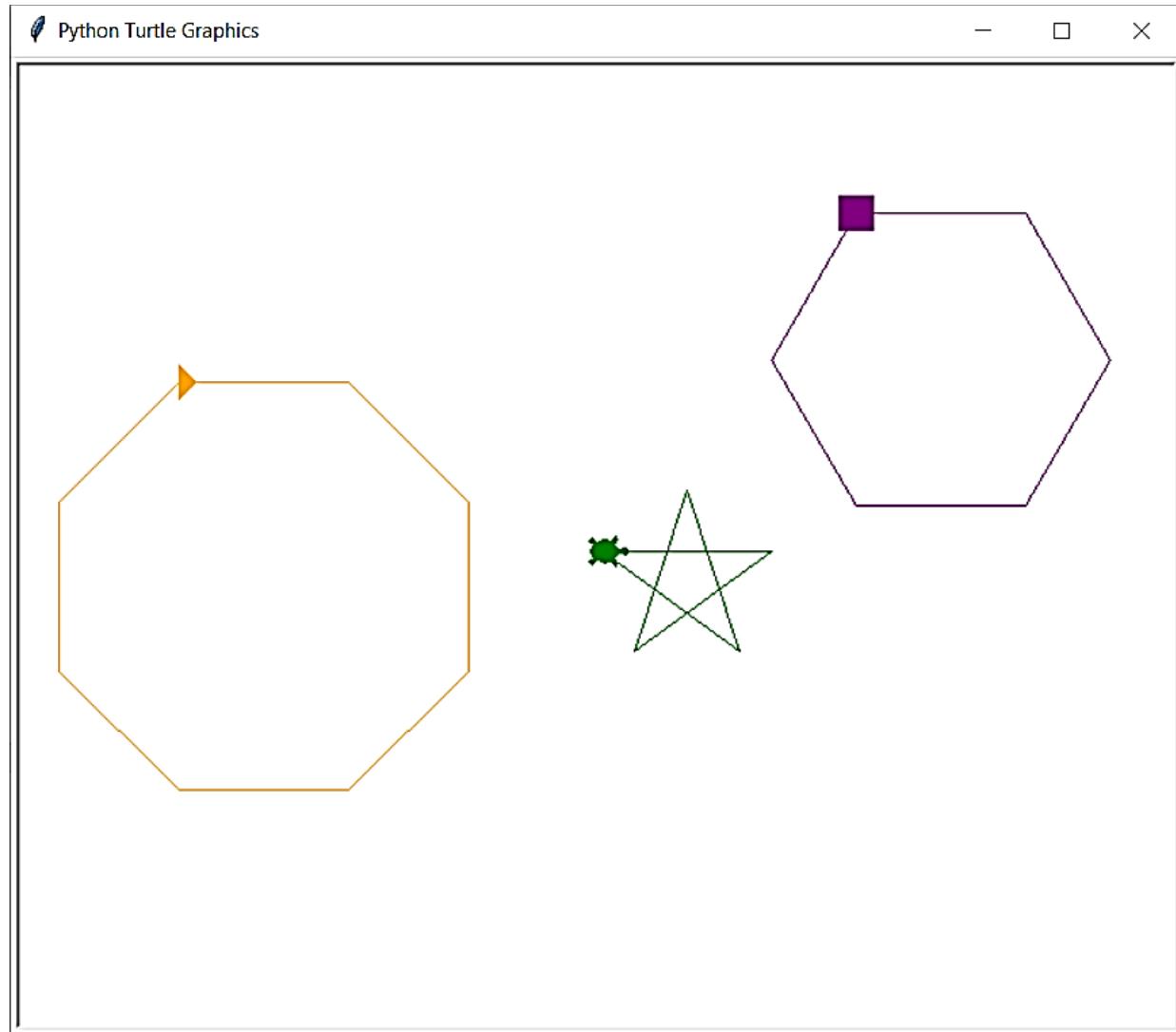


Figure 6.6: Turtle creating different shapes

To draw a star shape, we first define a turtle instance and execute a loop five times for five sides of a star. During each loop iteration, the turtle moves 100 pixels forward and turns right by 144 degrees to make an internal angle of 36 degrees inside the star shape.

Next, for creating a hexagon, we will use the for loop six times because it has six sides. Additionally, we can rotate the turtle to the right by 60 degrees (or 300 degrees left) to create angular sides. Finally, when the code flow exits the for loop, a hexagon is displayed.

Similarly, for an octagon, we use the for loop eight times for eight sides. First, the turtle moves forward by 100 units and then turns it to 45 degrees

right (or 135 degrees left) to obtain a perfect octagon.

Draw circle and oval

The turtle library's **circle()** method enables users to create circles and ovals with any radius. The number in brackets indicates the diameter of the circle. We can enlarge or reduce the circle's radius by changing the value. The code syntax of the circle method is as follows:

```
turtle.circle(rad, extent=None)
```

Here, the first parameter **rad** is the radius of the circle in pixels, whereas the other argument **extent** defines the arc-shaped portion of the circle in degrees:

```
import turtle

window1 = turtle.Screen()      # Create a new Window
Canvas

window1.bgcolor("white")

circle_turtle = turtle.Turtle()    # Draw a circle
circle_turtle.penup()
circle_turtle.goto(-100, -50)
circle_turtle.shape("triangle")
circle_turtle.color("blue")
circle_turtle.pendown()
circle_turtle.circle(100)
circle_turtle.penup()

oval_turtle = turtle.Turtle()    # Draw an oval
oval_turtle.penup()
oval_turtle.goto(200, -50)
oval_turtle.pendown()
```

```
for i in range(2):  
    oval_turtle.circle(120, 90)  
    oval_turtle.circle(250, 90)  
window1.exitonclick()
```

Output:

The output can be seen in *Figure 6.7*:

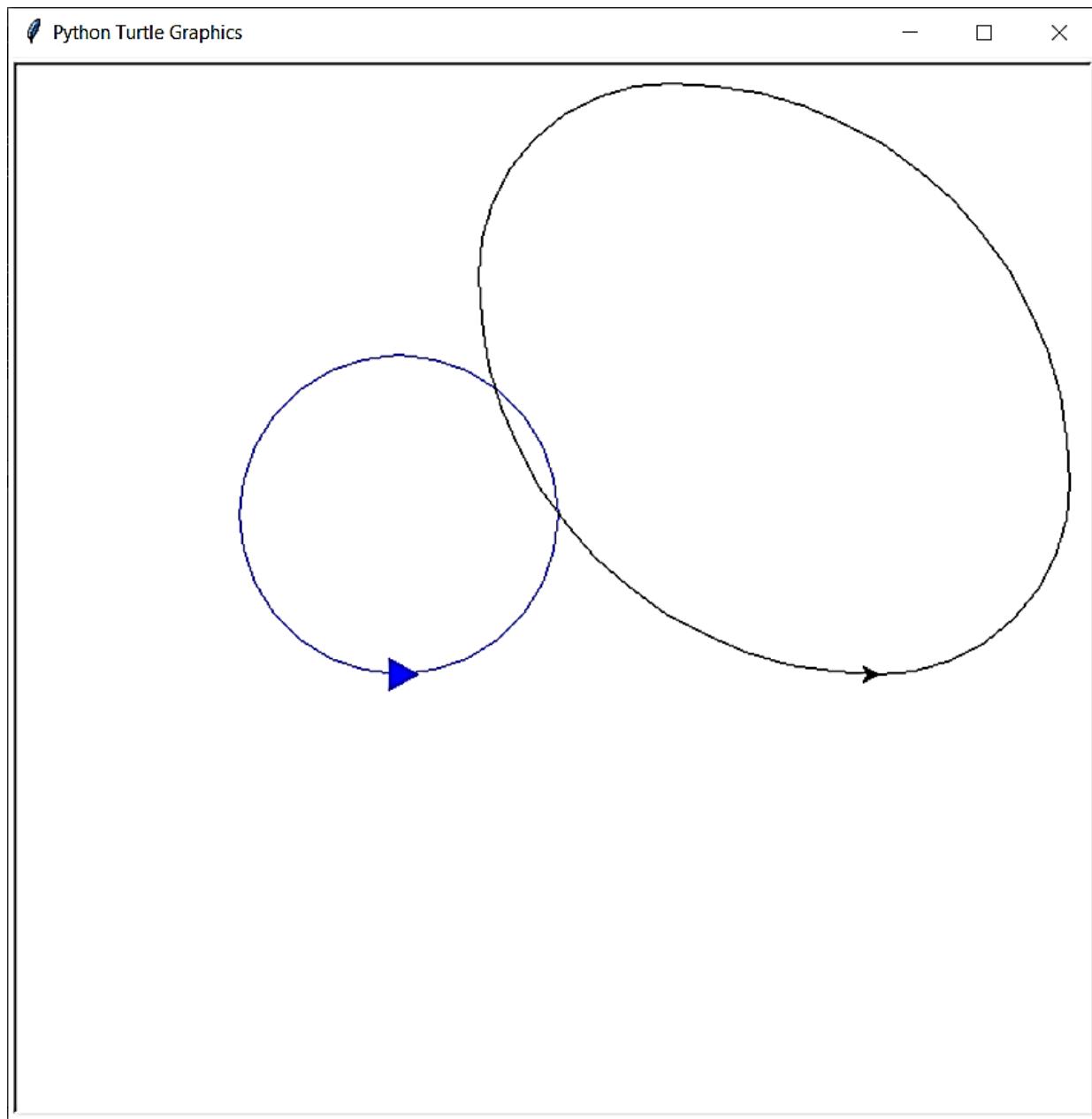


Figure 6.7: Turtle creating different circular shapes

An oval in the preceding example is created by drawing two arcs of radius 120 and 250 each while orienting both at the extent of 90 degrees. To form full oval shape, these arcs are created inside **for** loop that creates the upper and lower half of the oval.

Draw spiral

The next shape we are going to discuss is the spiral. Here, we create a colorful spiral structure with a turtle pen size set to 2 using the **pensize()** method. For adding several colors to the object, we use a list and select a different color from it each time the loop is executed. We can use turtle methods such as **forward()**, **backward()**, **circle()**, **right()**, or **left()** to create a desired pattern. Readers are advised to run the code and make their own modifications to get a new spiral every time:

```
import turtle  
a = turtle.Turtle()  
colors = ["green", "red", "yellow", "blue",  
"orange", "cyan"]  
for i in range(100):  
    a.pensize(5)  
    a.color(colors[i % 6])  
    a.forward(7+i)  
    a.right(21)  
turtle.done()
```

Output:

The output can be seen in *Figure 6.8*:

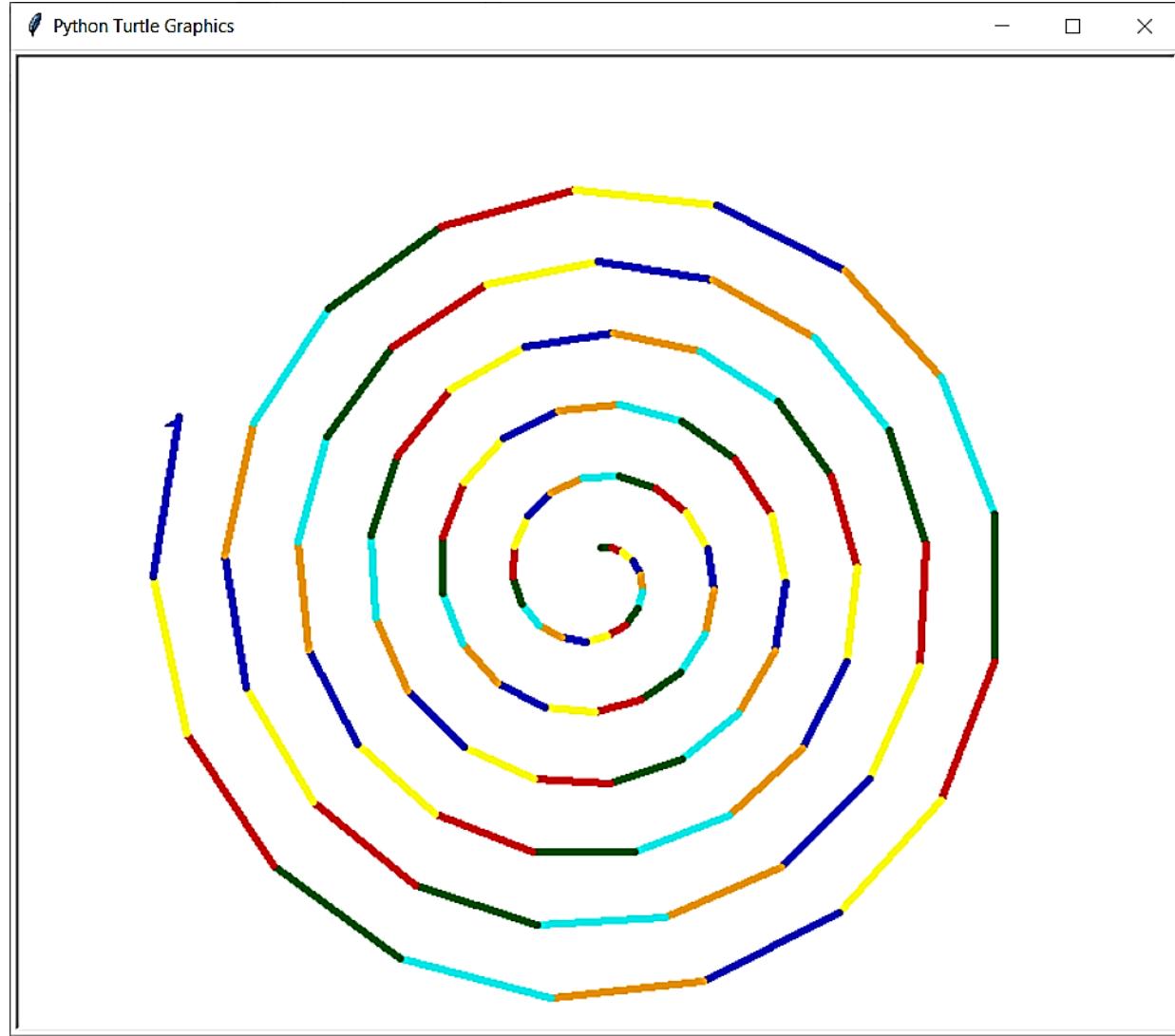


Figure 6.8: Turtle creating spiral

Fill colors in shapes

To fill the colors in the shapes drawn by the turtle, the turtle provides the following three functions:

- **fillcolor():** The **fillcolor()** method facilitates to choice of a color to fill the given shape. It applies the selected color to the specified objects by using the color name or hexadecimal value as an input parameter. Color names, such as **red**, **green**, **blue**, **cyan**, and so on, can be used in **fillcolor()**.

- **begin_fill()**: This method enables the turtle to fill the given object with the chosen color.
- **end_fill()**: This method stops the color filling in the given objects.

In the following example, we configure the speed of turtle to 5 (a bit slow speed) using **speed()** method and also set the color of the object as well as the canvas window using the previously discussed methods:

```
import turtle

# Set up the turtle screen and set the background
color to white

window = turtle.Screen()

window.bgcolor("white")

# Create a new turtle turtle1

# Set speed of turtle to 5 (normal)

turtle1 = turtle.Turtle()

turtle1.speed(5)

# Set the fill color to yellow

turtle1.fillcolor("yellow")

turtle1.begin_fill()

# Draw the circle with a radius of 150 pixels

turtle1.circle(150)

# End the fill and stop drawing

turtle1.end_fill()

# Wait for user to manually close the window

turtle.done()
```

Output:

Figure 6.9 shows the turtle slowly drawing the circle shape:

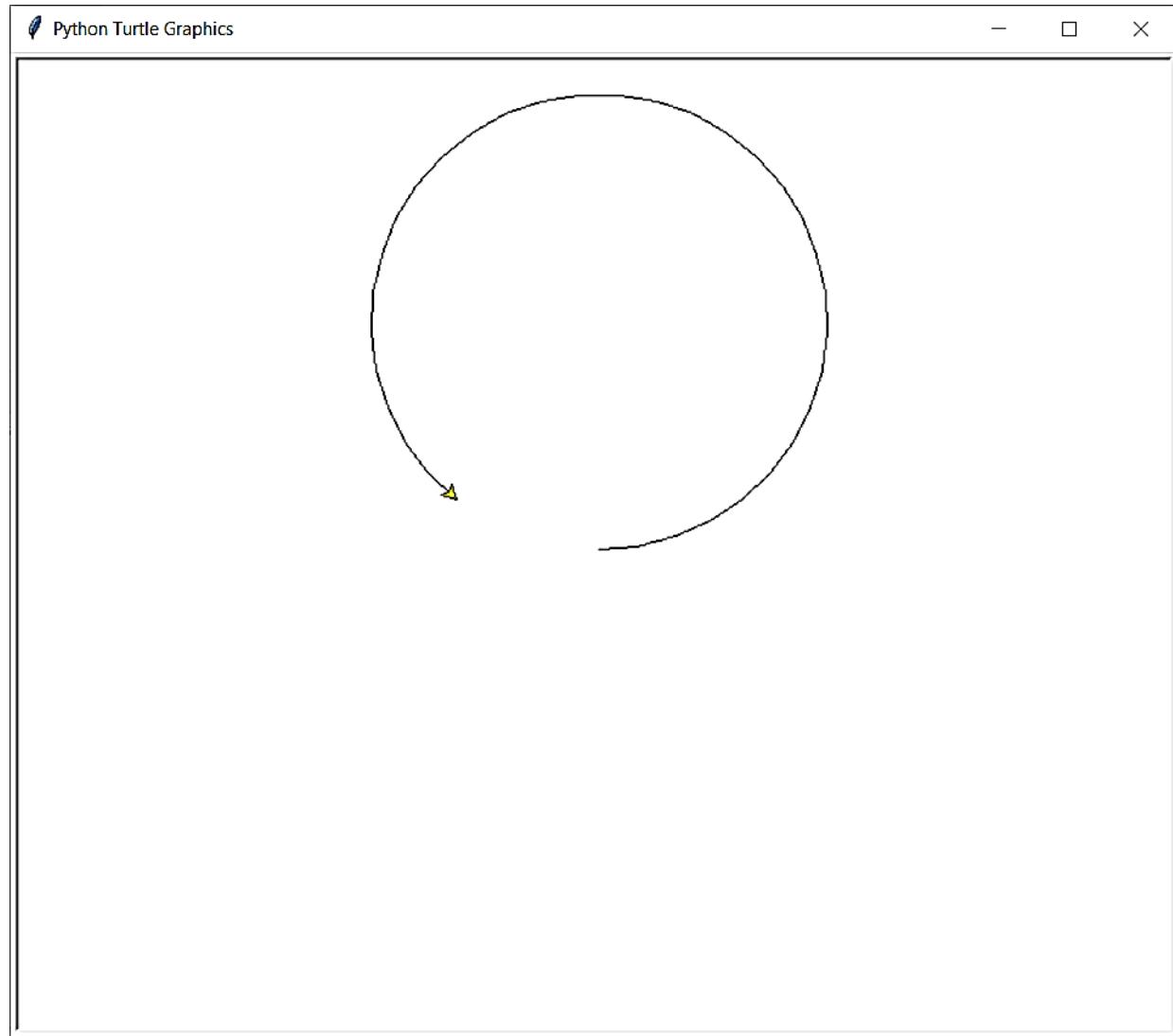


Figure 6.9: Turtle creating circle boundary

Once the figure is complete, the circle is filled with yellow color, as seen in *Figure 6.10*:

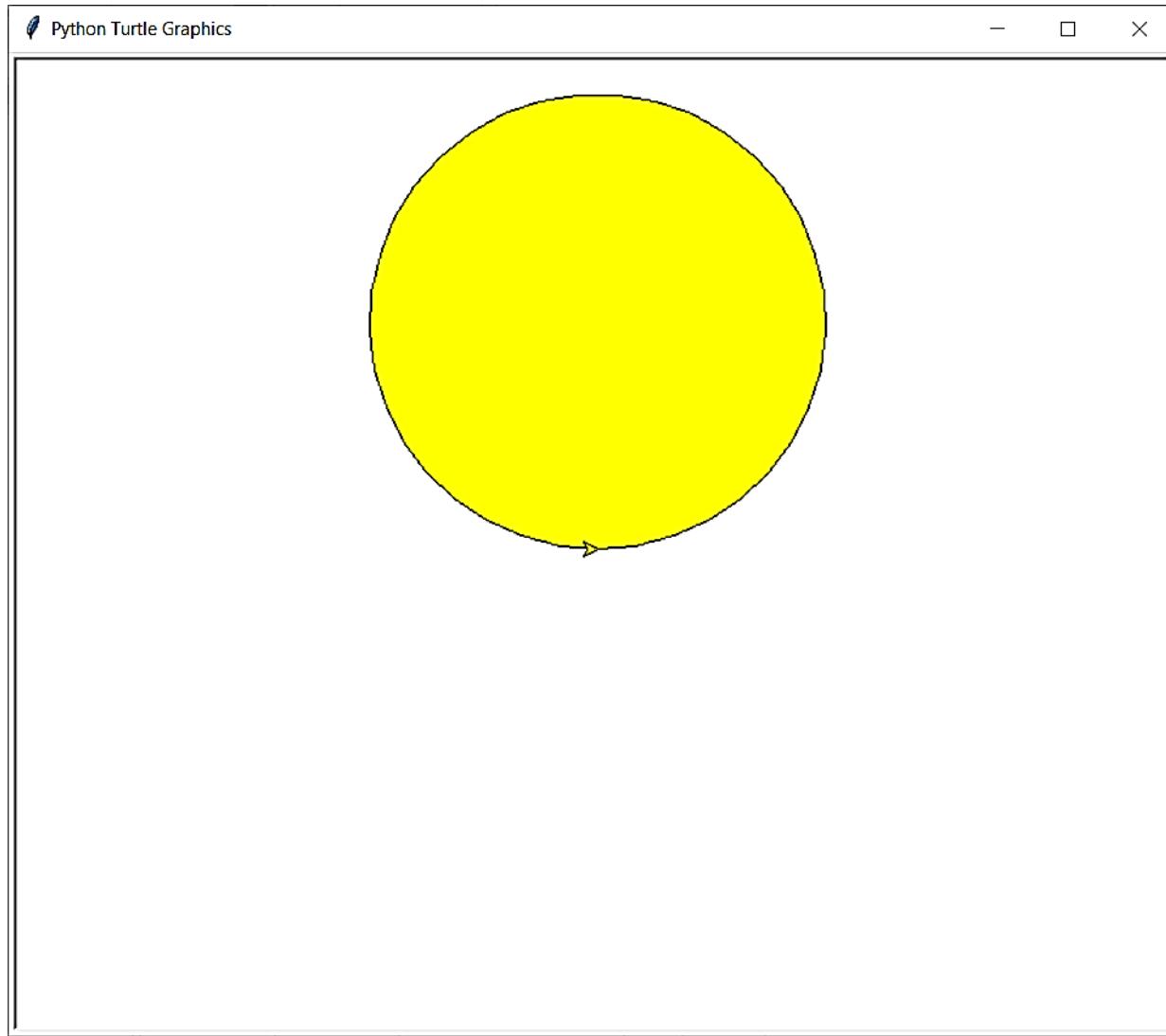


Figure 6.10: Fill color in circle

Event programming with Turtle

We must move the turtle in order to draw something on the screen. When a user presses a particular key on the keyboard or clicks or moves the mouse, the turtle can be programmed to respond by acting in a certain way. The actions performed by the turtle when the user clicks the mouse or touches a keyboard key are known as events in Python, and the handler is the function that is called in response to the occurrence of the event. There are the following two ways to perform an event:

- When a mouse button is pressed, an event is called a **Mouse Event**.

- When a key on the keyboard is pressed, an event is caused, known as a **Key Event**.

Mouse events

A mouse event is an action that is started by clicking the mouse button.

There are four ways to carry out a mouse event such as **onclick()**, **onrelease()**, **ondrag()**, and **onscreenclick()**. It is important to note that all four mouse event methods accept the same set of three parameters given as follows:

- **func**: It refers to a user-defined function having two arguments representing the coordinates of the clicked position on the canvas window.
- **btn**: The next parameter of the event methods is the number of the mouse button, such as:
 - Mouse button value 1 refers to the left button. It is also the default value used.
 - Mouse button value 2 refers to the mouse middle button.
 - Mouse button 3 refers to the right mouse button.
- **add**: The third parameter is the add value, which can be **True** (default value) or **False**. If **True**, then a new binding will be added to the event; otherwise, the previous binding will be used, if any. Here, binding refers to associating a function to an event; that is, whenever the event occurs, the associated function will be called to handle it.

Let us see the mouse event methods with their syntactic details in [Table 6.3](#). Here, we use **btn=1** in the syntax to denote an event raised on the left button of a mouse:

Event	Detail	Syntax
onclick()	When user clicks the mouse, then the turtle responds with an action.	onclick(func, btn=1, add=True)

onrelease()	Only when the clicked mouse is released, then turtle carries out an action.	onrelease(func, btn=1, add=True)
ondrag()	When the user drags the mouse, then the turtle takes a specific action.	ondrag(func, btn=1, add=True)
onscreenclick()	When the user clicks inside anywhere in the canvas window, the turtle responds by taking an action.	onscreenclick(func, btn=1, add=True)

Table 6.3: Mouse events methods

Now, let us see an example using the discussed mouse events:

```
import turtle

ws1 = turtle.Screen() # Define a turtle screen
object

tut1 = turtle.Turtle() # Define a turtle object
tut2 = turtle.Turtle() # Define a turtle object

def draw_line(i, j): # Method to draw a line
    using turtle
        tut1.showturtle() # Make turtle visible on
    screen
        tut1.fillcolor('white') # Fill white color in
    turtle
        tut1.left(90) # Turn turtle left by 90
    degrees
        tut1.forward(150) # Move turtle forward by
    150 units
```

```
def write_screen(i, j):    # Method to write
coordinates of click

    tut2.penup()        # Stop turtle from drawing

    tut2.pencolor('black')    # Change turtle pen
color to black

    tut2.goto(i, j)    # Relocate turtle cursor to
location clicked

    tut2.write(str(i) + "," + str(j))    # Write
location coordinates

def color_turtle(i, j):    # To modify turtle
properties on click

    tut1.shape('turtle')    # Change turtle shape to
'turtle'

    tut1.turtlesize(2)    # Change turtle size to 2

    tut1.fillcolor('blue')    # Fill blue color in
turtle

    tut1.onclick(draw_line)    # Call onclick() to
draw new line

def draw_drag(i, j):    # Method to draw while
dragging the mouse

    tut2.ondrag(None)    # Call ondrag() with None
object

    tut2.pendown()        # Set turtle to begin
drawing

    tut2.pencolor('red')        # Set turtle pencolor
to red
```

```
    tut2.setheading(90) # Set the turtle
orientation to 90 degrees

    tut2.goto(i, j) # Turtle will relocate to (i,
j) location

    tut2.ondrag(draw_drag) # Call ondrag() to
continue dragging

def reset_screen(i, j): # Method to reset screen
on mouse click

    ws1.resetscreen() # Call resetscreen() with
screen object

def change_color(i, j): # To change background
color of screen

    ws1.bgcolor('cyan') # Set back ground color
to 'cyan'

tut1.speed(5) # Initially set turtle speed to 5

tut1.forward(150) # Initially move turtle
forward by 150 units

tut1.onclick(draw_line, btn=1) # Bind onclick()
with draw_line()

tut1.onclick(reset_screen, btn=3) # Set right
click to reset_screen

tut1.onclick(change_color, btn=2) # middle button
change_color()

tut1.onrelease(color_turtle) # onrelease() event
with color_turtle

turtle.onscreenclick(write_screen) #
onscreenclick() write_screen()
```

```
tut2.ondrag(draw_drag) # Bind ondrag() mouse event
with draw_drag()

ws1.mainloop() # Wait until user closes the
window
```

Output:

Figure 6.11 shows the output for the first left click:

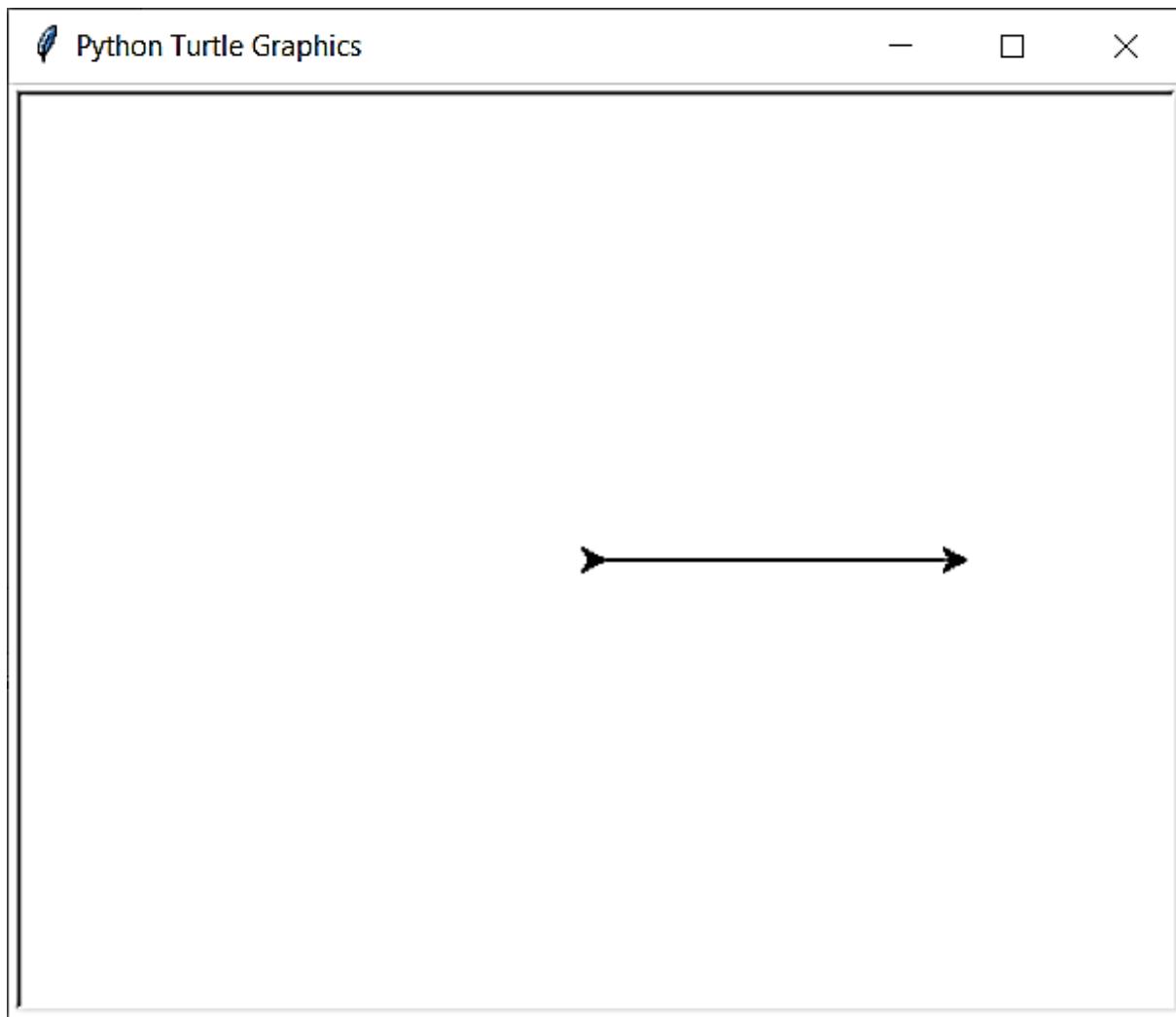


Figure 6.11: Output for first left click

Figure 6.12 shows the output after releasing mouse click:

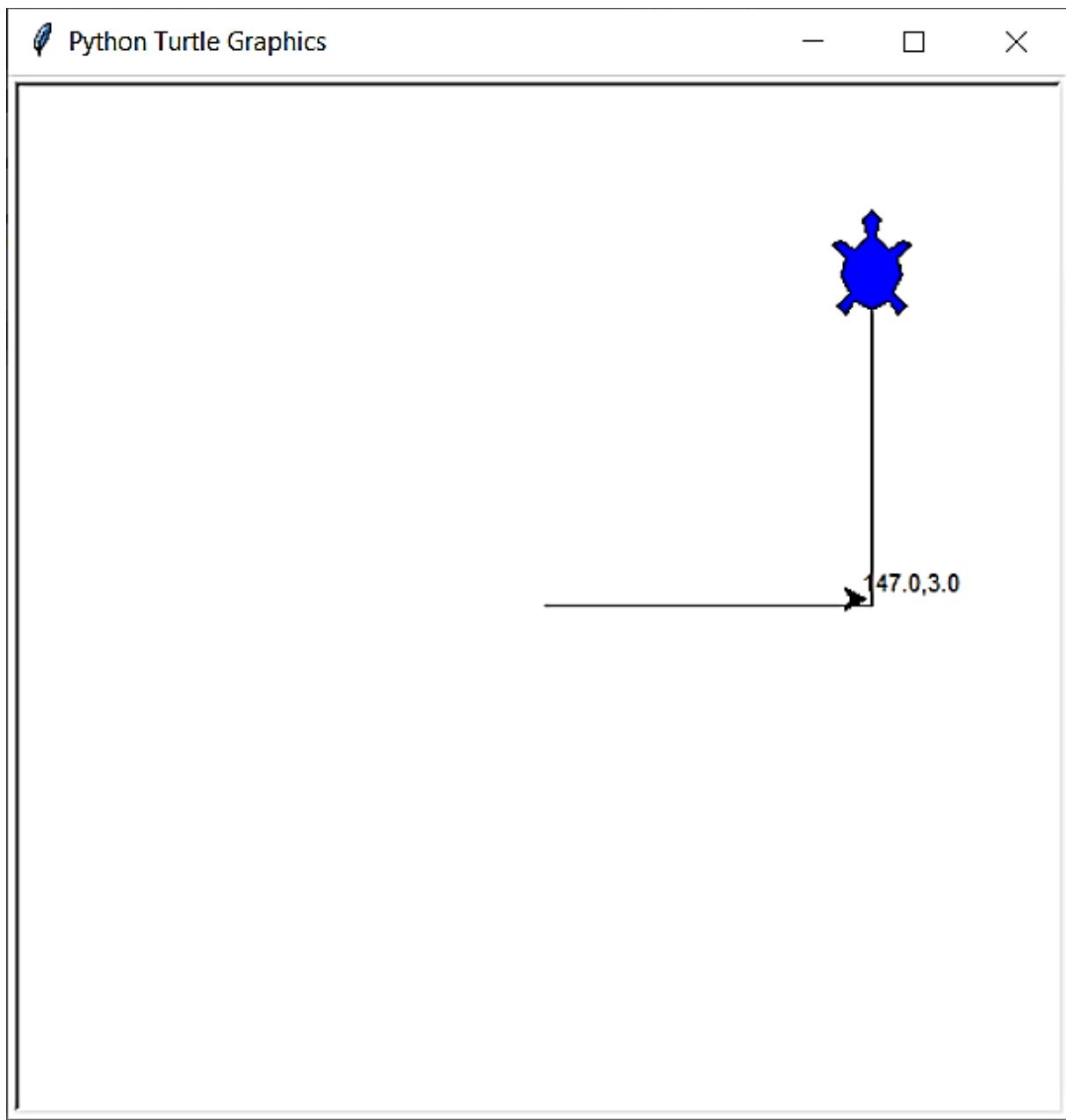


Figure 6.12: Output after releasing mouse click

[Figure 6.13](#) shows the output after clicking on screen:

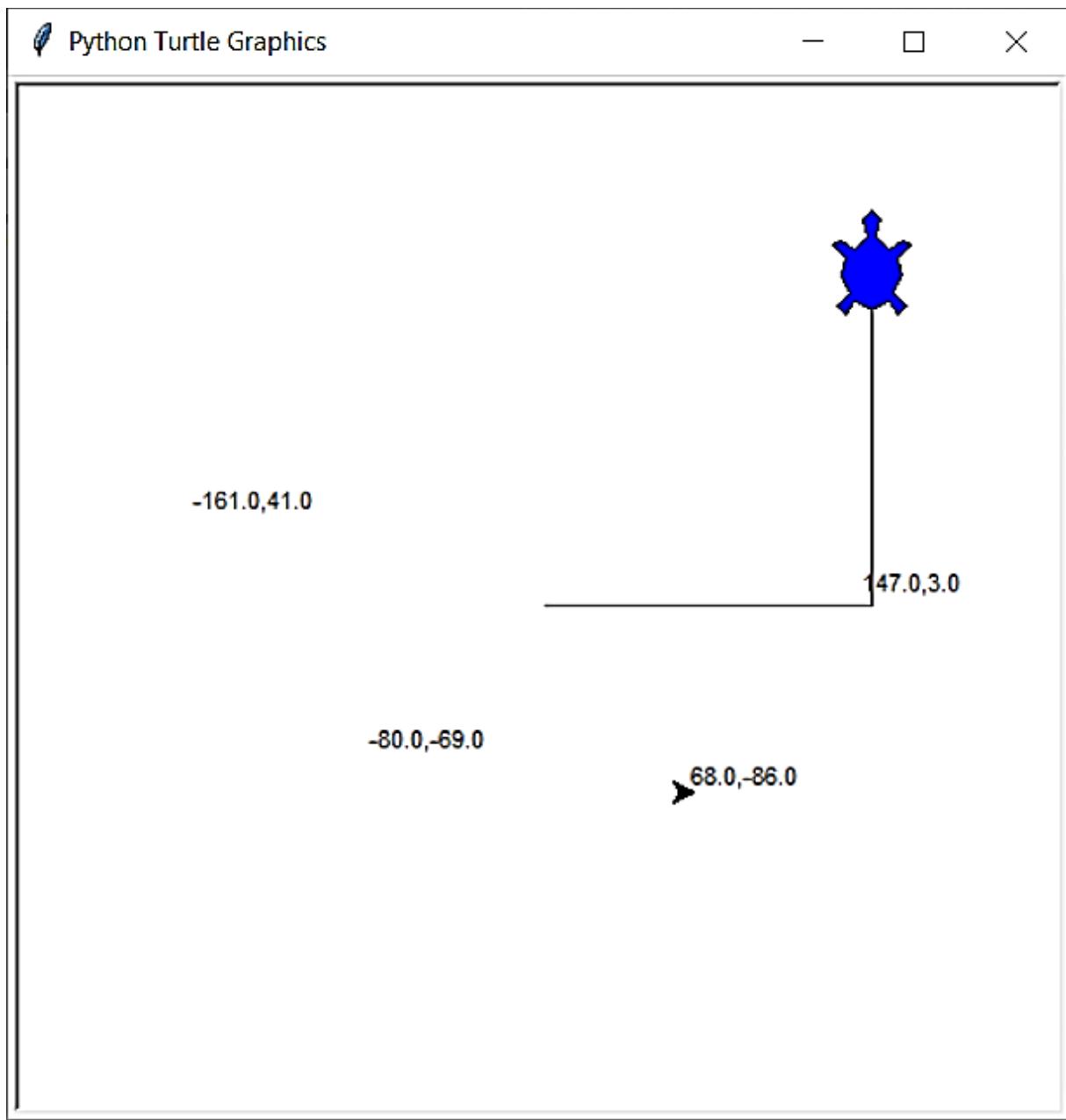


Figure 6.13: Output after clicking on screen

Figure 6.14 shows the output of dragging the mouse:

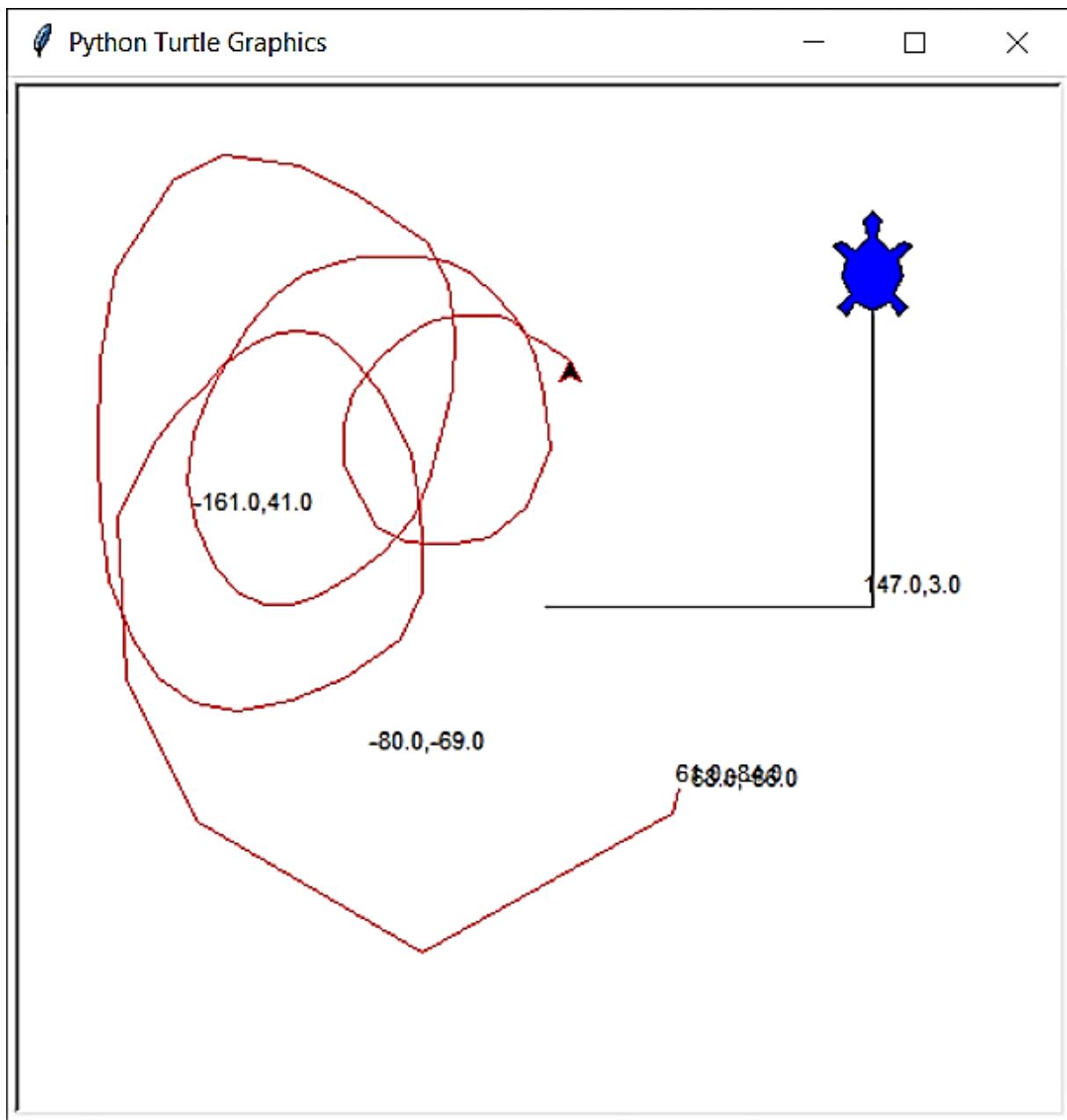


Figure 6.14: Output of dragging mouse

[Figure 6.15](#) shows the change in screen color:

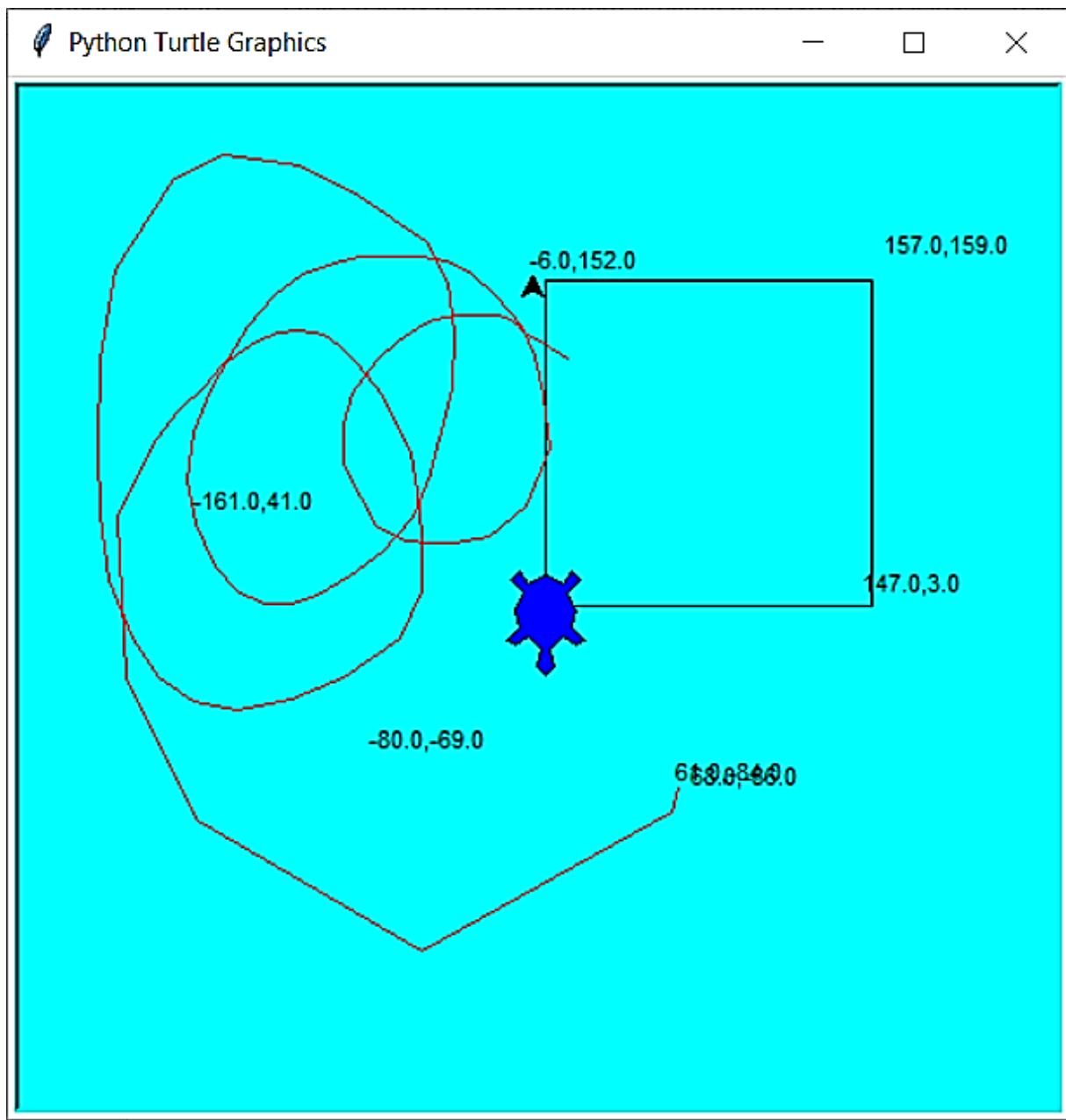


Figure 6.15: Change screen color

Figure 6.16 shows reset screen:

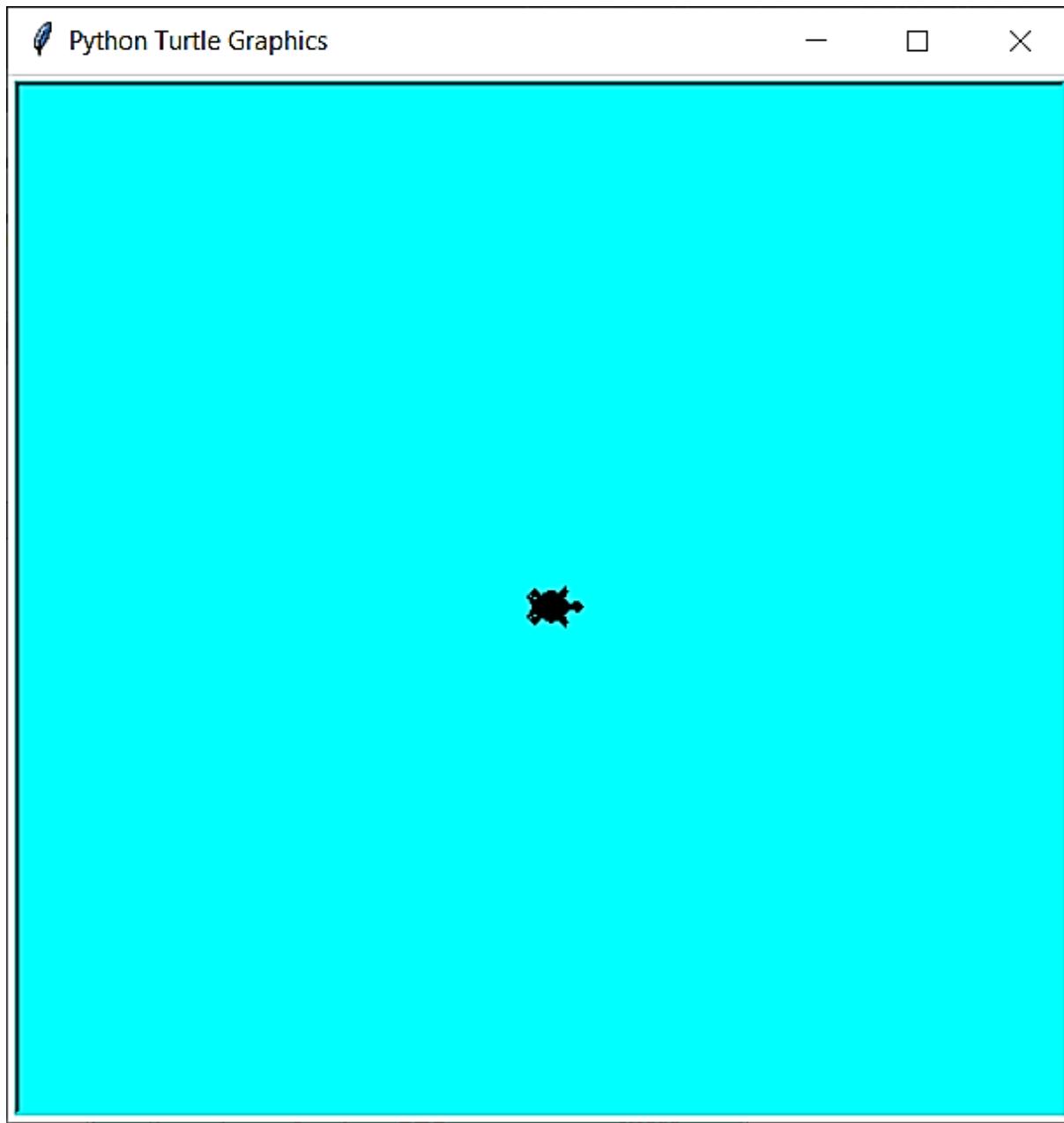


Figure 6.16: Reset screen

In [Figure 6.11](#), we can see the initial turtle moved forward to 150 units, as in line 41 of the code. When we do a left mouse click, the **onclick()** event calls the **draw_line()** function with the current mouse click coordinates. It then sets the turtle color to white with a left turn of 90 degrees and forward movement by 150 units. As soon as we release the mouse click, the **onrelease()** event is raised that calls the **color_turtle()** function with

the current mouse click coordinates. It then changes the shape of the turtle to “turtle” and fills the turtle color to blue, as shown in output [Figure 6.12](#).

When we click anywhere on the screen, the `onscreenclick()` window event calls `write_screen()` method with the turtle to print the current screen coordinates as depicted in [Figure 6.13](#) of output. Next, as we drag the turtle with the mouse on the screen, the `ondrag()` event calls the `draw_drag()` method that exactly draws the way we move the turtle on screen, as given in [Figure 6.14](#) of output. Now, in line 44, as we click on the turtle with the mouse middle button, it raises the `onclick()` event, which changes the screen color to cyan by further calling the `change_color()` method in [Figure 6.15](#). Finally, when we right-click the turtle, line 43 executes and raises the `onclick()` event and resets the screen by calling the `reset_screen()` method.

Key events

When a user presses and releases a key on a keyboard, specific actions are initiated. These actions are known as keyboard events. Various methods are used to handle different key events in Python turtle. All these event methods accept two common parameters as described:

- **func:** It refers to the name of a function without any arguments. It is called when the related key event is raised by the user through the keyboard.
- **key:** It refers to the name of the key in the keyboard to which the key event will be associated. We can refer to keys on the keyboard by their character code in string format, such as “a”, “b”, “x”, and so on, or by their symbolic names. Some of the examples of symbolic names commonly used are Cancel (the Break key), BackSpace, Tab, Return (the Enter key), Shift_L (any Shift key), Control_L (any Control key), Alt_L (any Alt key), Pause, Caps_Lock, Escape, Prior (Page Up), Next (Page Down), End, Home, Left, Up, Right, Down, Print, Insert, Delete, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, Num_Lock, and Scroll_Lock.

Let us discuss the methods for key events in the turtle. Refer to [Table 6.4](#):

Key event	Syntax	Details
<code>listen()</code>	<pre data-bbox="545 297 1090 382">turtle.listen(xdummy=None, ydummy=None)</pre> <p data-bbox="545 397 1090 523">Here, Dummy arguments are provided in order to be able to pass <code>listen()</code> to the onclick method.</p>	<p data-bbox="1132 297 1428 424">To be able to register or bind key events the turtle screen must have the focus.</p> <p data-bbox="1132 424 1428 830">To set focus on the Turtle screen, the keyboard event method is used in conjunction with the turtle module' <code>listen()</code> method which takes dummy arguments.</p>
<code>onkey()</code> <code>onkeyrelease()</code>	<pre data-bbox="545 846 1057 889">turtle.onkey(func, key)</pre> <pre data-bbox="545 903 1067 988">turtle.onkeyrelease(func, key)</pre>	<p data-bbox="1132 846 1428 1349">The <code>onkey()</code> and <code>onkeyrelease()</code> methods call the user-defined function bind to them and perform the action defined inside the function when the specified key is pressed. Both require same set of parameters.</p>

Key event	Syntax	Details
onkeypress()	<code>turtle.onkeypress(fun, key=None)</code>	The only difference between the two methods is that the second parameter is the <code>onkey()</code> method is optional whereas it is required in the <code>onkey()</code> method. Both the <code>onkey()</code> method and the <code>onkeypress()</code> method carry out comparable actions. If the key is not specifically indicated in the <code>onkeypress()</code> method, any key will be used to start the action.

Table 6.4: Key events methods

Now, we shall see an example using all the key events to perform specific actions:

```
import turtle

ws1 = turtle.Screen() # Create a screen object
turtle = turtle.Turtle() # Create a turtle object
turtle.speed(0) # Set the turtle speed to 0
turtle.showturtle() # Make the turtle visible on screen
```

```
def up(): # Define method to set turtle settings
    turtle.setheading(90)
    turtle.forward(100)

def left(): # Define method to set turtle
orientation
    turtle.left(90)
    turtle.forward(100)

def change_blue(): # Define method to set turtle
properties
    turtle.shape('circle')
    turtle.color("blue")

def inc_size():
    size = turtle.turtlesize()
    newsize = tuple(2 * elem for elem in size)
    print(newsize)
    turtle.turtlesize(*newsize)

# Note: The turtlesize() function does not accept
# a tuple of three values as it accepts only int or
# float, so we need to unpack the tuple using a *
# as we pass the tuple to the function to obtain
# three separate values.

ws1.listen() # To set focus on turtle screen &
register key events
```

```
ws1.onkey(up, 'Up')  # Bind up() method with up
direction key

ws1.onkey(left, 'Left')  # Bind left() with left
direction key

ws1.onkeyrelease(change_blue, 'b')  # Bind
change_blue() to 'b' key

ws1.onkeypress(inc_size, '+')  # Increase size of
turtle '+' key

ws1.mainloop()
```

Output:

Figure 6.17 shows the output for the key **Up** press:

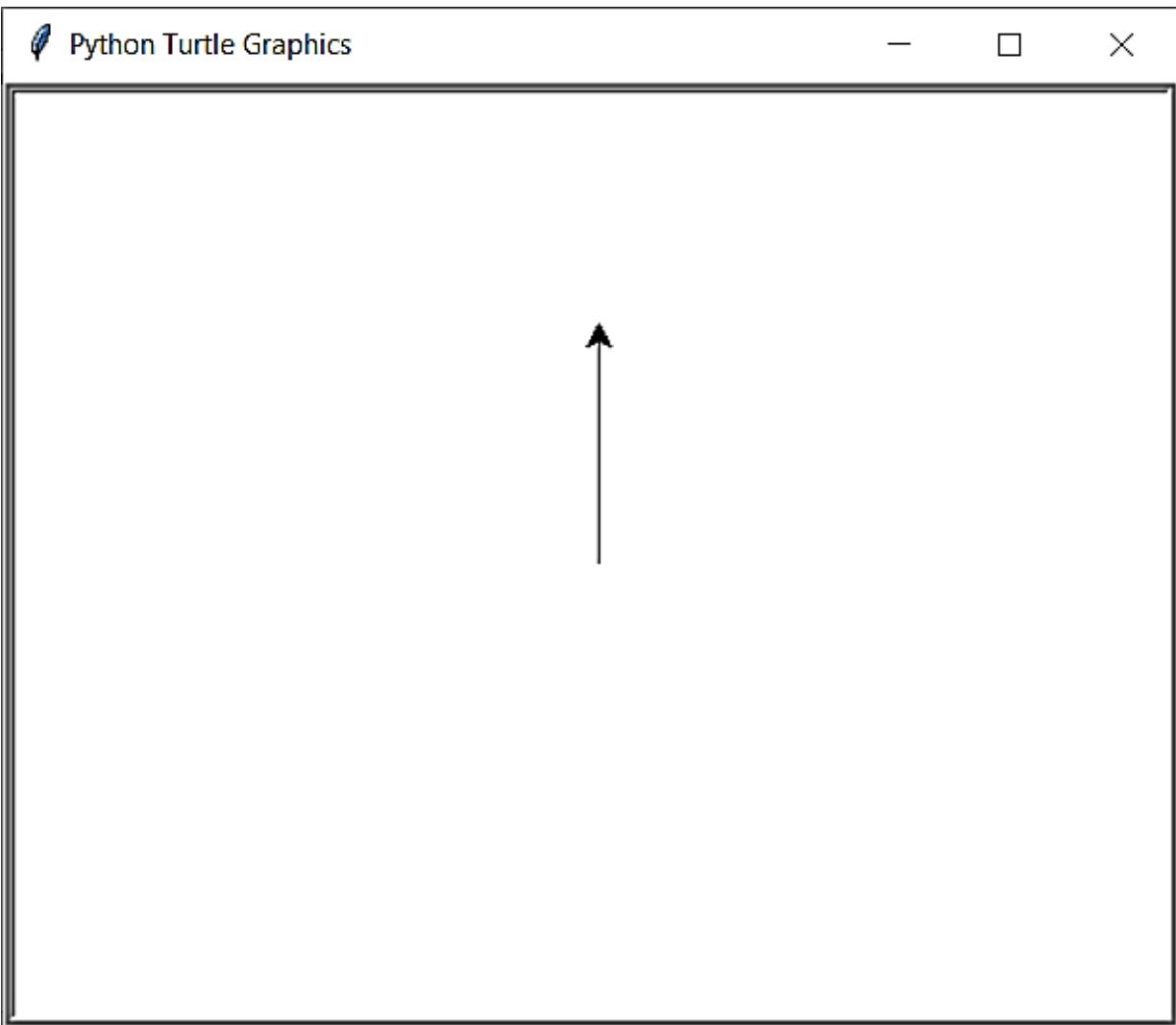


Figure 6.17: Output for key “Up” press

[Figure 6.18](#) shows the output for the key **Left** press:

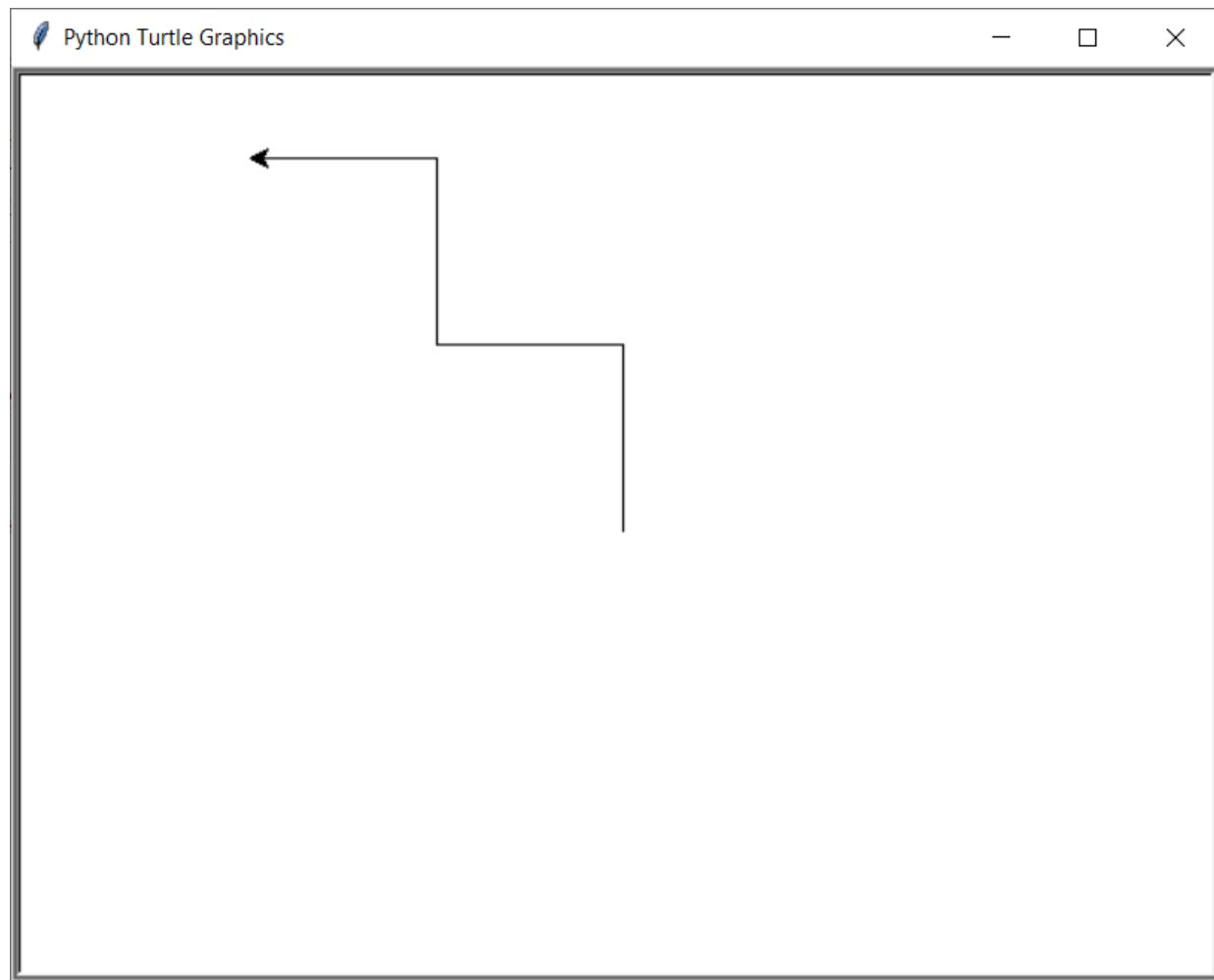


Figure 6.18: Output for key “Left” press

Figure 6.19 shows the output for key **b** press:

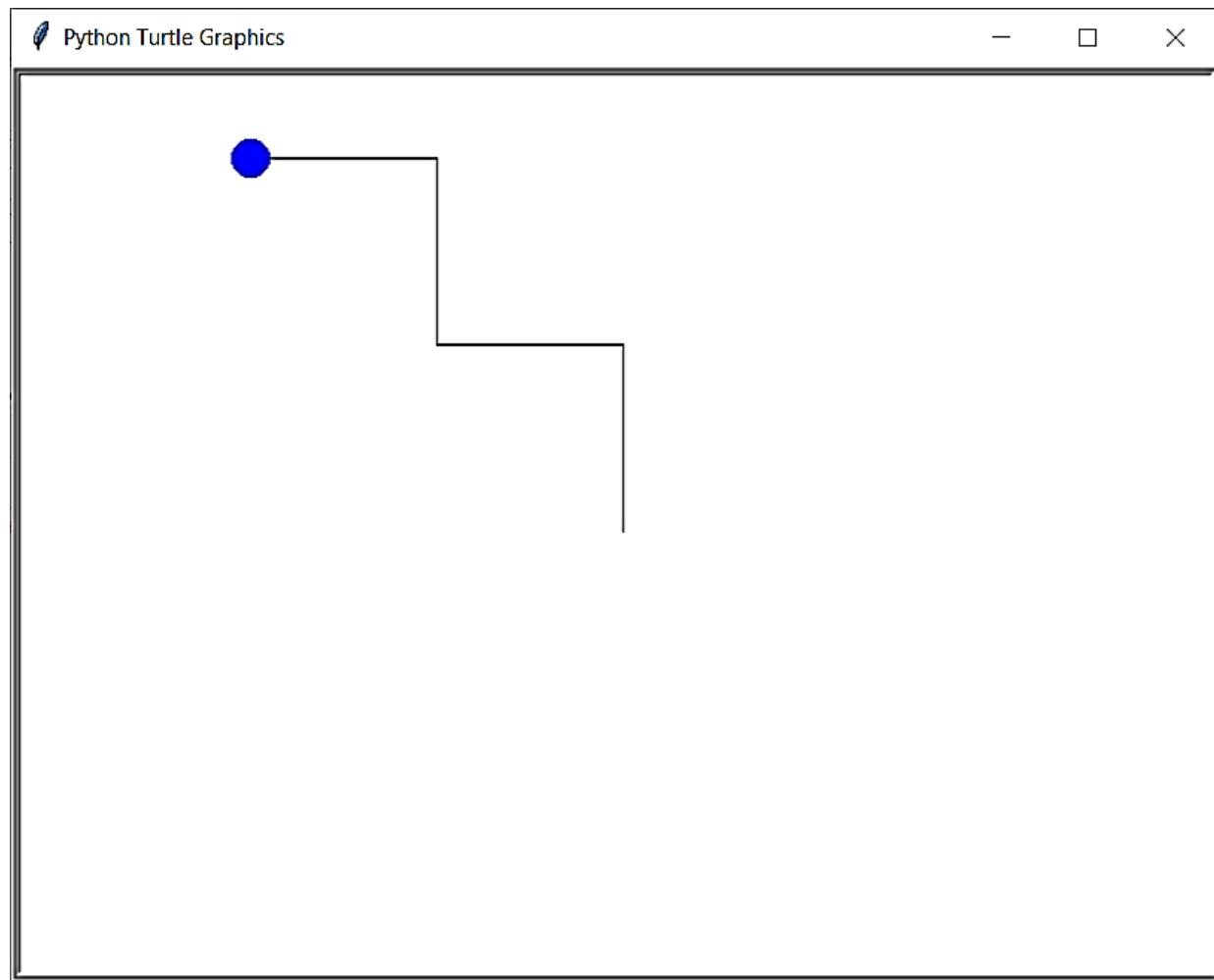


Figure 6.19: Output for key “b” press

Figure 6.20 shows the output for key + press:

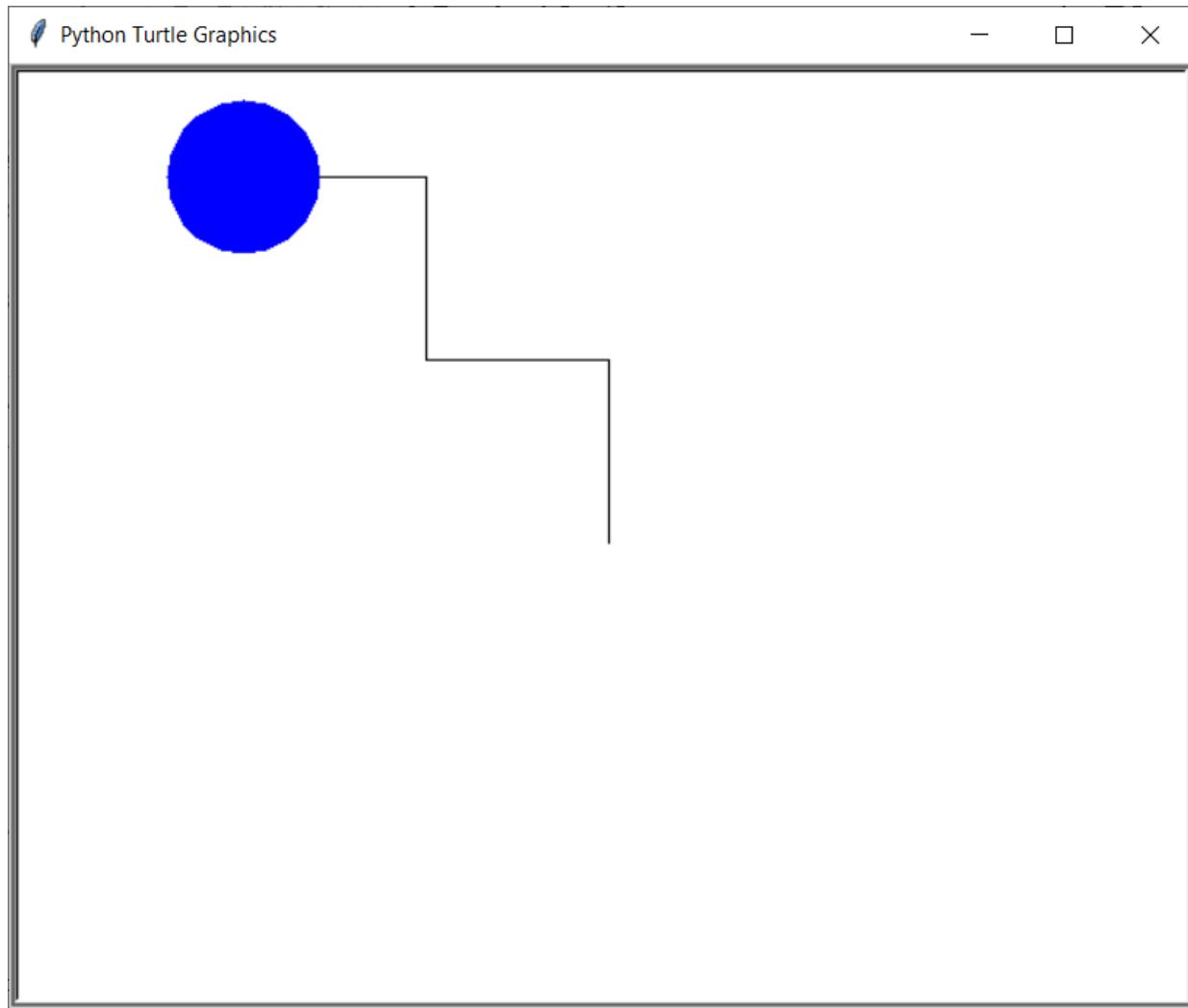


Figure 6.20: Output for key “+” press

In line 27, we need the call to the `listen` method of the window to make our keypresses noticeable. The handler functions are named as `up()`, `left()`, `change_blue()`, and `inc_size()`. These functions are registered to different key presses, as shown in lines 28 to 31. On pressing the upwards arrow key on the keyboard, it calls the `up()` method. Similarly, the left arrow key press calls the `left()` method, the `b` key press calls the `change_blue()` method, and `+` key press calls the `inc_size()` method.

Conclusion

This chapter introduced users to the creation of graphics and animations through logo-based programming using the turtle library in Python. After

going through this chapter, users can use the turtle library to extend their imagination beyond limits to create various games. This chapter lays a firm foundation for further GUI programming by creating different shapes, animated objects, and interaction games in Python.

Points to remember

- Python comes with a built-in module called Turtle that functions like a virtual canvas on which we may create beautiful images and forms.
- The turtle module is first imported, and after that, a window can be constructed if desired. Using the turtle library in Python, we can draw different shapes by moving the turtle object in different directions on the screen canvas.
- The Python Turtle canvas is divided into four quadrants by an x- and y-axis, and the turtle always starts at (0, 0), which is the exact center of the canvas.
- The actions performed by the turtle when the user clicks the mouse or touches a keyboard key are known as events in Python, and the handler is the function that is called in response to the occurrence of the event.

Exercise

Attempt the following project.

Sample project with solutions

Create a two-player box completion game using Turtle in Python. The player who completes the maximum boxes and stays on the window screen becomes the winner. The player who first moves out of the window screen (that is, touches the window boundary first) is the loser.

The project has two players: a red turtle and a blue turtle. The user must press **r** first to initiate the red turtle and then press key **b** to initiate the blue turtle. After this, the user must press the *Enter* key to begin the game. As the game begins, both turtles begin to draw edges and create square boxes in

different directions. Each turtle can move in random directions based on the value of the variable **flag**. Moreover, a turtle can repeatedly draw previously created boxes by itself or another turtle. The turtle that first touches the window boundary is the loser, whereas the one who creates maximum boxes inside the window screen without touching the screen edge is the winner:

```
import random
import turtle

ws1 = turtle.Screen()
ws1.screensize(canvwidth=320, canvheight=320,
bg="white")
pos = 0
Red = turtle.Turtle()
Blue = turtle.Turtle()

Red.fillcolor("red")    # set colors of turtles
Blue.fillcolor("blue")

# function to check whether turtle
# is in Screen or not
def isInWindow(wind, turt):
    # Set edge values as boundaries based on screen
    # size of window
    leftedge = -320
    rightedge = 320
    topedge = 320
```

```
bottomedge = -320

# getting the current position of the turtle
turtle_X = turt.xcor()
turtle_Y = turt.ycor()

# variable to store whether in screen or not
inside = True

# condition to check whether in screen or not
if turtle_X > rightedge or turtle_X < leftedge:
    inside = False

if turtle_Y > topedge or turtle_Y < bottomedge:
    inside = False

# returning the result
return inside

# function to check whether both turtle have
# different position
def sameposition(Red, Blue):
    if Red.pos() == Blue.pos():
        return False
    else:
        return True

def red_up():
    global pos
```

```
# set pencolor as red
Red.pencolor("red")
# set pensize as 5
Red.pensize(5)
# set tutleshape as turtle
Red.shape('turtle')
pos = Red.pos()

def blue_up():
    # Turtle Blue initialization and set pencolor
    # as blue
    Blue.pencolor("blue")
    Blue.pensize(5)      # set pensize as 5
    Blue.shape('turtle')      # set tutleshape as
    turtle
    Blue.hideturtle()      # make the turtle
    invisible
    Blue.penup()      # don't draw when turtle moves
    # Set the turtle to a location 100 pixels away
    # from Red turtle
    Blue.goto(pos[0] + 100, pos[1])
    Blue.showturtle()      # make the turtle visible
    Blue.pendown()      # draw when the turtle
    moves

def game():
```

```
# To check whether turtles are inside screen or
not

rT = True      # To store red turtle location
bT = True      # To store blue turtle location

# loop for the game

while rT and bT and sameposition(Red, Blue):

    flagRed = random.randrange(0, 2) # flag
value for Red turtle          # change angle for Red
turtle to 90 to complete the box

    angleRed = 90

    # condition for left or right based on flag

    if flagRed == 0:

        Red.left(angleRed)

    else:

        Red.right(angleRed)

    Red.forward(50)    # draw for Red box of
edge of length 50

    flagBlue = random.randrange(0, 2)    # flag
value for Blue turtle

    angleBlue = 90    # set angle for Blue to 90
to complete box

    # condition for left or right turn based on
flag value

    if flagBlue == 0:

        Blue.left(angleBlue)
```

```
        else:
            Blue.right(angleBlue)
            Blue.forward(50)    # draw for Blue
            # checking whether turtles are inside
            window or not
            bT = isInWindow(ws1, Blue)
            rT = isInWindow(ws1, Red)
            # condition check for draw or win
            if rT == True and bT == False:
                # writing results
                Red.write("Red Won", True, align="center",
                          font=("arial", 15, "bold"))
            elif bT == True and rT == False:
                # writing results
                Blue.write("Blue Won", True,
                           align="center",
                           font=("arial", 15, "bold"))
            else:
                # writing results
                Red.write("Draw", True, align="center",
                          font=("arial", 15, "bold"))
                Blue.write("Draw", True, align="center",
                          font=("arial", 15, "bold"))
```

```
ws1.listen() # Helps to set foocus on turtle  
screen  
  
ws1.onkey(red_up, 'r') # call red_up() method with  
'r' key  
  
ws1.onkey(blue_up, 'b') # call red_left() with 'b'  
key  
  
ws1.onkey(game, 'Return') # Press Enter key to  
start game  
  
ws1.mainloop()
```

Output:

Figure 6.21 features the output for two turtle players:

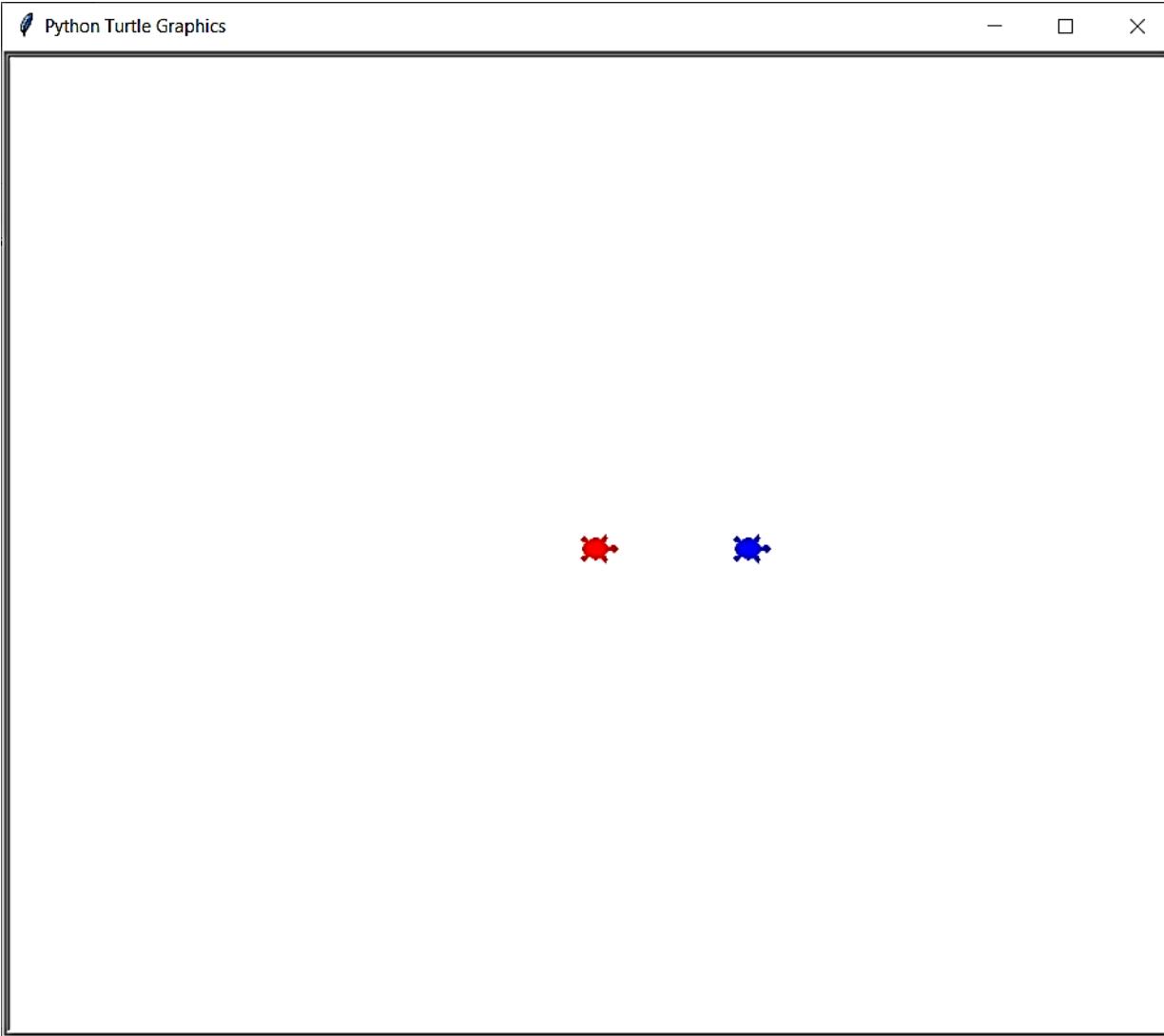


Figure 6.21: Output for two turtle players

[**Figure 6.22**](#) shows the screen when you press Enter to start play:

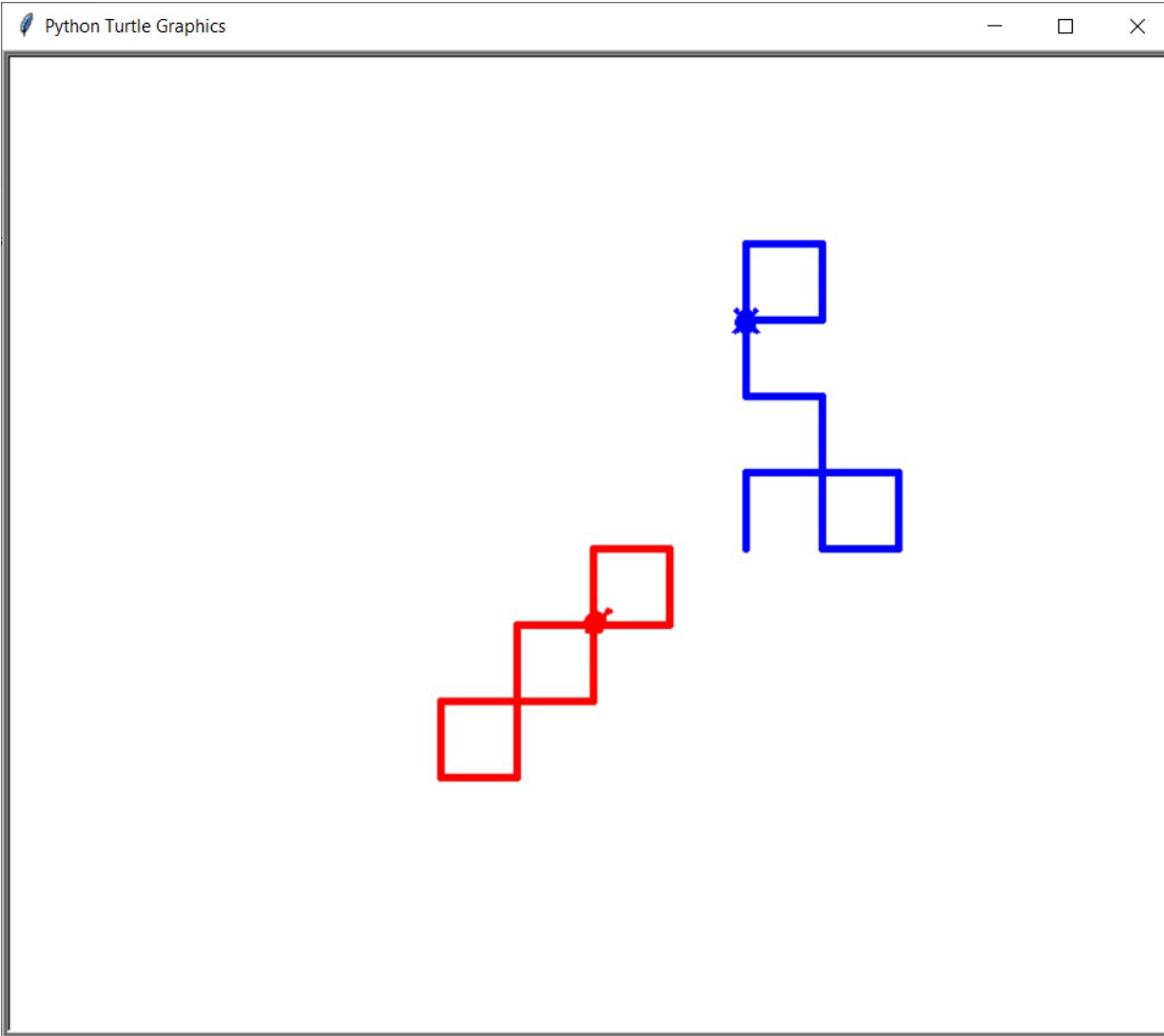


Figure 6.22: Press Enter key to start play

[Figure 6.23](#) features the result after one play:

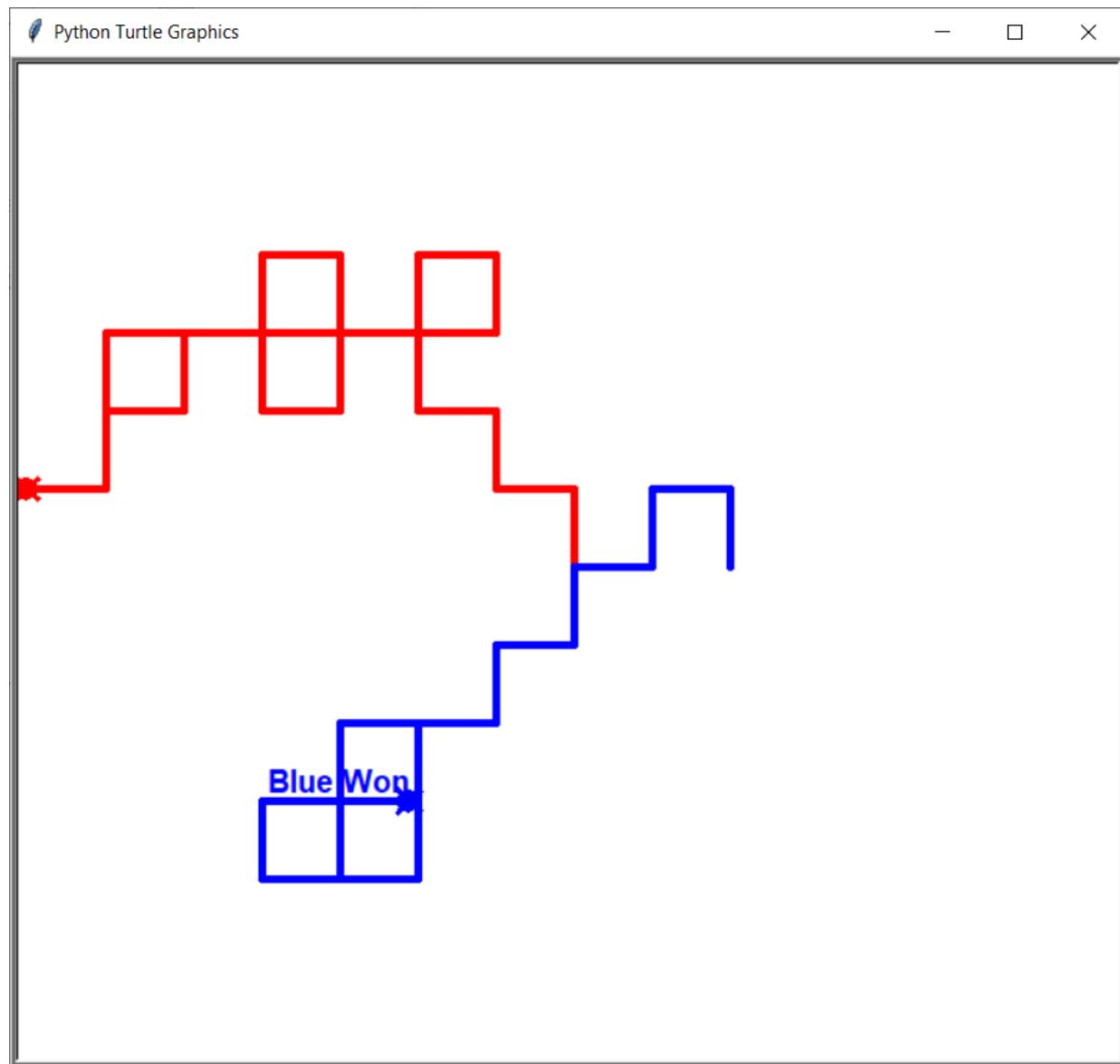


Figure 6.23: Result after one play

Figure 6.24 features the result after playing again:

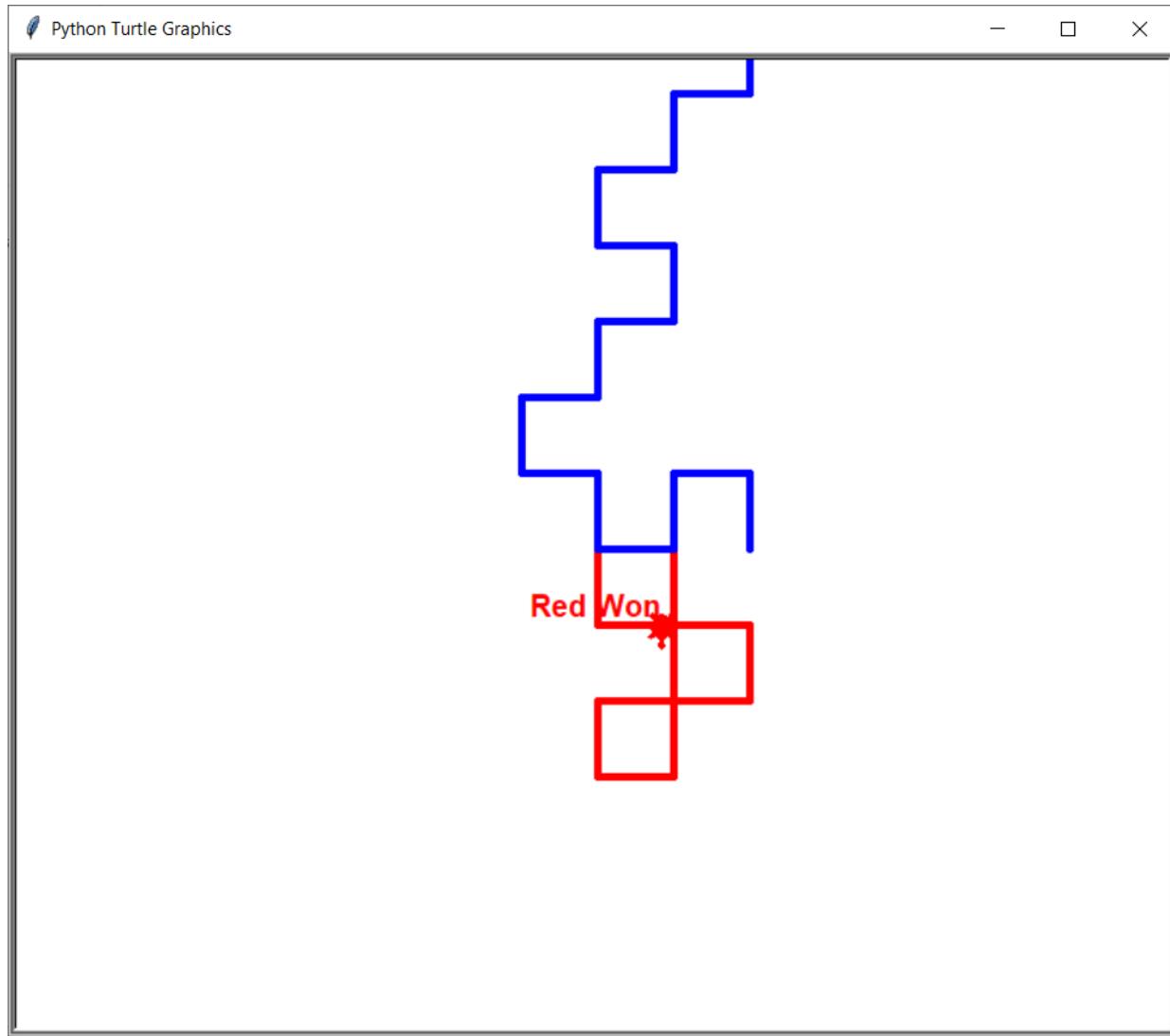


Figure 6.24: Result after playing again

Practice project

Create an animated Beating Heart using Turtle and Python. You can use the desired color to fill the heart.

Hint: Use necessary event programming to display the animated effect of the heartbeat by repeatedly increasing and decreasing the size of the figure.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 7

Database Handling Using SQLite

Introduction

Data storage and its management is the most crucial aspect of programming large real-world projects. This chapter introduces the concept of data persistence using the SQLite3 library in Python, which provides the necessary support for creating and managing a stable database for an application. Various topics related to **Database Management System (DBMS)**, beginning from SQLite installation along with types of queries (DDL, DML, and DQL), constraints, different clauses, Aggregate functions, parametrized queries, sub-queries, and so on, are discussed with numerous examples to help readers understand the need and application of a database system in a software project.

Structure

In this chapter, we will discuss the following topics:

- Introduction to data and database
- SQLite for database handling
- Datatypes in SQLite
- Exception handling tasks
- Database management with SQLite
- Commands in SQLite
- Joins in SQLite
- Parameterized queries and sub-queries
- BLOB and DATE TIME in SQLite

Objectives

The purpose of this chapter is to provide readers with a clear understanding of the database handling concepts using the `sqlite3` module in Python programming. By the end of this chapter, users will be able to create and manage a stable and persistent database system for their applications.

Introduction to data and database

Data refers to raw facts and figures without any context. Simple values such as 15, 1999, 230, 14, 90, and so on do not give much information to analyze. Therefore, we can say that the data itself is not that useful and needs a relevant context for further exploration. Once data is provided with a context, aggregated, and analyzed, it becomes useful for the organizational decision-making process.

Figure 7.1 features the data processing steps to derive knowledge and wisdom:

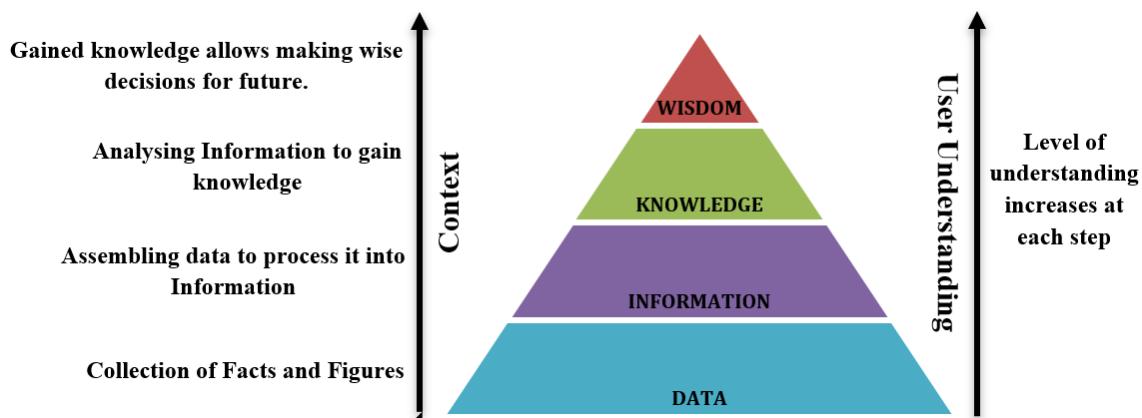


Figure 7.1: Data processing steps to derive knowledge and wisdom

In the preceding pyramid diagram in *Figure 7.1*, we can observe the process of converting raw data bits and pieces into a processed form of information, which is later used to derive knowledge by deeply analyzing it. The knowledge gained from this information helps users to forecast the future and thereby make wise decisions for better future prospects.

Relational versus non-relational database

A database is a structured collection of relevant data. It is a well-organized collection because all data in a database is described and linked to other data. Separate databases should be used to manage unrelated information because a

database should only include related data. For instance, a database with employee information should not simultaneously have stock price information of the company. Data can be stored, managed, organized, and retrieved using a DBMS.

Databases are used in most modern applications, and an operational database system will store much of the data an application needs to function, keeping the data organized and allowing users to access the data. There are various types of databases, but the most common are relational and non-relational databases.

Relational databases

Information is stored in tables in a relational database, often known as a **Relational Database Management System (RDBMS)**. Often, these tables have shared information between them, causing a relationship to form between tables. A relational database derives its name from this interconnectivity of tables.

Structured Query Language (SQL) is the most popular language for communicating with relational database systems. MySQL, SQLite, Microsoft SQL Server, and Oracle are a few examples of RDBMS. Each data table in an RDBMS schema has a primary key that helps to uniquely identify each record. A table may possess one or more foreign keys to connect with the primary keys of other tables. These common columns between tables help in joining data from multiple tables and design queries across tables.

Figure 7.2 features a sample database schema for a **Company** database having three tables, namely, **Employee**, **Department**, and **Project**. Here, **Emp_ID**, **Dept_ID**, and **Proj_ID** are the primary keys of **Employee**, **Department**, and **Project** tables, respectively. In order to join the data of Employee with Department and Project tables, the foreign key columns **Dept_ID** and **Project_ID** are included in the **Employee** table.

Employee Table

Emp_ID (P.K.)	Emp_Name	Dept_ID (F.K.)	Project_ID (F.K.)	Salary
---------------	----------	----------------	-------------------	--------

Department Table

Dept_ID (P.K.)	D_Name	D_Location	Dept_Manager
----------------	--------	------------	--------------

Project Table

Proj_ID (P.K.)	Proj_Name	Proj_Manager	Proj_Duration
----------------	-----------	--------------	---------------

Note: P.K. refers to Primary Key and F.K. refers to Foreign Key

Figure 7.2: Company database schema with three sample tables

Non-relational databases

Any database that does not use relational databases' tables, fields, and columns for structured data is referred to as a non-relational database, often known as **NoSQL (Not Only SQL)**. The Non-relational databases excel at horizontal scaling because they were created with the cloud in mind. It is used for storing and fetching data in a database and is generally used to store a large amount of data. Examples include Cassandra, MongoDB, and Redis.

SQLite for database handling

One of the most common and straightforward relational database systems is SQLite. A relational database management system is offered by SQLite, a software library. In terms of setup, database management, and resource requirements, the word *lite* in SQLite refers to it being *light-weight*. Self-contained, serverless, zero-configuration, and transactional are some of SQLite's distinguishing characteristics. Let us discuss all features in detail:

- **Server-less:** Typically, for an RDBMS, such as MySQL and PostgreSQL, to run, a separate server process is needed, but SQLite does not need a server to run or function.
- **Self-contained:** SQLite is self-contained since it only needs a small amount of help from the operating system or an outside library.
- **No configuration setting required:** SQLite does not make use of configuration files. There is no server process that requires setting up, starting, and stopping to run SQLite.
- **Follows ACID properties:** SQLite provides support for transactions' ACID properties of Atomicity, Consistency, Isolation, and Durability in data transactions.

In this section, we discuss the download and installation process of SQLite to get started with database handling in Python.

Downloading SQLite

To download SQLite, follow the following steps:

1. First, visit the official SQLite website using the URL <https://www.sqlite.org> and visit the download page <https://www.sqlite.org/download.html> for the desired SQLite set-up.
2. SQLite offers several tools for working across platforms, including Windows, Linux, and Mac. So, one must pick the proper version for download. For instance, to download the command-line shell program to use SQLite on Windows, we must select the pre-compiled binaries for the 32-bit or 64-bit Windows operation system, or we may select the complete bundle of command-line tools for managing the SQLite database. Here, to work with SQLite on Windows, we download the command-line shell program as shown in *Figure 7.3*:

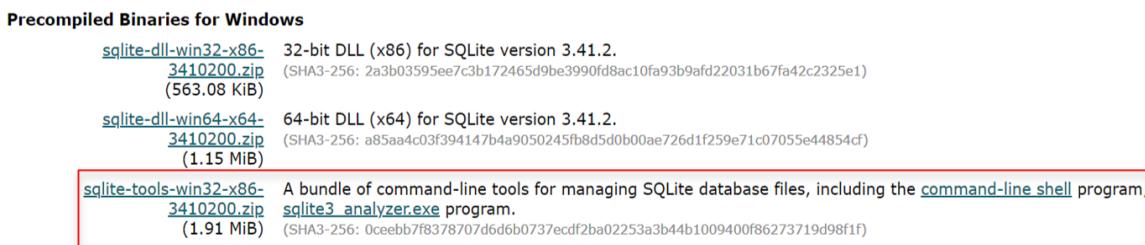


Figure 7.3: Snapshot for downloading SQLite tools

Installing SQLite in command-line

The installation process of SQLite is quite simple and straightforward. Follow the following steps to install SQLite in Windows:

1. We must create a new folder to extract the downloaded set-up files here, for example, **C:\sqlite**.
2. Next, extract the contents of the folder downloaded earlier to the destination **C:\sqlite** folder just created. After extracting files, we must be able to get three resulting programs in the **C:\sqlite** folder, as shown in *Figure 7.4*:

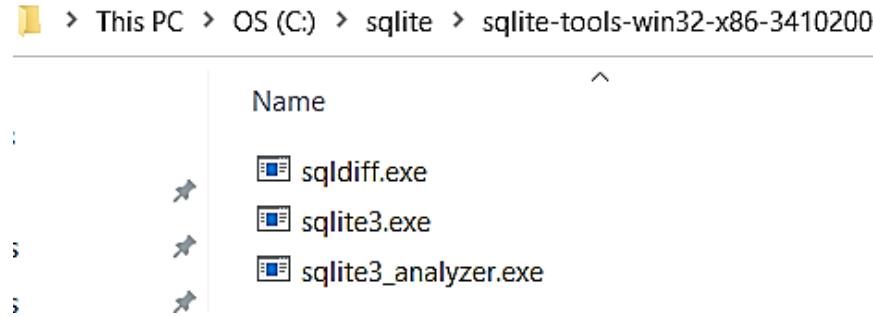


Figure 7.4: Extracted files from downloaded SQLite folder

3. Now open the command line window and navigate to the extracted files (or folder) inside the **C:\sqlite** folder using the **cd** command.
4. Now type the command **sqlite3** and press *Enter* to obtain the following output:

```
C:\sqlite>cd C:\sqlite\sqlite-tools-win32-x86-3410200
C:\sqlite\sqlite-tools-win32-x86-3410200>sqlite3
SQLite version 3.41.2 2023-03-22 11:56:21
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> =
```

```
sqlite> .help
.archive ...          Manage SQL archives
.auth ON|OFF          Show authorizer callbacks
.backup ?DB? FILE    Backup DB (default "main") to FILE
.bail on|off          Stop after hitting an error. Default OFF
.binary on|off         Turn binary output on or off. Default OFF
.cd DIRECTORY         Change the working directory to DIRECTORY
.changes on|off        Show number of rows changed by SQL
```

Figure 7.5: Snapshot of command line prompt for starting SQLite

5. We can use several commands from the prompt **sqlite>** to work in SQLite3. For example, we can use the command **.help** to see all available commands. Moreover, to quit the **sqlite>** prompt, we can use the **.quit** command.

GUI tools for SQLite

A free GUI tool available for managing SQLite databases is SQLiteStudio. It is a cross-platform, portable, and free tool for working with SQLite databases. It also offers support for importing and exporting data in different formats such as CSV, XML, and JSON. First, visit the download page of <https://sqlitestudio.pl/> to get the SQLiteStudio installer for Windows. Now, the downloaded file can then be launched after being extracted and installed in the desired directory location. Some other free SQLite GUI tools, such as DBeaver and DB Browser for SQLite, are available in addition to SQLite Studio.

SQLite working in Python

Python SQLite is used to create Python database applications with the SQLite database. Most laptops, smartphones, and browsers include SQLite by default. We may interact with the updated SQLite version 3.0 database with the help of the standard **sqlite3** Python library. The Python **sqlite3** module follows the Python Database API Specification v2.0 (PEP 249). *Figure 7.6* features the working of the **sqlite3** module in Python:

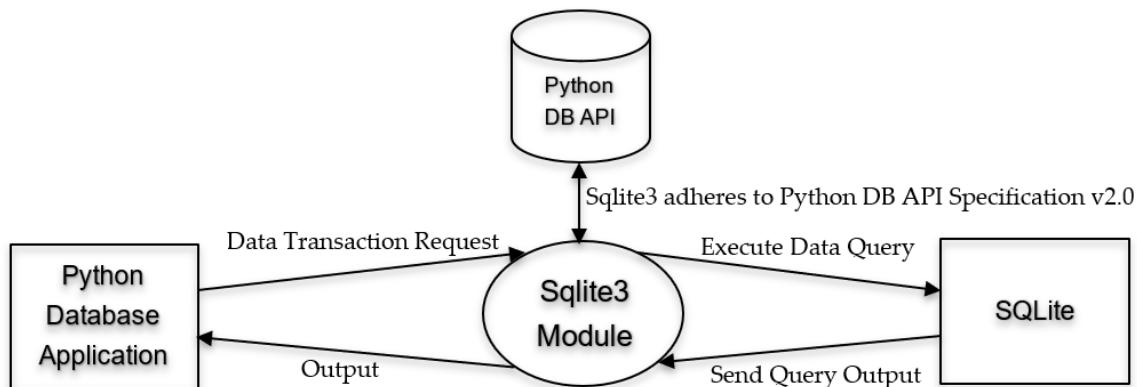


Figure 7.6: Working of SQLite3 module in Python

Please note that PEP 249 offers an SQL interface that has been created to promote and maintain the commonality between the Python modules used to access databases. This section will help readers understand the workings of SQLite database in Python using the **sqlite3** module. An internal table called **sqlite_master** can be found in all SQLite databases. This table's data provides a description of the database's schema. As its name suggests, **sqlite_master** is a master table that contains vital information about the database tables.

Connecting SQLite database in Python

In order to connect to the SQLite database in Python, we need to follow a step-wise procedure using the **sqlite3** python module. Instead of writing commands SQLite in the command prompt as in *Figure 7.5*, we can write Python database applications in PyCharm IDE using the **sqlite3** module to create and manage our database. Let us discuss each step involved along with its related syntax for using the **sqlite3** module in Python.

Step 1: Import **sqlite3** module

The import statement helps import the SQLite3 module in the application. We can interact with the SQLite database using the methods and classes defined in the Python SQLite3 module. The syntax is as follows:

```
import sqlite3
```

Step 2: Use the **connect()** method

The **connect()** method of the connector class opens a connection with the database name specified. Thus, we need to pass the database name as the parameter in **connect()**. If the specified database file already exists on the disk, then a connection will be established to it. Otherwise, if the database file does not exist, then SQLite creates a new database of the specified name. It returns the SQLite connection object if the connection is successful. The syntax is as follows:

```
sqlite3.connect(database [, timeout, other optional arguments])
```

Note: In the scenario of multiple connections, when a transaction is executing to modify the database, a lock is applied on the SQLite database until that transaction is committed. The timeout parameter denotes the wait time of the connection before the lock expires. The default for the timeout parameter is 5 seconds.

Step 3: Apply the **cursor()** method

The **cursor()** method of the Connection class is used to create a cursor object to execute SQLite queries from a Python program. It creates a cursor that is used repeatedly in our database application. Afterward, the cursor object can be used to call the **execute()** method to execute any SQL queries. The syntax is as follows:

```
connection.cursor()
```

Step 4: Apply the **execute()** method

The **execute()** method is used to execute an SQL query and return the result. The SQL statement may be parameterized with several user arguments passed at

run-time instead of SQL literals. The SQLite3 module supports two kinds of parameters: question marks and named arguments. The syntax is as follows:

```
cursor.execute(sql [, optional parameters])
```

Step 5: Obtain result using fetchall(), fetchone() or fetchmany()

The cursor methods such as **cursor.fetchall()**, **fetchone()**, or **fetchmany()** are used to read query results. The **cursor.fetchone()** method retrieves the next row of a query result, returning just a single sequence, or None at all, in case no more data is available in the result. The **cursor.fetchall()** method retrieves all the rows of a query result, thereby returning a list of results. In case no rows are available in the query result, an empty list is returned as output. Another method is **cursor.fetchmany()**, it retrieves a number of rows from the query result, returning a list of rows as output. The number of rows to be retrieved in output is indicated by the size parameter of the method. In case no more rows are available, an empty list is returned. The syntax is as follows:

```
cursor.fetchone()
```

```
cursor.fetchall()
```

```
cursor.fetchmany([size = cursor.arraysize])
```

Step 6: Close the cursor and related connection objects

After completing data handling tasks, we can use **cursor.close()** and **connection.close()** methods to close the cursor and SQLite connections, respectively. These methods do not automatically call the **commit()** method. Therefore, if we directly close the database connection without calling the **commit()** method before, then any database changes made by the user will be lost. The syntax is as follows:

```
cursor.close()
```

```
connection.close()
```

Please note that during the connection process, it is desirable to perform the necessary exception handling to catch any database exceptions that may be raised at any step.

Datatypes in SQLite

There are five primitive data types supported by SQLite and are referred to as **Storage Classes**. These storage classes define the data formats used to store data on disk by SQLite. Any data column can still store any type of data but the preferred storage class for a column is called its **Affinity**. [Table 7.1](#) gives a brief description of various storage classes in SQLite:

Storage class	Description
NULL	NULL values are special values denoting unknown or missing information.
INTEGER	Positive and negative whole numbers are Integers. An integer can have variable sizes such as 1, 2, 3, 4, or 8 bytes depending upon the number.
REAL	The 8-byte float values are real numbers with decimal positions.
TEXT	Character and string values are stored in the TEXT storage class. Unlimited length of TEXT is supported in SQLite.
BLOB	Binary Large Object (BLOB) can store any type of data. Unlimited size of data is supported by BLOB.

Table 7.1: *Datatypes in SQLite*

Exception handling tasks

An event that occurs during the execution of a program that obstructs the regular flow of the program's instructions is known as an *Exception*. Python typically stops and produces an error message when an error, or exception as we refer to it, happens. The exception raised must be handled properly. Otherwise, it leads to abrupt termination of the program. The **try...except** block can be used to deal with these exceptions. The following keywords, in the given order, play a vital role in exception handling process in Python:

- **try:** We can check a code block for errors or exceptions with the help of a **try** block.
- **except:** We may manage the exceptions using the **except** block by providing the necessary handling mechanism inside it. A single **try** statement can have multiple **except** statements.
- **else:** When there is no error, then we can run the code using the **else** block. The **else** block contains that code, which must be run when no exception is raised.

- **finally:** No matter whether an exception is raised or not inside the try- and **except** block, the **finally** block is always executed, allowing the user to run some mandatory code.

The syntax used for appropriate Exception handling mechanism is as follows:

```
try:
    # code that may cause exception
except:
    # code to run when exception occurs
else:
    # optional code to run when no exception occurs
    in try block
finally:
    # optional code to run definitely whether
    exception occurs or not.
```

Each database module must contain a certain number of errors as specified by the DB API. These exceptions are listed in *Table 7.2*:

Exception class	Description
Error	Error is the base class for all errors. Essential subclass StandardError .
DatabaseError	Used for errors in the database. Must subclass Error .
DataError	DataError is the subclass of class DatabaseError and represents errors in the data.
OperationalError	OperationalError is another subclass of DatabaseError that represents errors that are outside user control such as loss of connection to database.
IntegrityError	IntegrityError is also a subclass of DatabaseError for errors such as loss of the relational integrity constraints of uniqueness and foreign keys.

Table 7.2: Database exceptions in SQLite

Please note that Exception handling tasks in this chapter are discussed only with relevance to database handling in Python. Detailed discussion on all Exceptions in Python is beyond the scope of the chapter.

Database management with SQLite

Several **Create, Read, Update, and Delete (CRUD)** operations, such as creating a new database, creating a new table in the database, selecting data from the table, inserting new data into a table, updating table data, and so on, can be done using SQLite3 module in Python and the results can be viewed in SQLiteStudio in GUI mode thereby helping users to efficiently perform database management tasks.

Create new database

Let us consider the following Python application executed in Pycharm IDE that creates a new cursor object and a connection object to connect to a newly created database **New_SQLite_Python.db** and prints the installed SQLite version details:

```
import sqlite3      # Import sqlite3 module
try:
    # try keyword encloses a block of code to test for
    # any errors.
    # Create a new connection object
    newConnection =
    sqlite3.connect(r'C:\sqlite\db\New_SQLite_Python.db')
    cursor = newConnection.cursor()      # Create a new
    cursor object
    print("New SQLite Database Successfully created and
    Connected.")
    query1 = "select sqlite_version();"    # New sql
    query to execute
    cursor.execute(query1)      # Execute query using
    cursor object
```

```

        result = cursor.fetchall()      # Retrieve all rows
from result

        print("The current version of SQLite Database is:
", result)

        cursor.close()      # Close the cursor object

except sqlite3.Error as error:

        # except defines a block to handle error raised in
try block.

        print("Error while connecting to sqlite", error)

finally:

        # Check if the connection object exists and close
the connection

        if newConnection:

            newConnection.close()

            print("Closing the current SQLite connection")

```

Output:

The output can be seen in *Figure 7.7*:

```

New SQLite Database Successfully created and Connected.
The current version of SQLite Database is:  [('3.28.0',)]
Closing the current SQLite connection

```

Figure 7.7: Output

As we can see in the preceding output, the new database is successfully created and connected. We can view this database in SQLiteStudio also, which we had installed earlier. The steps to view in SQLiteStudio are as follows:

1. Open the **SQLiteStudio** by double-clicking on its launcher.
2. Click on the **Database** menu button present at the top and select the option to **Add a database**.

3. As the window to add a new database opens, navigate to the location of the database file created. In our case, we can navigate to the location:
C:\sqlite\db\New_SQLite_Python.db
4. Select the Name of the correct database that you want to connect.
5. Click on the **Test Connection** button to check if the database is connected or not. Click **OK**.
6. Now, to view the connected database, click the **View** menu button and select **Databases**.
7. In the left side **Databases** pane, we can see the newly added database. Right-click the database and select **Connect to the database** option to see the related tables and views.
8. The first image in *Figure 7.8* features connecting database in SQLite Studio while the second image demonstrates viewing the database in the SQLite Studio:

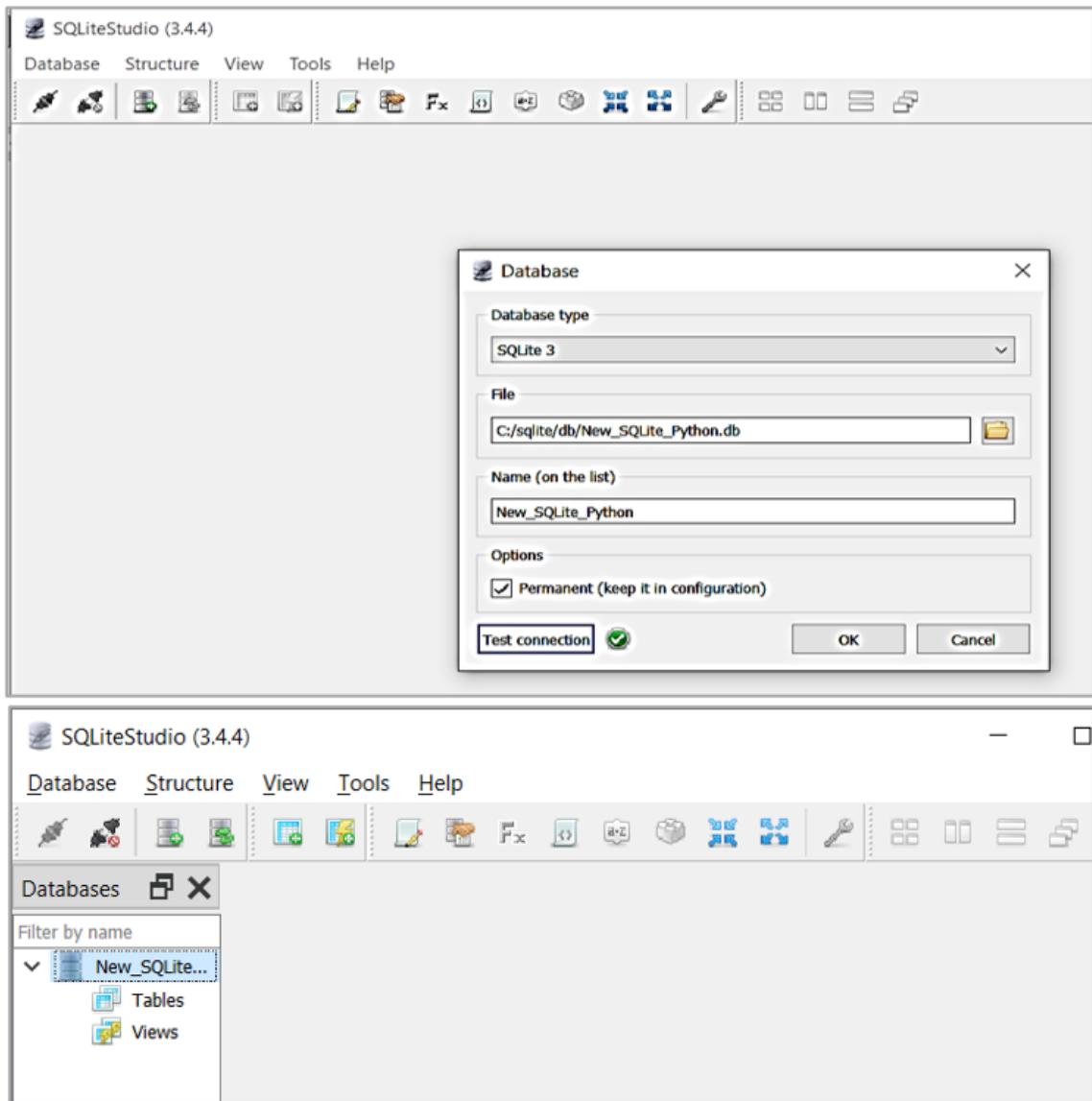


Figure 7.8: Create and View database in SQLite studio

Commands in SQLite

Commands in SQLite are comparable to those in SQL. Three different classes of SQLite commands exist:

- **DDL:** Data Definition Language
- **DML:** Data Manipulation Language
- **DQL:** Data Query Language

Data Definition Language (DDL) commands—CREATE, ALTER, and DROP

The database structure or schema can be defined with the use of data definition language. In this category, there are three commands, as discussed in [Table 7.3](#):

DDL Commands	Syntax	Example
CREATE Create a table, a view of a table, or another database object with the CREATE command.	CREATE TABLE table_name (Column1 column_type [constraint], Column2 column_type [constraint], ColumnN column_type [constraint]);	CREATE TABLE Department (DId INTEGER PRIMARY KEY, DName TEXT NOT NULL, DPhone TEXT DEFAULT 'UNKNOWN', UNIQUE (DName, DPhone));
ALTER A table can be changed using the ALTER statement. This alteration in table can be done by two ways: <i>Case 1.</i> Adding new column in existing table. <i>Case 2.</i> Rename a table with new name.	Syntax case1: ALTER TABLE table_name ADD new_column_name column_definition; Here, <code>column_definition</code> refers to the data type and constraints of the <code>new_column_name</code> (NULL or NOT NULL, and so on) to be added in table.	ALTER TABLE Department ADD COLUMN Location TEXT;
	Syntax Case 2: ALTER TABLE table_name RENAME TO new_table_name;	ALTER TABLE Department RENAME TO Old_Department;
DROP The DROP command is used to remove all data from a table, a table's view, or another database object.	DROP TABLE table_name;	DROP TABLE Old_Department;

Table 7.3: DDL commands in SQLite

Let us see the implementation of DDL commands in Python using SQLite3 module and DB API:

```
import sqlite3

try:
    sqliteConnection =
        sqlite3.connect(r'C:\sqlite\db\CompanyDatabase.db')

    query1 = '''CREATE TABLE Staff ( s_id INTEGER ,
s_name TEXT, s_email text, s_joining_date TEXT,
s_salary REAL);'''

    cursor1 = sqliteConnection.cursor()
    print("Successfully Connected to SQLite Database")
    cursor1.execute(query1)
    sqliteConnection.commit()
    print("SQLite table created successfully")
    cursor1.close()

    cursor2 = sqliteConnection.cursor()
    query2 = '''ALTER TABLE Staff ADD s_age INTEGER'''
    cursor2.execute(query2)
    sqliteConnection.commit()
    print("SQLite table altered successfully")
    cursor2.close()

except sqlite3.Error as error:
    print("Error raised on creating table", error)

finally:
    if sqliteConnection:
```

```

        sqliteConnection.close()

        print("sqlite connection closed
successfully!!")
    
```

Output:

The first image in *Figure 7.9* shows output in PyCharm IDE, whereas the second image displays the table structure created in SQLite Studio after refreshing the database schema:

Successfully Connected to SQLite Database
 SQLite table created successfully
 SQLite table altered successfully
 sqlite connection closed successfully!!

The screenshot shows the SQLite Studio interface. The left sidebar shows the 'Databases' tree with 'New_SQLite_Python (SQLITE 3)' selected, and the 'Tables (1)' node is expanded, showing the 'Staff' table. The main area displays the 'Structure' tab for the 'Staff' table. The table definition is as follows:

Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate	Generated	Default value
1 s_id	INTEGER								NULL
2 s_name	TEXT								NULL
3 s_email	text								NULL
4 s_joining_date	TEXT								NULL
5 s_salary	REAL								NULL
6 s_age	INTEGER								NULL

Figure 7.9: Output

SQLite table constraints

In the **CREATE Table** command, we can use several special keywords, such as **PRIMARY KEY**, **NOT NULL**, **UNIQUE**, and so on, to specify certain restrictions on data being stored in that table. These special keywords lay the guidelines and rules for the data in the table and are known as *SQLite constraints*. When a defined

constraint and an action on data conflict, the constraint will stop the appropriate action. There are three different types of integrities for constraints in SQLite:

- **Domain Integrity Constraints:** Data in table columns are validated using integrity constraints in the SQLite domain, and these constraints make sure that the data in the column falls within the given range of valid values. The different types of Domain Integrity constraints are described in [Table 7.4](#):

Domain Constraint	Description	Example
CHECK	<p>When inserting or updating data into table columns in SQLite, the CHECK constraint is used to establish specific requirements on a column and validate those conditions.</p> <p>Example: To check if book price>0</p>	<pre>CREATE TABLE BookDetail (B_id INTEGER PRIMARY KEY, B_name TEXT NOT NULL, B_Price INTEGER CHECK (B_Price > 0));</pre>
DEFAULT	<p>The default values for a column in SQLite are specified using the DEFAULT constraint.</p> <p>If a column's value is null or empty, the DEFAULT constraint will insert the default value into the column.</p>	<pre>CREATE TABLE BookDetail (B_id INTEGER PRIMARY KEY, B_name TEXT NOT NULL, B_Price INTEGER DEFAULT 150);</pre>
NOT NULL	Using the NOT NULL constraint in SQLite, it can be specified that NULL values cannot be stored in a column.	<pre>CREATE TABLE BookDetail (B_id INTEGER, B_name TEXT NOT NULL);</pre>

Table 7.4: Domain integrity constraints in SQLite

- **Entity integrity constraints:** Entity integrity constraints are used in SQLite to ensure that each data table row can be uniquely recognized. These constraints enable the identification and retrieval of data uniquely and in

accordance with user requirements. *Table 7.5* describes different Entity Integrity constraints used:

Entity integrity constraint	Description	Example
PRIMARY KEY	In general, SQLite allows to apply a primary key constraint on a single column; however, primary key constraint can also be applied on many columns together, such primary key is known as a composite key.	<pre>CREATE TABLE BookDetail (B_id INTEGER PRIMARY KEY, B_name TEXT, Publisher_Name TEXT);</pre>
UNIQUE	In SQLite, a unique constraint is used to restrict a column such that it can only contain unique data.	<pre>CREATE TABLE BookDetail (B_id INTEGER UNIQUE, B_name TEXT, B_Price TEXT);</pre>

Table 7.5: Entity integrity constraints in SQLite

- **Referential integrity constraints:** Referential integrity constraints are employed in SQLite to preserve the relationship between the tables, and they ensure that a value in one table references a value in another table, much like a foreign key relationship. Let us discuss the Foreign Key constraint in detail, as given in *Table 7.6*:

Referential Constraint	Description	Example
FOREIGN KEY	<p>The FOREIGN KEY constraint in SQLite is used to preserve the relationship between several tables and enables a unique identification of records across tables.</p> <p>In order to maintain the relationship across tables, the Foreign Key field in a table is declared using the</p>	<pre>CREATE TABLE PublisherDetail (P_id INTEGER PRIMARY KEY, P_name TEXT NOT NULL);</pre>

<p>FOREIGN KEY keyword, and it references the primary key of another table using the REFERENCES keyword.</p>	<pre>CREATE TABLE BookDetail (B_id INTEGER PRIMARY KEY, B_name TEXT NOT NULL, pid INTEGER NOT NULL, FOREIGN KEY (pid) REFERENCES PublisherDetail);</pre>
--	--

Table 7.6: Referential integrity constraints in SQLite

Let us see an example of creating **BookDetail** and **PublisherDetail** tables and applying the previously discussed constraints to them:

```
import sqlite3

try:
    sqliteConnection =
    sqlite3.connect(r'C:\sqlite\db\CompanyDatabase.db')

    cursor1 = sqliteConnection.cursor()
    print("Successfully Connected to SQLite Database")

    # executescript method allows executing whole SQL
    code in 1 step

    cursor1.executescript("""
        DROP TABLE IF EXISTS PublisherDetail; #Drop
        table if exists

        CREATE TABLE PublisherDetail (P_id INTEGER
        PRIMARY KEY,
        P_name TEXT NOT NULL);

        INSERT INTO PublisherDetail VALUES(101, 'BPB
        Publication');
    """)

```

```
        INSERT INTO PublisherDetail VALUES(102, 'Wiley
Publication');

        INSERT INTO PublisherDetail VALUES(103, 'McGraw
Hill');

        INSERT INTO PublisherDetail VALUES(104, 'CRC
Press');

        DROP TABLE IF EXISTS BookDetail;

        CREATE TABLE BookDetail (B_id INTEGER PRIMARY
KEY, B_name TEXT NOT NULL, B_type TEXT NOT NULL,
B_price REAL, pid INTEGER,
        FOREIGN KEY (pid) REFERENCES PublisherDetail);

        INSERT INTO BookDetail VALUES(9001, 'Learn
Python', 'Programming', 500, 101);

        INSERT INTO BookDetail VALUES(9002, 'Operating
Systems', 'Programming', 700, 102);

        INSERT INTO BookDetail VALUES(9003, 'Learn C++',
'Programming', 400, 101);

        INSERT INTO BookDetail VALUES(9004, 'Europe and
the East', 'Literature', 1000, 104);

        INSERT INTO BookDetail VALUES(9005, 'The Book
Thief', 'Fiction', 1500, NULL);

        """)

print("SQLite tables created successfully!!")
print("The details of Books are:")
cursor1.execute("SELECT * FROM BookDetail")
rows = cursor1.fetchall()
for r1 in rows:
    print(r1)
```

```
print("The details of Publishers are:")
cursor1.execute("SELECT * FROM PublisherDetail")
rows = cursor1.fetchall()
for r2 in rows:
    print(r2)
    sqliteConnection.commit()    # All changes must be
committed
except sqlite3.Error as error:
    # In case of an error, the changes are rolled back
    # and an error message is printed to the terminal.
    print("Error raised on executing 1 or more
commands", error)
    if sqliteConnection:
        sqliteConnection.rollback()
finally:
    if sqliteConnection:
        sqliteConnection.close()
        print("sqlite connection closed
successfully!!")
```

Output:

The output can be seen in *Figure 7.10*:

```
Successfully Connected to SQLite Database
SQLite tables created successfully!!
The details of Books are:
(9001, 'Learn Python', 'Programming', 500.0, 101)
(9002, 'Operating Systems', 'Programming', 700.0, 102)
(9003, 'Learn C++', 'Programming', 400.0, 101)
(9004, 'Europe and the East', 'Literature', 1000.0, 104)
(9005, 'The Book Thief', 'Fiction', 1500.0, None)

The details of Publishers are:
(101, 'BPB Publication')
(102, 'Wiley Publication')
(103, 'McGraw Hill')
(104, 'CRC Press')
sqlite connection closed successfully!!
```

Figure 7.10: Output

Please note that readers are advised to observe the preceding results and the upcoming examples of the chapter in SQLite Studio themselves as a practice task.

Data Manipulation Language (DML) commands

DML refers to Data Manipulation Language in a database. It is used for selecting data, inserting new data, deleting unwanted data, and updating table data in a database. It helps to retrieve data rows from tables and manipulate data in a relational database. The set of DML commands consists of the following:

DML command	Syntax	Example
-------------	--------	---------

DML command	Syntax	Example
INSERT To insert a new data record, that is, one or more new rows in a table, this command is used.	<pre>INSERT INTO table_name [(column1, column2, column3, ...columnN)] VALUES (value1, value2, value3, ...valueN);</pre> <p>NOTE: No need to give column names if data is to be inserted in all the columns and not specific ones. The syntax is modified as follows:</p> <pre>INSERT INTO table_name VALUES (value1, value2, value3,..valueN);</pre>	To insert values in all columns of the PublisherDetail table: <pre>INSERT INTO PublisherDetail VALUES (101, 'BPB Publication');</pre>
UPDATE In order to modify the existing records in a table this command is used.	<pre>UPDATE table_name SET column1 = value1, column2 = value2...., columnN = valueN WHERE [condition];</pre>	Update publisher name for publisher id 101. <pre>UPDATE PublisherDetail SET P_name = 'BPB Publication House' WHERE P_id = 101;</pre>
DELETE For deleting one or more unwanted rows from a table, we use the DELETE command. DELETE removes only rows/ tuples, but the table still persists in memory, whereas DROP completely removes the table from system memory.	<pre>DELETE FROM table_name WHERE [condition(s)];</pre>	Delete the publisher record whose id = 101. <pre>DELETE FROM PublisherDetail WHERE P_id = 101;</pre>

Table 7.7: DML commands in SQLite

Here, **WHERE** keyword in the preceding syntax defines a clause that helps to filter out data from a table based on a condition or criteria that needs to be satisfied. The **WHERE** clause returns the set of records for which the specified condition is **True**. It can be applied with queries such as **UPDATE**, **SELECT**, **DELETE**, **ALTER**, and so on.

Data Query Language (DQL) command: **SELECT**

DQL refers to Data Query Language. In order to retrieve data from a data table, **Data Query Language (DQL)** is used. In SQLite, a **SELECT** statement is used to retrieve data from the data table. The syntax is as follows:

```
SELECT column1, column2, columnN FROM table_name;
```

Alternatively, we can also use the ***** symbol to select all columns from the data table instead of specifying column names. The syntax is modified as follows:

```
SELECT * FROM table_name;
```

For example, to select all the data from the **PublisherDetail** table:

```
SELECT * FROM PublisherDetail;
```

Clauses in SQLite commands

Clauses are the conditions that can be used along with the **SELECT** command to retrieve records based on user requirements. Several clauses used in SQLite are defined in *Table 7.8*:

SQLite clause	Syntax	Example
WHERE: WHERE clause is used with the SELECT , UPDATE , and DELETE statements to apply a condition to filter the records and retrieve only the data that satisfies the given condition	SELECT column1, column2, columnN FROM table_name WHERE [condition]	To retrieve a record having P_id = 101: SELECT P_id , P_name FROM PublisherDetail WHERE P_id == 101;

from one or more SQLite clause tables.	Syntax	Example
AND: AND is a conjunctive operator and is used to combine multiple conditions. AND returns True if all conditions are True; otherwise, False.	SELECT column1, column2, columnN FROM table_name WHERE [condition1] AND [condition2] AND [conditionN];	To find record having P_id is 101 and P_name is BPB: SELECT P_id, P_name FROM PublisherDetail WHERE P_id == 101 AND P_name == 'BPB';

OR: OR is also a conjunctive operator and is used to combine multiple conditions. The OR operator returns TRUE only if at least one condition is True else False.	SELECT column1, column2, columnN FROM table_name WHERE [condition1] OR [condition2] OR [conditionN];	To retrieve a record having P_id = 101 or P_name = BPB. SELECT P_id, P_name FROM PublisherDetail WHERE P_id == 101 OR P_name == 'BPB';
ORDER BY: The fetched data can be sorted using the SQLite ORDER BY clause in either ascending (using ASC in syntax) or descending order (using DESC in syntax) depending on one or more columns.	SELECT column-list FROM table_name [WHERE condition] [ORDER BY column1, column2,..columnN] [ASC DESC]; Note: Default order is ASC.	To find all records with Publisher names in ascending order: SELECT * FROM STUDENT ORDER BY FEES ASC;

SQLite clause	Syntax	Example
---------------	--------	---------

<p>LIKE:</p> <p>It is used to match and compare text values with a specified pattern by using wildcard characters. If a correct match is found, LIKE returns True (that is, 1) or False (that is, 0).</p> <p>There are the following two wildcard characters used with the LIKE operator:</p> <p>The percent sign (%) denotes zero, one, or multiple numbers or characters.</p>	<pre>SELECT FROM table_name WHERE column LIKE [pattern]</pre>	<p>To find all records of publisher names having second character a:</p> <pre>SELECT * FROM PublisherDetail WHERE P_name LIKE '_A%';</pre> <p>To find all records for publisher names having b inside and ending with ion:</p> <pre>SELECT * FROM PublisherDetail WHERE P_name LIKE '%b%ion';</pre>
--	---	---

SQLite clause denotes a single number or character.	Syntax	Example

<p>GROUP BY:</p> <p>The SELECT statement combines identical elements into groups using the SQLite GROUP BY clause. Some major features are as follows:</p> <p>In the SELECT statement, the GROUP BY clause is used alongside the WHERE clause and comes before the ORDER BY clause.</p> <p>One row for each group is returned by the GROUP BY clause.</p>	<pre>SELECT column_1, aggregate_function(column_2) FROM table_name GROUP BY column_1, column_2 ORDER BY column1, column2....columnN;</pre> <p>Note: Aggregate functions such as MIN, MAX, SUM, COUNT, or AVG can be used with GROUP BY to do further operations on each group members.</p>	<p>To find the number of books of each book type and view in descending order of the count of books:</p> <pre>SELECT B_Type, COUNT(B_id) FROM BookDetail GROUP BY B_Type ORDER BY COUNT(B_id) DESC;</pre>
HAVING:	<pre>SUM(column_1) > column_2</pre>	<p>Applying the</p>

SQLite clause	aggregate function syntax	preceding GROUP BY example to filter groups with the count of books between 10 and 20:
The HAVING clause is used in association with the GROUP BY clause to filter groups based on a specified condition.	SELECT column_1, column_2 FROM table_name GROUP BY column_3 HAVING condition;	<pre>SELECT B_Type, COUNT(B_id) FROM BookDetail GROUP BY B_Type HAVING COUNT(B_id) BETWEEN 10 AND 20 ORDER BY COUNT(B_id);</pre>

DISTINCT: To remove all duplicate records and retrieve just unique records, use the DISTINCT clause.	<pre>SELECT DISTINCT column1, column2,..columnN FROM table_name WHERE [condition]</pre>	Find unique book types only: <pre>SELECT DISTINCT B_Type FROM BookDetail;</pre>
---	---	--

Table 7.8: Clauses used in SQLite commands

Aggregate functions

Let us now go over the following aggregate functions given in [Table 7.9](#):

Function	Description	Example
MIN	The SQLite MIN function is used to select the lowest (minimum) value for a certain column.	<p>To find the books of the lowest price where the smallest price of each book is less than 1,000:</p> <pre>SELECT B_name, B_type, MIN(B_price) FROM BookDetail GROUP BY B_type HAVING MIN(B_price) < 1000;</pre>

Function	Description	Example
MAX	It is used to select the highest (maximum) value for a certain column.	To find maximum price of books from table: <pre>SELECT MAX(B_price) AS "MaxPrice" FROM BookDetail;</pre>
AVG	The SQLite AVG function is used to select the average value for certain table columns.	To find the average price of Programming books: <pre>SELECT AVG(B_price) AS "Average Price" FROM BookDetail WHERE B_type == 'Programming'</pre>
COUNT	The SQLite COUNT function is used to count the number of rows in a database table.	Count all the number of books from BookDetail where the price is greater than 1,000: <pre>SELECT COUNT(*) AS "Number of Books" FROM BookDetail WHERE B_price > 1000;</pre>
SUM	The SQLite SUM function is used to select the total for a numeric column.	Find total worth of Fiction type books in terms of book price: <pre>SELECT SUM(B_price) AS "Total Price" FROM BookDetail WHERE B_type == 'Fiction';</pre>

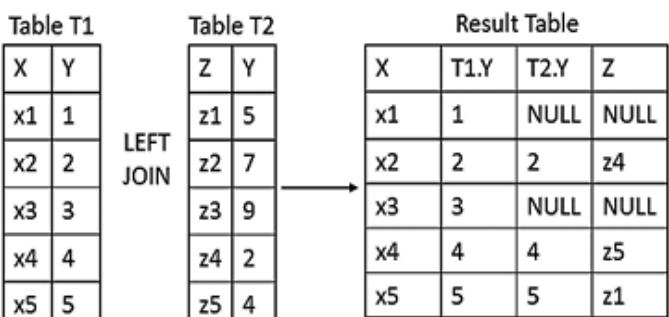
Table 7.9: Aggregate function in SQLite

Joins in SQLite

Records from two or more tables in a database are combined using the **JOIN** clause in SQLite. It joins fields from two different tables by using shared common values from both. There are three types of Joins defined in SQLite, as shown in [Table 7.10](#):

Join type	Visualizing join	Syntax

Join type	Visualizing join				Syntax																																						
<p>INNER JOIN</p> <ul style="list-style-type: none"> The INNER JOIN clause is the default join and returns rows from <i>Table T1</i> that have the corresponding row in <i>Table T2</i>. Returns only common rows from joined tables. 	<p>Table T1</p> <table border="1"> <thead> <tr> <th>X</th> <th>Y</th> </tr> </thead> <tbody> <tr><td>x1</td><td>1</td></tr> <tr><td>x2</td><td>2</td></tr> <tr><td>x3</td><td>3</td></tr> <tr><td>x4</td><td>4</td></tr> <tr><td>x5</td><td>5</td></tr> </tbody> </table> <p>INNER JOIN</p> <p>Table T2</p> <table border="1"> <thead> <tr> <th>Z</th> <th>Y</th> </tr> </thead> <tbody> <tr><td>z1</td><td>2</td></tr> <tr><td>z2</td><td>7</td></tr> <tr><td>z3</td><td>1</td></tr> <tr><td>z4</td><td>3</td></tr> <tr><td>z5</td><td>9</td></tr> </tbody> </table> <p>→</p> <table border="1"> <thead> <tr> <th>X</th> <th>T1.Y</th> <th>T2.Y</th> <th>Z</th> </tr> </thead> <tbody> <tr><td>x1</td><td>1</td><td>1</td><td>z3</td></tr> <tr><td>x2</td><td>2</td><td>2</td><td>z1</td></tr> <tr><td>x3</td><td>3</td><td>3</td><td>z4</td></tr> </tbody> </table>	X	Y	x1	1	x2	2	x3	3	x4	4	x5	5	Z	Y	z1	2	z2	7	z3	1	z4	3	z5	9	X	T1.Y	T2.Y	Z	x1	1	1	z3	x2	2	2	z1	x3	3	3	z4	<p>INNER JOIN compares the value of the common Y column of T1 with Y column in T2 table and returns combined data from both tables in result set where the condition T1.Y = T2.Y is True.</p>	<pre>SELECT column(s) FROM table1 [INNER JOIN table2 ON table1.column = table2.column</pre> <p>NOTE: INNER keyword is optional</p>
X	Y																																										
x1	1																																										
x2	2																																										
x3	3																																										
x4	4																																										
x5	5																																										
Z	Y																																										
z1	2																																										
z2	7																																										
z3	1																																										
z4	3																																										
z5	9																																										
X	T1.Y	T2.Y	Z																																								
x1	1	1	z3																																								
x2	2	2	z1																																								
x3	3	3	z4																																								

Join type	Visualizing join				Syntax
LEFT OUTER JOIN • LEFT OUTER JOIN returns all rows from the left-side table in the join query and only the matching rows from the table on the right. • In case of any unmatched rows in the right-side table, SQLite will insert a special NULL value for all such rows in the result.	 <p>The LEFT JOIN returns a result set that includes the following:</p> <ul style="list-style-type: none"> • All rows of table T1 with possible matching rows in T2. • If rows in <i>Table T1</i> have unmatched rows in <i>Table T2</i>, such T2 rows are filled with NULL in this case. 	<pre> SELECT column(s) FROM table1 LEFT JOIN table2 ON table1.column = table2.column WHERE condition ORDER BY column; </pre> <p>Note: WHERE and ORDER BY are optional clauses.</p>			

Join type	Visualizing join			Syntax																																									
CROSS JOIN <ul style="list-style-type: none"> • CROSS JOIN for two tables returns each row from the first table matched with each row from the second table. • It is also known as the Cartesian Product of two tables. 	<p>Table T1</p> <table border="1"> <tr><th>X</th><th>Y</th></tr> <tr><td>x1</td><td>1</td></tr> <tr><td>x2</td><td>2</td></tr> <tr><td>x3</td><td>3</td></tr> </table> <p>CROSS JOIN</p> <p>Table T2</p> <table border="1"> <tr><th>P</th><th>Q</th></tr> <tr><td>z1</td><td>5</td></tr> <tr><td>z2</td><td>7</td></tr> </table> <p>Result Table</p> <table border="1"> <tr><th>X</th><th>Y</th><th>P</th><th>Q</th></tr> <tr><td>x1</td><td>1</td><td>z1</td><td>5</td></tr> <tr><td>x2</td><td>2</td><td>z1</td><td>5</td></tr> <tr><td>x3</td><td>3</td><td>z1</td><td>5</td></tr> <tr><td>x1</td><td>1</td><td>z2</td><td>7</td></tr> <tr><td>x2</td><td>2</td><td>z2</td><td>7</td></tr> <tr><td>x3</td><td>3</td><td>z2</td><td>7</td></tr> </table>	X	Y	x1	1	x2	2	x3	3	P	Q	z1	5	z2	7	X	Y	P	Q	x1	1	z1	5	x2	2	z1	5	x3	3	z1	5	x1	1	z2	7	x2	2	z2	7	x3	3	z2	7	Since table T1 has three rows and T2 has two rows, the cross join will return six rows (because $3 \times 2 = 6$).	<pre>SELECT column(s) FROM table1 CROSS JOIN table2;</pre> <p>Note: Unlike an INNER or OUTER join, a CROSS JOIN has no condition to join the two tables</p>
X	Y																																												
x1	1																																												
x2	2																																												
x3	3																																												
P	Q																																												
z1	5																																												
z2	7																																												
X	Y	P	Q																																										
x1	1	z1	5																																										
x2	2	z1	5																																										
x3	3	z1	5																																										
x1	1	z2	7																																										
x2	2	z2	7																																										
x3	3	z2	7																																										

Table 7.10: Joins in SQLite

Parameterized Query and sub-queries in SQLite

When using *Parameterized Queries*, we use placeholders rather than entering the values directly into the statements. Queries with parameters improve security and efficiency. Question marks (?) and named placeholders in the form of dictionary **{key: value}** pairs are the symbols supported by the Python SQLite3 module for implementing parameterized queries. The named placeholders always start with a colon (:) character: The syntax is given as follows:

```
import sqlite3

try:
    sqliteConnection =
        sqlite3.connect(r'C:\sqlite\db\CompanyDatabase.db')

    cursor1 = sqliteConnection.cursor()
    print("Successfully Connected to SQLite Database")

    BookId = input("Enter the book id to update : ")
```

```

price = input("Enter the updated price : ")

# The question marks ? are placeholders for values.

# The values of price and Book_Id added to the ?
placeholders.

cursor1.execute("UPDATE BookDetail SET B_price=? WHERE B_id=?",
                (price, BookId))

# The rowcount property returns the number of
updated rows. Here, 1 row is updated.

print("Number of rows updated:
{}".format(cursor1.rowcount))

# We select Book id using named placeholder id
specified in dictionary form. Therefore here :id is
used as named placeholder.

cursor1.execute("SELECT * FROM BookDetail WHERE
B_id= :id", {"id": BookId})

row = cursor1.fetchone() # We retrieve the one
selected row

print(row)

sqliteConnection.commit()

except sqlite3.Error as error:
    print("Error raised on updating table", error)

finally:
    if sqliteConnection:
        sqliteConnection.close()

        print("sqlite connection closed
successfully!!")

```

Output:

The output can be seen in [Figure 7.11](#):

```
Successfully Connected to SQLite Database
Enter the book id to update : 9002
Enter the updated price : 750
Number of rows updated: 1
(9002, 'Operating Systems', 'Programming', 750.0, 102)
sqlite connection closed successfully!!
```

Figure 7.11: Output

A query nested inside another SQLite query and contained within the **WHERE** clause is known as a Subquery, Inner Query, or Nested Query. The data returned by a subquery is used as a condition in the main query to further limit the data that can be retrieved. The operators, such as **=**, **>**, **<**, **>=**, **<=**, **IN**, **BETWEEN**, and so on, can be used with subqueries in conjunction with the **SELECT**, **INSERT**, **UPDATE**, and **DELETE** statements. The syntax for subqueries is as follows:

```
SELECT column_name [, column_name ] FROM table1 [, table2 ]
WHERE column_name OPERATOR
      (SELECT column_name [, column_name ]
       FROM table1 [, table2]
      [WHERE])
```

For example, to find publishers who published books of price 500 and above, use the following syntax:

```
SELECT * FROM PublisherDetail WHERE P_id IN
      (SELECT pid FROM BookDetail WHERE B_price >= 500);
```

BLOB and DATE TIME in SQLite

The two most important and special datatypes in SQLite are **BLOB** and **DATE TIME**, which are frequently used for storing real-time data. An SQLite data type called a **BLOB** (*Binary Large Object*) is used to store large items, often massive files such as photographs, audio, video, documents, PDFs, and so on. For storage in an SQLite database, we must transform our files and photos into binary data (that is, a byte array in Python).

Apart from this, we frequently have to insert a Python **Date** or **DateTime** value also into an SQLite table. The SQLite date and timestamp values must also be read from the SQLite3 database and converted into Python **Date** and **DateTime** types. Please note that there is no built-in support for date and/or time storage classes in SQLite. Therefore, one may use the **TEXT**, **REAL**, or **INTEGER** storage classes to store date and time values. For example, we need to use the ISO8601 string format to store Date as a **TEXT** storage class in the format of **YYYY-MM-DD HH:MM:SS.SSS**. For example, **2023-01-01 10:30:05.112**. In the Python SQLite3 database, we can easily store dates or times by importing the **Datetime** module. Refer to [Table 7.11](#):

Date functions	Example
DATE The date() function accepts a time string and zero or more modifiers as arguments. It returns a date string in this format: YYYY-MM-DD .	Syntax: <code>DATE(timestring, modifier, modifier, ...)</code> Example: To find the last day of the month: <code>SELECT DATE('now', 'start of month', '+1 month', '-1 day');</code> +1 month results in the first day of next month from now on which –1 day is applied to find the last day of the previous month.
TIME To estimate a time value based on several date modifiers.	Syntax: <code>time(time_string[, modifier, ...])</code> Example: To add 2 hours to the given time and display the result: <code>SELECT time('10:30:30', '+2 hours');</code>
DATETIME To estimate a date and time value based on different date modifiers. The datetime() function returns a datetime value in this format: YYYY-MM-DD HH:MM:SS	Syntax: <code>datetime(time_string, modifier, ...)</code> Example: To find the current time of yesterday: <code>SELECT datetime('now', '-1 day', 'localtime');</code> First, the now time string returns the current date and time. Next, the –1-day modifier is applied to the current date-time that results in the current time of yesterday. Also, the localtime modifier instructs the function to return the local time.
STRFTIME	Syntax: <code>strftime(format_string,</code>

To estimate a date, month, and year value in a format string. A list of valid format specifiers for datetime are as follows:	<code>time_string [, modifier, ...])</code> Extracting day from a date time: <code>SELECT strftime('%d', '2023-11-10');</code> Output: 10.														
<table> <tr> <td><code>%d</code></td><td>For day of the month: 01-31</td></tr> <tr> <td><code>%f</code></td><td>For fractional seconds: SS.SSS</td></tr> <tr> <td><code>%H</code></td><td>For hour: 00-24</td></tr> <tr> <td><code>%m</code></td><td>For month: 01-12</td></tr> <tr> <td><code>%M</code></td><td>For minute: 00-59</td></tr> <tr> <td><code>%s</code></td><td>For seconds since 1970-01-01</td></tr> <tr> <td><code>%Y</code></td><td>For year: 0000-9999</td></tr> </table>	<code>%d</code>	For day of the month: 01-31	<code>%f</code>	For fractional seconds: SS.SSS	<code>%H</code>	For hour: 00-24	<code>%m</code>	For month: 01-12	<code>%M</code>	For minute: 00-59	<code>%s</code>	For seconds since 1970-01-01	<code>%Y</code>	For year: 0000-9999	Extracting month from a date time: <code>SELECT strftime('%m', '2023-11-10');</code> Output: 11.
<code>%d</code>	For day of the month: 01-31														
<code>%f</code>	For fractional seconds: SS.SSS														
<code>%H</code>	For hour: 00-24														
<code>%m</code>	For month: 01-12														
<code>%M</code>	For minute: 00-59														
<code>%s</code>	For seconds since 1970-01-01														
<code>%Y</code>	For year: 0000-9999														

Table 7.11: Date functions in SQLite

Readers are advised to go through the solved project under the project section to understand the application of different datatypes while creating tables and inserting data in these tables.

Conclusion

In this chapter, we discussed different aspects of database management techniques using SQLite in Python. The main focus is creating a persistent database consisting of multiple tables. Each table is created using the **create table** command with a set of columns having a specific datatype such as **TEXT**, **REAL**, or **INTEGER** and is bounded by Entity Integrity and Referential Integrity constraints. Data is inserted in a table using the **insert** command or via parametrized query. Data manipulation commands such as **Update** and **Delete** can be used to modify and delete existing records from the data table. The readers are advised to implement all the examples and projects given in the chapter for a better understanding of database concepts.

Points to remember

- Python SQLite is used to create Python database applications with the SQLite database. The Python SQLite3 module follows the Python Database API Specification v2.0 (PEP 249).

- There are five primitive data types supported by SQLite and are referred to as storage classes that define the data formats used to store data on disk by SQLite. These are **NULL**, **INTEGER**, **REAL**, **TEXT**, and **BLOB**.
- An event that occurs during the execution of a program that obstructs the regular flow of the program's instructions is known as an Exception.
- Several Create, Read, Update, and Delete (CRUD) operations, such as create a new database, create a new table in the database, select data from table, insert new data into table, update table data, and so on, can be done using SQLite3 module in Python and the results can be viewed in SQLite Studio in GUI mode.

Exercise

Attempt the following project.

Sample project with solution

Create a CRUD application for the hospital database, maintaining tables for doctors and patients with their diagnosis reports:

```
import sqlite3

connection =
sqlite3.connect(r'C:\sqlite\db\HMSDatabase.db')

cursor = connection.cursor()

error = 1

cursor.execute("""select count(name) from sqlite_master
where type='table' and name='DoctorDetails'""")

if cursor.fetchone()[0] == 0:

    cursor.execute("""CREATE TABLE DoctorDetails (
    doc_id INTEGER primary key, dnamedfirst TEXT,
    dnamedlast TEXT, password TEXT not null,
    speciality TEXT not null, shift TEXT not null,
    phone number(10) not null);""")
```

```
cursor.execute("""select count(name) from sqlite_master
where type='table' and name='PatientDetails'""")

if cursor.fetchone()[0] == 0:
    cursor.execute("""CREATE TABLE PatientDetails (
    pat_id number primary key, pfist TEXT, pdlast TEXT,
    City TEXT not null, DOB date not null, age INTEGER not
    null,
    DOA date not null, number number(10) not null);""")

    cursor.execute("""CREATE TABLE ViralDisease (
    pat_id number not null, dname TEXT primary key, vname
    TEXT, treatment TEXT, symptoms TEXT not null);""")

    cursor.execute("""CREATE TABLE BacteriaDisease (
    pat_id number not null, dname TEXT primary key, bname
    TEXT, treatment TXET, symptoms TEXT not null);""")

    cursor.execute("""CREATE TABLE WoundsInjury (pat_id
    number not null, iname TEXT primary key, idiagnosis
    TEXT, type TEXT not null);""")

    print("Database created successfully")

else:
    e = 1

    while e != 0:
        e = int(input("1. Log In To My Account\n2.
Create a New Doctor Account\n"))

        if e == 2:
            d_id = int(input('Enter Doctor id : '))
            d_nf = input('Enter Doctor first name : ')
            d_nl = input('Enter Doctor last name : ')
            d_pas = input('Enter password : ')
```

```

        d_spec = input('Enter Doctor speciality : ')
        d_shf = input('Enter working shift : ')
        d_ph = int(input('Enter Doctor phone
number : '))
        cursor.execute("""insert into DoctorDetails
values(?,?,?,?,?,?)""", (d_id, d_nf, d_nl, d_pas,
d_spec, d_shf, d_ph))
        e = 1
    elif e == 1:
        while error == 1:
            i = int(input("Enter your ID : "))
            p = input("Enter your Password : ")
            cursor.execute("""select count(doc_id)
from DoctorDetails where doc_id=(?)""", (i,))
            if cursor.fetchone()[0] == 1:
                cursor.execute("""select
count(password) from DoctorDetails where password=?""",
(p,))
                if cursor.fetchone()[0] == 1:
                    print("\nSign in successful!")
                    error = 0
                    e = 0
                    r = 1
                    cursor.execute("""select * from
DoctorDetails where doc_id=(?)""", (i,))
                    for row in cursor.fetchall():

```

```

        print("ID-", row[0], "Name,
row[1], row[2], "Speciality -", row[4], "\nShift -",
                           row[5], " Phone
Number -", row[6])

    while r != 0:
        print("\n1. View Patient
details\n2. Add a New Patient\n3. Delete Patient
Details\n0. Exit")

        r = int(input())
        if r == 1:
            access = input("\nEnter
Patient ID:- ")

cursor.execute("""select count(*) from PatientDetails
where pat_id=(?)""", (access,))

        if cursor.fetchone()[0]
!= 0:

            cursor.execute("""select * from PatientDetails where
pat_id=(?)""", (access,))

            print("\nPatient
Details - ")

            for row in
cursor.fetchall():

                print("Id: ",
row[0])
                print("First
Name: ", row[1])
                print("Last
Name: ", row[2])

```

```
        print("City: ",  
row[3])  
        print("Date of  
Birth: ", row[4])  
        print("Age: ",  
row[5])  
        print("Date of  
Admission: ", row[6])  
        print("\nDiagnosis  
Report - ")  
  
cursor.execute("""select count(*) from ViralDisease  
where pat_id=(?)""", (access,))  
if  
cursor.fetchone()[0] != 0:  
  
    cursor.execute("""select * from ViralDisease where  
pat_id=(?)""", (access,))  
    for row in  
cursor.fetchall():  
        print("Id:  
", row[0])  
  
    print("Disease Name: ", row[1])  
  
    print("ViralDisease Name: ", row[2])  
  
    print("Treatment: ", row[3])  
  
    print("Symptoms: ", row[4])
```

```
        print("\n")

cursor.execute("""select count(*) from BacteriaDisease
where pat_id=(?)""", (access,))

        if
cursor.fetchone()[0] != 0:

cursor.execute("""select * from BacteriaDisease where
pat_id=(?)""", (access,))

        for row in
cursor.fetchall():

        print("Id:
", row[0])

print("Disease Name: ", row[1])

print("BacteriaDisease Name: ", row[2])

print("Treatment: ", row[3])

print("Symptoms: ", row[4])

        print("\n")

cursor.execute("""select count(*) from WoundsInjury
where pat_id=(?)""", (access,))

        if
cursor.fetchone()[0] != 0:

cursor.execute("""select * from WoundsInjury where
pat_id=(?)""", (access,))
```

```
for row in
cursor.fetchall():

    print("Id:
", row[0])

    print("WoundsInjury Name: ", row[1])

    print("Diagnosis Name: ", row[2])

    print("Type: ", row[3])

    print("\n")

else:

    print("Incorrect
Patient id")

    elif r == 2:

        pid =
int(input('\nEnter id - '))

        pnf = input('Enter
first name - ')

        pnl = input('Enter last
name - ')

        pcity = input('Enter
city - ')

        pdob = input('Enter
date of birth - ')

        page = int(input('Enter
age - '))

        pdoa = input('Enter
date of admission - ')
```

```
        pnum = int(input('Enter
phone number - '))

cursor.execute("""insert into PatientDetails
values(?,?,?,?,?,?)""", (pid, pnf, pnl, pcity, pdob,
page, pdoa, pnum))

        print("1.
ViralDisease\n2. BacteriaDisease\n3. Wounds")

        m = int(input())
        if m == 1:
            dname =
input("\nEnter disease name - ")

            bname =
input("Enter ViralDisease name - ")

            treatment =
input("Enter treatment - ")

            symptoms =
input("Enter symptoms - ")

cursor.execute("""insert into ViralDisease
values(?,?,?,?,?)""", (pid, dname, bname, treatment,
symptoms))

        elif m == 2:
            dname =
input("\nEnter disease name - ")

            bname =
input("Enter BacteriaDisease name - ")

            treatment =
input("Enter treatment - ")
```

```
                symptoms =  
input("Enter symptoms - ")  
  
cursor.execute("""insert into BacteriaDisease  
values(?,?,?,?,?)""", (pid, dname, bname, treatment,  
symptoms))  
  
        elif m == 3:  
  
                iname =  
input("\nEnter WoundsInjury name - ")  
  
                idiag =  
input("Enter diagnosis - ")  
  
                itype =  
input("Enter WoundsInjury type - ")  
  
cursor.execute("""insert into WoundsInjury  
values(?,?,?,?)""", (pid, iname, idiag, itype))  
  
                print("\nPatient  
Added")  
  
                connection.commit()  
  
        elif r == 3:  
  
                access = input("\nEnter  
Patient ID:- ")  
  
cursor.execute("""select count(*) from PatientDetails  
where pat_id=(?)""", (access,))  
  
                if cursor.fetchone()[0]  
!= 0:  
  
cursor.execute("""delete from PatientDetails where  
pat_id=(?)""", (access,))
```

```
cursor.execute("""select count(*) from ViralDisease
where pat_id=(?)""", (access,))

        if
cursor.fetchone()[0] != 0:

cursor.execute("""delete from ViralDisease where
pat_id=(?)""", (access,))

cursor.execute("""select count(*) from BacteriaDisease
where pat_id=(?)""", (access,))

        if
cursor.fetchone()[0] != 0:

cursor.execute("""delete from BacteriaDisease where
pat_id=(?)""", (access,))

cursor.execute("""select count(*) from WoundsInjury
where pat_id=(?)""", (access,))

        if
cursor.fetchone()[0] != 0:

cursor.execute("""delete from WoundsInjury where
pat_id=(?)""", (access,))

        else: print("Incorrect
ID Patient does not exist")

        print("\nPatient
Deleted")

        connection.commit()

        elif r == 0: break
```

```

                else: print("Incorrect password.
Please retry ")

                else: print("Incorrect User ID. Please
retry ")

            break

        elif e == 2212:

            cursor.execute("""select * from
DoctorDetails""")

            print(cursor.fetchall())

            cursor.execute("""select * from
ViralDisease""")

            print(cursor.fetchall())

            cursor.execute("""select * from
BacteriaDisease""")

            print(cursor.fetchall())

            cursor.execute("""select * from
WoundsInjury""")

            print(cursor.fetchall())

        break

connection.commit()

connection.close()

```

Output:

The output can be seen in *Figure 7.12*:

```

1. Log In To My Account          Enter Patient ID:- 501
2. Create a New Doctor Account
2
Enter Doctor id : 101
Enter Doctor first name : Aman
Enter Doctor last name : Kumar
Enter password : 123456
Enter Doctor speciality : ENT
Enter working shift : Morning
Enter Doctor phone number : 9992229922
1. Log In To My Account
2. Create a New Doctor Account
1
Enter your ID : 101
Enter your Password : 123456

Patient Details -
Id: 501
First Name: Rohit
Last Name: Sharma
City: Delhi
Date of Birth:
Age: 20
Date of Admission: 20-08-2023

Diagnosis Report -
Id: 501
Disease Name: Viral Fever
ViralDisease Name: Viral Flu
Treatment: Medicine Dolo 650
Symptoms: Cold, Cough, Fever

Sign in successful!
ID- 101 Name Aman Kumar Speciality - ENT
Shift - Morning Phone Number - 9992229922
1. View Patient details
2. Add a New Patient
3. Delete Patient Details
0. Exit
2
Enter Patient ID:- 501
Patient Deleted

1. View Patient details
2. Add a New Patient
3. Delete Patient Details
0. Exit
3

Enter id - 501
Enter first name - Rohit
Enter last name - Sharma
Enter city - Delhi
Enter date of birth -
Enter age - 20
Enter date of admission - 20-08-2023
Enter phone number - 8999998888
1. ViralDisease
2. BacteriaDisease
3. Wounds
1

Enter disease name - Viral Fever
Enter ViralDisease name - Viral Flu
Enter treatment - Medicine Dolo 650
Enter symptoms - Cold, Cough, Fever

Patient Added

```

Figure 7.12: Output

Practice project

Create a CRUD application for the Student Management System. The system must enable adding new student details, modifying student details through his roll

number, and deleting details of a student after taking back-up.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline.com](https://discord(bpbonline.com)



CHAPTER 8

GUI Application Development Using Tkinter

Introduction

In the previous chapters, we have seen quite a lot of applications in Python, all of which were implemented using commands and user input given manually. The output of such applications is also command-based and can be seen on the terminal itself. In contrast to this, it is required that applications should be more interactive and user-friendly in nature. Actions and output must be graphical, consisting of icons such as windows, buttons, labels, checkboxes, radio buttons, and so on, instead of plain text and commands. This led to the use of the Tkinter library in Python for creating interactive and efficient **Graphical User Interface (GUI)** applications. This chapter will introduce users to the development of standard desktop applications in Python using the Tkinter library.

Structure

In this chapter, we will discuss the following topics:

- GUI programming in Python
- Getting started with Tkinter
- Introducing Tkinter widgets
- Organizing widgets with layout managers
- Event binding in Tkinter

Objectives

The purpose of this chapter is to enable users to develop interactive GUI applications consisting of user-friendly icons such as labels, buttons, menus, images, frames and so on. These widgets give a very professional look and feel to

the user, whereas using these applications that triggers actions based on user input via key press or mouse click.

GUI programming in Python

GUI stands for Graphical User Interface, which represents simple visual interaction between the user and machine instead of typing commands for performing a specific task. In contrast to the traditional text-based or command-based interface, communication in GUI is carried out by interactive components such as buttons and icons. Python has several frameworks for creating interactive GUI applications, such as JPython, wxPython, and Tkinter. However, the default GUI library used for Python is tkinter, which was written by Steen Lumholt and Guido van Rossum and then later revised by Fredrik Lundh. The combination of Python and Tkinter makes it quick and simple to develop GUI apps. The following features of tkinter make it a popular choice among users:

- **Pre-installed library:** Tkinter comes pre-installed with the standard Python library. So, no separate installation is required.
- **Easy to work:** It is quick to learn and easy to understand. Users can create a useful desktop application with very little coding.
- **Platform portability:** It provides the feature of platform portability as the applications developed can run across all operating platforms, including Windows, macOS, and Linux.

Getting started with Tkinter

As discussed earlier, Tkinter comes by default as a part of the standard Python library. Therefore, no separate installation is required to use Tkinter. In case one finds difficulty in importing the Tkinter module, then the following steps can be followed to install and use Tkinter in Pycharm IDE:

1. Open the command prompt to install a package named **future** to use tkinter.
2. Next, open Pycharm IDE and click on **File | New Project Setup | Settings for New Projects**, as shown in *Figure 8.1*:

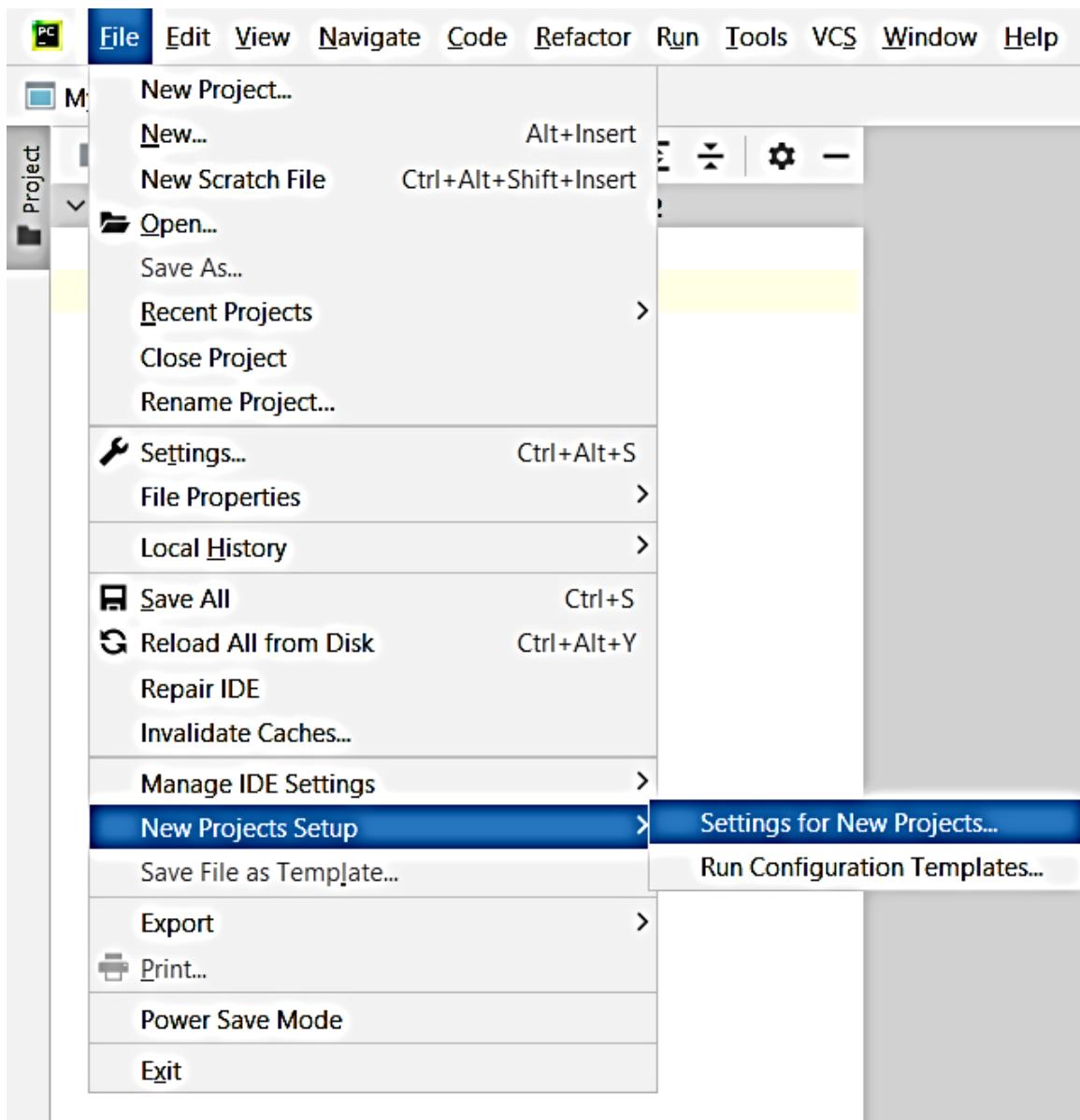


Figure 8.1: Project set-up for new application

3. Once the **Settings** window opens up, choose the correct **Python Interpreter** and click on the + icon given to add a new package, as shown in *Figure 8.2*:
 4. Now, from the list of available packages, search and select package **future** and click on the **Install Package** button.
 5. Finally, click **Apply** and then the **OK** button, as shown in *Figure 8.2*:

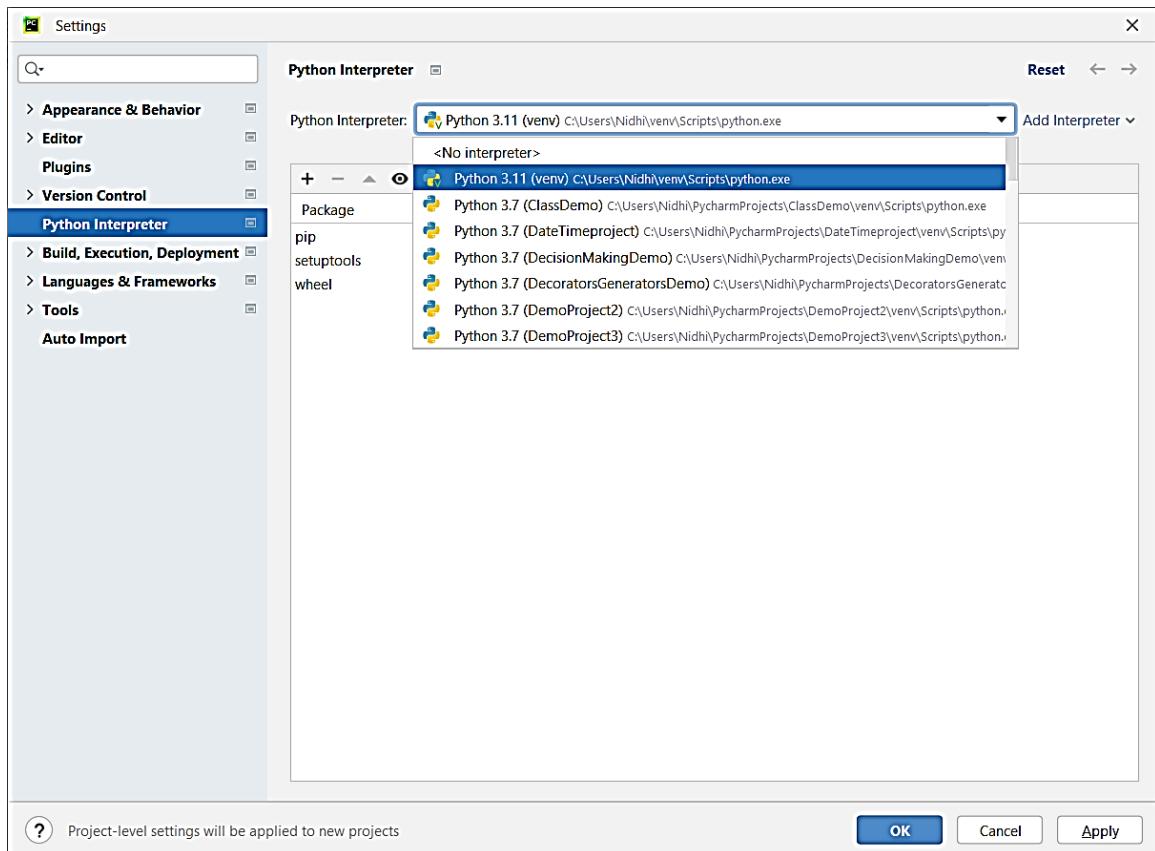


Figure 8.2: Python interpreter

6. Restart Pycharm IDE and start coding.

The process of creating desktop applications with Python Tkinter is not at all difficult. Following the given steps, a blank Tkinter top-level window can be created easily:

1. Import the Tkinter module for use in an application.
2. A main application window should be created.
3. Labels, buttons, frames, and other widgets should be added to the window.
4. In order for the actions to appear on the user's screen, call the main event loop.

Let us begin creating the first application in tkinter in Pycharm IDE. The following code generates an empty window with a given title. In the next section, we will learn adding different widgets to create more interactive windows:

```
# Import tkinter module
```

```
import tkinter
```

```
# Create a new blank window using Tk() method of
# tkinter module

window1 = tkinter.Tk()

# Write code below to add widgets.

# set window title

window1.wm_title("First Tkinter window")

# geometry() method defines the size of window

window1.geometry("300x200+10+20")

# show window

window1.mainloop()
```

Output:

Refer to [*Figure 8.3*](#):

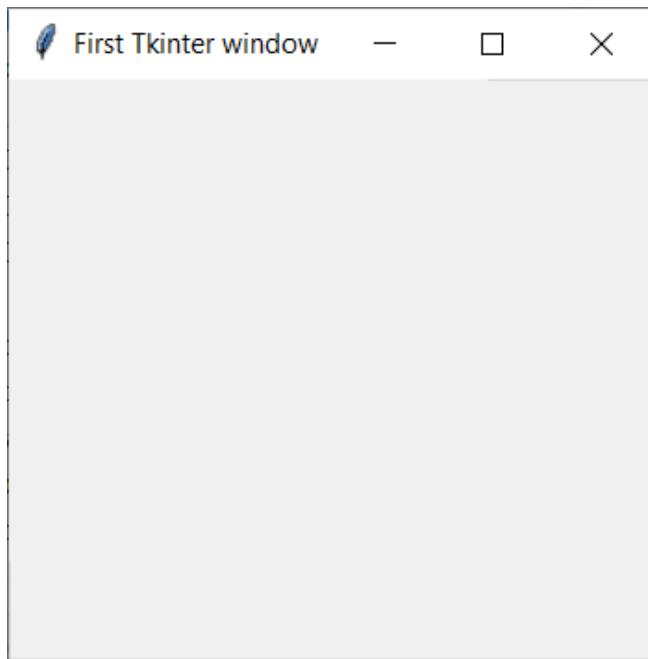


Figure 8.3: Output

In the preceding example, import the `tkinter` module first. After importing, run the `Tk()` function to initialize the application object. This will produce a top-level window (root) with a frame, title bar and an empty area to hold additional

widgets. The width, height, and coordinates of the frame's top left corner are defined inside the **geometry()** method in the form of pixel values. The syntax for the **geometry()** method is as follows:

```
window.geometry("widthxheight+XPOS+YPOS")
```

When we create a simple Tkinter application, we must call **window.mainloop()** to start the event listening loop. Any activity that takes place during the event loop and may cause the application to behave in a certain way, such as the pressing of a key or mouse button, is referred to as an *Event*. Tkinter already includes an event loop, and the user does not need to create any code that manually checks for events. However, the user must give the code that will be run in response to an event. For the events that we use in your application in Tkinter, we write functions referred to as *Event Handlers*.

In [Figure 8.3](#), the program checks to see if an event has occurred during the event loop. However, we observe a blank screen since no widgets or events are specified in the program. When the close button in the title bar is clicked, the event loop will end.

Introducing Tkinter widgets

Tkinter offers a variety of controls for GUI applications, including buttons, labels, and text boxes. These controls are usually referred to as **Widgets**. The Python Tkinter module supports numerous types of widgets, such as Button, Label, Entry, Text, MessageBox, and so on, for creating GUI applications. Most of these widgets support several common properties that are set while calling the constructors of each of these widget classes. These common properties are discussed in [Table 8.1](#):

Parameter	Description
container	Container is the parent component where we place the widget.
text	The text is the label of the widget.
command	It specifies the function to be called automatically when the widget is clicked.
fg	To set the Foreground color of the widget.
bg	It represents the background color of the widget.
bd	It denotes the border of widget.

Parameter	Description
image	It is set to the image displayed on the widget.
state	The default state of a widget (such as Button) is NORMAL. In the DISABLED state, a widget is greyed out and does not respond to mouse events and keyboard presses.
underline	Set this option to make the widget text underlined.
height	It specifies the height of the widget.
width	It specifies the width of the widget.
padx	Additional padding to the widget in the horizontal direction.
pady	Additional padding to the widget in the vertical direction.

Table 8.1: Properties used as options in widgets settings

Now, let us discuss each widget class along with their additional properties, if any.

Button

The Button class can be used to construct the button. In the applications, a clickable item is represented by a button widget. Usually, you represent the action that will be taken when a button is pressed using text or an image. When a button is clicked, a function or method of the class is immediately invoked if its command option is assigned to the function or method. In Tkinter, this is referred to as the **Command Binding**. The syntax is as follows:

```
btnObject = Button(container, option=value, ...)
```

The **Button** class constructor supports all the properties given in [Table 8.1](#) as parameters or options in the given syntax. Let us consider an example to demonstrate the working of the **Button** widget:

```
import tkinter as tkn
from tkinter import *
from tkinter import messagebox
def func():    # display message box on button click
    messagebox.showinfo(title="Information", message="Button clicked")    # change button state to DISABLED
```

```

if (btn['state'] == tkn.NORMAL):
    btn['state'] = tkn.DISABLED

window=Tk()

btn=Button(window, text="Click Me!", command=func,
fg='black', bg='white')
btn.place(x=100, y=100)

window.title('Tkinter Window')
window.geometry("300x200+10+10")
window.mainloop()

```

Output:

The output can be seen in *Figure 8.4*:

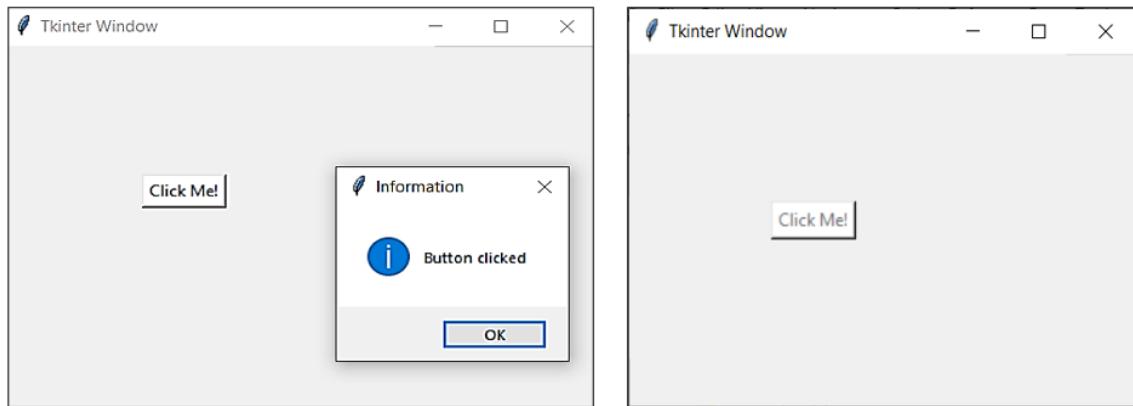


Figure 8.4: Output

Label

Label is used to provide some static information or message to the user regarding other widgets present in the Python application. It works similar to a display box which can contain some text or images. The syntax of using **Label** widget is as follows:

```
labelObject = Label (container, option=value, ...)
```

The parameters used while creating a label widget are similar to button object as described in the previous [Table 8.1](#). Apart from those, some other options used in label widget are shown in [Table 8.2](#):

Parameter	Description
anchor	It controls the position of text inside the widget. The default value is CENTER, which centers the text in the given space.
cursor	We can set the cursor value to a pattern arrow, dot, and so on so that the mouse pointer will change to that pattern when it is over the label.
highlightcolor	The color of the focus highlight when the widget has focus.
relief	It specifies the type of the border. Some of the values are FLAT, SUNKEN, RAISED, GROOVE, and RIDGE.
textvariable	It is used to bind the text displayed in a label widget to a control variable of class defined by <code>StringVar()</code> .

Table 8.2: Options used as parameters for label widget

Let us consider an example to highlight the working of label widget:

```
from tkinter import *
window1 = Tk()

# Create a control variable named cvar and set its
value

cvar = StringVar()
cvar.set("This is a label widget text!!")

# Assign value of cvar to textvariable property of
label

label1 = Label(window1, textvariable=cvar,
relief=RIDGE)

# pack() is a geometry manager that adds and arranges
widgets to a window

label1.pack()
```

```
window1.title('Tkinter Window')
window1.geometry("200x100+10+10")
window1.mainloop()
```

Output:

The output can be seen in *Figure 8.5*:

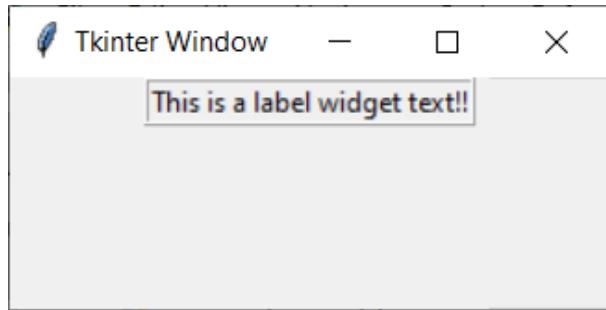


Figure 8.5: Output

In the preceding example, *line 9* contains the command **label1.pack()**, where **pack()** denotes the Pack layout manager that adds and arranges widgets to the window. We will discuss the topic of Layout Managers in detail later in the chapter.

Entry widget

The Entry widget is used as textbox GUI component to accept user values. It displays only a single-line text field to the user for writing and editing the text inside it. The syntax is as follows:

```
entryObject = Entry (container, option=value, ...)
```

Apart from the properties mentioned in *Table 8.3*, the Entry widget class constructor supports the following additional options as well:

Parameters	Description
exportselection	To enable automatic copying of text to the clipboard. Set to 0 to disable
highlightthickness	The thickness of the focus highlight. The default value is 1.

Parameters	Description
<code>xscrollcommand</code>	Horizontal scrollbar is displayed if user enters more text, then the actual width of the widget.
<code>insertofftime</code>	The value is a non-negative integer indicating the number of milliseconds the cursor should remain “off” in each blink cycle. If set to zero, then the cursor does not blink and remains on all the time.
<code>insertontime</code>	Specifies a non-negative integer indicating the number of milliseconds the cursor should remain “on” in each blink cycle.
<code>show</code>	It hides the original string being entered and instead displays some other characters. For example, the password is typed using an asterisk (*).
<code>textvariable</code>	It binds the text displayed in Entry widget to a control variable of class defined by <code>StringVar()</code> .
<code>highlightbackground</code>	It represents the color to display when the widget does not have input focus.
<code>highlightcolor</code>	The color of the focus highlight when the widget has the focus.

Table 8.3: Options properties for entry widget

The entry widget offers various methods to manipulate user input. These methods are described in [Table 8.4](#):

Method	Description
<code>delete(first, last = none)</code>	It is used to delete the specified characters inside the widget.
<code>get()</code>	It is used to get the text written inside the widget.
<code>icursor(index)</code>	It changes cursor position by placing it before the character present at the given index position.
<code>index(index)</code>	It places the cursor to the left of the character present at given index.
<code>insert(index, s)</code>	It inserts the string “s” before the character present at the specified index.

Table 8.4: Methods supported by an entry widget

Let us consider an example to create a calculator that takes user input and produces results after doing specified calculation:

```
from tkinter import *
# Syntax to create a Window
window1 = Tk()
window1.title('Entry Widget Demo') # To give the title of the window
window1.geometry("500x500") # To initialize the size of the window.
n1 = IntVar() # initialize the first Integer variable
n2 = IntVar() # initialize the second Integer variable
# Label Heading
heading1=Label(window1, text="Addition Calculator", font= ("Arial", 25))
heading1.place(x=100, y=50) # Used to place label at a position
# Labels for Inputs
num1 = Label(window1, text="Enter first number: ", font= ("Arial", 10))
num1.place(x=80, y=100)
num2 = Label(window1, text="Enter second number: ", font= ("Arial", 10))
num2.place(x=80, y=150)
outputLabel = Label(window1, text="Output:", font= ("Arial", 10))
outputLabel.place(x=150, y=300)
# Use functions to give action to the button.
```

```
def addCalc():
    sum = n1.get() + n2.get()
    addEntry.delete(0, END)
    addEntry.insert(0, str(sum))

# Entries text boxes Syntax

inputentry1 = Entry(window1, textvariable=n1)
inputentry1.place(x=230, y=100, height=25, width=143)
inputentry2 = Entry(window1, textvariable=n2)
inputentry2.place(x=230, y=150, height=25, width=143)

# Result text entry

addEntry = Entry(window1)
addEntry.place(x=230, y=300, height=25, width=143)

# Use command to call the functions in Add button.

buttonAdd = Button(window1, text="ADD (+)", fg="black",
bg="cyan", font=("Arial", 15), command=addCalc)
buttonAdd.place(x=210, y=200, height=50, width=100)

window1.mainloop()
```

Output:

The output can be seen in *Figure 8.6*:

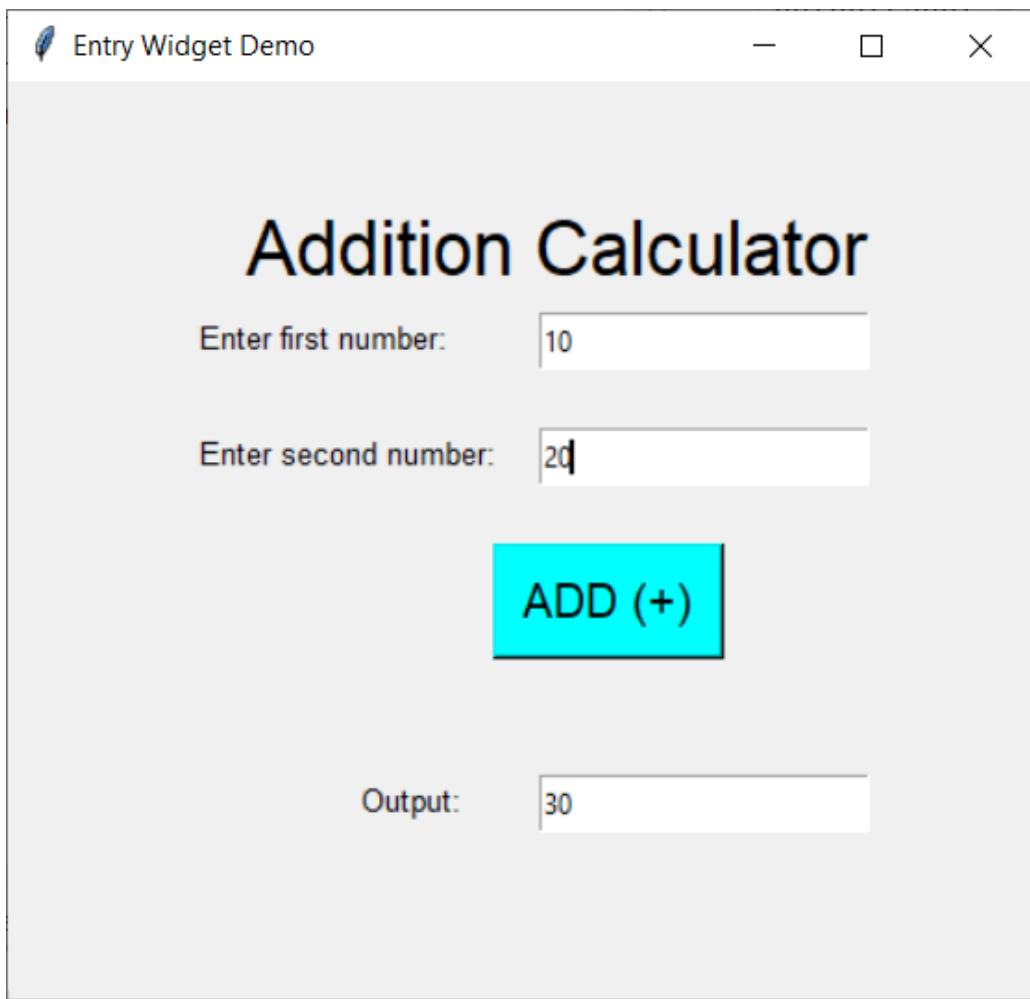


Figure 8.6: Output

Text widget

Similar to the Entry widget, the Text widget is also used as textbox GUI component to accept user values. While the Entry widget displays only a single-line text field to the user, the Text widget provides a multi-line text field to enable the user to write and edit the text inside it. The syntax is as follows:

```
textObject = Text (container, option=value, ...)
```

Text widget class constructor supports the following additional options as well:

Parameters	Description
exportselection	To enable automatic copying of text to the clipboard. Set to 0 to disable

Parameters	Description
highlightthickness	The thickness of the focus highlight. The default value is 1.
xscrollcommand	Horizontal scrollbar is displayed if user enters more text than actual width.
yscrollcommand	To make the Text widget vertically scrollable, we can use this option.
insertofftime	It represents a non-negative integer value indicating the number of milliseconds the insertion cursor should remain “off” in each blink cycle. If this option is zero, then the cursor does not blink; it is on all the time.
insertontime	Specifies a non-negative integer value indicating the number of milliseconds the insertion cursor should remain “on” in each blink cycle.
highlightcolor	The color of the focus highlights when the widget has the focus.
spacing1	It specifies the vertical space given above each line of the text. The default is 0.
spacing2	It specifies extra vertical space to add between displayed lines of text when a logical line wraps. Default is 0.
spacing3	It specifies the vertical space to insert below each line of the text.
wrap	This option is used to wrap the wider lines into multiple lines. Set this option to the WORD to wrap the lines. Default value is CHAR.

Table 8.5: Options parameters for text widget

Several methods can be used with the Text widget to manipulate input value entered by user. The list of methods is given in [Table 8.6](#):

Method	Description
delete(startindex, endindex)	This method deletes the characters of the specified range.
get(startindex, endindex)	It returns the characters present in the specified range.

<code>index(index)</code>	It is used to get the absolute index of the specified index.
<code>insert(index, string)</code>	It is used to insert the specified string at the given index.
<code>see(index)</code>	It returns a Boolean value, true or false, depending upon whether the text at the specified index is visible or not.

Table 8.6: Methods supported by text widget

Let us see an example using text widget to get user input and display the same on screen:

```
from tkinter import *
window1=Tk()
# Method to fetch user input from text widget and
# display on screen.
def retrieve_userinput():
    inputVal=textBox1.get("1.0", "end-1c")
    print(inputVal)
# Define text widget and button
textBox1=Text(window1, height=5, width=20)
button1=Button(window1, height=1, width=10,
text="Display", command=retrieve_userinput)
# Add and arrange widgets on window
textBox1.pack()
button1.pack()
window1.mainloop()
```

Output:

The output can be seen in *Figure 8.7*:



Figure 8.7: Output

In order to get Tkinter input from the Text widget, we must add a few more attributes to the `textbox1.get()` function in preceding line 5 for retrieving its input. The first part, “1.0”, means that the input must be read from line one, that is, the first character. Here, `END` is an imported constant, which is set to the string “end.” It means to read until the end of the text box is reached. The only issue with this is that it actually adds a newline to our input. In order to resolve this issue, instead of `END`, we use `end -1c`. Here, `-1c` deletes 1 character, whereas `-2c` would mean delete two characters, and so on.

Radiobutton

The `tkinter` package provides support for the following four types of selection widgets that enable the selection of one or more options based on user input. These selection widgets are as follows: **Radiobutton**, **Checkbutton**, **Combobox**, and **Listbox**.

The Radiobutton widget shows a toggle button with an ON/OFF state, or a radio button. Out of multiple radio buttons, only one of them can be selected by the user to be active at a time. The syntax is as follows:

```
radiobtn = Radiobutton(container, option=value, ...)
```

Table 8.7 lists the add-on options supported by the **Radiobutton** object apart from those mentioned in *Table 8.1*:

Option	Description
<code>image</code>	It can be set to an image object to be displayed on the radiobutton instead the text.
<code>selectcolor</code>	It specifies the color of the radio button when it is selected by user.

Option	Description
selectimage	We can specify image to be displayed on the radiobutton when it is selected by user.
state	It represents the state of the radio button. Default state is NORMAL. It can be set to DISABLED to make the radiobutton unresponsive.
text	It is used to specify text to be displayed on the radiobutton.
value	Value option of each radiobutton is assigned to the control variable when it is turned on by the user.
variable	It represents the control variable which keeps track of the user's choices. It is shared among all the radiobuttons.

Table 8.7: Options parameters supported by radio button

Various methods are supported by Radiobutton widget, as shown in [Table 8.8](#):

Method	Description
deselect()	It is used to turn off the radiobutton.
flash()	It is used to flash the radiobutton between its active and normal colors few times.
invoke()	It is used to call any procedure associated when the state of a Radiobutton is changed.
select()	It is used to select the radiobutton.

Table 8.8: Methods supported by radio button

Checkbutton

This button also acts as a toggle. Its caption is followed by a rectangular checkbox. The tick mark in the box, which appears when it is clicked to be in the ON state, vanishes when it is in the OFF state. Unlike Radiobutton, more than one checkbox can be selected by the user at a time. The syntax is as follows:

```
checkbtn = Checkbutton(container, option=value, ...)
```

[Table 8.9](#) lists the add-on options supported by **Checkbutton** object:

Option	Description
--------	-------------

Option	Description
activebackground	It specifies the background color when the cursor hovers on checkbutton.
activeforeground	It specifies the foreground color when the cursor hovers on checkbutton.
disabledforeground	It is the color which represents the text of a disabled checkbutton.
offvalue	The associated control variable is set to 0 by default if the button is unchecked. We can change the state of an unchecked variable with offvalue
onvalue	The associated control variable is set to 1 by default if the button is checked. We can change the state of the checked variable with onvalue.
selectcolor	It represents the color of the checkbutton when selected. Default is red.
selectimage	It represents the image displayed on the checkbutton when selected.
state	It represents the state of the checkbutton. Default state is NORMAL. It can be set to DISABLED to make the checkbutton unresponsive.
variable	It is the control variable that keeps track of the state of the checkbutton.

Table 8.9: Options parameters supported by check button

The methods that can be called with the checkbuttons are described in the following table:

Method	Description
deselect()	This method is called to turn off the checkbutton.
flash()	The method flashes the checkbutton between active and normal colors, quite a few times and finally leaves it the way it started.
invoke()	This method invokes the method associated with the checkbutton.
select()	This method is called to turn on or select the checkbutton.
toggle()	This method is used to toggle between the different checkbuttons.

Table 8.10: Methods supported by check button

Combobox

The **ttk** module of the **tkinter** package contains the definition of the class **combobox**. A combo box combines an entry field and a listbox. One of the Tkinter widgets has a down arrow to choose from a menu of possibilities. It enables users in choosing from the list of available options. A pop-up of the scrolling Listbox is displayed down the entry field when the user clicks on the drop-down arrow on the entry field. Only when an option from the Listbox is selected by the user and the selected option be shown in the entry field. The syntax is as follows:

```
from tkinter import *
from tkinter.ttk import *
cmbboxObject = ttk.Combobox(container, option=value,
...)
```

The following table describes a list of options or properties supported by Combobox:

Option	Description
justify	The alignment of text within the widget.
height	The height of the pop-down listbox.
postcommand	It is called immediately before displaying the values.
textvariable	It specifies a name whose value is linked to the widget value.
values	It specifies the list of values to display in the drop-down listbox.
width	It specifies the width of the entry window.

Table 8.11: Options parameters supported by Combobox

Listbox

This widget displays the whole collection of string items, unlike Combobox. One or more objects may be chosen by the user. The syntax is as follows:

```
lstboxObject = Listbox(container, option=value, ...)
```

The following table highlights the option parameter(s) supported by Listbox:

Option	Description
font	The font type of the Listbox items.
height	It represents the count of the lines shown in the Listbox. Default value is 10.
highlightcolor	The color of the Listbox items when the widget is under focus.
highlightthickness	The thickness of the highlight.
relief	The type of the border. The default is SUNKEN.
selectbackground	The background color that is used to display the selected text.
selectmode	It determines the number of list items that can be selected from the listbox. It can set to BROWSE, SINGLE, MULTIPLE, and EXTENDED.
width	It represents the width of the widget in characters.
xscrollcommand	It is used to let the user scroll the Listbox horizontally.
yscrollcommand	It is used to let the user scroll the Listbox vertically.

Table 8.12: Options parameters supported by Listbox

There are the following methods associated with the Listbox:

Method	Description
activate(index)	It is used to select the list value at the specified index.
curselection()	It returns a tuple containing the selected elements, counting from 0. If nothing is selected, this method returns an empty tuple.
delete(start, last = None)	It is used to delete the lines which exist in the given range.
get(start, last = None)	It is used to get the list items that exist in the given range.
size()	It returns the number of lines that are present in the Listbox widget.
xview()	This is used to make the widget horizontally scrollable.

Method	Description
yview()	It allows the Listbox to be vertically scrollable.

Table 8.13: Methods supported by Listbox

Let us consider the following example demonstrating the application of all selection widgets, namely, **Radiobutton**, **Checkbutton**, **Combobox**, and **Listbox**. In this example, we try to create a sample window for the Pizza online order system:

```

import tkinter
from tkinter import *
from tkinter import messagebox
from tkinter.ttk import *
# A sample window for Online pizza Ordering
window1 = Tk()
var1 = StringVar()
var1.set("one")
label0 = Label(text="Pizza Order Online")
label0.place(x=80, y=100)
v0 = IntVar()
v0.set(1)
# Add Radio buttons
r1 = Radiobutton(window1, text="Dine-in", variable=v0,
value=1)
r2 = Radiobutton(window1, text="Delivery", variable=v0,
value=2)
r1.place(x=150, y=50)
r2.place(x=250, y=50)

```

```
# Add ComboBox to select only one option
label1 = Label(text="Choose your outlet :")
label1.place(x=80, y=100)
data1 = ("North Region", "East Region", "West Zone",
"South Extension", "Country Side")
cbox = tkinter.ttk.Combobox(window1, values=data1)
cbox.place(x=220, y=100)

# Add listbox to select multiple options
label2 = Label(text="Choose your Pizza :")
label2.place(x=80, y=150)
lbox = Listbox(window1, height=10,
selectmode='multiple')
data2 = ("Veg-Delight", "Farmhouse", "Indo-Italian",
"Margherita", "Cheese & Corn", "Pasta-Pizza")
for values in data2:
    lbox.insert(END, values)
lbox.place(x=220, y=150)

# Add checkbuttons to check multiple values
label3 = Label(text="Choose your Add-Ons :")
label3.place(x=80, y=350)
v1 = IntVar()
v2 = IntVar()
v3 = IntVar()
C1 = Checkbutton(window1, text="Cheese", variable=v1)
C2 = Checkbutton(window1, text="Jalapeno", variable=v2)
C3 = Checkbutton(window1, text="Olive", variable=v3)
```

```
C1.place(x=220, y=350)
C2.place(x=290, y=350)
C3.place(x=370, y=350)

# Add button and message box

def func():    # display message box on button click
    messagebox.showinfo(title="Order
Confirmation",message="Thank you!! \n Your Order will
be ready soon!!.")

btn=Button(window1, text="Place Order!", command=func)
btn.place(x=200, y=400)

window1.title('Selection Widgets')
window1.geometry("400x300+10+10")
window1.mainloop()
```

Output:

Refer to [*Figure 8.8*](#):

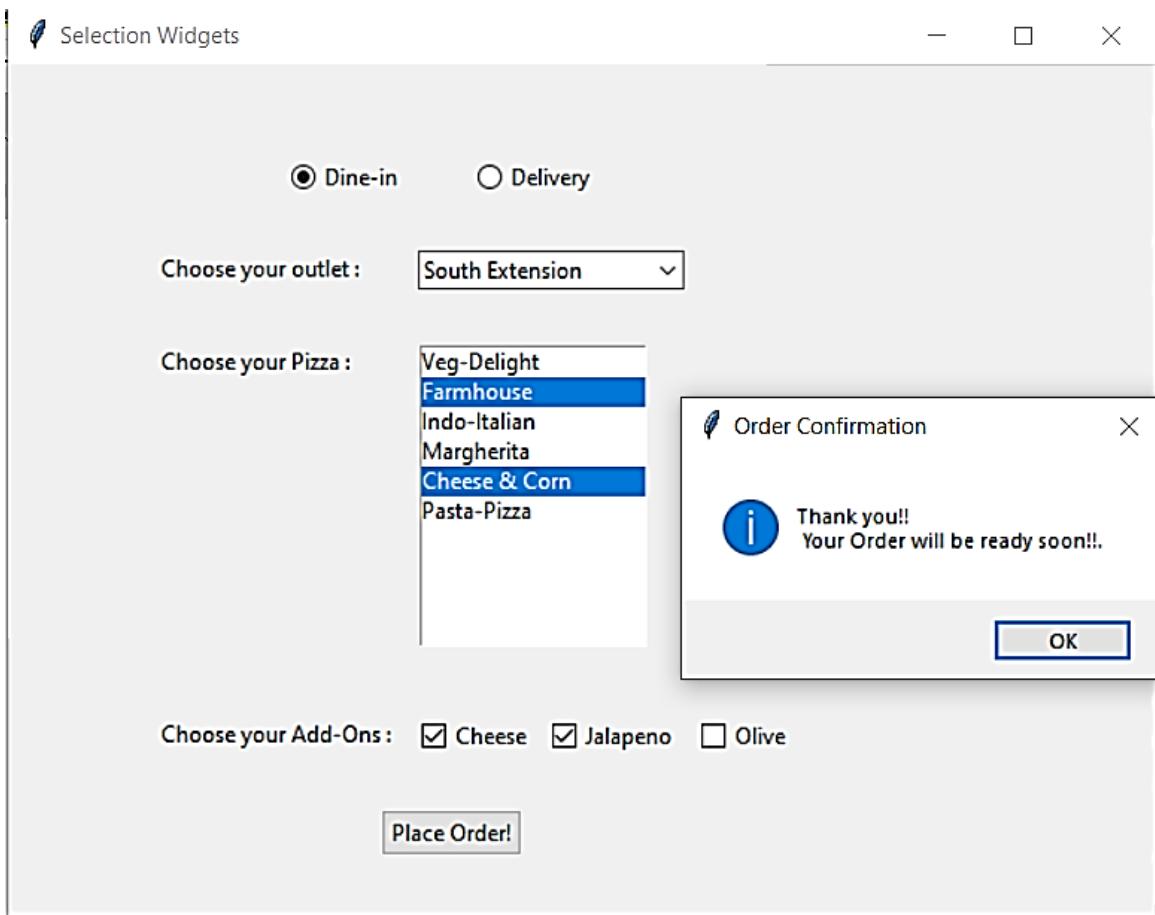


Figure 8.8: Output

Menu

Use menus to organize the numerous functions that an application has so that you can navigate it more easily. To organize similar operations closely, you typically use a menu. For instance, the **File** menu, along with its sub-menu and other options, can be found in most text editors. The **Menu** widget support is widely used in Tkinter. In the Python application, different types of menus (top level, pull down, and pop up) are created using the **Menu** widget. Just below the parent window's title bar is where one may find the top-level menus. We must build a new instance of the **Menu** widget and use the **add()** method to add additional commands to it. The syntax for creating a menu object is as follows:

```
menuObject = Menu(container, option=value, ...)
```

The options that are supported by **Menu** widget are as follows:

Option	Description
<code>bg</code>	It specifies the background color of the widget.
<code>bd</code>	To assign border width of the widget.
<code>cursor</code>	To change cursor type when it hovers the widget.
<code>postcommand</code>	<code>postcommand</code> is set to the function that is called when mouse hovers the menu.
<code>relief</code>	To specify border of the widget. Default value is RAISED.
<code>image</code>	To display an image on the menu.
<code>tearoff</code>	The choices in menu start taking position index at 1 by default. If set <code>tearoff = 1</code> , then it will start from 0th position.
<code>title</code>	To change the title of the tear-off menu window.

Table 8.14: Options parameters supported by menu widget

The menu widget also supports the following methods:

Method	Description
<code>add_command(options)</code>	To add the menu items to the top-level menu.
<code>add_radiobutton(options)</code>	To add radiobutton to the menu.
<code>add_checkbutton(options)</code>	To add the checkbuttons to the menu.
<code>add_cascade(options)</code>	To create a hierarchical menu to the parent menu by associating one menu to another menu.
<code>add_seperator()</code>	To add the separator line to the menu.
<code>add(type, options)</code>	It is used to add the specific menu item to the menu.
<code>delete(startindex, endindex)</code>	It is used to delete the menu items exist in the specified range.
<code>entryconfig(index, options)</code>	It is used to configure a menu item identified by the given index.
<code>index(item)</code>	It is used to get the index of the specified menu item.
<code>insert_seperator(index)</code>	It is used to insert a separator at the specified index.

invoke(index)	It is used to invoke the command callback associated with the choice given at the specified index (position).
type(index)	It is used to get the type of the choice specified by the index.

Table 8.15: Methods supported by menu widget

Let us consider an example to create top-level menus along with submenus:

```
from tkinter import Tk, Menu, messagebox

# root window
window1 = Tk()
window1.geometry('400x400')
window1.title('Menu Widget Demo')
# create a top level menubar
menubar = Menu(window1)
window1.config(menu=menubar)
# create the File Menu option
file_menu1 = Menu(menubar, tearoff=0)
# add menu items to the File menu
file_menu1.add_command(label='New File')
file_menu1.add_command(label='Open File')
file_menu1.add_command(label='Close')
file_menu1.add_separator()
# add a submenu to File Menu
sub_menu = Menu(file_menu1, tearoff=0)
sub_menu.add_command(label='Keyboard Shortcuts')
sub_menu.add_command(label='Color Themes')
```

```
# add the File menu to the menubar
file_menu1.add_cascade(label="Preferences",
menu=sub_menu)

# add a separator and add Exit menu item
file_menu1.add_separator()

file_menu1.add_command(label='Exit',
command=window1.destroy)

menubar.add_cascade(label="File", menu=file_menu1,
underline=0)

# Adding Group By Menu
search = Menu(menubar, tearoff = 0)

menubar.add_cascade(label = 'Search', menu = search)

# Adding Checkbuttons to Search Menu
search.add_checkbutton(label="By Name")
search.add_checkbutton(label="By Date Modified")
search.add_checkbutton(label="By Type")
search.add_checkbutton(label="By Size")

def about():

    messagebox.showinfo('About Editor', 'Sample Text
Editor displaying Menu widget!!')

# create the Help menu

help_menu1 = Menu(menubar, tearoff=0)

help_menu1.add_command(label='About Editor',
command=about)

# add the Help menu to the menubar

menubar.add_cascade(label="Help",
menu=help_menu1,underline=0)
```

```
window1.mainloop()
```

Output:

Refer to output in *Figure 8.9*:

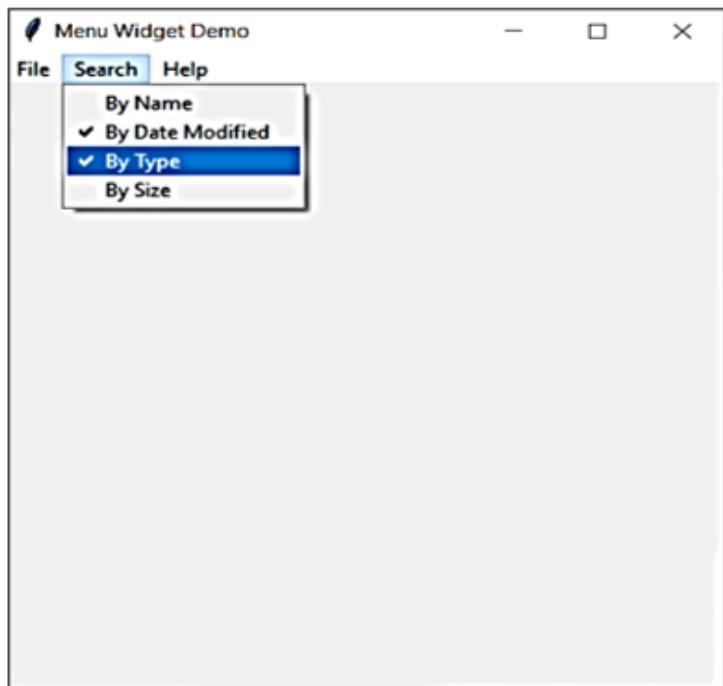
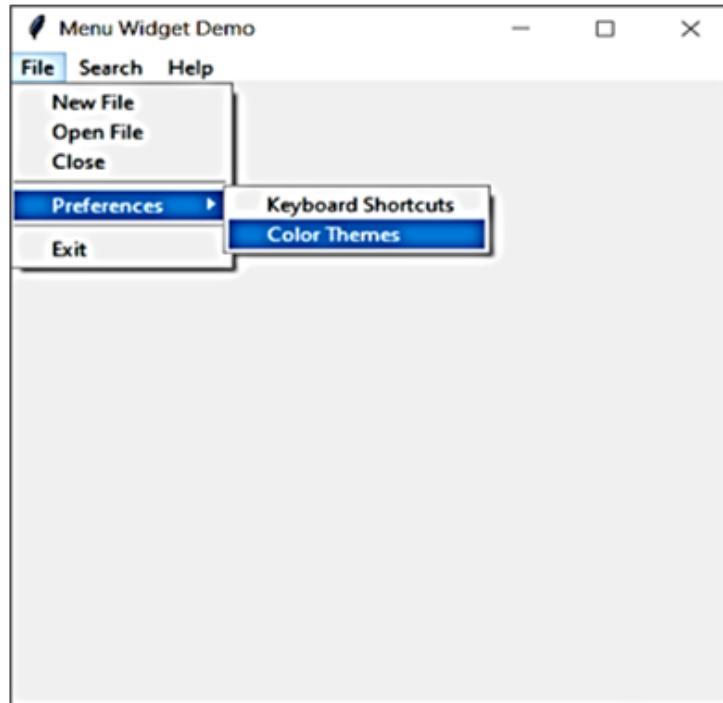


Figure 8.9: Output

Spinbox

The Entry widget can be substituted with the **Spinbox** widget. It offers the user a selection of a range of values from which to choose. It is used when a user is presented with a limited number of options to choose from. With the **Spinbox**, there are several ways to customize the widget. **Spinbox** uses the following syntax:

```
spinboxObject = Spinbox(container, option=value, ...)
```

The following table contains the list of options that can be used to configure a **Spinbox**:

Option	Description
command	The associated callback with the widget, which is called each time the state of the widget is called.
Cursor	The mouse pointer is changed to the cursor type assigned to this option.
Format	This option is used for the format string. It has no default value.
From_	It is used to show the starting range of the widget.
Justify	It is used to specify the justification of the multi-line content. Default is LEFT.
Relief	It is used to specify the type of the border. The default is SUNKEN.
Repeatdelay	It is used to control the button auto repeat. The value is given in milliseconds.
Repeatinterval	It is similar to repeatdelay. The value is given in milliseconds.
State	The default state is NORMAL. Other values: NORMAL, DISABLED, or readonly.
Textvariable	The control variable to control the behavior of the widget's text.
To	It specify the maximum limit of the widget value.

Option	Description													
Validate	<p>This option controls how the widget value is validated.</p> <p>The following are the values that are used for validate option:</p> <table border="1"> <tr> <td>focus</td><td>Validate when the widget gets or loses focus.</td></tr> <tr> <td>focusin</td><td>Validate whenever the widget gets focus.</td></tr> <tr> <td>focusout</td><td>Validate whenever the widget loses focus.</td></tr> <tr> <td>key</td><td>Validate when a keystroke changes contents.</td></tr> <tr> <td>all</td><td>Validate in all the preceding situations.</td></tr> <tr> <td>none</td><td>Default option value it turns off validation.</td></tr> </table>		focus	Validate when the widget gets or loses focus.	focusin	Validate whenever the widget gets focus.	focusout	Validate whenever the widget loses focus.	key	Validate when a keystroke changes contents.	all	Validate in all the preceding situations.	none	Default option value it turns off validation.
focus	Validate when the widget gets or loses focus.													
focusin	Validate whenever the widget gets focus.													
focusout	Validate whenever the widget loses focus.													
key	Validate when a keystroke changes contents.													
all	Validate in all the preceding situations.													
none	Default option value it turns off validation.													
Validatecommand or vcmd	It is associated to the function callback, which is used for the validation of the widget content.													
values	It is the tuple containing the possible values for this widget.													
xscrollcommand	Used with set() method to make this widget horizontally scrollable.													

Table 8.16: Options parameters supported by spinbox widget

The methods associated with **Spinbox** are as follows:

Option	Description
delete(start, end)	This method is deletes the characters present at the specified range.
get(start, end)	It is used to retrieve the characters present in the specified range.
identify(x, y)	It is used to identify an element present within the specified range.
index(idx)	It retrieves the absolute value present at the given index.
insert(idx, string)	It enables to insert the string at the specified index.
invoke(element)	It is used to invoke the callback associated with the widget.

Table 8.17: Methods supported by spinbox widget

Let us consider the following example to demonstrate the working of a spinbox:

```
from tkinter import *
# Use widget config method to update widget settings
# such as text
def show_selected():
    lb2.config(text=f'You selected {var.get()}', bg='yellow')

window1 = Tk()
window1.geometry("200x200")
lb1= Label(window1, text="Select your age: ")
var = StringVar()
spin = Spinbox(window1, from_=1, to=100, textvariable=var, command=show_selected)
lb2= Label(window1, text="")
lb1.pack()
spin.pack()
lb2.pack()
window1.mainloop()
```

Output:

The output can be seen in *Figure 8.10*:

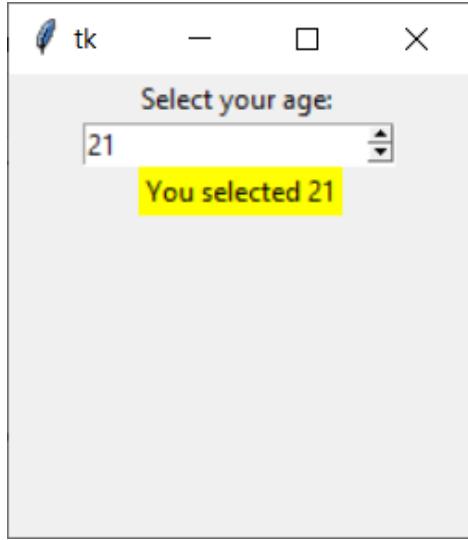


Figure 8.10: Output

When creating a widget in Tkinter, we must provide a number of values as parameters, which are used to set specific characteristics for that widget. Whether we can modify these parameters once the widget has been generated is an important question for the developer. The answer to this query is the **config()** method. The Tkinter **config()** method may be used on any widget to modify and adjust settings that we may have previously applied or have not yet applied. In the preceding example, we use the **config()** method inside a user-defined method **show_selected()** to modify the text settings of the label **lb2** and set the text to the currently selected element of spinbox.

The Treeview Widget and Treeview Scrollbar

Tree structures are frequently used in programming applications. The purpose of the **ttk.Treeview** widget presents a hierarchical structure so that the user can use mouse actions to display or hide any part of the structure. The hierarchy displayed by a Treeview widget is technically a forest structure: there is no single root, simply a collection of top-level nodes, each of which may contain second-level nodes, each of which may have third-level nodes, and so forth. In the Windows operating system, as we try navigating a specific directory or folder, we often come across such a hierarchical treeview arrangement of directories. Refer to [Figure 8.11](#):

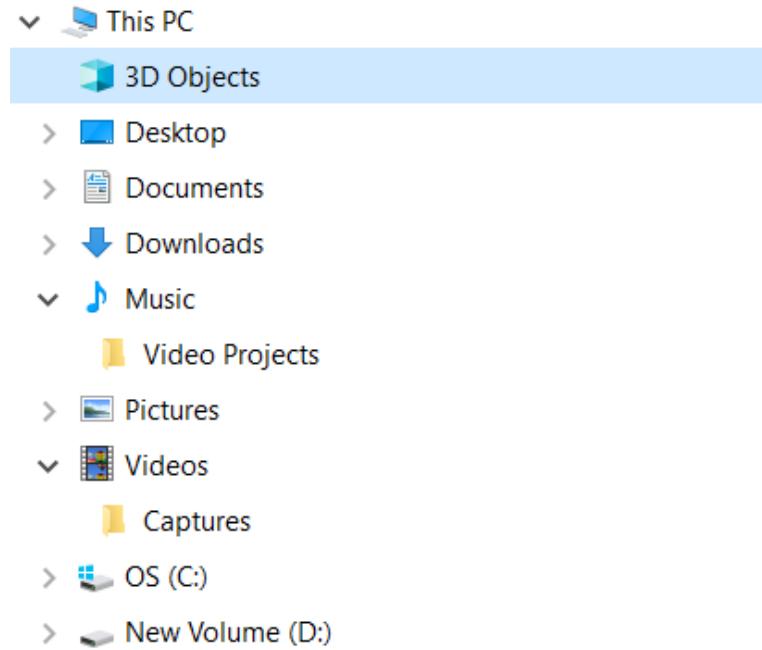


Figure 8.11: Directory structure in the form of Treeview hierarchy

The user may click on the icon of a directory to collapse it, thereby hiding all the sub-folders in it. They can click the icon again to expand it, revealing all the sub-folders present. The Treeview widget generalizes this idea so that users can use it to display any hierarchical structure and can collapse or expand subtrees of this hierarchy with a mouse click. We may show data in both tabular and hierarchical formats with a Treeview widget.

Tree view widgets can be of the following two different types:

- **Hierarchical widget:** This is a level-by-level representation of the data. Any number of top-level nodes is possible, and the sub-nodes will not show up until we click on the extension. Third-level and second-level nodes may also exist beneath a top node. An example of such formatting is shown in [Figure 8.12](#).
- **Tree table widget:** The tree table widget presents data in a tabular manner, with rows denoting objects or entities and columns denoting attributes of those items. The first column contains icons that collapse or expand items, whereas other columns display the remaining information. For instance, a simple file browser widget can have directory icons in the first column, file names in the second column, file sizes, timestamp, and so on in the rest of the columns. An example of such formatting is shown in [Figure 8.13](#).

For creating a Tree view widget, we need to import the `tkinter.ttk` module. Every item in the tree has a unique identifier string called the `iid`. Users can provide `iid` values manually and `ttk` can also generate them. The syntax of generating `Treeview` widget is as follows:

```
treeobject = ttk.Treeview(container, option=value, ...)
```

The constructor returns the new Treeview widget. Its options include those shown in the following table:

Option	Description
<code>class_</code>	You may provide a widget class name when you create this widget.
<code>columns</code>	A set of column identifier strings to recognize columns within the widget.
<code>cursor</code>	It specifies the appearance of the mouse cursor when it is over the widget.
<code>displaycolumns</code>	Selects columns to be displayed and specify their presentation order.
<code>height</code>	To specify height of the widget in terms of rows.
<code>padding</code>	To place extra space around the contents inside the widget in the order of Left, Top, Right and Bottom.
<code>selectmode</code>	It controls the user selection with mouse. Values can be: <code>browse</code> : The user may select only one item at a time. <code>extended</code> : The user may select multiple items at once. <code>none</code> : The user cannot select items with the mouse.
<code>show</code>	To suppress the labels at the top of each column, specify <code>show="tree"</code> .

Table 8.18: Options parameters supported by Treeview widget

The following are some of the methods used with a `Treeview` widget:

Method	Description
<code>index(iid)</code>	This method returns the index of the item with the specified iid relative to its parent, counting from zero.
	This method adds a new item to the tree and returns the item's

<code>insert(parent, index, iid)</code>	iid value.
<code>item(iid)</code>	Use this method to set or retrieve the options within the item specified by iid.
<code>move(iid, parent, index)</code>	Move the item specified by iid to the values under the item specified by parent at the position index.
<code>next(iid)</code>	If the item specified by iid is not the last child of its parent, this method returns the iid of the following child; if it is the last child of its parent, this method returns an empty string.
<code>prev(iid)</code>	If the item specified by iid is not the first child of its parent, this method returns the iid of the previous child.

Table 8.19: Methods supported by Treeview widget

Example 1

To demonstrate the working of the hierarchical Treeview widget, we have the following syntax:

```
import tkinter as t
from tkinter import ttk

# Creating a top level window
window1 = t.Tk()
window1.title('Treeview widget Demo')
window1.geometry('500x200')
ttk.Label(window1, text="Demo Hierarchical
Treeview").pack()

# Creating treeview window
treehierarchy = ttk.Treeview(window1)
# Inserting Parent Node
treehierarchy.insert('', '0', 'item1', text='Programming
Language')
# Inserting child nodes (Sub-files) under parent node
```

```

treehierarchy.insert('', '1', 'item2', text='Python')
treehierarchy.insert('', '2', 'item3', text='Java')
treehierarchy.insert('', '3', 'item4', text='C++')
# Inserting more than one attribute of an item
treehierarchy.insert('item2', 'end', 'List',
text='List')
treehierarchy.insert('item2', 'end', 'Tuple',
text='Tuple')
treehierarchy.insert('item2', 'end', 'Dictionary', text='Dictionary')
treehierarchy.insert('item3', 'end', 'J2SE',
text='J2SE')
treehierarchy.insert('item3', 'end', 'J2EE',
text='J2EE')
treehierarchy.insert('item4', 'end', 'Class',
text='Class')
treehierarchy.insert('item4', 'end', 'Object',
text='Object')

# Placing each child items in parent widget
treehierarchy.move('item2', 'item1', 'end')
treehierarchy.move('item3', 'item1', 'end')
treehierarchy.move('item4', 'item1', 'end')
# Calling pack method on the treeview
treehierarchy.pack()
window1.mainloop()

```

Output:

The output can be seen in [Figure 8.12](#):

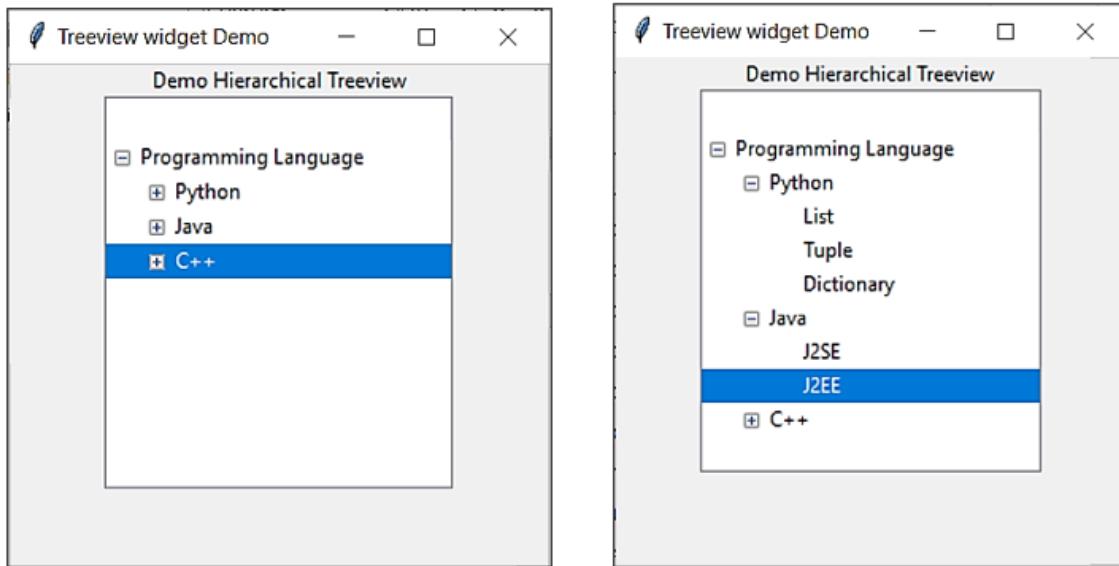


Figure 8.12: Output

Example 2

To demonstrate the working of the **Treeview** table widget, we have the following syntax:

```
import tkinter as tk
from tkinter import ttk, PhotoImage
window1 = tk.Tk()

# Specify image path for first column having folder
icon

file_img_folder = r"D:/images_folder/folder1.png"

# Load and convert image to a PhotoImage object that
tkinter can use

tk_img_folder = PhotoImage(file=file_img_folder)

# Define column names for reference the columns.

column_names = ("product_name_col", "price_col")

# Assign the column names when making the treeview
table of products
```

```
treeview_products = ttk.Treeview(columns=column_names)

# Assign text values for headings of treeview table
treeview_products

treeview_products.heading("product_name_col",
text="Shape")

treeview_products.heading("price_col", text="Price")

# Insert the parent rows

treeview_products.insert(parent="", iid="electronics",
index="end", image=tk_img_folder, values=
("Electronics",))

treeview_products.insert(parent="", iid="mobile",
index="end", image=tk_img_folder, values=("Mobile",))

treeview_products.insert(parent="", iid="laptops",
index="end", image=tk_img_folder, values=("Laptops",))

# Insert sub-rows for Electronics

treeview_products.insert(parent="electronics",
index="end", values=("T.V.", "Rs.20K"))

treeview_products.insert(parent="electronics",
index="end", values=("Refrigerator", "Rs.15K"))

# Insert sub-rows for Mobile

treeview_products.insert(parent="mobile", index="end",
values=("Apple", "Rs.70K"))

treeview_products.insert(parent="mobile", index="end",
values=("Samsung", "Rs.20K"))

# Insert sub-rows for Laptops

treeview_products.insert(parent="laptops", index="end",
values=("Dell", "Rs.30K"))

treeview_products.insert(parent="laptops", index="end",
values=("HP", "Rs.35K"))
```

```

# Make the image column's width 70 pixels wide, so it
# looks nicer.

treeview_products.column("#0", width=70)

treeview_products.pack()

window1.mainloop()

```

Output:

The output is shown in *Figure 8.13*:

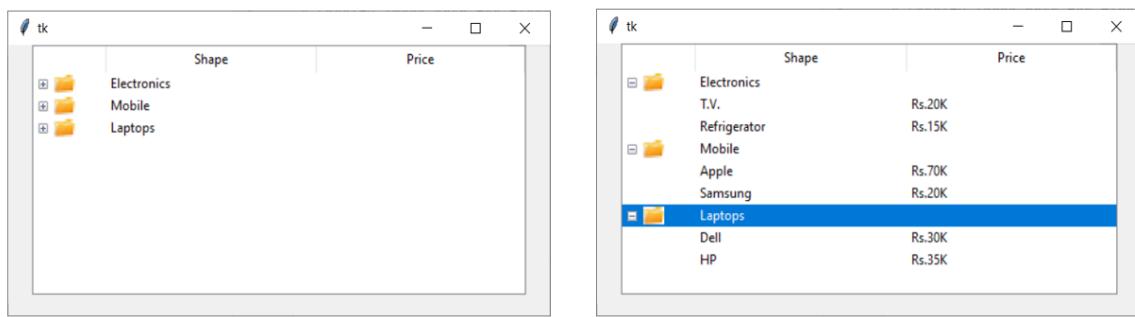


Figure 8.13: Output

Label frame

A **labelframe** is a type of container widget that contains other related widgets inside it. For example, one can group **Checkbuttons**, **Radiobuttons**, Labels, and other widgets and place the group heading on a **LabelFrame**. Thus, this widget offers the combined features of a frame container and label widget. The syntax of **labelFrame** is as follows:

```
labelObject = LabelFrame(container, option=value, ...)
```

The various options associated with **LabelFrame** are as follows:

Option	Description
bg	It specifies the background color of the label frame.
bd	It denotes size of the border. Default value is 2 pixels.
cursor	It specifies the shape of cursor when it hovers in the widget.
font	It configures the font type inside label frame.

height	The vertical dimension of the new frame.
labelAnchor	It denotes the place where the label is placed.
relief	It represents the border style such as FLAT, GROOVE, SUNKEN, and so on. The default value is GROOVE.
text	It enables to specify a string to be displayed inside the widget.
width	To give desired width for the window.

Table 8.20: Options parameters supported by label frame widget

Let us demonstrate Labelframe with an example:

```
import tkinter as tk
from tkinter import *
window1 = tk.Tk()
lf1 = LabelFrame(window1, text="Mark Gender")
lf1.pack()
val = IntVar()
Radiobutton(lf1, text="Male", variable=val, value=1).pack()
Radiobutton(lf1, text="Female", variable=val, value=2).pack()
lf2 = LabelFrame(window1, text="Choose Hobbies")
lf2.pack()
Checkbutton(lf2, text="Reading").pack()
Checkbutton(lf2, text="Cooking").pack()
Checkbutton(lf2, text="Swimming").pack()
window1.mainloop()
```

Output:

The output is shown in [Figure 8.14](#):

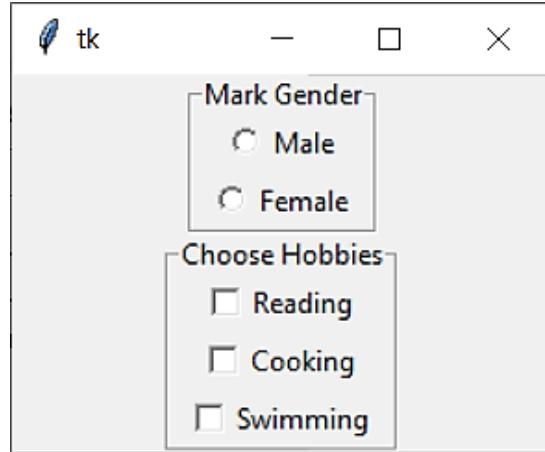


Figure 8.14: Output

Messagebox

The messagebox widget is a special pop-up box that displays some information or conveys a quick message. We must import the **messagebox** module of the tkinter library. There are six types of **messagebox** widgets such as:

1. **showinfo**: This type of message box displays a message with a sound and returns “OK”.
2. **showwarning**: Displays warning message with sound. Returns “OK”.
3. **showerror**: Displays error message with sound. Returns “OK”.
4. **askquestion**: Prompts “Yes” and “No” options. Returns Yes or No.
5. **askyesno**: Prompts with “Yes” and “No” options. Yes returns 1, and No returns 0.
6. **askretrycancel**: Prompts with the option to “retry” or “cancel”. With sound, Retry returns 1 and cancel returns 0.

Example to demonstrate various types of **messagebox**:

```
from tkinter.messagebox import *
# Showing various messages in messagebox
print(askokcancel("askokcancel Demo", "Ok or Cancel"))
print(askquestion("askquestion Demo", "Answer this
Question?"))
```

```

print(askretrycancel("askretrycancel Demo", "Retry or Cancel"))
print(askyesno("askyesno Demo", "Yes or No"))
print(askyesnocancel("askyesnocancel Demo", "Yes or No or Cancel"))
print(showerror("showerror Demo", "Error Message"))
print(showinfo("showinfo Demo", "Information Message"))
print(showwarning("showwarning Demo", "Warning Message!!"))

```

Output:

The output can be seen in *Figure 8.15*:

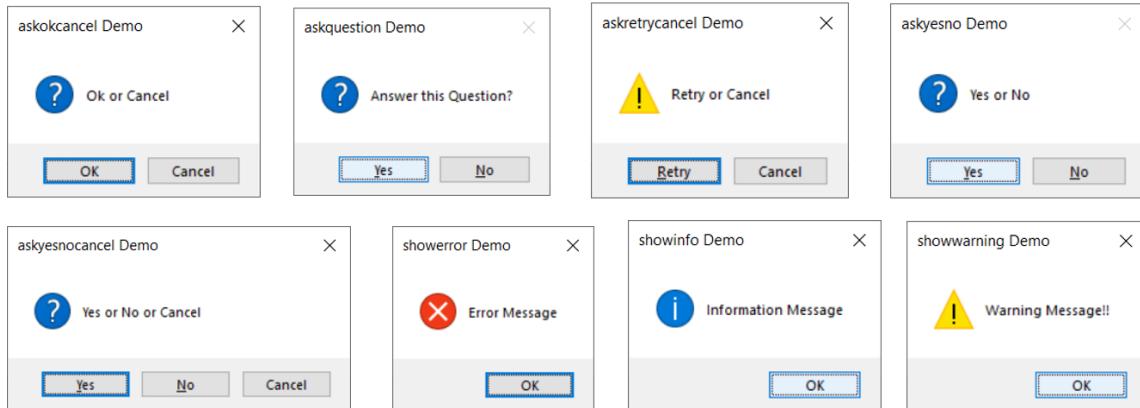


Figure 8.15: Output

Tkinter filedialog

We often require providing a dialog that allows file selections while working with a Tkinter GUI application that deals with the file system. For accomplishing this task, we can import and use the **tkinter.filedialog** module. Let us consider a list of all the different Dialog options available in Tkinter:

Returning a file path

Once we select a file, its file path will be returned back into the Python program for use with the help of the **askopenfilename()** method. Remember, you can

only select files with this, not folders. Use the `askopenfilenames()` function if you want to select multiple files. Let us see an example to select open a file and read the contents written in it to display on the terminal output:

```
import tkinter as tk
from tkinter import filedialog
filepath = filedialog.askopenfilename(initialdir="/",
title="Select a file to open",filetypes=(("txt files",
"*.txt"), ("all files", "*.*")))
```

Output:

The output can be seen in *Figure 8.16*:

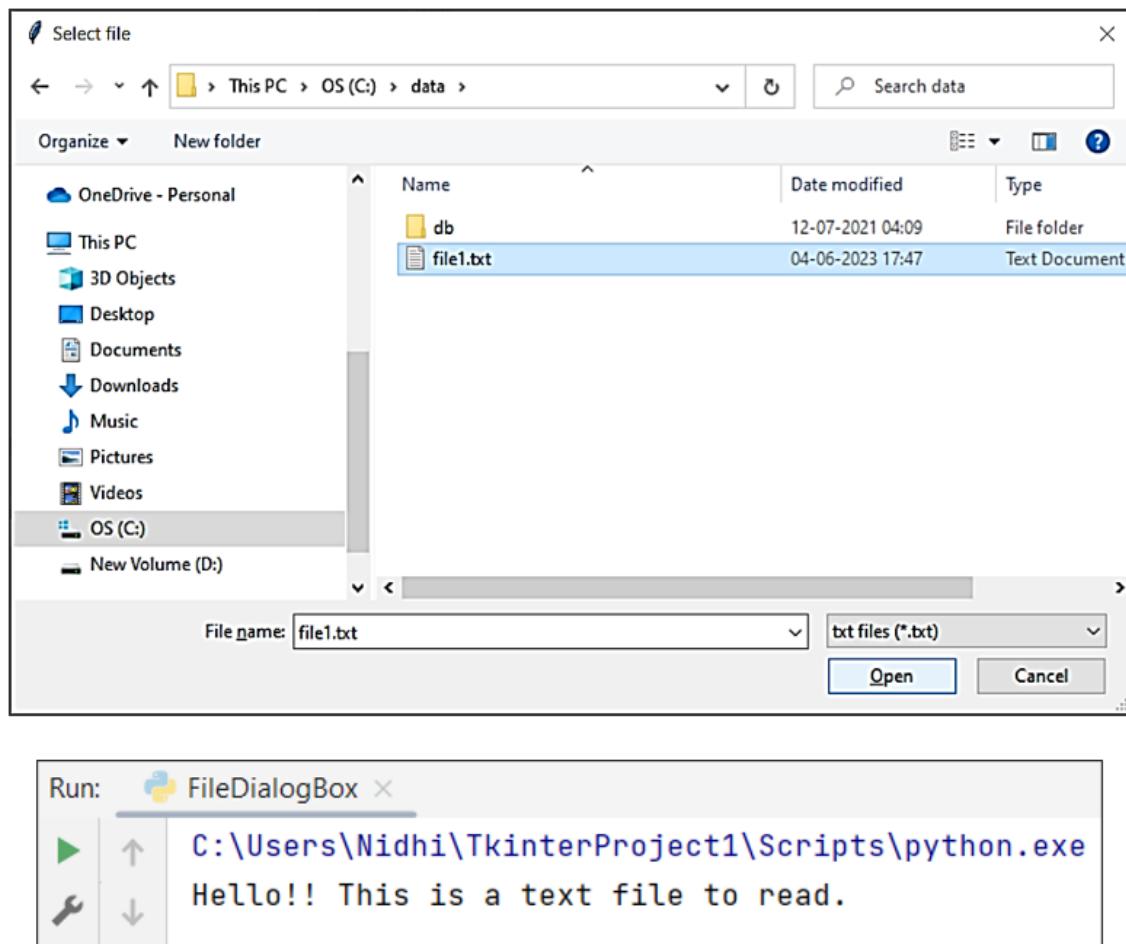


Figure 8.16: Output

The optional parameters used in the method `filedialog.askopenfilename()` in the preceding code example are described as follows:

- **title**: The title name is a string that appears on the file dialog box.
- **initialdir**: The initial directory location that the file dialog starts in.
- **initialfile**: It refers to the file selected upon opening of the file dialog.
- **filetypes**: It determines the file type to be loaded and/or saved using the file dialog box. The options are passed in the form of a tuple format. The * wild card can be used to allow selection of all filetypes; otherwise, specific file extensions can also be specified.
- **defaultextension**: This option only applies to the save file dialog box. It specifies the default file extension to be applied for selection.
- **multiple**: When this option is set **True**, it enables selecting multiple files.

Note: These options are used with same definition with other `filedialog` methods also.

Saving a file

To save a file at a particular place, use the `asksaveasfile()` function, we have the following syntax:

```
import tkinter as tk
from tkinter import filedialog
filepath = filedialog.asksaveasfile(initialdir="/",
title="Select to Save file",filetypes=(("txt files",
".txt"),("all files", "*.*")))
print(filepath.read())
```

Output:

The output can be seen in *Figure 8.17*:

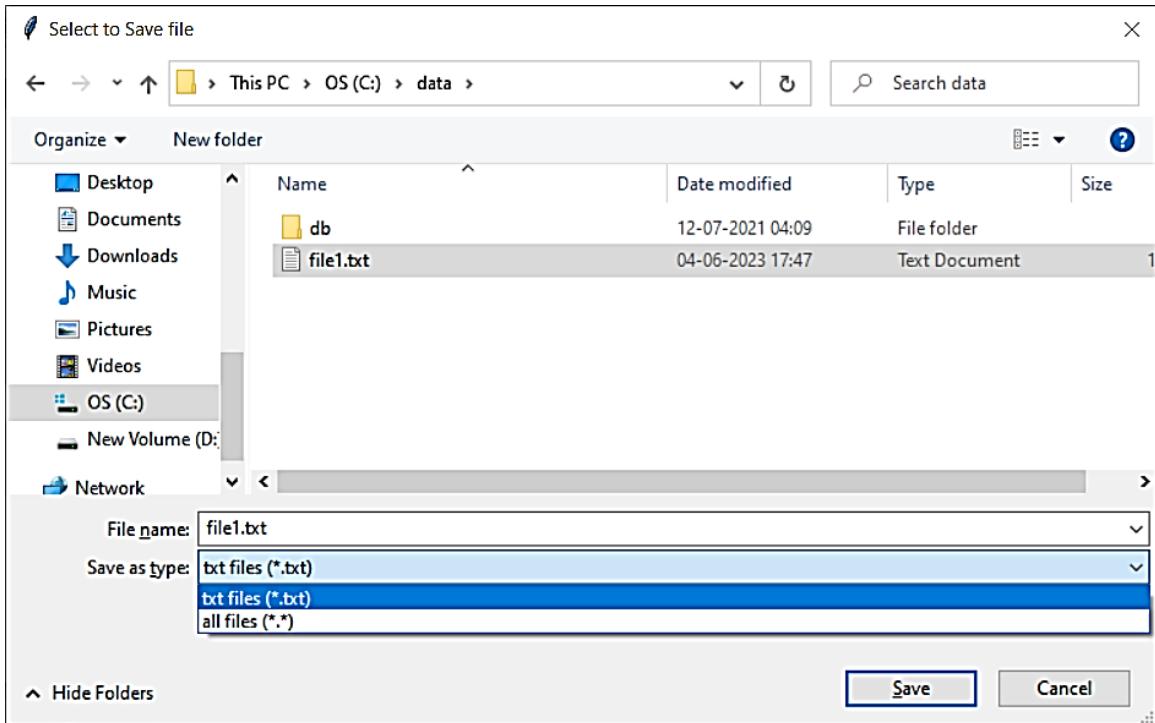


Figure 8.17: Output

Select directory

The **askdirectory()** method is identical to the **askopenfilename()** method, with the exception that it picks directories (folders) rather than files and returns their file paths. For example, to open the **Downloads** directory in the root directory, we can use the following code:

```
import tkinter as tk
from tkinter import filedialog
filepath =
filedialog.askdirectory(initialdir="/Downloads",
title="Select file")
print(filepath)
```

Output:

The output can be seen in [Figure 8.18](#):

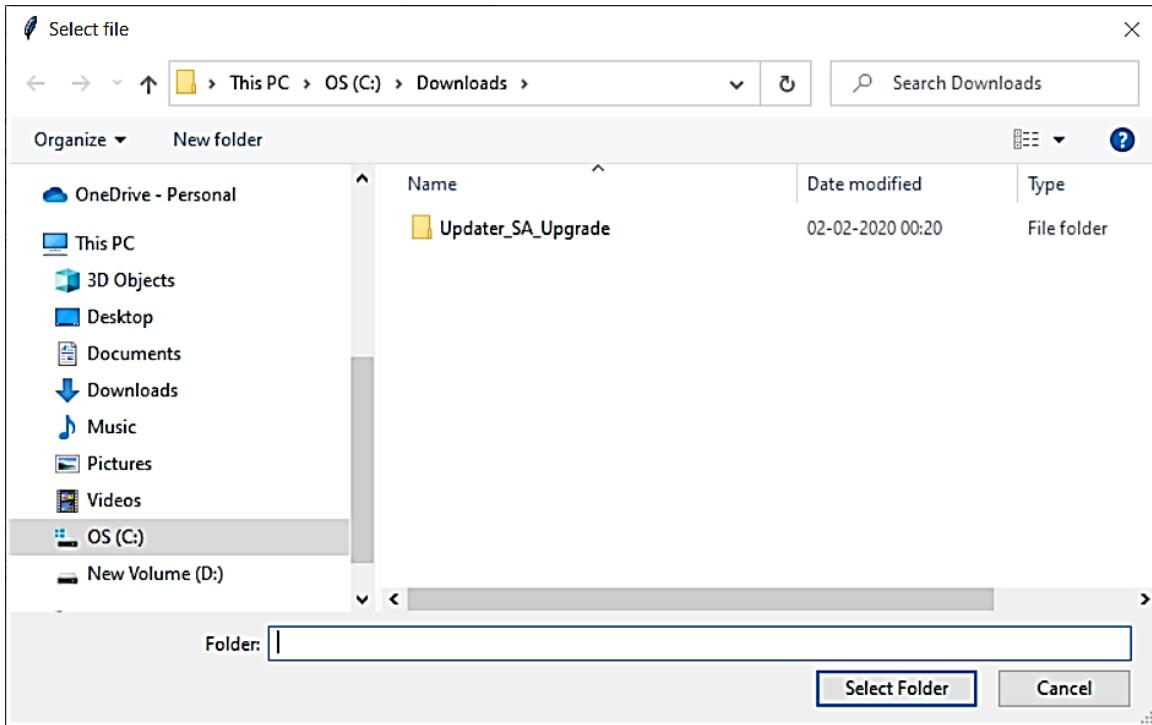


Figure 8.18: Output

Geometry management in Tkinter

The geometry manager is a tool that Tkinter uses to organise widgets on a window. We have so far learnt how to add widgets to a window using the **pack()** function. One of Tkinter's three geometry managers is the **pack()** function. **Grid()** and **place()** are the additional geometry managers. Widgets are placed around the container's edges by the pack. The root window or a frame might serve as the container. Let us discuss the organization of widgets inside a Tkinter container in detail.

Organizing widgets with layout managers

We use some layouts, such as Pack Layout, Grid Layout, and Place Layout, for positioning or arranging widgets in specific locations on our GUI applications. **Geometry Manager** is the formal name for these layout designs.

Pack layout

Among the three layout managers, the Pack layout is the most user-friendly. We can specify the relative placements of widgets using the pack command. It is the

simplest layout manager to use. Pack geometry manager is recommended for creating small and simple GUI applications. The syntax is as follows:

`widget.pack(option=value..)`

Various options available for configuring organization of widgets on screen are listed as follows:

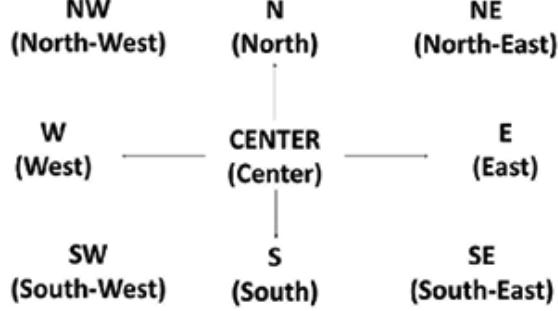
Options	Description
<code>fill</code>	The fill option fills or covers the space available horizontally or vertically. Values are as follows: <code>tkinter.X</code> : To fill space horizontally along the X-axis. <code>tkinter.Y</code> : To fill space vertically along the Y-axis. <code>tkinter.BOTH</code> : To fill space in both the X- and Y-axis directions.
<code>expand</code>	It specifies if the widgets should expand to fill extra available space in the layout or not. Default value is <code>False</code> .
<code>side</code>	It specifies the alignment of the widget by using the following options: <code>tkinter.TOP</code> , <code>tkinter.LEFT</code> , <code>tkinter.RIGHT</code> , and <code>tkinter.BOTTOM</code>
<code>anchor</code>	It allows you to anchor or place the widget to the edge of the allocated layout space. The values that can be used for location are as follows: NW, N, NE, CENTER, SW, S, and SW: 
<code>ipadx</code> , <code>ipady</code>	<code>ipadx</code> is padding inside the widget on the left and right sides, that is, padding along the X-axis. <code>ipady</code> is padding inside the widget on the top and bottom side, that is, padding along the Y-axis.
<code>padx</code> , <code>pady</code>	It specifies the number of padding spaces added outside widget's border.

Table 8.21: Options parameters supported by Pack layout

Let us consider an example of demonstrating widget organization using **pack()**:

```
import tkinter as tk

window1 = tk.Tk()
window1.title('Pack Demo')
window1.geometry("300x200")

# First place widgets top down vertically
label1 = tk.Label(window1, text='Top Block', bg="red")
label1.pack(ipadx= 10, ipady= 10, fill=tk.X)

label2 = tk.Label(window1, text='Center Block',
bg="yellow")
label2.pack(ipadx= 10, ipady= 10, fill=tk.X)

label3 = tk.Label(window1, text='Lower Block',
bg="cyan")
label3.pack(ipadx= 10, ipady= 10, fill=tk.X)

# Next place widgets side by side horizontally
label4 = tk.Label(window1, text='Left Block',
bg="green")
label4.pack(ipadx=10, ipady=10, expand=True, fill=tk.BOTH,
side=tk.LEFT)

label5 = tk.Label(window1, text='Center Block',
bg="blue", )
label5.pack(ipadx=10, ipady=10, expand=True, fill=tk.BOTH,
side=tk.LEFT)

label6 = tk.Label(window1, text='Right Block',
bg="orange")
label6.pack(ipadx=10, ipady=10, expand=True, fill=tk.BOTH,
side=tk.LEFT)
```

```
window1.mainloop()
```

Output:

Figure 8.19 shows the output:

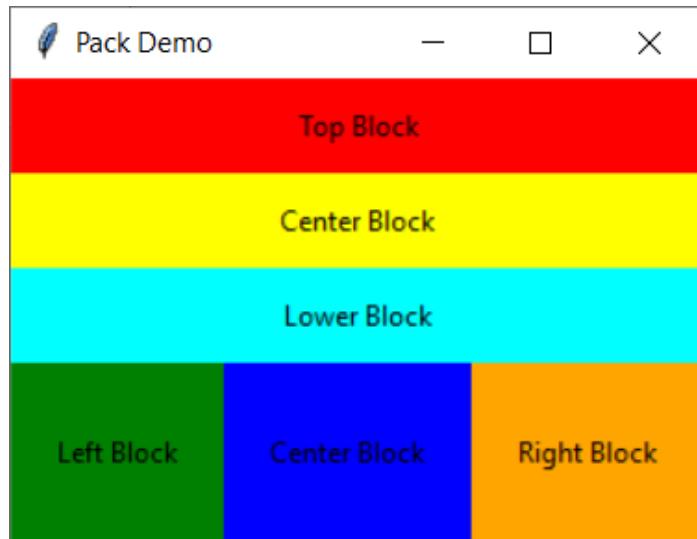


Figure 8.19: Output

Grid layout

The grid layout manager divides the entire area into rows and columns, similar to a 2-D table. It is a very versatile layout manager and, thus, popular among users. The intersection point of a row and a column is known as a *cell*, inside which a widget is placed. Note that a cell can hold only one widget at a time. If more widgets are placed in the same cell, then these will stack on top of each other. The following figure shows how both the row and column starts from 0 in the grid:

	Column 0	Column 1	Column 2
Row 0	Cell (0,0)	Cell (0,1)	Cell (0,2)
Row 1	Cell (1,0)	Cell (1,1)	Cell (1,2)
Row 2	Cell (2,0)	Cell (2,1)	Cell (2,2)

Note: Cell Number => (Row Number, Column Number)

Figure 8.20: Cell numbering in Grid Layout

The syntax is as follows:

```
widget.grid(option=value..)
```

The various options that can be used as parameters with **grid()** are as follows:

Option	Description
column	The column index where you want to place the widget.
row	The row index where you want to place the widget.
rowspan	Set the number of adjacent rows that the widget can span.
columnspan	Set the number of adjacent columns that the widget can span.
ipadx, ipady	ipadx is padding inside the widget on the left and right side, that is, padding along the X-axis. ipady is padding inside the widget on the top and bottom side, that is, padding along the Y-axis.

padx, pady	It specifies the number of padding spaces added outside widget's border.
Sticky	It is used to indicate the location of the widget within the cell if any cell is larger than a widget. The position of the widget is represented by sticky characters: N (North), S(South), E(East), W(West), NW(North West), NE(North East), SE(South East), SW(South West), NS(To stretch the widget vertically), EW(To stretch the widget horizontally), and NESW (widget takes up the full area of the cell).

Table 8.22: Options parameters supported by Grid layout

Let us demonstrate the working of **grid()** layout with an example:

```
import tkinter as tk
window1 = tk.Tk()
for i in range(3):
    for j in range(3):
        frame = tk.Frame(window1, borderwidth=1,
relief='sunken')
        frame.grid(row=i, column=j, padx=2, pady=2)
        label = tk.Label(master=frame, text=f"Row
{i}\nColumn {j}")
        label.pack()
window1.mainloop()
```

Output:

Refer to [Figure 8.21](#):

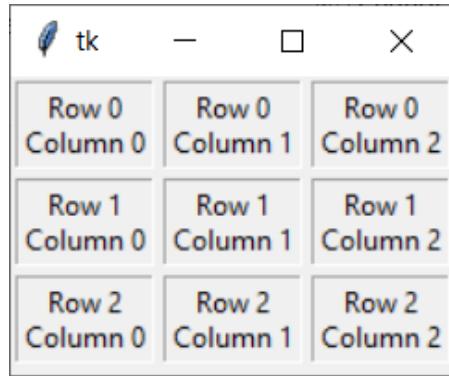


Figure 8.21: Output

Place layout

Place layout manager organizes the widget according to the specified x and y coordinates (x, y). Here, both x and y inputs are in pixels. The origin place is the top-left corner of the Frame or the window where the x and y coordinates are both 0. The syntax is as follows:

```
widget.place(option=value..)
```

Refer to [Table 8.23](#):

Option	Description
x, y	This option indicates the horizontal and vertical offset in the pixels.
height, width	the height and weight of the widget in the pixels.
anchor	Denotes the position of the widget within the container similar to anchor in pack().
relheight, relwidth	Represents a float value between 0.0 and 1.0 fraction of the parent's height and width.

Table 8.23: Options parameters supported by Place layout

Let us consider an example for `place()` geometry manager:

```
from tkinter import *
window1 = Tk()
window1.geometry("300x300")
usernameLabel = Label(window1, text = "Username")
```

```
usernameLabel.place(x = 50, y = 50)
emailLabel = Label(window1, text = "Email").place(x = 50, y = 100)
passwordLabel = Label(window1, text = "Password").place(x = 50, y = 150)
textbox1 = Entry(window1).place(x = 130, y = 50)
textbox2 = Entry(window1).place(x = 130, y = 100)
textbox3 = Entry(window1).place(x = 130, y = 150)
button1 = Button(window1, text="Submit").place(x=100, y=210)
window1.mainloop()
```

Output:

The output is as follows:

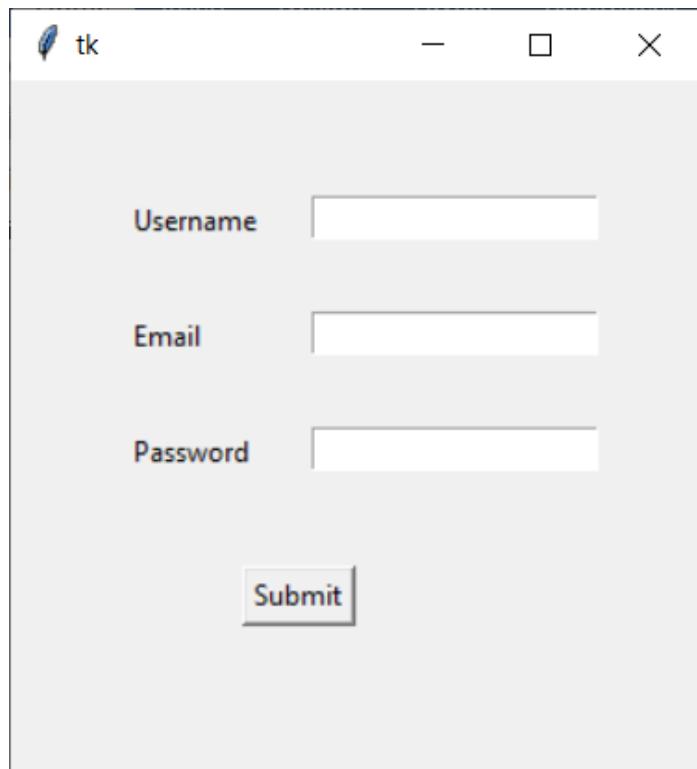


Figure 8.22: Output

Event binding

To develop an interactive and user-friendly application, it is required for the widgets to respond to the simple Mouse and Key events. To accomplish this, a callback function must be associated to handle a specific event. This process is known as **Event Binding**. The user need not explicitly call the event handler; rather, the assigned function is invoked automatically when the associated event occurs. There are the following two techniques available in Tkinter for binding events:

- **Command option:** In Tkinter, several widgets, such as Button, enable users to associate a callback function with an event by using *Command Binding*, where we can assign the name of a function to the command option of the widget. We have seen examples of this technique with the buttons widget.
- **bind():** Not all Tkinter widgets support the command option. Therefore, Tkinter provides you with an alternative way for event binding via the **bind()** method. The following shows the general syntax of the **bind()** method. The syntax is as follows:

```
widget.bind(event, event handler, add= None)
```

If the specified event is raised in the widget, then the associated **handler** function is called using an event object describing the event. We can also associate multiple event handlers that respond to the same event. In order to register an additional handler with the widget, we can pass the “+” character to the add argument.

Several events that can be associated with the Mouse and keyboard in Tkinter are listed in the following table:

Widget	Event	Description
Mouse	<Button-1>	Mouse left button click.
	<Button-2>	Mouse center button click.
	<Button-3>	Mouse right button click.
	<B1-Motion>	Mouse left button press and move
	<ButtonRelease-1>	Mouse Left button release, ButtonRelease—2 for Middle and 3 for right button
	<Double-Button-1>	Left Mouse key is double-clicked

Widget	Event	Description
	<Enter>	Mouse Entry over a widget
	<Leave>	Mouse Leave a widget
	<MouseWheel>	Mouse Wheel Up or Down rotation
Keyboard	<Key>	Any Key is pressed
	<Return>	Enter Key is pressed
	<KeyPress>	Key Press event of any widget
	<KeyRelease>	Key Release event of any widget
	<FocusIn>	When widget got focus
	<FocusOut>	When widget lost focus
	<Right>	When Right arrow is pressed
	<Left>	When Left arrow is pressed
	<Up>	When Up arrow is pressed
	<Down>	When Down arrow is pressed

Table 8.24: List of mouse and key events

Let us consider an example for handling these mouse and key events in Tkinter Python:

```
import tkinter as tk
from tkinter import *
window1 = tk.Tk()
window1.geometry("500x500")
def mouse_event(event):
    label1.config(text='Clicked
at : '+str(event.x)+", "+str(event.y))
label1 = tk.Label(window1, text='Label', bg='yellow',
font=('Times', 26, 'normal'))
label1.grid(row=0, column=1, padx=10, pady=10)
```

```

button1 =
Button(text="Click",height=5,width=10).place(x=100,y=10
0)

window1.bind( '<Button-1>',mouse_event)    # Mouse Left
click

window1.bind( '<Button-2>',mouse_event)    # Mouse middle
button click

window1.bind( '<Button-3>',mouse_event)    # Mouse right
button click

window1.bind( '<B1-Motion>',mouse_event)# Mouse left btn
pressed move

window1.bind( '<ButtonRelease-1>',mouse_event) # Mouse
left release

window1.bind( '<Double-Button-1>',mouse_event)    #
Double click

label1.bind( '<Enter>',lambda
e:label1.config(text='Inside Window'))    # Mouse enters
label

label1.bind( '<Leave>',lambda
e:label1.config(text='Outside Window'))    # Mouse
leaves label

window1.bind( '<MouseWheel>',mouse_event) # Mouse middle
button click

window1.mainloop()

```

Output:

The output can be seen in [Figure 8.23](#):

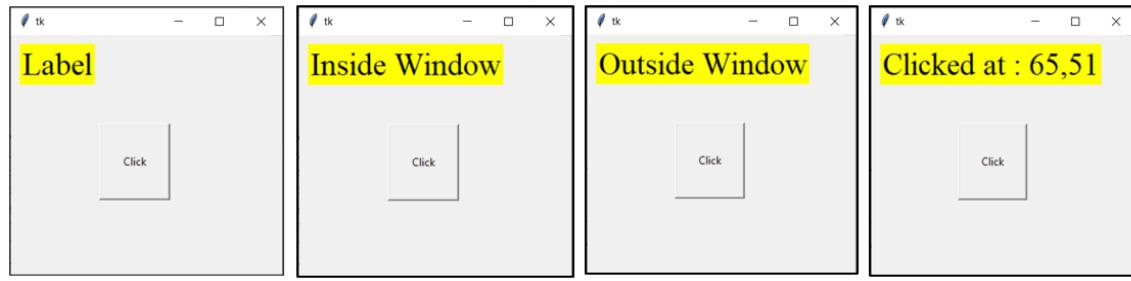


Figure 8.23: Output

Conclusion

This chapter introduced users to the development of GUI-based standard desktop applications in Python by using the Tkinter library and its widgets. Users can create various interactive and user-friendly applications that work on the concept of Event Binding. Some actions may be performed in the application when a specific mouse or key event is initiated by the user, such as mouse click, double-click, key press, and so on. Users can also integrate the Tkinter library with SQLite3 as well for storing and managing user data in the application.

Points to remember

- GUI stands for Graphical User Interface, which represents simple visual interaction between the user and machine instead of typing commands for a specific task.
- Python has several frameworks for creating interactive GUI applications, such as JPython, wxPython, and Tkinter.
- Any activity that takes place during the event loop and may cause the application to behave in a certain way, such as the pressing of a key or mouse button, is an Event.
- For the events that we use in your application in Tkinter, we write functions referred to as Event Handlers.
- The Python Tkinter module supports numerous types of widgets, such as Button, Label, Entry, Text, MessageBox, and so on, for creating GUI applications.
- We use some layouts, such as Pack Layout, Grid Layout, and Place Layout, for positioning or arranging widgets in specific locations on our GUI

applications.

Exercise

Attempt the following project.

Sample project with solution

Create a GUI application for a Café to order coffee for users. This application uses the **tkinter** library along with **sqlite3** for database management. The application works for managing customer orders and taking their feedback for further improvements in quality. The total payment to be done is calculated and filled in **Entry** widgets automatically on button click. The user can also view the menu card with a button click. The orders are constantly visible in a tabular format on the screen:

```
from tkinter import ttk
from tkinter import messagebox
import sqlite3
from tkinter import *

def system():
    root = Tk()
    root.geometry("1100x500")
    root.title("Cafe Coffee Heights Order")

def Database():
    global connectn, cursor
    connectn = sqlite3.connect("Cafe.db")
    cursor = connectn.cursor()
    # creating bill table
    cursor.execute("CREATE TABLE IF NOT EXISTS
OrderRecords(ordno text,fil_cof text,cafe_lat
```

```
text,cafe_moc text, cold_cof text, ct text, taxes text,
tot text)")

# variable datatype assignment
ordernum = StringVar()
filter_coffee = StringVar()
cafe_latte= StringVar()
cafe_mocha = StringVar()
cold_coffee = StringVar()
cost = StringVar()
tax = StringVar()
total = StringVar()

# defining total function
def tottal():
    # fetching the values from entry box
    order = (ordernum.get())
    fc = float(filter_coffee.get())
    cl = float(cafe_latte.get())
    cm = float(cafe_mocha.get())
    cc = float(cold_coffee.get())

    # computing the cost of items

    costfc = fc * 40
    costcl = cl * 110
    costcm = cm * 150
    costcc = cc * 180
```

```
# computing the charges
costofmeal = (costfc+costcl+costcm+costcc)
ptax = ((costfc+costcl+costcm+costcc) * 0.20)
paidtax = str(ptax)
overall = str(ptax + costofmeal)

# Displaying the values
cost.set(costofmeal)
total.set(overall)
tax.set(paidtax)

# defining reset function
def reset():
    ordernum.set("")
    filter_coffee.set("")
    cafe_latte.set("")
    cafe_mocha.set("")
    cold_coffee.set("")
    cost.set("")
    tax.set("")
    total.set("")

# defining exit function
def exit():
    root.destroy()

# Topframe
topframe = Frame(root, bg="white", width=1600,
height=50)
```

```
topframe.pack(side=TOP)

# Leftframe

leftframe = Frame(root, width=1000, height=900)
leftframe.pack(side=LEFT)

# rightframe

rightframe = Frame(root, width=1000, height=900)
rightframe.pack(side=RIGHT)

# Display data

def DisplayData():

    Database()

    my_tree.delete(*my_tree.get_children())

    cursor = connectn.execute("SELECT * FROM
OrderRecords")

    fetch = cursor.fetchall()

    for data in fetch:

        my_tree.insert('', 'end', values=(data))

    cursor.close()

    connectn.close()

# Create table

my_tree = ttk.Treeview(rightframe)

my_tree['columns'] = ("orderNo", "filter coffee",
"cafe latte", "cafe mocha", "cold coffee", "Price",
"Tax", "Total")

# create scrollbars for table
```

```
    horizontal_bar = ttk.Scrollbar(rightframe,
orient="horizontal")
    horizontal_bar.configure(command=my_tree.xview)

my_tree.configure(xscrollcommand=horizontal_bar.set)
    horizontal_bar.pack(fill=X, side=BOTTOM)

    vertical_bar = ttk.Scrollbar(rightframe,
orient="vertical")
    vertical_bar.configure(command=my_tree.yview)
my_tree.configure(yscrollcommand=vertical_bar.set)
    vertical_bar.pack(fill=Y, side=RIGHT)

# defining column for table
my_tree.column("#0", width=0, minwidth=0)
    my_tree.column("orderNo",
anchor=CENTER, width=80, minwidth=25)

my_tree.column("filtercoffee", anchor=CENTER, width=80, mi
nwidth=25)

    my_tree.column("cafe latte", anchor=CENTER,
width=80, minwidth=25)

    my_tree.column("cafe mocha", anchor=CENTER, width=80,
minwidth=25)

my_tree.column("cold coffee", anchor=CENTER, width=80,
minwidth=25)

my_tree.column("Price", anchor=CENTER, width=80,
minwidth=25)

my_tree.column("Tax", anchor=CENTER, width=80,
minwidth=25)
```

```
my_tree.column("Total", anchor=CENTER, width=80,
minwidth=25)

    # defining headings for table

my_tree.heading("orderNo", text="Order No",
anchor=CENTER)

my_tree.heading("filtercoffee", text="Filter
Coffee", anchor=CENTER)

my_tree.heading("cafe latte", text="Cafe
Latte", anchor=CENTER)

my_tree.heading("cafe mocha", text="Cafe
Mocha", anchor=CENTER)

my_tree.heading("cold coffee", text="Cold
Coffee", anchor=CENTER)

my_tree.heading("Price", text="Cost", anchor=CENTER)

my_tree.heading("Tax", text="Tax", anchor=CENTER)

my_tree.heading("Total", text="Total", anchor=CENTER)

my_tree.pack()

DisplayData()

    # defining add function to add record

def add():

    Database()

    # getting data

    orders = ordernum.get()

    fc1 = filter_coffee.get()

    cl1 = cafe_latte.get()

    cm1 = cafe_mocha.get()
```

```

cc1 = cold_coffee.get()
costs = cost.get()
taxes = tax.get()
totals = total.get()

if orders == "" or fc1 == "" or cl1 == "" or
cm1 == "" or cc1 == "" or costs == "" or taxes == "" or
totals == "":
    messagebox.showinfo("Error!!","Fill all
fields!!!!")

print(orders,fc1,cl1,cm1,cc1,costs,taxes,totals)

else:
    connectn.execute('INSERT INTO OrderRecords
VALUES (?,?,?,?,?,?,?,?,?)',(orders,
fc1,cl1,cm1,cc1,costs,taxes,totals));

    connectn.commit()

    messagebox.showinfo("Message","Stored Data")

# refresh table data

    DisplayData()

    connectn.close()

# defining function to access data from sqlite database

def DisplayData():

    Database()

    my_tree.delete(*my_tree.get_children())

    cursor=connectn.execute("SELECT * FROM
OrderRecords")

    fetch=cursor.fetchall()

```

```
        for data in fetch:
            my_tree.insert('', 'end', values=(data))
        cursor.close()
        connectn.close()

# defining function to delete record
def Delete():

# open database
    Database()
    if not my_tree.selection():
        messagebox.showwarning("Error!!", "No Data
selected to Delete!!")
    else:
        result =
messagebox.askquestion('Confirmation', 'Confirm to
delete the record?', icon="warning")
        if result == 'yes':
            curItem = my_tree.focus()
            contents = (my_tree.item(curItem))
            selecteditem = contents['values']
            my_tree.delete(curItem)
            cursor = connectn.execute("DELETE FROM
OrderRecords WHERE ordno= %d" % selecteditem[0])
            connectn.commit()
            cursor.close()
            connectn.close()

# Top heading
```

```
    main_lbl = Label(topframe, font=('Calibri', 30, 'bold'), text="**** Cafe Coffee Heights Order System ****", fg="navy blue", anchor=W)

    main_lbl.grid(row=0, column=0)

    labelframe=LabelFrame(leftframe, text="Fill Order Details")

    labelframe.grid(column=50, row=50)

    ordlbl = Label(labelframe, text="Order No.", fg="black", bd=5, anchor=W).grid(row=1, column=0)

    ordtxt = Entry(labelframe, insertwidth=4, justify='left', bd=5, text=num, textvariable=ordernum).grid(row=1, column=1)

    filtercoffeelbl = Label(labelframe, text="Filter Coffee", bd=5, fg="black", anchor=W).grid(row=3, column=0)

    filtercoffeetxt = Entry(labelframe, insertwidth=4, bd=5, justify='left', textvariable=filter_coffee).grid(row=3, column=1)

    cafelattelbl = Label(labelframe, text="Cafe Latte", bd=5, fg="black", anchor=W).grid(row=4, column=0)

    cafelattetxt = Entry(labelframe, insertwidth=4, bd=5, justify='left', textvariable=cafe_latte).grid(row=4, column=1)

    cafemochalbl = Label(labelframe, text="Cafe Mocha", bd=5, fg="black", anchor=W).grid(row=5, column=0)

    cafemochatxt = Entry(labelframe, insertwidth=4, bd=5, justify='left', textvariable=cafe_mocha).grid(row=5, column=1)

    coldcoffeelbl = Label(labelframe, text="Cold Coffee", bd=5, fg="black", anchor=W).grid(row=6, column=0)
```

```
    coldcoffeetxt = Entry(labelframe,
insertwidth=4, bd=5,
justify='left', textvariable=cold_coffee).grid(row=6,
column=1)

    costlbl = Label(labelframe, text="Cost", bd=5,
anchor=W).grid(row=7, column=0)

    costtxt = Entry(labelframe,
bd=5, justify='left', textvariable=cost).grid(row=7,
column=1)

    taxlbl = Label(labelframe, text="Tax",
bd=5, anchor=W).grid(row=9, column=0)

    taxtxt = Entry(labelframe,
bd=5, justify='left', textvariable=tax).grid(row=9,
column=1)

    totallbl = Label(labelframe, text="Total",
bd=5, anchor=W).grid(row=11, column=0)

    totaltxt = Entry(labelframe,
bd=5, justify='left', textvariable=total).grid(row=11,
column=1)

    totbtn =
Button(labelframe, text="Calculate", bd=10, bg="Lightgrey",
, command=tottal, width=15).grid(row=23, column=0)

    resetbtn =
Button(labelframe, text="Reset", bd=10, bg="lightgrey", com
mand=reset, width=15).grid(row=23, column=2)

    exitbtn = Button(labelframe, text="Exit",
bd=10, bg="lightgrey", command=exit,
width=15).grid(row=24, column=0)

    addbtn = Button(labelframe, text="Add
Order", bd=10, bg="lightgrey", command=add,
width=15).grid(row=24, column=2)
```

```
    deletebtn = Button(labelframe, text="Delete Record",
bd=10, bg="lightgrey", command=Delete,
width=15).grid(row=25, column=1)

# feedback form

def feedbackk():
    feed = Tk()
    feed.geometry("600x500")
    feed.title("Submit Feedback form")
    connectn = sqlite3.connect("Cafe.db")
    cursor = connectn.cursor()
    cursor.execute("CREATE TABLE IF NOT EXISTS
CustomerFeedback(n text,eid text,feedback5 text,com
text)")
    name = StringVar()
    email = StringVar()
    comments = StringVar()

# defining submit function

def submit():
    n = name.get()
    eid = email.get()
    com = txt.get('1.0', END)
    feedback1 = ""
    feedback2 = ""
    feedback3 = ""
    feedback4 = ""
    if (checkvar1.get() == "1"):
```

```

        feedback1 = "Excellent"

    if (checkvar2.get() == "1"):

        feedback2 = "Good"

    if (checkvar3.get() == "1"):

        feedback2 = "Average"

    if (checkvar4.get() == "1"):

        feedback2 = "Poor"

    feedback5=feedback1+" "+feedback2+
"+feedback3+" " + feedback4

    conn = sqlite3.connect("Cafe.db")

    cursor = conn.cursor()

    cursor.execute("INSERT INTO
CustomerFeedback VALUES ('" + n + "', '" + eid + "', '" +
com + "', '" + feedback5 + "')")

    messagebox.showinfo("Information", "Your
Feedback Saved Successfully!")

    feed.destroy()

# defining cancel button

def cancel():

    feed.destroy()

# label#
    lb1 = Label(feed, font=("vardana", 15, "bold"),
text="We appreciate your feedback!",
fg="red").pack(side=TOP)

    lb12 = Label(feed, font=("vardana", 15),
text="Spare some time to fill the form",fg="navy
blue").pack(side=TOP)

```

```
    Cnamelbl = Label(feed, font=('vardana', 15),
text="Customer Name:", fg="black", bd=10,
anchor=W).place(x=10, y=150)

    Cnametxt=Entry(feed, font=
('vardana', 15), bd=6, insertwidth=2, bg="white", justify='l
eft', textvariable=name).place(x=15, y=185)

    Cemaillbl = Label(feed, font=('vardana', 15),
text="Contact No:", fg="black", bd=10,
anchor=W).place(x=280, y=150)

    Cemaitxt=Entry(feed, font=
('vardana', 15), bd=6, bg="white", justify='left', textvaria
ble=email).place(x=285, y=185)

    Cratelbl = Label(feed, font=('vardana', 15),
text="How was your coffee?", fg="black", bd=10,
anchor=W).place(x=10, y=220)

    checkvar1 = StringVar()

    checkvar2 = StringVar()

    checkvar3 = StringVar()

    checkvar4 = StringVar()

    c1 = Checkbutton(feed, font=('Calibri', 15,
"bold"), text="Awesome", variable=checkvar1)

    c1.deselect()

    c1.place(x=15, y=265)

    c2 = Checkbutton(feed, font=('Calibri', 15,
"bold"), text="Good", variable=checkvar2, )

    c2.deselect()

    c2.place(x=150, y=265)

    c3 = Checkbutton(feed, font=('Calibri', 15,
"bold"), text="Average", variable=checkvar3, )
```

```

c3.deselect()
c3.place(x=250, y=265)

c4 = Checkbutton(feed, font=('Calibri', 15,
"bold"), text="Needs Improvement", variable=checkvar4,
)

c4.deselect()
c4.place(x=370, y=265)

# Add special comments"
commentslbl = Label(feed, font=('Calibri', 15),
text="Special Comments", fg="black", bd=10,
anchor=W).place(x=10, y=300)

txt = Text(feed, width=50, height=5)
txt.place(x=15, y=335)

# button

submit = Button(feed, font=("Calibri", 15),
text="Submit", fg="black", bd=2, command=submit).place(
x=145, y=430)

cancel = Button(feed, font=("Calibri", 15),
text="Cancel", fg="black", bd=2, command=cancel).place(
x=245, y=430)

feed.mainloop()

# Feedbackbutton

feedbtn = Button(labelframe, text="Feedback
Form", fg="black", bg="lightgrey", command=feedbackk,
bd=10, width=15).grid(row=23, column=1)

# Menu card

def menu():

```

```
roott = Tk()
roott.title("Cafe Price Menu")
roott.geometry("300x300")
lblinfo = Label(roott, font=("Calibri", 20, "bold"), text="Favourites", fg="red", bd=10)
lblinfo.grid(row=0, column=0)
lblcprice = Label(roott, font=("Calibri", 20, "bold"), text="Prices", fg="blue", bd=10)
lblcprice.grid(row=0, column=3)
lblfc = Label(roott, font=("Calibri", 20, "bold"), text="Filter Coffee", fg="black", bd=10)
lblfc.grid(row=1, column=0)
lblfcp = Label(roott, font=("Calibri", 20, "bold"), text="40/-", fg="black", bd=10)
lblfcp.grid(row=1, column=3)
lblcl = Label(roott, font=("Calibri", 20, "bold"), text="Cafe Latte", fg="black", bd=10)
lblcl.grid(row=3, column=0)
lblclp = Label(roott, font=("Calibri", 20, "bold"), text="110/-", fg="black", bd=10)
lblclp.grid(row=3, column=3)
lblcm = Label(roott, font=("Calibri", 20, "bold"), text="Cafe Mocha", fg="black", bd=10)
lblcm.grid(row=4, column=0)
lblcmp = Label(roott, font=("Calibri", 20, "bold"), text="150/-", fg="black", bd=10)
lblcmp.grid(row=4, column=3)
```

```
lblcc = Label(roott, font=("Calibri", 20,  
"bold"), text="Cold Coffee", fg="black", bd=10)  
lblcc.grid(row=5, column=0)  
lblccp = Label(roott, font=("Calibri", 20,  
"bold"), text="180/-", fg="black", bd=10)  
lblccp.grid(row=5, column=3)  
roott.mainloop()  
  
# menubutton  
  
menubtn = Button(labelframe, text="Coffee Menu",  
bg="lightgrey", command=menu, width=15,  
bd=10).grid(row=24, column=1)  
  
root.mainloop()  
system()
```

Output:

The output can be seen in *Figure 8.24*:

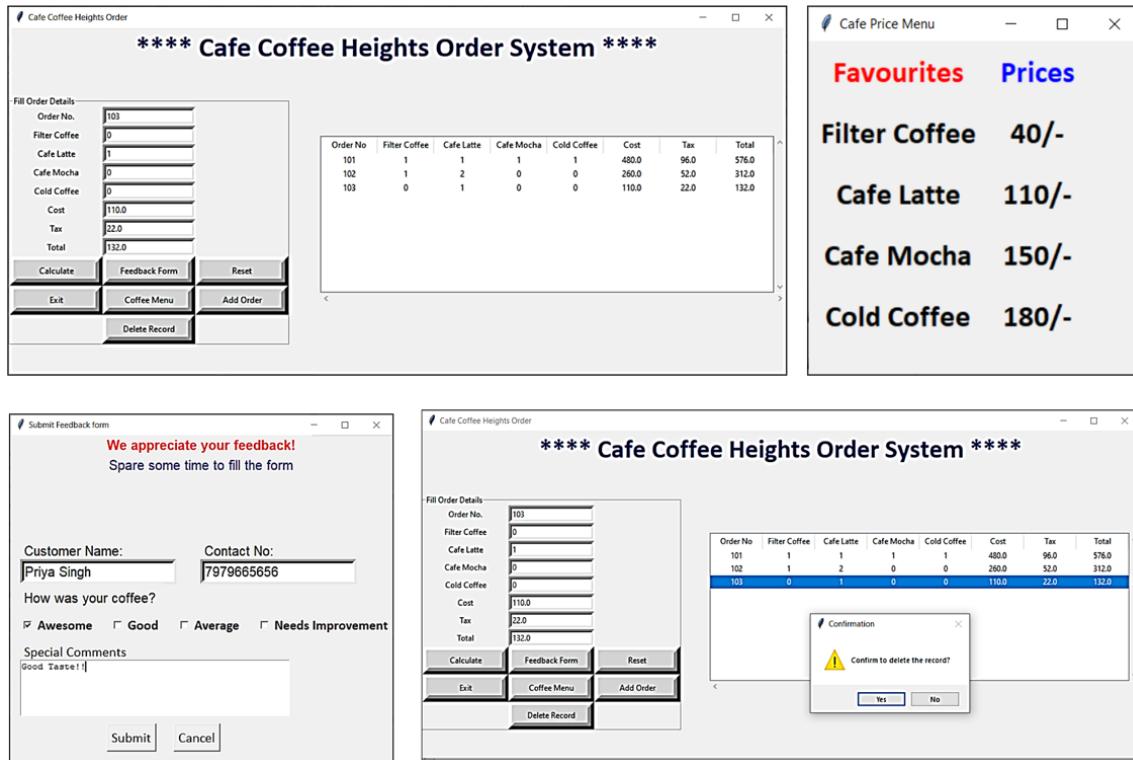


Figure 8.24: Output

Practice project

Create a GUI Text Editor Application that can create, open, edit, and save text files. Use different widgets of the Tkinter module, such as Button, Label, Menu, File Dialog, and so on, along with necessary event-driven programming to create an interactive and user-friendly text editor.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 9

Game Development with PyGame

Introduction

In previous chapters, we have learned to develop a variety of graphic-based and command-based applications in Python. This chapter introduces users to the development of a completely new and interactive set of GUI applications known as Games using the **pygame** library in Python. Games are very popular among users as these serve as favorite entertainment sources that can be extended to any level of imagination and competition.

Structure

In this chapter, we will discuss the following topics:

- Introduction to PyGame and installation
- Color objects, shapes, and surfaces in PyGame
- Adding images, text, and sound in applications
- Sprites and collision detection

Objectives

The purpose of this chapter is to equip users with the knowledge of the creation of interactive gaming applications using the **pygame** library in Python. Here, we will discuss the installation and usage of the **pygame** library along with its various sub-modules, in-built classes, and set of methods useful for creating animations and games in Python.

Introduction to PyGame

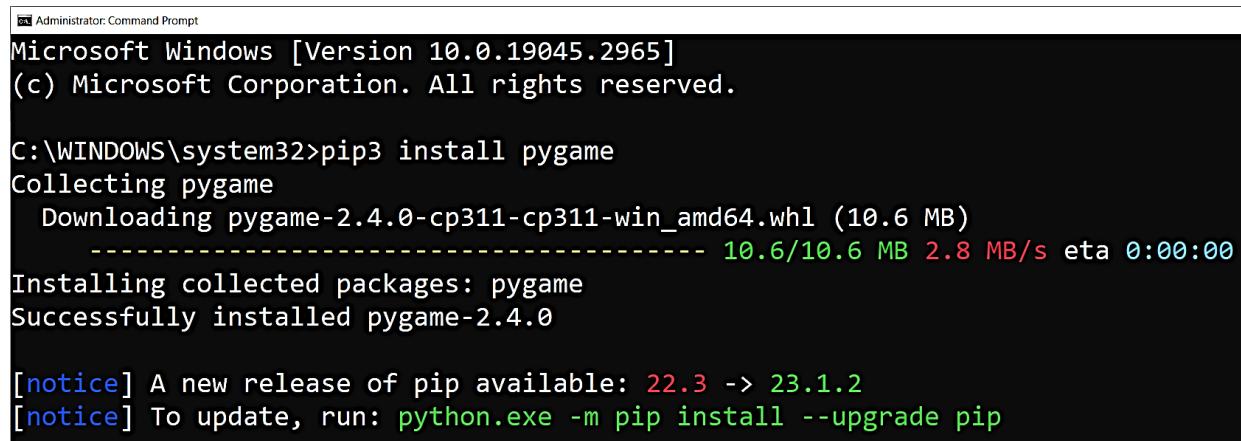
A popular Python library for creating gaming applications is called **Pygame**. It is a **Simple DirectMedia Library (SDL)** wrapper that is open-source, free, and cross-platform. The Pygame abstraction of SDL routines makes it relatively simple to create multi-media apps in Python. It was first created in October 2000 by *Peter Shinners, Lenard Lindstrom, René Dudfield*, and others. In addition to SDL capabilities, Pygame offers additional features such as camera and MIDI support, collision detection, vector math, and others. Applications for Pygame can be run on mobile devices with Android operating systems.

Installing PyGame

Python version 3.6.1 or later must be installed in our system before we start dealing with PyGame since these versions are faster and considerably more user-friendly for beginners. The pip tool, which Python uses to install packages, is the best way to install PyGame. The syntax is as follows:

```
pip3 install pygame
```

The installation of **pygame** in the command prompt is shown in *Figure 9.1*:



```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.19045.2965]
(c) Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>pip3 install pygame
Collecting pygame
  Downloading pygame-2.4.0-cp311-cp311-win_amd64.whl (10.6 MB)
    10.6/10.6 MB 2.8 MB/s eta 0:00:00
Installing collected packages: pygame
Successfully installed pygame-2.4.0

[notice] A new release of pip available: 22.3 -> 23.1.2
[notice] To update, run: python.exe -m pip install --upgrade pip
```

Figure 9.1: Installing PyGame in Windows

Getting started with PyGame

For using pygame in Python applications, it is required to import the library first in order to gain access to the pygame framework and its functions. After this, the `init()` method needs to be called to initialize the required modules of the library. The syntax is as follows:

```
import pygame  
pygame.init()
```

The **pygame** library is made up of various separate modules and several Python constructs. These modules offer uniform methods to interact with the hardware on your system as well as abstract access to that hardware. A list of several such modules present in **pygame** is given in the following table:

Module	Description
<code>pygame.camera</code>	Module for camera use in applications.
<code>pygame.cursors</code>	Module for cursor resources.
<code>pygame.display</code>	Module for controlling the display window and screen.
<code>pygame.draw</code>	Module for drawing different shapes.
<code>pygame.event</code>	Module for interacting with events and queues.
<code>pygame.examples</code>	Module for default example programs.
<code>pygame.font</code>	Module for loading and rendering fonts in pygame.
<code>pygame.gfxdraw</code>	Module for drawing advanced shapes.
<code>pygame.image</code>	Module for image transfer.
<code>pygame.joystick</code>	Module for interacting with joysticks, gamepads, and trackballs.
<code>pygame.key</code>	Module to work with the keyboard.
<code>pygame.mixer</code>	Module for loading and playing audio/sounds.
<code>pygame.mixer.music</code>	Module for controlling streamed audio in applications.
<code>pygame.mouse</code>	Module for working with the mouse and control actions.

<code>pygame.scrap</code>	Module for clipboard support.
<code>pygame.sprite</code>	Module for basic game object classes.
<code>pygame.time</code>	Module for monitoring time.
<code>pygame.transform</code>	Module for transforming surfaces.
<code>pygame.sdl2.touch</code>	Module for working with touch input.

Table 9.1: Modules in pygame

Let us consider a simple demonstration to understand the working of the **pygame** library.

```
import pygame
import sys

# Initialize pygame modules and functions
pygame.init()

screen = pygame.display.set_mode((300, 300))
pygame.display.set_caption("PyGame Demo")

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
```

Output:

Refer to [*Figure 9.2*](#):

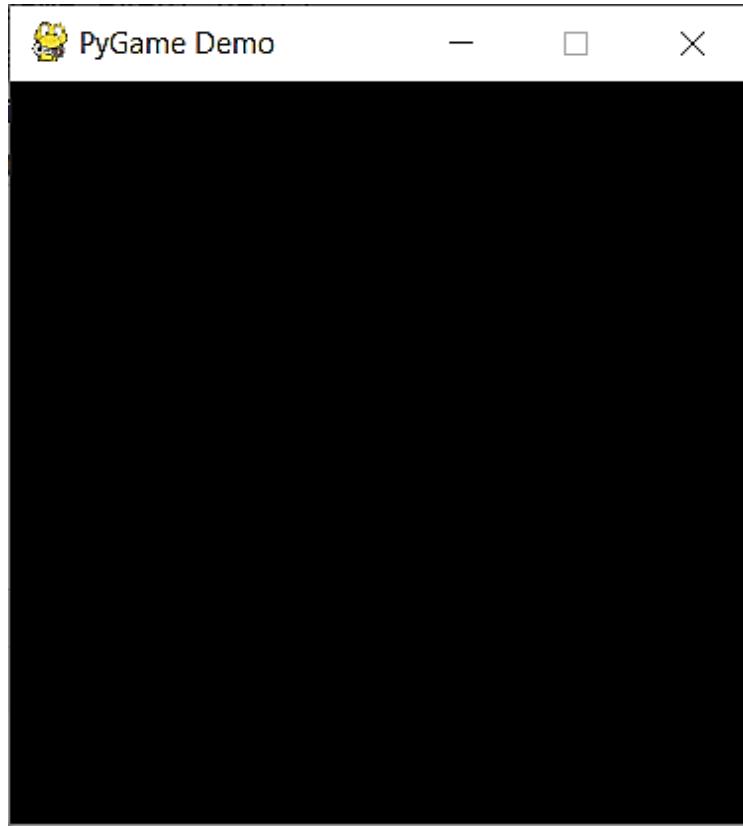


Figure 9.2: Output

In the preceding example, we notice that the first step is to import and initialize pygame modules with the help of the **init()** function. In line 5, a display surface is created by the **set_mode()** function defined in **pygame.display** module. The syntax for the function is as follows:

```
pygame.display.set_mode(size, flags, depth, display)
```

In the preceding syntax, the parameters are explained as follows:

- **size** parameter is a tuple of width and height in pixels.
- **flags** parameter controls the type of display. The following options can be used:

<code>pygame.FULLSCREEN</code>	To create a full-screen display
<code>pygame.RESIZABLE</code>	To display a window that should be re-sizeable
<code>pygame.NOFRAME</code>	To display a window with no border or controls

<code>pygame.SCALED</code>	To set resolution based on desktop size and scale graphics
<code>pygame.SHOWN</code>	Default mode. To open the window in visible mode
<code>pygame.HIDDEN</code>	To open a window in hidden mode

Table 9.2: Options for flag argument in `set_mode` method

- **display** index 0 means the default display is used.
- **depth** parameter enables setting the best and fastest color depth available for the system.

In line 6, we set up a Pygame display window of the preferred size and give it the caption **PyGame Demo**. This will render a game window that needs to be put in an infinite event loop starting from line 7. By using `pygame.event.get()` in line 8, all the event objects produced by user interactions, such as mouse drag, click, and so on, are stored in an event queue to handle them in order. Line 9 specifies that the event loop terminates when `pygame.QUIT` event is raised when the user clicks the CLOSE button on the title bar in the window generated.

Color object in pygame

Pygame uses the `Color` class to represent the color of the screen background, text, shapes, and any other objects. Red, Green, and Blue color values, and an optional alpha value, which denotes the degree of opacity, are provided to create it. These values range between 0 and 255. The syntax used for creating a color object in pygame is as follows:

```
colorObject = pygame.Color(r, g, b, a=255)
```

Here, the options used in creating a `Color` object are defined as follows:

<code>pygame.Color.r</code>	To set the red color value to the <code>Color</code> object.
<code>pygame.Color.g</code>	To set the green color value to the <code>Color</code> object.
<code>pygame.Color.b</code>	To set the blue color value to the <code>Color</code> object.
<code>pygame.Color.a</code>	To define the alpha value for the opacity of the <code>Color</code> object. The default value is 255, that is, the fully opaque

| object created.

Table 9.3: Options available for Color object

In addition to this, we may also use predefined string constants representing color names. The list of a few frequently used colors is given in [Figure 9.3](#):

'BLACK' = (0, 0, 0, 255),	'ORANGE' = (255, 165, 0, 255),
'BLUE' = (0, 0, 255, 255),	'PURPLE' = (160, 32, 240, 255),
'CYAN' = (0, 255, 255, 255),	'RED' = (255, 0, 0, 255),
'GOLD' = (255, 215, 0, 255),	'VIOLET' = (238, 130, 238, 255)
'GRAY' = (190, 190, 190, 255),	'YELLOW' = (255, 255, 0, 255),
'GREEN' = (0, 255, 0, 255),	'WHITE' = (255, 255, 255, 255)

Figure 9.3: String constants with color names

Surface and shapes in pygame

When using Pygame, surfaces are generally used to represent the appearance of the object and its position on the screen. All the objects, text, and images that we create in Pygame are created using surfaces. Creating surfaces in a **pygame** is quite easy. We just have to pass the height and the width with a tuple to the **pygame.Surface()** method. We can use various methods to format our surface as we want. For example, we can use **pygame.draw()** to draw shapes, and we can use the **surface.fill()** method to fill on the surface.

The syntax to create a **pygame** surface object is given as follows:

```
pygame.surface()
```

It requires two mandatory arguments, namely, **width** and **height**, and optional arguments, such as **flags**, **color** **depth**, and so on. An example code to create a new surface is as follows:

```
surface_size = width, height = (50, 50)  
new_surface = pygame.Surface(surface_size)
```

Different shapes such as rectangles, circles, ellipses, polygons, and lines can be drawn on the game window surface by using various functions in the

`pygame.draw` module as given as follows:

Shape	Syntax
Rectangle	<pre>pygame.draw.rect(Surface, color, rect, width=0)</pre> <p>Where:</p> <ul style="list-style-type: none">• <code>surface</code>: the surface is the screen where the shape is drawn.• <code>color</code>: It denotes the color value (and alpha) of the shape in the range 0–255.• <code>rect</code>: It represents a rectangular type area where the shape is drawn.• <code>width</code>: It specifies the width of the edges. If set to 0, then the shape will be filled. The default width is 1. For value > 1, the line thickness increases.
Polygon	<pre>pygame.draw.polygon(Surface, color, points, width=0)</pre> <p>Where:</p> <ul style="list-style-type: none">• <code>points</code>: The list of pixel coordinates (x, y) of polygon vertices.• Parameters like <code>surface</code>, <code>color</code>, and <code>width</code> have the usual meaning.
Circle	<pre>pygame.draw.circle(Surface, color, center, radius, width=0)</pre> <p>Where:</p> <ul style="list-style-type: none">• <code>center</code>: The pixel coordinates (x, y) of the center of the circle.• <code>radius</code>: The pixel value of the radius of the circle.• Parameters like <code>surface</code>, <code>color</code>, and <code>width</code> have the usual meaning.
Ellipse	<pre>pygame.draw.ellipse(Surface, color, rect, width=0)</pre> <p>Where</p> <ul style="list-style-type: none">• Parameters have the usual meaning.

Shape	Syntax
Elliptical arc	<pre>pygame.draw.arc(Surface, color, rect, start_angle, stop_angle, width=1)</pre> <p>Where</p> <ul style="list-style-type: none"> • start_angle: The beginning angle of the elliptical arc with value in radians. • stop_angle: The end angle of the elliptical arc with value in radians. • Parameters, such as surface, color, width, and so on, have the usual meaning.
Straight line	<pre>pygame.draw.line(Surface, color, start_pos, end_pos, width=1)</pre> <p>Where:</p> <ul style="list-style-type: none"> • start_pos: The starting pixel coordinate of the line. • end_pos: The end pixel coordinate of the line.

Table 9.4: Different shapes in `pygame.draw` module

While the **draw()** method is used to draw an object, the **fill()** method is used to fill the color on the surface. The syntax of **fill()** method is as follows:

```
pygame.Surface.fill(color, rect=None)
```

Where the **fill()** method fills the given Surface with a color specified by the **color** argument. Suppose the value of the **rect** argument is not defined; then the entire Surface will be filled with color. Now, once we create some shapes and fill colors on the surface window, the next task is to update the contents of the entire display area. This is accomplished using the **flip()** method of **pygame.display** module. It will refresh and reload the contents of the display. The syntax of **flip()** method is as follows:

```
pygame.display.flip()
```

Let us consider an example to demonstrate different shapes in a Pygame:

```
import pygame as pg
pg.init()
```

```
screenSurface = pg.display.set_mode((700, 700))

done = False

GRAY = (190, 190, 190, 255)
screenSurface.fill(GRAY)

RED = (255, 0, 0)

VIOLET = (238, 130, 238, 255)

GOLD = (255, 215, 0, 255)

CYAN = (0, 255, 255, 255)

ORANGE = (255, 165, 0, 255)

PURPLE = (160, 32, 240, 255)

while not done:

    for event in pg.event.get():

        if event.type == pg.QUIT:

            done = True

        pg.draw.rect(surface=screenSurface,
color=VIOLET, rect=(200, 130, 160, 160))

        pg.draw.polygon(surface=screenSurface,
color=CYAN, points=((20,380),(80,435),(135,300),
(100,325),(220, 380)))

        pg.draw.circle(surface=screenSurface,
color=PURPLE, center=(100,100), radius=80)

        pg.draw.line(surface=screenSurface, color=RED,
start_pos=(400, 600), end_pos=(550, 500), width=4)

        pg.draw.ellipse(surface=screenSurface,
color=GOLD, rect=(400, 200, 300, 70))
```

```
pg.draw.rect(surface=screenSurface,  
color=ORANGE, rect=(300, 20, 270, 50), width=5)  
  
pg.draw.circle(surface=screenSurface,  
color=CYAN, center=(380, 380), radius=60, width=2)  
  
pg.draw.ellipse(width=5, surface=screenSurface,  
color=PURPLE, rect=(250, 400, 130, 80))  
  
pg.display.flip()
```

Output:

Refer to *Figure 9.4*:

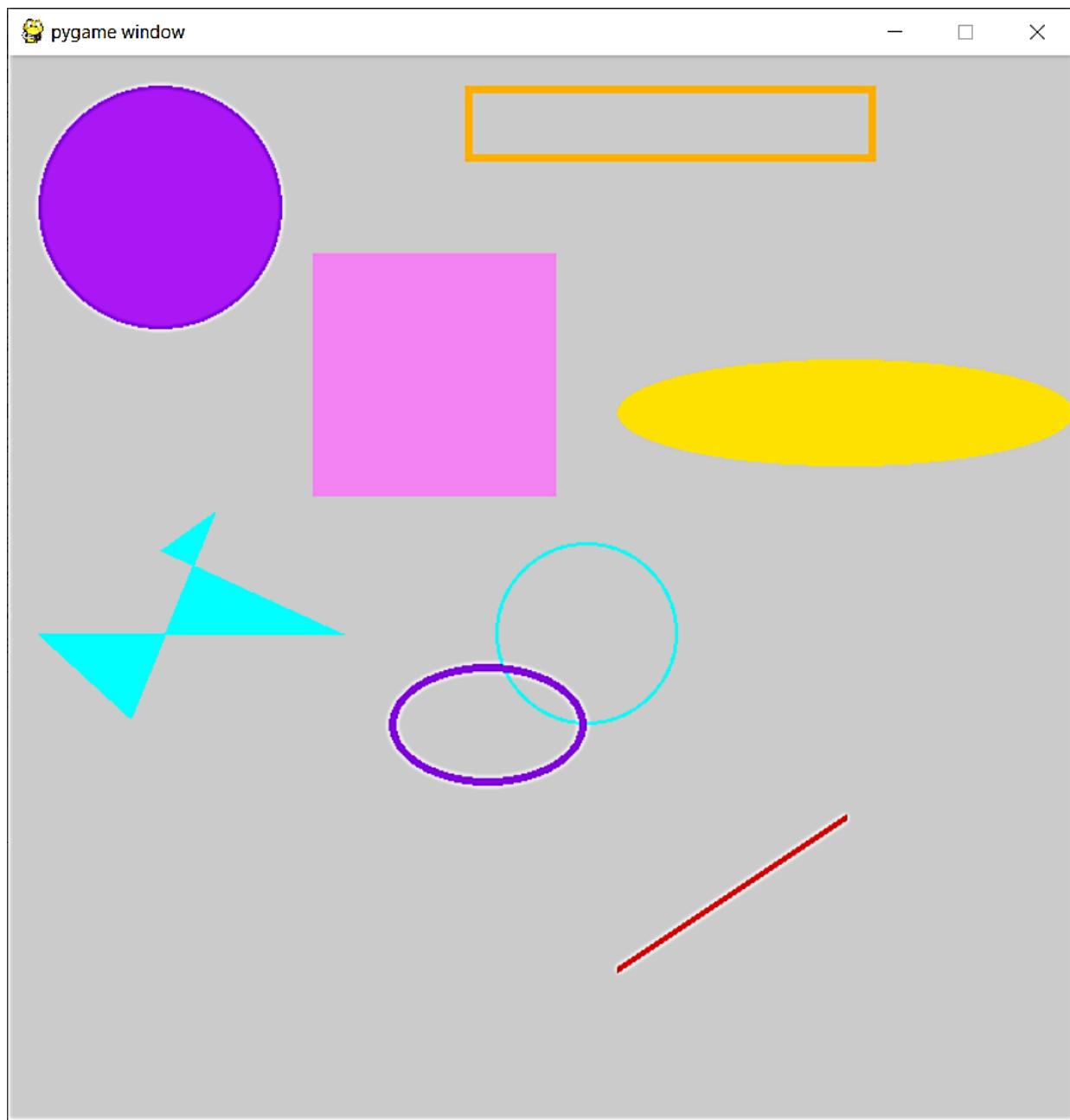


Figure 9.4: Output

Images in pygame

Images are an important component of any application. To use images, the **pygame** library supports **pygame.image** module that consists of various functions used for loading and saving images to and from the application. An image is loaded in the form of a Surface object and ultimately rendered for display on the screen. The set of functions used to load, convert, and

render images in **pygame.image** module is discussed in the following subsections:

Loading, Converting, and Blitting Images

The first step to use images in the **pygame** application is to load the image from its file/folder location. For this purpose, the **pygame.image.load()** function is used. The syntax is as follows:

```
imageObject = pygame.image.load(path_of_image)
```

The parameter used in the **load()** function is the address location of the image, which may be a relative or absolute path as per the users' choice. However, to increase the speed and performance factor of the overall application, it is recommended to optimize and convert the image to the same pixel format as the display screen. It is accomplished by invoking the Surface method **convert()** while loading the image. The syntax used is as follows:

```
imageObject =  
pygame.image.load(path_of_image).convert()
```

In case the image uses alpha values for maintaining transparency, then instead of calling the simple **convert()** method we can use **convert_alpha()** method. The syntax is as follows:

```
imageObject =  
pygame.image.load(path_of_image).convert_alpha()
```

A special process called Blitting used in pygame is defined as the process of rendering a game object by copying the object pixel by pixel from the source surface onto the display surface (which may be the screen as well). It is of utmost importance to render any game object created and used in the application. If the Blitting process is skipped while executing the application, then the resulting output will be a black window displayed. Thus, to enable the blitting process, the **blit()** method is used. The syntax used is as follows:

```
SurfaceObject.blit(source, dest, area=None, special_flags=0)
```

The parameters from the preceding syntax are explained as follows:

- The **source** parameter draws from the source Surface onto the display Surface
- The **dest** parameter is used as the destination coordinates where the draw can be positioned.
- In the **area** argument, the user may pass rect area coordinates can also be passed where the top-left vertex position of the rectangle denotes the position for the blit.
- The **special_flags** parameter may take the following values: **BLEND_RGBA_ADD**, **BLEND_RGBA_SUB**, **BLEND_RGBA_MULT**, **BLEND_RGBA_MIN**, **BLEND_RGBA_MAX** **BLEND_RGB_ADD**, **BLEND_RGB_SUB**, **BLEND_RGB_MULT**, **BLEND_RGB_MIN**, and **BLEND_RGB_MAX**.

Please note that it is possible to render and blit objects to Surfaces other than the display screen. Finally, to display blitted objects on screen, user may invoke either **pygame.display.update()** or **pygame.display.flip()** functions. Let us see a demonstration of using images in Pygame:

```
import pygame as pg
pg.init()
screenSurface = pg.display.set_mode((400, 400))
bird_img =
pg.image.load('D:/Images/game_image.png').convert_alpha()
flag = False
bgColor = (255, 255, 255)
while not flag:
    for ev in pg.event.get():
        screenSurface.fill(bgColor)
```

```
rect = bird_img.get_rect()
rect.center = 100, 100
screenSurface.blit(bird_img, rect)
if ev.type == pg.QUIT:
    flag = True
pg.display.update()
```

Output:

Refer to *Figure 9.5*:

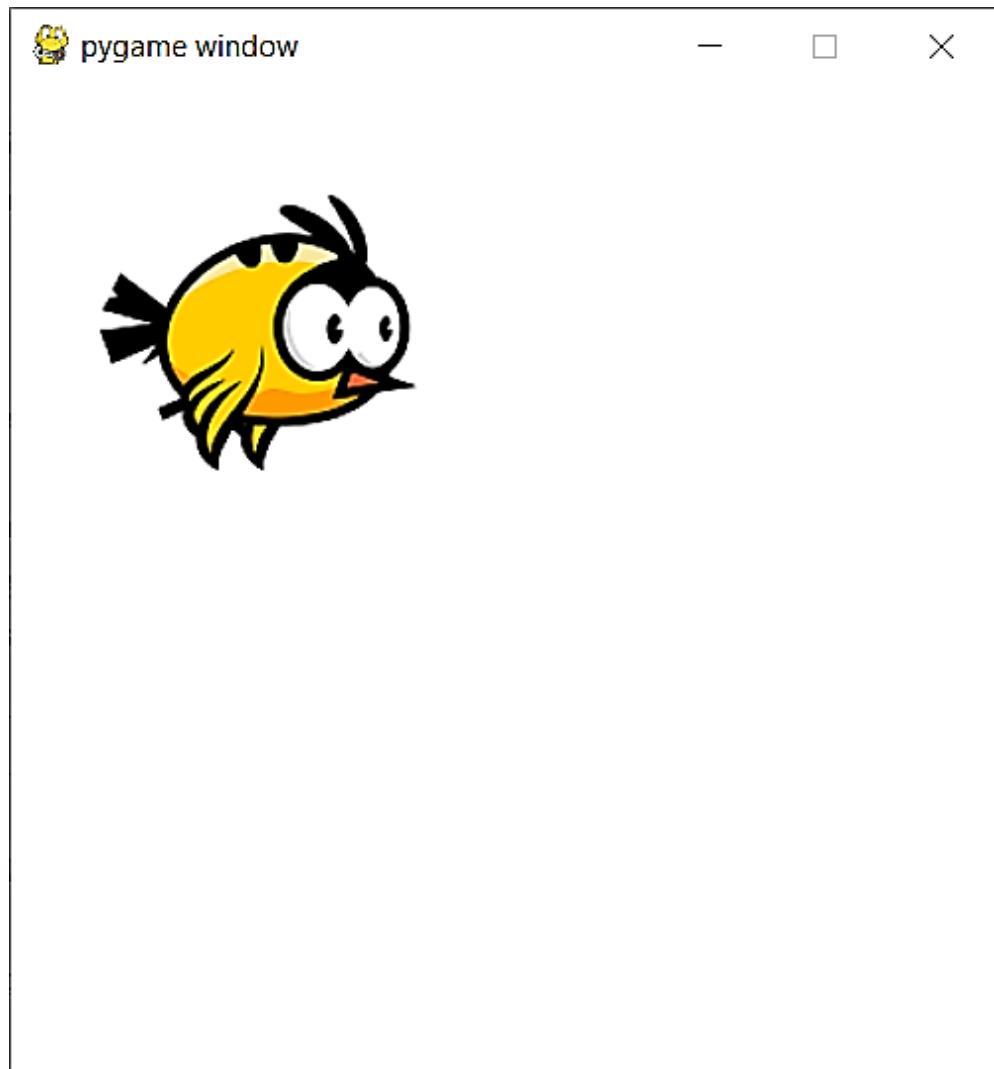


Figure 9.5: Output

Events in pygame

An event is a user-performed action performed to achieve the desired outcome. A click event, for instance, occurs when a user clicks a button to perform some action. All user-performed events are added to a queue data structure in FIFO order, often known as an **Event Queue**. The **First-In-First-Out (FIFO)** rule states that the element inserted first shall be removed first from the queue. In this scenario, a new event, when raised, is added to the rear end of the queue; it is processed and removed from the *front end* of the queue. The `pygame.event` module supports user input for the event handling process. Once the application starts, the program waits for user input to initiate any further action. Here, the role pygame event loop comes into play, where its primary task is to look for various user inputs via keyboard, mouse or any other device and take necessary actions. The following event functions can be used to manage events in the event queue:

- `pygame.event.get()`: Used to retrieve all events or event types.
- `pygame.event.poll()`: It enables to fetch a single event at a time from the queue.
- `pygame.event.wait()`: It is used to wait for a single event to occur from the queue.
- `pygame.event.clear()`: This method clears all events in the queue.
- `pygame.event.pump()`: If the user does not want to invoke any events, then `pygame.event.pump()` must be called in the game loop to process the events implicitly.

Key events

In `pygame`, there are two different kinds of key events: **KEYDOWN** and **KEYUP**. These events have a key attribute, a number that corresponds to a keyboard key. All of the common keys are represented by preset integer constants in the `pygame` module:

Device	Event	Event attributes

Keyboard	KEYDOWN	<ul style="list-style-type: none"> key: The name of the key, an underscore, and a capital K are used to identify the constants. For example, for the backspace key, use K_BACKSPACE; for key “b” use K_b and F5 is used as K_F5.
	KEYUP	<ul style="list-style-type: none"> mod: represents the modifier keys, such as ctrl, shift, alt, caps, and so on, pressed along with the key. Each modifier key is represented by string KMOD_ followed by its name such as Tab key is termed KMOD_TAB, Ctrl key as KMOD_CTRL, and Left shift as KMOD_LSHIFT, for Caps use KMOD_CAPS and so on.

Table 9.5: Key Events in pygame

Mouse events

Three different mouse events are supported in **pygame**: **MOUSEMOTION**, **MOUSEBUTTONDOWN**, and **MOUSEBUTTONUP**. Once the display mode is set, these events will be registered by **pygame**.

Device	Event	Functionality	Attributes
Mouse	MOUSEBUTTONDOWN	Event raised when the user presses a mouse button.	<ul style="list-style-type: none"> button: A tuple represents left, right or mouse-wheel buttons. pos: The pixel form coordinate location (x, y) of the cursor. rel: The pixel location relative to previous location (rel_x, rel_y).
	MOUSEBUTTONUP	Event is raised when the user releases the mouse button.	
	MOUSEMOTION	Event is raised when the user moves the cursor on the screen.	

Table 9.6: Mouse events in pygame

Let us demonstrate the usage of event handling by moving a bird image to a different direction using arrow keys from the keyboard, thereby giving a user-controlled animated effect of a flying bird:

```
import pygame as pg
from pygame.locals import *
from sys import exit
pg.init()
screenSurface = pg.display.set_mode((400, 400))
pg.display.set_caption("Flying Bird with arrow
keys")
bird_img =
pg.image.load('D:/Images/game_image.png').convert_a
lpha()
flag = False
xloc = 100
yloc= 100
screenSurface.blit(bird_img, (xloc, yloc))
while True:
    screenSurface.fill((255,255,255))
    screenSurface.blit(bird_img, (xloc, yloc))
    for ev in pg.event.get():
        if ev.type == QUIT:
            exit()
        if ev.type == KEYDOWN:
            if ev.key == K_RIGHT:
                xloc= xloc+10
            if ev.key == K_LEFT:
```

```

xloc=xloc-10

if ev.key == K_UP:
    yloc=yloc-10

if ev.key == K_DOWN:
    yloc=yloc+10

pg.display.update()

```

Output:

Refer to *Figure 9.6*:

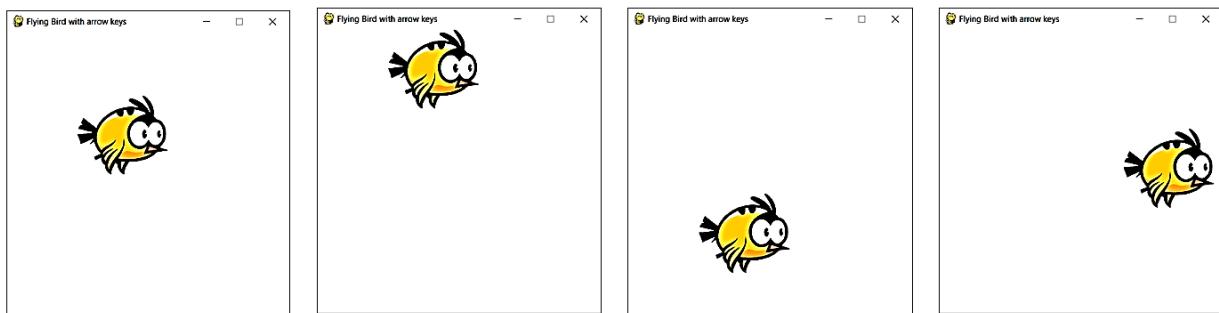


Figure 9.6: Output

Note: Moving the bird up with the UP arrow key, down with the DOWN key, and forward and backward with the RIGHT and LEFT arrow keys.

Adding text and music in pygame

We must first retrieve a font object with the aid of the **SysFont()** function provided by the **pygame.font** module in order to show text on the Pygame window. The syntax is as follows:

```
fontObject= pygame.font.SysFont(FontName, FontSize, bold, italic)
```

In the preceding syntax, the bold and italic parameters can be set to **True** or **False** as per user choice. To retrieve a list of all font names supported by

the current machine, we can use the function `get_fonts()`. The following example code retrieves and prints the list of all installed fonts.

```
fontlist = pygame.font.get_fonts()  
for l in fontlist:  
    print(l)
```

The next step is to create an image Surface of the text and then blit this image onto another surface. Consider the following syntax:

```
textImageObject = fontObject.render(Text_String,  
antialias, Color, Background)
```

Note that if the `antialias` argument is `False`, then the rendered image is of 8-bit size; otherwise, 24-bit size is set to `True`. Finally, we blit the text image (surface) to the center of the window screen:

```
screen.blit(textImageObject, (X_loc, Y_loc))
```

Similarly, to enhance our application with music and sounds Use the `mixer.music` module from the `pygame` library to add background music. `WAV`, `MP3`, or `OGG` files can be loaded using `Pygame` with the help of following syntax.

```
pygame.mixer.music.load(filename or object)
```

A music filename or file object will be loaded and ready for playback. The following actions are used to control playback:

```
play(loops=0, start=0.0, fade_ms = 0)
```

The stream of music that is loaded will be played. It will be restarted if the song is already playing. How often to repeat the music is indicated by the `loops` argument. If this option is set to `-1`, the music keeps playing forever. Here, the start designates where the song begins to play with position expressed in seconds of time. The song starts playing at 0 volume and gradually increases in loudness over the specified amount of time using the `fade_ms` argument. Some other functions used are listed in the following table:

Function	Description
<code>init()</code>	To initiate and start the mixer object for play.
<code>rewind()</code>	Playback of the current song is restarted at the beginning.
<code>Stop()</code>	It shuts off any music that is currently playing without unloading it.
<code>Pause()</code>	Put a temporary stop to the music stream's playback.
<code>Unpause()</code>	This will restart the paused music stream when it has been interrupted.
<code>Fadeout(time)</code>	Stop the music that is playing by fading out.
<code>Set_volume(volume)</code>	Decide how loud you want the song to play.
<code>Set_pos(pos)</code>	This determines where in the music file playback will begin.

Table 9.7: Functions for handling background music in pygame

Let us consider an example to add text and sound to our **pygame** application.

```
import pygame
from pygame.locals import *
from pygame import mixer
pygame.init()
wt = 900
ht = 500
window = pygame.display.set_mode((wt, ht))
# To display an image
flying_bird_img =
pygame.image.load('D:/images/bird1.png')
flying_bird_img =
pygame.transform.scale(flying_bird_img, (wt, ht))
```

```
# To display and configure text
newfont = pygame.font.SysFont("Arial", 50)
newtext = newfont.render("Welcome to Pygame!!",
True, (255,0,0))

# To add music to pygame application
mixer.init()
mixer.music.load('D:/images/HappyMusic.mp3')
mixer.music.play()

run_flag = True
while run_flag:
    window.blit(flying_bird_img, (0, 0))
    window.blit(newtext, (20, 20))
    for event in pygame.event.get():
        if event.type == QUIT:
            run_flag = False
    pygame.display.update()
pygame.quit()
```

Output:

Refer to *Figure 9.7*:



Figure 9.7: Output

Note: Users may download freely available music files and images to execute the preceding example.

Sprites and collisions

A two-dimensional image or portion of an image that can move is known as a **pygame sprite**. A sprite typically portrays an object in the scene. Working with sprites in groups is one of the biggest benefits. If the sprites are in a group, we can quickly move and draw them all with a single command. Classes and functionality beneficial for game development are found in the **pygame.sprite** module. In addition to a Sprite class that allows the creation of collections of sprite objects, there are functions that allow sprite object collision. When two things on the screen clash, the process is known as collision, which can be detected using the collision detection strategy. The Sprite class acts as a basis class for various game components. A sprite group that can manage several sprites at once and has several methods for quickly updating and generating sprites. Various functions are present in

`pygame.sprite.Sprite` class used to manipulate sprite objects are given as follows:

Function	Description
<code>__init__()</code>	To initialize the sprite object.
<code>update()</code>	Used to add required functionality to control the behavior of sprite object.
<code>draw()</code>	It takes the surface object as an argument and draws all the sprites in a specific group.
<code>add()</code>	Used to add the sprite to a specified group.
<code>remove()</code>	Used to delete sprite from the group.
<code>kill()</code>	To delete all sprites from the group, use <code>kill()</code> .
<code>alive()</code>	To check if the sprite object belongs to a specific group.
<code>groups()</code>	To display the list of groups containing specific sprite objects.

Table 9.8: Functions for managing sprites in pygame

The `self.image` and `self.rect` variables are defined in the `__init__` method to initialize the sprite. To produce movement for our Ball sprite, we will now define an update method. The speed variable in the form of a list of length 2, needs to be initialized inside the class:

```
import pygame
from sys import exit
screen_width = 1480
screen_height = 820
pygame.init()
clock = pygame.time.Clock()
class FootBall(pygame.sprite.Sprite):
    def __init__(self, pos_x, pos_y) -> None:
```

```
super().__init__()

self.pos_x = pos_x
self.pos_y = pos_y
self.speed = [5, 5]

self.image =
pygame.image.load('D:\images\FootBall.png')

self.rect = self.image.get_rect(center=
(self.pos_x, self.pos_y))

def update(self):

    self.detect_collision()

    # To update horizontal position of football
    self.rect.x += self.speed[0]

    # To update vertical position of the
    football

    self.rect.y += self.speed[1]

def detect_collision(self):

    # To detect collision with screen walls and
    change direction

    if self.rect.right >= screen_width:
        self.speed[0] *= -1
    elif self.rect.left <= 0:
        self.speed[0] *= -1
    if self.rect.top <= 0:
        self.speed[1] *= -1
```

```
        elif self.rect.bottom >= screen_height:
            self.speed[1] *= -1

# To Setup display screen

game_window =
pygame.display.set_mode((screen_width,
screen_height))

pygame.display.set_caption('Sprites and Collision')
# To create display caption

football_sprite = FootBall(screen_width / 2,
screen_height / 2)      # To create football class
instance

football_group = pygame.sprite.Group()
# To create a football sprite group

football_group.add(football_sprite)
# To add a football sprite to the sprite group

# Design a game loop

while True:

    game_window.fill((0, 125, 125))

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            pygame.quit()

            exit()

    football_group.update() # Call update() to
update the sprites

    football_group.draw(game_window) # To draw all
sprites in group
```

```
pygame.display.update()  
clock.tick(50)
```

Output:

Refer to *Figure 9.8*:

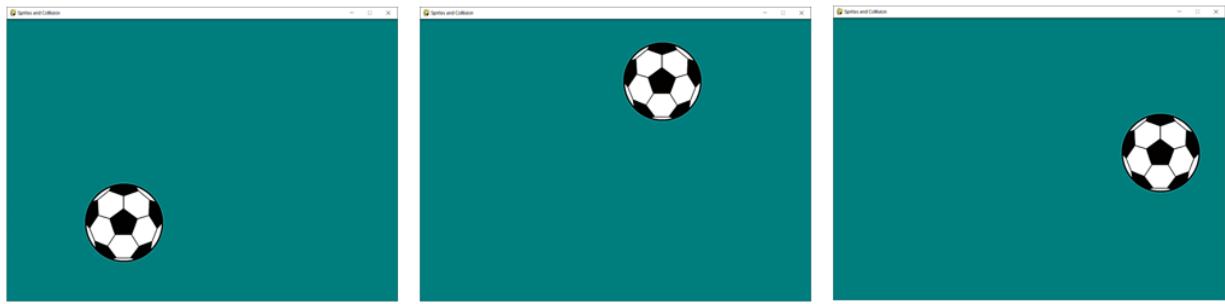


Figure 9.8: Output

The positions of the top-left corner of the football's rectangle are provided by the `self.rect.x` and `self.rect.y` methods. Currently, the football may be seen moving, but in order to keep it contained within the screen, collision detection with the screen's window or walls must be managed. For instance, the rightmost pixel of a ball that strikes the right side of the screen should be at least as wide as the screen. Also, the leftmost pixel of the ball should be less than or equal to zero if it strikes the left side of the screen. We can create a new method called `detect_collision()` to deal with wall collisions and call it from the update function to update the football's motion following a collision. In the `detect_collision()` method, we check if the ball's rightmost pixel is greater than or equal to the screen width. If it is, we multiply the horizontal speed represented as `self.speed[0]` by -1 to change the direction. Similarly, if the ball's leftmost pixel is less than or equal to 0, we again multiply the horizontal speed by -1 . This ensures that the ball bounces off the left and right walls. Additionally, if the ball's topmost pixel is less than or equal to 0, or its bottommost pixel is greater than or equal to the screen height, we multiply the vertical speed represented as `self.speed[1]` by -1 to make the ball bounce off the top and bottom walls.

Conclusion

This chapter gives a complete overview of developing interactive game applications in Python using **pygame** module along with its sub-modules, classes, and numerous functions. Users can extend their knowledge by developing popular games such as snake with prey, Flappy Bird crossing obstacles, and so on, using **pygame** in Python.

Points to remember

- A popular Python library for creating gaming applications is called Pygame. It is a Simple DirectMedia Library (SDL) wrapper that is open-source, free, and cross-platform.
- The pip tool, which Python uses to install packages, is the best way to install PyGame.
- The **pygame** library is made up of various separate modules and several Python constructs. These modules offer uniform methods to interact with the hardware on your system as well as abstract access to that hardware.
- To use images, the **pygame** library supports **pygame.image** module that consists of various functions used for loading and saving images to and from the application.
- The **pygame.event** module supports user input for the event handling process. Once the application starts, the program waits for user input to initiate any further action.
- We must first retrieve a font object with the aid of the **SysFont()** function provided by the **pygame.font** module in order to show text on the Pygame window.
- A two-dimensional image or portion of an image that can move is known as a **pygame** sprite. Classes and functionality beneficial for game development are found in the **pygame.sprite** module.

Exercise

Attempt the following projects.

Sample project with solution

Create a Snake-catching prey game using Pygame and display the total score based on the maximum prey caught. The player's main goal in this game is to catch as much prey as possible without colliding with the wall. Snake body increases in length with the increasing number of preys caught. We make sure that the snake will always travel to the right of the screen whenever the game starts. Once the snake collides with the screen walls or touches itself, the game ends by displaying the final score:

```
# importing libraries
import pygame as pg
import time
import random
snake_moving_speed = 10
# Set Screen size
screen_x = 800
screen_y = 500
# To define color constants
Black = pg.Color(0, 0, 0)
Green = pg.Color(0, 255, 0)
White = pg.Color(255, 255, 255)
Blue = pg.Color(0, 0, 255)
Red = pg.Color(255, 0, 0)
# Initialising pygame using init()
pg.init()
```

```
# To configure screen window
pg.display.set_caption('Snake Catching Prey')
screen_game = pg.display.set_mode((screen_x,
screen_y))

# Controlling FPS (frames per second) to display
frames_ps = pg.time.Clock()

# The initial location of snake
snake_location = [100, 100]

# Represent 3 blocks comprising snake body
body_snake = [[100, 100], [90, 100], [80, 100]]

# To define prey location
prey_location = [random.randrange(1, (screen_x // 10)) * 10, random.randrange(1, (screen_y // 10)) * 10]

prey_spawn = True

# Assign default direction for snake movement
# towards right
default_dir = 'RIGHT'

update_to = default_dir

# Define initial score for user
init_score = 0

# To define update Score function
def update_score(ch, color, font, size):
    # Define font object to display score
```

```
font_obj = pg.font.SysFont(font, size)

# Now create the display surface object

score_surface_obj = font_obj.render('Score : ' +
+ str(init_score), True, color)

# To define a rectangular object for the text
surface object

score_rect_obj = score_surface_obj.get_rect()

# To display user score

screen_game.blit(score_surface_obj,
score_rect_obj)

# Define function to over the game

def game_over():

    # creating font object my_font

    new_font = pg.font.SysFont('Arial', 30)

    # To define a text surface to draw score text

    game_end_surface = new_font.render('Final
Score: ' + str(init_score), True, Green)

    # To create a rectangular object for the text
surface object

    game_end_rect = game_end_surface.get_rect()

    # To define position of the text display

    game_end_rect.midtop = (screen_x / 2, screen_y
/ 4)

    screen_game.blit(game_end_surface,
game_end_rect)
```

```
pg.display.flip()

# To quit the program after 5 seconds
time.sleep(5)

pg.quit()

quit()

# Define event loop
while True:

    # To manage key events
    for event in pg.event.get():

        if event.type == pg.KEYDOWN:

            if event.key == pg.K_UP:

                update_to = 'UP'

            if event.key == pg.K_DOWN:

                update_to = 'DOWN'

            if event.key == pg.K_LEFT:

                update_to = 'LEFT'

            if event.key == pg.K_RIGHT:

                update_to = 'RIGHT'

        # No movement if two keys pressed
        # simultaneously
        if update_to == 'UP' and default_dir != 'DOWN':

            default_dir = 'UP'

        if update_to == 'DOWN' and default_dir != 'UP':
```

```
    default_dir = 'DOWN'

    if update_to == 'LEFT' and default_dir != 'RIGHT':
        default_dir = 'LEFT'

        if update_to == 'RIGHT' and default_dir != 'LEFT':
            default_dir = 'RIGHT'

# Define movement of the snake in directions

if default_dir == 'UP':
    snake_location[1] -= 10

if default_dir == 'DOWN':
    snake_location[1] += 10

if default_dir == 'LEFT':
    snake_location[0] -= 10

if default_dir == 'RIGHT':
    snake_location[0] += 10

# Snake body increasing on catching prey then
scores increased by 10

body_snake.insert(0, list(snake_location))

if snake_location[0] == prey_location[0] and
snake_location[1] == prey_location[1]:
    init_score += 10
    prey_spawn = False

else:
```

```
    body_snake.pop()

    if not prey_spawn:
        prey_location = [random.randrange(1,
(screen_x // 10)) * 10,
                           random.randrange(1,
(screen_y // 10)) * 10]
        prey_spawn = True

    screen_game.fill(white)

    for pos in body_snake:
        pg.draw.rect(screen_game, Red,
pg.Rect(pos[0], pos[1], 10, 10))

        pg.draw.rect(screen_game, Black, pg.Rect(
            prey_location[0], prey_location[1], 10,
10))

    # Conditions to exit Game

    if snake_location[0]<0 or snake_location[0] >
screen_x-10:
        game_over()

    if snake_location[1]<0 or snake_location[1] >
screen_y-10:
        game_over()

    # Looping or Touching body of snake

    for block in body_snake[1:]:
        if snake_location[0] == block[0] and
snake_location[1] == block[1]:
```

```

        game_over()

# Call Update score

update_score(1, Black, 'times new roman', 20)

# Refresh screen window

pg.display.update()

# Update Frame Per Second /Refresh Rate

frames_ps.tick(snake_moving_speed)

```

Output:

The output can be seen in *Figure 9.9*:

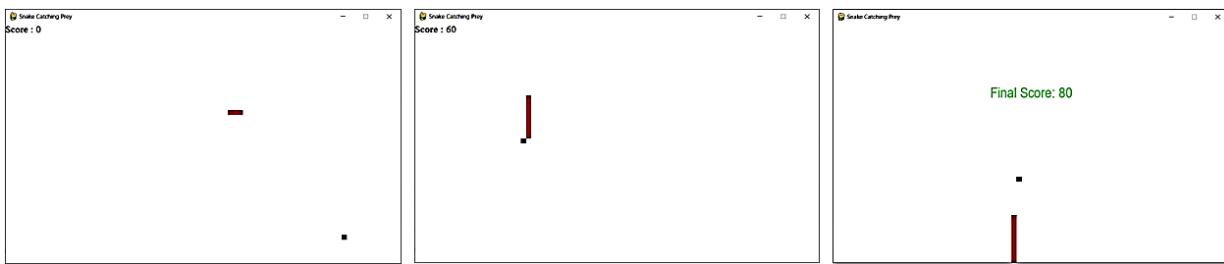


Figure 9.9: Output

Practice project

Create an interactive game of a flying airplane crossing the obstacles to reach its destination. In the game, the user controls the airplane to move it forward, upward, or downward. As an additional feature the airplane can either shoot the obstacles to destroy them or simply cross them without getting collided. If the airplane collides with an obstacle, we lose, else if we successfully shoot or cross all obstacles, we win the game. The user can also control the shooting action through a key press or mouse-click. Set a benchmark as the destination for the airplane and add a timer also to record the time taken. The following key mappings can be used in the game:

- Right Arrow key to move airplane forward.
- Up Arrow key to shift the airplane upwards.

- Down arrow key to shift the airplane downwards.
- Use Space key to shoot obstacles
- Use Esc key to exit the game

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



CHAPTER 10

Mobile App Development with Kivy

Introduction

In previous chapters, we have learned how to develop different applications like desktop applications using the Tkinter library and dynamic interactive gaming applications using the Pygame library. Nowadays, users prefer and feel more convenient to have handy applications in their Android phones rather than bulky laptops that require to be carried separately for use and are time-consuming to handle. Thus, the scope of this chapter revolves around understanding the basics of a simple mobile application development using the Kivy library in Python. By the end of this chapter, users will be able to develop and convert simple Kivy applications into Android APK files that can be used in most Android-based phones.

Structure

In this chapter, we will discuss the following topics:

- Introduction to Kivy and its characteristics
- Installation of Kivy
- Kivy app lifecycle
- Widgets, events, and binding function
- Geometry management using layout managers
- User pages with multiple screens
- Package Kivy files with buildozer

Objectives

The purpose of this chapter is to enable users to understand the basics of mobile application development using the Kivy library in Python. After going through this chapter, users can actively create well-defined widgets and arrange them using layout managers in innovative mobile apps.

Introduction to Kivy

A cross-platform, open-source Python framework called Kivy is used to create multi-touch applications with intuitive user interaction. It enables programmers to create programs that work on various operating systems, such as Windows, macOS, Linux, iOS, and Android. This feature makes it an excellent tool for app development because our code can run on all platforms. Kivy's app development interface is comparable to TKinter's. We can use widgets and layouts to build a GUI. It is completely free to use and is released under the MIT license. The API for the Kivy framework is clear and stable. Using a quick and contemporary pipeline, the graphics engine is constructed over OpenGL ES2. More than 20 widgets are included in the toolkit, and each can be configured easily for desirable usage.

Characteristics of Kivy

Kivy offers several characteristic features listed as follows:

- **Open-source:** Kivy is open-source and has an active developer base that actively contributes to the framework, offers support, and exchanges resources like documentation and tutorials.
- **Model-View-Controller (MVC) architecture:** The Model-View-Controller (MVC) architecture is the foundation of Kivy, and a declarative language is used to specify the user interface.
- **Support for widgets:** The framework offers a variety of widgets and features, such as buttons, labels, text inputs, and more, for building flexible and dynamic user interfaces.
- **Touch and gesture input:** One of Kivy's key characteristics is its support for touch and gesture input, which makes it ideal for creating apps for touch-based gadgets like smartphones and tablets. Other input devices, including a gamepad, keyboard, and mouse, are also supported by Kivy.
- **Based on OpenGL:** OpenGL enables quick graphics rendering and lets programmers design visually appealing apps with fluid animations and

transitions, which is the technology used to build Kivy.

Overall, Kivy is a robust and adaptable framework for creating cross-platform applications with a natural user experience, making it a popular option for developers wishing to create multi-touch applications that operate on several platforms.

Installation of Kivy

To make use of the Kivy library, we must ensure that our system is well configured with Python Environment setup and Python 3.x installation. The steps for installing the Kivy library on the Windows operating system are as follows:

1. Before installing Kivy, update the **pip** and wheel by typing the following command in the command prompt (with Administrator rights):

```
python -m pip install --upgrade pip wheel setuptools
```

2. The next step is to install the required dependencies using the following command:

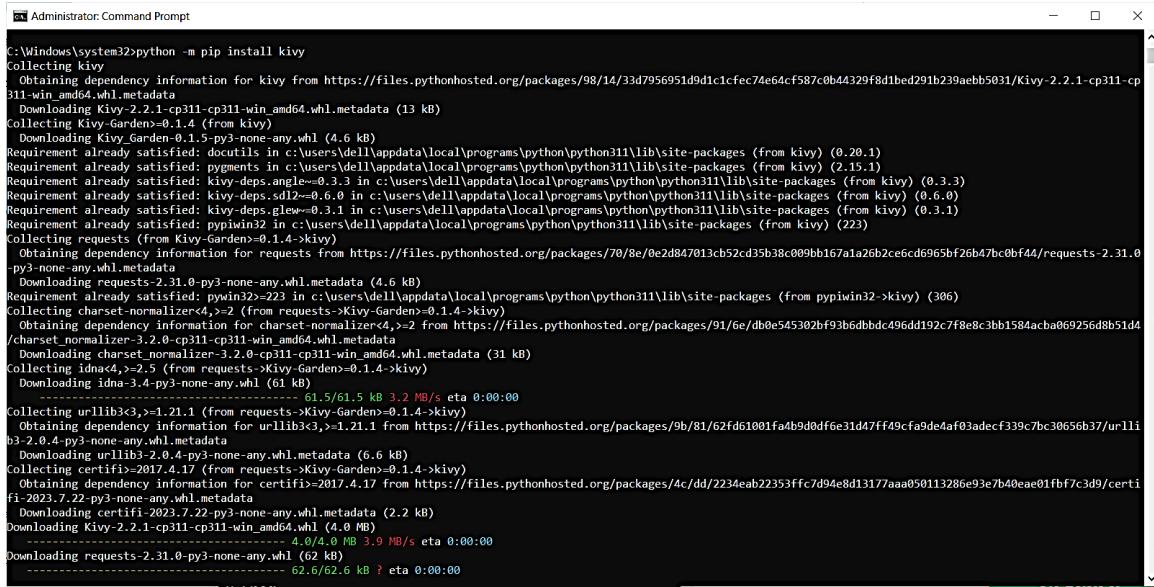
```
python -m pip install docutils pygments pypiwin32  
kivy.deps.sdl2 kivy.deps.glew
```

```
python -m pip install kivy.deps.gstreamer
```

```
python -m pip install kivy.deps.angle
```

3. Now, as a final step, we install Kivy using the following command:

```
python -m pip install kivy
```



```

Administrator: Command Prompt
C:\Windows\system32>python -m pip install kivy
Collecting kivy
  Obtaining dependency information for kivy from https://files.pythonhosted.org/packages/98/14/33d7956951d9d1c1cfec74e64cf587c0b44329f8d1bed291b239aebb5031/Kivy-2.2.1-cp311-cp
311-win_amd64.whl.metadata
    Downloading Kivy-2.2.1-cp311-cp311-win_amd64.whl.metadata (13 kB)
Collecting Kivy-Garden>=0.1.4 (from kivy)
    Downloading Kivy_Garden-0.1.5-py3-none-any.whl (4.6 kB)
Requirement already satisfied: docutils in c:\users\dell\appdata\local\programs\python\python311\lib\site-packages (from kivy) (0.20.1)
Requirement already satisfied: pygments in c:\users\dell\appdata\local\programs\python\python311\lib\site-packages (from kivy) (2.15.1)
Requirement already satisfied: kivy-deps.angle==0.3.3 in c:\users\dell\appdata\local\programs\python\python311\lib\site-packages (from kivy) (0.3.3)
Requirement already satisfied: kivy-deps.sdl2==0.6.0 in c:\users\dell\appdata\local\programs\python\python311\lib\site-packages (from kivy) (0.6.0)
Requirement already satisfied: kivy-deps.glew==0.3.1 in c:\users\dell\appdata\local\programs\python\python311\lib\site-packages (from kivy) (0.3.1)
Requirement already satisfied: pipwin32 in c:\users\dell\appdata\local\programs\python\python311\lib\site-packages (from kivy) (223)
Collecting requests (from Kivy-Garden>=0.1.4->kivy)
    Obtaining dependency information for requests from https://files.pythonhosted.org/packages/70/8e/0e2d847013cb52cd35b38c009bb167a126b2ce6cd6965bf26b47bc0bf44/requests-2.31.0
-py3-none-any.whl.metadata
    Downloading requests-2.31.0-py3-none-any.whl.metadata (4.6 kB)
Requirement already satisfied: pywin32>=223 in c:\users\dell\appdata\local\programs\python\python311\lib\site-packages (from requests->kivy) (306)
Collecting charset-normalizer<4,>=2 (from requests->Kivy-Garden>=0.1.4->kivy)
    Obtaining dependency information for charset-normalizer<4,>=2 from https://files.pythonhosted.org/packages/91/6e/db0e545302bf93b6dbbd496dd192c7f8e8c3bb1584acba069256d8b51d4
/charset_normalizer-3.2.0-cp311-cp311-win_amd64.whl.metadata
    Downloading charset_normalizer-3.2.0-cp311-cp311-win_amd64.whl.metadata (31 kB)
Collecting idna<4,>=2.5 (from requests->Kivy-Garden>=0.1.4->kivy)
    Downloading idna-3.4-py3-none-any.whl (61 kB)
----- 61.5/61.5 kB 3.2 MB/s eta 0:00:00
Collecting urllib3<3,>=1.21.1 (from requests->Kivy-Garden>=0.1.4->kivy)
    Obtaining dependency information for urllib3<3,>=1.21.1 from https://files.pythonhosted.org/packages/9b/81/62fd61001fa4b9d0df6e31d47ff49cfa9de4af03adecf339c7bc30656b37/urllib
3-2023.7.22-py3-none-any.whl.metadata
    Downloading urllib3-2.0.4-py3-none-any.whl.metadata (6.6 kB)
Collecting certifi==2017.4.17 (from requests->Kivy-Garden>=0.1.4->kivy)
    Obtaining dependency information for certifi==2017.4.17 from https://files.pythonhosted.org/packages/4c/dd/2234eb22353ffcd94e8d13177aa050113286e93e7b40eae01fbfc3d9/certi
fi-2023.7.22-py3-none-any.whl.metadata
    Downloading certifi-2023.7.22-py3-none-any.whl.metadata (2.2 kB)
Downloading Kivy-2.2.1-cp311-cp311-win_amd64.whl (4.0 MB)
----- 4.0/4.0 kB 3.9 MB/s eta 0:00:00
Downloading requests-2.31.0-py3-none-any.whl (62 kB)
----- 62.6/62.6 kB ? eta 0:00:00

```

Figure 10.1: Installation steps of Kivy library

A powerful Python IDE with numerous productivity tools for creating Python apps is PyCharm IDE. This IDE can be used to create powerful Kivy applications. We also use PyCharm IDE to implement all the examples described in this chapter.

Kivy app life cycle

Like any other mobile phone application, there are various phases in the Kivy application life cycle. It begins when the `start()` method is executed in the Python-based application. The `start()` method then sends a call to the `build()` method of the application in order to build and display the user interface. In short, to initialize our app with a widget or widget tree, we override the `build()` method in our app class and return the widget (or widget tree) so constructed. In the next phase, an infinite loop runs while waiting for the occurrence of different events, such as pausing the application, resuming the application, stopping, and destroying the application. A detailed view of a Kivy application's life cycle is portrayed in [Figure 10.2](#) as taken from Kivy's document website:

<https://kivy.org/doc/stable/guide/basic.html>. The `App` class is the base class used for creating a variety of Kivy applications. We may consider it as the major entry point into the Kivy run loop. To create our own app, we need to create a new class and make it a subclass of the `App` class. After this, an instance of the user-defined app class is created, and finally, in order to start the application's life cycle, we call the `App.run()` method with the class instance. The process of making a Kivy application is summarised in the following steps:

1. The first step is to inherit Kivy's **App** class, which represents the window for our widgets.
2. The second step is to create a **build()** method in which the content of the widgets will be given.
3. The final step is to call the **run()** method to execute the app.

The following figure displays the overall lifecycle of a simple Kivy application and summarises the steps performed for running the complete application:

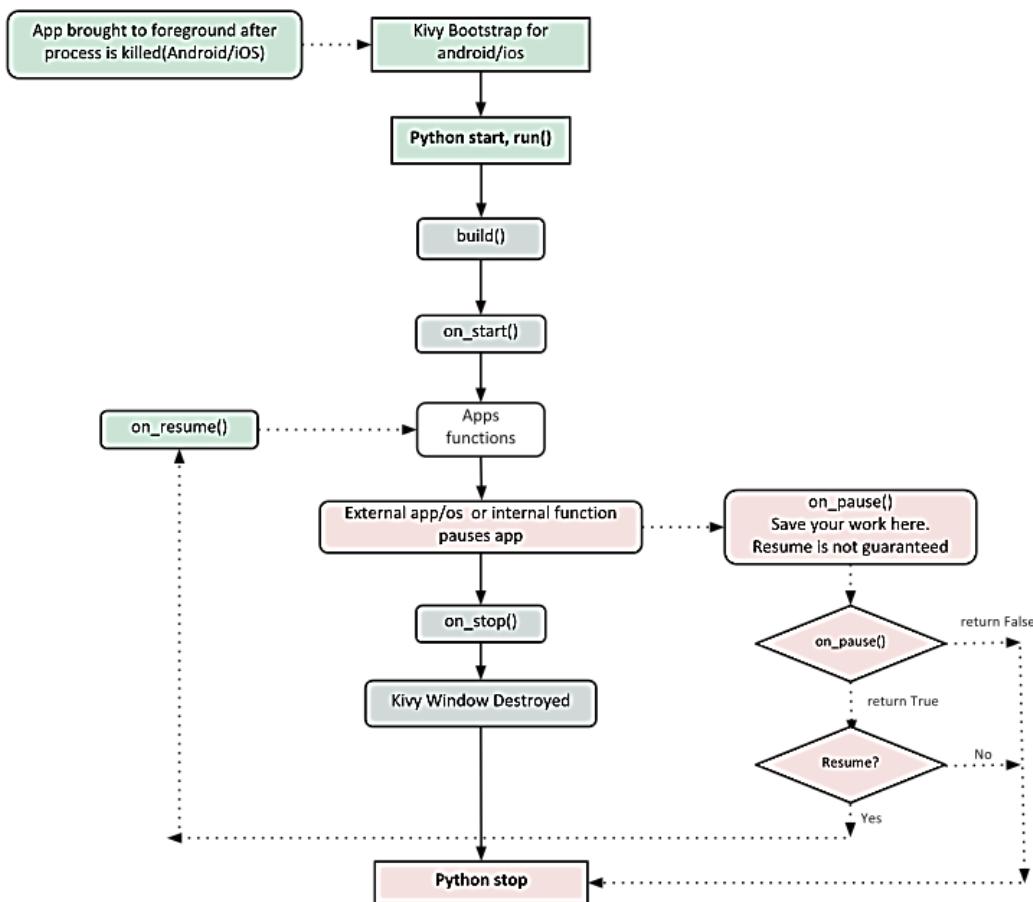


Figure 10.2: Lifecycle phases of Kivy Application1

1 Source: <https://kivy.org/doc/stable/guide/basic.html>

Let us create a simple application in Kivy to understand the overall lifecycle process. The code is given as follows:

```
# import the necessary modules
```

```
import kivy
from kivy.app import App
from kivy.uix.label import Label
# we create a class to represent the application
# window.

# The class inherits from the App class to use all its
# functions. Here the method build() is implemented to
# describe components placed on the screen. Here we return
# a Label with caption "This is a label"
class DemoApp(App):
    def build(self):
        return Label(text="This is a Label")
# To run the application we need to call run() method
# with newly created class
if __name__ == "__main__":
    DemoApp().run()
# A Python programme uses the condition if __name__ ==
# '__main__' to only run the code inside the if statement
# when the program is run directly by the Python
# interpreter. The code inside the if statement is not
# executed when the file's code is imported as a module.
```

Output:

The output can be seen in [*Figure 10.3*](#):

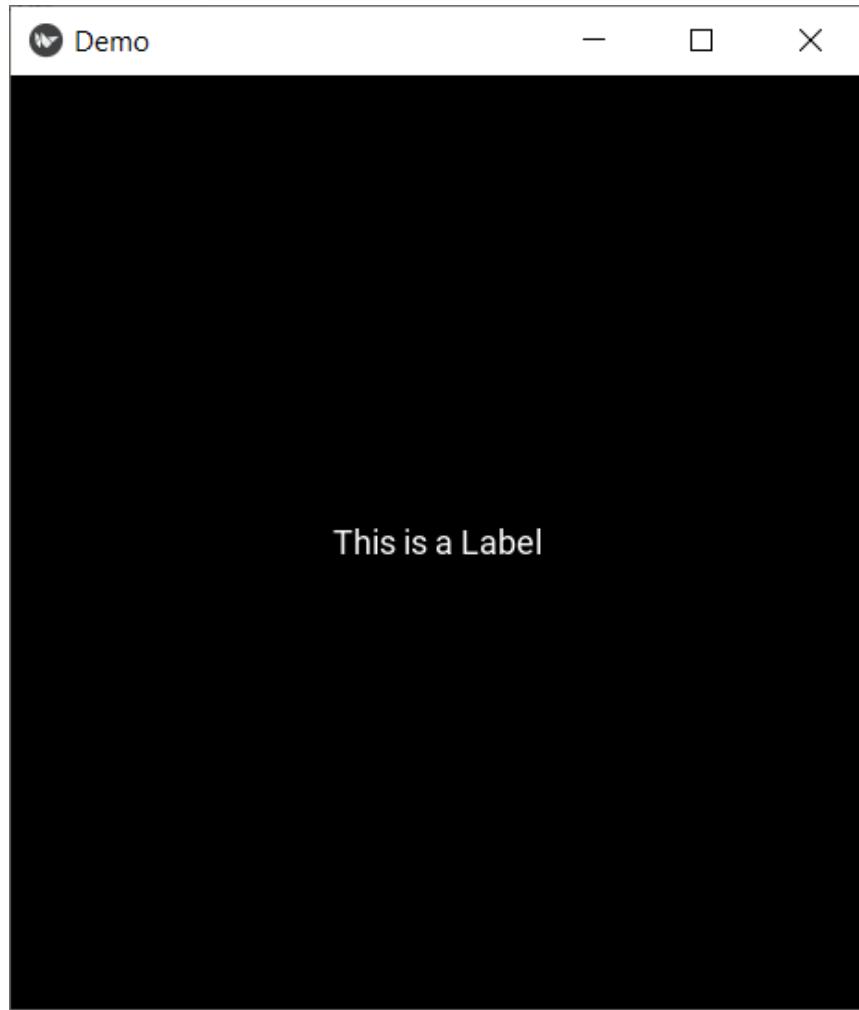


Figure 10.3: Output

Widgets and layouts

A user interface usually has many elements, such as input textboxes, labels, drop-down lists, buttons, radio buttons, and so on. These elements are called widgets in Kivy. Actually, everything in the user interface is a widget. Widgets are the building blocks of a Kivy graphic interface. Widgets are elements of a graphical user interface that form part of the User Experience. A widget is represented by a subclass of the `kivy.uix.widget.Widget` class. A widget may have properties such as `id`, `color`, `text`, `font_size`, and so on. A widget may trigger some events, such as touch down, touch move, and touch up. For example, the following kvlang code describes a label element whose text property is "`Label Demo`":

`Label:`

```
text: "Label Demo"
```

In Python convention, the class name uses so-called Pascal casing; the first letter of a word is an uppercase letter. Widget properties such as text are in lowercase. Kivy widgets can be categorized as follows:

- **UX widgets:** There are multiple UX Widgets such as Label, Button, CheckBox, Image, Slider, Progress Bar, Text Input, Toggle button, and Switch.
- **Layout Managers:** Layouts are containers used to arrange widgets in a particular manner, such as AnchorLayout, BoxLayout, FloatLayout, GridLayout, and so on.
- **Complex UX widgets:** Non-atomic widgets that are the result of combining multiple classic widgets. We call them complex because their assembly and usage are not as generic as the classical widgets. Examples: Bubble, Drop-Down List, FileChooser, Popup, Spinner, RecycleView, TabbedPanel, Video player, and VKeyboard.
- **Behaviors widgets:** These widgets do no rendering but act on the graphics instructions or interaction (touch) behavior of their children. Examples: Scatter and Stencil View.
- **Screen manager:** Manages screens and transitions when switching from one to another. Example: Screen Manager.

In this chapter, we will discuss UX widgets and Layout managers in detail. Discussion on complex UX widgets and behavior widgets is beyond the scope of this book. Users can go through these advanced widgets and their features listed in the official Kivy documentation link: <https://kivy.org/doc/stable/api-kivy.ux.html>.

UX widgets, events, and binding function

Classical user interface widgets are ready to be assembled to create more complex widgets. Examples: Label, Button, Check Box, Image, Slider, Progress Bar, Text Input, Toggle button, Switch, and Video. Let us discuss the characteristics of each of these widgets in the following table:

Widget description with syntax	Properties and events
--------------------------------	-----------------------

Widget description with syntax	Properties and events
<p>1. Label Hello World</p> <p>The label widget is used for rendering text. It supports both ASCII and Unicode strings.</p> <p>Syntax:</p> <pre data-bbox="221 502 926 819">from kivy.uix.label import Label lb1 = Label (text="Hello World!", font_size=20) lb2 = Label (text='[color=ff3333]Kivy[/color] [color=3333ff]App[/color]', markup = True)</pre>	<p>Properties</p> <ul style="list-style-type: none"> • color: The label text color in (r, g, b, a) • bold: Boolean <code>True/False</code> label text Bold. • italic: Boolean <code>True/False</code> italicize label text. • underline: Boolean <code>True/False</code> underline label text. • text: Label Text (string) value • font_size: Size of label text • markup: To format the label Text Markup tags such as: <ul style="list-style-type: none"> [b][/b] for bold text [i][/i] for italic text [u][/u] for Underlined [s][/s] for Strikethrough text [font=<str>][/font] to the font [color=<Hexadecimal code>] [/color]

Widget description with syntax	Properties and events
<p>2. Button</p>  <p>The button gets triggered and performs specified actions when it is clicked, pressed, or released:</p> <pre data-bbox="225 502 964 946"> from kivy.uix.button import Button def callback(instance): print('The <%s> button pressed' % instance.text) btn = Button(text ="Click Here", color =(1,0,.65,1), background_normal = 'D:/Images/smaple.png', size_hint = (.2, .2), pos_hint = {"x":0.41, "y":0.41}) btn1.bind(on_press=callback) </pre>	<p>Properties</p> <ul style="list-style-type: none"> • background_color: In the g, b, a) • border: List of four values (right, top, left). Default value (16) • background_normal: uses Background image • background_down: The image of the button used when is pressed. • State: determines the button changes <p>Events and methods</p> <ul style="list-style-type: none"> • bind: To attach a method to a button press (click), we use this method. • on_press: Event fired when the button is pressed. • on_release: Event fired when the button is released • trigger_action(duration): Trigger whatever action(s) bound to the button by calling on_press and on_release callbacks.

Widget description with syntax	Properties and events
<p>3. TextInput</p>  <p>This provides an editable box for inserting plain text:</p> <pre>from kivy.uix.textinput import TextInput text1= TextInput (font_size=20, multiline= False)</pre>	<p>Properties</p> <ul style="list-style-type: none"> • multiline: To create a multi-line TextInput, set the multiline property to True. • text: The text is stored in the TextInput.text property. • Focus: We can set the focus on a TextInput, meaning that the text will be highlighted, and keyboard input will be requested. No TextInput is defocused if the key is pressed. • selection_text: We can get the currently selected text in the TextInput.selection_text property. <p>Events and methods</p> <ul style="list-style-type: none"> • insert_text(): We can insert text which can be added to the TextInput by calling the insert_text() method. If you are overwriting it, you can reject unwanted characters. • on_text_validate: Event fired when the user hits the <i>Enter</i> key. It also ends the text input. • on_double_tap: Event fired when a double tap happens in the text input. • on_triple_tap: Event fired when a triple tap happens in the text input. • on_quad_touch: Event fired when four fingers are touching the text input.

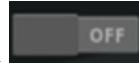
Widget description with syntax	Properties and events
<p>4. Checkbox  </p> <p>A Checkbox is a button with two states i.e., it can be checked (selected) or unchecked. Multiple checkboxes can be selected at a time:</p> <pre data-bbox="218 481 964 840">from kivy.uix.checkbox import CheckBox def on_checkbox_active(checkbox, value): if value: print('Checkbox is Active') else: print('Checkbox is Inactive')</pre>	<p>Properties</p> <ul style="list-style-type: none"> • active: Boolean value that the checkbox is Checked (Unchecked (False)). • background_checkbox: Background image of the checkbox when it is in active or checked state. • background_checkbox_inactive: Background image of the checkbox when it is not active. • color: color in (r, g, b, a) for selected checkbox.
<pre data-bbox="218 861 964 956">check1= CheckBox(active = False) check1.bind(active=on_checkbox_active)</pre>	<p>Events</p> <ul style="list-style-type: none"> • on_checkbox_active(checkbox, value): The event is raised when the checkbox is selected. The parameter refers to the current checkbox and the value selected is True (True) or False (False).

Widget description with syntax	Properties and events
<p>5. Radio Button </p> <p>No separate class for the Radio button in Kivy. A checkbox can be converted to a radio button by setting the group property. Only one Radio button can be selected at a time when the CheckBox.group property is set:</p> <pre data-bbox="220 623 964 844">from kivy.uix.checkbox import CheckBox radio1= CheckBox(group="abc", active = False) radio2= CheckBox(group="abc", active = True)</pre>	<p>Properties</p> <ul style="list-style-type: none"> • active: Boolean value that the radio button is Checked Unchecked (False). • background_rad Background image of the radio when it is in active or checked • background_radio Background image of the radio when it is not active. • color: color in (r, g, b, a) for selected radio. <p>Events</p> <ul style="list-style-type: none"> • on_checkbox_active(ctrl, value): The event is raised when the checkbox is selected. The parameter refers to the current checkbox and the value selected is True or False.

Widget description with syntax	Properties and events
<p>5. Image</p> <p>Image widget is used to display an image:</p> <pre data-bbox="213 403 850 566">from kivy.uix.image import Image img1 = Image (source = "D:/Images/sample2.png" , fit_mode="fill")</pre>	<p>Properties</p> <ul style="list-style-type: none"> • source: Filename/source of y • fit_mode: If the size of the image is different than the size of the widget, it will determine how the image is resized to fit inside the widget. Available options are as follows: <ul style="list-style-type: none"> ◦ scale-down: the image will be scaled down to fit inside the widget box, maintaining its aspect ratio and without stretching. If the size of the image is smaller than the widget, it will be displayed at its original size. ◦ fill: the image is stretched to fit the widget, regardless of its aspect ratio and dimensions. ◦ contain: the image is resized to fit inside the widget box. ◦ cover: Image will be stretched horizontally or vertically to fill the widget box, maintaining its aspect ratio.
	<ul style="list-style-type: none"> • texture: Texture object of the image. • size: size of image in terms of width and height. • pos: position of the image in terms of coordinates. • color: Image color, in the form (r, g, b, a). Default value is [1,1,1,1] <p>Events and methods</p> <ul style="list-style-type: none"> • reload(): method to reload image from disk. • remove_from_cache(): method to remove image from cache.

Widget description with syntax	Properties and events
<p>6. Video</p> <p>It is used to display video files or streams:</p> <pre data-bbox="213 397 948 967">from kivy.uix.video import Video def on_position_change(instance, value): print('Position in the video is', value) def on_duration_change(instance, value): print('Duration of the video is', value) vi1 = Video(source='clip1.avi') vi1.bind(position=on_position_change, duration=on_duration_change)</pre> <p>One can define a preview image which gets displayed until the video is started/loaded by passing the preview to the constructor:</p> <pre data-bbox="213 1115 861 1189">vi2 = Video(source='clip1.avi', preview='D:/images/img/kivy.png')</pre> <p>One can display the placeholder image when the video stops by reacting on eos:</p> <pre data-bbox="213 1315 899 1543">def on_eos_change(self, inst, val): if val and self.preview: self.set_texture_from_resource (self.preview) vi2.bind(eos=on_eos_change)</pre>	<p>Properties</p> <ul style="list-style-type: none"> • duration: Duration of the video. Defaults to -1, and is set to the duration when the video is loaded. • eos: Boolean value indicates whether the video has reached the end of the file. Defaults value False. • loaded: Boolean value indicating whether the video is loaded and ready for playback or not. • position: Position of the video, between 0 and duration. • preview: Filename/source of the image displayed before the video starts. • state: String indicates the current state: “play”, “pause”, or “stop” the video. • volume: Volume of the video, in the range 0–1. 1 means full volume, 0 means mute. <p>Events and Methods</p> <ul style="list-style-type: none"> • unload(): Unload the video. The playback will be stopped. • seek(percent, precise): Change the position to a percentage. The percentage must be between 0 and 1.

Widget description with syntax	Properties and events
<p>7. Slider </p> <p>The slider widget supports horizontal and vertical orientations and is used as a scrollbar:</p> <pre data-bbox="225 466 926 677">from kivy.uix.slider import Slider slider1 = Slider(orientation='vertical', value_track=True, value_track_color=(1,0,0,1))</pre>	<p>Properties</p> <ul style="list-style-type: none"> • cursor_size: Size of the cursor image. • cursor_width: Width of the cursor image. The default value is 32s. • max: Maximum value allowed. The default value is 100. • min: Minimum value allowed. The default value is 0. • orientation: horizontal or vertical. Orientation of the slider. The default value is horizontal. • range: The range of the slider. Format (minimum value and maximum value) • step: Step size of the slider. • value: Current value used for the slider. <p>Events and Methods</p> <ul style="list-style-type: none"> • on_touch_down(touch): touch-down event. • on_touch_move(touch): touch move event. • on_touch_up(touch): touch-up event.
<p>8. Progress Bar </p> <p>It is used to track the progress of any task. Only the horizontal mode is currently supported; the vertical mode is not yet available. The progress bar has no interactive elements and is a display-only widget:</p> <pre data-bbox="225 1558 861 1746">from kivy.uix.progressbar import ProgressBar progress1 = ProgressBar (max=100) progress1.value = 40</pre>	<p>Properties</p> <ul style="list-style-type: none"> • max: Maximum value allowed. The default value is 100. • value: Current value used for the progress bar. If the value is < 0 or > max, it is normalized to those boundaries.

Widget description with syntax	Properties and events
<p>9. Switch</p> <p>It is just like a mechanical switch </p> <p>that can be either on or off. The user can swipe to the left/right to activate/deactivate it. If you want to control the state with a single touch instead of a swipe, use the <code>ToggleButton</code> instead.</p>	<p>Properties</p> <ul style="list-style-type: none"> • <code>active</code>: Indicate whether the switch is active or inactive. <p>Events and Methods</p> <ul style="list-style-type: none"> • <code>on_touch_down(touch)</code>: touch-down event.
<pre>from kivy.uix.switch import Switch sw1 = Switch(active=True)</pre>	<ul style="list-style-type: none"> • <code>on_touch_move(touch)</code>: touch move event. • <code>on_touch_up(touch)</code>: touch-up event.
<p>10. Toggle Button </p> <p>It acts like a checkbox; when you touch or click it, the state toggles. When you click on the toggle button, it changes state from <code>normal</code> to <code>down</code>.</p> <pre>from kivy.uix.togglebutton import ToggleButton tg1= ToggleButton (text = "python", border= (26, 26, 26, 26), font_size=200)</pre>	<p>Properties</p> <ul style="list-style-type: none"> • <code>group</code>: Group of the button. If a group will be used (the buttons are independent). <p>Note: We can use the same properties as the <code>Button</code> class.</p> <p>Events and Methods</p> <ul style="list-style-type: none"> • <code>on_state</code>: Refers to the state of the button “normal” and “down”. Toggle buttons can also be used to make radio buttons—only one group can be in a “down” state

Widget description with syntax	Properties and events
<p>11. Clock</p> <p>The Clock object allows you to schedule a function call in the future, once or repeatedly, at specified intervals. You can get the time elapsed between the scheduling and the calling of the callback via the dt (delta-time) argument.</p> <pre data-bbox="218 557 904 994"> def func1(dt): pass # call func1 every 0.5 seconds Clock.schedule_interval(func1, 0.5) # call func1 in 5 seconds once Clock.schedule_once(func1, 5) # call func1 before next frame ev=Clock.create_trigger(func1) </pre>	<p>Properties</p> <ul style="list-style-type: none"> • <code>max_iteration</code>: To set the limit of the number of callbacks. It has a default value of 0. <p>Events and methods</p> <ul style="list-style-type: none"> • <code>schedule_interval</code>: To schedule a function call repeatedly at regular intervals. • <code>schedule_once</code>: To schedule a function call once or only at a specific time. • <code>create_trigger</code>: It will call the callback function once before the next frame.

Table 10.1: Various UX widgets in the Kivy library with properties and syntax

Geometry management using layout managers

Layouts are containers used to arrange widgets in a particular manner. A layout widget does no rendering but just acts as a trigger that arranges its children in a specific way. Major Kivy layouts are Anchor Layout, Box Layout, Float Layout, Grid Layout, and Stack Layout. Let us discuss each of these in detail in the following table.

Layout description	Module with syntax

Layout description	Module with syntax	
<p>1. AnchorLayout</p> <p>Widgets can be anchored to the “top”, “bottom”, “left”, “right”, or “center.”</p>	<pre>from kivy.uix.anchorlayout import AnchorLayout layout=AnchorLayout(anchor_x='right', anchor_y='bottom') btn = Button(text='Hello World') layout.add_widget(btn)</pre>	
<p>2. BoxLayout</p> <p>Widgets are arranged in a sequence in either a “vertical” or a “horizontal” order. Position hints are partially working, depending on the orientation:</p> <ul style="list-style-type: none"> • If the orientation is vertical, x, right, and center_x will be used. • If the orientation is horizontal, y, top, and center_y will be used. 	<pre>from kivy.uix.boxlayout import BoxLayout To position widgets above/below each other, use a vertical BoxLayout: layout = BoxLayout(orientation='vertical') btn1= Button(text='Java') btn2= Button(text='Python') layout.add_widget(btn1) layout.add_widget(btn2) Here, we use a horizontal BoxLayout with 10-pixel spacing between children. The first button covers 60% of the horizontal space, and the second button covers 40% of the horizontal space: layout=BoxLayout(spacing=10) btn1 = Button (text='Java', size_hint= (.6, 1)) btn2 = Button (text='Python', size_hint= (.4, 1)) layout.add_widget(btn1) layout.add_widget(btn2)</pre>	

Layout description	Module with syntax	
<p>3. FloatLayout</p> <p>Widgets are essentially unrestricted. <code>FloatLayout</code> sets the <code>pos_hint</code> and the <code>size_hint</code> properties of its widgets for positioning and managing them.</p>	<pre>from kivy.uix.floatlayout import FloatLayout layout = FloatLayout(size=(300, 300)) By default, all widgets have their <code>size_hint = (1, 1)</code>, so this button will adopt the same size as the layout. To create a button 50% of the width and 25% of the height of the layout and positioned at (20, 20), you can do the following:</pre> <pre>button = Button(text='Hello world', size_hint=(.5, .25), pos=(20, 20))</pre>	S P
<p>4. GridLayout</p> <p>Widgets are arranged in a grid defined by the <code>rows</code> and <code>cols</code> properties. A <code>GridLayout</code> must always have at least one input constraint: <code>GridLayout.cols</code> or <code>GridLayout.rows</code>. The properties used in Grid Layout:</p> <ul style="list-style-type: none"> • <code>cols</code>: Number of columns in the grid. • <code>rows</code>: Number of rows in the grid. • <code>Spacing</code>: Spacing between inside widgets (children): <code>[spacing_horizontal, spacing_vertical]</code> 	<pre>from kivy.uix.gridlayout import GridLayout layout = GridLayout(cols=2, row_force_default=True, row_default_height=40) layout.add_widget(Button(text='Hello 1', size_hint_x=None, width=100)) layout.add_widget(Button(text='World 1')) layout.add_widget(Button(text='Hello 2', size_hint_x=None, width=100)) layout.add_widget(Button(text='World 2')) Here, <code>row_force_default</code> parameter in <code>GridLayout</code>, when set to <code>True</code>, ignores the height and <code>size_hint_y</code> of the child and uses the default row height set by the <code>row_default_height</code> parameter.</pre>	

Layout description	Module with syntax	
<p>5. StackLayout</p> <p>Widgets are stacked in a lr-tb (left-to-right, then top-to-bottom) or tb-lr order. It arranges children in vertical order or in horizontal, as many as the layout can fit. The size of the individual children's widgets need not be uniform.</p>	<pre data-bbox="641 280 1334 671">from kivy.uix.stacklayout import StackLayout layout1 = StackLayout (orientation = 'lr-tb') for i in range(10): nbtn = Button(text= str(i), width=100, size_hint=(None,0.20)) layout1.add_widget(nbtn)</pre>	

Table 10.2: Different layout managers in Kivy with diagrams

When you add a widget to a layout, the following properties are used to determine the widget's size and position, depending on the type of layout:

- `size_hint`: defines the size of a widget as a fraction of the parents' size. Values are restricted to the range 0.0–1.0, that is, 0.01 = 1/100th of the parent size (1%) and 1 = same size (100%).
- `pos_hint`: is used to place the widget relative to the parent.

The `size_hint` and `pos_hint` are used to calculate a widget's size and position only if the value(s) are not set to `None`. If you set these values to `None`, the layout will not position/size the widget, and you can specify the values (x, y, width, height) directly in screen coordinates. Users can try using complex widgets and different behavior widgets in combination with layout managers for creating.

Basics of KV language

An alternative approach to define a user interface is to use the Kv language for app development. As the application grows, the interface becomes complex and hard to maintain. In order to overcome these shortcomings, we can use KV language to create user applications. The KV language is also known as `kvlang` or simply the Kivy language. It allows us to create widgets in a declarative way and enable us to bind widget properties to each other or to callbacks in a natural manner. Quick prototypes and stable configuration changes can be easily added to the application user interface. This enables users to separate the application logic from the UI. The KV language is a simple markup language, just like HTML. It

uses various tags and configuration properties to create a user interface with the help of a simple markup language with the flavor of Python in it.

Loading the KV File

There are the following two ways to load KV code into your application:

- **Using the naming convention:** The Kivy application looks for a Kv file having the same name as the user-defined App class, but all characters must be in lowercase. Also, the word “App” is not used in the Kv file name. For example: If the user app class name is **FirstDemoApp**, then the name of the subsequent Kv file is **firstdemo.kv**.

If this file defines a Root Widget, it will be attached to the App’s root attribute and used as the base of the application widget tree.

- **Using the Builder class:** We can directly load a string or a file in a Kivy application using the Builder class. If a specific file defines a root widget, it will be returned by the method **load_file()**. The syntax is as follows:

```
Builder.load_file('path/to/file.kv')
```

In case a string defines a root widget, it will be returned by the method **load_string()**. The syntax is as follows:

```
Builder.load_string(kv_string)
```

Syntax guidelines for KV file

A KV source consists of rules which are used to describe the content of a Widget. You can have one root rule and any number of class or template rules. The root rule is declared by declaring the class of your root widget, without any indentation, followed by a colon “:” and will be set as the root attribute of the App instance. Widget describes a class rule, declared by the name of a widget class between < > and followed by a colon “:” and defines the appearance and behavior of any instance of that class such as: <MyWidget>:

Rules use indentation for delimitation, like Python. Indentation should be four spaces per level, as the Python style guide recommends.

Modules and widgets

There is a special syntax to define values for the whole Kv context. To access Python modules and classes from kv, use **#:import**. For example:

```
#:import isdir os.path.isdir
#:import np numpy
```

To set a global value, use **#:set**. For example:

```
#:set name value
```

In Python it is equivalent to: `name = value`

The following are syntax definitions defined in Kvlang Reference. Consider the following syntax as a part of the rule definition. The point to note here is that several widgets can share the same definition as the way tags do in CSS. The syntax is given as follows:

```
<Widget1, Widget2>:
    # .. rule definitions ...
<Widget3>:
    # .. rule definitions ...
```

Let us see an example of the same. In case both classes share the same **.kv** style, then we can reuse the style for both widgets in the **.kv** file:

```
<FirstWidget, SecondWidget>:
    Button:
        on_press: root.text(text_input.text)
    TextInput:
        id: text_input
```

By separating the class names with a comma, all the classes listed in the declaration will have the same kv properties. Now consider the following syntax for creating a root widget:

```
NewRootClassName:
    # .. rule definitions ...
```

Also, here, we can observe the syntax for creating a dynamic class:

```
<NewWidget@BaseClass>:
```

```
# .. rule definitions ...
```

Regardless of whether it is a root widget or a dynamic class, the definition should look like the syntax given as follows. With the brackets `<>` it is a rule. Without brackets, it is a root widget:

```
<NewClassName>:  
    prop1: value1  
    prop2: value2  
  
    canvas:  
        CanvasInstruction1:  
            canvasprop1: value1  
        CanvasInstruction2:  
            canvasprop2: value2
```

```
SomeOtherClass:  
    prop3: value1
```

Here, `prop1` and `prop2` are the properties of `NewClassName`, and `prop3` is the property of `SomeOtherClass`. If the widget does not have a property with the given name, a property will be automatically created and added to the widget. In the preceding example, an instance of the `SomeOtherClass` will be created and added as a child of the `NewClassName` instance. Let us consider an example of the same:

```
NewRootWidget:  
    BoxLayout:  
        Label:  
        Button:
```

The preceding example defines that our root widget, an instance of `NewRootWidget`, has a child that is an instance of the `BoxLayout`, and that `BoxLayout` further has two children, instances of the `Label` class and `Button` class.

When you specify a property's value or an event callback, the value is evaluated as a Python expression. This expression can be static or dynamic, which means that the value can use the values of other properties using reserved keywords. The list is given as follows:

- `self`: It refers to the current widget instance. For example, `self.state`
- `root`: It refers to the root widget. For example, `root.property1`
- `app`: The keyword `app` denotes the Kivy app instance. For example, `app.name`
- `args`: The `args` keyword is used in `on_<action>` callbacks referring to the arguments passed to the callback: `args[1]`.
- `Ids`: The keyword `id` is only used in kvlang for external references. It is a `weakref` to a widget, not the widget itself. The original widget can be accessed with `id.__self__`. For example, if a button has an id of `btn1`, the button is `btn1.__self__`, and its state is `id.state`. When a KV file is loaded, all defined ids are added to the ids dictionary of the root widget. It can be accessed in two methods: `root.ids[btn1].state` or `root.ids.btn1.state`.

An expression as a property value must be in a single line. If an expression is used as an `on_<action>` callback, it can have multiple lines, but all lines must have the same indentation level else the code stands invalid. **For example**, the following is valid:

```
on_state:  
    if self.state == 'normal': \  
        print('State is normal')
```

Events and properties

The Kivy language detects properties in your value expression and will create callbacks to automatically update the property via your expression when changes occur. For example,

Button:

```
text: str(self.state)
```

In this example, the parser detects that `self.state` is a property of the `Button` widget. The state property of the button can change when the user touches it. We now want this button to display its own state as text, even as the state changes. We also convert the state to a string representation. Now, whenever the button state changes, the text property will be updated automatically. We can bind to events in KV using the “`:`” syntax, that is, associating a callback to an event:

Widget:

```
on_size: my_callback()
```

You can pass the values dispatched by the signal using the `args` keyword. You can also handle `on_` events inside your KV language. For example, the `TextInput` class has a `focus` property whose auto-generated `on_focus` event can be accessed inside the KV language like the following:

`TextInput`:

```
on_focus: print(args)
```

Dynamic class

Dynamic classes allow you to create new widgets on-the-fly, without any Python declaration in the first place. The syntax of the dynamic classes is like the Rules, but you need to specify the base classes you want to subclass. Any new properties, usually added in Python code, should be declared first. If the property does not exist in the dynamic class, it will be automatically created as an appropriate typed property. Instead of having to repeat the same values for every button, we can just use a template instead with a dynamic class, like the following:

```
<DynamicButton@Button>:
```

```
    text_size: self.size
    font_size: '30sp'
    markup: True
```

```
<MyWidget>:
```

```
    DynamicButton:
        text: "Click First Button"
```

```
    DynamicButton:
```

```
    text: "Second Button "
DynamicButton:
    text: "Another Button"
```

The **DynamicButton** class, created just by the declaration of this rule, inherits from the **Button** class and allows us to change default values and create bindings for all its instances without adding any new code on the Python side.

Widget reference

In a widget tree, there is often a need to access or reference other widgets. The KV Language provides a way to do this using id. Think of them as class-level variables that can only be used in the KV language. Consider the following code snippet:

```
<FirstWidgetDemo>:
    Button:
        id: f_id1
    TextInput:
        text: f_id1.state
<SecondWidgetDemo>:
    Button:
        id: s_id1
    TextInput:
        text: s_id1.state
```

An ID is limited in scope to the rule it is declared in, so in the preceding code **s_id1**, but cannot be accessed outside the **<SecondWidgetDemo>** rule. An id is a weak reference, that is, **weakref** to the widget and is not the widget itself. Consequently, storing the id is not sufficient to keep the widget from being garbage collected. To demonstrate the following:

```
<WidgetAddRemove>:
    lbl_wgt: lbl_wgt
```

Button:

```
text: 'Click to Add widget'  
on_press: root.add_widget(lbl_wgt)
```

Button:

```
text: 'Click to Remove widget'  
on_press: root.remove_widget(lbl_wgt)
```

Label:

```
id: lbl_wgt  
text: 'Label Widget'
```

The line **lbl_wgt: lbl_wgt** means that the label with the ID **lbl_wgt** will be mapped to **lbl_wgt** in the MainDemo.py file so that any action that manipulates **lbl_wgt** will be mapped on the label with the specified name. Although a reference to **lbl_wgt** is stored in **WidgetAddRemove**, it is not sufficient to keep the object alive once other references have been removed because it is only a weakref. Therefore, after the remove button is clicked, it removes any direct reference to the widget, and the window is resized. This calls the garbage collector, resulting in the deletion of **lbl_wgt**; when the Add Button is clicked to add the widget back, a **ReferenceError** weakly-referenced object no longer exists will be raised. To keep the widget alive, a direct reference to the **lbl_wgt** widget must be kept. This is achieved using **id.__self__** or **lbl_wgt.__self__** in this case. The alternate way to achieve it in the correct way would be the following:

<WidgetAddRemove>:

```
lbl_wgt: lbl_wgt.__self__
```

Let us consider the complete example to add or remove a label widget using button widgets in a .kv file. The following code is written in a file named **WidgetAddRemove.py**:

```
from kivy.app import App  
from kivy.uix.widget import Widget  
from kivy.uix.boxlayout import BoxLayout
```

```

class WidgetAddRemove(BoxLayout):
    pass

class KivyDemoApp(App):
    def build(self):
        new_widget = WidgetAddRemove()
        return new_widget

if __name__ == '__main__':
    KivyDemoApp().run()

```

As per the KV language file naming convention, the subsequent KV file is named as **kivydemo.kv** because the class that is the entry point of the application is **KivyDemoApp**. The code is written in **kivydemo.kv** file is as follows:

```

<WidgetAddRemove>:
    lbl_wgt: lbl_wgt.__self__
    Button:
        text: 'Click to Add widget'
        on_press: root.add_widget(lbl_wgt)
    Button:
        text: 'Click to Remove widget'
        on_press: root.remove_widget(lbl_wgt)
    Label:
        id: lbl_wgt
        text: 'Label Widget'

```

Output:

The output can be seen in *Figure 10.4*:

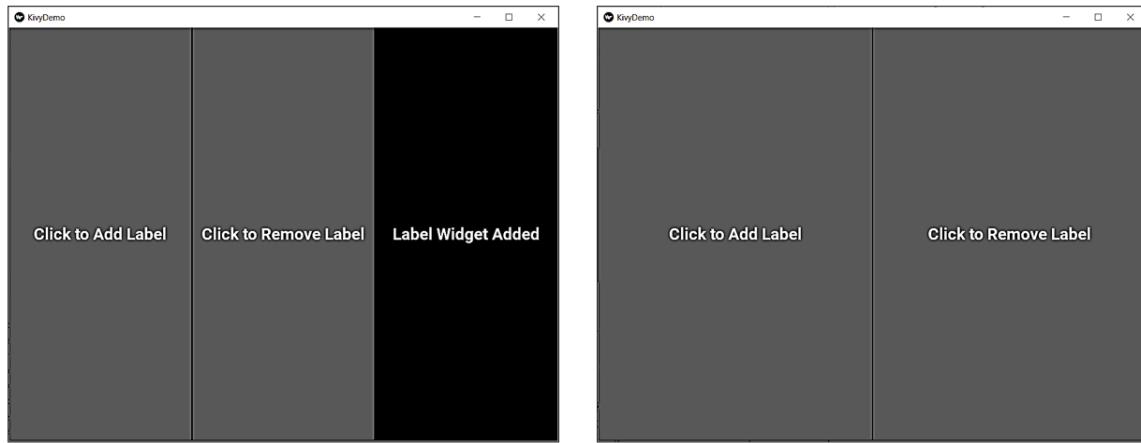


Figure 10.4: Output

On executing the `WidgetAddRemove.py` we obtained the preceding output:

User pages with multiple screens

A widget called the screen manager is used to handle several screens in your program. The default ScreenManager only shows one screen at a time and switches between screens using a TransitionBase. There are several transitions supported. The classes imported are the ScreenManager and Screen classes. The import statement required for these classes is as follows:

```
from kivy.uix.screenmanager import ScreenManager,  
Screen
```

SlideTransition with parameters for direction and duration is the transition type used by default by **ScreenManager**. By modifying the **ScreenManager.transition** property, we may quickly change transitions. We have multiple transitions available by default, such as:

- **NoTransition**: This type of transition switches screens instantly with no animation.
- **SlideTransition**: This transition slides the screen in/out from any direction. Default transition.
- **CardTransition**: In this type of transition, a new screen slides over the previous one depending on the usage mode.
- **SwapTransition**: This transition is an implementation of the iOS swap transition.

- **FadeTransition**: Transition that shades to fade the screen in or out.
- **WipeTransition**: Transition that shades to wipe the screens from right to left.
- **FallOutTransition**: Transition in which the old screen “falls” and becomes transparent, displaying the new one behind it.
- **RiseInTransition**: Here new screen rises from the center of the screen while fading from transparent to opaque.

Let us create a simple application with multiple screens.

First, create a file named **main.py** with the following code defining classes for multiple screens:

```
from kivy.app import App
from kivy.lang import Builder
from kivy.uix.screenmanager import (ScreenManager, Screen, NoTransition, SlideTransition, CardTransition, SwapTransition, FadeTransition, WipeTransition, FallOutTransition, RiseInTransition)

class FirstScreen(Screen):
    pass

class SecondScreen(Screen):
    pass

class ScreenManagement(ScreenManager):
    pass

demo = Builder.load_file("multidemo.kv")
screen_manager =
ScreenManager(transition=WipeTransition())
```

```
# Add the screens to the manager and then supply a name
# that is used to switch screens
screen_manager.add_widget(FirstScreen(name="FirstScreen"))
screen_manager.add_widget(SecondScreen(name="SecondScreen"))

class MultiDemoApp(App):
    def build(self):
        return screen_manager

MultiDemoApp().run()
```

The second file is the KV file named as **multidemo.kv** with the following code:

ScreenManagement:

FirstScreen:

SecondScreen:

<FirstScreen>:

name: 'first'

Button:

on_release: app.root.current = 'SecondScreen'

text: 'Click to view Screen'

font_size: 100

<SecondScreen>:

name: 'second'

Button:

```

on_release: app.root.current = 'FirstScreen'
text: 'Click to navigate back'
font_size: 100

```

Output:

The output can be seen in *Figure 10.5*:

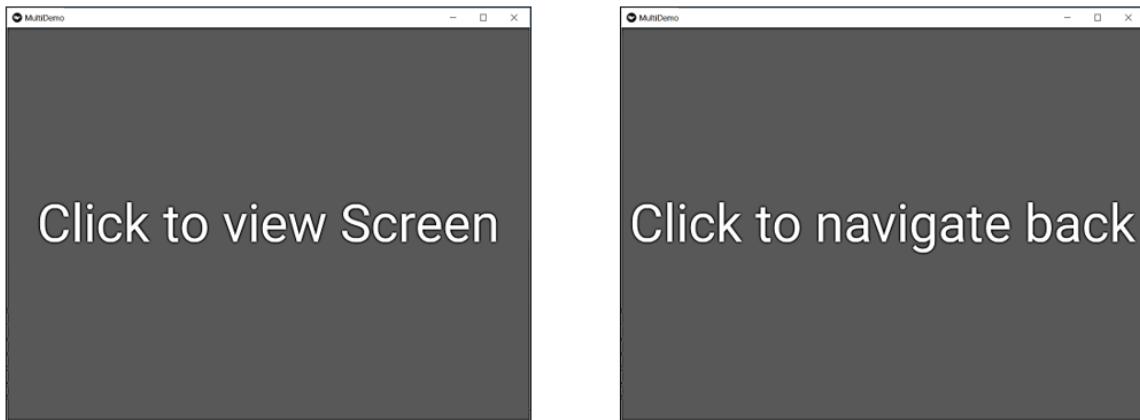


Figure 10.5: Output

Here, we define the screen manager as a parent and attach our two screens to it. Then, we include our screen parents. We give each one a name within those. This name is an associated name that we may use to switch to it in the screen manager. Next, we add a button that opens the other screen on both screens. With the exception of the **on_release** option, the buttons have the parameters that we have already seen thus far. When the button is released, we may write some code using the **on_release** function. We are requesting that the **app.root** update the current attribute in this situation to the other page. We are requesting that the **app.root** update the current attribute in this situation to the other page. We obtain the **.current** function from the **ScreenManager** class, which is referred to by **app.root** and inherits from the **ScreenManager**.

Package Kivy applications with buildozer

As Android phones only allow **Android Application Packages (APKs)**, the Python programs created with Kivy cannot be transmitted straight to these devices. Instead, we must package them properly. The key conversion components, buildozer, and Python-for-Android are only supported on Linux-based computers at the moment; hence, this conversion procedure can only be

performed on a Linux machine (for the time being). This presents a hurdle for aspiring programmers who often use Windows machines for all of their development needs. Other difficulties include unsuccessful app conversions, app crashes at launch, or internet connectivity issues. For Windows systems, Google offers a free Linux environment known as “Google Colaboratory” that can be used freely for the migration of Kivy to apk files. Here, we get a virtual computer with 75 GB of storage, 12 GB of RAM, and roughly 12GB of GPU power. This platform can be used for model training, log checking, or Python code execution. The only thing left to do is install the requirements and start the procedure because this system is already Linux-based. Before continuing, let us examine the conversion’s flow:

- Ensuring that the main.py file is the app’s entry point
- Setting up the prerequisites
- Start up the bulldozer
- Change the specs file
- Begin the procedure

The stepwise procedure is given as follows:

1. Search and open a new notebook in Google Colaboratory at <https://colab.google/>:

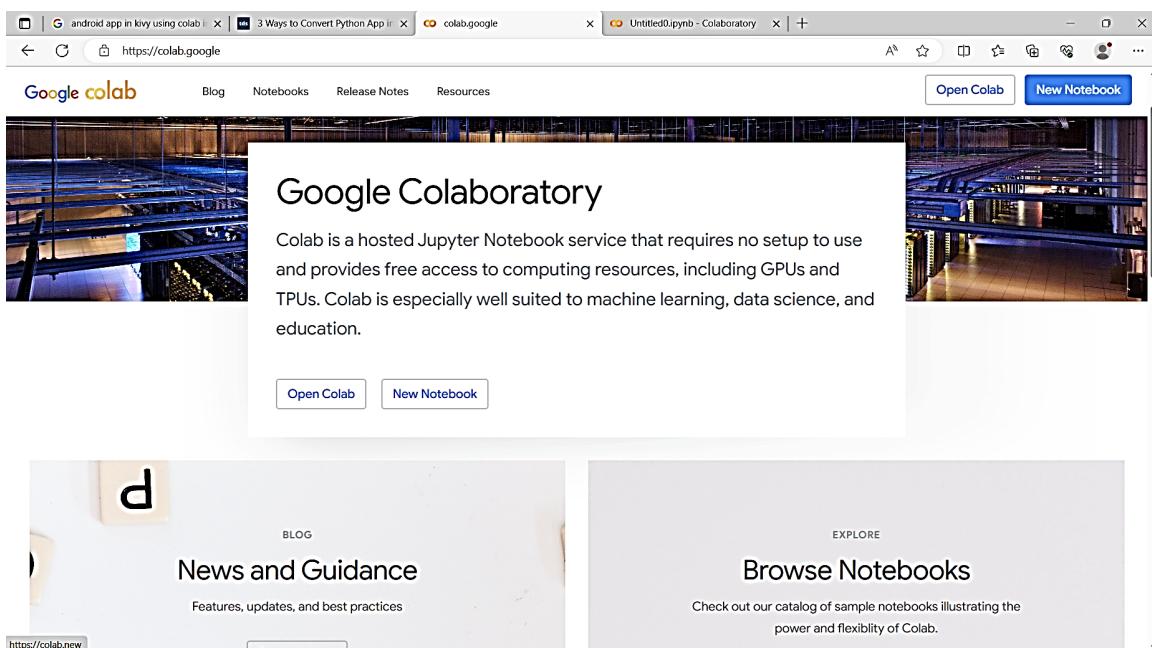


Figure 10.6: Google collaborative home page

2. In the leftmost pane named Files, click to upload the Python file along with the KV file. Please note that the Python file to be uploaded must be named as **main.py**. Here, for demonstration purposes, we use the same example that we created in the preceding section for navigating multiple screens:

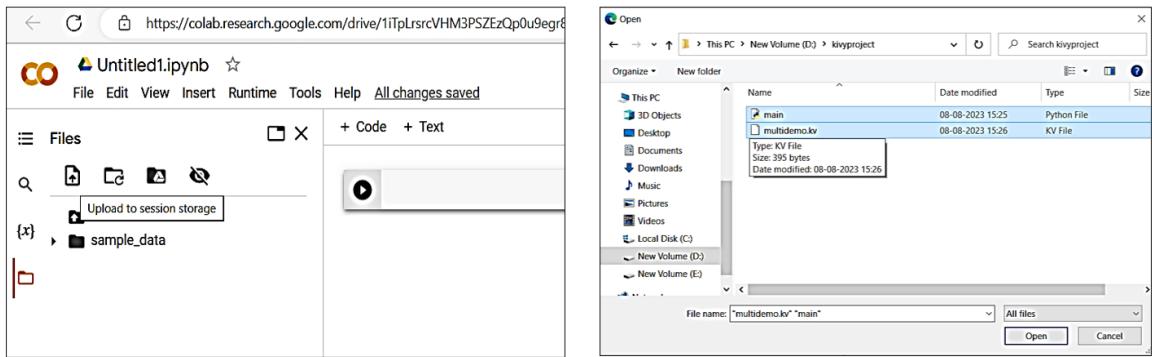


Figure 10.7: Selecting and uploading files in collab runtime

3. After uploading the necessary files, we need to write the following code step-wise in collab code cells:

The image shows a screenshot of a web browser window for Google Colab. The 'Files' pane on the left shows 'sample_data', 'main.py', and 'multidemo.kv'. The main area contains a code cell with the following Python code:

```

1 from kivy.app import App
2 from kivy.lang import Builder
3 from kivy.uix.screenmanager import (ScreenManager,
4 SlideTransition, CardTransition, SwapTransition,
5 FaderTransition, WipeTransition, FallOutTransition,
6
7 class FirstScreen(Screen):
8     pass
9
10 class SecondScreen(Screen):
11     pass
12
13 class ScreenManagement(ScreenManager):
14     pass
15
16 def = Builder.load_file("multidemo.kv")
17 screen_manager = ScreenManager(transition=wipeTransition)
18
19 # Add the screens to the manager and then supply a
20 # that is used to switch screens
21 screen_manager.add_widget(firstScreen(name="firstScreen"))
22 screen_manager.add_widget(secondScreen(name="secondScreen"))
23
24 class MultiDemoApp(App):
25     def build(self):
26         return screen_manager
27
28 MultiDemoApp().run()

```

Figure 10.8: Adding code cell to write conversion code

4. Now, we add a new code cell and run the following commands as one step. The following code will update the libraries and install Cython along with a virtual environment:

```

!sudo apt update

!sudo apt install -y git zip unzip openjdk-17-jdk
python3-pip autoconf libtool pkg-config zlib1g-dev
libncurses5-dev libncursesw5-dev libtinfo5 cmake libffi-
dev libssl-dev

!pip3 install --user --upgrade Cython==0.29.33 virtualenv

```

5. Next, we add a new code cell and insert the following code to copy, clone, and install **buildozer** library from the GitHub repository:

```

# git clone, for working on buildozer
!git clone https://github.com/kivy/buildozer
%cd buildozer
!python setup.py build
!pip install -e .
%cd ..

```

6. After the files have been uploaded, run the following command:

```
!buildozer init
```

This will create a file named **buildozer.spec** and a buildozer repository as well. Next, we double-click the file to edit the title name and the package name as needed. Here, we use the default title and package names as given in **buildozer.spec** file.

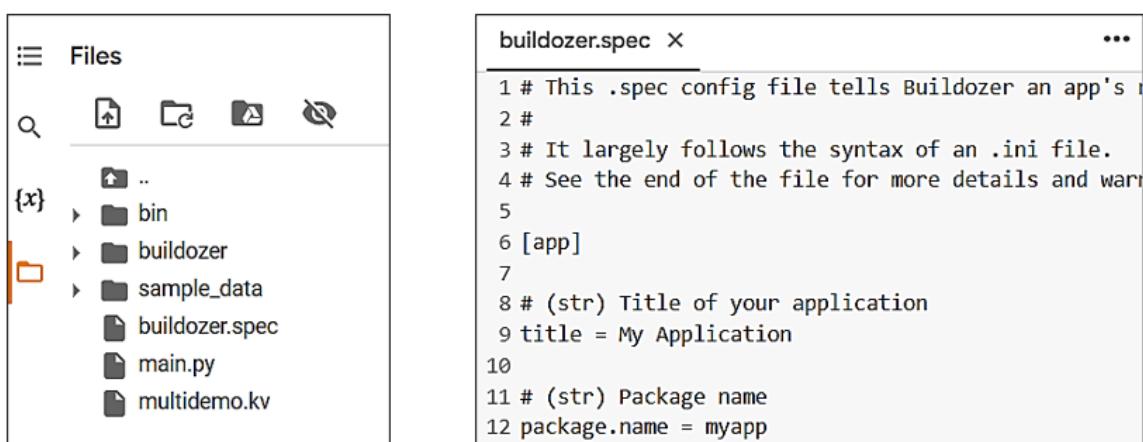


Figure 10.9: buildozer.spec file along with folder created

7. Next, scroll down to the requirements and change them as follows:

```
requirements = python3, kivy==2.1.0, kivymd==1.1.1,  
sdl2_ttf==2.0.15, pillow
```

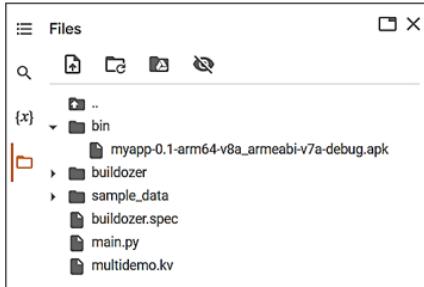
```
43 # (list) Application requirements  
44 # comma separated e.g. requirements = sqlite3,kivy  
45 requirements = python3, kivy==2.1.0, kivymd==1.1.1, sdl2_ttf==2.0.15, pillow
```

Figure 10.10: Updating requirements list in buildozer.spec file

4. If your application requires special permissions such as internet or storage, scroll down to Android.permissions and uncomment the required permissions.

5. Now, to package our application. Run the following command:

```
!buildozer -v android debug
```



```
[INFO]: myapp has compatible recipes, using this one  
[INFO]: # Copying android package to current directory  
[INFO]: # Android package filename not found in build output. Guessing...  
[INFO]: # Found android package file: /content/.buildozer/android/platform/build-arm64-v8a_armeabi-v7a/dists/myapp/bui  
[INFO]: # Add version number to android package  
[INFO]: # Android package renamed to myapp-debug-0.1.apk  
[DEBUG]: -> running cp /content/.buildozer/android/platform/build-arm64-v8a_armeabi-v7a/dists/myapp/build/outputs/apk/d  
WARNING: Received a --sdk argument, but this argument is deprecated and does nothing.  
No setup.py/pyproject.toml used, copying full private data into .apk.  
Applying Java source code patches...  
Applying patch: src/patches/SDLActivity.java.patch  
# Android packaging done!  
# APK myapp-0.1-arm64-v8a_armeabi-v7a-debug.apk available in the bin directory
```

Figure 10.11: The myapp.apk android package created in the bin folder

This is a long process and can take up to 30 minutes to complete.

At the end of this process, we can find the apk file generated in the bin folder. We can download this file by right-clicking and selecting the download option. We can copy the .apk to any Android phone to use.

Conclusion

Mobile applications have become a part of users' daily lives because they are easy to use, quite handy, and very interactive in nature. Python allows us to create beautiful mobile applications using the Kivy library in Pycharm IDE. This chapter lays a firm foundation for developing Kivy mobile applications that can be converted to Android-usable apk files with the help of buildozer library and related dependencies.

Points to remember

- A cross-platform, open-source Python framework called Kivy is used to create multi-touch applications with intuitive user interaction.
- We can use widgets and layouts to build a GUI. It is completely free to use and is released under the MIT license. The API for the Kivy framework is clear and stable.
- The Kivy application life cycle begins when the **start()** method is executed in the Python-based application. It then sends a call to the **build()** method of the application in order to build and display the user interface. Finally, in order to start the application's life cycle, we call the **App.run()** method with the class instance.
- A widget is represented by a subclass of the **kivy.uix.widget.Widget** class. A widget may have properties such as id, color, text, font size, and so on.
- Layouts are containers used to arrange widgets in a particular manner. Major Kivy layouts are Anchor Layout, Box Layout, Float Layout, Grid Layout, and Stack Layout.
- An alternative approach to define a user interface is to use the Kv language for app development.

Exercise

Attempt the following projects.

Sample project with solution

Create a simple mobile application using the Kivy library to demonstrate the working of a simple calculator. We can use a Python file for defining application

classes and a KV file to define widgets for the application along with their properties. Also, create an apk file for the application to use on an Android phone:

main.py

```
from kivy.app import App
from kivy.uix.gridlayout import GridLayout
from kivy.config import Config
import math
Config.set('graphics', 'resizable', 1)

class ProjectKivyCalculator(GridLayout):
    # method called when = button (equals) is clicked/pressed
    def res_calculate(self, operation):
        if operation:
            try:
                self.display.text =
str(eval(operation))
            except Exception:
                self.display.text = "Error"
# Now creating an App class for calculator
class KivyCalculatorApp(App):
    def build(self):
        myapp = ProjectKivyCalculator()
        return myapp

# creating object and running it
```

```
demoApp = KivyCalculatorApp()
demoApp.run()

kivycalculator.kv:

# Create a custom button to add new buttons
<CustButton@Button>:
    font_size: 40
    background_color: (0,0.5,0.5,1)

<ProjectKivyCalculator>:
    id: calculator
    display: entry
    rows: 6
    padding: 10
    spacing: 10

    # Text Input where input will be displayed
    BoxLayout:
        TextInput:
            id: entry
            font_size: 40
            multiline: False

    # When buttons are pressed update the entry
    BoxLayout:
        spacing: 10
        CustButton:
```

```
text: "7"
on_press: entry.text += self.text
CustButton:
    text: "8"
    on_press: entry.text += self.text
CustButton:
    text: "9"
    on_press: entry.text += self.text
CustButton:
    text: "/"
    on_press: entry.text += self.text
CustButton:
    text: "sin"
    on_press: entry.text += "math.sin("
CustButton:
    text: "("
    on_press: entry.text += self.text
BoxLayout:
    spacing: 10
CustButton:
    text: "4"
    on_press: entry.text += self.text
CustButton:
    text: "5"
```

```
    on_press: entry.text += self.text
CustButton:
    text: "6"
    on_press: entry.text += self.text
CustButton:
    text: "*"
    on_press: entry.text += self.text
CustButton:
    text: "cos"
    on_press: entry.text += "math.cos("
CustButton:
    text: ")"
    on_press: entry.text += self.text

BoxLayout:
    spacing: 10
CustButton:
    text: "1"
    on_press: entry.text += self.text
CustButton:
    text: "2"
    on_press: entry.text += self.text
CustButton:
    text: "3"
    on_press: entry.text += self.text
```

```
CustButton:
    text: "-"
    on_press: entry.text += self.text

CustButton:
    text: "tan"
    on_press: entry.text += "math.tan("

CustButton:
    text: "sqrt"
    on_press: entry.text += "math.sqrt("

# When equals is pressed pass text in the entry
# to the calculate function

BoxLayout:
    spacing: 10

CustButton:
    text: "CLS"
    on_press: entry.text = ""

CustButton:
    text: "0"
    on_press: entry.text += self.text

CustButton:
    text: "="
    on_press: calculator.res_calculate(entry.text)

CustButton:
    text: "+"
```

```

    on_press: entry.text += self.text

CustButton:
    text: "log"
    on_press: entry.text += "math.log()"

CustButton:
    text: "."
    on_press: entry.text += self.text

BoxLayout:
    CustButton:
        font_size: 50
        bold: True
        text: "Calculate"
        on_press: calculator.res_calculate(entry.text)

```

Output:

The output can be seen in *Figure 10.12*:

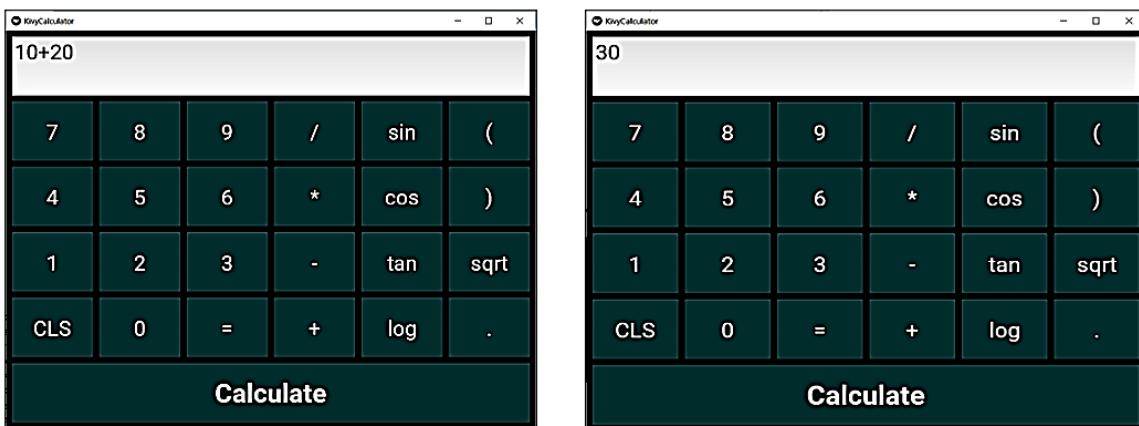


Figure 10.12: Output

The apk file was generated in the same manner as shown in the previous section. A view of the Google collab depicting the generated files and folders is as follows

in *Figure 10.13*:

Figure 10.13: Output

Practice project

Create a temperature converter app for converting Degree temperature to Celsius using Kivy. Also, create an apk file to run the application on Android phones.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 11

Image and Video Processing with Python

Introduction

In the preceding chapters, we have learned about various types of applications, such as GUI desktop applications, gaming applications, and mobile applications that can be developed using different Python libraries. In this chapter, we will discuss about the processing tasks related to image and video multimedia files. Images can be transformed by changing their color scheme, applying resizing and rotation operations, blurring and filtering images, blending multiple images to create an altogether new image, and so on. In the subsequent sections, we will also discuss frame-by-frame processing of videos that can be read either from a file or by capturing live feed from a Webcam. All these tasks are accomplished using the OpenCV library in Python.

Structure

In this chapter, we will discuss the following topics:

- Introduction to image processing
- Manipulating images
- Edge detection and object detection
- Video processing tasks

Objectives

The purpose of this chapter is to enable users to understand the concepts of image and video processing tasks for multimedia files. After going through this chapter, users can actively work on computer vision-related projects.

Introduction to image processing

Before beginning image processing, it is important for one to understand what an image is. A computer image is a picture composed of an array of elements called pixels, or in other words, the smallest unit of information in an image in digital imaging is called a pixel (or picture element). Squares are used to depict a 2-dimensional grid of pixels. Each pixel serves as a component of the original image, and more components usually result in more accurate reconstructions of the original. Each pixel's intensity can vary in color systems, and each pixel often consists of three or four colors, such as red, green, and blue, or cyan, magenta, yellow, and black. The word "pixel" is a logical combination of the two words, namely, "pix" (for "pictures") and "el" (for "element"). The term **Image Resolution** refers to the number or quantity of pixels in a digital image and is represented as a set of two numbers where the first number is the image width and the second one is the image height. For instance, if an image is 500×500 (width \times height), then 250,000 pixels make up the entire image:

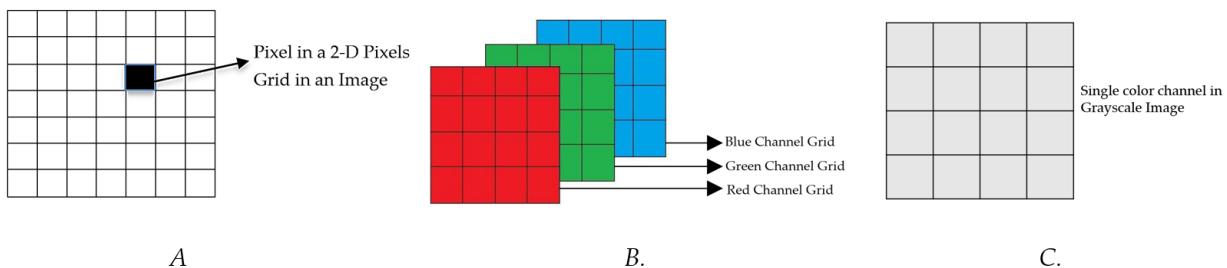


Figure 11.1: (A) Image view in the form of 2-D Pixel Grid, (B) color channels in an RGB Image, and (C) grayscale image with a single color channel

Most images come in the following two basic forms:

- **RGB images:** This two-dimensional RGB picture has three layers made up of red, green, and blue channels. It may be thought of as a mixture of three distinct images, such as a red scale image, a green

scale image, and a blue scale image, that are layered on top of each other and then blended to make an array of various colors. A pixel is determined by the combination of three numbers, each ranging from 0 to 255 (both inclusive), representing intensities stored in the red, green, and blue channels, respectively.

- **Grayscale images:** The images in the grayscale format only have one channel and various degrees of black and white. Since there is just one channel in a grayscale picture, each pixel value has a single value between 0 and 255 (inclusive). White is represented by the pixel value 255, and black by the pixel value 0.

Manipulating images

The process of converting an image into a digital format and carrying out specific procedures to extract some usable information from it is known as image processing. In most cases, when using specific specified signal processing techniques, the image processing system interprets all pictures as 2D signals. Many libraries are available in Python for image processing, including the following:

- **OpenCV:** OpenCV is a real-time computer vision-focused image processing library that finds utility in a variety of fields, including object identification, mobile robots, 2D and 3D feature toolkits, facial and gesture recognition, and human–computer interaction.
- **Numpy and Scipy:** For processing and manipulating images, use the Numpy and Scipy libraries.
- **Scikit-Image:** Numerous image processing algorithms are offered by Scikit-Image for the purpose of Image Segmentation, Transformations, Filtering, Feature detection, and so on in images.
- **Python Imaging Library (PIL):** To carry out fundamental operations on pictures, such as thumbnail creation, resizing, rotation, and file format conversion.

Image processing libraries in Python

To begin with image processing tasks, we need to ensure that our system is well configured with Python Environment set-up and Python 3.x installation. Our next step will be installing the necessary libraries, such as **opencv**, **pillow**, or others, that we want to employ for image processing. The **pip** command may be used to install the necessary libraries.

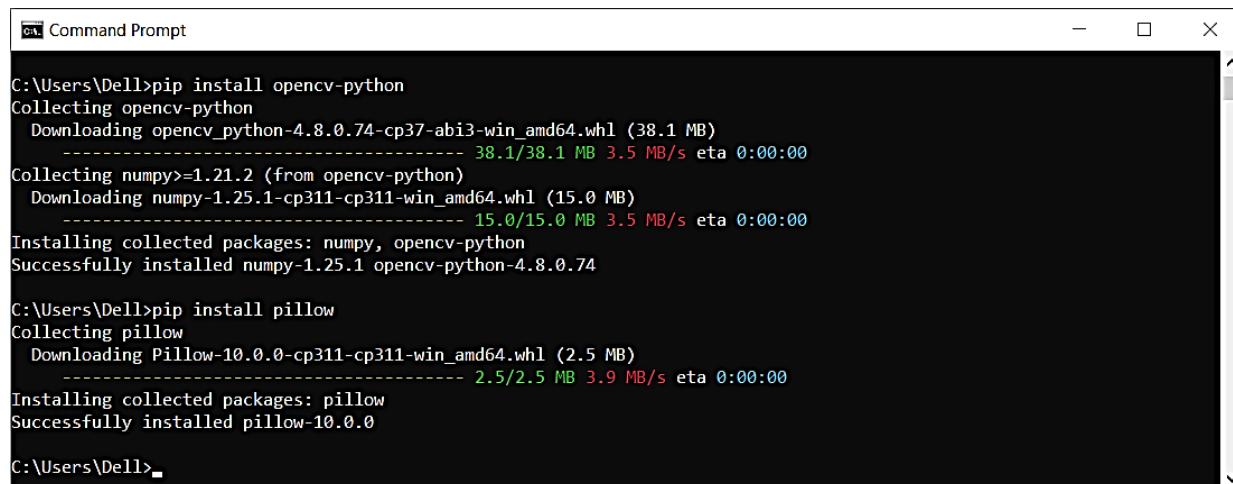
To install OpenCV the syntax used is as follows:

```
pip install opencv-python
```

To install pillow use:

```
pip install pillow
```

The following figure displays the installation of these libraries:



```
C:\Users\DELL>pip install opencv-python
Collecting opencv-python
  Downloading opencv_python-4.8.0.74-cp37-abi3-win_amd64.whl (38.1 MB)
    38.1/38.1 MB 3.5 MB/s eta 0:00:00
Collecting numpy>=1.21.2 (from opencv-python)
  Downloading numpy-1.25.1-cp311-cp311-win_amd64.whl (15.0 MB)
    15.0/15.0 MB 3.5 MB/s eta 0:00:00
Installing collected packages: numpy, opencv-python
Successfully installed numpy-1.25.1 opencv-python-4.8.0.74

C:\Users\DELL>pip install pillow
Collecting pillow
  Downloading Pillow-10.0.0-cp311-cp311-win_amd64.whl (2.5 MB)
    2.5/2.5 MB 3.9 MB/s eta 0:00:00
Installing collected packages: pillow
Successfully installed pillow-10.0.0

C:\Users\DELL>
```

Figure 11.2: Installation commands for OpenCV and pillow libraries

After installation, we can import the libraries using the **import** statement as follows:

```
import cv2                      # To import OpenCV-Python
library

import PIL                      # To import and use
compatible pillow module

from PIL import Image            # To import Image sub
module from PIL
```

To make it backward compatible with an earlier module named **Python Imaging Library (PIL)**, we shall invoke the PIL module instead of the Pillow module. Now, let us begin working with images using these libraries.

Reading an image

Before doing any processing-related tasks, we must first load and read the image in our application. Let us consider the following code to read an image file:

```
import cv2

new_img =
cv2.imread('D:\images\HeritageBuilding.png')

cv2.imshow('Heritage Building', new_img)

cv2.waitKey(0)
```

Output:

The output can be seen in *Figure 11.3*:

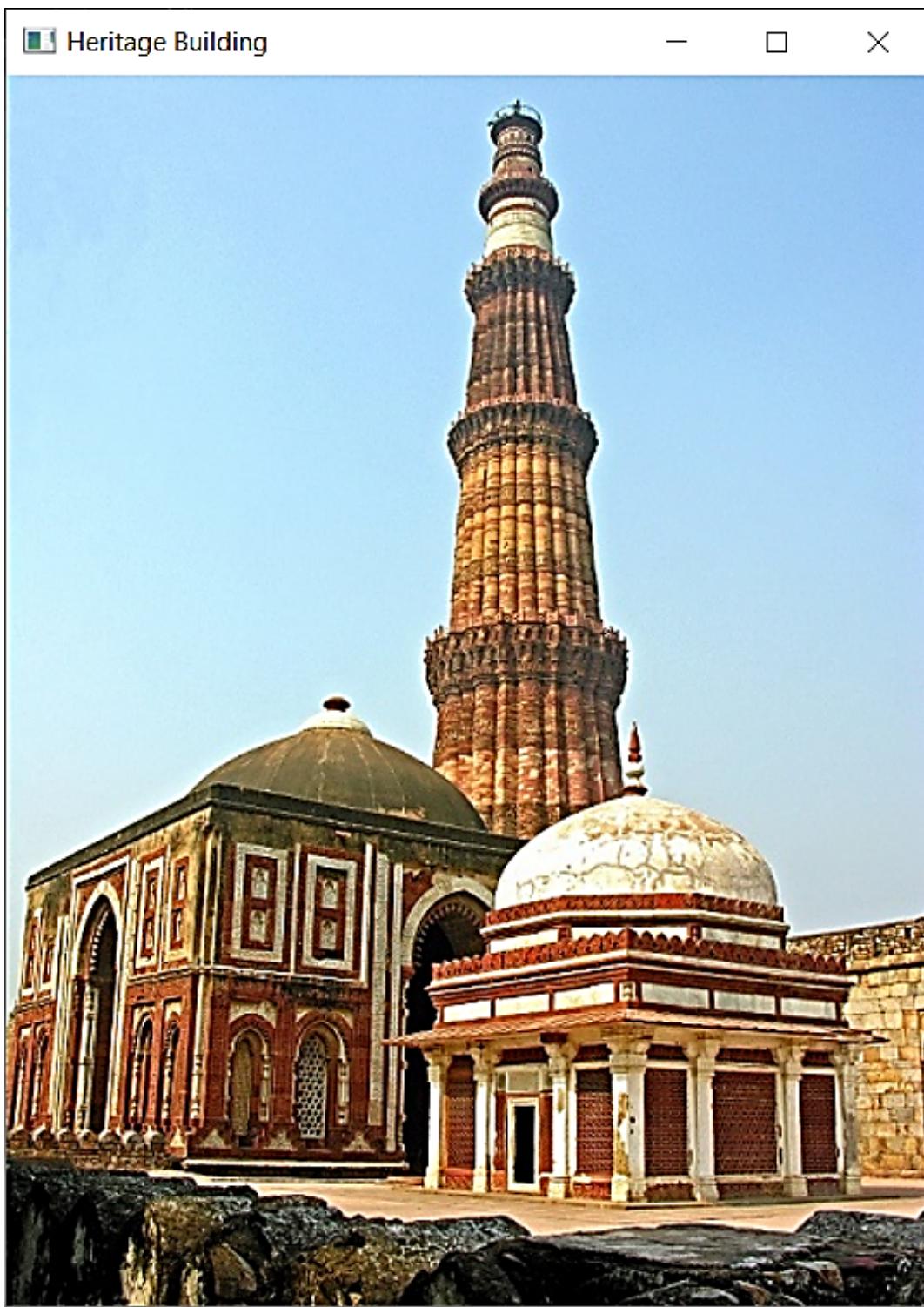


Figure 11.3: Output

We start by first importing the OpenCV module in Line 1. After this, the function **imread()** is used to read an image in a variable named **new_img**

into our application. To display the image on the screen, the **cv2** method **imshow()** is used. Inside this method, the first argument specified is the title of the image window, whereas the second argument is the variable containing the image file. In Line 4, the method **cv2.waitKey(0)** prevents the display window from closing abruptly and waits for user input via a key press to signal the closing of the image window.

Grayscale conversion and image blurring

Grayscale (or black and white) images are used for various image processing tasks because they are easier to interpret due to having only two colors. Therefore, a picture is first converted to grayscale with the help of the **cvtColor()** function. The image to be converted is the first parameter, followed by the color conversion code. Various color conversion codes are supported by OpenCV, such as **COLOR_BGR2RGB**, **COLOR_RGB2BGR**, **COLOR_BGR2GRAY**, **COLOR_RGB2GRAY**, **COLOR_GRAY2BGR**, and **COLOR_GRAY2RGB**. In our application to convert to Gray, we use **COLOR_BGR2GRAY** code because OpenCV follows the BGR color scheme by default, where BGR refers to Blue, Green, and Red. Finally, save the picture with a new file name by using the **cv2.imwrite()** function. Let us implement this in the following example:

```
import cv2

new_img =
cv2.imread('D:\images\HeritageBuilding.png')

# To convert BGR image to Grayscale

convert_gray_image = cv2.cvtColor(new_img,
cv2.COLOR_BGR2GRAY)

cv2.imshow('Converted Gray Image',
convert_gray_image)

cv2.waitKey(0)
```

Output:

The output can be seen in *Figure 11.4*:

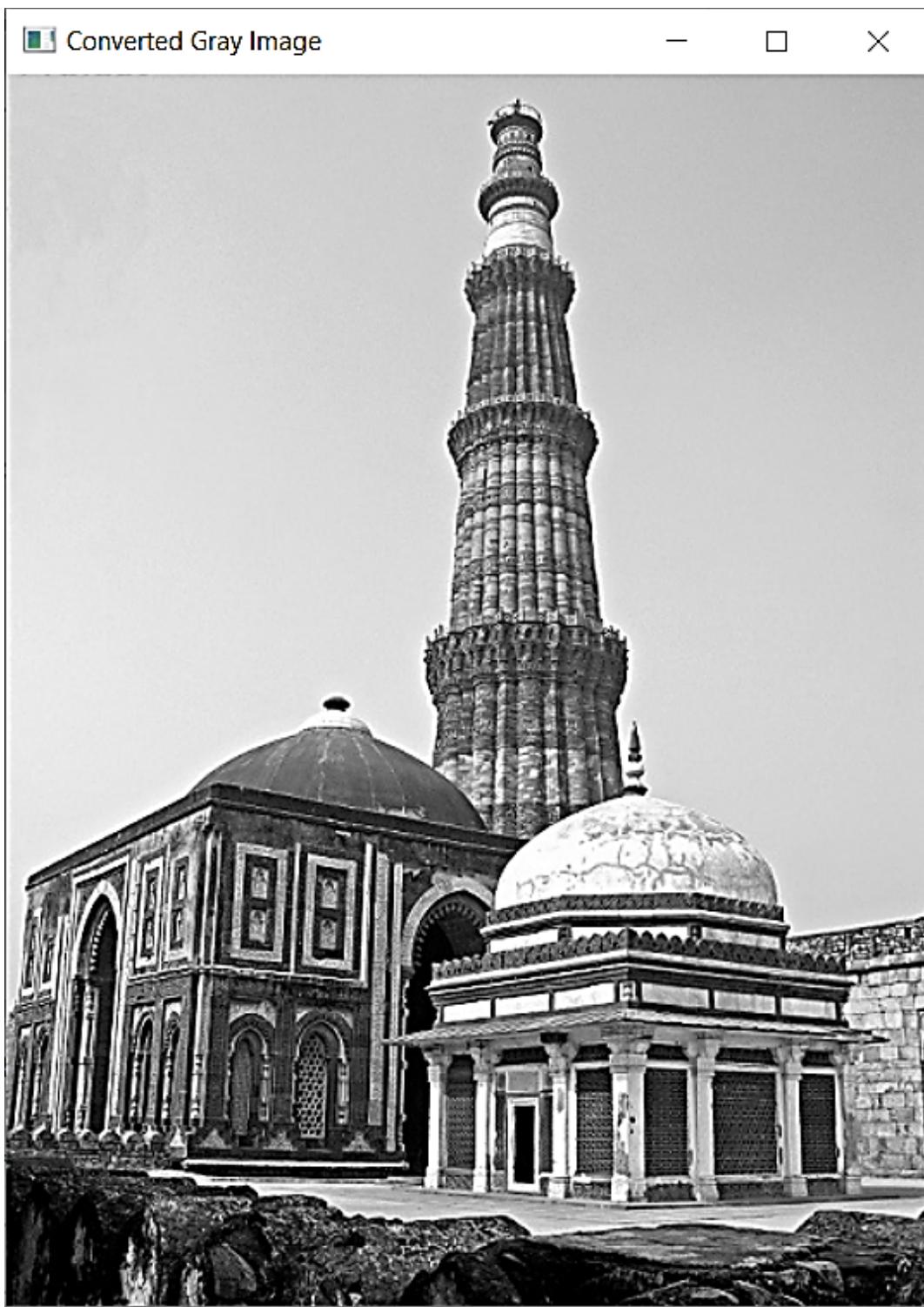


Figure 11.4: Output

If we want to reduce the level of detail in an image, we can blur the image. **Image Blurring** is the technique of reducing the clarity or distinctness in an

image with the help of several low-pass filters, also known as **Kernels**. Blurring an image helps in smoothing the image by removing noise, that is, high pass signals along with low-intensity edges from the image. Popular blurring techniques used are as follows:

- **Average Blur:** A simple type of blurring is Average blurring that takes an area of pixels in the neighborhood of a central pixel, then calculates the average intensity of these pixels, and finally substitutes the central pixel intensity with the calculated average. This technique reduces noise and the level of detail in the original image:

The syntax is as follows:

```
cv2.blur(source_image, kernel_size)
```

- The picture that will be blurred with the **blur()** method is called **source_image**.
- The kernel's size is represented by the matrix known as **kernel_size**. The smaller the value of kernel size, the less visible is the blur in the image. Odd sizes (3×3, 5×5, 7×7, and so on) are used.
- **Gaussian Blur:** The technique of Gaussian blurring is comparable to average blurring because here, instead of calculating the average or mean intensity of surrounding pixels, we compute a weighted mean such that the pixels closer to the central pixel are assigned a higher weightage in contrast to the far-off located pixels.

The syntax is as follows:

```
cv2.GaussianBlur(source_image, kernel_size, sigmaX)
```

- The picture that will be blurred with the Gaussian **Blur()** method is called **source_image**.
- The kernel's size is represented by the matrix known as **kernel_size**. The smaller the value of kernel size, the less visible is the blur in the image. Odd sizes (3×3, 5×5, 7×7, and so on) are used.

- The double-typed variable **sigmaX** represents the standard deviation of the Gaussian kernel in the X direction.
- **Median Blur:** Conventionally, the method of Median blurring is found to be more effective in reducing salt-and-pepper noise from images. Here, the intensity of the central pixel is replaced with the median of the surrounding pixels. This ensures replacement with a pixel intensity that always exists in the image itself.

The syntax is as follows:

cv2.medianBlur(source_image, kernel_size)

- The picture that will be blurred with the Median **Blur()** method is called **source_image**.
- The kernel’s size is represented by the matrix known as **kernel_size**.
- **Bilateral Blur:** A non-linear, noise-reducing, and edge-preserving filter for images is called a Bilateral Filter. Each pixel’s intensity is changed to a weighted average of intensity values from surrounding pixels. This weight may be determined using a Gaussian distribution. Thus, weak edges are discarded while sharp edges are kept.

The syntax is as follows:

cv2.bilateralFilter(source_image, kernel_size, d, sigmaColor, sigmaSpace)

- The picture that will be blurred with the blur() method is called **source_image**.
- The kernel’s size is represented by the matrix known as **kernel_size**.
- Here, **d** is an integer variable denoting the diameter of the pixel surrounding it.
- **sigmaColor** represents the sigma value in the color space. The colors that are further apart will start to mix the higher their value.

- **sigmaSpace** represents the coordinate space's sigma value. If the pixels' colors fall within the sigmaColor range, the bigger value will result in more pixels mixing together.

Let us consider an example depicting the use of all these blurring techniques:

```
import cv2

new_img =
cv2.imread('D:\images\HeritageBuilding.png')

cv2.imshow('Original Image', new_img)
cv2.waitKey(0)

Average.blur = cv2.blur(new_img, (10,10)) # Applying Average Blur
cv2.imshow('Average Blurring', Average.blur)
cv2.waitKey(0)

Gaussian = cv2.GaussianBlur(new_img, (11,11), 0) # Apply Gaussian Blur
cv2.imshow('Gaussian Blurring', Gaussian)
cv2.waitKey(0)

median = cv2.medianBlur(new_img, 9) # Applying Median Blur
cv2.imshow('Median Blurring', median)
cv2.waitKey(0)

bilateral = cv2.bilateralFilter(new_img, 11, 70, 70) # Bilateral Blur
cv2.imshow('Bilateral Blurring', bilateral)
cv2.waitKey(0)

cv2.destroyAllWindows()
```

Output:

The output can be seen in *Figure 11.5*:

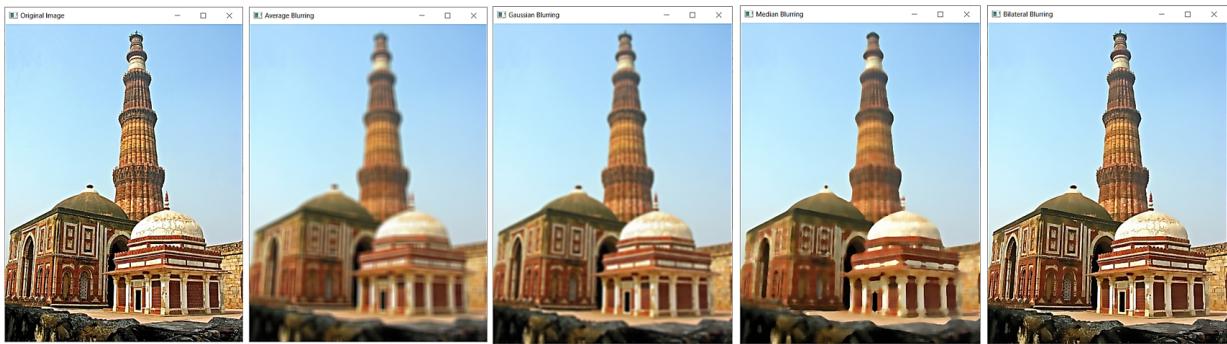


Figure 11.5: Output

Image edge detection

The ability to detect edges in images is a helpful feature. Finding an object's edges in a picture is a process known as edge detection. To locate the edges, the algorithm searches for factors such as color and brightness variations.

John Canny developed an edge detection algorithm in 1986, and his algorithm continues to be the most often used. With the use of a multi-stage algorithm, the Canny edge detector is an edge detection operator that can identify a variety of edges in pictures. The Canny edge detection algorithm consists of the following five steps:

- 1. Noise reduction:** The edge detection results are highly sensitive to image noise. One way to get rid of the noise in the image is by applying Gaussian blur to smooth it.
- 2. Gradient calculation:** The Gradient calculation step detects the edge intensity and direction by calculating the gradient of the image using edge detection operators. Sobel filters are applied to the image, and how get both intensity and edge direction matrices.
- 3. Non-maximum suppression:** Ideally, the final image should have thin edges. Thus, we must perform non-maximum suppression to thin out the edges.

4. Double threshold: The double threshold step aims at identifying three kinds of pixels: strong, weak, and non-relevant:

- a. The high threshold is used to identify the strong pixels (intensity higher than the high threshold)
- b. The low threshold is used to identify the non-relevant pixels (intensity lower than the low threshold)
- c. All pixels with intensity between both thresholds are flagged as weak, and the Hysteresis mechanism (next step) will help us identify the ones that could be considered as strong and the ones that are considered as non-relevant.

5. Edge tracking by hysteresis: Based on the threshold results, the hysteresis consists of transforming weak pixels into strong ones if and only if at least one of the pixels around the one being processed is a strong one.

We can perform an edge detection process on an image using the OpenCv **Canny()** method using the following syntax:

```
canny(source_image, threshold1, threshold2)
```

Here, **source_image** refers to the source or input image to detect its edges.

- **threshold1:** It represents the first threshold for the hysteresis process.
- **threshold2:** It represents the second threshold for the hysteresis procedure.

Let us consider the code for detecting edges in our sample image:

```
import cv2
import sys
# The first argument is the image
new_img =
cv2.imread('D:\images\HeritageBuilding.png')
```

```
# First convert to grayscale for easy handling

gray_image = cv2.cvtColor(new_img,
cv2.COLOR_BGR2GRAY)

# Next apply Gaussian blur operation to converted
image

gaussian_blurred_image =
cv2.GaussianBlur(gray_image, (5,5), 0)

cv2.imshow("Blurred Image", new_img)

cv2.waitKey(0)

# Apply canny() for edge detection

apply_canny1 = cv2.Canny(gaussian_blurred_image,
10, 50)

cv2.imshow("Applied Canny with lower thresholds",
apply_canny1)

cv2.waitKey(0)

apply_canny2 = cv2.Canny(gaussian_blurred_image,
60, 200)

cv2.imshow("Applied Canny with higher thresholds",
apply_canny2)

cv2.waitKey(0)
```

Output:

The output can be seen in *Figure 11.6*:

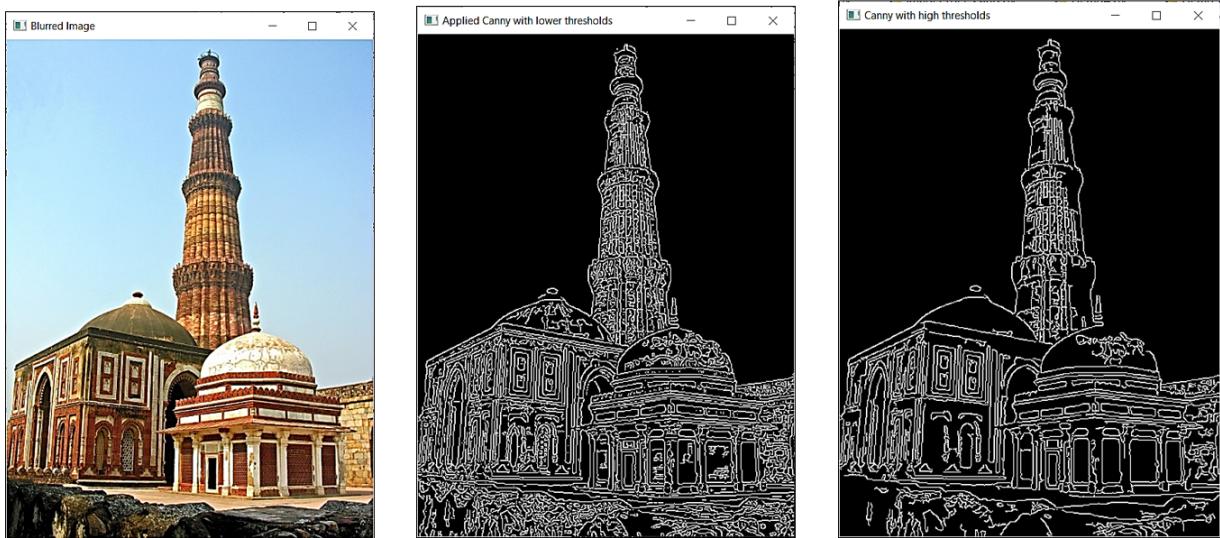


Figure 11.6: Output

Here, we can observe that when we use **canny()** with low values of thresholds, as in Line 12, the output displays a more detailed level of edges, whereas with higher values of thresholds, fewer edges are displayed. Users can choose the optimal thresholds by trying different values and selecting those that best suit their requirements.

Object detection in image

We can apply the knowledge gained through the edge detection strategy that we learned in the preceding section to perform advanced tasks such as detecting and counting different objects visible in an image. In this regard, the term **Contour** refers to a curve connecting each continuous point along a border having the same color or intensity. In order to analyze shapes and find objects, the contours are quite helpful. We can use the **findContours()** function to locate the contours in the image, and later, the **drawContours()** function can be used to draw contours on the original image.

The syntax for **findContours()** function is given as follows:

```
cv2.findContours(image, mode, method, contours)
```

- **image**: It refers 8-bit single-channel picture as the source of the image.

- **mode:** It is the contour retrieval mode used. Its values can be the following:
 - **cv.RETR_EXTERNAL:** To find out extreme outer contours only.
 - **cv.RETR_LIST:** To detect all contours without considering hierarchical relationships.
 - **cv.RETR_CCOMP:** To detect all contours and arrange them into a hierarchy.
 - **cv.RETR_TREE:** To find all contours and arrange nested contours into the hierarchy.
- **method:** mode is the contour approximation technique. Its values can be as follows:
 - **cv.CHAIN_APPROX_NONE:** To save all the contour points.
 - **cv.CHAIN_APPROX_SIMPLE:** To store only end points of horizontal, vertical, and diagonal elements.

The syntax for **drawContours()** function is as follows:

```
cv2.drawContours(image, contours, contourIdx, color)
```

- **image:** It refers to the destination image with contours.
- **Contours:** It stores the input contours in the form of point vectors.
- **contourIdx:** This parameter represents a contour for drawing. When its value is set to a negative number, then all the contours are drawn.
- **Color:** It refers to the color of the boundaries or contours.

Let us consider an example to detect and count the number of coin objects present in an image:

```
import cv2
# Read the image
new_img = cv2.imread('D:\images\coins.png')
```

```
# First convert image to grayscale format
grayscale_image = cv2.cvtColor(new_img,
cv2.COLOR_BGR2GRAY)

# Use any blurring method to modify image
blur_img = cv2.GaussianBlur(grayscale_image,
(11,11), 0)

# Apply Canny edge detector
apply_canny = cv2.Canny(blur_img, 150, 250)

# Use findContours() to find out the contour edges
# and draw them
contours, hierarchy= cv2.findContours(apply_canny,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

cv2.drawContours(new_img, contours, -1, (0,255,0),
2)

cv2.imshow("Detecting Objects", new_img)
print("Number of objects found = ", len(contours))
cv2.waitKey(0)
```

Output:

The output can be seen in [*Figure 11.7*](#):



Figure 11.7: Output

Image resize and rotation

Scaling an image is referred to as image resizing. Many applications of image processing and machine learning benefit from scaling. To fit the required size, we frequently need to resize the image, either scaling it up or down and zooming in or out on images. The syntax for resize is as follows:

```
cv2.resize(source, dsize, dest, fx, fy,  
interpolation)
```

- **source** refers to the Image input for resizing
- **dsize** represents the output array size after resizing
- **dest** is the optional parameter representing the new output image after scaling
- **fx** is optional and represents the scaling factor in the horizontal direction
- is optional and represents the scaling factor in the vertical direction
- **interpolation**: For scaling a picture, OpenCV offers us numerous interpolation techniques. It is an optional parameter.
 - **cv2.INTER_AREA**: Used when an image has to be shrunk.

- **cv2.INTER_CUBIC**: This is slower but more effective than.
- **cv2.INTER_LINEAR**: Default interpolation and used for zooming in the image.

For rotation, we have another method called **cv2.rotate()**, which is used to rotate an image in multiples of 90 degrees either clockwise or anti-clockwise. The function **cv2.rotate** rotates the array in three different ways. The syntax for the rotate method is as follows:

```
cv2.rotate(source, rotation Code, dest)
```

- **source** refers to the image input for rotation
- **rotateCode** specifies the type of rotation for the image. The three values used are as follows:
 - **cv2.ROTATE_90_CLOCKWISE**: To rotate the image by 90 degrees clockwise
 - **cv2.ROTATE_90_COUNTERCLOCKWISE**: To rotate the image by 90 degrees counter-clockwise or, more specifically, 270 degrees clockwise.
 - **cv2.ROTATE_180**: To rotate the image by 180 degrees clockwise.
- **dest** is the optional parameter representing a new output image after rotation

Let us consider an example of image scaling and rotation:

```
import cv2

new_img =
cv2.imread('D:\images\HeritageBuilding.png')

cv2.imshow('Original Image', new_img)

img_height, img_width = new_img.shape[:2]

resize_img = cv2.resize(new_img, (int(img_width/2),
int(img_height/2)), interpolation = cv2.INTER_AREA)

cv2.imshow('Scaled Image', resize_img)
```

```

img_center = (img_width / 2, img_height / 2)

rotate_matrix = cv2.getRotationMatrix2D(img_center,
90, -1)

rotated_img = cv2.warpAffine(new_img,
rotate_matrix, (img_height, img_width))

cv2.imshow('Rotated Image', rotated_img)

cv2.waitKey(0)

```

Output:

The output can be seen in *Figure 11.8*:

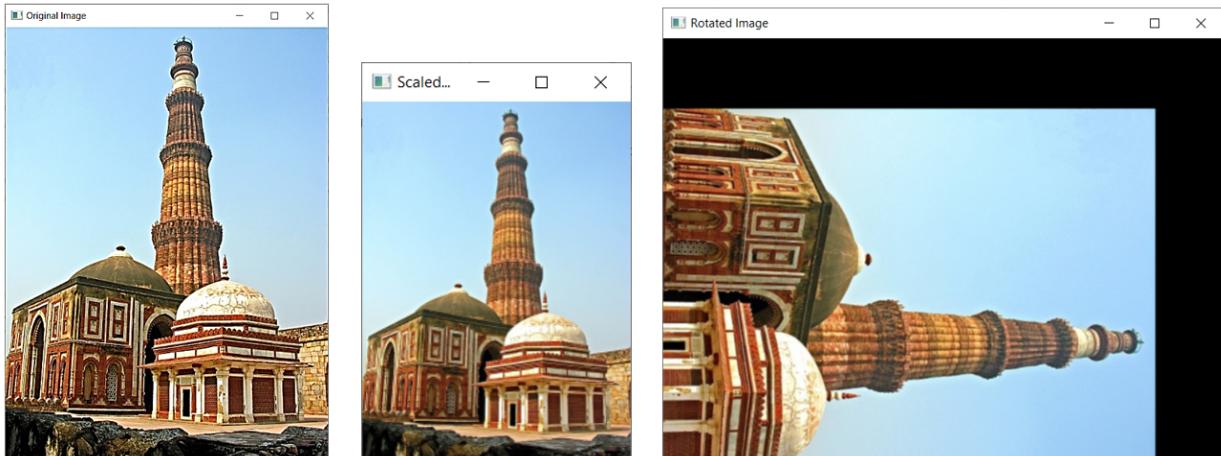


Figure 11.8: Output

Image addition, subtraction, and blending

To add two images together, we can use the `cv2.add(image1, image2)` function, which takes as input the two images to be added into a single image. Please note that the shape of the two input images must be equal. If required, we can resize the images using `numpy.resize()` function to match their shapes. Image Blending is a type of image addition with a variation in which different transition factors or weights are allocated to different input images for mixing or blending the pixels of these images. The following equation is used for blending images:

$$G(x) = (1-\alpha) F_0(x) + \alpha F_1(x)$$

Here, **F0(x)** and **F1(x)** are image functions, and α is the transition factor whose value ranges between 0 and 1. For image blending, OpenCV-Python provides the **addWeighted()** function. The syntax used is as follows:

```
cv2.addWeighted(src1, alpha, src2, beta, gamma)
```

- Here, **src1** and **src2** represent the two input images for blending.
- **alpha** represents the weight factor given to the first image for blending
- **beta** is the weight factor given to the second image for blending, and its value is $(1 - \alpha)$
- **gamma** refers to a scalar value added for the correctness of the image.

Let us consider an example of adding and blending images together:

```
import cv2
import numpy as np
new_image1 = cv2.imread(r"D:\images\boat.png")
new_image2 = cv2.imread(r"D:\images\nature.png")
# Resize one image to match sizes of both images
new_image2 = np.resize(new_image2, new_image1.shape)
print(new_image1.shape, new_image2.shape)
added_image = cv2.add(new_image1, new_image2)
blended_image = cv2.addWeighted(new_image1, 0.4,
new_image2, 0.6, 0)
cv2.imshow('First Image', new_image1)
cv2.imshow('Second Image', new_image2)
cv2.imshow('Added Image', added_image)
```

```
cv2.imshow('Weighted Image', blended_image)
```

```
cv2.waitKey(0)
```

Output:

The output can be seen in *Figure 11.9*:

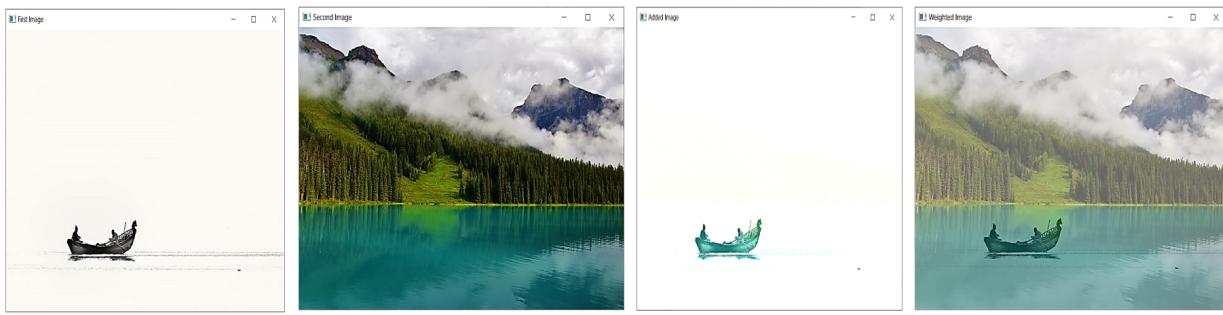


Figure 11.9: Output

Video processing tasks in Python

Processing a video refers to applying operations frame by frame to a video. Simply put, a frame is a specific instance of the video at any point in time. Thus, even in a single second of a video, there might exist many frames. We may handle frames just like images and can perform most operations on video frames that were executed on images in previous sections of the chapter. In OpenCV, a video may be read either by reading a video file or by using the feed from a camera that is attached to a computer.

Read and play a video file

Making a **VideoCapture** object is the initial step in reading a video file. Either the device index or the name of the video file to be read may be used as its parameter. The syntax for reading an input video file is as follows:

```
read_video = cv2.VideoCapture(filename)
```

Once a video file is read, it can be display frame by frame by using the **imshow()** function. Just like images, we need to use the **waitKey()** function as well to pause and capture each frame in the video. Please note that for images, parameter **0** is passed to the **waitKey()** function, whereas a

number greater than “0” must be used as a parameter for playing a video file. The reason is that “0” would force a video frame to pause infinitely, but each frame in a video must be displayed for a specific period only. Therefore, **waitKey(1)** is used in this case to specify a delay of 1 ms in **waitKey()**. After this, the next step is to capture the current frame and write it into an image file. For this, the **imwrite()** function can be used:

```
cv2.imwrite("image.png", frame)
```

If a video file lacks any sound, then it can be easily rendered and played. But in case we require audio to be accompanied by the video, then we must install the **FFPyPlayer** module in order to accomplish this. For playing and writing media files, the **FFPyPlayer** module provides a Python binding for the **FFmpeg** library. The following is the pip command that can be used to install **FFPyPlayer**:

```
pip install ffpypyplayer
```

Once installed successfully, the **get_frame()** method of **MediaPlayer** class object in **ffpyplayer** module can be used to return the audio frame which will be played simultaneously to the video frame read from the video file. Let us consider an example to read a video file and play it along its audio in sync.:

```
import cv2

from ffpypyplayer.player import MediaPlayer

video_file = r"D:\videos\samplevideo.mp4"

video_read = cv2.VideoCapture(video_file)

player = MediaPlayer(video_file)

while True:

    flag, frame_read = video_read.read()

    audio_frame, value = player.get_frame()

    if not flag:
```

```

        print("End of video")
        break

    if cv2.waitKey(1) == ord("q"):
        break

    cv2.imshow("Video", frame_read)

    if value != 'eof' and audio_frame is not None:
        image, time = audio_frame

    video_read.release()

cv2.destroyAllWindows()

```

Here in Line 12 the condition `cv2.waitKey(1) == ord("q")` denotes two things; first, the underlying statement runs only once per frame. Second, to end the while loop, the user must press key `q`. Another example to convert the playing video into frame-by-frame images is given as follows. This is useful when a certain frame needs to be modified and transformed in the video:

```

import cv2

video_file =
cv2.VideoCapture(r"D:\videos\samplevideo.mp4")

frameID = 0

while(True):
    flag, frame_read = video_file.read()

    if flag == True:
        # Continue creating image frames till video
        ends

        framename = str(frameID) + '.png'

```

```

        print ('Next frame read...' + name)
        cv2.imwrite(framename, frame_read)
        frameID += 1
    else: break
video_file.release()
cv2.destroyAllWindows()

```

Output:

The output can be seen in *Figure 11.10*:

```

C:\Users\Dell\PycharmProjects\PygameDemo1\venv\Scripts\python.exe C:\Users\Dell\PycharmProjects\PygameDemo1\Demo10.py
Next frame read...0.png
Next frame read...1.png
Next frame read...2.png
Next frame read...3.png
Next frame read...4.png
Next frame read...5.png
Next frame read...6.png
Next frame read...7.png
Next frame read...8.png
Next frame read...9.png
Next frame read...10.png
Next frame read...11.png
Next frame read...12.png
Next frame read...13.png
Next frame read...14.png
Next frame read...15.png
Next frame read...16.png

```

Figure 11.10: Output

Capture live video from Webcam

The syntax for taking live video feed from an integrated Web camera device is as follows:

```
camera_record = cv2.VideoCapture(0)
```

Most of the time, the system just has one camera attached. Therefore, OpenCV uses the built-in camera that is connected to the computer when we pass **0** for the device index. In case of multiple cameras linked to the system,

we may choose the second camera by using device index **1**, device index **2** for the third one, and so on. Once the camera opens successfully, the **read()** function can be used to read successive frames.

```
flag, frame_read = camera_record.read()
```

The **read()** function enables reading the next frame available in the streaming video and returns a True or False **flag** value depending on the availability of the frame. We may be required to save the live feed from the Web camera into a video file for later use. For this, we may use the **VideoWriter()** function with the following syntax:

```
cv2.VideoWriter(filename, fourcc, fps, frameSize)
```

Here, the **filename** parameter refers to the destination file along with its folder location where the video file will be stored. The second parameter **fourcc** is a four-digit code for video codecs. OpenCV supports various codecs such as **H264**, **DIVX**, **MP4S**, **XVID**, **MJPG**, **MPG1**, and so on. The next parameter, **fps**, is the Frame rate of the video stream created, and **frameSize** denotes the video frame size. In the end, we must release the frame and VideoWriter objects as the final steps of video creation. Let us consider the code to capture live video from a built-in Web camera:

```
import cv2

newfile = cv2.VideoWriter('D:\video\MyVideo.avi',
cv2.VideoWriter_fourcc('M','J','P','G'), 15, (720,
560))

# To define a video capture object for built-in
webcam

video_read = cv2.VideoCapture(0)

while (True):

    flag, frame_read = video_read.read() # To
capture frame by frame video
```

```

cv2.imshow('frame', frame_read) # To view the
resultant frame

newfile.write(frame_read)

if cv2.waitKey(1) & 0xFF == ord('q'):

    break

# After the loop release the video capture and
video write objects

video_read.release()

newfile.release()

cv2.destroyAllWindows()

```

The users are advised to run the code to view results on their systems.

Conclusion

This chapter revolves around the image and video processing tasks. As images and videos are major multimedia components on any platform nowadays. Thus, it becomes a crucial step to carefully modify, store, and render these multimedia files. This chapter gives a deeper knowledge on the processing tasks related to resizing, blurring, filtering images, and blending multiple images to create new images altogether. Also, we discussed video processing issues where video can be read and modified frame-by-frame either by reading a prestored video file or by capturing live video feed from integrated Web cameras. The knowledge gained through these topics will help users to work with advanced multimedia projects in python using OpenCv library.

Points to remember

- A computer image is a picture composed of an array of elements called pixels, or in other words, the smallest unit of information in an image in digital imaging is called a pixel (or picture element).

- The term Image Resolution refers to the number or quantity of pixels in a digital image and is represented as a set of two numbers where the first number is the image width and the second one is the image height.
- OpenCV is a real-time computer vision-focused image processing library that finds utility in a variety of fields, including object identification, mobile robots, 2D and 3D feature toolkits, facial and gesture recognition, and human-computer interaction.
- Grayscale (or black and white) images are used for various image processing tasks because they are easier to interpret due to having only two colors. Therefore, a picture is first converted to grayscale with the help of the **cvtColor()** function.
- The ability to detect edges in images is a helpful feature. Finding an object's edges in a picture is a process known as edge detection.
- Scaling an image is referred to as image resizing. Many applications of image processing and machine learning benefit from scaling.
- Processing a video refers to applying operations frame by frame to a video. Simply put, a frame is a specific instance of the video at any point in time.

Exercise

Attempt the following project.

Sample project with solution

Create an application for Face Detection in an image. Along with that, once a face is detected, then we try to locate and highlight eyes in the face. Furthermore, the application detects a smile on the face and draws a boundary to highlight the smile as well. Also, the program counts and displays the total number of faces detected in an image. The OpenCV library uses haarcascades XML files as filters to make the detections related to a face and various parts of it like eyes and smile. The GitHub link for these files is **opencv/data/haarcascades at master · opencv/opencv · GitHub**

```
import cv2

# Import xml files for face detection and eye
detection

face_cascade_file =
cv2.CascadeClassifier('haarcascade_frontalface_defa
ult.xml')

eye_cascade_file =
cv2.CascadeClassifier('haarcascade_eye.xml')

smile_cascade_file =
cv2.CascadeClassifier('haarcascade_smile.xml')

image_file = cv2.imread(r'D:\images\movie.png')

# convert to gray scale of each frames

grayscale_image = cv2.cvtColor(image_file,
cv2.COLOR_BGR2GRAY)

# Detects faces of different sizes in the input
image

# The detectMultiScale() detects different size
faces in input image and returns rectangles
positioned on the faces.

# The first parameter is the input image, the
second one is scale factor which defines the
reduction size of image

# and the third argument is the minNeighbors i.e.
neighbors each rectangle must have.The
scaleFactor=1.3 and minNeighbors=5 are chosen on
experimental basis.

faces_detect =
face_cascade_file.detectMultiScale(grayscale_image,
scaleFactor = 1.3, minNeighbors = 5)
```

```
# Define loop for each face detected utilizing
generated coordinates in above function

for (x,y,w,h) in faces_detect:

    # Next draw a rectangle in the input image. Here
    (255,0,0) is the color of the frame in RGB. # The
    last parameter value 2 denotes the thickness of the
    rectangle. Here x refers to the initial horizontal
    position,

        # w refers to the width, the vertical initial
        position denoted by y and h denotes height.

        image_rect =cv2.rectangle(image_file,(x,y),
        (x+w,y+h),(255,0,0),2)

    # Next define roi_gray_img as the region of
    major interest i.e. face area to look for the eyes.

    roi_gray_img = grayscale_image[y:y+h, x:x+w]

    # Now set the same region of interest in the
    original imageframe.

    roi_color_img = image_rect[y:y+h, x:x+w]

    # Now apply same function to detect eyes and
    draw rectangles

    eyes_detect =
eye_cascade_file.detectMultiScale(roi_gray_img)

    for (ex,ey,ew,eh) in eyes_detect:

        cv2.rectangle(roi_color_img,(ex,ey),
(ex+ew,ey+eh),(0,255,0),2)

    # Further apply detectMultiScale() function to
    detect smile and draw rectangles
```

```
smiles_detect =  
smile_cascade_file.detectMultiScale(roi_gray_img,  
1.1, 22)  
  
for (sx, sy, sw, sh) in smiles_detect:  
    cv2.rectangle(roi_color_img, (sx, sy), (sx +  
sw, sy + sh), (0, 0, 255), 2)  
  
    print("There are {}faces in the  
image".format(len(faces_detect)))  
  
cv2.imshow('Face and Eye Detected  
Image',image_file)  
  
cv2.waitKey(0)  
  
cv2.destroyAllWindows()
```

Output:

The output can be seen in [*Figure 11.11*](#):

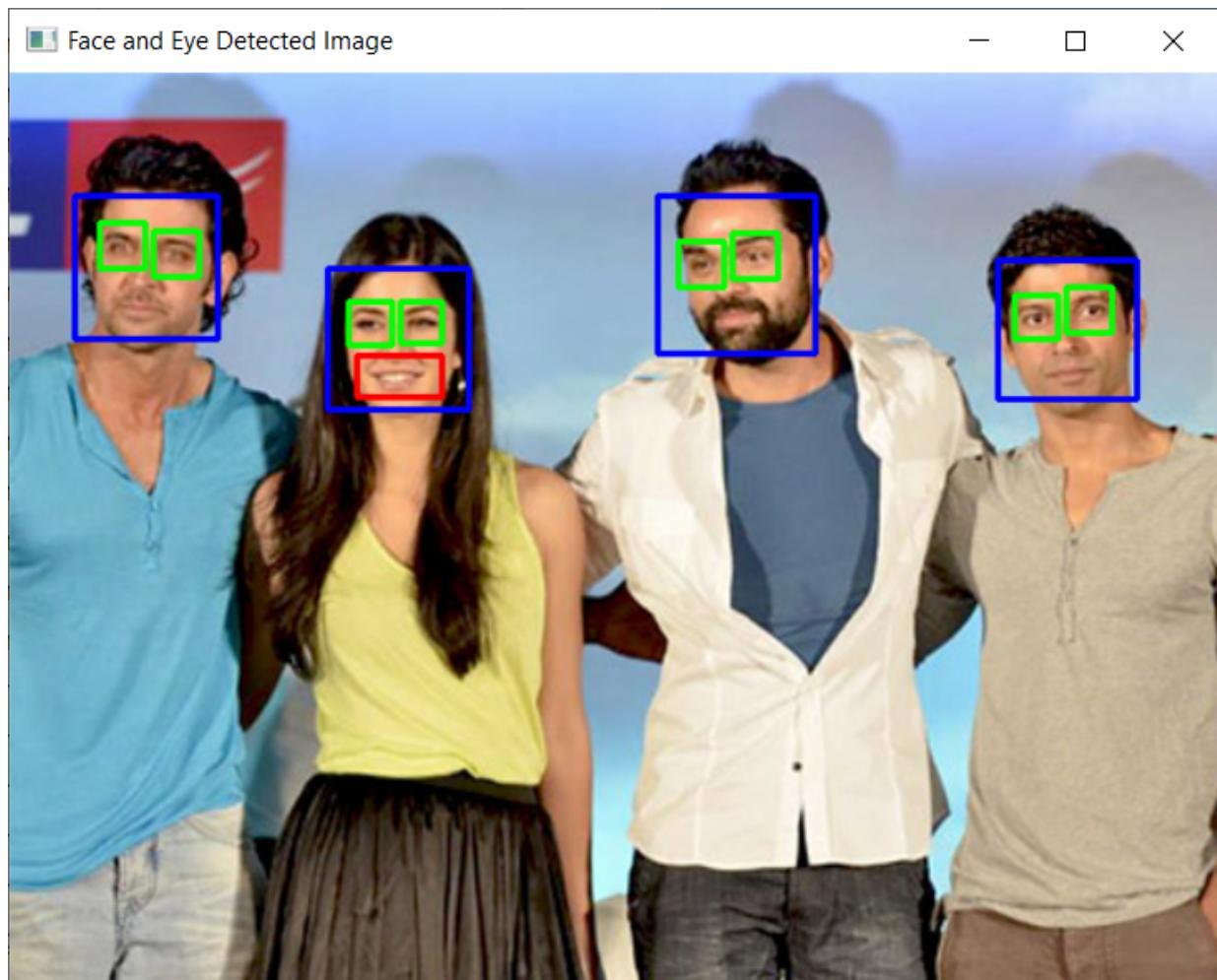


Figure 11.11: Output
(Source: File:ZNMD cast.jpg - Wikimedia Commons)

Note: In the output above we can observe big boxes which are the boundaries for face in each human figure. Inside each face boundary we observe two small boxes encircling the eyes detected and below eyes we can see another box present if smile is detected inside the face boundary as in the case of second human figure in the image above.

Practice project

Create an application to add a watermark label on an input image.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Appendix

Multiple choice questions

Choose the correct answer for the following questions:

1. What can be the maximum possible length of an identifier in Python?

- a. 16
- b. 32
- c. 64
- d. No fixed identifier length in Python

2. Choose the invalid variable name among the following:

- a. 1_mystring
- b. my_string_1
- c. _my_string
- d. Mystring

3. Which one of the following is the correct extension of the Python file?

- a. .py
- b. .python

- c. .p
 - d. None of these
- 4. Identify the correct order of operator precedence in Python?**
- a. Parentheses, Exponential, Multiplication, Division, Addition, Subtraction
 - b. Multiplication, Division, Addition, Subtraction, Parentheses, Exponential
 - c. Division, Multiplication, Addition, Subtraction, Parentheses, Exponential
 - d. Exponential, Parentheses, Multiplication, Division, Addition, Subtraction
- 5. Which character is used in Python to make a single line comment?**
- a. /
 - b. //
 - c. #
 - d. !
- 6. The output of this Python code would be:**
- ```
str='{}', {}, and {}'
str.format('hello', 'welcome', 'bye!')
```
- a. 'welcome, hello and bye!'
  - b. 'hello welcome and bye!'
  - c. 'hello, welcome, bye!'
  - d. 'bye!, welcome and hello'

**7. Which of the following operators is the correct option for power(ab)?**

a.  $a \wedge b$

b.  $a^{**}b$

c.  $a \wedge \wedge b$

d.  $a \wedge * b$

**8. What will be the output of the following code snippet?**

```
print(2**3 + (5 + 6)**(1 + 1))
```

a. 121

b. 8

c. 129

d. 0

**9. What will be the datatype of the var in the below code snippet?**

```
var = 100
```

```
print(type(var))
```

```
var = "Hello, Welcome!"
```

```
print(type(var))
```

a. str and int

b. int and int

c. str and str

d. int and str

**10. What will be the output of the following code snippet?**

```
x = 3
```

```
y = 1
print(x, y)
x, y = y, x
print(x, y)
```

- a. 31 13
- b. 31 31
- c. 13 13
- d. 13 31

**11. Which of the following is not used as loop in Python?**

- a. for loop
- b. while loop
- c. do-while loop
- d. None of the above

**12. Choose the incorrect statement regarding loops in Python?**

- a. Loops are used to perform certain tasks repeatedly.
- b. While loop is used when multiple statements are to be executed repeatedly until the given condition becomes False
- c. While loop is used when multiple statements are to be executed repeatedly until the given condition becomes True.
- d. for loop can be used to iterate through the elements of lists.

**13. Which one of the following is a valid Python if statement.**

- a. if a>=2 :
- b. if (a >= 2),

- c. if (a => 22);
  - d. if a >= 22..
- 14. Which keyword is used to add a counter condition to an if statement?**
- a. else if
  - b. elseif
  - c. elif
  - d. None of the above
- 15. Give the output of the given below program?**
- ```
if 3 + 2 == 7:  
    print("Hello User")  
else:  
    print("Else Bye!")
```
- a. Hello User
 - b. Else Bye!
 - c. Hello User Else Bye!
 - d. Error
- 16. Give the output of given Python code?**
- ```
string1="hello"
count=0
for x in string1:
 if(x!="l"):
```

```
 count=count+1
else:
 pass
print(c)
```

a. 2

b. 0

c. 4

d. 3

**17. Predict number of times the loop may run?**

```
i=2
while(i>0):
 i=i-1
```

a. 2

b. 3

c. 1

d. 0

**18. Which of the following is a valid for loop in Python?**

a. for(i=0; i < n; i++)

b. for i in range(0,10):

c. for i in range(0,10)

d. for i in range(10)

**19. What will be the output of the following Python program?**

```
i = 0
while i < 5:
 print(i)
 i += 1
 if i == 3:
 break
else:
 print(0)
```

- a. error
- b. 0 1 2 0
- c. 0 1 2
- d. none of the mentioned

**20. The continue statement can be used in?**

- a. while loop
- b. for loop
- c. do-while
- d. Both A and B

**21. What will be the output of the following Python code?**

```
mystr1="22/4"
print("mystr1")
```

- a. 1
- b. 22/4

c. 5.5

d. mystr1

**22. Which of the following will result in an error?**

```
mystr1="Hello World!"
```

a. print(mystr1[2])

b. mystr1[1] = "m"

c. print(mystr1[0:9])

d. Both (b) and (c)

**23. Suppose tuple1 = (1, 2, 4, 3, 5), which of the following is incorrect?**

a. tuple1[3] = 35

b. print(tuple1[4])

c. print(max(tuple1))

d. print(len(tuple1))

**24. Choose the correct output of the following Python code snippet?**

```
dict1={11:"A",22:"B",33:"C"}
```

```
for i, j in dict1.items():
```

```
print(i, j, end=" ")
```

a. 11 22 33

b. A B C

c. 11 A 22 B 33 C

d. 11:"A" 22:"B" 33:"C"

**25. Choose the output of following Python code?**

```
myset1={22,55,33}
myset2={3,1}
myset3={}
myset3=myset1&myset2
print(myset3)
```

- a. {33}
- b. {}
- c. {22,55,33,11}
- d. {22,55,11}

**26. The recursive function is defined as:**

- a. A function that calls many other functions.
- b. A function which calls itself once or many times.
- c. Both A and B
- d. None of the above

**27. What is the output of the following program?**

```
result = lambda n : n * n
print(result(5))
```

- a. 5
- b. 25
- c. 0
- d. Error

**28. Which of the following arguments are passed to a function in definite positional order.**

- a. Required arguments
- b. Keyword arguments
- c. Default arguments
- d. Variable-length arguments

**29. What is the output of the following code?**

```
def outer_func(x, y):
 def inner_func(p, q):
 return p + q
 return inner_func(x, y)
result = outer_func(5, 10)
print(result)
```

- a. (5, 10)
- b. Syntax Error
- c. 15
- d. None of these

**30. What is the output of the following display( ) function call?**

```
def display(**kwargs):
 for i in kwargs:
 print(i)
display(emp="John", salary=5000)
```

- a. TypeError
  - b. John  
5000
  - c. ('emp', John')  
(‘salary’, 5000)
  - d. emp  
salary
31. **Choose the number of objects and reference variables created in below code snippet?**
- ```
class MyClass:  
    print("Inside MyClass")  
MyClass()  
MyClass()  
obj=MyClass()
```
- a. 2 and 1
 - b. 3 and 3
 - c. 3 and 1
 - d. 3 and 2
32. **Which of the following is False with respect Python code?**

```
class Emp:  
    def __init__(self, e_id, e_age):  
        self.e_id=e_id  
        self.e_age=e_age
```

```
obj1=Emp(101, 25)
```

- a. “obj1” is the reference variable for object Emp(101, 25)
- b. e_id and e_age are called the parameters.
- c. It is mandatory for every class to have a constructor.
- d. None of the above

33. What will be the output of below Python code?

```
class MyClass():  
    def __init__(self, count=100):  
        self.count=count  
obj1=MyClass()  
obj2= MyClass(102)  
print(obj1.count)  
print(obj2.count)
```

- a. 100, 100
- b. 100, 102
- c. 102, 102
- d. Error

34. Which of the following is correct?

```
class MyClass:  
    def __init__(self, name):  
        self.name=name  
obj1=MyClass("Simon")
```

```
obj2=MyClass("Simon")
```

- a. Both id(obj1) and id(obj2) will have same value.
- b. Each id(obj1) and id(obj2) will have different values.
- c. No two objects with same value of attribute can be created.
- d. None of the above

35. What are setattr() and getattr() used for?

- a. To set an attribute value
- b. To delete an attribute value
- c. To find whether an attribute exists or not
- d. To set the attribute value and to access the attribute value respectively.

36. What will be the output shape of the following Python code?

```
import turtle  
turt=turtle.Pen()  
for i in range(0,4):  
    turt.forward(110)  
    turt.left(130)
```

a. square

b. rectangle

c. triangle

d. kite

37. Select the turtle command to reset the turtle pen

- a. `turtle.reset`
- b. `turtle.penreset`
- c. `turtle.penreset()`
- d. `turtle.reset()`

38. Select the correct output of the following Python code?

```
import turtle  
turt=turtle.Pen()  
turt.goto(100,9)  
turt.position()
```

- a. 100.00, 9.00
- b. 9, 100
- c. 100, 9
- d. 9.00, 100.00

39. Choose the direction in which the turtle is by default pointed?

- a. North
- b. South
- c. East
- d. West

40. What will be the output of the following Python code?

```
import turtle  
turt=turtle.Pen()  
turt.clear()
```

`turt.isVisible()`

a. Yes

b. True

c. No

d. False

41. SQLite is which type of database?

a. NoSQL database

b. Distributed database

c. Relational database

d. Operational database

42. Which of the following features supported by SQLite?

a. self-contained

b. serverless

c. zero-configuration

d. All of the above

43. Which join is supported in SQLite?

a. LEFT OUTER JOIN

b. RIGHT OUTER JOIN

c. FULL OUTER JOIN

d. All of the above

44. Which command is used regarding SQLite ALTER TABLE Statement for Rename operation?

- a. ALTER TABLE table_name ADD COLUMN column_def...;
 - b. ALTER TABLE table_name RENAME TO new_table_name;
 - c. ALTER TABLE table_name COLUMN column_def...;
 - d. ALTER TABLE table_name TO new_table_name;
45. **User can delete a whole table or a table view, from system memory using the _____ command.**
- a. DELETE
 - b. DROP
 - c. REMOVE
 - d. None
46. **Which of the following is the major usage of config() in Python Tkinter ?**
- a. To place the widget
 - b. To destroy the widget
 - c. To configure the widget
 - d. To change property of the widget
47. **Which of the following describes the correct way to draw a line in canvas Tkinter ?**
- a. Line
 - b. create_line(canvas)
 - c. canvas.create_line()
 - d. None of the above
48. **The _____ widget helps to retrieve data from the user.**

- a. Entry
 - b. Label
 - c. Button
 - d. None of the above
- 49. Which among the following methods can be used to put a widget at the screen?**
- a. grid()
 - b. pack()
 - c. place()
 - d. All of the above
- 50. In which way the place() function will place the widget on the screen ?**
- a. According to only rows
 - b. According to x, y coordinate
 - c. According to left, right, up, down
 - d. According to row and column
- 51. What is the use of the pack() function for the Tkinter widget ?**
- a. To destroy the widget
 - b. To perform a task by a widget
 - c. To pack the widget on the screen
 - d. To define a size of the widget
- 52. What is the use of the mainloop() in Python Tkinter ?**

- a. To create a window screen
- b. To Hold the window Screen
- c. To Destroy the window screen
- d. None of the above

53. What best defines a Pixel?

- a. Spatial coordinates
- b. Two-dimensional function
- c. Image elements
- d. Plane coordinates

54. How many colour values are present in binary image?

- a. 2
- b. 5
- c. 6
- d. 1

55. Find the combination of the following image formats to their number of channels

- 1) GrayScale
 - 2) RGB
 - I. 1 channel
 - II. 2 channels
 - III. 3 channels
 - IV. 4 channels
- a. RGB -> I and GrayScale-> III

b. RGB -> IV and GrayScale-> II

c. RGB -> II and GrayScale -> I

d. RGB -> III and GrayScale -> I

Solutions

1.	d
2.	a
3.	a
4.	a
5.	c
6.	b
7.	b
8.	c
9.	d
10.	a

11.	c
12.	c
13.	a
14.	c
15.	b
16.	d
17.	a
18.	b
19.	c
20.	d

21.	d
22.	b
23.	a
24.	c
25.	a
26.	b
27.	b
28.	a
29.	c
30.	d

31.	c
32.	c
33.	b
34.	b
35.	d
36.	c
37.	d
38.	a
39.	c
40.	b

41.	c
42.	d
43.	a
44.	b
45.	b
46.	d
47.	c
48.	a
49.	d
50.	b



Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Index

A

Affinity 176
aggregate functions 190, 191
Android Application Packages (APKs) 310
anonymous functions 97
demonstrations 97, 98
anonymous (or Lambda) functions 91
arithmetic operators 25
assignment operator 26
Average Blur 326

B

begin_fill() method 150
Bilateral Blur 327, 328
binary search algorithm 80, 81
bitwise operators 28
BLOB and DATE TIME
in SQLite 194, 195
Boolean 13, 14
bubble sort 75
buldozer 310
built-in class attributes 114
built-in data structures 59, 60
Built-in Namespace 32
Button widget 210, 211

C

Canny edge detection algorithm 328
Centrum Wiskunde & Informatica (CWI) 1
Checkbutton widget 218, 219
Child class 119
class 107, 108
class method 111, 112
class variable 108
Combobox widget 219, 220
Command Binding 210
commands, SQLite
aggregate functions 190

clauses 188-190
DDL commands 181, 182
DML commands 187, 188
DQL commands 188
comments 8
comparison operators 27
Compile-time polymorphism 129
constructor overloading 130
method overloading 129
operator overloading 130, 131
constructor overloading 130
Constructor Overriding 133
constructors 115
default constructor 117
non-parameterized constructor 116, 117
parameterized constructor 115, 116
Contour 330
CRUD operations 178

D

data 170
database 171
non-relational databases 172
relational, versus non-relational database 171
Database Management System (DBMS) 169
database management, with SQLite
database, creating 178-180
Data Definition Language (DDL) commands 181, 182
SQLite table constraints 183
data hiding 118, 119
Data Manipulation Language (DML) commands 187, 188
Data Query Language (DQL) command 188
data structures 59, 60
dictionary 70, 71
list 65, 66
set 67
string 60-64
tuple 67
data types 10
Boolean 13, 14
dictionaries 12, 13
numbers 10-12
sequence types 15
set 14, 15
decision-making 38

default arguments 93
default constructor 117
Derived class 119
dictionary 12, 13, 70
 method 70, 71
Domain Integrity constraints 183, 184
drawContours() function 331
Dynamic Method Dispatch 132

E

edge detection 328, 329
encapsulation 118, 119
end_fill() method 150
Entity Integrity constraints 184
Entry widget 212-215
escape sequences 17
Event Binding 245, 246, 247
Event Handlers 209
event programming, with Turtle
 key events 151-161
 mouse events 151-157
Event Queue 271
events, Pygame
 key events 271
 mouse events 271-273
exception handling 177, 178
Explicit Type Conversion 20, 21
expressions 25

F

fillcolor() method 150
findContours() function 331
First-In-First-Out (FIFO) 74
flow control statements 49
 working 49-51
for loop 45, 46
function arguments 92, 93
 default arguments 93
 keyword arguments 93, 94
 required arguments 94
 variable-length arguments 95
functions 90
 anonymous (or Lambda) functions 91
 benefits 90
 calling 91, 92

declaration 91, 92
pre-defined functions 91
types 91
user-defined functions 91
versus methods 90, 91

G

Gaussian Blur 326
geometry management, Tkinter
 grid layout 241, 242, 243
 pack layout 240, 241
 place layout 243, 244
 widgets, organizing with layout managers 239
geometry management, with layout manager in Kivy
 AnchorLayout 298
 BoxLayout 298
 FloatLayout 299
 GridLayout 299
 StackLayout 300
Geometry Manager 239
Global Namespace 32
global variable 98
GNU General Public License (GPL) 1
Graphical User Interface (GUI) application 205
Grayscale images 323
GUI programming 206
 features 206

H

hierarchical inheritance 122
hybrid inheritance 124, 125

I

identifier 7
 naming rules 7, 8
identity operators 29
IDLE 5
if...elif...else statement 39-42
if...else statement 38-41
if statement 38-40
Image Blending 334
Image Blurring 326
 Average Blur 326
 Bilateral Blur 327, 328

Gaussian Blur 326
Median Blur 327
image edge detection 328-330
image processing 322
grayscale conversion 325
Grayscale images 323
image blurring 325, 326
image, reading 324, 325
images, manipulating 323
libraries, in Python 323, 324
RGB images 322
imperative programming 106
Implicit Type Conversion 20
infinite while loop 47, 48
inheritance 119
 advantages 120
 hierarchical inheritance 122, 123
 hybrid inheritance 124, 125
 multilevel inheritance 121
 multiple inheritance 123
 single inheritance 120, 121
 types 120
input/output process 22-24
insertion sort algorithm 77-79
instance method 110
instance variable 109
Integrated Development Environment (IDE) 5, 58

J

joins, SQLite 191, 192

K

Kernels 326
key events 157
 example 158-161
keyword arguments 93, 95
keywords 7
Kivy 286
 app life cycle 288-290
 behavior widgets 291
 complex UX widgets 291
 features 286
 geometry management 297-300
 installing 287
 Layout managers 291

layouts 290
Screen Manager 291
UX widgets 291-297
widgets 290, 291
Kivy applications
 packaging, with buildozer 310-314
Kivy application with multiple screens
 creating 307-310
KV language
 basics 300
 dynamic class 304, 305
 event and properties 304
 file loading 301
 modules 301
 syntax guidelines, for file 301
 widget reference 305-307
 widgets 301-303

L

LabelFrame widget 234, 235
Label widget 211, 212
lambda function 97
Last-In-First-Out (LIFO) 72
linear search algorithm 79, 80
line of code (LOC) 90
linked list 71, 72
lists 17, 18, 65
 functions 66
 methods 65
Listbox widget 220, 221
Local Namespace 32
local variables 98
logical operators 27
loop control 44, 45
 for loop 45
 infinite while loop 47, 48
 nested loop 48, 49
 while loop 45-47

M

Magic Methods 130
Median Blur 327
membership operators 30
Menu widget 223-226
Messagebox widget 235, 236

method overloading 129, 132
Method Resolution Order (MRO) 125-127
methods 108
modules 99, 100
mouse events 152
example 152-157
multilevel inheritance 121
multiple inheritance 123

N

Named arguments 93
Namespace 32
 Built-in Namespace 32
 Global Namespace 32
 Local Namespace 32
nested function 98
nested if statement 42, 43
nested loop 48, 49
Non-Keyword Arguments 95
nonlocal variables 98
non-parameterized constructor 116, 117
non-relational databases 172
NoSQL (Not Only SQL) 172
number data types 10-12
Numpy 323

O

object detection, in image 330-332
 image addition 334, 335
 image blending 334, 335
 image resizing 332
 image rotation 333
 image subtraction 334, 335
object-oriented programming (OOP) 105, 107
 class 107, 108
 objects 107, 108
OpenCV 323
operator overloading 24, 130
operators 25
 arithmetic operators 25
 assignment operators 26
 bitwise operators 28
 comparison operators 27
 identity operators 29
 logical operators 27

membership operators 30
precedence 31
outer function 98
output formatting
 with format() method 24, 25
Overflow condition 72, 74

P

packages 100
parameterized constructor 115
Parameterized Queries
 using, in SQLite 192-194
polymorphism 129
 Compile-time polymorphism 129
 Runtime polymorphism 132, 133
precedence 31
procedural programming 106
programming paradigms 106
 object-oriented Programming 107
 procedural programming 106
PyCharm IDE 58
 installation 58, 59
PyGame 262
 collisions 276, 278
 color object 265
 demo 265
 events 271
 images 269, 270
 installing 262
 shapes 266-268
 sprites 276, 278
 surfaces 265, 266
 text and music, adding 273-275
 working with 262-265
Python 1
 features 2, 3
 installing 3-7
 Type Conversion 19
 URL 3
Python Imaging Library (PIL) 323

Q

queue 74

R

RadioButton widget 217, 218
recursion 96
Referential integrity constraints 184
relational database 171
Relational Database Management System (RDBMS) 171
relational operators 27
required arguments 94
RGB images 322
Runtime polymorphism 132, 133

S

Scikit-Image 323
Scipy 323
searching algorithms 79
 binary search 80, 81
 linear search 79, 80
selection sort algorithm 76, 77
sequence 15
 sequence types 15
 strings 15, 16
set 14-67
 methods 68
set operations 68
 set difference 69
 set intersection 69
 set symmetric difference 69, 70
 set union 69
shapes, creating with Turtle 141, 142
 circle, drawing 147, 148
 colors, filling 150, 151
 connecting lines, drawing 142
 hexagon, drawing 145-147
 octagon, drawing 145-147
 oval, drawing 147, 148
 rectangle, drawing 143
 spiral, drawing 149
 square, drawing 143
 star pentagon, drawing 145-147
 triangle, drawing 143
Simple DirectMedia Library (SDL) 262
single inheritance 120, 121
sinking sort 75
sorting 75
 sorting algorithms 75

bubble sort 75
insertion sort 77-79
selection sort 76, 77
space complexity 75
time complexity 75
space complexity 75
Spinbox widget 227-229
SQLite 172
 BLOB and DATE TIME 194, 195
 characteristics 172
 commands 181
 database, connecting in Python 175, 176
 database management 178
 datatypes 176, 177
 downloading 172
 for database handling 172
 GUI tools 174
 installing, in command-line 173
 joins 191
 Parameterized Queries 192-194
 working, in Python 174
SQLite table constraints
 Domain integrity constraints 183
 Entity integrity constraints 184
 Referential integrity constraints 184-187
stack 72, 73
static method 113
storage classes 176
str.format() method 24
strings 15, 16, 60
 escape sequences 17
 functions 64
 methods 62, 63
 operators 60-62
Structured Query Language (SQL) 171
super() function 127
 in multiple inheritance 128, 129
 in single inheritance 127

T

Text widget 215-217
time complexity 75
Tkinter 206
 geometry management 239
 installing 206-209

Tkinter filedialog 237
 directory, selecting 239
 file path, returning 237, 238
 file, saving 238
Tkinter widgets 209, 210
 Button 210, 211
 Checkbutton 218, 219
 Combobox 219
 Entry 212-215
 Label 211, 212
 LabelFrame 234, 235
 Listbox 220, 221
 Menu 223-226
 Messagebox 235, 236
 Radiobutton 217, 218
 Spinbox 227-229
 Text 215-217
 Treeview 229, 230
Treeview widget 229, 230
 example 231-234
 hierarchical widget 230
 tree table widget 230
tuple 18, 19, 67
Turtle programming 138
 event programming 151
 plotting with 138-140
 shapes, creating 141, 142
Type Conversion 19
 Explicit Type Conversion 20, 21
 Implicit Type Conversion 20

U

Underflow condition 72
user-defined data structures 71
 linked list 71, 72
 queue 74
 stack 72, 73
UX widgets, Kivy 291
 button widget 292
 checkbox widget 293
 clock widget 297
 image widget 294
 label widget 292
 progress bar widget 296
 radio button widget 294

slider widget 296
switch widget 296
textinput widget 293
toggle button widget 297
video widget 295

V

variable 8, 9
variable-length arguments 95
variables
 global variable 98
 local variables 98
 nonlocal variable 98
 scope 98, 99
video processing 335-337
 live video, capturing from Webcam 338, 339

W

while loop 45-47